

2ND EDITION

Python Feature Engineering Cookbook

Over 70 recipes for creating, engineering, and transforming features to build machine learning models



SOLEDAD GALLI



BIRMINGHAM—MUMBAI

Python Feature Engineering Cookbook

Copyright © 2022 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Publishing Product Manager: Ali Abidi

Senior Editor: Nathanya Dias

Technical Editor: Sweety Pagaria

Copy Editor: Safis Editing

Project Coordinator: Farheen Fathima

Proofreader: Safis Editing

Indexer: Pratik Shirodkar

Production Designer: Alishon Mendonca

Marketing Coordinator: Shifa Ansari

First published: January 2020

Second edition: October 2022

Production reference: 1211022

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80461-130-2

www.packt.com

Contributors

About the author

Soledad Galli is a data scientist, instructor, and software developer with more than 10 years of experience in world-class academic institutions and renowned businesses. She has developed and put into production machine learning models to assess insurance claims and credit risk and prevent fraud. She teaches multiple online courses on machine learning, which have enrolled 44,000+ students worldwide and consistently receive good student reviews. She is also the developer and maintainer of the open source Python library Feature-engine, which is currently downloaded 100,000+ times per month. Soledad received a Data Science Leaders Award in 2018 and was recognized as one of LinkedIn's voices in data science and analytics in 2019.

About the reviewers

Edoardo Argiolas was born in Sardinia, Italy, in 1996. He studied a humanities major in Cagliari and subsequently enrolled at the physics university in the town, where he graduated in 2018. He then moved to Turin, northern Italy, where he continued his education as a physics of complex systems student and started developing his affinity with programming as well as AI. As part of his MSc curriculum, he spent one semester at the University of Groningen, Netherlands, and later worked as a developer intern at ASML in Eindhoven. Having attained his master's degree, he landed his first full-time job in March 2022 as a software engineer at Italian financial giant Intesa Sanpaolo, where he contributed to the Python backend of the upcoming cloud-native ISY bank.

Morgan Sell works as a data scientist for a Fortune 5 company where he designs experiments, applying behavioral economics and machine learning.

Additionally, he has performed user segmentation and developed predictive models for a consumer fintech start-up. Moreover, Morgan has created economic/statistical models to operate solar and wind projects and raise financing for the construction of the projects. Morgan is a core contributor to the Feature-engine package. When Morgan is not applying data science to solve problems, he enjoys surfing, snowboarding, and adventure travel.

Hector Patiño has been involved with machine learning for geosciences since 2015, especially for subjects related to satellite imagery. He has strong knowledge of Python and SQL and is an experienced user of PostgreSQL, ArcGIS, QGIS, and more GIS-related software. He has experience as a backend developer with Django and Django REST framework for the COVID-19 vaccination process in the state of Cundinamarca in Colombia. He also enjoys solving regex puzzles. He lives in Cúcuta, Colombia, with his dog, Kanvas, and his husband. He loves playing tennis.

Table of Contents

Preface

1

Imputing Missing Data

Technical requirements

Removing observations with missing data

How to do it...

How it works...

Performing mean or median imputation

How to do it...

How it works...

Imputing categorical variables

How to do it...

How it works...

Replacing missing values with an arbitrary number

How to do it...

How it works...

Finding extreme values for imputation

How to do it...

How it works...

Marking imputed values

How to do it...

How it works...

Performing multivariate imputation by chained equations

How to do it...

How it works...

See also

Estimating missing data with nearest neighbors

How to do it...

How it works...

2

Encoding Categorical Variables

Technical requirements

Creating binary variables through one-hot encoding

How to do it...

How it works...

There's more...

Performing one-hot encoding of frequent categories

How to do it...

How it works...

There's more...

Replacing categories with counts or the frequency of observations

[How to do it...](#)

[How it works...](#)

[Replacing categories with ordinal numbers](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Performing ordinal encoding based on the target value](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Implementing target mean encoding](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Encoding with the Weight of Evidence](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Grouping rare or infrequent categories](#)

[How to do it...](#)

[How it works...](#)

[Performing binary encoding](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[3](#)

[Transforming Numerical Variables](#)

[Transforming variables with the logarithm function](#)

[Getting ready](#)

[How to do it...](#)

How it works...

There's more...

Transforming variables with the reciprocal function

How to do it...

How it works...

Using the square root to transform variables

How to do it...

How it works...

Using power transformations

How to do it...

How it works...

Performing Box-Cox transformation

How to do it...

How it works...

There's more...

Performing Yeo-Johnson transformation

How to do it...

How it works...

There's more...

4

Performing Variable Discretization

Technical requirements

Performing equal-width discretization

How to do it...

How it works...

See also

Implementing equal-frequency discretization

How to do it...

How it works...

Discretizing the variable into arbitrary intervals

[How to do it...](#)

[How it works...](#)

[Performing discretization with k-means clustering](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Implementing feature binarization](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Using decision trees for discretization](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

Working with Outliers

Technical requirements

Visualizing outliers with boxplots

How to do it...

How it works...

Finding outliers using the mean and standard deviation

How to do it...

How it works...

Finding outliers with the interquartile range proximity rule

How to do it...

How it works...

Removing outliers

How to do it...

How it works...

Capping or censoring outliers

How to do it...

How it works...

There's more...

Capping outliers using quantiles

How to do it...

How it works...

6

Extracting Features from Date and Time Variables

Technical requirements

Extracting features from dates with pandas

Getting ready

How to do it...

How it works...

[There's more...](#)

[See also](#)

[Extracting features from time with pandas](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Capturing the elapsed time between datetime variables](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Working with time in different time zones](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

Automating feature extraction with Feature-engine

How to do it...

How it works...

7

Performing Feature Scaling

Technical requirements

Standardizing the features

How to do it...

How it works...

Scaling to the maximum and minimum values

How to do it...

How it works...

Scaling with the median and quantiles

How to do it...

How it works...

Performing mean normalization

How to do it...

How it works...

There's more...

Implementing maximum absolute scaling

Getting ready

How to do it...

How it works...

There's more...

Scaling to vector unit length

How to do it...

How it works...

8

Creating New Features

Technical requirements

[Combining features with mathematical functions](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Comparing features to reference variables](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Performing polynomial expansion](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Combining features with decision trees](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Creating periodic features from cyclical variables](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[Creating spline features](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

Extracting Features from Relational Data with Featuretools

Technical requirements

Setting up an entity set and creating features automatically

Getting ready

How to do it...

How it works...

See also

Creating features with general and cumulative operations

Getting ready

How to do it...

How it works...

Combining numerical features

How to do it...

How it works...

Extracting features from date and time

How to do it...

How it works...

There's more...

Extracting features from text

Getting ready

How to do it...

How it works...

Creating features with aggregation primitives

Getting ready

How to do it...

How it works...

10

Creating Features from a Time Series with
tsfresh

Technical requirements

Extracting features automatically from a time series

Getting ready

How to do it...

How it works...

See also

Creating and selecting features for a time series

How to do it...

How it works...

See also

Tailoring feature creation to different time series

How to do it...

How it works...

Creating pre-selected features

How to do it...

[How it works...](#)

[Embedding feature creation in a scikit-learn pipeline](#)

[How to do it...](#)

[How it works...](#)

[See also](#)

[11](#)

[Extracting Features from Text Variables](#)

[Technical requirements](#)

[Counting characters, words, and vocabulary](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

Estimating text complexity by counting sentences

Getting ready

How to do it...

How it works...

There's more...

Creating features with bag-of-words and n-grams

Getting ready

How to do it...

How it works...

See also

Implementing term frequency-inverse document frequency

Getting ready

How to do it...

How it works...

See also

[Cleaning and stemming text variables](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Index](#)

[Other Books You May Enjoy](#)

Preface

Python Feature Engineering Cookbook, Second Edition, covers almost every aspect of feature engineering for tabular data, including missing data imputation, categorical encoding, variable transformation, discretization, scaling, and the handling of outliers. It also discusses how to extract features from date and time, text, time series, and relational datasets.

This book will take the pain out of feature engineering by showing you how to use open source Python libraries to accelerate the feature engineering process via multiple practical, hands-on recipes. Throughout the book, you will transform and create new features utilizing pandas, scikit-learn, and also the four major open source feature engineering libraries: Feature-engine, Category Encoders, Featuretools, and tsfresh.

Who this book is for

This book is for machine learning and data science students and professionals, as well as software engineers working on machine learning model deployment, who want to learn more about how to transform their data and create new features to train better machine learning models.

What this book covers

Chapter 1, Imputing Missing Data, discusses various techniques to fill in missing values with estimates of missing data that are suitable for numerical and categorical features.

Chapter 2, Encoding Categorical Variables, introduces various widely used techniques to transform categorical variables into numbers. It starts by describing commonly used methods such as one-hot and ordinal encoding, then it moves on to domain-specific methods such as the weight of the evidence, and finally, it shows you how to encode variables that are highly cardinal.

Chapter 3, Transforming Numerical Variables, explain when we need to transform variables for use in machine learning models and then discusses common transformations and their suitability, based on variable characteristics.

Chapter 4, Performing Variable Discretization, introduces discretization and when it is useful, and then moves on to describe various discretization methods and their advantages and limitations. It covers the basic equal-width and equal-frequency discretization procedures, as well as discretization using decision trees and k-means.

Chapter 5, Working with Outliers, shows commonly used methods to remove outliers from the variables. You will learn how to detect outliers, how to cap variables at a given arbitrary value, and how to remove outliers.

Chapter 6, Extracting Features from Date and Time, describes how to create features from dates and time variables. It covers how to extract date and time components from features, as well as how to combine datetime variables and how to work with different time zones.

Chapter 7, Performing Feature Scaling, covers methods to put the variables on a similar scale. It discusses standardization, how to scale to maximum and minimum values, and how to perform more robust forms of variable scaling.

Chapter 8, Creating New Features, describes multiple methods with which we can combine existing variables to create new features. It shows the use of mathematical operations and also decision trees to create variables from two or more existing features.

Chapter 9, Extracting Features from Relational Data with Featuretools, introduces relational datasets and then moves on to explain how we can create features at different data aggregation levels, utilizing Featuretools. You will learn how to automatically create dozens of features from numerical and categorical variables, datetime, and text.

Chapter 10, Creating Features from Time Series with tsfresh, discusses how to automatically create several hundreds of features from time series data, for use in supervised classification or regression. You will learn how to automatically create and select relevant features from your time series with tsfresh.

Chapter 11, Extracting Features from Text Variables, covers simple methods to clean and extract value from short pieces of text. You will learn how to count words, sentences, characters, and lexical diversity. You will

discover how to clean your text pieces and how to create feature matrices by counting words.

To get the most out of this book

Python Feature Engineering Cookbook will give you the practice, tools, and techniques to streamline your feature engineering pipelines and simplify and improve the quality of your code. The book discusses feature engineering methods to transform and create features to train machine learning models using Python. Therefore, some knowledge of machine learning and Python programming will be an asset.

The recipes have been tested in the following library versions: **category-encoders == 2.4.0, Feature-engine == 1.4.0, featuretools == 1.4.0, 1.5.0, matplotlib==3.4.2, numpy==1.22.0, pandas==1.5.0, scikit-learn==1.1.0, scipy==1.7.0, seaborn==0.11.1, statsmodels==0.12.2, and tsfresh==0.19.0.**

Software/hardware covered in the book	OS requirements
Python 3.3 or greater	Windows, macOS, or Linux
Jupyter Notebook	Windows, macOS, or Linux

Note that earlier versions or newer versions than those displayed in the table may prevent code from running. If you are using newer versions, make sure to check their documentation online for changes in parameter names. That usually solves the issue.

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/Python-Feature-Engineering-Cookbook-Second-Edition>. If there's an update to the code, it will be updated in the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

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

Conventions used

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

Code in text: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: “Next, we applied the

fit_transform() method to replace missing data in the train set and the **transform()** method to replace missing data in the test set.”

A block of code is set as follows:

```
X_train = pd.DataFrame(  
    X_train,  
    columns=numeric_vars + remaining_vars,  
)
```

Any command-line input or output is written as follows:

```
pip install feature-engine
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: “Click on **crx.data** to download the data.”

TIPS OR IMPORTANT NOTES

Appear like this.

Sections

In this book, you will find several headings that appear frequently (*Getting ready*, *How to do it...*, *How it works...*, *There's more...*, and *See also*).

To give clear instructions on how to complete a recipe, use these sections as follows:

Getting ready

This section tells you what to expect in the recipe and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make you more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Get in touch

Feedback from our readers is always welcome.

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

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit

www.packtpub.com/support/errata, select your book, click on the **Errata Submission Form** link, and enter the details.

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

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

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

Share Your Thoughts

Once you've read *Python Feature Engineering Cookbook*, we'd love to hear your thoughts! Please [click here to go straight to the Amazon review page](#) for this book and share your feedback.

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

Download a Free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781804611302>

2. Submit your proof of purchase

3. That's it! We'll send your free PDF and other benefits to your email directly

1

Imputing Missing Data

Missing data, that is, the absence of values for certain observations, is an unavoidable problem in most data sources. Scikit-learn, the most commonly used Python library for machine learning, does not support missing values as input to machine learning models. Thus, we must remove observations with missing data or transform them into permitted values.

The act of replacing missing data with statistical estimates of missing values is called **imputation**. The goal of any imputation technique is to produce a complete dataset. There are multiple imputation methods that we can use, depending on whether the data is missing at random, the proportion of missing values, and the machine learning model we intend to use. In this chapter, we will discuss several imputation methods.

This chapter will cover the following recipes:

- Removing observations with missing data
- Performing mean or median imputation
- Imputing categorical variables
- Replacing missing values with an arbitrary number
- Finding extreme values for imputation
- Marking imputed values
- Performing multivariate imputation by chained equations

- Estimating missing data with nearest neighbors

Technical requirements

In this chapter, we will use the Matplotlib, **pandas**, NumPy, scikit-learn, and **feature-engine** Python libraries. If you need to install Python, the free Anaconda Python distribution (<https://www.anaconda.com/>) comes with most numerical computing libraries out of the box.

Feature-engine can be installed with **pip**:

```
pip install feature-engine
```

If you use Anaconda, you can install **feature-engine** with **conda**:

```
conda install -c conda-forge feature_engine
```

We will use the **Credit Approval** dataset from the *UCI Machine Learning Repository* (<https://archive.ics.uci.edu/>). To prepare the dataset, follow these steps:

1. Visit <https://archive.ics.uci.edu/ml/machine-learning-databases/credit-screening/>.
2. Click on **crx.data** to download the data.

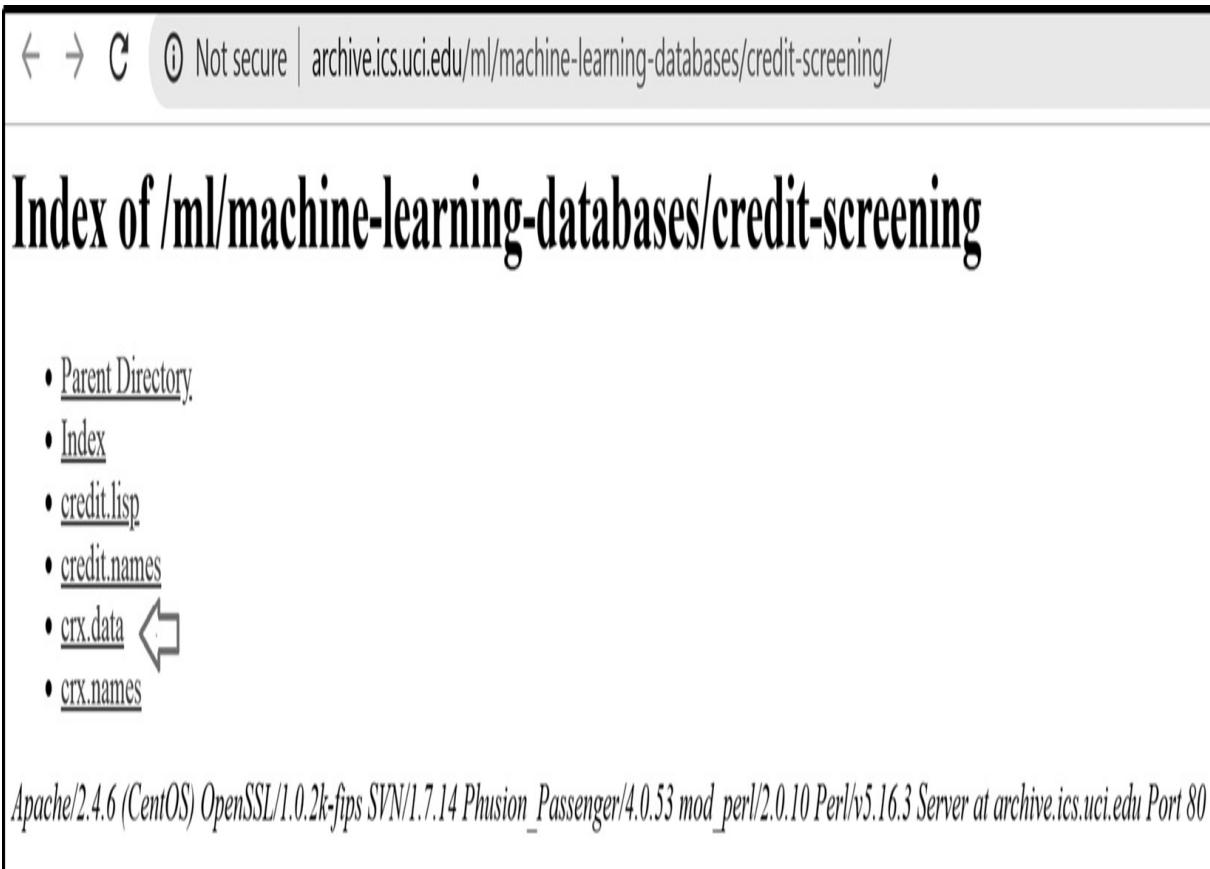


Figure 1.1 – Screenshot of the dataset download page

3. Save **crx.data** to the folder where you will run the following commands.

Open a Jupyter notebook and run the following commands.

4. Import the required Python libraries:

```
import random  
import numpy as np  
import pandas as pd
```

5. Load the data into a pandas DataFrame:

```
data = pd.read_csv("crx.data", header=None)
```

6. Create a list with variable names:

```
varnames = [f"A{s}" for s in range(1, 17)]
```

7. Add the variable names to the DataFrame:

```
data.columns = varnames
```

8. Replace the question marks (?) in the dataset with NumPy NaN values:

```
data = data.replace("?", np.nan)
```

9. Cast some numerical variables as **float** data types:

```
data["A2"] = data["A2"].astype("float")
```

```
data["A14"] = data["A14"].astype("float")
```

10. Encode the target variable as binary:

```
data["A16"] = data["A16"].map({"+":1, "-":0})
```

11. Rename the target variable:

```
data.rename(columns={"A16": "target"}, inplace=True)
```

We will introduce missing data at random places in four variables.

12. Add missing values at random positions in four variables:

```
random.seed(9001)
```

```
values = list(set([random.randint(0, len(data)) for p in range(0, 100)]))
```

```
data.loc[values, ["A3", "A8", "A9", "A10"]] = np.nan
```

13. Save your prepared data:

```
data.to_csv("credit_approval_uci.csv", index=False)
```

NOTE

Make sure you store the dataset in the same folder from which you will execute the code in the recipes.

Now, you are ready to carry on with the recipes in this chapter.

Removing observations with missing data

Complete Case Analysis (CCA), also called list-wise deletion of cases, consists of discarding observations with missing data. CCA can be applied to both categorical and numerical variables. With CCA, we preserve the distribution of the variables after the imputation, provided the data is missing at random and only in a small proportion of observations. However, if data is missing for many variables, CCA may lead to the removal of a large portion of the dataset.

How to do it...

Let's begin by making some imports and loading the dataset:

1. Let's import the **pandas** and **matplotlib** libraries:

```
import matplotlib.pyplot as plt  
import pandas as pd
```

2. Load the dataset that we prepared in the *Technical requirements* section:

```
data = pd.read_csv("credit_approval_uci.csv")
```

3. Find the proportion of missing values per variable, sort them in ascending order, and then make a bar plot, rotating the ticks on the xaxis and adding a title and the y-axis label:

```
with plt.style.context("seaborn"):  
    data.isnull().mean().sort_values(  
        ascending=True).plot.bar(rot=4  
5)  
    plt.ylabel("Proportion of missing data")  
    plt.title("Proportion of missing data per  
variable")
```

The preceding code block returns the following bar plot:

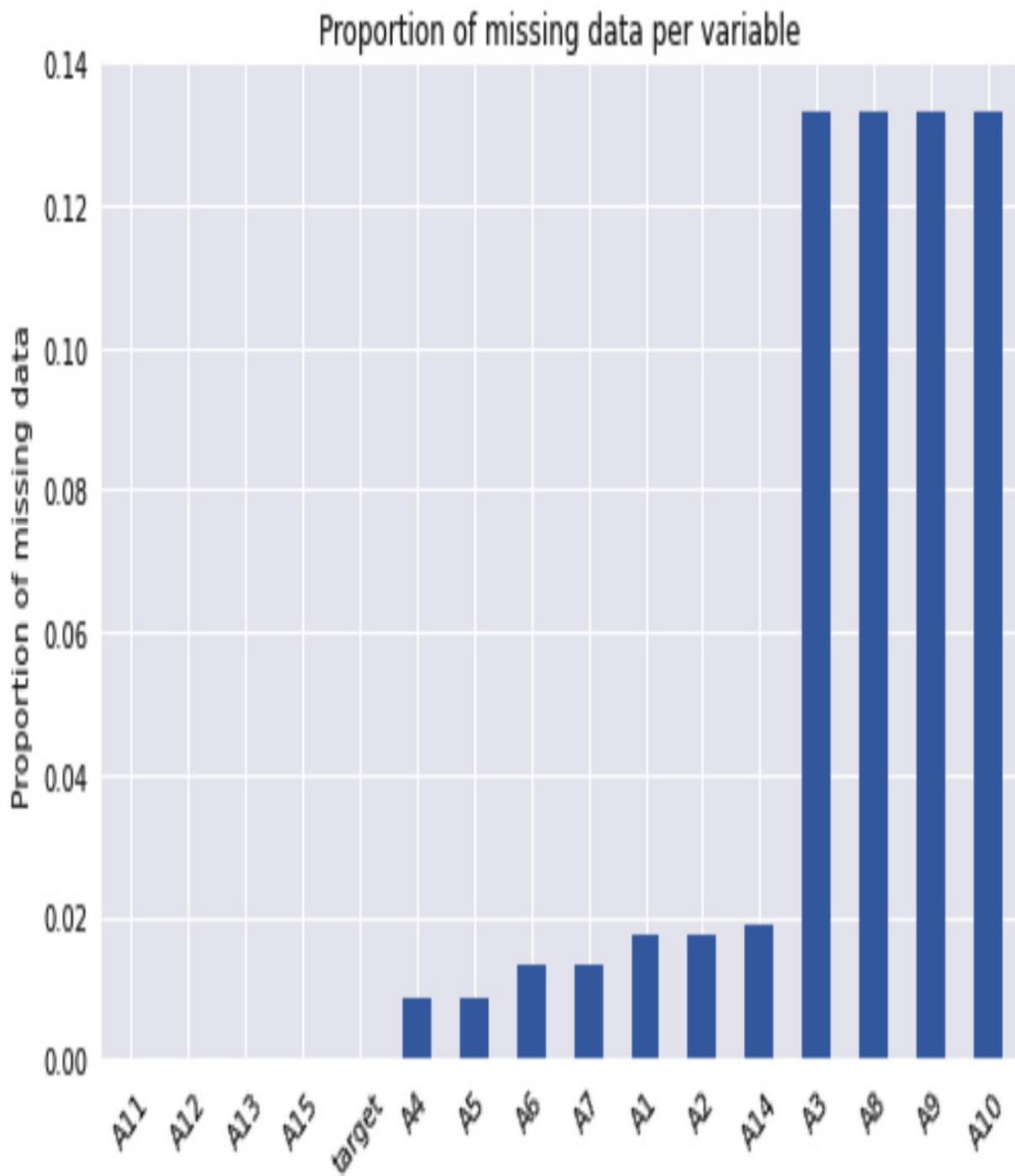


Figure 1.2 – Proportion of missing data per variable

4. Now, we'll remove observations if they have missing values in any variable:

```
data_cca = data.dropna()
```

NOTE

The **dropna()** method drops observations with any missing value by default. We can remove observations with missing data in a subset of variables like this: **data.dropna(subset=["A3", "A4"])**.

5. Let's print and compare the size of the original and complete case datasets:

```
print(f"Total number of observations: {len(data)}")  
print(f"Number of observations without missing  
data{len(  
    data_cca)}")
```

We removed more than 100 observations with missing data, as shown in the following output:

```
Total number of observations: 690
```

```
Number of observations without missing data 564
```

6. To drop observations with missing data utilizing **feature-engine**, let's import the required transformer:

```
from feature_engine.imputation import DropMissingData
```

7. Set up the imputer to automatically find the variables with missing data:

```
cca = DropMissingData(variables=None,  
missing_only=True)
```

8. Make the transformer identify the variables with missing data:

```
cca.fit(data)
```

9. Let's inspect the variables with missing data:

```
cca.variables_
```

The preceding command returns the names of the variables with missing data:

```
['A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A7', 'A8', 'A9',
'A10', 'A14']
```

10. Remove observations with missing data in the previous variables:

```
data_cca = cca.transform(data)
```

Use the code in *step 3* to corroborate the absence of missing data in the complete case dataset.

TIP

To remove observations with missing data in a subset of variables, use **DropMissingData(variables=['A3', 'A4'])**. To remove observations with missing values in at least 5% of the variables, use **DropMissingData(threshold=0.95)**.

How it works...

In this recipe, we identified and plotted the proportion of missing data in each variable and then removed all observations with missing values.

We used the **pandas isnull()** and **mean()** methods to determine the proportion of missing observations in each variable. The **isnull()** method created a Boolean vector per variable with the **True** and **False** values to indicate whether a value is missing. The **mean()** method took the average

of these values and returned the proportion of missing data. The **pandas sort_values()** method ordered the variables from that with the least to that with the most missing data. We then used the **pandas plot.bar()** method to create a bar plot. To remove observations with missing values in *any* variable, we used the **pandas dropna()** method, thereby obtaining a complete case dataset.

Finally, we removed missing data using **feature-engine**'s **DropMissingData()**, which automatically identified and stored the variables with missing data from the train set when we called the **fit()** method. With the **transform()** method, the imputer removed observations with missing data in those variables.

Performing mean or median imputation

Mean or median imputation consists of replacing missing values with the mean or median variable. The mean or median is calculated using a train set, and these values are used to impute missing data in train and test sets, as well as in all future data we intend to use with the machine learning model. Scikit-learn and **feature-engine** transformers learn the mean or median from the train set and store these parameters for future use out of the box. In this recipe, we will perform mean and median imputation using **pandas**, scikit-learn, and **feature-engine**.

TIP

Use mean imputation if variables are normally distributed and median imputation otherwise. Mean and median imputation may distort the

distribution of the original variables if there is a high percentage of missing data.

How to do it...

Let's begin this recipe:

1. First, we'll import **pandas** and the required functions and classes from scikit-learn and **feature-engine**:

```
import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.impute import SimpleImputer

from sklearn.compose import ColumnTransformer

from feature_engine.imputation import MeanMedianImputer
```

2. Let's load the dataset that we prepared in the *Technical requirements* section:

```
data = pd.read_csv("credit_approval_uci.csv")
```

3. Then, separate the data into train and test sets and their respective targets:

```
X_train, X_test, y_train, y_test = train_test_split(
    data.drop("target", axis=1),
    data["target"],
    test_size=0.3,
    random_state=0,
)
```

4. Capture the numerical variables in a list. To do this, we exclude all variables cast as objects:

```
numeric_vars =  
X_train.select_dtypes(exclude="O").columns.to_list()
```

If you execute **numeric_vars**, you will see the names of the numerical variables: `['A2', 'A3', 'A8', 'A11', 'A14', 'A15']`.

5. Capture the median values of the numerical variables in a dictionary:

```
median_values =  
X_train[numeric_vars].median().to_dict()
```

Note how we calculate the median using the train set. We will use these values to replace missing data in the train and test sets.

TIP

If instead of the median we want to capture the mean, we use **pandas mean()**.

If you execute **median_values**, you will see a dictionary with the median value per variable: `{'A2': 28.835, 'A3': 2.75, 'A8': 1.0, 'A11': 0.0, 'A14': 160.0, 'A15': 6.0}`.

6. Let's replace missing data with the median:

```
X_train = X_train.fillna(value=median_values)  
X_test = X_test.fillna(value=median_values)
```

If you execute **X_train[numeric_vars].isnull().sum()** after the imputation, the number of missing values in the numerical variables should be **0**.

TIP

The `fillna()` method returns a new dataset with imputed values by default. We can also replace missing data in the original DataFrame by setting `inplace` to `True`:

```
X_train.fillna(value=median_values, inplace=True).
```

Now, let's impute missing values with the median using scikit-learn. First, we separate the original dataset into train and test sets as we did in *step 3*.

7. Next, we capture the non-numerical variables in a list:

```
remaining_vars = [var for var in X_train.columns if var  
not in numeric_vars]
```

8. Let's set up the imputer to replace missing data with the median:

```
imputer = SimpleImputer(strategy="median")
```

MEAN IMPUTATION

To perform mean imputation, set `SimpleImputer()` like this:

```
imputer = SimpleImputer(strategy = "mean").
```

9. Indicate which variables we want to impute using the

`ColumnTransformer()` auxiliary class:

```
ct = ColumnTransformer(  
    [("imputer", imputer, numeric_vars)],  
    remainder="passthrough"  
)
```

10. Fit `SimpleImputer()` to the train set so that it learns the median values of the variables:

```
ct.fit(X_train)
```

11. Inspect the learned median values:

```
ct.named_transformers_.imputer.statistics_
```

The previous command returns the median values per variable:

```
array([28.835, 2.75, 1., 0., 160., 6.])
```

12. Let's replace missing values with the median:

```
X_train = ct.transform(X_train)
```

```
X_test = ct.transform(X_test)
```

13. **SimpleImputer()** returns NumPy arrays. Let's transform the array into a DataFrame:

```
X_train = pd.DataFrame(  
    X_train,  
    columns=numeric_vars + remaining_vars,  
)
```

14. Finally, let's perform median imputation using **feature-engine**. First, split the dataset into train and test sets, as in *step 3*. Next, let's set up the imputer to replace missing data in numerical variables with the median:

```
imputer = MeanMedianImputer(  
    imputation_method="median",  
    variables=numeric_vars,  
)
```

TIP

To perform mean imputation, change `imputation_method` to "mean".

15. Fit the imputer so that it learns the median values for the specified variables:

```
imputer.fit(X_train)
```

16. Inspect the learned medians:

```
imputer.imputer_dict_
```

The previous command returns the median values in a dictionary:

```
{'A2': 28.835, 'A3': 2.75, 'A8': 1.0, 'A11': 0.0, 'A14':  
160.0, 'A15': 6.0}
```

17. Finally, let's replace the missing values with the median:

```
X_train = imputer.transform(X_train)  
X_test = imputer.transform(X_test)
```

Feature-engine's `MeanMedianImputer()` returns a DataFrame. You can check that the imputed variables do not contain missing values using `X_train[numeric_vars].isnull().mean()`.

How it works...

In this recipe, we replaced missing data with the variable's median values using **pandas**, scikit-learn, and **feature-engine**.

The mean or median values should be learned from the train set variables. Thus, we divided the dataset into train and test sets using scikit-learn's `train_test_split()` function. The function takes the predictor variables,

the target, the fraction of observations to retain in the test set, and a `random_state` value for reproducibility as arguments. We obtained a train set with 70% of the original observations and a test set with 30% of the original observations. The 70:30 split was done at random.

To impute missing data with `pandas`, in *step 6*, we created a dictionary with the numerical variable names as keys and their medians as values, utilizing the training set. To capture the median, we used `pandas median()`, and to return a dictionary, we used `pandas to_dict()`. To replace missing data with the median, we used the `pandas fillna()` method in the train and test sets, passing the dictionary with the median values per variable as a parameter.

To replace the missing values using scikit-learn, we used `SimpleImputer()` with `strategy` set to "`median`". To impute only numerical variables, we used `ColumnTransformer()`, which takes the imputer and the numerical variable names in a list as parameters. With the `passthrough` argument set to "`remainder`", we make `ColumnTransformer()` return all the variables in the final output, the imputed ones followed by the remaining ones.

With the `fit()` method, `SimpleImputer()` learned the median of each numerical variable in the train set and stored them in its `statistics_` attribute. With `transform()`, the transformer replaced the missing values with the medians.

To replace missing values with `feature-engine`, we set up `MeanMedianImputer()` with `imputation_method` set to "`median`" and passed the names of the variables to impute in a list. With the `fit()`

method, the transformer learned and stored the median values of the specified variables in a dictionary in its `imputer_dict_` attribute. With the `transform()` method, the missing values were replaced, returning a pandas DataFrame.

NOTE

`SimpleImputer()` operates on the entire DataFrame and returns NumPy arrays. To impute just a subset of variables, we need to use `ColumnTransformer()`. In contrast, `MeanMedianImputer()` can take an entire DataFrame and yet it will only impute the specified variables, returning a pandas DataFrame.

Imputing categorical variables

Categorical variables usually contain strings as values, instead of numbers. We replace missing data in categorical variables with the most frequent category, or with a different string. Frequent categories are estimated using the train set and then used to impute values in the train, test, and future datasets. Thus, we need to learn and store these values, which we can do using scikit-learn and `feature-engine`'s out-of-the-box transformers. In this recipe, we will replace missing data in categorical variables with the most frequent category, or with an arbitrary string.

How to do it...

To begin, let's make a few imports and prepare the data:

1. Let's import `pandas` and the required functions and classes from scikit-learn and `feature-engine`:

```
import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.impute import SimpleImputer

from sklearn.compose import ColumnTransformer

from feature_engine.imputation import CategoricalImputer
```

2. Then, load the dataset that we prepared in the *Technical requirements* section:

```
data = pd.read_csv("credit_approval_uci.csv")
```

3. Separate the data into train and test sets and their respective targets:

```
X_train, X_test, y_train, y_test = train_test_split(
    data.drop("target", axis=1),
    data["target"],
    test_size=0.3,
    random_state=0, )
```

4. Let's capture the categorical variables in a list:

```
categorical_vars = X_train.select_dtypes(
    include="O").columns.to_list()
```

5. Now, capture the most frequent category per categorical variable in a dictionary:

```
frequent_values = X_train[
    categorical_vars].mode().iloc[0].to_dict()
```

6. Then, replace missing values with the frequent categories:

```
x_train = x_train.fillna(value=frequent_values)  
x_test = x_test.fillna(value=frequent_values)
```

TIP

`fillna()` returns a new dataset with imputed values by default. We can replace missing data in the original DataFrame by executing `x_train.fillna(value=frequent_values, inplace=True)`.

7. To replace missing data with a specific string, let's create an imputation dictionary with the categorical variable names as the keys and an arbitrary string as the values:

```
imputation_dict = {var: "no_data" for var  
in categorical_vars}
```

Now, we can use this dictionary and the code in *step 6* to replace missing data.

TIP

With the `pandas value_counts()` or `unique()` methods over each categorical variable, we can see the string added by the imputation. Try executing, for example, `x_train["A1"].value_counts()`.

Now, let's impute missing values by the most frequent category using scikit-learn. First, we separate the original dataset into train and test sets as we did in *step 3*.

8. Make a list with the numerical variables:

```
imputation_dict = {var: "no_data" for var  
in categorical_vars}
```

9. Set up the imputer to find the most frequent category:

```
imputer = SimpleImputer(strategy='most_frequent')
```

TIP

SimpleImputer() will learn the mode for numerical and categorical variables alike. But in practice, mode imputation is done for categorical variables only.

10. Let's indicate, using **ColumnTransformer()**, which variables to

impute:

```
ct = ColumnTransformer(  
    [("imputer", imputer, categorical_vars)],  
    remainder="passthrough"  
)
```

11. Fit the imputer to the train set so that it learns the most frequent values:

```
ct.fit(X_train)
```

12. Inspect the most frequent values learned by the imputer:

```
ct.named_transformers_.imputer.statistics_
```

The previous command returns the most frequent values per variable:

```
array(['b', 'u', 'g', 'c', 'v', 't', 'f', 'f', 'g'],  
      dtype=object)
```

13. Let's replace missing values with the frequent categories:

```
X_train = ct.transform(X_train)
```

```
X_test = ct.transform(X_test)
```

14. **SimpleImputer()** will return a NumPy array. Let's transform it into a pandas DataFrame:

```
x_train = pd.DataFrame(  
    x_train,  
    columns=categorical_vars + remaining_vars,  
)
```

Note that the imputed variables will be the first columns in the array.

TIP

To impute missing data with a string instead of the most frequent category, set **SimpleImputer()**, as follows:

```
imputer = SimpleImputer(  
    strategy="constant", fill_value="missing"  
)
```

Finally, let's impute missing values using **feature-engine**. First, we need to separate the data into train and test sets, just like we did in *step 3*.

15. Next, let's set up the imputer to replace missing data in categorical variables by their most frequent value:

```
imputer = CategoricalImputer(  
    imputation_method="frequent",  
    variables=categorical_vars,  
)
```

TIP

With the **variables** parameter set to **None**, **CategoricalImputer()** will automatically find all categorical variables in the train set for imputation.

16. Fit the imputer to the train set so that it learns the most frequent categories:

```
imputer.fit(X_train)
```

TIP

To impute categorical variables with a specific string, we set **imputation_method** to "missing" and **fill_value** to the desired string.

17. Inspect the learned frequent categories:

```
imputer.imputer_dict_
```

We can see the dictionary with the most frequent values in the following output:

```
{'A1': 'b', 'A4': 'u', 'A5': 'g', 'A6': 'c', 'A7': 'v',
'A9': 't', 'A10': 'f', 'A12': 'f', 'A13': 'g'}
```

18. Finally, let's replace the missing values with frequent categories:

```
X_train = imputer.transform(X_train)
```

```
X_test = imputer.transform(X_test)
```

CategoricalImputer() returns a pandas DataFrame with the imputed values.

How it works...

In this recipe, we replaced the missing values of categorical variables with the most frequent categories or an arbitrary string, using **pandas**, scikit-learn, and **feature-engine**.

To impute missing data with **pandas**, in *step 5*, we created a dictionary with the variable names as keys and the frequent categories as values. To capture the frequent categories, we used **pandas mode()**, and to return a dictionary, we used **pandas to_dict()**. To replace missing data with the mode, we used the **pandas fillna()** method, passing the dictionary with the variables and their frequent categories as parameters.

TIP

A variable can have more than one mode. If this is the case, we may want to choose a different imputation method.

To replace missing data with an arbitrary string, we created a dictionary with the "**no_data**" value for each variable and used this dictionary as an argument to the **pandas fillna() method**.

To replace the missing values using scikit-learn, we used **SimpleImputer()** with **strategy** set to "**frequent**". To impute only categorical variables, we used **ColumnTransformer()**, which takes the imputer and the variable list as arguments. With the **passthrough** argument set to "**remainder**", **ColumnTransformer()** returned all the variables in the final output, the imputed ones first followed by the remaining ones.

With the **fit()** method, **SimpleImputer()** learned the frequent categories for each categorical variable and stored them in its **statistics_** attribute.

With the `transform()` method, missing data was replaced with the learned parameters.

`SimpleImputer()` returns a NumPy array by default. We converted this array into a `pandas` DataFrame. We had to pass the variable names in the correct order: the imputed variables are located first in the array, followed by the remaining variables.

To replace missing values with `feature-engine`, we set up `CategoricalImputer()` with `imputation_method` set to "`frequent`" and passed the names of the variables to impute in a list. With the `fit()` method, the transformer learned and stored the most frequent categories of the specified variables in a dictionary in its `imputer_dict_` attribute. With the `transform()` method, the missing values were replaced with the learned parameters.

TIP

Note that, unlike `SimpleImputer()`, `CategoricalImputer()` will only impute categorical variables, unless specifically told not to do so by setting the `ignore_format` parameter to `True`.

Replacing missing values with an arbitrary number

Arbitrary number imputation consists of replacing missing data with an arbitrary value. Commonly used values include `999`, `9999`, or `-1` for positive distributions. This method is suitable for numerical variables. For categorical variables, the equivalent method is to replace missing data with

an arbitrary string, as described in the *Imputing categorical variables* recipe.

When replacing missing values with arbitrary numbers, we need to be careful not to select a value close to the mean, the median, or any other common value of the distribution.

TIP

Arbitrary number imputation can be used when data is not missing at random, when we are building non-linear models, and when the percentage of missing data is high. This imputation technique distorts the original variable distribution.

In this recipe, we will impute missing data with arbitrary numbers using **pandas**, scikit-learn, and **feature-engine**.

How to do it...

Let's begin by importing the necessary tools and loading the data:

1. Import **pandas** and the required functions and classes:

```
import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.impute import SimpleImputer

from feature_engine.imputation import
    ArbitraryNumberImputer
```

2. Let's load the dataset that we prepared in the *Technical requirements* section:

```
data = pd.read_csv("credit_approval_uci.csv")
```

3. Separate the data into train and test sets:

```
x_train, x_test, y_train, y_test = train_test_split(  
    data.drop("target", axis=1),  
    data["target"],  
    test_size=0.3,  
    random_state=0,  
)
```

4. We will select arbitrary values greater than the maximum value of the distribution. Let's find the maximum value of four numerical variables:

```
x_train[['A2', 'A3', 'A8', 'A11']].max()
```

The following is the output of the preceding code block:

A2	76.750
A3	26.335
A8	20.000
A11	67.000
dtype:	float64

Figure 1.3 – Maximum value of each variable

5. Replace the missing values with **99**, which is bigger than the maximum values of the numerical variables in *step 4*:

```
X_train[["A2", "A3", "A8", "A11"]] = X_train[["A2", "A3", "A8", "A11"]].fillna(99)  
X_test[["A2", "A3", "A8", "A11"]] = X_test[["A2", "A3", "A8", "A11"]].fillna(99)
```

TIP

To impute different variables with different values using **pandas** **fillna()**, pass a dictionary with the keys as variables and the values as arbitrary numbers, for example, **imputation_dict = {"A2": -1, "A3": -1, "A8": 999, "A11": 9999}**.

Now, we'll impute missing values with an arbitrary number using scikit-learn. First, we separate the data into train and test sets as in *step 3*.

6. Let's set up the imputer to replace missing values with **99**:

```
imputer = SimpleImputer(  
    strategy='constant', fill_value=99  
)
```

TIP

If your dataset contains categorical variables, **SimpleImputer()** will add **99** to those variables as well if any values are missing.

7. Fit the imputer to a slice of the train set containing the variables to impute:

```
imputer.fit(X_train[["A2", "A3", "A8", "A11"]])
```

8. Replace the missing values with **99** in the desired variables:

```
X_train[["A2", "A3", "A8", "A11"]] = imputer.transform(  
    X_train[["A2", "A3", "A8", "A11"]])  
  
X_test[["A2", "A3", "A8", "A11"]] = imputer.transform(  
    X_test[["A2", "A3", "A8", "A11"]])
```

To finish, let's impute missing values using **feature-engine**. First, we separate the data into train and test sets, just like we did in *step 3*.

9. Next, let's set up the imputer to replace missing values with **99**, specifying the variables to impute:

```
imputer = ArbitraryNumberImputer(  
    arbitrary_number=99,  
    variables=["A2", "A3", "A8", "A11"],  
)
```

TIP

ArbitraryNumberImputer() can automatically select all numerical variables in the train set if we set the **variables** parameter to **None**.

10. Finally, let's replace the missing values with **99**:

```
X_train = imputer.fit_transform(X_train)  
X_test = imputer.transform(X_test)
```

TIP

If you want to impute different variables with different numbers, pass a dictionary with the variable names as keys and the arbitrary

numbers as values to **ArbitraryNumberImputer**.

We have now replaced missing data with arbitrary numbers using three different open source libraries.

How it works...

In this recipe, we replaced missing values in numerical variables with an arbitrary number using **pandas**, scikit-learn, and **feature-engine**.

To determine which arbitrary value to use, we inspected the maximum values of four numerical variables using the **pandas max()** method. Next, we chose the value **99**, because it was greater than the maximum values of the selected variables. In *step 5*, we replaced missing data in the four numerical variables with the **pandas fillna()** method, passing **99** as an argument.

To replace missing values using scikit-learn, we utilized **SimpleImputer()**, with **strategy** set to "**constant**", and specified **99** in the **fill_value** argument. Next, we fitted the imputer to a slice of the train set with the numerical variables and also replaced missing values using the **transform()** method in a slice of the train and test sets containing the numerical variables. In doing so, we replaced the variables with their imputed version in the original data and avoided having to convert the resulting NumPy array back into a DataFrame.

Finally, we replaced missing values with **ArbitraryValueImputer()** from **feature-engine**, specifying the value **99** and the variables to impute as parameters. Next, we applied the **fit_transform()** method to replace

missing data in the train set and the `transform()` method to replace missing data in the test set.

Finding extreme values for imputation

Replacing missing values with a value at the end of the variable distribution (extreme values) is equivalent to replacing them with an arbitrary value, but instead of identifying the arbitrary values manually, these values are automatically selected as those at the very end of the variable distribution.

Missing data can be replaced with a value that is greater or smaller than the remaining values in the variable. To select a value that is greater, we can use the mean plus a factor of the standard deviation, or the 75th quantile + (IQR * 1.5), where IQR is the IQR given by the 75th quantile - the 25th quantile. To replace missing data with values that are smaller than the remaining values, we can use the mean minus a factor of the standard deviation, or the 25th quantile – (IQR * 1.5).

NOTE

End-of-tail imputation may distort the distribution of the original variables, so it may not be suitable for linear models.

In this recipe, we will implement end-of-tail or extreme value imputation using `pandas` and `feature-engine`.

How to do it...

To begin this recipe, let's import the necessary tools and load the data:

1. Let's import `pandas` and the required function and class:

```
import pandas as pd  
  
from sklearn.model_selection import train_test_split  
  
from feature_engine.imputation import EndTailImputer
```

2. Load the dataset we prepared in the *Technical requirements* section:

```
data = pd.read_csv("credit_approval_uci.csv")
```

3. Capture the numerical variables in a list, excluding the target:

```
numeric_vars = [  
    var for var in data.select_dtypes(  
        exclude="O").columns.to_list() if var  
    != "target"  
]
```

4. Separate the data into train and test sets, keeping only the numerical variables:

```
X_train, X_test, y_train, y_test = train_test_split(  
    data[numeric_vars],  
    data["target"],  
    test_size=0.3,  
    random_state=0,  
)
```

5. Calculate the IQR for all numerical variables:

```
IQR = X_train.quantile(0.75) - X_train.quantile(0.25)
```

If we execute **IQR**, we will see a **pandas** Series containing its values:

```
A2          16.4200
A3          6.5825
A8          2.8350
A11         3.0000
A14         212.0000
A15         450.0000
target      1.0000
dtype: float64
```

6. Let's create a dictionary with the variable names as keys and the imputation replacements as values, returned by adding 1.5 times the IQR to the 75th quantile:

```
imputation_dict = (
    X_train.quantile(0.75) + 1.5 * IQR).to_dict()
```

TIP

If we want to use the Gaussian approximation instead of the IQR proximity rule, we can calculate the value to replace missing data using `imputation_dict = (X_train.mean() + 3 * X_train.std()).to_dict()`.

7. Finally, let's replace missing data:

```
X_train = X_train.fillna(value=imputation_dict)
X_test = X_test.fillna(value=imputation_dict)
```

Note how we calculated the value to impute the missing data using the variables in the train set and then used these to impute train and test sets.

TIP

We can also replace missing data with values at the left tail of the distribution using `value = X_train[var].quantile(0.25) - 1.5 * IQR` or `value = X_train[var].mean() - 3*X_train[var].std()`.

To finish, let's impute missing values using **feature-engine**. First, we need to separate the data into train and test sets, just like in *step 4* of this recipe.

8. Next, let's set up the imputer to estimate a value at the right of the distribution using the IQR proximity rule, specifying the variables to impute:

```
imputer = EndTailImputer(  
    imputation_method="iqr",  
    tail="right",  
    fold=3,  
    variables=None,  
)
```

TIP

To use the mean and standard deviation to calculate the replacement values, we need to set `imputation_method="Gaussian"`. We can use '`left`' or '`right`' in the `tail` argument to specify the side of the distribution where we'll place the missing values.

9. Let's fit **EndTailImputer()** to the train set so that it learns the parameters:

```
imputer.fit(X_train)
```

10. Let's inspect the learned values:

```
imputer.imputer_dict_
```

The previous command returns a dictionary with the imputation values per variable:

```
{'A2': 88.18,  
'A3': 27.31,  
'A8': 11.504999999999999,  
'A11': 12.0,  
'A14': 908.0,  
'A15': 1800.0}
```

11. Finally, let's replace the missing values:

```
X_train = imputer.transform(X_train)  
X_test = imputer.transform(X_test)
```

Remember that you can corroborate that the missing values were replaced by using `X_train[['A2', 'A3', 'A8', 'A11', 'A14', 'A15']].isnull().mean()`.

How it works...

In this recipe, we replaced the missing values in numerical variables with a number at the end of the distribution using **pandas** and **feature-engine**.

We first calculated the values at the end of the distributions manually according to the formulas described in the introduction to this recipe. We determined the quantiles using **pandas quantile()** and the mean and standard deviation using **pandas mean()** and **std()**. Next, we used **pandas fillna()** to replace the missing values.

To replace missing values with **EndTailImputer()** from **feature-engine**, we set **distribution** to '**iqr**' to calculate the values with the IQR proximity rule and **tail** to '**right**' to place values at the right tail. We also specified the variables to impute in a list. With the **fit()** method, the imputer learned and stored the values to use for the imputation in a dictionary in the **imputer_dict_** attribute. With the **transform()** method, the missing values were replaced, returning DataFrames.

Marking imputed values

A missing indicator is a binary variable that takes the value **1** or **True** to indicate whether a value was missing, or **0** or **False** otherwise. It is common practice to replace missing observations with the mean, median, or most frequent category while simultaneously marking those missing observations with missing indicators. In this recipe, we will learn how to add missing indicators using **pandas**, scikit-learn, and **feature-engine**.

How to do it...

Let's begin by making some imports and loading the data:

1. Let's import the required libraries, functions, and classes:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from feature_engine.imputation import(
    AddMissingIndicator, CategoricalImputer,
    MeanMedianImputer
)
```

2. Load the dataset that we prepared in the *Technical requirements* section:

```
data = pd.read_csv("credit_approval_uci.csv")
```

3. Separate the data into train and test sets:

```
X_train, X_test, y_train, y_test = train_test_split(
    data.drop("target", axis=1),
    data["target"],
    test_size=0.3,
    random_state=0,
)
```

4. Capture the variable names in a list:

```
varnames = ["A1", "A3", "A4", "A5", "A6", "A7", "A8"]
```

5. Create names for the missing indicators in a list:

```
indicators = [f"var}_na" for var in varnames]
```

If we execute **indicators**, we will see the names of the new variables:

```
['A1_na', 'A3_na', 'A4_na', 'A5_na', 'A6_na', 'A7_na',  
'A8_na'].
```

6. Add the missing indicators with the **1** or **0** value:

```
X_train[indicators] =  
X_train[varnames].isna().astype(int)  
  
X_test[indicators] =  
X_test[varnames].isna().astype(int)
```

TIP

If you want the indicators to have the **True** and **False** values, remove **astype(int)** in step 6.

7. Let's inspect the result of the preceding code block:

```
X_train.head()
```

We can see the newly added variables at the end of the DataFrame:

	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	...	A13	A14	A15	A1_na	A3_na	A4_na	A5_na	A6_na	A7_na	A8_na
596	a	46.08	3.000	u	g	c	v	2.375	t	t ...	g	396.0	4159	0	0	0	0	0	0	0	
303	a	15.92	2.875	u	g	q	v	0.085	f	f ...	g	120.0	0	0	0	0	0	0	0	0	
204	b	36.33	2.125	y	p	w	v	0.085	t	t ...	g	50.0	1187	0	0	0	0	0	0	0	
351	b	22.17	0.585	y	p	ff	ff	0.000	f	f ...	g	100.0	0	0	0	0	0	0	0	0	
118	b	57.83	7.040	u	g	m	v	14.000	t	t ...	g	360.0	1332	0	0	0	0	0	0	0	

Figure 1.4 – DataFrame with the missing indicators

Now, let's add missing indicators using **feature-engine** instead. First, we need to divide the data, just like we did in *step 3*.

8. Next, set up the imputer to add binary indicators to every variable with missing data:

```
imputer = AddMissingIndicator(
    variables=None, missing_only=True
)
```

9. Fit the imputer to the train set so that it finds the variables with missing data:

```
imputer.fit(X_train)
```

If we execute **imputer.variables_**, we will find the variables for which missing indicators will be added.

10. Finally, let's add the missing indicators:

```
X_train = imputer.transform(X_train)  
X_test = imputer.transform(X_test)
```

11. Create a pipeline to add missing indicators to categorical and numerical variables, then impute categorical variables with the most frequent category, and numerical variables with the mean:

```
pipe = Pipeline(  
    [  
        ("ind",  
            AddMissingIndicator(missing_only=True)  
        ),  
        ("cat",  
            CategoricalImputer(  
                imputation_method="frequent")  
        ),  
        ("num", MeanMedianImputer()),  
    ]  
)
```

NOTE

Feature-engine imputers automatically identify all numerical or categorical variables, modifying only the appropriate variables. So there is no need to slice the data or pass the variable names as arguments to the transformers in this case.

12. Now, let's add the indicators and impute missing values:

```
X_train = pipe.fit_transform(X_train)  
X_test = pipe.transform(X_test)
```

Use `X_train.isnull().sum()` to corroborate that there is no data missing.

Finally, let's add missing indicators and simultaneously impute numerical and categorical variables with the mean and most frequent categories respectively, utilizing scikit-learn.

13. Make a list with the names of the numerical and categorical variables:

```
numvars = X_train.select_dtypes(  
    exclude="O").columns.to_list()  
  
catvars = X_train.select_dtypes(  
    include="O").columns.to_list()
```

14. Set up a pipeline to perform mean and frequent category imputation while marking the missing data:

```
pipe = ColumnTransformer(  
    [  
        ("num_imputer",  
         SimpleImputer(  
             strategy="mean",  
             add_indicator=True
```

```
        ),
    numvars
),
(
    "cat_imputer",
    SimpleImputer(
        strategy="most_frequent",
        add_indicator=True
    ),
    catvars,
),
]
)
```

15. Now, let's carry out the imputation:

```
X_train = pipe.fit_transform(X_train)
X_test = pipe.transform(X_test)
```

X_train is a NumPy array with the numerical variables and their missing indicators first, followed by the categorical variables with their indicators.

How it works...

To add missing indicators using **pandas**, we used the **isna()** method, which created a new vector assigning the value of **True** if there was a

missing value or **False** otherwise. We followed **isna()** with **astype(int)** to convert the Boolean vectors into binary vectors with values **1** and **0**.

To add a missing indicator with **feature-engine**, we used **AddMissingIndicator()** and fitted it to the train set, so that the transformer identifies the variables with missing data. Then, we used the **transform()** method to add missing indicators to the train and test sets.

Finally, we added missing indicators and replaced missing data with the mean and most frequent category using scikit-learn. Utilizing **Pipeline**, we lined up two instances of **SimpleImputer()**, the first to impute data with the mean and the second to impute data with the most frequent category. In both cases, we set the **add_indicator** parameter to **True**, so that **SimpleImputer()** also added the missing indicators. We wrapped **SimpleImputer()** with **ColumnTransformer()** to specifically modify numerical or categorical variables.

Performing multivariate imputation by chained equations

Multivariate imputation methods, as opposed to univariate imputation, use multiple variables to estimate the missing values. In other words, the missing values of a variable are modeled based on the other variables in the dataset. **Multivariate Imputation by Chained Equations (MICE)** models each variable with missing values as a function of the remaining variables and uses that estimate for imputation.

The following steps are required to perform MICE:

1. A simple univariate imputation is performed for every variable with missing data, for example, median imputation.
2. One specific variable is selected, say, **var_1**, and the missing values are set back to missing.
3. A model is trained to predict **var_1** using the remaining variables as input features.
4. The missing values of **var_1** are replaced with the new estimates.
5. *Steps 2 to 4* are repeated for each of the remaining variables.

Once all the variables have been modeled based on the rest, a cycle of imputation is concluded. Multiple imputation cycles are carried out, typically 10. The idea is that by the end of the cycles, the distribution of the imputation parameters should have converged, which means that we should have found the best estimates for the missing data.

In this recipe, we will implement MICE using scikit-learn.

How to do it...

To begin the recipe, let's import the required libraries and load the data:

1. Let's import the required Python libraries and a function to split the data:

```
import pandas as pd  
import matplotlib.pyplot as plt  
from sklearn.model_selection import train_test_split
```

2. Import a regression algorithm from scikit-learn:

```
from sklearn.linear_model import BayesianRidge
```

3. Import **IterativeImputer** to carry out multivariate imputation:

```
from sklearn.experimental import  
enable_iterative_imputer  
  
from sklearn.impute import IterativeImputer
```

4. Load the dataset that we prepared in the *Technical requirements* section only with some numerical variables:

```
variables = ["A2", "A3", "A8", "A11", "A14", "A15",  
"target"]  
  
data = pd.read_csv("credit_approval_uci.csv",  
                   usecols=variables)
```

5. Divide the data into train and test sets:

```
X_train, X_test, y_train, y_test = train_test_split(  
    data.drop("target", axis=1),  
    data["target"],  
    test_size=0.3,  
    random_state=0,  
)
```

6. Let's create a MICE imputer using Bayes regression, specifying the number of iteration cycles and setting **random_state** for reproducibility:

```
imputer = IterativeImputer(  
    estimator= BayesianRidge(),
```

```
    max_iter=10,  
    random_state=0,  
)
```

TIP

IterativeImputer() contains other useful arguments. For example, we can specify the first imputation strategy using the **initial_strategy** parameter. We can choose from the mean, median, mode, or arbitrary imputation. We can also specify how we want to cycle over the variables, either randomly or from the one with the fewest missing values to the one with the most.

7. Let's fit **IterativeImputer()** to the train set so that it trains the estimators to predict the missing values in each variable:

```
imputer.fit(X_train)
```

TIP

We can use any regression model to estimate the missing data with **IterativeImputer()**.

8. Finally, let's fill in missing values in both the train and test set:

```
X_train = imputer.transform(X_train)  
X_test = imputer.transform(X_test)
```

Remember that scikit-learn returns NumPy arrays and not DataFrames. To corroborate the lack of missing data, we can execute
pd.DataFrame(X_train).isnull().sum().

How it works...

In this recipe, we performed multivariate imputation using **IterativeImputer()** from scikit-learn. We specified Bayes regression as the estimator and carried out 10 rounds of imputation over the entire dataset. By default, the first imputation round is done with the mean of the variable. We fitted **IterativeImputer()** to the train set so that each variable was modeled based on the remaining variables in the dataset. Next, we transformed the train and test sets with the **transform()** method in order to replace missing data with their estimates.

NOTE

This transformer can only impute missing data in numerical variables based on numerical variables.

See also

To learn more about MICE, take a look at the following links:

- A multivariate technique for multiplying imputing missing values using a sequence of regression models:
https://www.researchgate.net/publication/244959137_A_Multivariate_Technique_for_Multiply_Imputing_Missing_Values_Using_a_Sequence_of_Regression_Models.
- Multiple Imputation by Chained Equations: What is it and how does it work?: <https://www.jstatsoft.org/article/download/v045i03/550>.

Estimating missing data with nearest neighbors

In imputation with **K-Nearest Neighbors (KNN)**, missing values are replaced with the mean value from their k closest neighbors. The neighbors of each observation are found utilizing distances like the Euclidean distance, and the replacement value can be estimated as the mean or weighted mean of the neighbor's value, where further neighbors have less influence on the replacement value. In this recipe, we will perform KNN imputation using scikit-learn.

How to do it...

To proceed with the recipe, let's import the required libraries and prepare the data:

1. Let's import the required libraries, classes, and functions:

```
import matplotlib.pyplot as plt  
import pandas as pd  
from sklearn.model_selection import train_test_split  
from sklearn.impute import KNNImputer
```

2. Let's load the dataset that we prepared in the *Technical requirements* section only with some numerical variables:

```
variables = ["A2", "A3", "A8", "A11", "A14", "A15",  
"target"]  
  
data = pd.read_csv("credit_approval_uci.csv",  
usecols=variables)
```

3. Let's divide the data into train and test sets:

```
x_train, x_test, y_train, y_test = train_test_split(  
    data.drop("target", axis=1),  
    data["target"],  
    test_size=0.3,  
    random_state=0,  
)
```

4. Set up the imputer to find the closest **5** neighbors, utilizing Euclidean distance and weighting the neighbors so that the furthest neighbors have a smaller influence over the value of the replacement:

```
imputer = KNNImputer(n_neighbors=5, weights="distance")
```

5. Find the closest neighbors:

```
imputer.fit(x_train)
```

6. Replace the missing values with the weighted mean of the values shown by the neighbors:

```
x_train = imputer.transform(x_train)  
x_test = imputer.transform(x_test)
```

The result is a NumPy array with the missing data replaced.

How it works...

In this recipe, we replaced missing data with the average value shown by each observation's closest k neighbors. We set up **KNNImputer()** to find

each observation's five closest neighbors based on the Euclidean distance. The replacement values were estimated as the weighted average of the values shown by the five closest neighbors. With **transform()**, the imputer replaced the missing data.

2

Encoding Categorical Variables

Categorical variables are those whose values are selected from a group of categories or labels. For example, the **Gender** variable with the values of **Male** and **Female** is categorical, and so is the **marital status** variable with the values of **never married**, **married**, **divorced**, and **widowed**. In some categorical variables, the labels have an intrinsic order; for example, in the **Student's grade** variable, the values of **A**, **B**, **C**, and **Fail** are ordered, with **A** being the highest grade and **Fail** being the lowest. These are called ordinal categorical variables. Variables in which the categories do not have an intrinsic order are called nominal categorical variables, such as the **City** variable, with the values of **London**, **Manchester**, **Bristol**, and so on.

The values of categorical variables are often encoded as strings. To train mathematical or machine learning models, we need to transform those strings into numbers. The act of replacing strings with numbers is called **categorical encoding**. In this chapter, we will discuss multiple categorical encoding methods.

This chapter will cover the following recipes:

- Creating binary variables through one-hot encoding
- Performing one-hot encoding of frequent categories
- Replacing categories with counts or the frequency of observations
- Replacing categories with ordinal numbers

- Performing ordinal encoding based on the target value
- Implementing target mean encoding
- Encoding with the Weight of Evidence
- Grouping rare or infrequent categories
- Performing binary encoding

Technical requirements

In this chapter, we will use the pandas, NumPy, and Matplotlib Python libraries, as well as scikit-learn and Feature-engine. For guidelines on how to obtain these libraries, visit the *Technical requirements* section of [*Chapter 1, Imputing Missing Data*](#).

We will also use the open-source **Category Encoders** Python library, which can be installed using **pip**:

```
pip install category_encoders
```

To learn more about **Category Encoders**, visit the following link:
https://contrib.scikit-learn.org/category_encoders/.

We will also use the Credit Approval dataset, which is available in the UCI Machine Learning Repository at
<https://archive.ics.uci.edu/ml/datasets/credit+approval>.

To prepare the dataset, follow these steps:

1. Visit <http://archive.ics.uci.edu/ml/machine-learning-databases/credit-screening/> and click on **crx.data** to download the data:



C

ⓘ Not secure | archive.ics.uci.edu/ml/machine-learning-databases/credit-screening/

Index of /ml/machine-learning-databases/credit-screening

- [Parent Directory](#)
- [Index](#)
- [credit.lisp](#)
- [credit.names](#)
- [crx.data](#) ↗
- [crx.names](#)

Apache/2.4.6 (CentOS) OpenSSL/1.0.2k-fips SVN/1.7.14 Phusion Passenger/4.0.53 mod_perl/2.0.10 Perl/v5.16.3 Server at archive.ics.uci.edu Port 80

Figure 2.1 – The index directory for the Credit Approval dataset

2. Save **crx.data** to the folder where you will run the following commands.

After downloading the data, open up a Jupyter Notebook and run the following commands.

3. Import the required libraries:

```
import random  
import numpy as np  
import pandas as pd
```

4. Load the data:

```
data = pd.read_csv("crx.data", header=None)
```

5. Create a list containing the variable names:

```
varnames = [f"A{s}" for s in range(1, 17)]
```

6. Add the variable names to the DataFrame:

```
data.columns = varnames
```

7. Replace the question marks in the dataset with NumPy NaN values:

```
data = data.replace("?", np.nan)
```

8. Cast some numerical variables as **float** data types:

```
data["A2"] = data["A2"].astype("float")
```

```
data["A14"] = data["A14"].astype("float")
```

9. Encode the target variable as binary:

```
data["A16"] = data["A16"].map({"+": 1, "-": 0})
```

10. Rename the target variable:

```
data.rename(columns={"A16": "target"}, inplace=True)
```

11. Make lists that contain categorical and numerical variables:

```
cat_cols = [  
    c for c in data.columns if data[c].dtypes=="O"]  
  
num_cols = [  
    c for c in data.columns if data[c].dtypes!="O"]
```

12. Fill in the missing data:

```
data[num_cols] = data[num_cols].fillna(0)
```

```
data[cat_cols] = data[cat_cols].fillna("Missing")
```

13. Save the prepared data:

```
data.to_csv("credit_approval_uci.csv", index=False)
```

You can find a Jupyter Notebook that contains these commands in this book's GitHub repository at <https://github.com/PacktPublishing/Python-Feature-Engineering-Cookbook-Second-Edition/blob/main/ch02-categorical-encoding/donwload-prepare-store-credit-approval-dataset.ipynb>.

NOTE

Some libraries require that you have **already imputed missing data**, for which you can use any of the recipes from [Chapter 1, Imputing Missing Data](#).

Creating binary variables through one-hot encoding

In one-hot encoding, we represent a categorical variable as a group of binary variables, where each binary variable represents one category. The binary variable takes a value of 1 if the category is present in an observation, or 0 otherwise.

The following table shows the one-hot encoded representation of the **Gender** variable with the categories of **Male** and **Female**:

Gender	Female	Male
Female	1	0
Male	0	1
Male	0	1
Female	1	0
Female	1	0

Figure 2.2 – One-hot encoded representation of the **Gender** variable

As shown in *Figure 2.2*, from the **Gender** variable, we can derive the binary variable of **Female**, which shows the value of **1** for females, or the binary variable of **Male**, which takes the value of **1** for the males in the dataset.

For the categorical variable of **Color** with the values of **red**, **blue**, and **green**, we can create three variables called red, blue, and green. These variables will take the value of **1** if the observation is red, blue, or green, respectively, or 0 otherwise.

A categorical variable with k unique categories can be encoded using $k-1$ binary variables. For **Gender**, k is 2 as it contains two labels (male and

female), so we only need to create one binary variable ($k - 1 = 1$) to capture all of the information. For the **Color** variable, which has three categories ($k=3$; red, blue, and green), we need to create two ($k - 1 = 2$) binary variables to capture all the information so that the following occurs:

- If the observation is red, it will be captured by the **red** variable (red = 1, blue = 0).
- If the observation is blue, it will be captured by the **blue** variable (red = 0, blue = 1)
- If the observation is green, it will be captured by the combination of **red** and **blue** (red = 0, blue = 0)

Encoding into $k-1$ binary variables is well-suited for linear models. There are a few occasions in which we may prefer to encode the categorical variables with k binary variables:

- When training decision trees since they do not evaluate the entire feature space at the same time
- When selecting features recursively
- When determining the importance of each category within a variable

In this recipe, we will compare the one-hot encoding implementations of pandas, scikit-learn, Feature-engine, and Category Encoders.

How to do it...

First, let's make a few imports and get the data ready:

1. Import **pandas** and the **train_test_split** function from scikit-learn:

```
import pandas as pd  
  
from sklearn.model_selection import train_test_split
```

2. Let's load the Credit Approval dataset:

```
data = pd.read_csv("credit_approval_uci.csv")
```

3. Let's separate the data into train and test sets:

```
X_train, X_test, y_train, y_test = train_test_split(  
    data.drop(labels=["target"], axis=1),  
    data["target"],  
    test_size=0.3,  
    random_state=0,  
)
```

4. Let's inspect the unique categories of the **A4** variable:

```
X_train["A4"].unique()
```

We can see the unique values of **A4** in the following output:

```
array(['u', 'y', 'Missing', 'l'], dtype=object)
```

5. Let's encode **A4** into k-1 binary variables using pandas and then inspect the first five rows of the resulting DataFrame:

```
dummies = pd.get_dummies(X_train["A4"],  
    drop_first=True)  
  
dummies.head()
```

NOTE

With pandas **get_dummies()**, we can either ignore or encode missing data through the **dummy_na** parameter. By setting **dummy_na=True**, missing data will be encoded in a new binary variable. To encode the variable into **k** dummies, use **drop_first=False** instead.

Here, we can see the output of *step 5*, where each label is now a binary variable:

```
l u y  
596 0 1 0  
303 0 1 0  
204 0 0 1  
351 0 0 1  
118 0 1 0
```

6. Now, let's encode all of the categorical variables into $k-1$ binaries, capturing the result in a new DataFrame:

```
x_train_enc = pd.get_dummies(X_train, drop_first=True)  
x_test_enc = pd.get_dummies(X_test, drop_first=True)
```

NOTE

The **get_dummies** method from pandas will automatically encode all variables of the object or type. We can encode a subset of the variables by passing the variable names in a list to the **columns** parameter.

7. Let's inspect the first five rows of the binary variables created in *step 6*:

```
x_train_enc.head()
```

NOTE

When encoding more than one variable, `get_dummies()` captures the variable name – say, `A1` – and places an underscore followed by the category name to identify the resulting binary variables.

We can see the binary variables in the following output:

	A2	A3	A8	A11	A14	A15	A1_a	A1_b	A4_l	A4_u	...	A7_j	A7_n	A7_o	A7_y	A7_z	A9_t	A10_t	A12_t	A13_p	A13_s
596	46.08	3.000	2.375	8	396.0	4159	1	0	0	1	...	0	0	0	1	0	1	1	1	0	0
303	15.92	2.875	0.085	0	120.0	0	1	0	0	1	...	0	0	0	1	0	0	0	0	0	0
204	36.33	2.125	0.085	1	50.0	1187	0	1	0	0	...	0	0	0	1	0	1	1	0	0	0
351	22.17	0.585	0.000	0	100.0	0	0	1	0	0	...	0	0	0	0	0	0	0	0	0	0
118	57.83	7.040	14.000	6	360.0	1332	0	1	0	1	...	0	0	0	1	0	1	1	1	0	0

5 rows x 42 columns

Figure 2.3 – Transformed DataFrame showing the dummy variables on the right

NOTE

The `get_dummies()` method will create one binary variable per `seen` category. Hence, if there are more categories in the train set than in the test set, `get_dummies()` will return more columns in the transformed train set than in the transformed test set, and vice versa. To avoid this, it is better to carry out one-hot encoding with scikit-learn or Feature-engine, as we will discuss later in this recipe.

8. Let's concatenate the binary variables to the original dataset:

```
X_test_enc = pd.concat([X_test, X_test_enc], axis=1)
```

9. Now, let's drop the categorical variables from the data:

```
X_test_enc.drop(  
    labels=X_test_enc.select_dtypes(  
        include="O").columns,  
    axis=1,  
    inplace=True,  
)
```

And that's it! Now, we can use our categorical variables to train mathematical models. To inspect the result, use `X_test_enc.head()`.

Now, let's do one-hot encoding using scikit-learn.

10. Import the encoder from scikit-learn:

```
from sklearn.preprocessing import OneHotEncoder
```

11. Let's set up the transformer. By setting `drop` to "**first**", we encode into $k-1$ binary variables, and by setting `sparse` to **False**, the transformer will return a NumPy array (instead of a sparse matrix):

```
encoder = OneHotEncoder(drop="first", sparse=False)
```

TIP

We can encode variables into k dummies by setting the `drop` parameter to **None**. We can also encode into $k-1$ if variables contain two categories and into **k** if variables contain more than two

categories by setting the **drop** parameter to “**if_binary**”. The latter is useful because encoding binary variables into **k** dummies is redundant.

12. First, let's create a list containing the variable names:

```
vars_categorical = X_train.select_dtypes(  
    include="O").columns.to_list()
```

13. Let's fit the encoder to a slice of the train set with the categorical variables:

```
encoder.fit(X_train[vars_categorical])
```

14. Let's inspect the categories for which dummy variables will be created:

```
encoder.categories_
```

We can see the result of the preceding command here:

```
[array(['Missing', 'a', 'b'], dtype=object),
 array(['Missing', 'l', 'u', 'y'], dtype=object),
 array(['Missing', 'g', 'gg', 'p'], dtype=object),
 array(['Missing', 'aa', 'c', 'cc', 'd', 'e', 'ff', 'i', 'j', 'k', 'm',
        'q', 'r', 'w', 'x'], dtype=object),
 array(['Missing', 'bb', 'dd', 'ff', 'h', 'j', 'n', 'o', 'v', 'z'],
       dtype=object),
 array(['f', 't'], dtype=object),
 array(['f', 't'], dtype=object),
 array(['f', 't'], dtype=object),
 array(['g', 'p', 's'], dtype=object)]
```

Figure 2.4 – Arrays with the categories that will be encoded into binary variables, one array per variable

NOTE

Scikit-learn's **OneHotEncoder()** will only encode the categories learned from the train set. If there are new categories in the test set, we can instruct the encoder to ignore them or to return an error by setting the **handle_unknown** parameter to '**ignore**' or '**error**', respectively.

15. Let's create the NumPy arrays with the binary variables for the train and test sets:

```
X_train_enc = encoder.transform(
```

```
x_train[vars_categorical])  
x_test_enc = encoder.transform(  
    x_test[vars_categorical])
```

16. Let's extract the names of the binary variables:

```
encoder.get_feature_names_out()
```

We can see the binary variable names that were returned in the following output:

```
array(['A1_a', 'A1_b', 'A4_l', 'A4_u', 'A4_y', 'A5_g', 'A5_gg', 'A5_p',  
       'A6_aa', 'A6_c', 'A6_cc', 'A6_d', 'A6_e', 'A6_ff', 'A6_i', 'A6_j',  
       'A6_k', 'A6_m', 'A6_q', 'A6_r', 'A6_w', 'A6_x', 'A7_bb', 'A7_dd',  
       'A7_ff', 'A7_h', 'A7_j', 'A7_n', 'A7_o', 'A7_v', 'A7_z', 'A9_t',  
       'A10_t', 'A12_t', 'A13_p', 'A13_s'], dtype=object)
```

Figure 2.5 – Arrays with the names of the one-hot encoded variables

17. Let's convert the array into a pandas DataFrame and add the variable names:

```
x_test_enc = pd.DataFrame(x_test_enc)  
x_test_enc.columns = encoder.get_feature_names_out()
```

18. To concatenate the one-hot encoded data to the original dataset, we need to make their indexes match:

```
x_test_enc.index = x_test.index
```

Now, we are ready to concatenate the one-hot encoded variables to the original data and then remove the categorical variables using *steps 8* and *9* from this recipe.

To follow up, let's perform one-hot encoding with Feature-engine.

19. Let's import the encoder from Feature-engine:

```
from feature_engine.encoding import OneHotEncoder
```

20. Next, let's set up the encoder so that it returns $k-1$ binary variables:

```
ohe_enc = OneHotEncoder(drop_last=True)
```

TIP

Feature-engine automatically finds the categorical variables. To encode only a subset of the variables, we can pass the variable names in a list: **OneHotCategoricalEncoder(variables=["A1", "A4"])**. To encode numerical variables, we can set the **ignore_format** parameter to **True** or cast the variables as the object type. This is useful because sometimes, numerical variables are used to represent categories, such as postcodes.

21. Let's fit the encoder to the train set so that it learns the categories and variables to encode:

```
ohe_enc.fit(X_train)
```

22. Let's explore the variables that will be encoded:

```
ohe_enc.variables_
```

The transformer found and stored the variables of the object or categorical type, as shown in the following output:

```
[ 'A1', 'A4', 'A5', 'A6', 'A7', 'A9', 'A10', 'A12', 'A13' ]
```

NOTE

Feature-engine's **OneHotEncoder** has the option to encode most variables into k dummies, while only returning k-1 dummies for binary variables. For this behavior, set the **drop_last_binary** parameter to **True**.

23. Let's explore the categories for which dummy variables will be created:

```
ohe_enc.encoder_dict_
```

The following dictionary contains the categories that will be encoded in each variable:

```
{'A1': ['a', 'b'],
'A4': ['u', 'y', 'Missing'],
'A5': ['g', 'p', 'Missing'],
'A6': ['c', 'q', 'w', 'ff', 'm', 'i', 'e', 'cc', 'x',
'd', 'k', 'j', 'Missing', 'aa'],
'A7': ['v', 'ff', 'h', 'dd', 'z', 'bb', 'j', 'Missing',
'n'],
'A9': ['t'],
'A10': ['t'],
'A12': ['t'],
'A13': ['g', 's']}
```

24. Let's encode the categorical variables in train and test sets:

```
x_train_enc = ohe_enc.transform(X_train)  
x_test_enc = ohe_enc.transform(X_test)
```

TIP

Feature-engine's **OneHotEncoder()** returns a copy of the original dataset plus the binary variables and without the original categorical variables. Thus, this data is ready to train machine learning models.

If we execute **X_train_enc.head()**, we will see the following DataFrame:

	A2	A3	A8	A11	A14	A15	A1_a	A1_b	A4_u	A4_y	...	A7_z	A7_bb	A7_j	A7_Missing	A7_n	A9_t	A10_t	A12_t	A13_g	A13_s
596	46.08	3.000	2.375	8	396.0	4159	1	0	1	0	...	0	0	0	0	0	1	1	1	1	0
303	15.92	2.875	0.085	0	120.0	0	1	0	1	0	...	0	0	0	0	0	0	0	0	1	0
204	36.33	2.125	0.085	1	50.0	1187	0	1	0	1	...	0	0	0	0	0	1	1	0	1	0
351	22.17	0.585	0.000	0	100.0	0	0	1	0	1	...	0	0	0	0	0	0	0	0	1	0
118	57.83	7.040	14.000	6	360.0	1332	0	1	1	0	...	0	0	0	0	0	1	1	1	1	0

Figure 2.6 – Transformed DataFrame with the one-hot encoded variables on the right

Note how the **A4** categorical variable was replaced with **A4_u**, **A4_y**, and so on.

NOTE

We can get the names of all the variables in the transformed dataset by executing **ohe_enc.get_feature_names_out()**.

How it works...

In this recipe, we performed a one-hot encoding of categorical variables using pandas, scikit-learn, Feature-engine, and Category Encoders.

With `get_dummies()` from pandas, we automatically created binary variables for each of the categories in the categorical variables.

The `OneHotEncoder` transformers from the scikit-learn and Feature-engine libraries share the `fit()` and `transform()` methods. With `fit()`, the encoders learned the categories for which the dummy variables should be created. With `transform()`, they returned the binary variables either in a NumPy array or added them to the original DataFrame.

TIP

One-hot encoding expands the feature space. From nine original categorical variables, we created 36 binary ones. If our datasets contain many categorical variables or highly cardinal variables, we will easily increase the feature space dramatically, which increases the computational cost of training machine learning models or obtaining their predictions and may also deteriorate their performance.

There's more...

We can also perform one-hot encoding using `OneHotEncoder` from the Category Encoders library.

`OneHotEncoder()` from Feature-engine and Category Encoders can automatically identify and encode categorical variables – that is, those of the object or categorical type. So does pandas `get_dummies()`. Scikit-

learn's **OneHotEncoder()**, on the other hand, will encode all variables in the dataset.

With pandas, Feature-engine, and Category Encoders, we can only encode a subset of the variables, indicating their names in a list. With scikit-learn, we need to use an additional class, **ColumnTransformer()**, to slice the data before the transformation.

With Feature-engine and Category Encoders, the dummy variables are added to the original dataset and the categorical variables are removed after the encoding. With scikit-learn and pandas, we need to manually perform these procedures.

Finally, using **OneHotEncoder()** from scikit-learn, Feature-engine, and Category Encoders, we can perform the encoding step within a scikit-learn pipeline, which is more convenient if we have various feature engineering steps or want to put the pipelines into production. pandas **get_dummies()** is otherwise well suited for data analysis and visualization.

Performing one-hot encoding of frequent categories

One-hot encoding represents each variable's category with a binary variable. Hence, one-hot encoding of highly cardinal variables or datasets with multiple categorical features can expand the feature space dramatically. This, in turn, may increase the computational cost of using machine learning models or deteriorate their performance. To reduce the number of binary variables, we can perform one-hot encoding of the most

frequent categories. One-hot encoding the top categories is equivalent to treating the remaining, less frequent categories as a single, unique category.

In this recipe, we will implement one-hot encoding of the most popular categories using pandas and Feature-engine.

How to do it...

First, let's import the necessary Python libraries and get the dataset ready:

1. Import the required Python libraries, functions, and classes:

```
import pandas as pd  
  
import numpy as np  
  
from sklearn.model_selection import train_test_split  
  
from feature_engine.encoding import OneHotEncoder
```

2. Let's load the dataset and divide it into train and test sets:

```
data = pd.read_csv("credit_approval_uci.csv")  
  
X_train, X_test, y_train, y_test = train_test_split(  
  
    data.drop(  
  
        labels=["target"], axis=1),  
  
    data["target"],  
  
    test_size=0.3,  
  
    random_state=0,  
  
)
```

TIP

The most frequent categories need to be determined in the train set. This is to avoid data leakage.

3. Let's inspect the unique categories of the **A6** variable:

```
x_train["A6"].unique()
```

The unique values of **A6** are displayed in the following output:

```
array(['c', 'q', 'w', 'ff', 'm', 'i', 'e', 'cc', 'x', 'd',
'k', 'j', 'Missing', 'aa', 'r'], dtype=object)
```

4. Let's count the number of observations per category of **A6**, sort them in decreasing order, and then display the five most frequent categories:

```
x_train["A6"].value_counts().sort_values(
    ascending=False).head(5)
```

We can see the five most frequent categories and the number of observations per category in the following output:

```
c      93 q      56 w      48 i      41 ff      38
Name: A6, dtype: int64
```

5. Now, let's capture the most frequent categories of **A6** in a list by using the code in *step 4* inside a list comprehension:

```
top_5 = [
    x for x in
x_train["A6"].value_counts().sort_values(
    ascending=False).head(5).index
]
```

6. Now, let's add a binary variable per top category to the train and test sets:

```
for label in top_5:  
    X_train[f"A6_{label}"] = np.where(  
        X_train["A6"] == label, 1, 0)  
  
    X_test[f"A6_{label}"] = np.where(  
        X_test["A6"] == label, 1, 0)
```

7. Let's display the top **10** rows of the original and encoded variable, **A6**, in the train set:

```
X_train[["A6"] + [f"A6_{label}" for label in  
top_5]].head(10)
```

In the output of *step 7*, we can see the **A6** variable, followed by the binary variables:

	A6	A6_c	A6_q	A6_w	A6_i	A6_ff
596	c	1	0	0	0	0
303	q	0	1	0	0	0
204	w	0	0	1	0	0
351	ff	0	0	0	0	1
118	m	0	0	0	0	0
247	q	0	1	0	0	0
652	i	0	0	0	1	0
513	e	0	0	0	0	0
230	cc	0	0	0	0	0
250	e	0	0	0	0	0

We can automate one-hot encoding of frequent categories with Feature-engine. First, let's load and divide the dataset, as we did in *step 2*.

8. Let's set up the one-hot encoder to encode the five most frequent categories of the **A6** and **A7** variables:

```
ohe_enc = OneHotEncoder(  
    top_categories=5,  
    variables=["A6", "A7"]  
)
```

TIP

Feature-engine's **OneHotEncoder()** will encode all categorical variables in the dataset by default unless we specify the variables to encode, as we did in *step 8*.

9. Let's fit the encoder to the train set so that it learns and stores the most frequent categories of **A6** and **A7**:

```
ohe_enc.fit(X_train)
```

NOTE

The number of frequent categories to encode is arbitrarily determined by the user.

10. Finally, let's encode **A6** and **A7** in the train and test sets:

```
X_train_enc = ohe_enc.transform(X_train)  
X_test_enc = ohe_enc.transform(X_test)
```

You can view the new binary variables in the DataFrame by executing `X_train_enc.head()`. You can also find the top five categories learned by the encoder by executing `ohe_enc.encoder_dict_`.

NOTE

Feature-engine replaces the original variable with the binary ones returned by one-hot encoding, leaving the dataset ready to use in machine learning.

How it works...

In this recipe, we performed one-hot encoding of the five most popular categories using NumPy and Feature-engine.

In the first part of this recipe, we worked with the **A6** categorical variable. We inspected its unique categories with pandas `unique()`. Next, we counted the number of observations per category using pandas `value_counts()`, which returned a pandas series with the categories as the index and the number of observations as values. Next, we sorted the categories from the one with the most to the one with the least observations using pandas `sort_values()`. Next, we reduced the series to the five most popular categories by using pandas `head()`. Then, we used this series in a list comprehension to capture the name of the most frequent categories. After that, we looped over each category, and with NumPy's `where()` method, we created binary variables by placing a value of **1** if the observation showed the category, or **0** otherwise.

To perform a one-hot encoding of the five most popular categories of the **A6** and **A7** variables with Feature-engine, we used `OneHotEncoder()`,

indicating **5** in the **top_categories** argument, and passing the variable names in a list to the **variables** argument. With **fit()**, the encoder learned the top categories from the train set and stored them in its **encoder_dict_** attribute. Then, with **transform()**, **OneHotEncoder()** replaced the original variables with the set of binary ones.

There's more...

This recipe is based on the winning solution of the KDD 2009 cup, *Winning the KDD Cup Orange Challenge with Ensemble Selection* (<http://proceedings.mlr.press/v7/niculescu09/niculescu09.pdf>), where the authors limited one-hot encoding to the 10 most frequent categories of each variable.

Replacing categories with counts or the frequency of observations

In count or frequency encoding, we replace the categories with the count or the fraction of observations showing that category. That is, if 10 out of 100 observations show the category **blue** for the **Color** variable, we would replace **blue** with 10 when doing count encoding, or with 0.1 if performing frequency encoding. These encoding methods, which capture the representation of each label in a dataset, are very popular in data science competitions. The assumption is that the number of observations per category is somewhat predictive of the target.

TIP

Note that if two different categories are present in the same number of observations, they will be replaced by the same value, which leads to information loss.

In this recipe, we will perform count and frequency encoding using pandas, Feature-engine, and Category Encoders.

How to do it...

Let's begin by making some imports and preparing the data:

1. Import **pandas** and the required function:

```
import pandas as pd  
  
from sklearn.model_selection import train_test_split
```

2. Let's load the dataset and divide it into train and test sets:

```
data = pd.read_csv("credit_approval_uci.csv")  
  
X_train, X_test, y_train, y_test = train_test_split(  
    data.drop(labels=["target"], axis=1),  
    data["target"],  
    test_size=0.3,  
    random_state=0,  
)
```

3. Let's count the number of observations per category of the **A7** variable and capture it in a dictionary:

```
counts = X_train["A7"].value_counts().to_dict()
```

TIP

To encode categories with their frequency, execute

```
X_train["A6"].value_counts(normalize=True).to_dict().
```

If we execute **print(counts)**, we can observe the count of observations per category:

```
{'v': 277, 'h': 101, 'ff': 41, 'bb': 39, 'z': 7, 'dd': 5,  
'j': 5, 'Missing': 4, 'n': 3, 'o': 1}
```

4. Let's replace the categories in **A7** with the counts:

```
X_train["A7"] = X_train["A7"].map(counts)  
X_test["A7"] = X_test["A7"].map(counts)
```

Go ahead and inspect the data by executing **X_train.head()** to corroborate that the categories have been replaced by the counts.

Now, let's carry out count encoding using Feature-engine. First, let's load and divide the dataset, as we did in *step 2*.

5. Let's import the count encoder from Feature-engine:

```
from feature_engine.encoding import  
CountFrequencyEncoder
```

6. Let's set up the encoder so that it encodes all categorical variables with the count of observations:

```
count_enc = CountFrequencyEncoder(  
    encoding_method="count", variables=None,  
)
```

TIP

CountFrequencyEncoder() will automatically find and encode all categorical variables in the train set. To encode only a subset of the variables, we can pass the variable names in a list to the **variables** argument.

7. Let's fit the encoder to the train set so that it stores the number of observations per category per variable:

```
count_enc.fit(X_train)
```

TIP

The dictionaries with the category-to-counts pairs are stored in the **encoder_dict_** attribute and can be displayed by executing **count_enc.encoder_dict_**.

8. Finally, let's replace the categories with counts in the train and test sets:

```
X_train_enc = count_enc.transform(X_train)  
X_test_enc = count_enc.transform(X_test)
```

TIP

If there are categories in the test set that were not present in the train set, the transformer will replace those with **np.nan** and return a warning to make you aware of this. A good idea to prevent this behavior is to group infrequent labels, as described in the *Grouping rare or infrequent categories* recipe.

The encoder returns pandas DataFrames with the strings of the categorical variables replaced with the counts of observations, leaving the variables ready to use in machine learning models.

To wrap up this recipe, let's encode the variables using Category Encoders.

9. Let's import the encoder from Category Encoders:

```
from category_encoders.count import CountEncoder
```

10. Let's set up the encoder so that it encodes all categorical variables with the count of observations:

```
count_enc = CountEncoder(cols=None)
```

NOTE

CountEncoder() automatically finds and encodes *all* categorical variables in the train set. To encode only a subset of the categorical variables, we can pass the variable names in a list to the **cols** argument. To replace the categories by frequency instead, we need to set the **Normalize** parameter to **True**.

11. Let's fit the encoder to the train set so that it counts and stores the number of observations per category per variable:

```
count_enc.fit(X_train)
```

TIP

The values used to replace the categories are stored in the mapping attribute and can be displayed by executing **count_enc.mapping**.

12. Finally, let's replace the categories with counts in the train and test sets:

```
X_train_enc = count_enc.transform(X_train)
```

```
X_test_enc = count_enc.transform(X_test)
```

NOTE

Categories present in the test set that were not seen in the train set are referred to as unknown categories. **CountEncoder()** has

different options to handle unknown categories, including returning an error, treating them as missing data, or replacing them with an indicated integer. **CountEncoder()** can also automatically group categories with few observations.

The encoder returns pandas DataFrames with the strings of the categorical variables replaced with the counts of observations, leaving the variables ready to use in machine learning models.

How it works...

In this recipe, we replaced categories by the count of observations using pandas, Feature-engine, and Category Encoders.

Using pandas **value_counts()**, we determined the number of observations per category of the **A7** variable, and with pandas **to_dict()**, we captured these values in a dictionary, where each key was a unique category, and each value the number of observations for that category. With pandas **map()** and using this dictionary, we replaced the categories with the observation counts in both the train and test sets.

To perform count encoding with Feature-engine, we used **CountFrequencyEncoder()** and set **encoding_method** to '**count**'. We left the **variables** argument set to **None** so that the encoder automatically finds all of the categorical variables in the dataset. With the **fit()** method, the transformer found the categorical variables and stored the observation counts per category in the **encoder_dict_** attribute. With the **transform()** method, the transformer replaced the categories with the counts, returning a pandas DataFrame.

Finally, we performed count encoding with **CountEncoder()** by setting **Normalize** to **False**. We left the **cols** argument set to **None** so that the encoder automatically finds the categorical variables in the dataset. With the **fit()** method, the transformer found the categorical variables and stored the category to count mappings in the **mapping** attribute. With the **transform()** method, the transformer replaced the categories with the counts in, returning a pandas DataFrame.

Replacing categories with ordinal numbers

Ordinal encoding consists of replacing the categories with digits from 1 to k (or 0 to $k-1$, depending on the implementation), where k is the number of distinct categories of the variable. The numbers are assigned arbitrarily. Ordinal encoding is better suited for non-linear machine learning models, which can navigate through the arbitrarily assigned digits to find patterns that relate to the target.

In this recipe, we will perform ordinal encoding using pandas, scikit-learn, and Feature-engine.

How to do it...

First, let's import the necessary Python libraries and prepare the dataset:

1. Import **pandas** and the **data split** function:

```
import pandas as pd  
from sklearn.model_selection import train_test_split
```

2. Let's load the dataset and divide it into train and test sets:

```
data = pd.read_csv("credit_approval_uci.csv")
X_train, X_test, y_train, y_test = train_test_split(
    data.drop(labels=["target"], axis=1),
    data["target"],
    test_size=0.3,
    random_state=0,
)
```

3. To encode the **A7** variable, let's make a dictionary of category-to-integer pairs:

```
ordinal_mapping = {k: i for i, k in enumerate(
    X_train["A7"].unique(), 0)
}
```

If we execute **print(ordinal_mapping)**, we will see the digits that will replace each category:

```
{'v': 0, 'ff': 1, 'h': 2, 'dd': 3, 'z': 4, 'bb': 5, 'j': 6, 'Missing': 7, 'n': 8, 'o': 9}
```

4. Now, let's replace the categories with numbers in the original variables:

```
X_train["A7"] = X_train["A7"].map(ordinal_mapping)
X_test["A7"] = X_test["A7"].map(ordinal_mapping)
```

With **print(X_train["A7"].head(10))**, we can see the result of the preceding operation, where the original categories were replaced by

numbers:

596 0

303 0

204 0

351 1

118 0

247 2

652 0

513 3

230 0

250 4

Name: A7, dtype: int64

Next, let's carry out ordinal encoding using scikit-learn. First, we need to divide the data into train and test sets, as we did in *step 2*.

5. Let's import the required classes:

```
from sklearn.preprocessing import OrdinalEncoder  
from sklearn.compose import ColumnTransformer
```

TIP

Do not confuse **OrdinalEncoder()** with **LabelEncoder()** from scikit-learn. The former is intended to encode predictive features, whereas the latter is intended to modify the target variable.

6. Let's set up the encoder:

```
enc = OrdinalEncoder()
```

NOTE

Scikit-learn's **OrdinalEncoder()** will encode the entire dataset. To encode only a selection of variables, we need to use scikit-learn's **ColumnTransformer()**.

7. Let's make a list containing the categorical variables to encode:

```
vars_categorical = X_train.select_dtypes(  
    include="O").columns.to_list()
```

8. Let's make a list containing the remaining variables:

```
vars_remainder = X_train.select_dtypes(  
    exclude="O").columns.to_list()
```

9. Now, let's set up **ColumnTransformer()** to encode the categorical variables. By setting the **remainder** parameter to "**passthrough**", we make **ColumnTransformer()** concatenate the variables that are not encoded at the back of the encoded features:

```
ct = ColumnTransformer(  
    [("encoder", enc, vars_categorical)],  
    remainder="passthrough",  
)
```

10. Let's fit the encoder to the train set so that it creates and stores representations of categories to digits:

```
ct.fit(X_train)
```

By executing `ct.named_transformers_["encoder"].categories_`, you can visualize the unique categories per variable.

11. Now, let's encode the categorical variables in the train and test sets:

```
X_train_enc = ct.transform(X_train)  
X_test_enc = ct.transform(X_test)
```

Remember that scikit-learn returns a NumPy array.

12. Let's transform the arrays into pandas DataFrames by adding the columns:

```
X_train_enc = pd.DataFrame(  
    X_train_enc,  
    columns=vars_categorical+vars_remainder)  
X_test_enc = pd.DataFrame(  
    X_test_enc,  
    columns=vars_categorical+vars_remainder)
```

NOTE

Note that, with `ColumnTransformer()`, the variables that were not encoded will be returned to the right of the DataFrame, following the encoded variables. You can visualize the output of *step 12* with `X_train_enc.head()`.

Now, let's do ordinal encoding with Feature-engine. First, we must divide the dataset, as we did in *step 2*.

13. Let's import the encoder:

```
from feature_engine.encoding import OrdinalEncoder
```

14. Let's set up the encoder so that it replaces categories with arbitrary integers in the categorical variables specified in *step 7*:

```
enc = OrdinalEncoder(encoding_method="arbitrary",
variables=vars_categorical)
```

NOTE

Feature-engine's **OrdinalEncoder** automatically finds and encodes all categorical variables if the **variables** parameter is left set to **None**. Alternatively, it will encode the variables indicated in the list. In addition, Feature-engine's **OrdinalEncoder()** can assign the integers according to the target mean value (see the *Performing ordinal encoding based on the target value* recipe).

15. Let's fit the encoder to the train set so that it learns and stores the category-to-integer mappings:

```
enc.fit(X_train)
```

TIP

The category to integer mappings are stored in the **encoder_dict_** attribute and can be accessed by executing **enc.encoder_dict_**.

16. Finally, let's encode the categorical variables in the train and test sets:

```
X_train_enc = enc.transform(X_train)
X_test_enc = enc.transform(X_test)
```

Feature-engine returns pandas DataFrames where the values of the original variables are replaced with numbers, leaving the DataFrame ready to use in machine learning models.

How it works...

In this recipe, we replaced categories with integers assigned arbitrarily.

With pandas **unique()**, we returned the unique values of the **A7** variable, and using Python's list comprehension syntax, we created a dictionary of key-value pairs, where each key was one of the **A7** variable's unique categories, and each value was the digit that would replace the category.

Finally, we used pandas **map()** to replace the strings in **A7** with the integers.

Next, we carried out ordinal encoding using scikit-learn's **OrdinalEncoder()** and used **ColumnTransformer()** to select the columns to encode. With the **fit()** method, the transformer created the category-to-integer mappings based on the categories in the train set. With the **transform()** method, the categories were replaced with integers, returning a NumPy array. **ColumnTransformer()** sliced the DataFrame into the categorical variables to encode, and then concatenated the remaining variables at the right of the encoded features.

To perform ordinal encoding with Feature-engine, we used **OrdinalEncoder()**, indicating that the integers should be assigned arbitrarily in **encoding_method** and passing a list with the variables to encode in the **variables** argument. With the **fit()** method, the encoder assigned integers to each variable's categories, which were stored in the **encoder_dict_** attribute. These mappings were then used by the **transform()** method to replace the categories in the train and test sets, returning DataFrames.

There's more...

You can also carry out ordinal encoding with **OrdinalEncoder()** from Category Encoders.

The transformers from Feature-engine and Category Encoders can automatically identify and encode categorical variables – that is, those of the object or categorical type. They also allow us to encode only a subset of the variables.

scikit-learn's transformer will otherwise encode all variables in the dataset. To encode just a subset, we need to use an additional class, **ColumnTransformer()**, to slice the data before the transformation.

Feature-engine and Category Encoders return pandas DataFrames, whereas scikit-learn returns NumPy arrays.

Finally, each class has additional functionality. For example, with scikit-learn, we can encode only a subset of the categories, whereas Feature-engine allows us to replace categories with integers that are assigned based on the target mean value. On the other hand, Category Encoders can automatically handle missing data and offers alternative options to work with unseen categories.

Performing ordinal encoding based on the target value

In the previous recipe, we replaced categories with integers, which were assigned arbitrarily. We can also assign integers to the categories given the target values. To do this, first, we must calculate the mean value of the target per category. Next, we must order the categories from the one with the lowest to the one with the highest target mean value. Finally, we must

assign digits to the ordered categories, starting with 0 to the first category up to $k-1$ to the last category, where k is the number of distinct categories.

This encoding method creates a monotonic relationship between the categorical variable and the response and therefore makes the variables more adequate for use in linear models.

In this recipe, we will encode categories while following the target value using pandas and Feature-engine.

How to do it...

First, let's import the necessary Python libraries and get the dataset ready:

1. Import the required Python libraries, functions, and classes:

```
import pandas as pd  
  
import matplotlib.pyplot as plt  
  
from sklearn.model_selection import train_test_split
```

2. Let's load the dataset and divide it into train and test sets:

```
data = pd.read_csv("credit_approval_uci.csv")  
  
X_train, X_test, y_train, y_test = train_test_split(  
    data.drop(labels=["target"], axis=1),  
    data["target"],  
    test_size=0.3,  
    random_state=0,  
)
```

3. Let's determine the mean target value per category in **A7**, then sort the categories from that with the lowest to that with the highest target value:

```
y_train.groupby(x_train["A7"]).mean().sort_values()
```

The following is the output of the preceding command:

```
A7
0      0.000000
ff     0.146341
j      0.200000
dd     0.400000
v      0.418773
bb     0.512821
h      0.603960
n      0.666667
z      0.714286
Missing 1.000000
Name: target, dtype: float64
```

4. Now, let's repeat the computation in *step 3*, but this time, let's retain the ordered category names:

```
ordered_labels = y_train.groupby(
    x_train["A7"]).mean().sort_values().index
```

To display the output of the preceding command, we can execute
print(ordered_labels):

```
Index(['o', 'ff', 'j', 'dd', 'v', 'bb', 'h', 'n', 'z',
'Missing'], dtype='object', name='A7')
```

5. Let's create a dictionary of category-to-integer pairs, using the ordered list we created in *step 4*:

```
ordinal_mapping = {

    k: i for i, k in enumerate(
        ordered_labels, 0)

}
```

We can visualize the result of the preceding code by executing

```
print(ordinal_mapping):
```

```
{'o': 0, 'ff': 1, 'j': 2, 'dd': 3, 'v': 4, 'bb': 5, 'h':
6, 'n': 7, 'z': 8, 'Missing': 9}
```

6. Let's use the dictionary we created in *step 5* to replace the categories in **A7** in the train and test sets, returning the encoded features as new columns:

```
X_train["A7_enc"] = X_train["A7"].map(ordinal_mapping)
X_test["A7_enc"] = X_test["A7"].map(ordinal_mapping)
```

TIP

Note that if the test set contains a category not present in the train set, the preceding code will introduce `np.nan`.

To better understand the monotonic relationship concept, let's plot the relationship of the categories of the **A7** variable with the target before and after the encoding.

7. Let's plot the mean target response per category of the **A7** variable:

```
y_train.groupby(X_train["A7"]).mean().plot()  
plt.title("Relationship between A7 and the target")  
plt.ylabel("Mean of target")  
plt.show()
```

We can see the non-monotonic relationship between categories of **A7** and the target in the following plot:

Relationship between A7 and the target

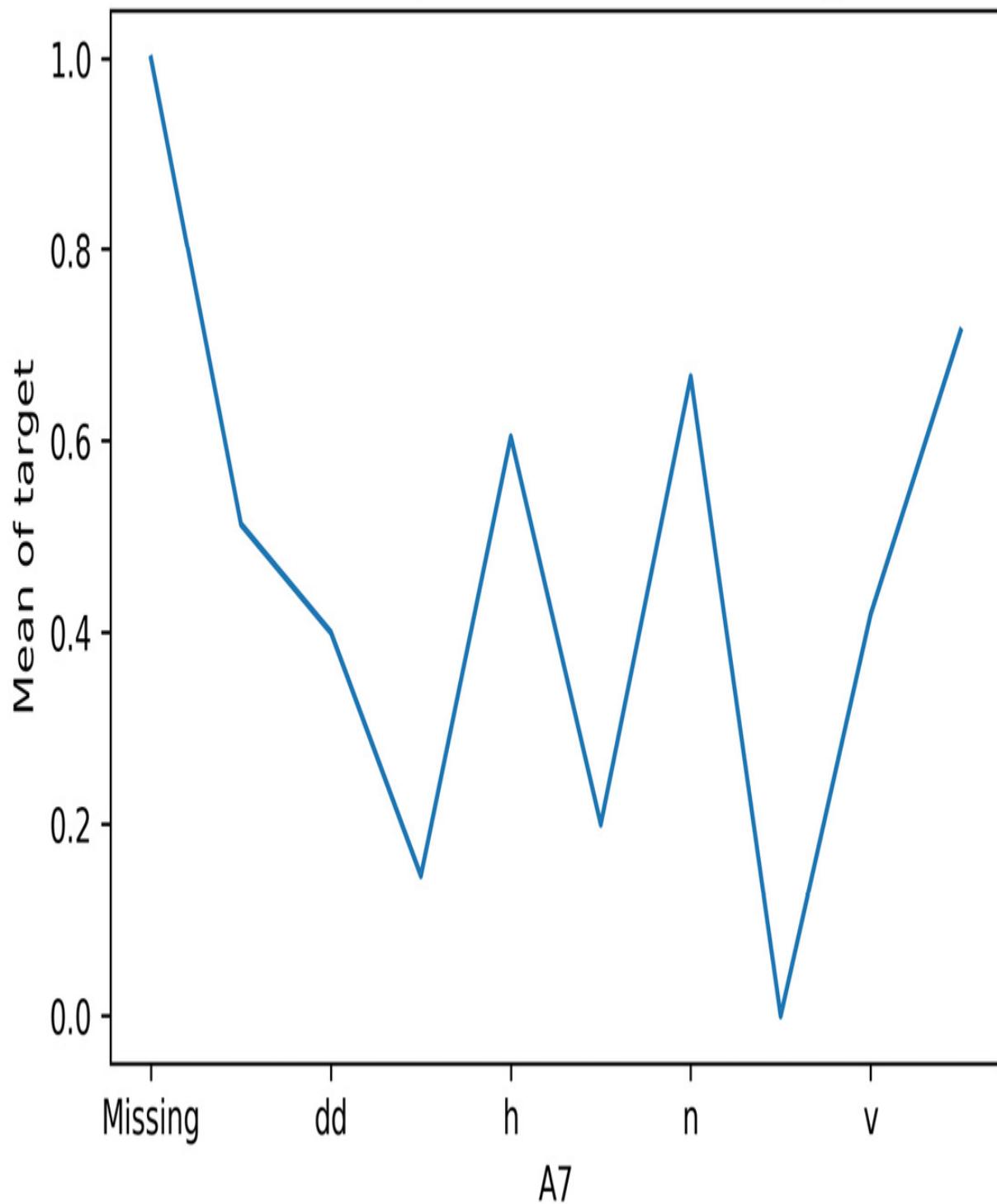


Figure 2.7 – Relationship between the categories of A7 and the target

8. Let's plot the mean target value per category in the encoded variable:

```
y_train.groupby(X_train["A7_enc"]).mean().plot()  
plt.title("Relationship between A7 and the target")  
plt.ylabel("Mean of target")  
plt.show()
```

The encoded variable shows a monotonic relationship with the target – the higher the mean target value, the higher the digit assigned to the category:

Relationship between A7 after encoding and the target

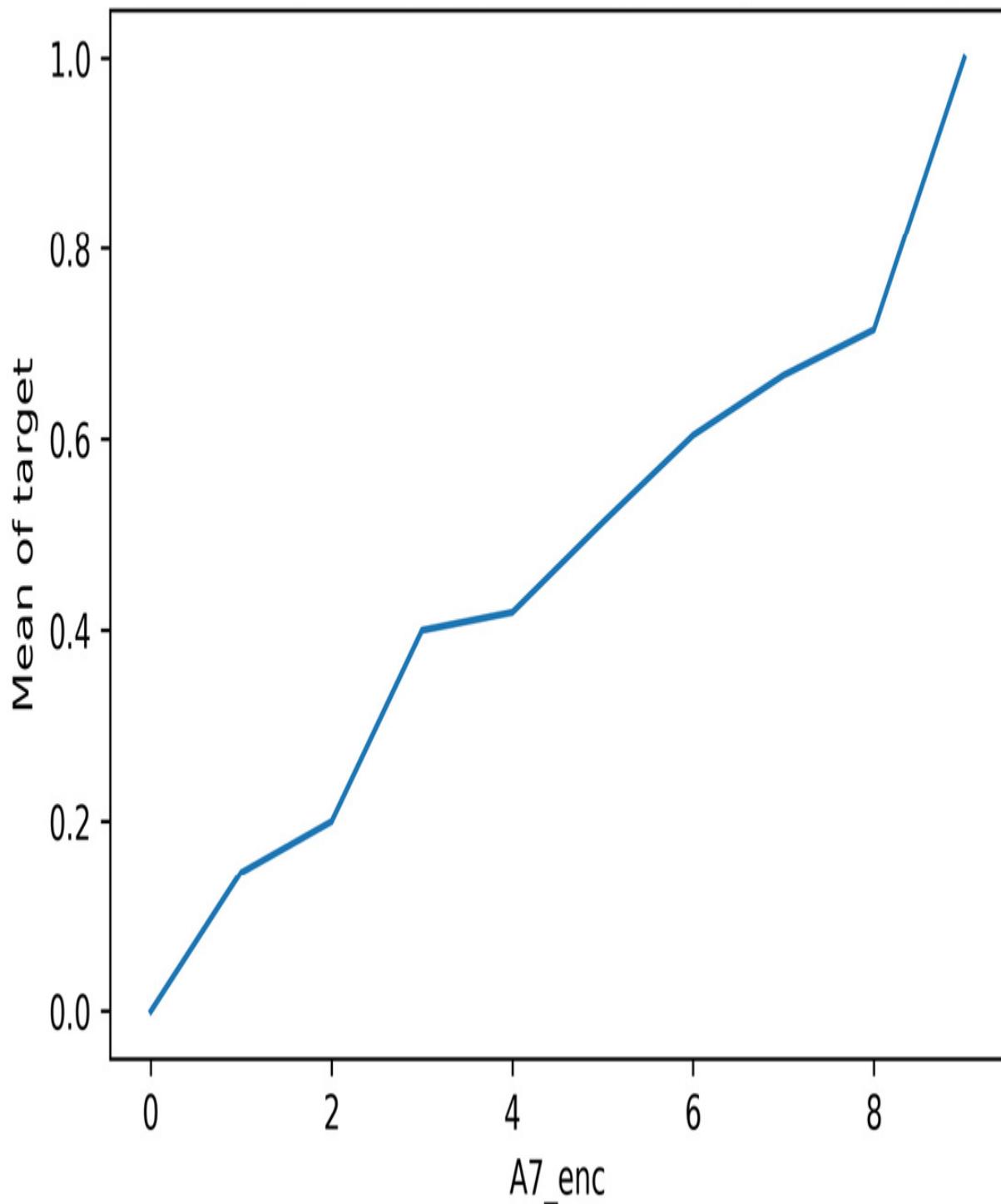


Figure 2.8 – Relationship between A7 and the target after the encoding

Now, let's perform ordered ordinal encoding using Feature-engine. First, we must divide the dataset into train and test sets, as we did in *step 2*.

9. Let's import the encoder:

```
from feature_engine.encoding import OrdinalEncoder
```

10. Next, let's set up the encoder so that it assigns integers by following the target value to all categorical variables in the dataset:

```
ordinal_enc = OrdinalEncoder(  
    encoding_method="ordered",  
    variables=None)
```

TIP

OrdinalEncoder() will find and encode all categorical variables automatically. Alternatively, we can indicate which variables to encode by passing their names in a list to the `variables` argument.

11. Let's fit the encoder to the train set so that it finds the categorical variables, and then stores the category and integer mappings:

```
ordinal_enc.fit(X_train, y_train)
```

TIP

When fitting the encoder, we need to pass the train set and the target, like with many scikit-learn predictor classes.

12. Finally, let's replace the categories with numbers in the train and test sets:

```
X_train_enc = ordinal_enc.transform(X_train)
```

```
X_test_enc = ordinal_enc.transform(X_test)
```

TIP

A list of the categorical variables is stored in the `variables_` attribute of `OrdinalEncoder()` and the dictionaries with the category-to-integer mappings in the `encoder_dict_` attribute. When fitting the encoder, we need to pass the train set and the target, like with many scikit-learn predictor classes.

Go ahead and check the monotonic relationship between other encoded categorical variables and the target by using the code in step 7 and changing the variable name in the `groupby()` method.

How it works...

In this recipe, we replaced the categories with integers according to the target mean.

In the first part of this recipe, we worked with the `A7` categorical variable. With pandas `groupby()`, we grouped the data based on the categories of `A7`, and with pandas `mean()`, we determined the mean value of the target for each of the categories of `A7`. Next, we ordered the categories with pandas `sort_values()` from the ones with the lowest to the ones with the highest target mean response. The output of this operation was a pandas Series, with the categories as indices and the target mean as values. With pandas `index`, we captured the ordered categories in an array; then, with Python dictionary comprehension, we created a dictionary of category-to-integer pairs. Finally, we used this dictionary to replace the category with integers using pandas `map()` in the train and test sets.

Then, we plotted the relationship of the original and encoded variables with the target to visualize the monotonic relationship after the transformation.

We determined the mean target value per category of **A7** using pandas **groupby()**, followed by pandas **mean()**, as described in the preceding paragraph. We followed up with pandas **plot()** to create a plot of category versus target mean value. We added a title and *y* labels with Matplotlib's **title()** and **ylabel()** methods.

To perform the encoding with Feature-engine, we used **OrdinalEncoder()** and indicated "**ordered**" in the **encoding_method** argument. We left the argument variables set to **None** so that the encoder automatically detects all categorical variables in the dataset. With the **fit()** method, the encoder found the categorical variables to encode and assigned digits to their categories, according to the target mean value. The variables to encode and dictionaries with category-to-digit pairs were stored in the **variables_** and **encoder_dict_** attributes, respectively. Finally, using the **transform()** method, the transformer replaced the categories with digits in the train and test sets, returning pandas DataFrames.

See also

For an implementation of this recipe with Category Encoders, visit this book's GitHub repository.

Implementing target mean encoding

Mean encoding or target encoding maps each category to the probability estimate of the target attribute. If the target is binary, the numerical mapping

is the posterior probability of the target conditioned to the value of the category. If the target is continuous, the numerical representation is given by the expected value of the target given the value of the category.

In its simplest form, the numerical representation for each category is given by the mean value of the target variable for a particular category group. For example, if we have a **City** variable, with the categories of **London**, **Manchester**, and **Bristol**, and we want to predict the default rate (the target takes values 0 and 1); if the default rate for **London** is 30%, we replace **London** with 0.3; if the default rate for **Manchester** is 20%, we replace **Manchester** with 0.2; and so on. If the target is continuous – say we want to predict income – then we would replace London, Manchester, and Bristol with the mean income earned in each city.

In mathematical terms, if the target is binary, the replacement value, S , is determined like so:

$$S_i = n_i(y=1)/n_i$$

Here, the numerator is the number of observations with a target value of 1 for category i and the denominator is the number of observations with a category value of i .

If the target is continuous, S , this is determined by the following formula:

$$S_i = \frac{\sum y_i}{n_i}$$

Here, the numerator is the sum of the target across observations in category i and n_i is the total number of observations in category i .

These formulas provide a good approximation of the target estimate if there is a sufficiently large number of observations with each category value – in other words, if n_i is large. However, in most datasets, categorical variables will only have categorical values present in a few observations. In these cases, target estimates derived from the precedent formulas can be unreliable.

To mitigate poor estimates returned for rare categories, the target estimates can be determined as a mixture of two probabilities: those returned by the preceding formulas and the prior probability of the target based on the entire training set. The two probabilities are *blended* using a weighting factor, which is a function of the category group size:

$$S_i = \lambda \frac{n_{i(Y=1)}}{n_i} + (1 - \lambda) \frac{n_\lambda}{N}$$

In this formula, n_y is the total number of cases where the target takes a value of 1, N is the size of the train set, and λ is the weighting factor.

When the category group is large, λ approximates 1, so more weight is given to the first term of the equation. When the category group size is small, then λ tends to 0, so the estimate is mostly driven by the second term of the equation – that is, the target’s prior probability. In other words, if the group size is small, knowing the value of the category does not tell us anything about the value of the target.

The weighting factor, λ , is a function of the group size, k , and a smoothing parameter, f , controls the rate of transition between the first and second term of the preceding equation:

$$\lambda = \frac{1}{1 + e^{-(n-k)/f}}$$

Here, k is half of the minimal size for which we *fully trust* the first term of the equation. The f parameter is selected by the user either arbitrarily or with optimization.

TIP

Mean encoding was designed to encode highly cardinal categorical variables without expanding the feature space. For more details, check out the following article: Micci-Barreca D. *A Preprocessing Scheme for High-Cardinality Categorical Attributes in Classification and Prediction Problems*. ACM SIGKDD Explorations Newsletter, 2001.

In this recipe, we will perform mean encoding using pandas, Feature-engine, and Category Encoders.

How to do it...

In the first part of this recipe, we will replace categories with the target mean value, regardless of the number of observations per category. We will use pandas and Feature-engine to do this. In the second part of this recipe,

we will introduce the weighting factor using Category Encoders. Let's begin with this recipe:

1. Import **pandas** and the data split function:

```
import pandas as pd  
  
from sklearn.model_selection import train_test_split
```

2. Let's load the dataset and divide it into train and test sets:

```
data = pd.read_csv("credit_approval_uci.csv")  
  
X_train, X_test, y_train, y_test = train_test_split(  
    data.drop(labels=["target"], axis=1),  
    data["target"],  
    test_size=0.3,  
    random_state=0,  
)
```

3. Let's determine the mean target value per category of the **A7** variable and then store them in a dictionary:

```
mapping =  
y_train.groupby(X_train["A7"]).mean().to_dict()
```

We can display the content of the dictionary by executing

```
print(mapping):
```

```
{'Missing': 1.0,  
'bb': 0.5128205128205128,  
'dd': 0.4,
```

```
'ff': 0.14634146341463414,  
'h': 0.6039603960396039,  
'j': 0.2,  
'n': 0.6666666666666666,  
'o': 0.0,  
'v': 0.4187725631768953,  
'z': 0.7142857142857143}
```

4. Let's replace the categories with the mean target value using the dictionary we created in *step 3* in the train and test sets:

```
X_train["A7"] = X_train["A7"].map(mapping)  
X_test["A7"] = X_test["A7"].map(mapping)
```

You can inspect the encoded **A7** variable by executing

```
X_train["A7"].head().
```

Now, let's perform target encoding with Feature-engine. First, we must split the data, as we did in *step 2*.

5. Let's import the encoder:

```
from feature_engine.encoding import MeanEncoder
```

6. Let's set up the target mean encoder to encode all categorical variables:

```
mean_enc = MeanEncoder(variables=None)
```

TIP

MeanEncoder() will find and encode all categorical variables by default. Alternatively, we can indicate the variables to encode by

passing their names in a list to the variables argument.

7. Let's fit the transformer to the train set so that it learns and stores the mean target value per category per variable. Note that we need to pass both the train set and target to fit the encoder:

```
mean_enc.fit(X_train, y_train)
```

8. Finally, let's encode the train and test sets:

```
X_train_enc = mean_enc.transform(X_train)
```

```
X_test_enc = mean_enc.transform(X_test)
```

TIP

The category-to-number pairs are stored as a dictionary of dictionaries in the `encoder_dict_` attribute. To display the stored parameters, execute `mean_enc.encoder_dict_`.

Feature-engine returns pandas DataFrames containing the categorical variables, ready to use in machine learning models.

To wrap up, let's implement mean encoding with Category Encoders blending the probabilities.

9. Let's import the encoder:

```
from category_encoders.target_encoder import  
TargetEncoder
```

10. Let's set up the encoder so that it encodes all categorical variables using blended probabilities when there are less than 25 observations in the category group:

```
mean_enc = TargetEncoder(
```

```
    cols=None, min_samples_leaf=25,  
    smoothing=1.0  
)
```

TIP

TargetEncoder() finds categorical variables automatically by default. Alternatively, we can indicate the variables to encode by passing their names in a list to the **cols** argument. The **smoothing** parameter controls the blend of the prior and posterior probability. Higher values decrease the contribution of the posterior probability to the encoding.

11. Let's fit the transformer to the train set so that it learns and stores the numerical representations for each category:

```
mean_enc.fit(X_train, y_train)
```

NOTE

The **min_samples_leaf** parameter refers to the minimum number of observations per category that a group should have to solely use the posterior probability. It is the equivalent of **k** in our weighting factor formula. In the original article, **k** was set to $\frac{1}{2}$ of **min_samples_leaf**. Category encoders expose this value and thus, we can optimize it with cross-validation.

12. Finally, let's encode the train and test sets:

```
X_train_enc = mean_enc.transform(X_train)  
X_test_enc = mean_enc.transform(X_test)
```

Category Encoders returns pandas DataFrames by default, where the original categorical variable values are replaced by their numerical representation. You can inspect the results by executing `X_train_enc.head()`.

How it works...

In this recipe, we replaced the categories with the mean target value using pandas, Feature-engine, and Category Encoders.

With pandas `groupby()`, using the `A7` categorical variable, followed by pandas `mean()` over the target variable, we created a pandas Series with the categories as indices and the target mean as values. With pandas `to_dict()`, we converted this Series into a dictionary. Finally, we used this dictionary to replace the categories in the train and test sets using pandas `map()`.

To perform the encoding with Feature-engine, we used `MeanEncoder()`. With `fit()`, the transformer found and stored the categorical variables and the mean target value per category. With `transform()`, categories were replaced with numbers in the train and test sets, returning pandas DataFrames.

Finally, we used `TargetEncoder()` from Category Encoders to replace categories with a blend of prior and posterior probability estimates of the target. We set `min_samples_leaf` to 25, which meant that if a category group had 25 observations or more, then the posterior probability was used for the encoding; alternatively, a blend of probabilities was used for the encoding. With `fit()`, the transformer found the categorical variables and

the numerical representation of the categories, while with `transform()`, the categories were replaced with numbers, returning pandas DataFrames with their encoded values.

There's more...

There is an alternative way to return *better* target estimates when the category groups are small. The replacement value for each category is determined as follows:

$$Si = \frac{n_{i(Y=1)} + pY \times m}{n_i + m}$$

Here, $n_{i(Y=1)}$ is the target mean for category i and n_i is the number of observations with category i . The target prior is given by pY and m is the weighting factor. With this adjustment, the only parameter that we have to set is the weight, m . If m is large, then more importance is given to the target's prior probability. This adjustment affects target estimates for all categories but mostly for those with fewer observations because, in such cases, m could be much larger than n_i in the formula's denominator.

For an implementation of this encoding using `MEstimateEncoder()`, visit this book's GitHub repository.

Encoding with the Weight of Evidence

The **Weight of Evidence (WoE)** was developed primarily for credit and financial industries to facilitate variable screening and exploratory analysis and to build more predictive linear models to evaluate the risk of loan defaults.

The WoE is computed from the basic odds ratio:

$$WoE = \log\left(\frac{\text{proportion positive cases}}{\text{proportion negative cases}}\right)$$

Here, positive and negative refer to the values of the target being 1 or 0, respectively. The proportion of positive cases per category is determined as the sum of positive cases per category group divided by the total positive cases in the training set, and the proportion of negative cases per category is determined as the sum of negative cases per category group divided by the total number of negative observations in the training set.

The WoE has the following characteristics:

- $WoE = 0$ if $p(\text{positive}) / p(\text{negative}) = 1$; that is, if the outcome is random
- $WoE > 0$ if $p(\text{positive}) > p(\text{negative})$
- $WoE < 0$ if $p(\text{negative}) > p(\text{positive})$

This allows us to directly visualize the predictive power of the category in the variable: the higher the WoE, the more likely the event will occur. If the WoE is positive, the event is likely to occur:

$$\log\left(\frac{p(Y=1)}{p(Y=0)}\right) = b_0 + b_1X_1 + b_2X_2 + \dots + b_nX_n$$

Logistic regression models a binary response, Y , based on X predictor variables, assuming that there is a linear relationship between X and the log of odds of Y .

Here, $\log(p(Y=1)/p(Y=0))$ is the log of odds. As you can see, the WoE encodes the categories in the same scale – that is, the log of odds – as the outcome of the logistic regression.

Therefore, by using WoE, the predictors are prepared and coded on the same scale, and the parameters in the logistic regression model – that is, the coefficients – can be directly compared.

In this recipe, we will perform WoE encoding using pandas and Feature-engine.

How to do it...

Let's begin by making some imports and preparing the data:

1. Import the required libraries and functions:

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
```

2. Let's load the dataset and divide it into train and test sets:

```
data = pd.read_csv("credit_approval_uci.csv")
X_train, X_test, y_train, y_test = train_test_split(
    data.drop(labels=["target"], axis=1),
    data["target"],
    test_size=0.3,
    random_state=0,
)
```

3. Let's get the inverse of the target values to be able to calculate the negative cases:

```
neg_y_train = pd.Series(
    np.where(y_train == 1, 0, 1),
    index=y_train.index
)
```

4. Let's determine the number of observations where the target variable takes a value of 1 or 0:

```
total_pos = y_train.sum()
total_neg = neg_y_train.sum()
```

5. Now, let's calculate the numerator and denominator of the WoE's formula, which we discussed earlier in this recipe:

```
pos = y_train.groupby(
    X_train["A1"]).sum() / total_pos
neg = neg_y_train.groupby(
    X_train["A1"]).sum() / total_neg
```

6. Now, let's calculate the WoE per category:

```
woe = np.log(pos/neg)
```

We can display the series with the category to WoE pairs by executing
print(woe):

A1

```
Missing      0.203599
a            0.092373
b           -0.042410
dtype: float64
```

7. Finally, let's replace the categories of **A1** with the WoE:

```
X_train["A1"] = X_train["A1"].map(woe)
X_test["A1"] = X_test["A1"].map(woe)
```

You can inspect the encoded variable by executing

```
X_train["A1"].head()
```

Now, let's perform WoE encoding using Feature-engine. First, we need to separate the data into train and test sets, as we did in *step 2*.

8. Let's import the encoder:

```
from feature_engine.encoding import WoEEncoder
```

9. Next, let's set up the encoder so that we can encode three categorical variables:

```
woe_enc = WoEEncoder(variables = ["A1", "A9", "A12"])
```

TIP

Feature-engine's **WoEEncoder()** will return an error if $p(0)=0$ for any category because the division by 0 is not defined. To avoid this error, we can group infrequent categories, as we will discuss in the next recipe, *Grouping rare or infrequent categories*.

10. Let's fit the transformer to the train set so that it learns and stores the WoE of the different categories:

```
woe_enc.fit(X_train, y_train)
```

TIP

We can display the dictionaries with the categories to WoE pairs by executing **woe_enc.encoder_dict_**.

11. Finally, let's encode the three categorical variables in the train and test sets:

```
X_train_enc = woe_enc.transform(X_train)  
X_test_enc = woe_enc.transform(X_test)
```

Feature-engine returns pandas DataFrames containing the encoded categorical variables ready to use in machine learning models.

How it works...

First, with pandas **sum()**, we determined the total number of positive and negative cases. Next, using pandas **groupby()**, we determined the fraction of positive and negative cases per category. And with that, we calculated the WoE per category.

Finally, we automated the procedure with Feature-engine. We used **WoEEncoder()**, which learned the WoE per category with the **fit()**

method, and then used `transform()`, which replaced the categories with the corresponding numbers.

See also

For an implementation of WoE with Category Encoders, visit this book's GitHub repository.

Grouping rare or infrequent categories

Rare categories are those present only in a small fraction of the observations. There is no rule of thumb to determine how small a small fraction is, but typically, any value below 5% can be considered rare.

Infrequent labels often appear only on the train set or only on the test set, thus making the algorithms prone to overfitting or being unable to score an observation. In addition, when encoding categories to numbers, we only create mappings for those categories observed in the train set, so we won't know how to encode new labels. To avoid these complications, we can group infrequent categories into a single category called **Rare** or **Other**.

In this recipe, we will group infrequent categories using pandas and Feature-engine.

How to do it...

First, let's import the necessary Python libraries and get the dataset ready:

1. Import the necessary Python libraries, functions, and classes:

```
import numpy as np
```

```
import pandas as pd

from sklearn.model_selection import train_test_split

from feature_engine.categorical_encoders import
RareLabelEncoder
```

2. Let's load the dataset and divide it into train and test sets:

```
data = pd.read_csv("credit_approval_uci.csv")

X_train, X_test, y_train, y_test = train_test_split(
    data.drop(labels=["target"], axis=1),
    data["target"],
    test_size=0.3,
    random_state=0,
)
```

3. Let's capture the fraction of observations per category in **A7** in a variable:

```
freqs = X_train["A7"].value_counts(normalize=True)
```

We can see the percentage of observations per category of **A7**, expressed as decimals, in the following output after executing **print(freqs)**:

v 0.573499

h 0.209110

ff 0.084886

bb 0.080745

z 0.014493

dd 0.010352

```
j 0.010352  
Missing 0.008282  
n 0.006211  
o 0.002070  
Name: A7, dtype: float64
```

If we consider those labels present in less than 5% of the observations as rare, then **z**, **dd**, **j**, **Missing**, **n**, and **o** are rare categories.

4. Let's create a list containing the names of the categories present in more than 5% of the observations:

```
frequent_cat = [  
    x for x in freqs.loc[freqs > 0.05].index.values]
```

If we execute **print(frequent_cat)**, we will see the frequent categories of **A7**:

```
['v', 'h', 'ff', 'bb'].
```

5. Let's replace rare labels – that is, those present in $\leq 5\%$ of the observations – with the "**Rare**" string:

```
x_train["A7"] = np.where(  
    x_train["A7"].isin(frequent_cat),  
    x_train["A7"], "Rare"  
)  
x_test["A7"] = np.where(  
    x_test["A7"].isin(frequent_cat),
```

```
x_test["A7"], "Rare"  
)
```

6. Let's determine the percentage of observations in the encoded variable:

```
x_train["A7"].value_counts(normalize=True)
```

We can see that the infrequent labels have now been re-grouped into the **Rare** category:

```
v      0.573499 h      0.209110 ff      0.084886  
bb     0.080745 Rare  0.051760 Name: A7, dtype: float64
```

Now, let's group rare labels using Feature-engine. First, we must divide the dataset into train and test sets, as we did in *step 2*.

7. Let's create a rare label encoder that groups categories present in less than 5% of the observations, provided that the categorical variable has more than four distinct values:

```
rare_encoder = RareLabelEncoder(tol=0.05,  
n_categories=4)
```

8. Let's fit the encoder so that it finds the categorical variables and then learns their most frequent categories:

```
rare_encoder.fit(X_train)
```

TIP

Upon fitting, the transformer will raise warnings, indicating that many categorical variables have less than four categories, thus their values will not be grouped. The transformer just lets you know that this is happening.

We can display the frequent categories per variable by executing `rare_encoder.encoder_dict_`, as well as the variables that will be encoded by executing `rare_encoder.variables_`.

9. Finally, let's group rare labels in the train and test sets:

```
x_train_enc = rare_encoder.transform(X_train)  
x_test_enc = rare_encoder.transform(X_test)
```

Now that we have grouped rare labels, we are ready to encode the categorical variables, as we've done in other recipes in this chapter.

How it works...

In this recipe, we grouped infrequent categories using pandas and Feature-engine.

We determined the fraction of observations per category of the **A7** variable using pandas `value_counts()` by setting the `normalize` parameter to `True`. Using list comprehension, we captured the names of the variables present in more than 5% of the observations. Finally, using NumPy's `where()`, we searched each row of **A7**, and if the observation was one of the frequent categories in the list, which we checked using the pandas `isin()` method, its value was kept; otherwise, its original value was replaced with "**Rare**".

We automated the preceding steps for multiple categorical variables using Feature-engine. For this, we used Feature-engine's `RareLabelEncoder()`. By setting `tol` to **0.05**, we retained categories present in more than 5% of the observations. By setting `n_categories` to **4**, we only group rare

categories in variables with more than four unique values. With the `fit()` method, the transformer identified the categorical variables and then learned and stored their frequent categories. With the `transform()` method, the transformer replaced infrequent categories with the "`Rare`" string.

Performing binary encoding

Binary encoding is a categorical encoding technique that uses binary code – that is, a sequence of zeroes and ones – to represent the different categories of the variable. How does it work? First, the categories are arbitrarily replaced with ordinal numbers, as shown in the intermediate step of the following table. Then, those numbers are converted into binary code. For example, integer 1 can be represented as sequence 10, integer 2 as 01, integer 3 as 11, and integer 0 as 00. The digits in the two positions of the binary string become the columns, which are the encoded representations of the original variable:

Color	Intermediate step	1st	2nd
Blue	1	1	0
Red	2	0	1
Green	3	1	1
Yellow	0	0	0

Figure 2.9 – Table showing the steps required for binary encoding of the color variable

Binary encoding encodes the data in fewer dimensions than one-hot encoding. In our example, the **Color** variable would be encoded into $k-1$ categories by one-hot encoding – that is, three variables – but with binary encoding, we can represent the variable with only two features. More generally, we determine the number of binary features needed to encode a variable as $\log_2(\text{number of distinct categories})$; in our example, $\log_2(4) = 2$ binary features.

Binary encoding is an alternative method to one-hot encoding where we do not lose information about the variable, yet we obtain fewer features after the encoding. This is particularly useful when we have highly cardinal variables. For example, if a variable contains 128 unique categories, with

one-hot encoding, we would need 127 features to encode the variable, whereas with binary encoding, we would only need 7 ($\log_2(128)=7$). Thus, this encoding prevents the feature space from exploding. In addition, binary-encoded features are also suitable for linear models. On the downside, the derived binary features **lack human interpretability**, so if we need to interpret the decisions made by our models, this encoding method may not be a suitable option.

In this recipe, we will learn how to perform binary encoding using Category Encoders.

How to do it...

First, let's import the necessary Python libraries and get the dataset ready:

1. Import the required Python library, function, and class:

```
import pandas as pd

from sklearn.model_selection import train_test_split

from category_encoders.binary import BinaryEncoder
```

2. Let's load the dataset and divide it into train and test sets:

```
data = pd.read_csv("credit_approval_uci.csv")

X_train, X_test, y_train, y_test = train_test_split(
    data.drop(labels=["target"], axis=1),
    data["target"],
    test_size=0.3,
    random_state=0,
```

```
)
```

3. Let's inspect the unique categories in **A7**:

```
x_train["A7"].unique()
```

In the following output, we can see that **A7** has 10 different categories:

```
array(['v', 'ff', 'h', 'dd', 'z', 'bb', 'j', 'Missing',
'n', 'o'], dtype=object)
```

4. Let's create a binary encoder to encode **A7**:

```
encoder = BinaryEncoder(cols=["A7"],
drop_invariant=True)
```

TIP

BinaryEncoder(), as well as other encoders from the Category Encoders package, allow us to select the variables to encode. We simply pass the column names in a list to the **cols** argument.

5. Let's fit the transformer to the train set so that it calculates how many binary variables it needs and creates the variable-to-binary code representations:

```
encoder.fit(X_train)
```

6. Finally, let's encode **A7** in the train and test sets:

```
X_train_enc = encoder.transform(X_train)
X_test_enc = encoder.transform(X_test)
```

We can display the top rows of the transformed train set by executing **print(X_train_enc.head())**, which returns the following output:

	A1	A2	A3	A4	A5	A6	A7_0	A7_1	A7_2	A7_3	A8	A9	A10	A11	A12	A13	A14	A15
596	a	46.08	3.000	u	g	c	0	0	0	1	2.375	t	t	8	t	g	396.0	4159
303	a	15.92	2.875	u	g	q	0	0	0	1	0.085	f	f	0	f	g	120.0	0
204	b	36.33	2.125	y	p	w	0	0	0	1	0.085	t	t	1	f	g	50.0	1187
351	b	22.17	0.585	y	p	ff	0	0	1	0	0.000	f	f	0	f	g	100.0	0
118	b	57.83	7.040	u	g	m	0	0	0	1	14.000	t	t	6	t	g	360.0	1332

Figure 2.10 – DataFrame with the variables after binary encoding

Binary encoding returned four binary variables for **A7**, which are **A7_0**, **A7_1**, **A7_2**, and **A7_3**, instead of the nine that would have been returned by one-hot encoding.

How it works...

In this recipe, we performed binary encoding using the Category Encoders package. First, we loaded the dataset and divided it into train and test sets using `train_test_split()` from scikit-learn. Next, we used `BinaryEncoder()` to encode the **A7** variable. With the `fit()` method, `BinaryEncoder()` created a mapping from category to set of binary columns, and with the `transform()` method, the encoder encoded the **A7** variable in both the train and test sets.

TIP

With one-hot encoding, we would have created nine binary variables (**k-1 = 10 unique categories - 1 = 9**) to encode all of the information in **A7**. With binary encoding, we can represent the variable in fewer dimensions by using **log2(10)=3.3**; that is, we only need four binary variables.

See also

For more information about **BinaryEncoder()**, visit https://contrib.scikit-learn.org/category_encoders/binary.xhtml.

For a nice example of the output of binary encoding, check out the following resource:

<https://stats.stackexchange.com/questions/325263/binary-encoding-vs-one-hot-encoding>.

For a comparative study of categorical encoding techniques for neural network classifiers, visit

https://www.researchgate.net/publication/320465713_A_Comparative_Study_of_Categorical_Variable_Encoding_Techniques_for_Neural_Network_Classifiers.

3

Transforming Numerical Variables

Statistical methods used in data analysis make certain assumptions about the data. For example, in the general linear model, it is assumed that the values of the dependent variable (the target) are independent, that there is a linear relationship between the target and the independent (predictor) variables, and that the residuals – that is, the difference between the predictions and the real values of the target – are normally distributed and centered at 0. When these assumptions are not met, the resulting probabilistic statements might not be accurate. To correct for failure in the assumptions and thus improve the performance of the models, we can transform variables before the analysis.

Variable transformation consists of replacing the original variable values with a function of that variable. More generally, transforming variables with mathematical functions helps reduce variable skewness, improve the value spread, and sometimes unmask linear and additive relationships between predictors and the target. Commonly used mathematical transformations include the logarithm, reciprocal, power, square, and cube root transformations, as well as the Box-Cox and Yeo-Johnson transformations. This set of transformations is commonly referred to as **variance stabilizing transformations**. Variance stabilizing transformations intend to bring the distribution of the variable to a more symmetric – that is, Gaussian – shape.

In this chapter, we will discuss when to use each transformation and then implement them using NumPy, SciPy, scikit-learn, and Feature-engine.

This chapter will cover the following recipes:

- Transforming variables with the logarithm function
- Transforming variables with the reciprocal function
- Using the square root to transform variables
- Using power transformations
- Performing Box-Cox transformation
- Performing Yeo-Johnson transformation

Transforming variables with the logarithm function

The logarithm function is a powerful transformation for dealing with positive data with a right-skewed distribution (observations accumulate at lower values of the variable). A common example is variable income, with a heavy accumulation of values toward smaller salaries. The log transform has a strong effect on the shape of the variable distribution.

In this recipe, we will perform logarithmic transformation using NumPy, scikit-learn, and Feature-engine. We will also create a diagnostic plot function to evaluate the effect of the transformation on the variable distribution.

Getting ready

To evaluate the variable distribution and understand whether a transformation improves value spread and stabilizes the variance, we can visually inspect the data with histograms and **Quantile-Quantile (Q-Q)** plots. A Q-Q plot helps us determine whether two variables show a similar distribution. In a Q-Q plot, we plot the quantiles of one variable against the quantiles of the second variable. If we plot the quantiles of the variable of interest against the expected quantiles of the normal distribution, then we can determine whether our variable is also normally distributed. If the variable is normally distributed, the points in the Q-Q plot will fall along a 45-degree diagonal.

NOTE

A quantile is the fraction of points below a given value. Thus, the 0.2 quantile is the point in the distribution at which 20% of the observations fall below and 80% above that value.

How to do it...

Let's begin by importing the libraries and getting the dataset ready:

1. Import the required Python libraries and dataset:

```
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
import scipy.stats as stats  
from sklearn.datasets import fetch_california_housing
```

2. Let's load the California housing dataset into a **pandas** DataFrame:

```
X, y = fetch_california_housing(return_X_y=True,  
                                 as_frame=True)
```

3. Let's explore the distributions of all the variables in the dataset by plotting histograms with pandas:

```
X.hist(bins=30, figsize=(12, 12))  
plt.show()
```

In the following output, we can see that the **MedInc** variable shows a mild right-skewed distribution, variables such as **AveRooms** and **Population** are heavily right-skewed, and that the **HouseAge** variable shows an even spread of values across its range:

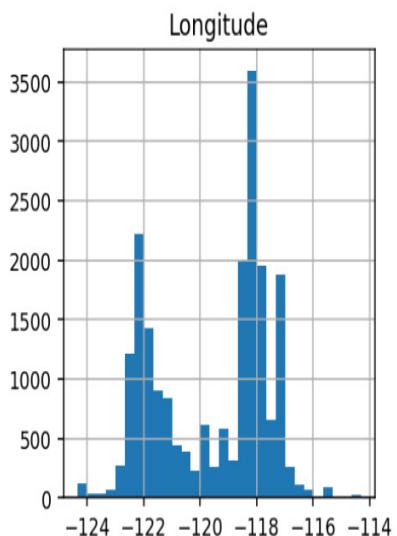
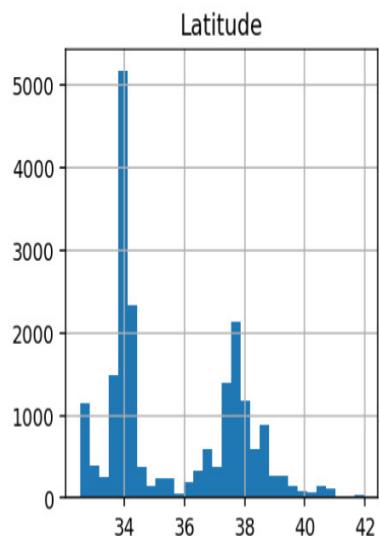
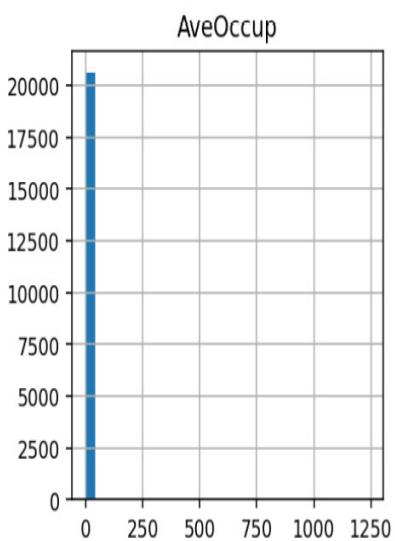
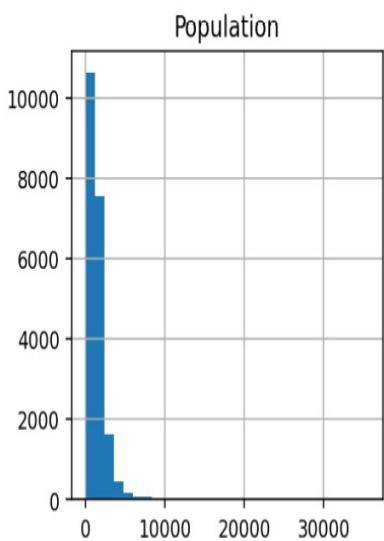
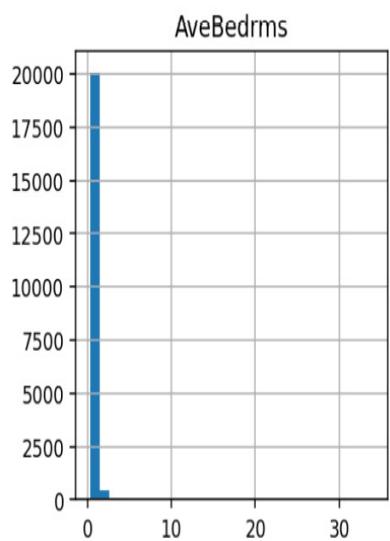
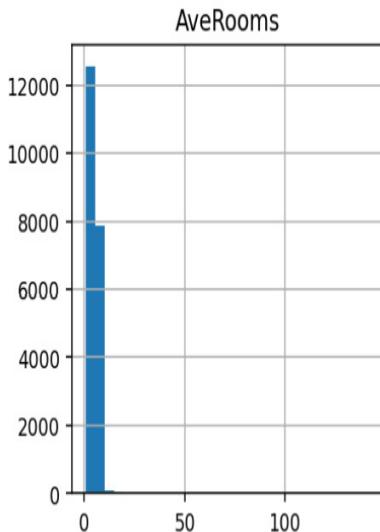
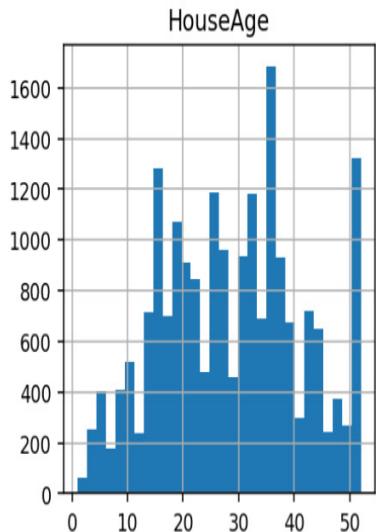
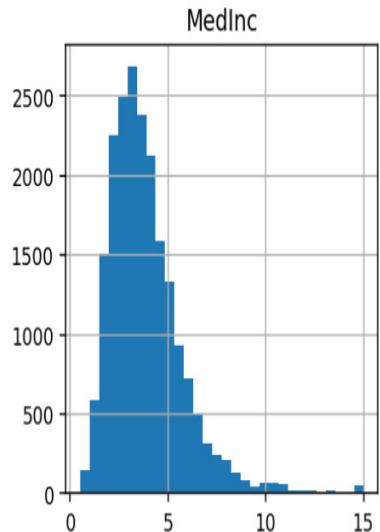


Figure 3.1 – Histograms with the distribution of the numerical variables

4. To evaluate the effect of the transformation on the variable distribution, we'll create a function that takes a DataFrame and a variable name as inputs and plots a histogram next to a Q-Q plot:

```
def diagnostic_plots(df, variable):  
  
    plt.figure(figsize=(15, 6))  
  
    plt.subplot(1, 2, 1)  
  
    df[variable].hist(bins=30)  
  
    plt.title(f"Histogram of {variable}")  
  
    plt.subplot(1, 2, 2)  
  
    stats.probplot(df[variable], dist="norm", plot=plt)  
  
    plt.title(f"Q-Q plot of {variable}")  
  
    plt.show()
```

5. Now, let's plot the distribution of the **MedInc** variable:

```
diagnostic_plots(X, "MedInc")
```

The following output shows that **MedInc** has a mild right-skewed distribution:

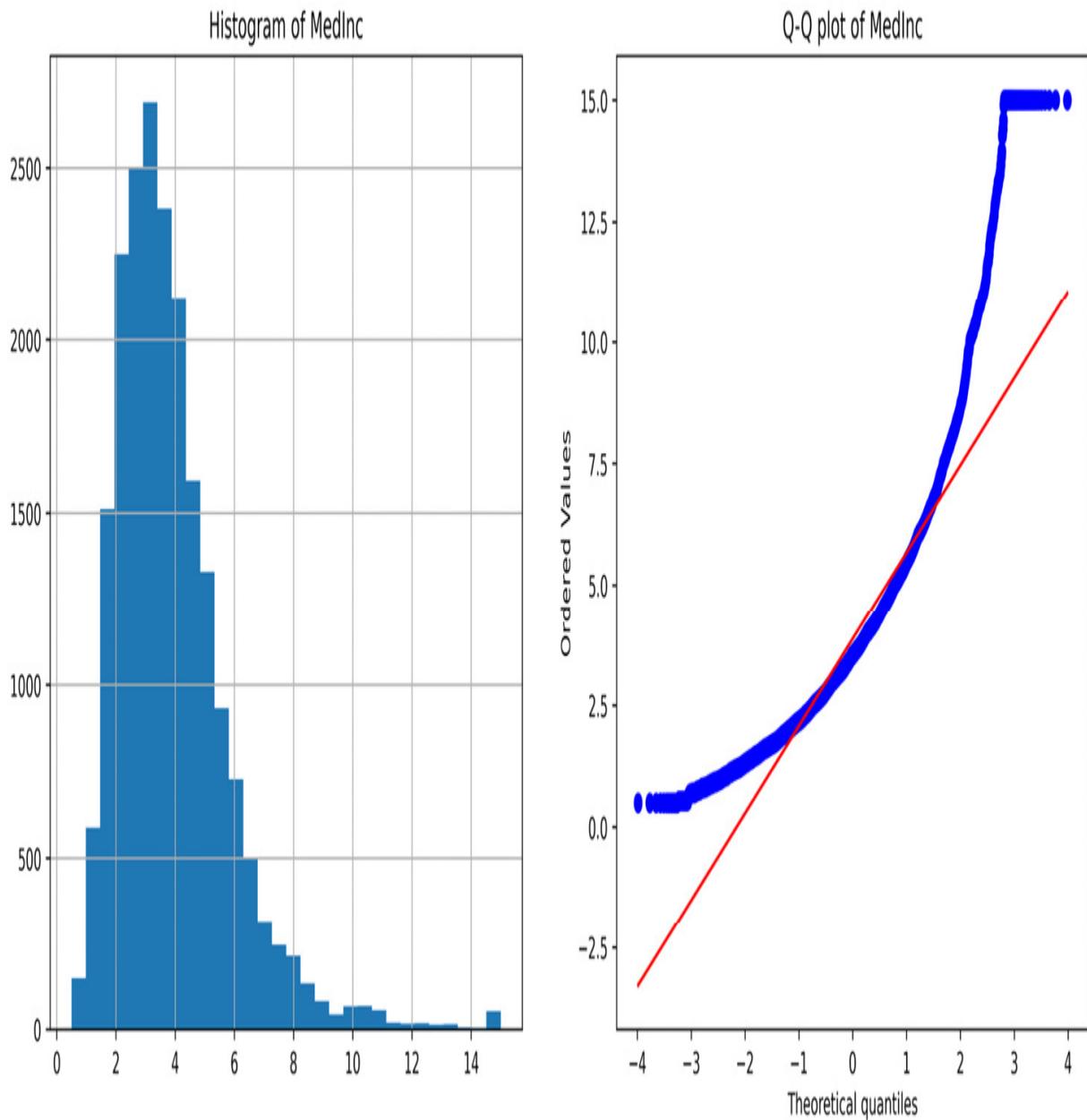


Figure 3.2 – Histogram and Q-Q plot of the MedInc variable

Now, let's transform the data with the logarithm.

6. First, let's make a copy of the original DataFrame using **pandas**

copy():

```
X_tf = X.copy()
```

We've created a copy so that we can modify the values in the copy and not in the original DataFrame, which we need for the rest of this recipe.

NOTE

If we execute `X_tf = X` instead of using `pandas copy()`, `X_tf` will not be a copy of the DataFrame; instead, it will be another view of the same data. Therefore, changes made in `X_tf` will be reflected in `X` as well.

7. Let's make a list with the variables that we want to transform:

```
variables = ["MedInc", "AveRooms", "AveBedrms",
             "Population"]
```

8. Let's apply the logarithmic transformation with NumPy to the variables from *step 7* and capture the transformed variables in the new DataFrame:

```
X_tf[variables] = np.log(X[variables])
```

NOTE

Remember that the logarithm transformation can only be applied to strictly positive variables. If the variables have zero or negative values, sometimes, it is useful to add a constant to make those values positive. We could add a constant value of `1` using `X_tf[variables] = np.log(X[variables] + 1)`.

9. Let's check the distribution of `MedInc` after the transformation with the diagnostic function from *step 4*:

```
diagnostic_plots(X_tf, "MedInc")
```

In the following output, we can see that the logarithmic transformation returned a more evenly distributed variable that approximates the theoretical normal distribution in the Q-Q plot better:

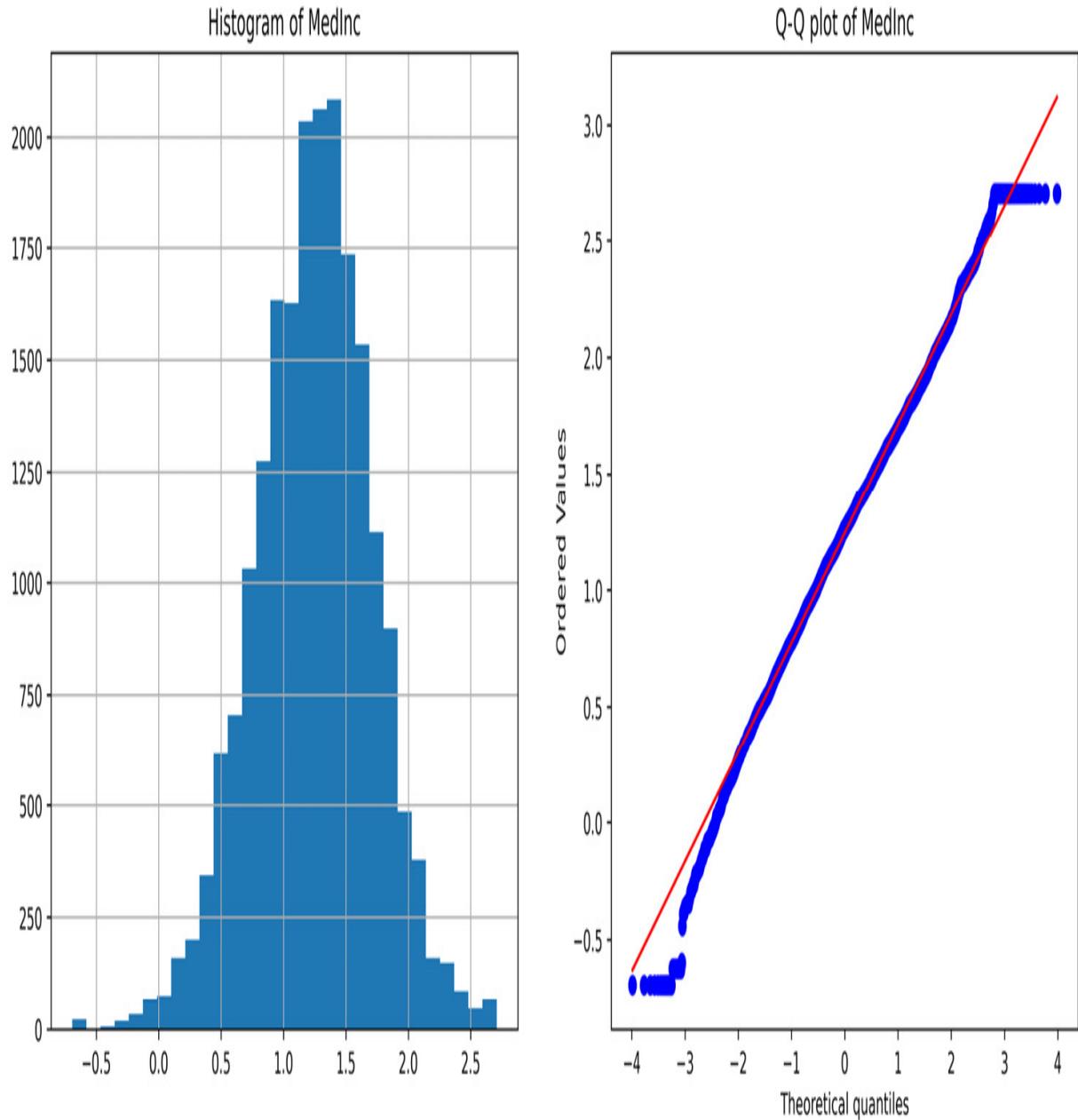


Figure 3.3 – Histogram and Q-Q plot of the MedInc variable after the log transformation

Go ahead and plot the other transformed variables to familiarize yourself with the effect of the log transform on distributions.

Now, let's apply the logarithmic transformation with scikit-learn.

10. Let's import **FunctionTransformer()**:

```
from sklearn.preprocessing import FunctionTransformer
```

Before we proceed, we need to take a copy of the original dataset, as we did in *step 6*.

11. Let's set up the transformer so that we can apply the logarithm and also revert the transformed variable to its original representation:

```
transformer = FunctionTransformer(np.log,  
                                inverse_func=np.exp)
```

NOTE

If we set up **FunctionTransformer()** with the default parameter, **validate=False**, we don't need to fit the transformer before transforming the data. If we set **validate=True**, the transformer will check the input data to the **fit** method. The latter is useful when fitting the transformer with a DataFrame so that it learns and stores the variable names.

12. Let's transform the positive variables from *step 7*:

```
X_tf[variables] = transformer.transform(X[variables])
```

NOTE

Scikit-learn transformers return NumPy arrays by default. In this case, we assigned the results of the array directly to our existing

`DataFrame`. Go ahead and check the results of the transformation with the diagnostic function from *step 4*.

13. Because we set up the exponential function in the inverse function of `FunctionTransformer()` in *step 11*, we can revert the transformed variables to their original representations, as follows:

```
x_tf[variables] = transformer.inverse_transform(  
    x_tf[variables])
```

If you check the distribution by executing `diagnostic_plots(x_tf, "MedInc")`, you should see a plot identical to that returned by *step 5*.

NOTE

To add a constant value to the variables, in case they are not strictly positive, we can set up `FunctionTransformer()` using `transformer = FunctionTransformer(lambda x: np.log(x + 1))`.

Now, let's do logarithm transformation with Feature-engine.

14. Let's import `LogTransformer()`:

```
from feature_engine.transformation import  
LogTransformer
```

15. Let's set up the transformer so that it modifies the variables from *step 7* and then fits them to the dataset:

```
lt = LogTransformer(variables = variables)  
lt.fit(X)
```

NOTE

If the variables argument is left as **None**, **LogTransformer()** identifies and applies the logarithm to all the numerical variables in the dataset.

Alternatively, we can indicate which variables to modify, as we did in *step 15*.

16. Finally, let's transform the data:

```
X_tf = lt.transform(X)
```

Note that **X_tf** is a pandas DataFrame that contains the original variables, where only the variables in the list from *step 7* were transformed by the logarithm.

17. Feature-engine's **LogTransformer()** comes with the **inverse_transform** functionality out of the box to revert the transformed variable to its original representation:

```
X_tf = lt.inverse_transform(X_tf)
```

If you check the distribution of the variables after *step 17*, they should be identical to those of the original data.

NOTE

Feature-engine has a dedicated transformer that adds constant values to the variables before the log transform. Check the *There's more...* section later in this recipe for more details.

How it works...

In this recipe, we applied the logarithm transformation to a subset of positive variables using NumPy, scikit-learn, and Feature-engine.

To compare the effect of the transformation on the variable distribution, we created a diagnostic function to plot a histogram next to a Q-Q plot. To create the Q-Q plot, we used `scipy.stats.probplot()`, which plotted the quantiles of our variable of interest in the `y` axis versus the quantiles of a theoretical normal distribution, which we indicated by setting the `dist` parameter to `norm` in the `x` axis. We used `Matplotlib` to display the plot by setting the `plot` parameter to `plt`.

To follow up with the commands used to display the plots, with `plt.figure()` and `figsize`, we adjusted the size of the figure and, with `plt.subplot()`, we organized the two plots in **1** row with **2** columns – that is, one plot next to the other. The numbers within `plt.subplot()` indicated the number of rows, the number of columns, and the place of the plot in the figure, respectively. We placed the histogram in position 1 and the Q-Q plot in position 2 – that is, left and right, respectively.

To test the function, we plotted a histogram and a Q-Q plot for the `MedInc` variable before the transformation and observed that `MedInc` was not normally distributed: most observations were at the left of the histogram and the values deviated from the 45-degree line in the Q-Q plot at both ends of the distribution.

Next, using `np.log()`, we applied the logarithm to a slice of the DataFrame with four positive variables. To evaluate the effect of the transformation, we plotted a histogram and Q-Q plot of the transformed `MedInc` variable. We observed that, after the log transformation, the values

were more centered in the histogram and that, in the Q-Q plot, they only deviated from the 45-degree line toward the far ends of the distribution.

Next, we used **FunctionTransformer()** from scikit-learn, which applies any user-defined function to a dataset and returns the result of the operation in a NumPy array. We passed `np.log()` as an argument to apply the log transformation and Numpy's `exp()` for the inverse transformation to **FunctionTransformer()**. With the `transform()` method, we transformed a slice of the DataFrame with the positive variables. Finally, with `inverse_transform()`, we reverted the variable values to their original representation.

Finally, we used Feature-engine's **LogTransformer()** to specify the variables to transform in a list as an argument. The `fit()` method of the transformer checked that the variables were numerical and positive, and the `transform()` method called `np.log()` under the hood to transform the indicated variables. With `inverse_transform()`, we reverted the transformed variables to their original representations.

There's more...

Feature-engine has a dedicated transformer for adding a constant value to variables that are not strictly positive, before applying the logarithm: **LogCpTransformer()**. The advantages of this transformer are that it can (i) add the same constant to all variables, as we saw in this recipe, (ii) automatically identify the minimum value required to make the variables positive, or (iii) add different arbitrary values defined by the user to different variables.

You can find a code implementation of `LogCpTransformer()` in this book's GitHub repository: <https://github.com/PacktPublishing/Python-Feature-Engineering-Cookbook-Second-Edition/blob/main/ch03-variable-transformation/Recipe-6-Yeo-Johnson-transformation.ipynb>.

Transforming variables with the reciprocal function

The reciprocal function is defined as $1/x$. The reciprocal transformation is often useful when we have ratios – that is, values resulting from the division of two variables. Examples of this are *population density* – that is, people per area – and, as we will see in this recipe, *house occupancy* – that is, the number of occupants per house.

The reciprocal transformation is not defined for the value 0, and although defined for negative values, it is mainly useful for transforming positive variables.

In this recipe, we will implement the reciprocal transformation using NumPy, scikit-learn, and Feature-engine, and compare its effect on variable distribution using histograms and a Q-Q plot.

How to do it...

Let's begin by importing the libraries and getting the dataset ready:

1. Import the required Python libraries and data:

```
import numpy as np  
import pandas as pd
```

```
import matplotlib.pyplot as plt  
import scipy.stats as stats  
from sklearn.datasets import fetch_california_housing
```

2. Let's load the California housing dataset:

```
x, y = fetch_california_housing(return_X_y=True,  
                                 as_frame=True)
```

3. To evaluate variable distributions, we'll create a function that takes a DataFrame and a variable name as inputs and plots a histogram next to a Q-Q plot:

```
def diagnostic_plots(df, variable):  
    plt.figure(figsize=(15, 6))  
    plt.subplot(1, 2, 1)  
    df[variable].hist(bins=30)  
    plt.title(f"Histogram of {variable}")  
    plt.subplot(1, 2, 2)  
    stats.probplot(df[variable], dist="norm", plot=plt)  
    plt.title(f"Q-Q plot of {variable}")  
    plt.show()
```

4. Now, let's plot the distribution of the **AveOccup** variable, which specifies the **average occupancy** of the house:

```
diagnostic_plots(x, "AveOccup")
```

The **AveOccup** variable shows a very strong right-skewed distribution, as shown in the following output:

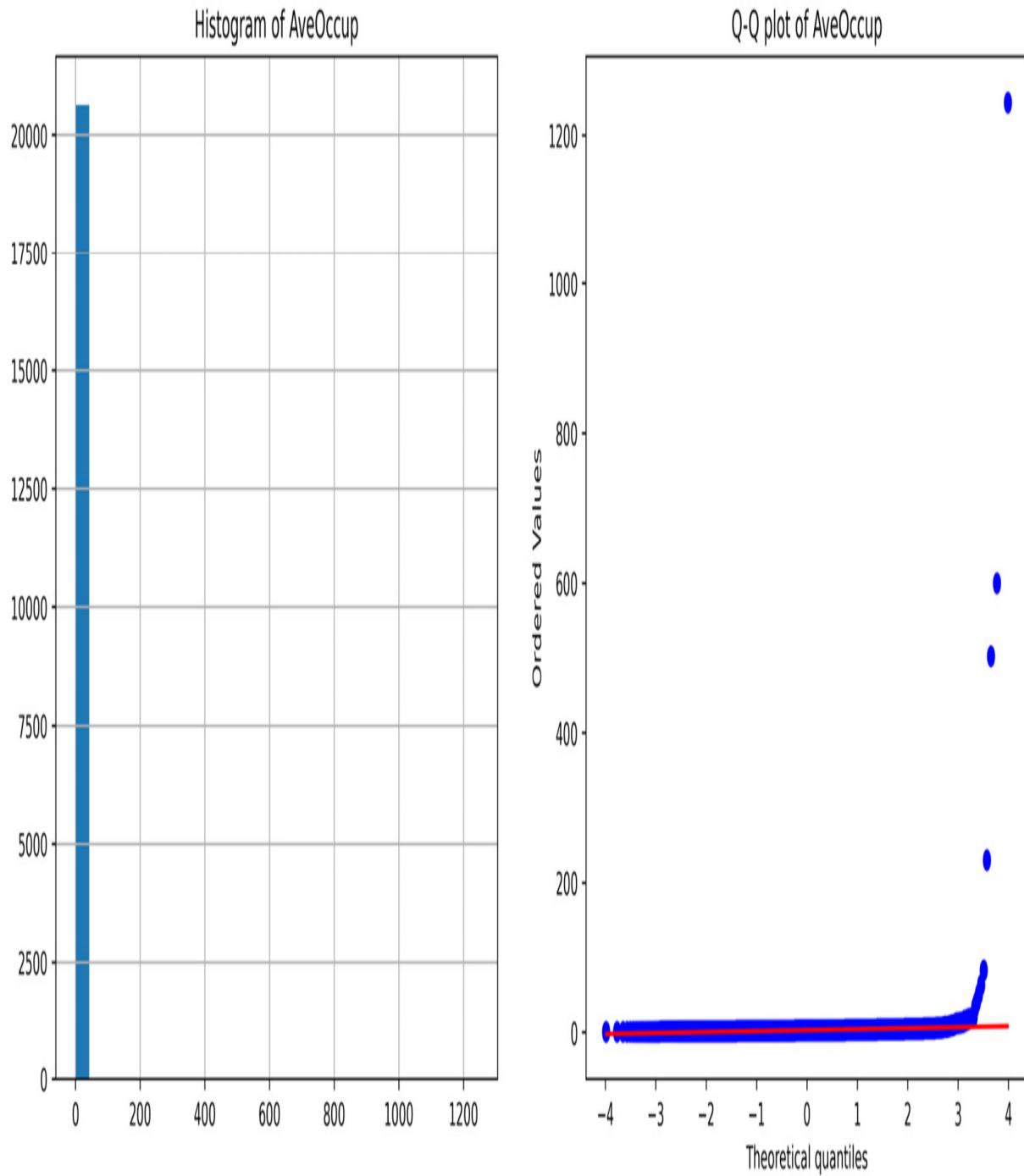


Figure 3.4 – Histogram and Q-Q plot of the AveOccup variable

NOTE

The **AveOccup** variable refers to the average number of household members – that is, the ratio between the number of people and the number of houses in a certain area. This is a promising variable for a reciprocal transformation. You can find more details about the variables and the dataset by executing `data = fetch_california_housing()` followed by `print(data.DESCR)`.

Now, let's apply the reciprocal transformation with NumPy.

5. First, let's make a copy of the original DataFrame so that we can modify the values in the copy and not in the original one, which we need for the rest of this recipe:

```
X_tf = X.copy()
```

NOTE

Remember that executing `X_tf = X`, instead of using `pandas copy()`, creates an additional view of the same data. Therefore, changes that are made in `X_tf` will be reflected in `X` as well.

6. Let's apply the reciprocal transformation to the **AveOccup** variable:

```
X_tf["AveOccup"] = np.reciprocal(X_tf["AveOccup"])
```

7. Let's check the distribution of the **AveOccup** variable after the transformation with the diagnostic function we created in *step 3*:

```
diagnostic_plots(X_tf, "AveOccup")
```

NOTE

After the transformation, **AveOccup** is now the ratio of the number of houses and the number of people in a certain area – in other words, houses per citizen.

Here, we can see a dramatic change in the distribution of the **AveOccup** variable after the reciprocal transformation:

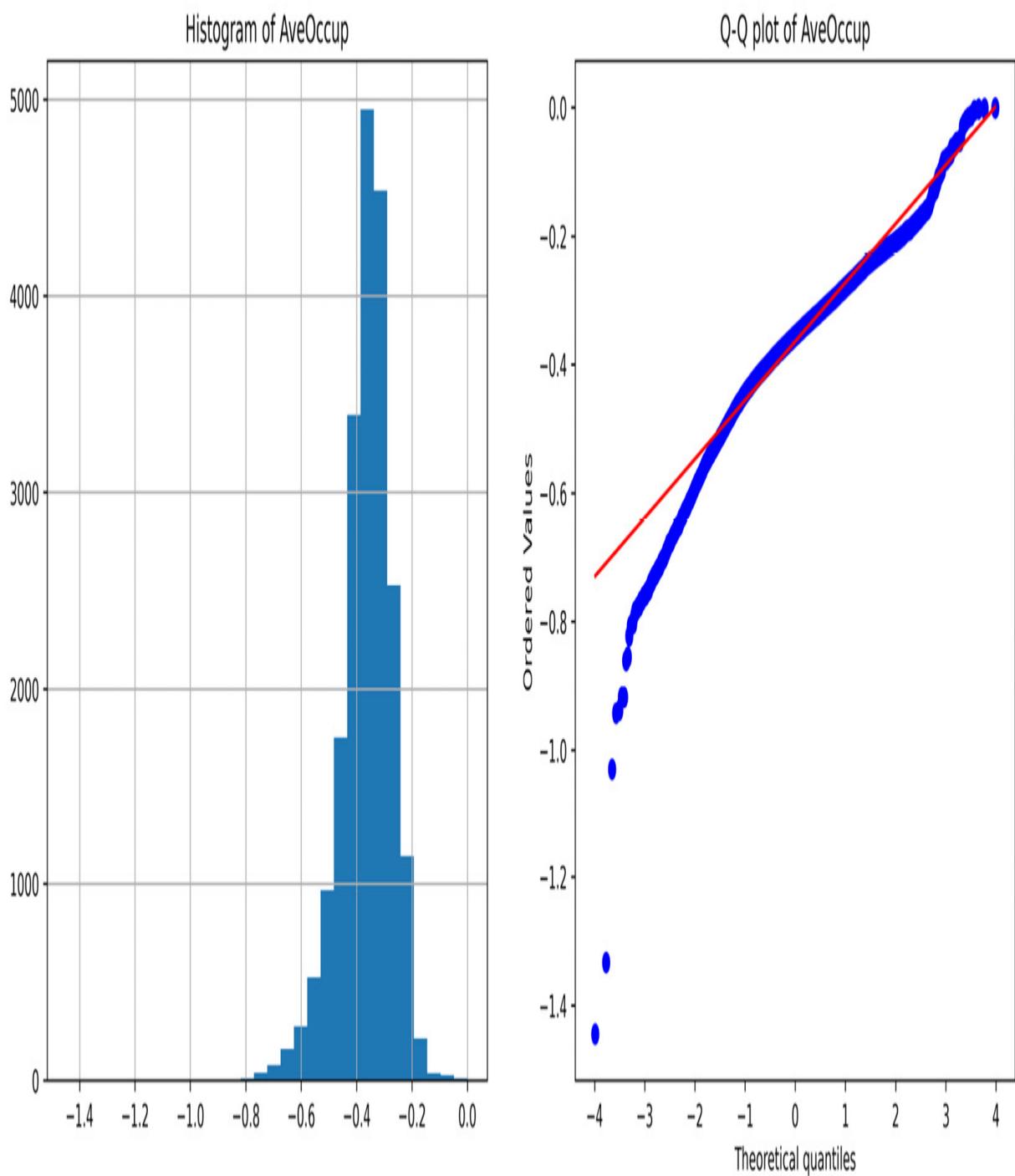


Figure 3.5 – Histogram and Q-Q plot of the AveOccup variable after the reciprocal transformation

Now, let's apply the reciprocal transformation with scikit-learn. We need to make a new copy of the data, as we did in *step 5*.

8. Let's import **FunctionTransformer()**:

```
from sklearn.preprocessing import FunctionTransformer
```

9. Let's set up the transformer by passing **np.reciprocal** as an argument:

```
transformer = FunctionTransformer(np.reciprocal)
```

NOTE

By default, **FunctionTransformer()** does not need to be fit before transforming the data.

10. Now, let's transform the variable of interest after making a copy of the original dataset:

```
X_tf = X.copy()
```

```
X_tf["AveOccup"] = transformer.transform(X["AveOccup"])
```

You can check the effect of the transformation using the function from *step 3*.

NOTE

Note that the inverse of the reciprocal function is the reciprocal function, so if you transform the transformed data, you revert it to its original representation.

Now, let's apply the reciprocal transformation with Feature-engine.

11. Let's import **ReciprocalTransformer()**:

```
from feature_engine.transformation import  
ReciprocalTransformer
```

12. We must set up the transformer so that it modifies the **AveOccup** variable and then fit it to the dataset:

```
rt = ReciprocalTransformer(variables="AveOccup")  
rt.fit(X)
```

NOTE

If the **variables** argument is **None**, the transformer applies the reciprocal function to *all the numerical variables* in the dataset. If some of the variables contain a value of 0, the transformer will raise an error.

13. Let's transform the selected variable in our dataset:

```
X_tf = rt.transform(X)
```

ReciprocalTransformer() will return a new pandas DataFrame containing the original variables, where the variable indicated in step 12 is transformed with the reciprocal function.

How it works...

In this recipe, we applied the reciprocal transformation using NumPy, scikit-learn, and Feature-engine.

To evaluate the variable distribution, we used the function to plot a histogram next to a Q-Q plot, which we described in the *How it works...* section of the *Transforming variables with the logarithm function* recipe, earlier in this chapter.

We plotted the histogram and Q-Q plot of the **AveOccup** variable, which showed a heavy right-skewed distribution; most of its values were at the left of the histogram and they deviated from the 45-degree line toward the right end of the distribution in the Q-Q plot.

To carry out the reciprocal transformation, we applied **np.reciprocal()** to the variable. After the transformation, AveOccup's values were more evenly distributed across the value range and followed the theoretical quantiles of the normal distribution in the Q-Q plot more closely.

Next, we used **np.reciprocal()** from within scikit-learn's **FunctionTransformer()**. The advantage of applying the reciprocal transformation this way is that we can line up this transformation step within a scikit-learn pipeline.

Finally, we used Feature-engine's **ReciprocalTransformer()** while specifying the variable to transform. The **fit()** method checked that the variables were numerical, while the **transform()** method called **np.reciprocal()** under the hood to transform the variables, returning a pandas DataFrame. Feature-engine's **ReciprocalTransformer()** provides functionality to revert the variable to its original representation out of the box via the **inverse_transform()** method.

Using the transformers from scikit-learn or Feature-engine, instead of **np.reciprocal()**, allows us to carry out this transformation as an additional step of a feature engineering pipeline within the **Pipeline** object from scikit-learn. The difference between **FunctionTransformer()** and **ReciprocalTransformer()** is that the first one can apply any user-specified transformation, whereas the latter only applies the reciprocal

function. scikit-learn also returns Numpy arrays by default and transforms all variables in the dataset, whereas Feature-engine returns pandas DataFrames and can modify subsets of variables within the data.

Using the square root to transform variables

The square root transformation, \sqrt{x} , as well as its variations, the Anscombe transformation, $\sqrt{(x+3/8)}$, and the Freeman-Tukey transformation, $\sqrt{x} + \sqrt{(x+1)}$, are variance stabilizing transformations that transform a variable with a Poisson distribution into one with an approximately standard Gaussian distribution. The square root transformation is a form of power transformation where the exponent is $1/2$ and is only defined for positive values.

The Poisson distribution is a probability distribution that indicates the number of times an event is likely to occur. In other words, it is a count distribution. It is right-skewed and its variance equals its mean. Examples of variables that could follow a Poisson distribution are the number of financial items of a customer, such as the number of current accounts or credit cards, the number of passengers in a vehicle, and the number of occupants in a household.

In this recipe, we will implement square root transformations using NumPy, scikit-learn, and Feature-engine.

How to do it...

In this recipe, we will create a toy dataset with two variables whose values are drawn from a Poisson distribution:

1. Let's begin by importing the necessary libraries:

```
import numpy as np import pandas as pd  
import scipy.stats as stats
```

2. Let's create a DataFrame with two variables drawn from a Poisson distribution with mean values of **2** and **3**, respectively, and **10000** observations:

```
df = pd.DataFrame()  
df["counts1"] = stats.poisson.rvs(mu=3, size=10000)  
df["counts2"] = stats.poisson.rvs(mu=2, size=10000)
```

3. Let's create a function that takes a DataFrame and a variable name as inputs and plots a bar graph with the number of observations per value next to a Q-Q plot:

```
def diagnostic_plots(df, variable):  
    plt.figure(figsize=(15, 6))  
    plt.subplot(1, 2, 1)  
    df[variable].value_counts().sort_index().  
    plot.bar()  
    plt.title(f"Histogram of {variable}")  
    plt.subplot(1, 2, 2)  
    stats.probplot(df[variable], dist="norm", plot=plt)  
    plt.title(f"Q-Q plot of {variable}")
```

```
plt.show()
```

4. Let's create a bar plot and a Q-Q plot for one of the variables in the data using the function from *step 3*:

```
diagnostic_plots(df, "counts1")
```

Here, we can see the Poisson distribution in the output:

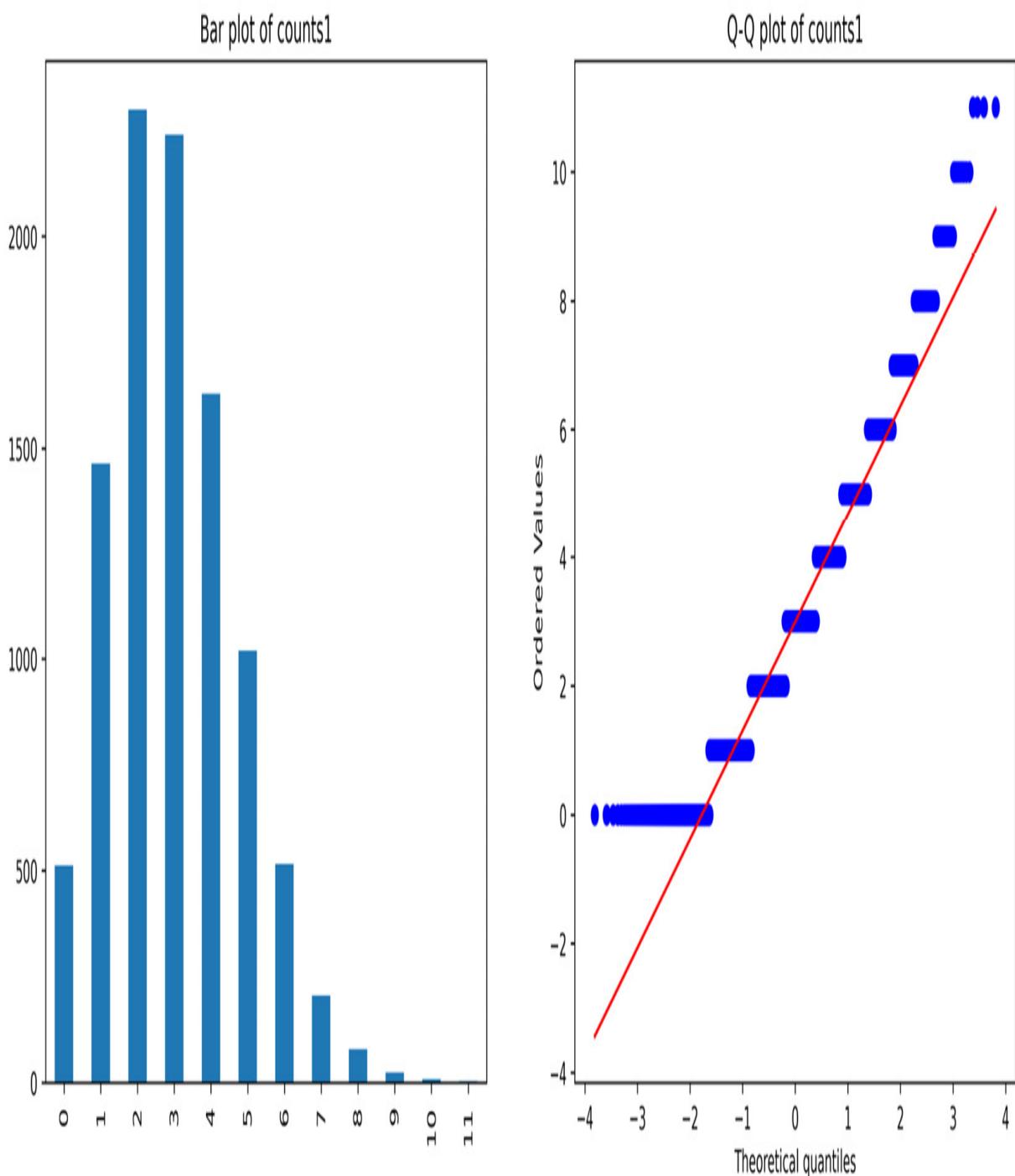


Figure 3.6 – Bar and Q-Q plots of the counts1 variable

5. Now, let's make a copy of the toy dataset:

```
df_tf = df.copy()
```

6. Let's apply the square root transformation to both variables:

```
df_tf[["counts1", "counts2"]] = np.sqrt(  
    df[["counts1", "counts2"]])
```

7. Let's round the values to **2** decimals for nicer visualization:

```
df_tf[["counts1", "counts2"]] = np.round(  
    df_tf[["counts1", "counts2"]], 2)
```

If we plot the variable distribution as we did in *step 4*, we will see a more *stabilized* variance, as the dots in the Q-Q plot follow the 45-degree diagonal more closely:

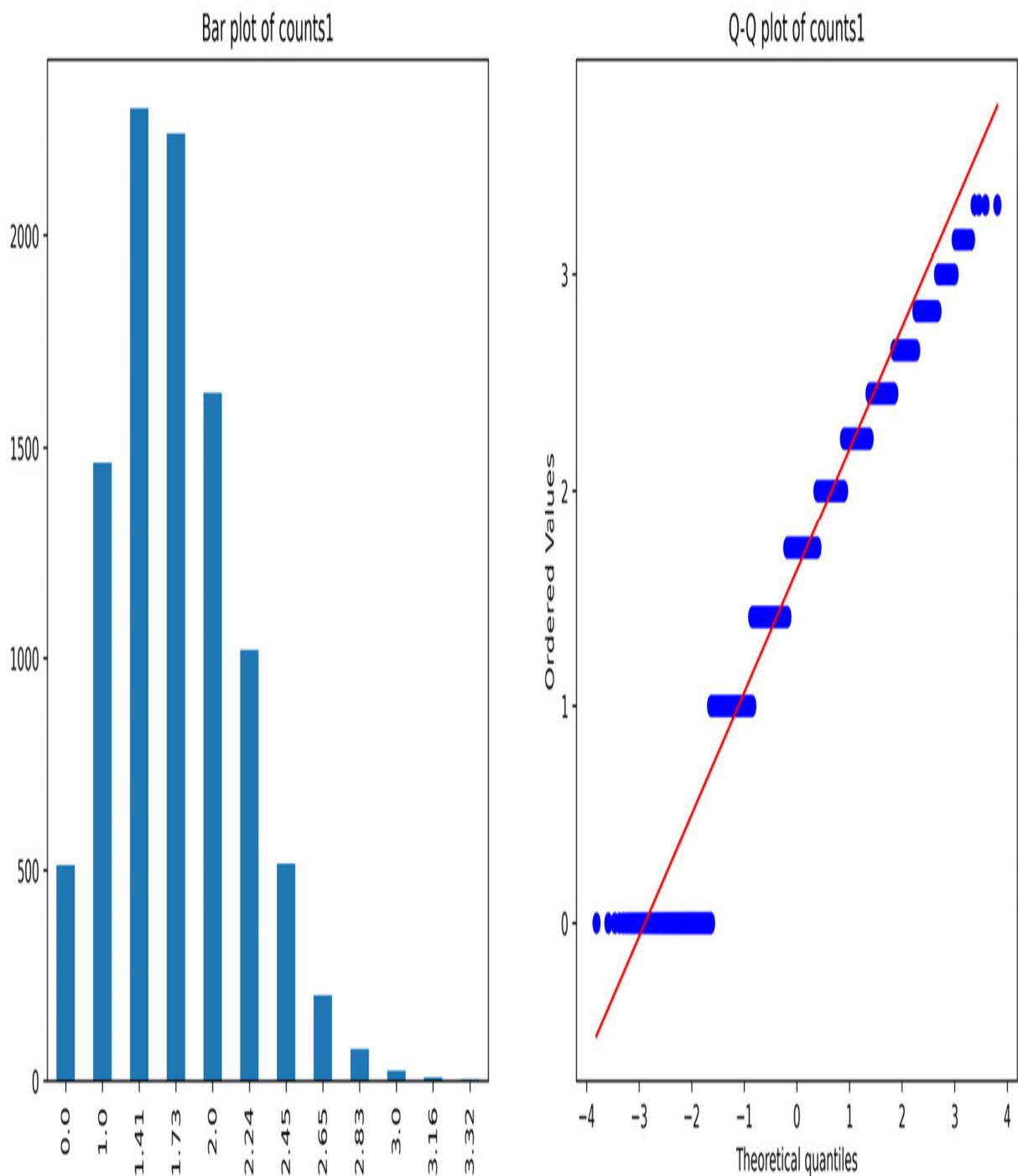


Figure 3.7 – Bar and Q-Q plots of the counts1 variable after the square root transformation

Now, let's apply the square root transformation with scikit-learn.

8. Let's import **FunctionTransformer()** and set up it to perform a square root transformation:

```
from sklearn.preprocessing import FunctionTransformer  
transformer = FunctionTransformer(np.sqrt)
```

NOTE

If we wanted to round the values as we did in *step 7*, we can set up the transformer using **transformer = FunctionTransformer(func=lambda x: np.round(np.sqrt(x), 2))**.

9. After making a copy of the data, as we did in *step 5*, we can go ahead and transform the variables:

```
df_tf = transformer.transform(df[["counts1",  
"counts2"]])
```

10. To apply the square root with Feature-engine instead, we must import **PowerTransformer()** and set it up, passing 0.5 as the exponent:

```
from feature_engine.transformation import  
PowerTransformer  
  
root_t = PowerTransformer(exp=1/2)
```

11. Next, we must fit the transformer to the data:

```
root_t.fit(df)
```

NOTE

The transformer automatically identifies the numerical variables, which we can explore by executing **root_t.variables_**.

12. Finally, we must transform the data:

```
df_tf = root_t.transform(df)
```

PowerTransformer() will return a new pandas DataFrame containing the original variables, where the variables from *step 9* are transformed with the square root function.

How it works...

In this recipe, we applied power transformations using NumPy, scikit-learn, and Feature-engine.

To apply power functions with NumPy, we created a copy of the original DataFrame with **copy()** from pandas. Next, we used the **power()** method on a slice of the dataset with the variables to transform and capture the transformed variables in the new DataFrame. This procedure returned a pandas DataFrame containing the original variables, where **MedInc** and **Population** were transformed with a power of **0.3**.

To apply an exponential transformation with scikit-learn, we used **FunctionTransformer()**, which applies a user-defined function. We set up the transformer with **np.power()** within a lambda function while using **0.3** as the exponent. The **transform()** method applied the power transformation to a slice of the dataset and the returned values were assigned to the variables in the copy of the DataFrame. This way, we captured the values directly in the DataFrame. Otherwise, scikit-learn returns the values of the NumPy array by default.

Finally, we used Feature-engine's **PowerTransformer()**. We set up the transformer with a list of the variables to transform and the exponent, which in this case is **0.3**. The **fit()** method checked that the indicated variables were numerical and that the **transform()** method applied the transformation, returning a DataFrame with the transformed variables among the original variables in the dataset.

Using power transformations

Power functions are mathematical transformations that follow

$$X_t = X^{\lambda}$$

, where lambda can take any value. The square and cube root transformations are special cases of power transformations where lambda is 1/2 or 1/3, respectively. The challenge resides in finding the value for the lambda parameter. The Box-Cox transformation, which is a generalization of the power transformations, finds the optimal lambda via maximum likelihood. We will discuss the Box-Cox transformation in the following recipe. In practice, we try different lambdas and visually inspect the variable distribution to determine which one offers the best transformation. In general, if the data is right-skewed – that is, observations accumulate toward lower values – we use lambda <1, while if the data is left-skewed – that is, there are more observations around higher values – then we use lambda >1.

In this recipe, we will carry out power transformations using NumPy, scikit-learn, and Feature-engine.

How to do it...

Let's begin by importing the libraries and getting the dataset ready:

1. Import the required Python libraries and classes:

```
import numpy as np  
  
import pandas as pd  
  
from sklearn.datasets import fetch_california_housing  
  
from sklearn.preprocessing import FunctionTransformer  
  
from feature_engine.transformation import  
PowerTransformer
```

2. Let's load the California housing dataset into a pandas DataFrame:

```
X, y = fetch_california_housing(  
    return_X_y=True, as_frame=True)
```

3. Let's make a copy of the DataFrame so that we can modify the values in the copy and not in the original, which we need for the rest of this recipe:

```
X_tf = X.copy()
```

4. To evaluate variable distributions, we'll create a function that takes a DataFrame and a variable name as inputs and plots a histogram next to a Q-Q plot:

```
def diagnostic_plots(df, variable):  
  
    plt.figure(figsize=(15, 6))  
  
    plt.subplot(1, 2, 1)  
  
    df[variable].hist(bins=30)  
  
    plt.title(f"Histogram of {variable}")
```

```
plt.subplot(1, 2, 2)

stats.probplot(df[variable], dist="norm", plot=plt)

plt.title(f"Q-Q plot of {variable}")

plt.show()
```

5. Let's plot the distribution of the **Population** variable with the function from *step 4*:

```
diagnostic_plots(x, "Population")
```

In the plots returned by the preceding command, we can see that **Population** is heavily skewed to the right:

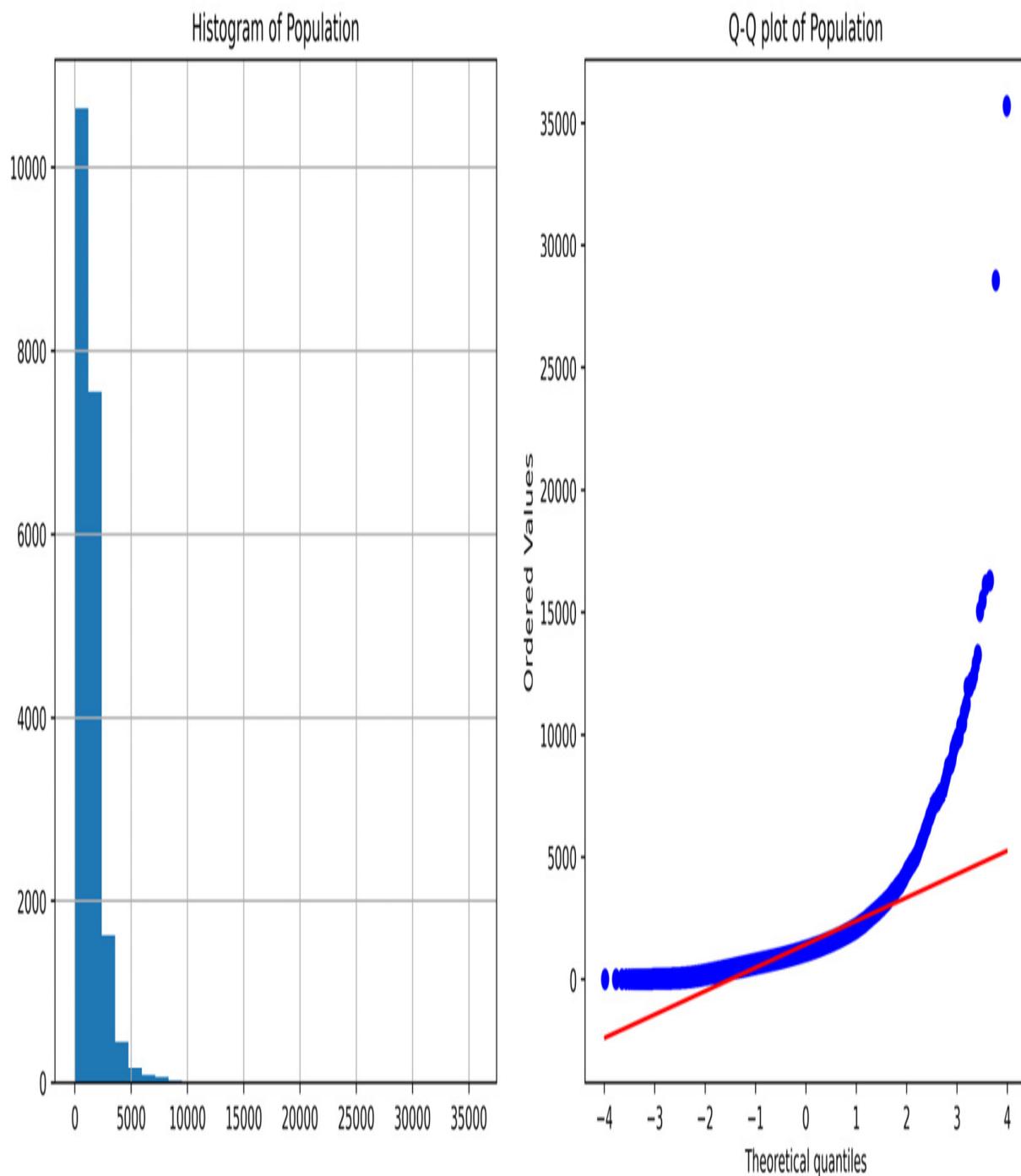


Figure 3.8 – Histogram and Q-Q plot of the Population variable

Now, let's apply a power transformation to the **MedInc** and **Population** variables. As both are skewed to the right, an exponent of <1 might return a

better spread of the variable values.

6. Let's capture the variables to transform in a list and then apply a power transformation where the exponent is **0.3**:

```
variables = ["MedInc", "Population"]
X_tf[variables] = np.power(X[variables], 0.3)
```

NOTE

With **np.power()**, we can apply any power transformation by changing the value of the exponent in the second position of the method.

7. Let's examine the change in the distribution of **Population**:

```
diagnostic_plots(X_tf, "Population")
```

As shown in the plots returned by the preceding command, **Population** is now more evenly distributed across the value range and follows the quantiles of the normal distribution more closely:

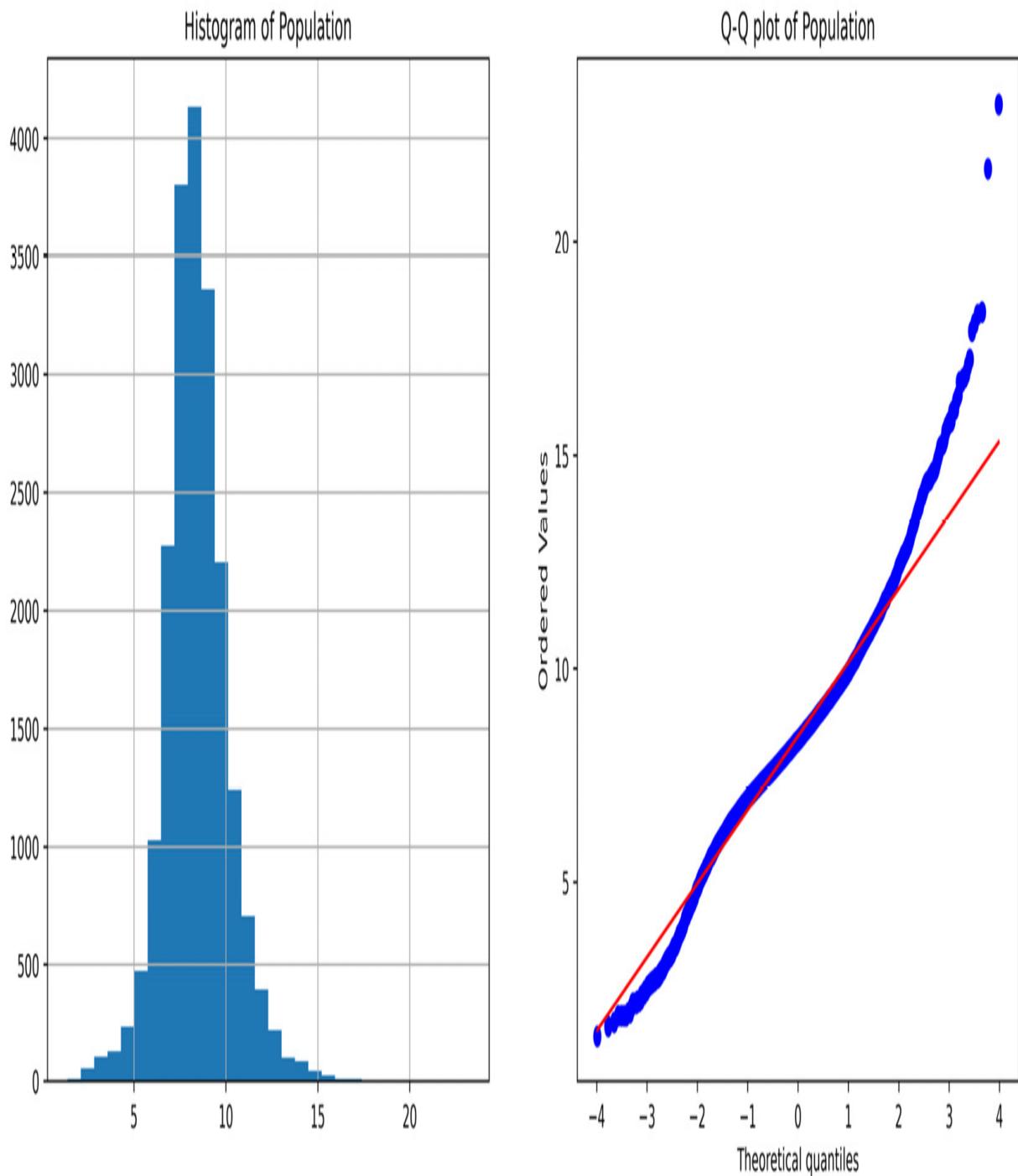


Figure 3.9 – Histogram and Q-Q plot of the Population variable after the transformation

Now, let's apply a power transformation with scikit-learn.

Let's set up **FunctionTransformer()** with a power transformation, as we did in *step 6*, within a **lambda** function:

```
transformer = FunctionTransformer(  
    lambda x: np.power(x, 0.3))
```

8. After making a copy of the data, as we did in *step 3*, we can transform the variables:

```
X_tf[variables] = transformer.transform(X[variables])
```

That's it – we can now examine the variable distribution. Finally, let's perform an exponential transformation with Feature-engine.

9. Let's set up **PowerTransformer()** with an exponent of **0.3** and the variables to transform. Then, we'll fit it to the data:

```
power_t = PowerTransformer(variables=variables,  
                           exp=0.3)  
  
power_t.fit(X)
```

NOTE

If we don't define the variables to transform, **PowerTransformer()** will select all the numerical variables in the DataFrame.

10. Finally, let's transform the variables in our dataset:

```
X_tf = power_t.transform(X)
```

The transformer returns a DataFrame containing the original variables, and the two variables specified in *step 6* are transformed with the power function.

How it works...

In this recipe, we applied power transformations using NumPy, scikit-learn, and Feature-engine.

To apply power functions with NumPy, we created a copy of the original DataFrame with **pandas copy()**. Next, we used the **power()** method on a slice of the dataset containing the variables to transform and captured the transformed variables in the new DataFrame. This procedure returned a pandas DataFrame containing the original variables, where **MedInc** and **Population** were transformed with a power of **0 . 3**.

To apply an exponential transformation with scikit-learn, we used **FunctionTransformer()**, which applies a user-defined function. We set up the transformer with **np.power()** within a **lambda** function while using **0 . 3** as the exponent. The **transform()** method applied the power transformation to a slice of the dataset and the returned values were assigned to the variables in the copy of the DataFrame. This way, we captured the values directly in the DataFrame. Otherwise, scikit-learn returns the values of the NumPy array by default.

Finally, we used Feature-engine's **PowerTransformer()**. We set up the transformer with a list of the variables to transform and an exponent of **0 . 3**. The **fit()** method checked that the indicated variables were numerical and the **transform()** method applied the transformation, returning a DataFrame with the transformed variables among the original variables in the dataset.

Performing Box-Cox transformation

The Box-Cox transformation is a generalization of the power family of transformations and is defined as follows:

$$y^{(\lambda)} = \frac{(y^\lambda - 1)}{\lambda} \text{ if } \lambda \neq 0$$

$$y^{(\lambda)} = \log(y) \text{ if } \lambda = 0$$

Here, y is the variable and λ is the transformation parameter. It includes important special cases of transformations, including untransformed ($\lambda = 1$), the logarithm ($\lambda = 0$), the reciprocal ($\lambda = -1$), the square root (when $\lambda = 0.5$, it applies a scaled and shifted version of the square root function) and the cube root.

In the Box-Cox transformation, several values of λ are evaluated using the maximum likelihood, and the λ parameter that returns the best transformation is selected.

In this recipe, we will perform Box-Cox transformation using scikit-learn and Feature-engine.

NOTE

The Box-Cox transformation can only be used on positive variables. If your variables have negative values, try the Yeo-Johnson transformation, which is described in the next recipe, *Performing Yeo-Johnson transformation*. Alternatively, you can shift the variable distribution by adding a constant before the transformation.

How to do it...

Let's begin by importing the necessary libraries and getting the dataset ready:

1. Import the required Python libraries and classes:

```
import numpy as np  
  
import pandas as pd  
  
import scipy.stats as stats  
  
from sklearn.datasets import fetch_california_housing  
  
from sklearn.preprocessing import PowerTransformer  
  
from feature_engine.transformation import  
BoxCoxTransformer
```

2. Let's load the California housing dataset into a pandas DataFrame:

```
X, y = fetch_california_housing(  
    return_X_y=True, as_frame=True)
```

3. Let's drop the **Latitude** and **Longitude** variables as we don't use them in machine learning models:

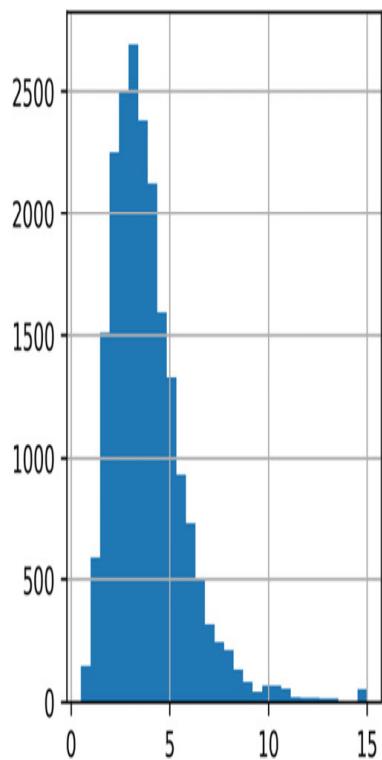
```
X.drop(labels=["Latitude", "Longitude"], axis=1,  
inplace=True)
```

4. Let's inspect the variable distributions with histograms:

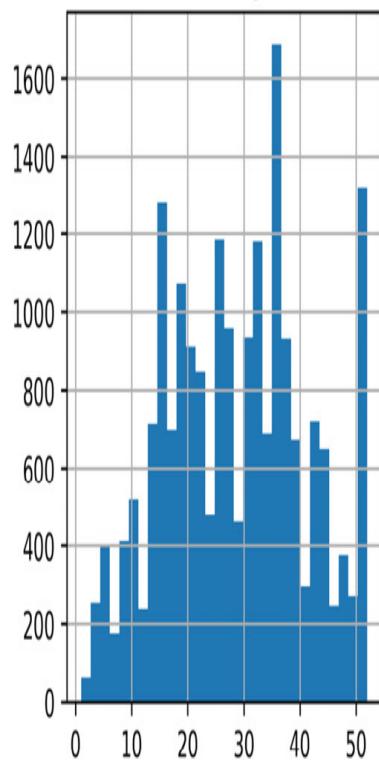
```
X.hist(bins=30, figsize=(12, 12), layout=(3, 3))  
plt.show()
```

In the following output, we can see that the **MedInc** variable shows a mild right-skewed distribution, variables such as **AveRooms** and **Population** are heavily right-skewed, and the **HouseAge** variable shows an even spread of values across its range:

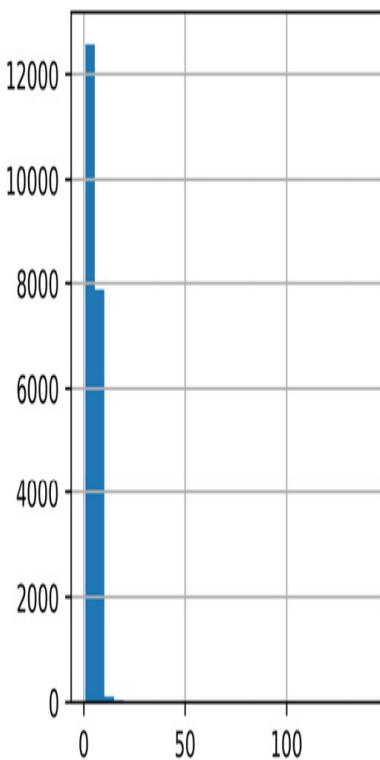
MedInc



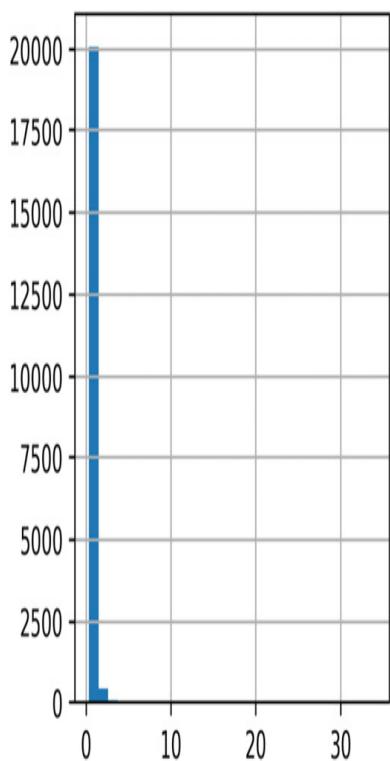
HouseAge



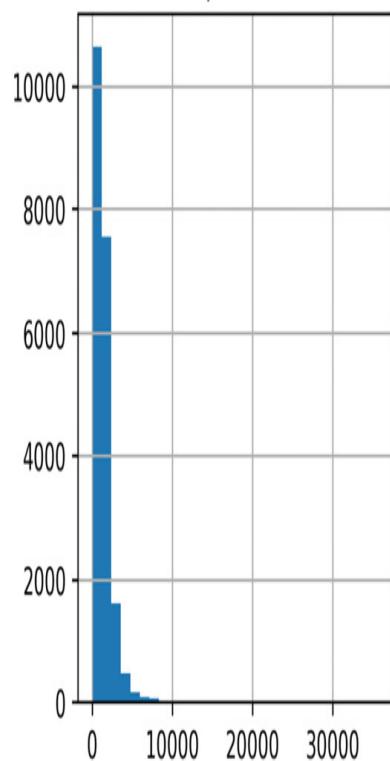
AveRooms



AveBedrms



Population



AveOccup

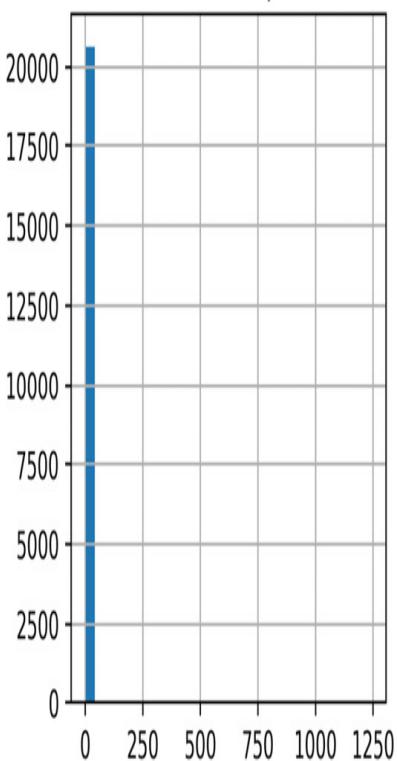


Figure 3.10 – Histograms of the numerical variables

5. Let's capture the variable names in a list since we will use these in the following step:

```
variables = list(X.columns)
```

6. Let's create a function that will plot Q-Q plots for all the variables in the data in **2** rows with **3** plots each:

```
def make_qqplot(df):  
    plt.figure(figsize=(10, 6),  
    constrained_layout=True)  
  
    for i in range(6):  
        # location in figure  
        ax = plt.subplot(2, 3, i + 1)  
        # variable to plot  
        var = variables[i]  
        # q-q plot  
        stats.probplot((df[var]), dist="norm",  
        plot=plt)  
        # add variable name as title  
        ax.set_title(var)  
  
    plt.show()
```

7. Now, let's display the Q-Q plots using the preceding function:

```
make_qqplot(X)
```

From this, we can corroborate that the variables are not normally distributed in the output of *step 7*:

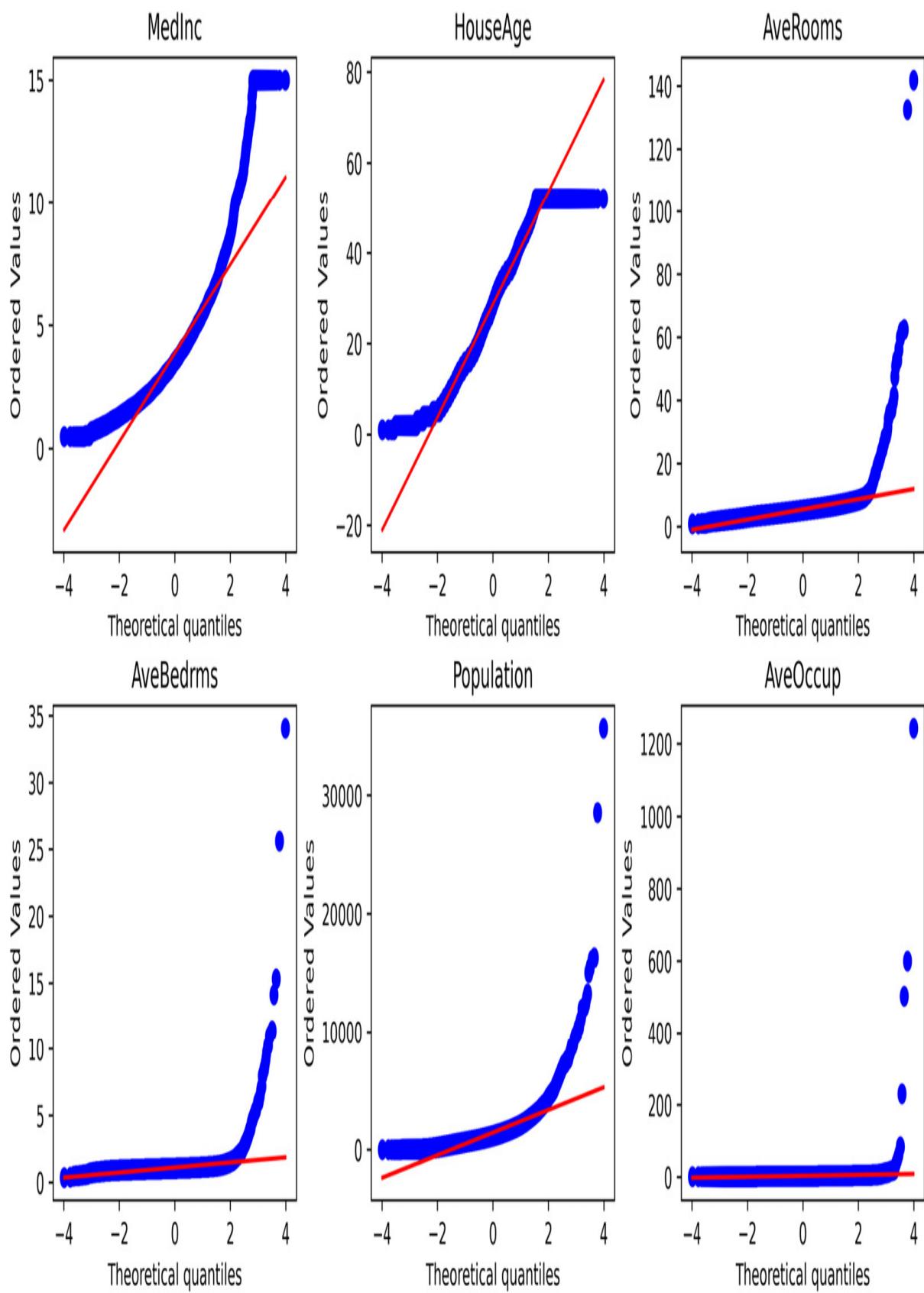


Figure 3.11 – Q-Q plots of the numerical variables

Next, let's carry out the Box-Cox transformation using scikit-learn's **PowerTransformer()**.

8. Let's set up **PowerTransformer()** so that it carries out the Box-Cox transformation and fits it to the data so that it finds the optimal λ parameter:

```
transformer = PowerTransformer()  
          method="box-cox", standardize=False)  
transformer.fit(X)
```

NOTE

The λ parameter should be learned from the train set and then used to transform the train and test sets. Thus, remember to separate your data into train and test sets before fitting **PowerTransformer()**.

9. Now, let's transform the dataset:

```
X_tf = transformer.transform(X)
```

10. Scikit-learn returns a NumPy array containing the transformed variables. Let's convert it into a pandas DataFrame:

```
X_tf = pd.DataFrame(X_tf, columns=variables)
```

NOTE

Scikit-learn's **PowerTransformer()** stores the learned lambdas in its **lambdas_** attribute, which you can display by executing **transformer.lambdas_**.

11. Let's inspect the distributions of the transformed data with histograms:

```
x_tf.hist(bins=30, figsize=(12, 12), layout=(3, 3))  
plt.show()
```

In the following output, we can see that the variables have a more even spread across their range:

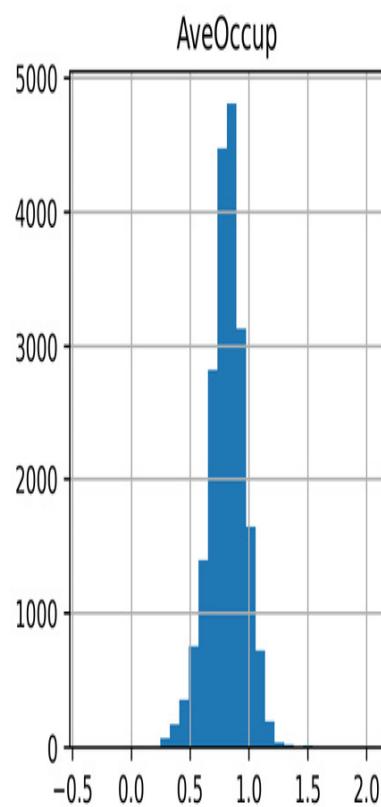
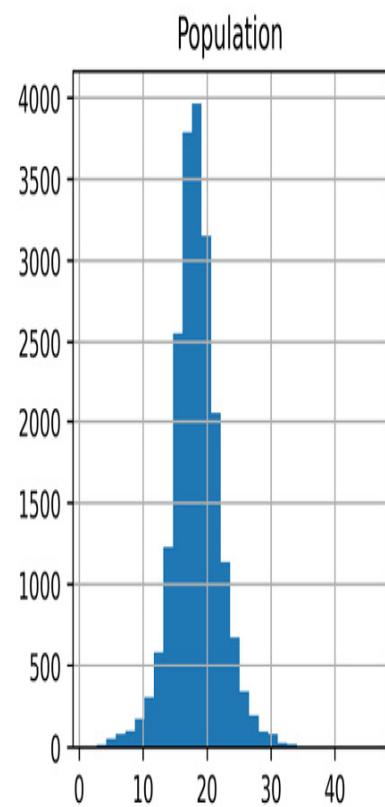
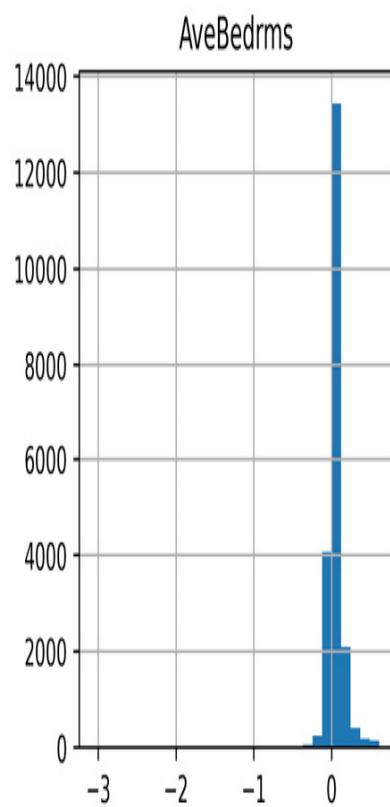
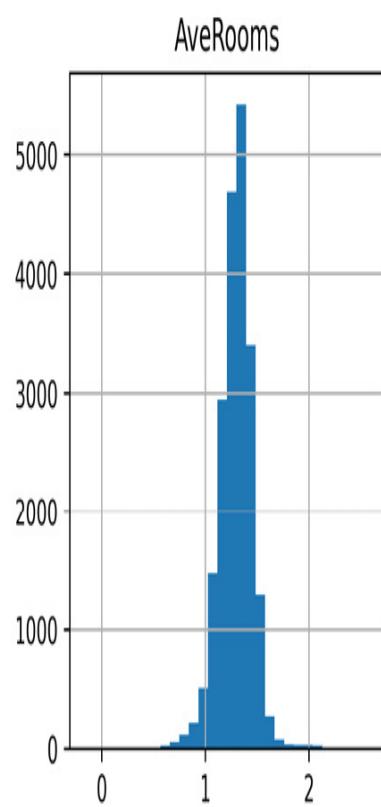
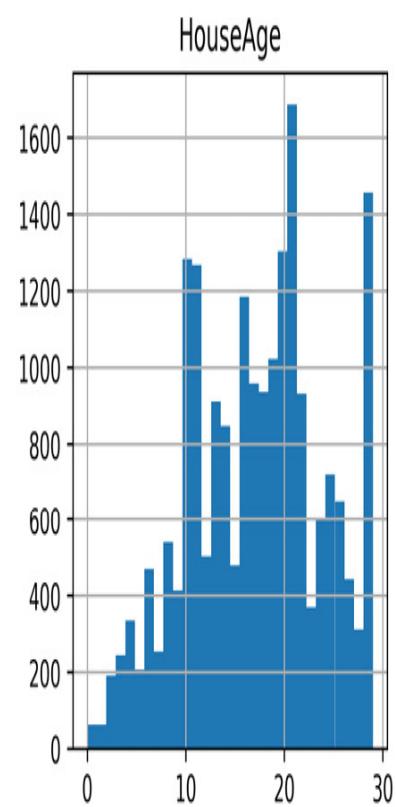
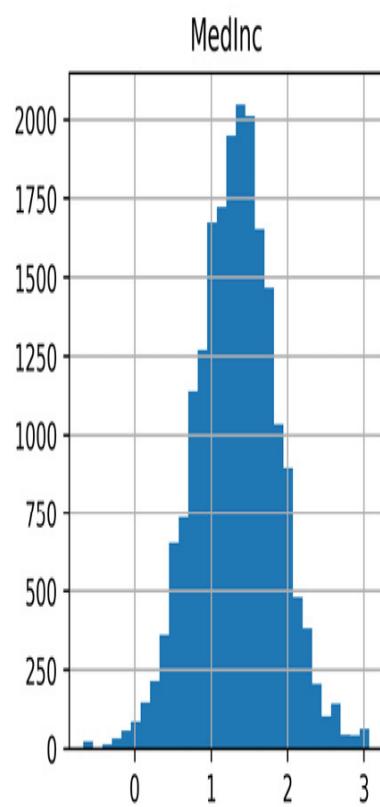


Figure 3.12 – Histograms of the variables after the transformation

12. Now, let's return Q-Q plots of the transformed variables:

```
make_qqplot(x_tf)
```

In the following output, we can see that, after the transformation, the variables follow the theoretical normal distribution more closely:

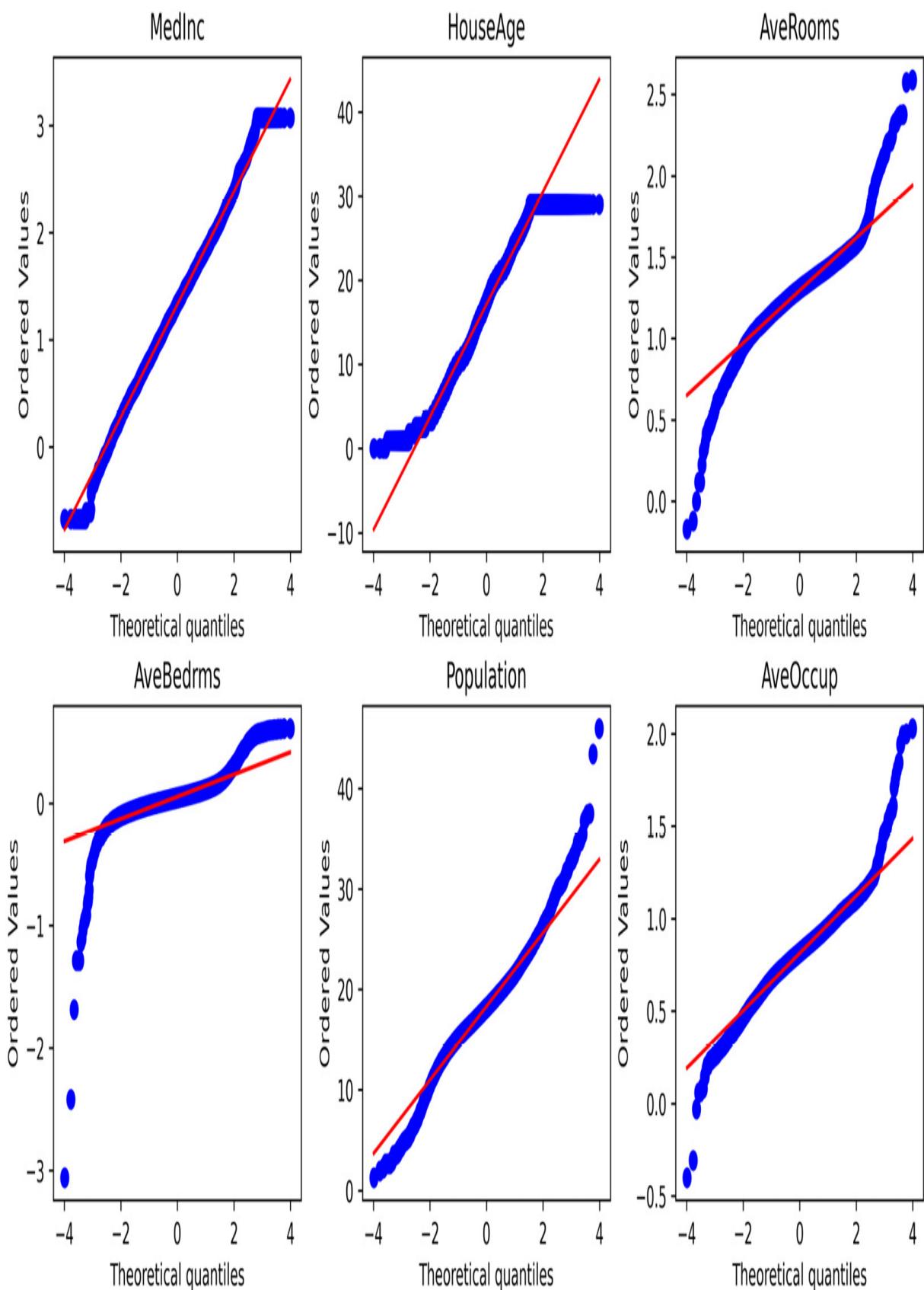


Figure 3.13 – Q-Q plots of the variables after the transformation

Now, let's implement the Box-Cox transformation with Feature-engine.

13. Let's set up **BoxCoxTransformer()** so that it transforms all the variables in the dataset, and then fits them to the data:

```
bct = BoxCoxTransformer()  
bct.fit(X)
```

14. Now, let's go ahead and transform the variables:

```
X_tf = bct.transform(X)
```

The transformation returns a pandas DataFrame containing the modified variables.

NOTE

PowerTransformer() from scikit-learn will transform the entire dataset. On the other hand, **BoxCoxTransformer()** from Feature-engine will only modify a subset of the variables: those indicated in the parameter `variables` when setting up the transformer. If `None`, the transformer automatically identifies all numerical variables.

15. The optimal lambdas for the Box-Cox transformation are stored in the `lambda_dict_` attribute. Let's inspect them:

```
bct.lambda_dict_
```

The output of the preceding line of code is as follows:

```
{'MedInc': 0.09085449361507383,  
'HouseAge': 0.8093980940712507,
```

```
'AveRooms': -0.2980048976549959,  
'AveBedrms': -1.6290002625859639,  
'Population': 0.23576757812051324,  
'AveOccup': -0.4763032278973292}
```

Now, you know how to implement the Box-Cox transformation with two different Python libraries.

How it works...

In this recipe, using scikit-learn and Feature-engine, we applied the Box-Cox transformation to a subset of variables of the California housing dataset.

Scikit-learn's **PowerTransformer()** can apply both Box-Cox and Yeo-Johnson transformations, so we specified the transformation when setting up the transformer by passing the "**box-cox**" string. Next, we fit the transformer to the data so that the transformer learned the optimal lambdas for each variable. The learned lambdas were stored in the **lambda_** attribute. Finally, we used the **transform()** method on a slice of the dataset to return the transformed variables in a NumPy array.

Finally, we applied the Box-Cox transformation using Feature-engine. We initialized **BoxCoxTransformer()**, leaving the parameter variables set the **None**. Due to this, the transformer automatically found the numerical variables in the data. We fit the transformer to the data so that it learned the optimal lambdas per variable, which were stored in **lambda_dict_**, and transformed the variables using the **transform()** method. Feature-engine's **BoxCoxTransformer()** can take the entire DataFrame as input, and it will

only transform the selected variables, returning the entire DataFrame with the selected variables transformed.

There's more...

We can apply the Box-Cox transformation with the SciPy library. For a code implementation, visit this book's GitHub repository:

<https://github.com/PacktPublishing/Python-Feature-Engineering-Cookbook-Second-Edition/blob/main/ch03-variable-transformation/Recipe-6-Yeo-Johnson-transformation.ipynb>.

Performing Yeo-Johnson transformation

The Yeo-Johnson transformation is an extension of the Box-Cox transformation that is no longer constrained to positive values. In other words, the Yeo-Johnson transformation can be used on variables with zero and negative values, as well as positive values. These transformations are defined as follows:

- $\frac{(X + 1)^\lambda - 1}{\lambda}$; if $\lambda \neq 0$ and $X \geq 0$
- $\ln(X + 1)$; if $\lambda = 0$ and $X \geq 0$
- $-\frac{(-X + 1)^{2-\lambda} - 1}{2 - \lambda}$; if $\lambda \neq 2$ and $X < 0$
- $-\ln(-X + 1)$; if $\lambda = 2$ and $X < 0$

When the variable has only positive values, then the Yeo-Johnson transformation is like the Box-Cox transformation of the variable plus one. If the variable has only negative values, then the Yeo-Johnson transformation is like the BoxCox transformation of the negative of the variable plus one at the power of 2. If the variable has a mix of positive and negative values, the Yeo-Johnson transformation applies different powers to the positive and negative values.

In this recipe, we will perform the Yeo-Johnson transformation using scikit-learn and Feature-engine.

How to do it...

Let's begin by importing the necessary libraries and getting the dataset ready:

1. Import the required Python libraries and classes:

```
import numpy as np import pandas as pd  
import scipy.stats as stats  
from sklearn.datasets import fetch_california_housing  
from sklearn.preprocessing import PowerTransformer  
from feature_engine.transformation import  
YeoJohnsonTransformer
```

2. Let's load the California housing dataset into a pandas DataFrame and then drop the **Latitude** and **Longitude** variables:

```
x, y = fetch_california_housing(  
    return_X_y=True, as_frame=True)
```

```
X.drop(labels=["Latitude", "Longitude"], axis=1,  
       inplace=True)
```

NOTE

We can evaluate the variable distribution with histograms and Q-Q plots, as we did in *steps 4 to 7* of the *Performing Box-Cox transformation* recipe.

Now, let's apply the Yeo-Johnson transformation with scikit-learn.

3. Let's set up **PowerTransformer()** with the **yeo-johnson** transformation:

```
transformer = PowerTransformer(method="yeo-johnson")
```

4. Let's fit the transformer to the data:

```
transformer.fit(X)
```

NOTE

The λ parameter should be learned from the train set and then used to transform the train and test sets. Thus, remember to separate your data into train and test sets before fitting **PowerTransformer()**.

5. Now, let's transform the dataset and return a NumPy array containing the transformed variables:

```
X_tf = transformer.transform(X)
```

NOTE

PowerTransformer() stores the learned parameters in its **lambda_** attribute, which you can return by executing **transformer.lambdas_**.

We can convert the NumPy array into a pandas DataFrame and then display the histograms and Q-Q plot of the transformed variables, as we did in *steps 10 to 12* of the *Performing Box-Cox transformation* recipe.

Finally, let's implement the Yeo-Johnson transformation with Feature-engine.

6. Let's set up **YeoJohnsonTransformer()** so that it modifies all numerical variables and then fits them to the data:

```
yjt = YeoJohnsonTransformer()  
yjt.fit(X)
```

NOTE

If the `variables` argument is left set to **None**, the transformer selects and transforms *all the numerical variables* in the dataset. Alternatively, we can pass the names of the variables to modify in a list.

Note that, compared to **PowerTransformer()** from scikit-learn, the Feature-engine's transformer can take the entire DataFrame as an argument of the `fit()` method, and yet it will only modify the selected variables.

7. Let's transform the variables:

```
X_tf = yjt.transform(X)
```

8. **YeoJohnsonTrasnformer()** stores the best parameters per variable in its `lambda_dict_` attribute, which we can display as follows:

```
yjt.lambda_dict_
```

The preceding code outputs the following dictionary:

```
{'MedInc': -0.1985098937827175,  
'HouseAge': 0.8081480895997063,  
'AveRooms': -0.5536698033957893,  
'AveBedrms': -4.3940822236920365,  
'Population': 0.23352363517075606,  
'AveOccup': -0.9013456270549428}
```

Now, you know how to implement the Yeo-Johnson transformation with two different open source libraries.

How it works...

In this recipe, we applied the Yeo-Johnson transformation using scikit-learn and Feature-engine.

scikit-learn's **PowerTransformer()** can apply both Box-Cox and Yeo-Johnson transformations, so we specified the transformation with the **yeo-johnson** string. The **standardize** argument allowed us to determine whether we wanted to standardize (scale) the transformed values. Next, we fit the transformer to the DataFrame so that the transformer learned the optimal lambdas for each variable. **PowerTransformer()** stored the learned lambdas in its **lambdas_** attribute. Finally, we used the **transform()** method to return the transformed variables in a NumPy array.

After that, we applied the Yeo-Johnson transformation using Feature-engine. We set up **YeoJohnsonTransformer()** so that it transforms all numerical variables in the data. We fitted the transformer to the data so that

it learned the optimal lambdas per variable, which were stored in `lambda_dict_`, and finally transformed the variables using the `transform()` method. Feature-engine's `YeoJohnsonTransformer()` can take the entire DataFrame as input, yet it will only transform the selected variables, returning a new pandas DataFrame.

There's more...

We can apply the Yeo-Johnson transformation with the SciPy library. For a code implementation, visit this book's GitHub repository:

<https://github.com/PacktPublishing/Python-Feature-Engineering-Cookbook-Second-Edition/blob/main/ch03-variable-transformation/Recipe-6-Yeo-Johnson-transformation.ipynb>.

4

Performing Variable Discretization

Discretization is the process of transforming continuous variables into discrete features by creating a set of contiguous intervals, also called bins, that span the range of the variable values. Subsequently, these intervals or bins are treated as categorical data.

Many machine learning models, such as decision trees and Naïve Bayes, work better with discrete attributes. In fact, decision tree-based models make decisions based on discrete partitions over the attributes. During induction, a decision tree evaluates all possible feature values to find the best cut-point. Therefore, the more values the feature has, the longer the induction time of the tree. In this sense, discretization can improve model performance and reduce the time it takes to train the models.

Discretization has additional advantages. Data is reduced and simplified; discrete features can be easier to understand by domain experts.

Discretization can change the distribution of skewed variables; if sorting observations across bins with equal frequency, the values are spread more homogeneously across the range. Additionally, discretization can minimize the influence of outliers by placing them at lower or higher intervals, together with the remaining *inlier* values of the distribution. Overall, discretization reduces and simplifies data, making the learning process faster and yielding more accurate results.

Discretization can also lead to a loss of information, for example, by combining values that are strongly associated with different classes or target values into the same bin. Therefore, the aim of a discretization algorithm is to find the minimal number of intervals without a significant loss of information. In practice, many discretization procedures require the user to input the number of intervals into which the values will be sorted. Then, the job of the algorithm is to find the cut-points for those intervals. Among these procedures, we find the most widely used equal-width and equal-frequency discretization methods. Discretization methods based on decision trees otherwise are able to find the optimal number of partitions, as well as the cut-points.

Discretization procedures can be classified as supervised and unsupervised. Unsupervised discretization methods do not use any information, other than the variable distribution, to create the contiguous bins. On the other hand, supervised methods use target information to create the intervals.

In this chapter, we will discuss widely used supervised and unsupervised discretization procedures that are available in established open source libraries. Among these, we will cover equal-width, equal-frequency, arbitrary, k-means, and decision tree-based discretization. More elaborate methods, such as ChiMerge and CAIM, are out of scope, as their implementation is not yet available open source.

This chapter will cover the following recipes:

- Performing equal-width discretization
- Implementing equal-frequency discretization
- Discretizing the variable into arbitrary intervals

- Performing discretization with k-means clustering
- Implementing feature binarization
- Using decision trees for discretization

Technical requirements

In addition to the numerical computing libraries such as **pandas**, **NumPy**, **Matplotlib**, **scikit-learn**, and **Feature-engine**, we will also use the **Yellowbrick** library. You can install **Yellowbrick** with **pip**:

```
pip install yellowbrick
```

For more details about **Yellowbrick**, please visit the documentation at <https://www.scikit-yb.org/en/latest/index.xhtml>.

Performing equal-width discretization

Equal-width discretization is the simplest discretization method, which consists of dividing the range of observed values for a variable into k equally sized intervals, where k is supplied by the user. The interval width for the X variable is given by the following:

$$\text{Width} = \frac{\text{Max}(X) - \text{Min}(X)}{k}$$

Then, if the values of the variable vary between 0 and 100, we can create five bins like this: $\text{width} = (100-0) / 5 = 20$; the bins will be 0–20, 20–40, 40–60, and 80–100. The first and final bins (0–20 and 80–100) can be

expanded to accommodate values smaller than 0 or greater than 100, by extending the limits to minus and plus infinity.

In this recipe, we will carry out equal-width discretization using **pandas**, **scikit-learn**, and **Feature-engine**.

How to do it...

First, let's import the necessary Python libraries and get the dataset ready:

1. Import the Python libraries and the data:

```
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
from sklearn.datasets import fetch_california_housing  
from sklearn.model_selection import train_test_split
```

2. Let's load the predictor and target variables of the California housing dataset:

```
x, y = fetch_california_housing(  
    return_X_y=True, as_frame=True)
```

NOTE

The cut-points for the intervals should be learned using variables in the train set only and then they should be used to discretize the variables in the train and test sets.

3. Let's divide the data into train and test sets:

```
x_train, x_test, y_train, y_test = train_test_split(
```

```
x, y, test_size=0.3, random_state=0,  
)
```

Next, we will divide the **HouseAge** continuous variable into 10 intervals using **pandas** and the formula described at the beginning of the recipe.

4. Let's capture the minimum and maximum values of **HouseAge**:

```
var = "HouseAge"  
  
min_value = int(X_train[var].min())  
  
max_value = int(X_train[var].max())
```

5. Let's determine the interval width, which is the variable's value range divided by the number of bins:

```
width = int((max_value - min_value) / 10)
```

If we execute **print(width)**, we will obtain **5**, which is the size of the intervals.

6. Now we need to define the interval limits in a list:

```
interval_limits = [i for i in range(  
    min_value, max_value, width)]
```

If we now execute **print(interval_limits)**, we will see the interval limits:

```
[1, 6, 11, 16, 21, 26, 31, 36, 41, 46, 51]
```

7. Let's expand the limits of the first and last interval to accommodate smaller or greater values:

```
interval_limits[0] = -np.inf
```

```
interval_limits[-1] = np.inf
```

8. To divide a **pandas** series into intervals, we will use the **pandas cut()** method using the interval limits from *step 6*, for both the train and test sets:

```
var_disc = "HouseAge_disc"  
  
X_train[var_disc] = pd.cut(  
  
    x=X_train[var], bins=interval_limits,  
  
    include_lowest=True)  
  
X_test[var_disc] = pd.cut(  
  
    x=X_test[var], bins=interval_limits,  
  
    include_lowest=True)
```

TIP

We set **include_lowest=True** to include the lowest value in the first interval. Through the parameter labels, we can return the bins as integers (default), as interval limits, or replace them with user-specified labels.

9. Let's print the top **10** observations of the discretized and original variables, side by side:

```
print(X_train[[var, var_disc]].head(10))
```

In following the output, we can see that the value of **52** was allocated to the 46–infinite interval, the value of **43** was allocated to the 41–46 interval, and so on:

	HouseAge	HouseAge_disc	1989	52.0	(46.0,
inf]256	43.0	(41.0, 46.0]	7887	17.0	(16.0,
21.0]4581	17.0	(16.0, 21.0]	1993	50.0	(46.0,
inf]10326	11.0	(6.0, 11.0]	10339	14.0	(11.0,
16.0]12992	17.0	(16.0, 21.0]	10458	6.0	(-inf,
6.0]1700	10.0	(6.0, 11.0]			

NOTE

The parentheses and brackets in the intervals denote whether a value has been included or not in the interval. For example, the **(41, 46]** interval contains all values greater than 41 and smaller than or equal to 46.

In equal-width discretization, a different number of observations is allocated to each interval.

10. Let's make a bar plot with the proportion of observations across the **HouseAge** intervals in the train and test sets:

```
# determine proportion of observations per bin
t1 = X_train[var_disc].value_counts(normalize=True,
sort=False)

t2 = X_test[var_disc].value_counts(normalize=True,
sort=False)

# concatenate proportions
tmp = pd.concat([t1, t2], axis=1)

tmp.columns = ['train', 'test']

# plot
tmp.plot.bar(figsize=(8,5))
```

```

plt.xticks(rotation=45)
plt.ylabel('Number of observations per bin')
plt.title("HouseAge")
plt.show()

```

In the following output, we can see that the proportion of observations per interval is approximately the same in the train and test sets, but different across intervals:

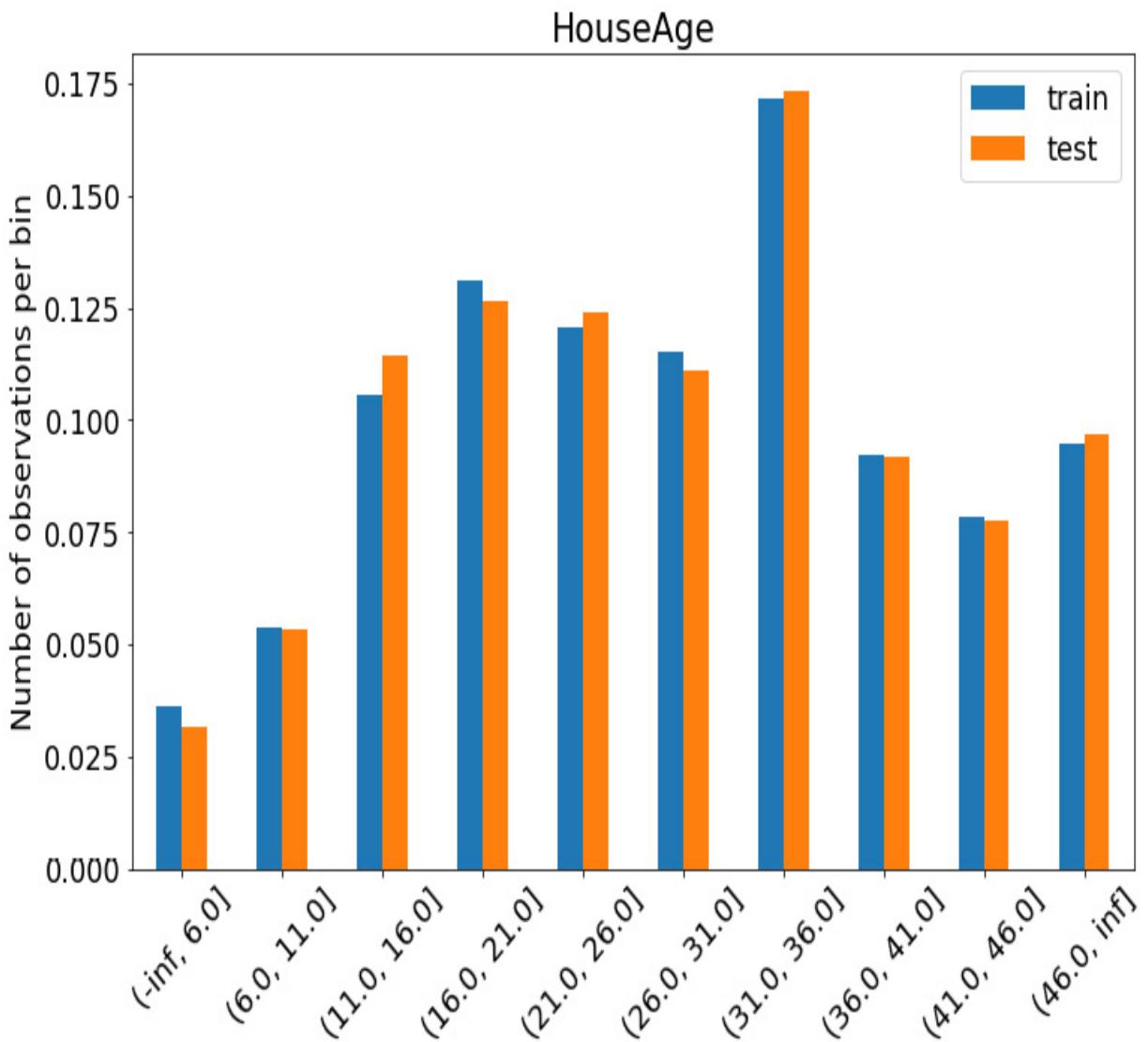


Figure 4.1 – Proportion of observations per interval

With **Feature-engine**, we can perform equal-width discretization in fewer lines of code and for many variables at a time.

11. First, let's divide the data into train and test sets, as shown in *step 3*:

```
x_train, X_test, y_train, y_test = train_test_split(  
    X,  
    y,  
    test_size=0.3,  
    random_state=0,  
)
```

12. Next, let's import the discretizer:

```
from feature_engine.discretisation import  
EqualWidthDiscretiser
```

13. Let's create an equal-width discretizer to sort three continuous variables into eight intervals, thus returning the interval limits after the transformation:

```
variables = ['MedInc', 'HouseAge', 'AveRooms']  
disc = EqualWidthDiscretiser(  
    bins=8,  
    variables=variables,  
    return_boundaries=True,  
)
```

NOTE

The **EqualWidthDiscretiser()** transformer returns an integer indicating whether the value was sorted in the first, second, or eighth bin, by default. That is the equivalent of ordinal encoding, which we described in the *Replacing categories with ordinal numbers* recipe of [Chapter 2, Encoding Categorical Variables](#). We can set **return_object=True** to return the variables as objects and then use any of the **Feature-engine** or **Category Encoders** transformers. Alternatively, you can return the interval limits as we did in *step 13*.

14. Let's fit the discretizer to the train set so that it learns the cut-points for each variable:

```
disc.fit(X_train)
```

We can inspect the cut-points in the **disc.binner_dict_** attribute. Note that **Feature-engine** will automatically extend the limits of the lower and upper intervals to infinite, to accommodate potential outliers in future data.

15. Let's discretize the variables in the train and test sets:

```
train_t = disc.transform(X_train)  
test_t = disc.transform(X_test)
```

EqualWidthDiscretiser() returns a dataframe where the indicated variables are discretized. If we run **test_t.head()** from a Jupyter notebook, we will see the following output where the original variable values are replaced by the intervals:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
14740	(4.125, 5.937]	(20.125, 26.5]	(-inf, 17.307]	1.075472	1551.0	4.180593	32.58	-117.05
10101	(4.125, 5.937]	(26.5, 32.875]	(-inf, 17.307]	0.927739	1296.0	3.020979	33.92	-117.97
20566	(4.125, 5.937]	(26.5, 32.875]	(-inf, 17.307]	1.026217	1554.0	2.910112	38.65	-121.84
2670	(2.312, 4.125]	(32.875, 39.25]	(-inf, 17.307]	1.316901	390.0	2.746479	33.20	-115.60
15709	(4.125, 5.937]	(20.125, 26.5]	(-inf, 17.307]	1.039578	649.0	1.712401	37.79	-122.43

Figure 4.2 – The dataframe with the discretized variables

16. Now, let's make bar plots with the proportion of observations per interval to better understand the effect of equal-width discretization:

```
plt.figure(figsize=(6, 12), constrained_layout=True)
for i in range(3):
    # location of plot in figure
    ax = plt.subplot(3, 1, i + 1)
    # the variable to plot
    var = variables[i]
    # determine proportion of observations per bin
    t1 = train_t[var].value_counts(normalize=True,
                                    sort=False)
    t2 = test_t[var].value_counts(normalize=True,
                                 sort=False)
```

```
# concatenate proportions

tmp = pd.concat([t1, t2], axis=1)

tmp.columns = ['train', 'test']

# sort the intervals

tmp.sort_index(inplace=True)

# make plot

tmp.plot.bar(ax=ax)

plt.xticks(rotation=45)

plt.ylabel('Observations per bin')

# add variable name as title

ax.set_title(var)

plt.show()
```

The intervals contain a different number of observations, as shown in the following plots:

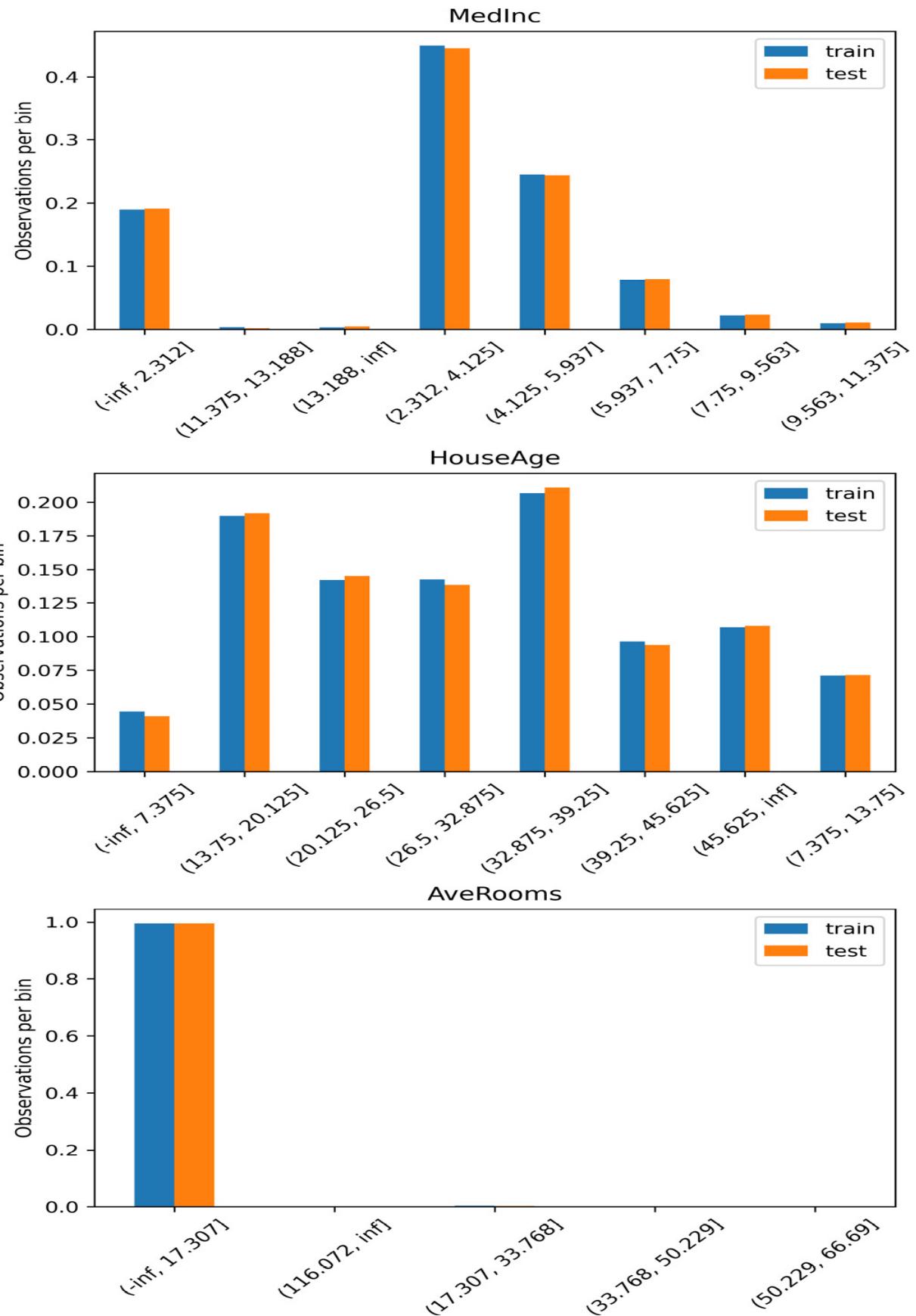


Figure 4.3 – Bar plots with the proportion of observations per interval

Now, let's implement equal-width discretization with **scikit-learn**.

17. Import classes from **scikit-learn**:

```
from sklearn.compose import ColumnTransformer  
from sklearn.preprocessing import KBinsDiscretizer
```

18. Let's set up an equal-width discretizer by setting its **strategy** to **uniform**:

```
disc = KBinsDiscretizer(n_bins=8, encode='ordinal',  
strategy='uniform')
```

NOTE

The **KBinsDiscretiser()** class can return the bins as integers by setting **encoding='ordinal'** or one-hot encoding by setting **encoding='onehot-dense'**.

19. Let's use **ColumnTransformer()** to apply the discretization to a slice of the dataframe with the selected variables:

```
ct = ColumnTransformer(  
[("discretizer", disc, variables)],  
remainder="passthrough",  
)
```

NOTE

The **remainder** parameter is used to indicate whether we want to keep all the variables in the dataframe after the transformation, in which case we use **passthrough**, or only the transformed ones, in which case we use **drop**.

20. Let's fit the discretizer to the train set so that the transformer learns the interval limits for each variable:

```
ct.fit(X_train)
```

21. Finally, let's transform the train and test sets:

```
train_t = ct.transform(X_train)  
test_t = ct.transform(X_test)
```

We can inspect the cut-points learned by the transformer by executing

```
ct.named_transformers_["discretizer"].bin_edges_.
```

Remember that **scikit-learn** returns **NumPy** arrays.

22. Let's convert the array into a **pandas** dataframe:

```
output_vars = variables + ['AveBedrms', 'Population',  
'AveOccup', "Latitud", "Longitude"]  
  
train_t = pd.DataFrame(train_t, columns = output_vars)  
test_t = pd.DataFrame(test_t, columns = output_vars)
```

That's it. Now we can make the bar plots as shown in *step 14* or use the data to train machine learning models.

How it works...

In this recipe, we sorted the variable values into equidistant intervals. To perform discretization with **pandas**, we calculated the maximum and minimum values of the **HouseAge** variable using the **pandas** methods of **max()** and **min()**, capturing the results as integers. Then, we estimated the interval width by dividing the value range, that is, the maximum values minus the minimum values, by the number of arbitrary bins. With the width and the minimum and maximum values, we captured the interval limits in a list. We used this list with **pandas cut()** to allocate the variable values into the intervals.

After discretization, the intervals are normally treated as categorical values. By default, **pandas cut()** returns the interval values as ordered integers, which is the equivalent of ordinal encoding. Through the **labels** parameter, we can modify this behavior to return the interval boundaries or user-defined labels.

We created a bar plot with the fraction of observations per interval. We used **pandas value_counts()** to obtain the fraction of observations per interval. To plot these proportions, first, we concatenated the train and test series using **pandas concat()** in a temporary dataframe, and then we assigned the column names of **train** and **test** to it. Finally, we used pandas' **plot.bar()** to display a bar plot. We rotated the labels with **Matplotlib's xticks()** method and added the y legend with **ylabel()** and the title with **title()**.

To perform equal-width discretization with **Feature-engine**, we used **EqualWidthDiscretiser()**, which takes the number of bins and the variables to discretize as arguments. Using the **fit()** method, the

discretizer learned the interval limits for each variable. With the `transform()` method, the discretizer sorted the values into each bin.

`EqualWidthDiscretiser()` offers a few options to follow up the discretization with encoding. By default, it will return the bins as sorted integers, which is the equivalent of ordinal encoding. Alternatively, it can return the interval boundaries as with `pandas cut()`. To follow up the discretization with any other encoding procedure available in `Feature-engine` or `Category Encoders`, we need to return the bins cast as objects, which can be achieved by setting `return_object` to `True`.

We followed the discretization with bar plots to identify the fraction of observation per interval, for each of the transformed variables. Because the `HouseAge` variable had a fairly homogeneous value spread before the transformation, with the equal-width discretization, we allocated values to all the intervals. However, some of the intervals of the `MedInc` and `AveRooms` variables, which had skewed distributions, contained very few observations. In particular, even though we wanted to create eight bins for `AveRooms`, there were enough values to create only five, and most values of the variables were allocated to the first interval. Therefore, equal-width discretization for very skewed variables incurs an important loss of information.

Finally, we discretized three continuous variables into equal-width bins with `KBinsDiscretizer()` from `scikit-learn`, indicating the number of bins and setting `strategy` to `uniform`. With the `fit()` method, the transformer learned the limits of the intervals, and with the `transform()` method, the transformer sorted the values into each interval, returning a `NumPy` array with the discretized variables. The values of the discretized

variables were integers representing the intervals. `KBinsDiscretizer` can return the intervals as ordinal numbers or as one-hot-encoded variables. The behavior can be modified through the `encode` parameter.

To apply the discretization to a subset of variables, we used `ColumnTransformer()`, which sliced the data into the selected variables, then applied the discretizer to this slice of data, and finally concatenated the remaining variables to the transformed ones in a `NumPy` array.

See also

For a comparison of equal-width discretization with more sophisticated methods, see Dougherty J, Kohavi R, Sahami M. *Supervised and unsupervised discretization of continuous features*. In: Proceedings of the 12th international conference on machine learning. San Francisco: Morgan Kaufmann; 1995. p. 194–202.

Implementing equal-frequency discretization

Equal-width discretization is easy to compute. However, if the variables are skewed, then there will be many empty bins or bins with only a few values, while most observations will be allocated to a few intervals. This could result in a loss of information. This problem can be solved by adaptively finding the interval cut-points so that each interval contains a similar fraction of observations.

Equal-frequency discretization divides the values of the variable into intervals that carry the same proportion of observations. The interval width

is determined by quantiles. Quantiles are values that divide data into equal portions. For example, the median is a quantile that divides the data into two halves. Quartiles divide the data into 4 portions, and percentiles divide the data into 100 portions. As a result, the intervals will most likely have different widths. The number of intervals is defined by the user.

Equal-frequency discretization is particularly useful for skewed variables as it spreads the observations over the different bins equally. In this recipe, we will perform equal-frequency discretization using **pandas**, **scikit-learn**, and **Feature-engine**.

How to do it...

First, let's import the necessary Python libraries and get the dataset ready:

1. Import the required Python libraries and classes:

```
import pandas as pd  
  
import matplotlib.pyplot as plt  
  
from sklearn.datasets import fetch_california_housing  
  
from sklearn.model_selection import train_test_split
```

2. Let's load the California housing dataset into a dataframe:

```
X, y = fetch_california_housing(  
    return_X_y=True, as_frame=True)
```

The interval boundaries or quantiles should be learned using variables in the train set only.

3. Let's divide the data into train and test sets:

```
x_train, x_test, y_train, y_test = train_test_split(  
    X, y, test_size=0.3, random_state=0,  
)
```

Let's implement equal-frequency discretization with **pandas**.

4. To divide the **HouseAge** variable into eight quantiles, we use **pandas qcut()**, which returns both, the discretized variable and the quantile limits. We capture them as a column of the dataframe and an individual variable, respectively:

```
var = "HouseAge"  
  
var_disc = "House_disc"  
  
x_train[var_disc], interval_limits = pd.qcut(  
    x=x_train[var], q=8, labels=None, retbins=True,  
)
```

We can visualize the cut-points for the intervals by executing
print(interval_limits):

```
array([ 1., 14., 18., 24., 29., 34., 37., 44., 52.])
```

5. Let's print the top 10 observations of the discretized and original variables, side by side:

```
print(x_train[[var, var_disc]].head(10))
```

In the following output, we can see that the value of **52** was allocated to the 44–52 interval, the value of **43** was allocated to the 37–44 interval, and so on:

	HouseAge	House_disc
1989	52.0	(44.0, 52.0]
256	43.0	(37.0, 44.0]
7887	17.0	(14.0, 18.0]
4581	17.0	(14.0, 18.0]
1993	50.0	(44.0, 52.0]
10326	11.0	(0.999, 14.0]
10339	14.0	(0.999, 14.0]
12992	17.0	(14.0, 18.0]
10458	6.0	(0.999, 14.0]
1700	10.0	(0.999, 14.0]

6. Now, let's discretize **HouseAge** in the test set, using **pandas cut()** with the interval limits determined in *step 4*:

```
X_test[var_disc] = pd.cut(
    x = X_test[var], bins=interval_limits)
```

In equal-frequency discretization, similar fractions of observations are allocated to each interval.

7. Let's make a bar plot with the proportion of observations across the **HouseAge** intervals in the train and test sets:

```
# determine proportion of observations per bin
t1 = X_train[var_disc].value_counts(normalize=True)
t2 = X_test[var_disc].value_counts(normalize=True)
```

```
# concatenate proportions  
tmp = pd.concat([t1, t2], axis=1)  
tmp.columns = ['train', 'test']  
tmp.sort_index(inplace=True)  
  
# plot  
tmp.plot.bar()  
plt.xticks(rotation=45)  
plt.ylabel('Number of observations per bin')  
plt.title("HouseAge")  
plt.show()
```

In the output of the preceding code block, we can see that most bins contain a similar fraction of observations, as expected:

HouseAge

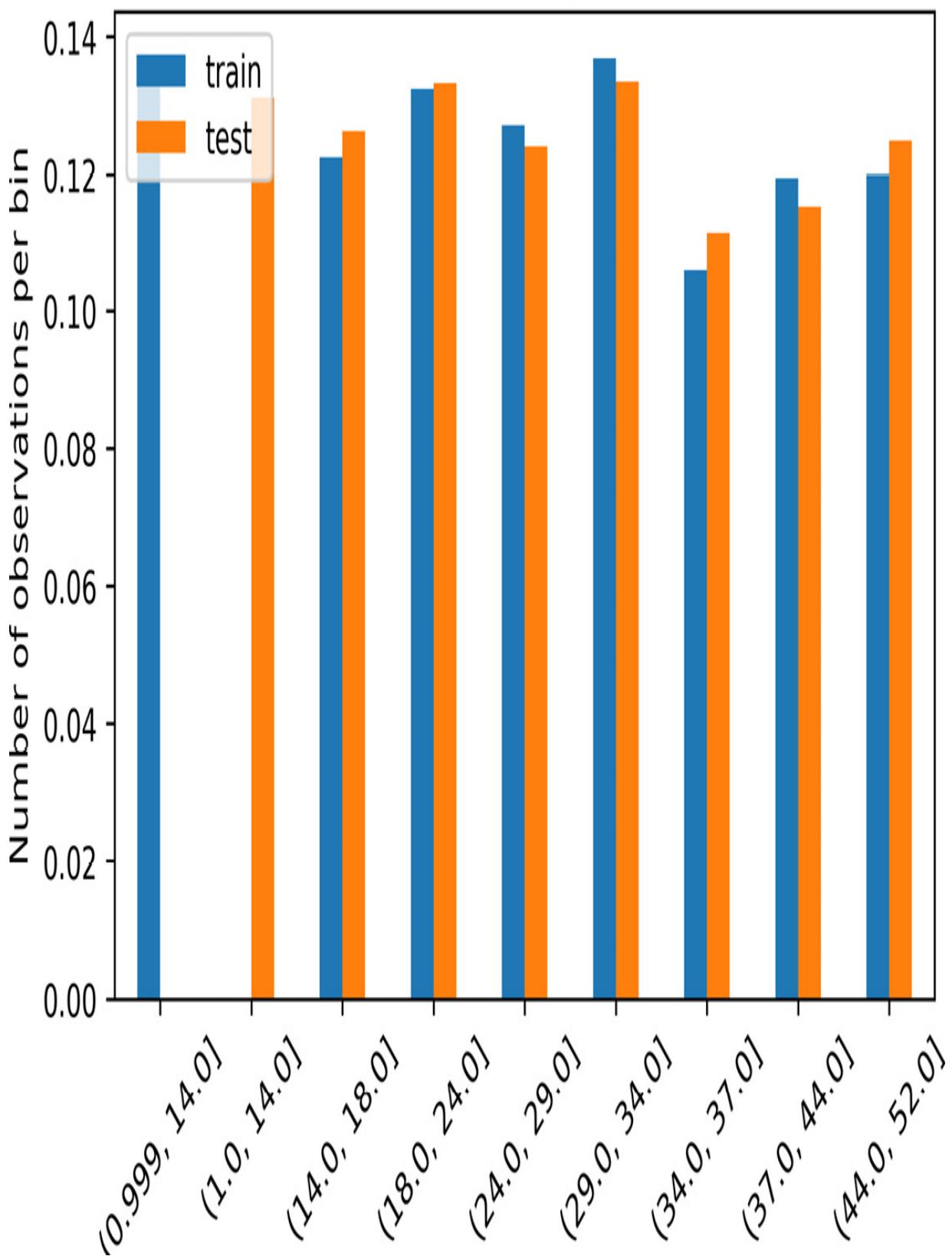


Figure 4.4 – The proportion of observations per interval of HouseAge

NOTE

Note that the limits for the first interval in the test set and train set differ, being 0.99–14 and 1–14, respectively. This occurred because the minimum value observed in the train set was 1, whereas, in the test set, it was smaller than 1. Therefore, the cut-point was modified to accommodate all values. Alternatively, we can set the limits of the first and last intervals to minus and plus infinity, as we did in the *Performing equal width discretization* recipe.

With **Feature-engine**, we can perform equal frequency for many variables at the same time. First, let's divide the data into train and test sets, as shown in step 3. Next, let's set up a discretizer.

8. Let's import the discretizer:

```
from feature_engine.discretisation import  
EqualFrequencyDiscretiser
```

9. Let's set up the transformer to discretize three continuous variables into eight quantiles, returning the interval limits after the transformation:

```
variables = ['MedInc', 'HouseAge', 'AveRooms']  
  
disc = EqualFrequencyDiscretiser(  
  
    q=8, variables =variables,  
  
    return_boundaries=True)
```

10. Let's fit the discretizer to the train set so that it learns the interval limits:

```
disc.fit(X_train)
```

The transformer stores the limits of the intervals for each variable in a dictionary in its `disc.binner_dict_` attribute. **Feature-engine** will automatically extend the limits of the lower and upper intervals to infinite, to accommodate potential outliers in future data.

11. Let's transform the variables in the train and test sets:

```
train_t = disc.transform(X_train)  
test_t = disc.transform(X_test)
```

NOTE

The **EqualFrequencyDiscretiser()** class returns an integer indicating whether the value was sorted in the first, second, or eighth bin, by default. That is the equivalent of ordinal encoding, which we described in the *Replacing categories with ordinal numbers* recipe of [Chapter 2, Encoding Categorical Variables](#). If we want to follow up the discretization with a different type of encoding, we can set `return_object=True` to return the variables cast as objects and then use any of the **Feature-engine** or **Category Encoders** transformers. Alternatively, we can return the interval limits, as we did in *step 9*.

12. Now, let's make bar plots with the fraction of observations per interval to better understand the effect of equal-frequency discretization:

```
plt.figure(figsize=(6, 12), constrained_layout=True)  
for i in range(3):  
    # location of plot in figure  
    ax = plt.subplot(3, 1, i + 1)  
    # the variable to plot
```

```
var = variables[i]

# determine proportion of observations per bin
t1 = train_t[var].value_counts(normalize=True)
t2 = test_t[var].value_counts(normalize=True)

# concatenate proportions
tmp = pd.concat([t1, t2], axis=1)

tmp.columns = ['train', 'test']

# sort the intervals
tmp.sort_index(inplace=True)

# make plot
tmp.plot.bar(ax=ax)

plt.xticks(rotation=45)
plt.ylabel("Observations per bin")

# add variable name as title
ax.set_title(var)

plt.show()
```

The intervals have a similar proportion of observations, as shown in the following output:

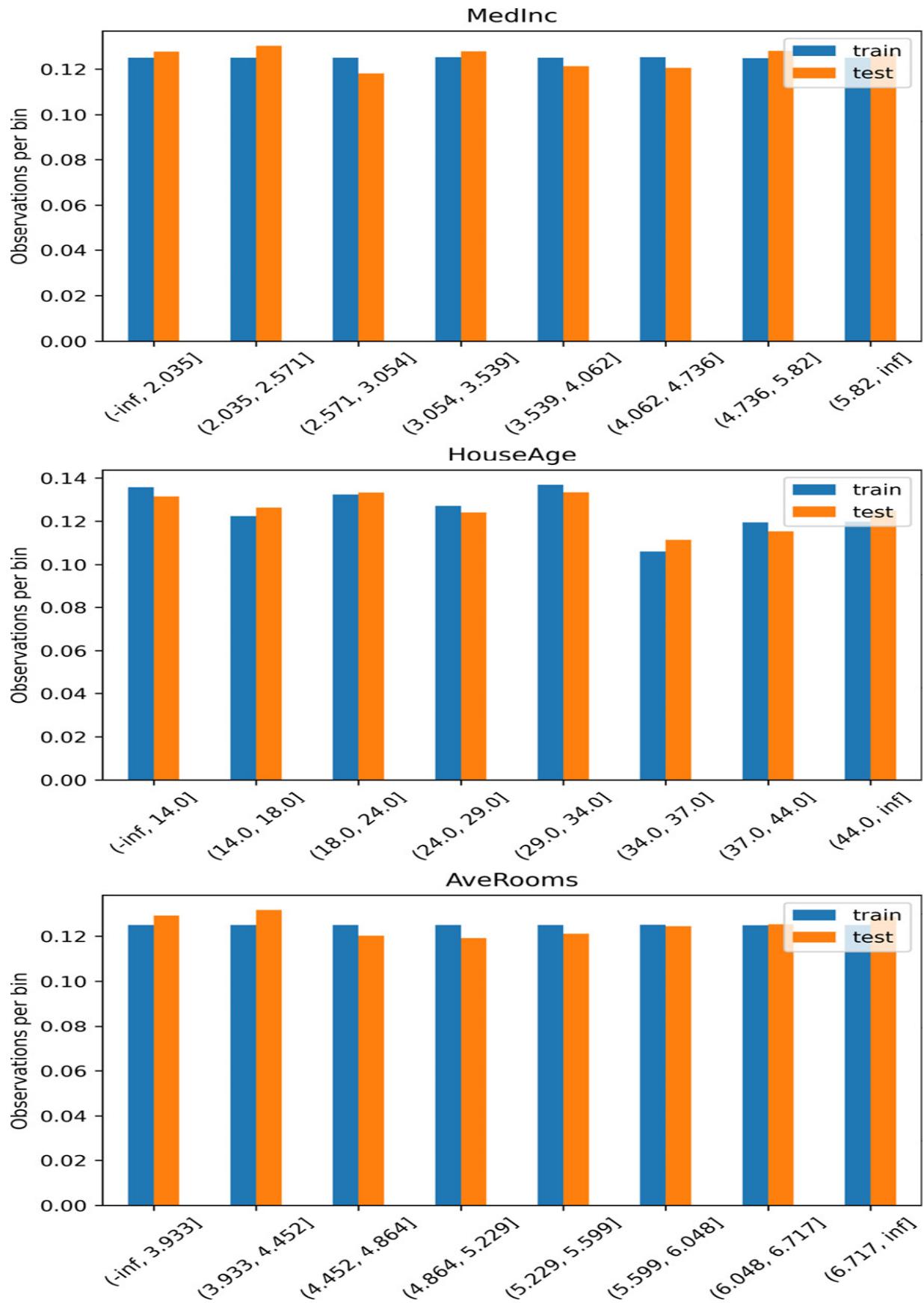


Figure 4.5 – The proportion of observations per interval

Now, let's carry out equal-frequency discretization with **scikit-learn**.

13. Let's import the transformer:

```
from sklearn.preprocessing import KBinsDiscretizer
```

14. Let's set up an equal-frequency discretizer by setting **strategy** to **quantile** and the number of bins to **8**:

```
disc = KBinsDiscretizer(n_bins=8, encode='ordinal',  
strategy='quantile')
```

15. Let's fit the discretizer to a slice of the train set containing the variables to transform so that it learns the interval limits:

```
disc.fit(X_train[variables])
```

NOTE

Scikit-learn's KBinsDiscretiser() will discretize all of the variables in the dataset, so we only need to use the transformer in the slice of the dataframe that contains the variables of interest. In the *Performing equal-width discretization* recipe, we used the discretizer within **ColumnTransformer()** as an alternative implementation.

16. Let's make a copy of the dataframe where we will capture the discretized variables:

```
train_t = X_train.copy()  
test_t = X_test.copy()
```

17. Finally, let's transform the variables in both train and test sets:

```
train_t[variables] = disc.transform(X_train[variables])
test_t[variables] = disc.transform(X_test[variables])
```

TIP

We can inspect the cut-points by executing `disc.bin_edges_`.

Scikit-learn returns a **NumPy** array, the values which we assigned directly to the variables in our dataframe.

How it works...

With equal-frequency discretization, we sorted the variable values into intervals with a similar proportion of observations.

We used **pandas qcut()** to determine the limits of the intervals based on train set variable quantiles and sorted the values of the **HouseAge** variable into those intervals. Next, we used those limits together with **pandas cut()** to discretize **HouseAge** in the test set. Note that **pandas qcut()**, like **pandas cut()**, returned the interval values as ordered integers. Through the **labels** parameter, we can modify this behavior to return the interval boundaries or user-defined labels.

To perform equal-frequency discretization with **Feature-engine**, we used **EqualFrequencyDiscretiser()** and indicated the number of quantiles and the variables to discretize as arguments. With **fit()**, the discretizer learned and stored the interval limits in its **binner_dict_** attribute. With **transform()**, the observations were allocated to the bins. The values of the discretized variables are integers, representing the 1st, 2nd, 3rd, and so on bins.

The **EqualFrequencyDiscretiser()** class offers a few options to follow up the discretization with encoding. By default, it will return the bins as sorted integers, which is the equivalent of ordinal encoding. Alternatively, it can return the interval boundaries as with **pandas cut()**. To follow up the discretization with any other encoding procedure available in **Feature-engine** or **Category Encoders**, we need to return the variables cast as objects, which can be achieved by setting **return_object** to **True**.

To understand the effect of the discretization, we performed bar plots with the fraction of observation per interval. The variables showed a similar fraction of observations in each of the intervals, in contrast to what we observed after performing equal-width discretization in the previous recipe, where the skewed **MedInc** and **AveRooms** variables showed a few intervals that were almost empty.

NOTE

With equal-frequency discretization, many occurrences of values within a small continuous range could cause observations with very similar values, resulting in different intervals. This would not happen with equal-width discretization.

Finally, we discretized variables with **KBinsDiscretizer()** from **scikit-learn**, indicating the number of bins and setting **strategy** to **quantile**. With the **fit()** method, the transformer learned and stored the cut-points in its **bin_edges_** attribute, and with the **transform()** method, the discretizer sorted the values of the variables to each interval. Note that, differently from **EqualFrequencyDiscretiser()**, **KBinsDiscretizer()** will transform all of the variables in the dataset. To

avoid this, we only applied the discretizer on a slice of the data with the variables to modify. **Scikit-learns** `KBinsDiscretizer()` has the option to return the intervals as ordinal numbers or one-hot encoding. The behavior can be modified through the `encode` parameter.

Discretizing the variable into arbitrary intervals

In various industries, it is common to group variable values into segments that make sense for the business. For example, we might want to group the variable age in intervals representing children, young adults, middle-aged people, and retired people. Alternatively, we might group ratings into bad, good, and excellent. On other occasions, if we know that the variable is in a certain scale, for example, logarithmic, we might want to define the interval cut-points within that scale.

In this recipe, we will discretize a variable into pre-defined user intervals using **pandas** and **Feature-engine**.

How to do it...

First, let's import the necessary Python libraries and get the dataset ready:

1. Import the required Python libraries and classes:

```
import numpy as np  
  
import pandas as pd  
  
import matplotlib.pyplot as plt  
  
from sklearn.datasets import fetch_california_housing
```

2. Let's load the California housing dataset into a **pandas** dataframe:

```
x, y = fetch_california_housing(  
    return_X_y=True, as_frame=True)
```

3. Let's plot a histogram of the **Population** variable to find out its value range:

```
X["Population"].hist(bins=30)  
plt.title("Population")  
plt.ylabel("Number of observations")  
plt.show()
```

Population values vary from 0 to, approximately, 40,000:

Population

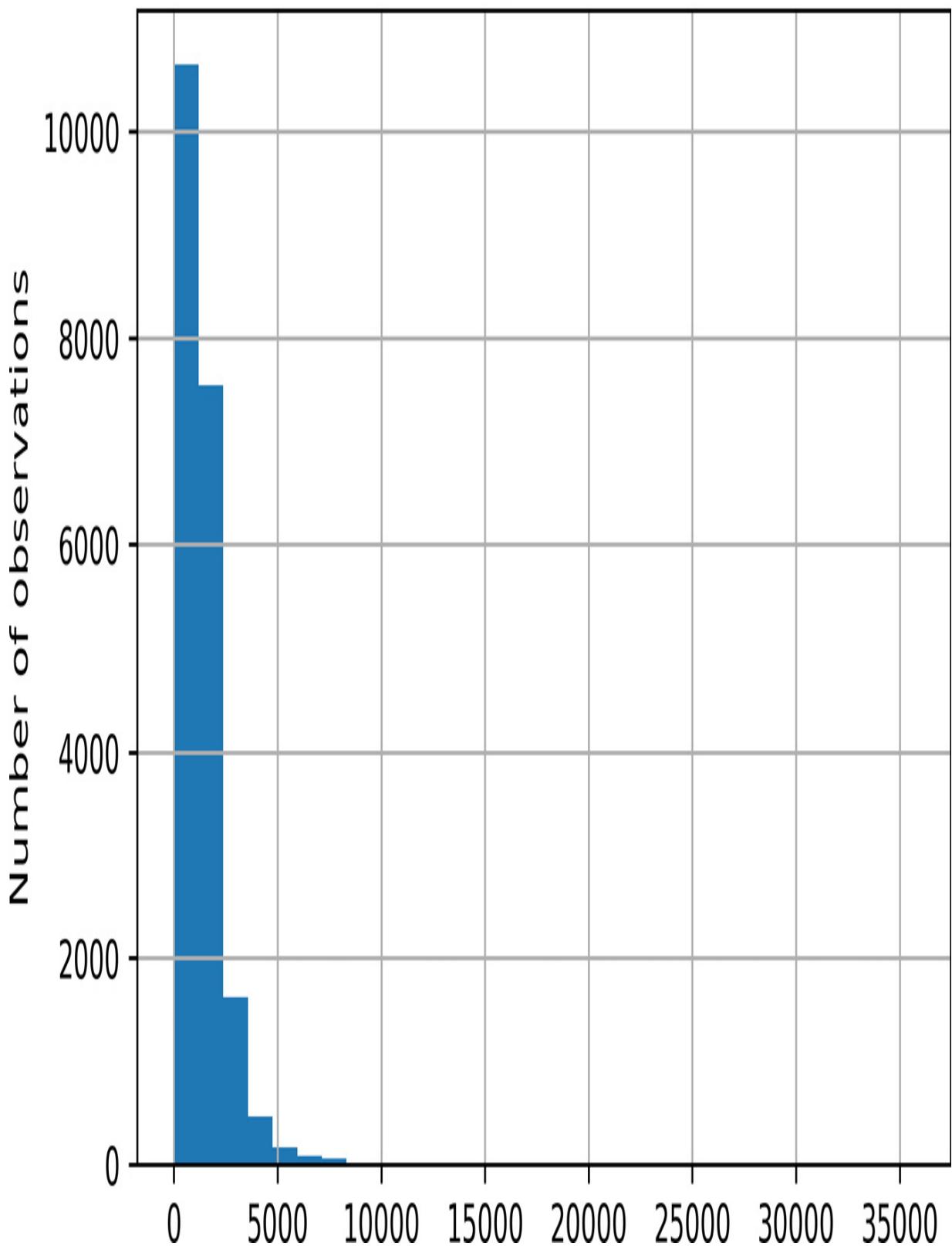


Figure 4.6 – A histogram of the Population variable

4. Let's create a list with arbitrary interval limits, setting the upper limit to infinity to accommodate bigger values:

```
intervals = [0, 200, 500, 1000, 2000, np.Inf]
```

5. Let's create a list with the interval limits as strings:

```
labels = ["0-200", "200-500", "500-1000", "1000-2000",  
">2000"]
```

6. Let's discretize the **Population** variable into the pre-defined limits from *step 4*, where the intervals take the values we created in *step 5* as values:

```
X["Population_range"] = pd.cut(  
    X["Population"],  
    bins=intervals,  
    labels=labels,  
    include_lowest=True,  
)
```

7. Now, let's discretize **Population** into pre-defined intervals and capture it in a new variable that takes the interval limits as values:

```
X["Population_limits"] = pd.cut(  
    X["Population"],  
    bins=intervals,  
    labels=None,  
    include_lowest=True,
```

```
)
```

8. Let's inspect the first five rows of the original and discretized variables:

```
X[['Population', 'Population_range',  
    'Population_limits']].head()
```

In the last two columns of the dataframe, we see the discretized variables: the first one with the strings we created in *step 5* as values, and the second one with the interval limits:

	Population	Population_range	Population_limits
0	322.0	200-500	(200.0, 500.0]
1	2401.0	>2000	(2000.0, inf]
2	496.0	200-500	(200.0, 500.0]
3	558.0	500-1000	(500.0, 1000.0]
4	565.0	500-1000	(500.0, 1000.0]

NOTE

We only need one of the variable versions, either the one with the value range or the one with the interval limits. In this recipe, I create both to highlight the different options offered by **pandas**.

9. Finally, we can count and plot the number of observations within each interval:

```
X['Population_range'].value_counts().sort_index().plot.  
bar()  
  
plt.xticks(rotation=0)  
plt.ylabel("Number of observations")
```

```
plt.title("Population")  
plt.show()
```

The number of observations per interval varies, as shown in the output of the proceeding code block:

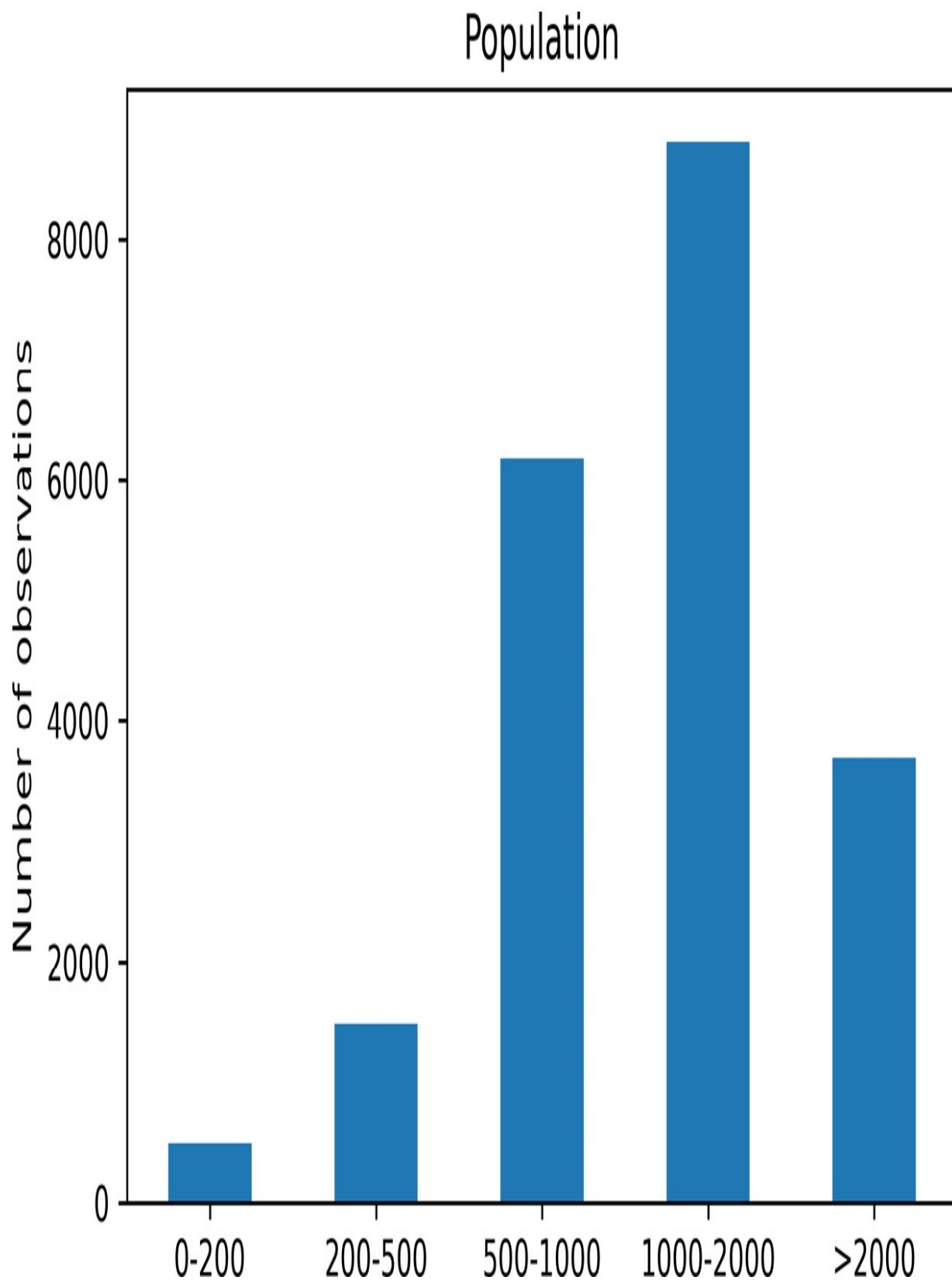


Figure 4.7 – The proportion of observations per interval

To wrap up the recipe, let's carry out arbitrary discretization utilizing **Feature-engine**. First, we need to reload the California housing dataset, as we did in *step 2*.

10. Let's import the transformer:

```
from feature_engine.discretisation import  
    ArbitraryDiscretiser
```

11. Let's set up the transformer by using a dictionary with the variables to modify, in this case, **Population** and the interval limits in a list:

```
discretizer = ArbitraryDiscretiser(  
    binning_dict={"Population": intervals},  
    return_boundaries=True,  
)
```

12. Now we can go ahead and discretize the variable:

```
X_t = discretizer.fit_transform(X)
```

Now, if we execute **X_t.head()**, we will see the following output, where the **Population** variable has been discretized:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
0	8.3252	41.0	6.984127	1.023810	(200.0, 500.0]	2.555556	37.88	-122.23
1	8.3014	21.0	6.238137	0.971880	(2000.0, inf]	2.109842	37.86	-122.22
2	7.2574	52.0	8.288136	1.073446	(200.0, 500.0]	2.802260	37.85	-122.24
3	5.6431	52.0	5.817352	1.073059	(500.0, 1000.0]	2.547945	37.85	-122.25
4	3.8462	52.0	6.281853	1.081081	(500.0, 1000.0]	2.181467	37.85	-122.25

Figure 4.8 – A dataframe containing the discretized variable

The advantage of using **Feature-engine** is that we can discretize multiple variables in the dataframe at the same time.

How it works...

In this recipe, we sorted the values of a variable into user-defined intervals. First, we plotted a histogram of the **Population** variable, to get an idea of the range of values of the variable. Next, we arbitrarily determined and captured the limits of the intervals in a list: we created intervals that vary from 0–200, 200–500, 500–1000, 1000–2000, and more than 2,000, by setting the upper limit to infinite with `np.Inf`. Next, we created a list with the interval names as strings.

Using the **pandas cut()** method and passing the list with the interval limits, we sorted the variable values into the pre-defined bins. We executed

the command twice; in the first run, we set the **labels** argument to the list that contained the label names as strings, and in the second run, we set the **labels** argument to **None**. We captured the returned output in two variables: the first one displaying the interval limits as values and the second one with interval names as values. Finally, we counted the number of observations per variable using the **pandas value_counts()** method.

Finally, we automated the procedure with **ArbitraryDiscretiser()** from **Feature-engine**. This transformer takes a dictionary with the variables to discretize as keys, and the interval limits in a list as values, and then uses **pandas cut** under the hood to discretize the variables. With **fit()**, the transformer does not learn any parameters but checks that the variables are numerical. With **transform()**, the variables indicated in the dictionary are discretized.

Performing discretization with k-means clustering

The aim of a discretization procedure is to find a set of cut-points that partition a variable into a small number of intervals that have good class coherence. To create partitions that group similar observations, we can use clustering algorithms, such as k-means.

In discretization using k-means clustering, the partitions are the clusters identified by the k-means algorithm. The k-means clustering algorithm has two main steps. In the initialization step, k observations are chosen randomly as the initial centers of the k clusters, and the remaining data points are assigned to the closest cluster. The proximity to the cluster is

measured by a distance measure such as the Euclidean distance. In the iteration step, the centers of the clusters are re-computed as the average points of all of the observations within the cluster, and the observations are reassigned to the newly created closest cluster. The iteration step continues until the optimal k centers are found.

Discretization with k-means requires one parameter, which is k , the number of clusters. There are a few methods to determine the optimal number of clusters: one of them is the elbow method, which we will use in this recipe. This method consists of training several k-means algorithms over the data using different values of k , and then determining the explained variation returned by the clustering. In the next step, we plot the explained variation as a function of the number of clusters, k , and pick the **elbow** of the curve as the number of clusters to use. The elbow is the inflection point that indicates that increasing the number of k further does not significantly increase the variance explained by the model. There are different metrics to quantify the explained variation. We will use the sum of square distances from each point to its assigned center.

In this recipe, we will use **Yellowbrick** to determine the optimal number of clusters and then carry out k-means discretization with **scikit-learn**.

How to do it...

First, let's import the necessary Python libraries and get the dataset ready:

1. Import the required Python libraries and classes:

```
import pandas as pd  
from sklearn.cluster import KMeans
```

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import KBinsDiscretizer
from yellowbrick.cluster import KElbowVisualizer
```

2. Let's load the California housing dataset into a **pandas** dataframe:

```
X, y = fetch_california_housing(
    return_X_y=True, as_frame=True)
```

3. The k-means optimal clusters should be determined using the train set, so let's divide the data into train and test sets:

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=0,
)
```

4. Let's make a list with the variables to transform:

```
variables = ['MedInc', 'HouseAge', 'AveRooms']
```

5. Let's set up a k-means clustering algorithm:

```
k_means = KMeans(random_state=10)
```

6. Now, using **Yellowbrick**'s visualizer and the elbow method, let's find the optimal number of clusters for each variable:

```
for variable in variables:
    # set up a visualizer
    visualizer = KElbowVisualizer(
        k_means, k=(4,12), metric='distortion',
```

```
    timings=False  
)  
  
# Fit the data to the visualizer  
visualizer.fit(X_train[variable].to_frame())  
  
# Finalize and render the figure  
visualizer.show()
```

In the following plots, we can see that the optimal number of clusters is six for each variable:

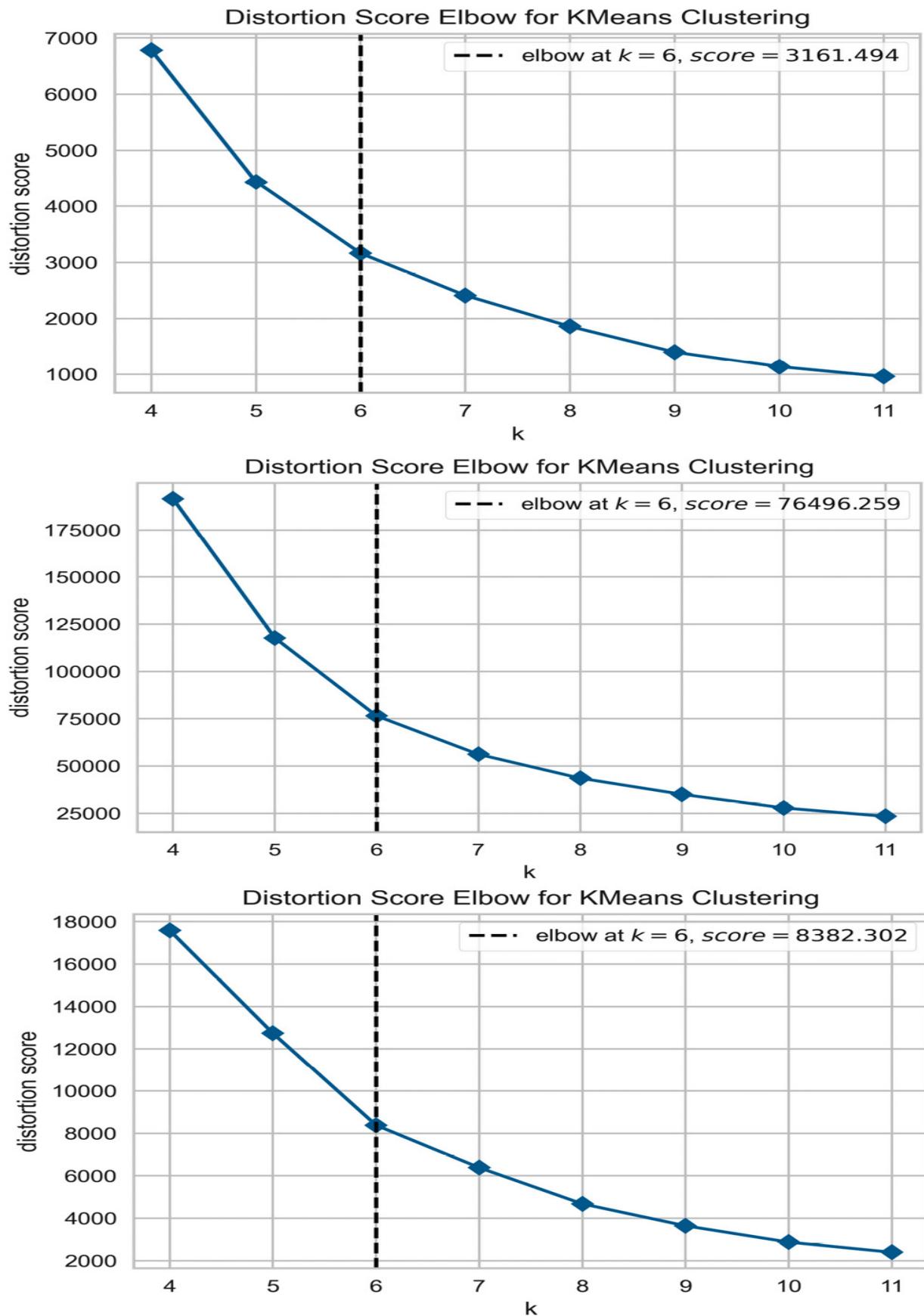


Figure 4.9 – Number of clusters versus the variability for the **MedInc**, **HouseAge**, and **AveRooms** variables from top to bottom

7. Let's set up a discretizer that uses k-means clustering to create six partitions and returns the clusters as one-hot-encoded variables:

```
k = 6
```

```
disc = KBinsDiscretizer(n_bins=k, encode="onehotdense",  
strategy="kmeans")
```

8. Let's fit the discretizer to the slice of the dataframe that contains the variables that we want to discretize. This is so that the transformer finds the optimal clusters for each variable:

```
disc.fit(X_train[variables])
```

TIP

If we had a different optimal number of clusters for different variables, we would have to set up one instance of the discretizer for each variable group with the same value of k , either by repeating steps 7 and 8 or by using **ColumnTransformer()** from **scikit-learn** to slice the dataframe into the requited variable group.

9. Let's inspect the cut-points:

```
disc.bin_edges_
```

Each array contains the cut-points for the six clusters for each of the three variables, **MedInc**, **HouseAge**, and **AveRooms**:

```
array([array([0.4999, 2.49587954, 3.66599029, 4.95730115,  
6.67700141, 9.67326677, 15.0001]),
```

```
array([1., 11.7038878, 19.88430419, 27.81472503,  
35.39424098, 43.90930314, 52.]),  
  
array([0.84615385, 4.84568771, 6.62222005, 15.24138445,  
37.60664483, 92.4473438, 132.53333333]), dtype=object)
```

10. Let's obtain the discretized form of the original variables in the train test sets:

```
train_features = disc.transform(X_train[variables])  
  
test_features = disc.transform(X_test[variables])
```

With **print(test_features)**, we can inspect the Numpy array returned by the discretizer, containing 18 binary variables corresponding to the one-hot-encoded transformation of the 6 clusters returned for each of the 3 numerical variables:

```
array([[0., 0., 1., ..., 0., 0., 0.],  
       [0., 0., 0., ..., 0., 0., 0.],  
       [0., 0., 1., ..., 0., 0., 0.],  
       ...,  
       [0., 0., 0., ..., 0., 0., 0.],  
       [1., 0., 0., ..., 0., 0., 0.],  
       [0., 0., 1., ..., 0., 0., 0.]]))
```

Now, let's concatenate the new features to the original dataframe.

11. Let's create names for the new features:

```
var_names = [  
    f"var_{i}_cluster_{i}"
```

```
    for var in variables  
        for i in range(k)  
    ]
```

By executing `print(var_names)`, we see the following names:

```
['MedInc_cluster_0',  
'MedInc_cluster_1',  
'MedInc_cluster_2',  
'MedInc_cluster_3',  
'MedInc_cluster_4',  
'MedInc_cluster_5',  
'HouseAge_cluster_0',  
'HouseAge_cluster_1',  
'HouseAge_cluster_2',  
'HouseAge_cluster_3',  
'HouseAge_cluster_4',  
'HouseAge_cluster_5',  
'AveRooms_cluster_0',  
'AveRooms_cluster_1',  
'AveRooms_cluster_2',  
'AveRooms_cluster_3',  
'AveRooms_cluster_4',  
'AveRooms_cluster_5']
```

12. Let's concatenate the new features to the original dataframe:

```
x_train = pd.concat([
    x_train,
    pd.DataFrame(train_features, columns=var_names,
                 index=x_train.index)
], axis=1)

x_test = pd.concat([
    x_test,
    pd.DataFrame(test_features, columns=var_names,
                 index=x_test.index)
], axis=1)
```

13. And finally, let's drop the original variables:

```
x_train.drop(labels=variables, axis=1, inplace=True)
x_test.drop(labels=variables, axis=1, inplace=True)
```

Now, we are done with the discretization using k-means clustering.

How it works...

In this recipe, we performed discretization with k-means clustering. First, we identified the optimal number of clusters utilizing the **elbow** method.

With **KElbowVisualizer()** from **Yellowbrick**, we identified the optimal cluster number for three variables.

To perform k-means discretization, we used `KBinsDiscretizer()` from `scikit-learn`, setting `strategy` to `kmeans` and the number of clusters to six in the `n_bins` argument. With the `fit()` method, the transformer learned the cluster boundaries using the k-means algorithm. With the `transform()` method, the discretizer sorted the values of the variable to their corresponding cluster, returning a `NumPy` array with the discretized variables, which we converted into a dataframe.

See also

- Discretization with k-means was described in Palaniappan and Hong, *Discretization of Continuous Valued Dimensions in OLAP Data Cubes*. International Journal of Computer Science and Network Security, VOL.8 No.11, November 2008.
- To learn more about the elbow method, visit the `Yellowbrick` documentation and references therein: <https://www.scikit-yb.org/en/latest/api/cluster/elbow.xhtml>.
- For other ways of determining the fit of k-means clustering, check out the additional visualizers in `Yellowbrick`: <https://www.scikit-yb.org/en/latest/api/cluster/index.xhtml>.

Implementing feature binarization

Some datasets contain sparse variables. Sparse variables are those where the majority of the values are 0. The classical example of sparse variables are those derived from text data through the bag-of-words model, where each variable is a word, and each value represents the number of times the word

appears in a certain document. Given that a document contains a limited number of words, whereas the feature space contains the words that appear across all documents, most documents, that is, most rows, will show the value of 0 for most columns. But words are not the sole example. If we think about house details data, the variable number of saunas will also be 0 for most houses. In summary, some variables have very skewed distributions, where most observations show the same value, usually 0, and only a few observations show different, usually higher, values.

A more robust representation of these sparse or highly skewed variables is to binarize them by clipping all values greater than 1 to 1. In fact, binarization is commonly performed on text count data, where we consider the presence or absence of a feature rather than a quantified number of occurrences of a word.

In this recipe, we will perform binarization using **scikit-learn**.

Getting ready

We will use a dataset consisting of a bag of words, which is available in the UCI Machine Learning Repository (<https://archive.ics.uci.edu/ml/datasets/Bag+of+Words>).

To prepare the dataset, follow these steps:

1. Visit <https://archive.ics.uci.edu/ml/machine-learning-databases/bag-of-words/>.
2. Click on **docword.enron.txt.gz** to download the data.

3. Unzip **docword.enron.txt.gz** and save **docword.enron.txt** in the folder where you will run the following commands.
4. Then, click on **vocab.enron.txt** to download the word names.
5. Save **vocab.enron.txt** in the folder where you will run the following commands.

After you've downloaded the data, open a Jupyter notebook and run the following commands.

6. Import **pandas**:

```
import pandas as pd
```

7. Load the data into a **pandas** dataframe and assign the column names:

```
data = pd.read_csv(  
    "docword.enron.txt", sep=" ",  
    skiprows=3, header=None)  
  
data.columns = ["docID", "wordID", "count"]
```

8. Load the words:

```
words = pd.read_csv("vocab.enron.txt", header=None)  
  
words.columns = ["words"]
```

9. Select 10 words at random:

```
words = words.sample(10, random_state=290917)
```

10. Merge the word counts to the 10 selected words:

```
data = words.merge(  
    data, left_index=True, right_on="wordID")
```

11. Reconstitute the bag of words:

```
bow = data.pivot(index="docID", columns="words",
                  values="count")
bow.fillna(0, inplace=True)
bow.reset_index(inplace=True, drop=True)
```

12. Save the data:

```
bow.to_csv("bag_of_words.csv", index=False)
```

Now, you are ready to carry on with the recipe.

You can find the instructions to download, prepare, and store the dataset in the notebook of our repository at

<https://github.com/PacktPublishing/Python-Feature-Engineering-Cookbook-Second-Edition/blob/main/ch04-discretization/download-prepare-store-enron-data.ipynb>.

How to do it...

Let's begin by importing the libraries and getting the data loaded:

1. Let's import the required Python libraries, classes, and datasets:

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import Binarizer
from feature_engine.wrappers import
SklearnTransformerWrapper
```

2. Let's load the data that we prepared in the *Getting ready* section:

```
data = pd.read_csv("bag_of_words.csv")
```

3. Let's display histograms to visualize the sparsity of the variables:

```
data.hist(bins=30, figsize=(20, 20), layout=(3, 4))  
plt.show()
```

In the following histograms, we can see that the words appear 0 times in most documents:

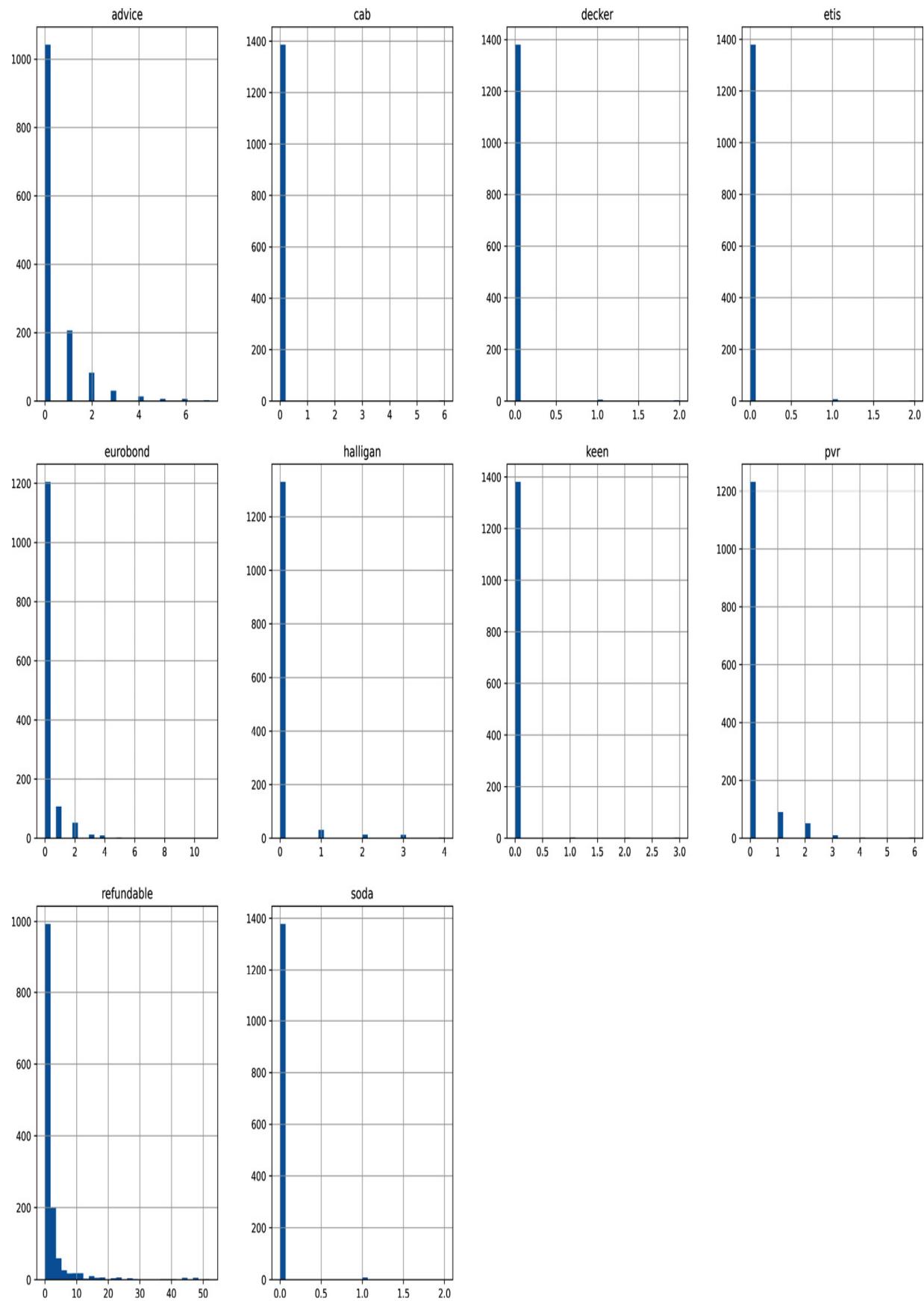


Figure 4.10 – A histogram of the sparse variables

4. To perform binarization, let's set up **binarizer** to clip all values greater than 1, to 1:

```
binarizer = Binarizer(threshold = 0)
```

5. **Binarizer()** from **scikit-learn** will return a NumPy array by default, so to return a dataframe, let's wrap it with the **Feature-engine** wrapper:

```
wrapper =  
SklearnTransformerWrapper(transformer=binarizer)
```

NOTE

SklearnTransformerWrapper, when wrapping classes that work with numerical variables, such as **Binarizer()**, will select all the numerical variables by default. Alternatively, we can indicate the variables to transform through the **variables** parameter.

6. Let's binarize the variables:

```
data_t = wrapper.fit_transform(data)
```

We can explore the variables that will be transformed by executing

```
print(wrapper.variables_):
```

```
['advice',  
'cab',  
'decker',  
'etis',
```

```
'eurobond',  
'halligan',  
'keen',  
'pvr',  
'refundable',  
'soda']
```

Now we can explore the distribution of the binarized variables by displaying the histograms as in *step 3*, or better, by creating bar plots.

7. Let's create a bar plot with the number of observations per bin per variable:

```
variables = data_t.columns.to_list()  
  
plt.figure(figsize=(20, 20), constrained_layout=True)  
  
for i in range(10):  
  
    # location in figure  
  
    ax = plt.subplot(3, 4, i + 1)  
  
    # variable to plot  
  
    var = variables[i]  
  
    # determine proportion of observations per bin  
  
    t = data_t[var].value_counts(normalize=True)  
  
    t.plot.bar(ax=ax)  
  
    plt.xticks(rotation=45)  
  
    plt.ylabel("Observations per bin")  
  
    # add variable name as title
```

```
ax.set_title(var)  
plt.show()
```

In the following plot, we can see the binarized variables, where most occurrences show the value of 0:

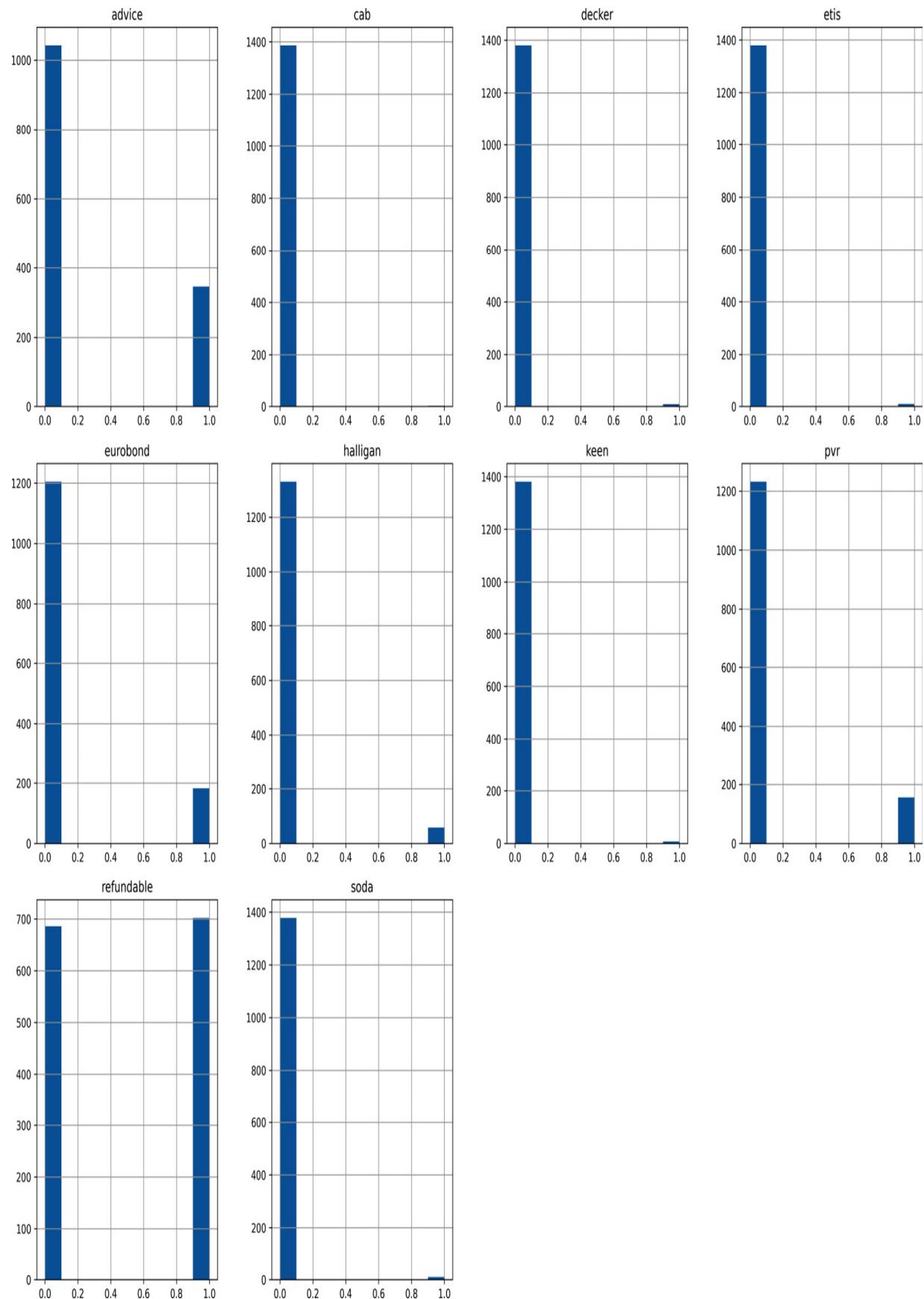


Figure 4.11 – The proportion of observations per interval

That's it; now our variables have a more succinct representation of the data.

How it works...

In this recipe, we changed the representation of sparse variables to consider the presence or absence of an occurrence, which, in our case, is a word. The data consisted of a bag of words, where each variable is a word, each row is a document, and the values represent the number of times a word appears in a document. Most words do not appear in most documents; therefore, most values in the data are 0. We corroborated the sparsity of our data with histograms.

Binarizer() from **scikit-learn** mapped values greater than the threshold, which, in our case, is 0, to the value 1, while values less than or equal to the threshold were mapped to 0. **Binarizer()** has the **fit()** and **transform()** methods, where **fit()** does not do anything and **transform()** binarizes the variables. **Binarizer()** modifies all variables in a dataset returning **NumPy** arrays. To return a **pandas** dataframe instead, we wrapped **Binarizer()** with **SklearnTransformerWrapper()** from **Feature-engine**. The wrapper found all numerical variables automatically, and applied **Binarizer()** to them, returning a **pandas** dataframe. The wrapper has the option to apply **Binarizer()** to only a subset of the variables; it does this by passing the variable names in a list when initializing the transformer.

NOTE

`sklearnTransformerWrapper()` is an alternative to `ColumnTransformer()` from `scikit-learn` and can wrap many transformers from this library.

Using decision trees for discretization

Decision tree methods discretize continuous attributes during the learning process. A decision tree evaluates all possible values of a feature and selects the cut-point that maximizes the class separation, by utilizing a performance metric such as the entropy or Gini impurity. Then, it repeats the process for each node of the first data separation, along with each node of the subsequent data splits, until a certain stopping criterion has been reached. Therefore, by design, decision trees can find the set of cut-points that partition a variable into intervals with good class coherence.

Discretization with decision trees consists of using a decision tree to identify the optimal partitions for each continuous variable. In the **Feature-engine** implementation of this method, the decision tree is fit using the variable to discretize, and the target. After fitting, the decision tree is able to assign each observation to one of the N end leaves, generating a discrete output, the values of which are the predictions at each of its leaves. Therefore, with **Feature-engine**, the values of continuous features are replaced by the predictions of a decision tree.

In this recipe, we will perform decision tree-based discretization manually using `scikit-learn` and then automate the procedure with **Feature-engine**.

How to do it...

Let's begin by importing some libraries and getting the data loaded:

1. Let's import the required Python libraries, classes, and datasets:

```
import pandas as pd  
  
import matplotlib.pyplot as plt  
  
from sklearn.datasets import fetch_california_housing  
  
from sklearn.model_selection import train_test_split  
  
from sklearn.tree import DecisionTreeRegressor,  
plot_tree
```

2. Let's load the California housing dataset into a **pandas** dataframe:

```
x, y = fetch_california_housing(return_X_y=True,  
                                 as_frame=True)
```

3. Let's divide the data into train and test sets:

```
x_train, x_test, y_train, y_test = train_test_split(  
    x, y, test_size=0.3, random_state=0,  
)
```

4. Let's assemble a decision tree to predict the house price, setting the maximum depth to 3 and **random_state** for reproducibility:

```
tree_model = DecisionTreeRegressor(  
    max_depth=3, random_state=0)
```

NOTE

For binary classification, we would use `DecisionTreeClassifier()` instead.

5. Let's fit the decision tree using the `MedInc` variable to predict the target:

```
tree_model.fit(X_train['MedInc'].to_frame(), y_train)
```

NOTE

`Scikit-learn` predictors take dataframes as inputs. A single variable is a `pandas` series, so we need to use the `to_frame()` method to transform it into a dataframe and make it compatible with `scikit-learn`.

6. Now, let's replace the values of `MedInc` with the predictions of the decision tree:

```
X_train['MedInc_tree'] = tree_model.predict(  
    X_train['MedInc'].to_frame())  
  
X_test['MedInc_tree'] = tree_model.predict(  
    X_test['MedInc'].to_frame())
```

NOTE

If we created a classification tree, we would use `predict_proba()` and retain the second column of the array, which is the probability of the target being 1.

7. Let's explore the end leaves, that is, the partitions the tree created:

```
X_test['MedInc_tree'].unique()
```

The decision tree produced eight different prediction outputs for all of the observations of the `MedInc` variable:

```
array([2.27113313, 2.76842266, 1.51756502, 3.46323449,  
1.91725491, 1.14863215, 4.31789332, 4.87645172])
```

8. In many cases, this discretization method returns a monotonic relation between the discretized variable and the target, so let's make a line plot to visualize this relationship:

```
y_test.groupby(X_test["MedInc_tree"]).mean().plot()  
plt.title("Monotonic relationship between discretized  
MedInc and target")  
plt.ylabel("Median Price")
```

We can observe the monotonic relationship between the tree-derived partitions and the target in the output of the preceding code block:

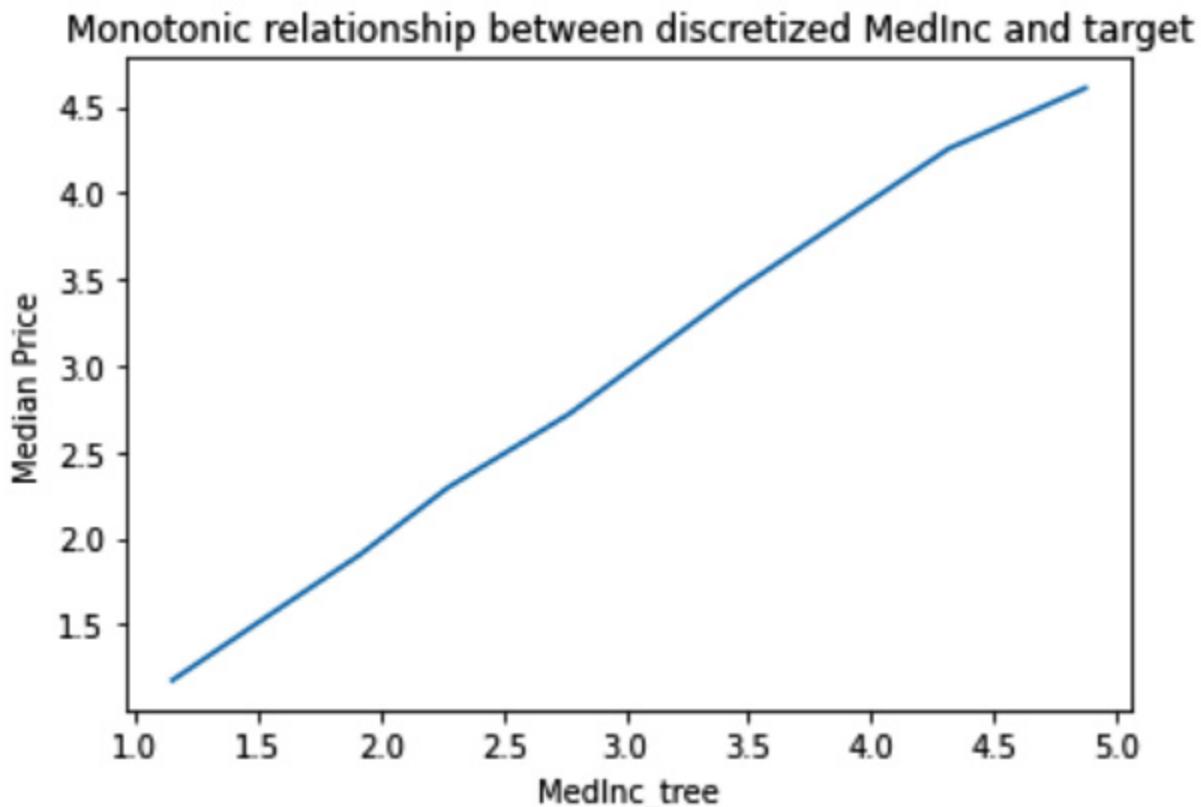


Figure 4.12 – The monotonic relationship between the discretized variable and the target

9. Also, we can explore the number of observations per end leave:

```
x_test["MedInc_tree"] = np.round(  
    x_test["MedInc_tree"], 2)  
  
x_test.groupby(  
    ["MedInc_tree"])["MedInc"].count().plot.bar()  
  
plt.title("Median Income")  
plt.ylabel("Number of observations")
```

We can see the number of observations per prediction in the following bar plot:

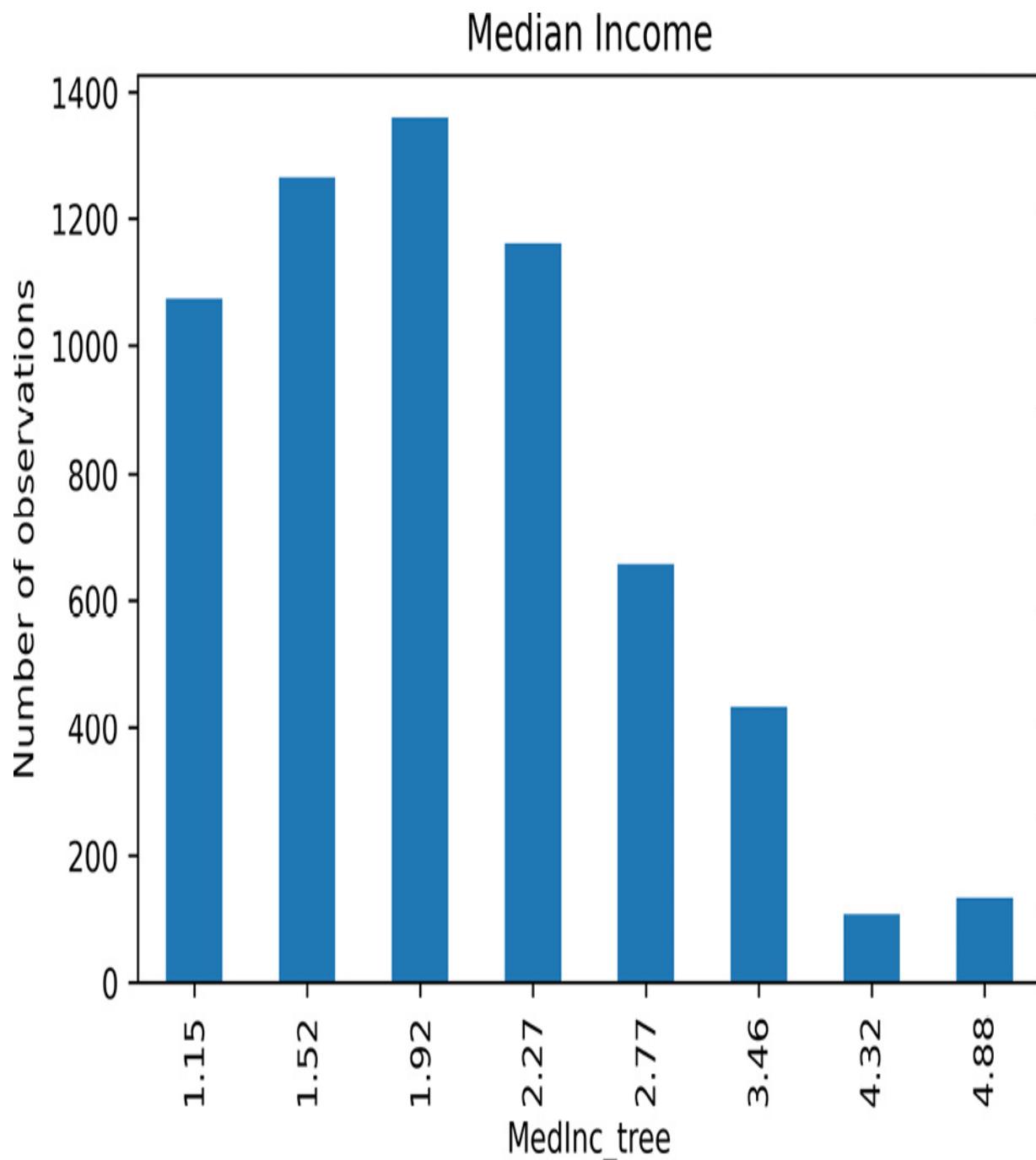


Figure 4.13 – The number of observations per bin

10. To better understand how the decision tree made the partitions, let's display them:

```
fig = plt.figure(figsize=(20, 6))
```

```

plot_tree(tree_model, fontsize=10, proportion=True)
plt.show()

```

In the following display, we can see how the decision tree partitioned the variable based on cut-points that maximized the entropy into eight final leaves or bins for our method:

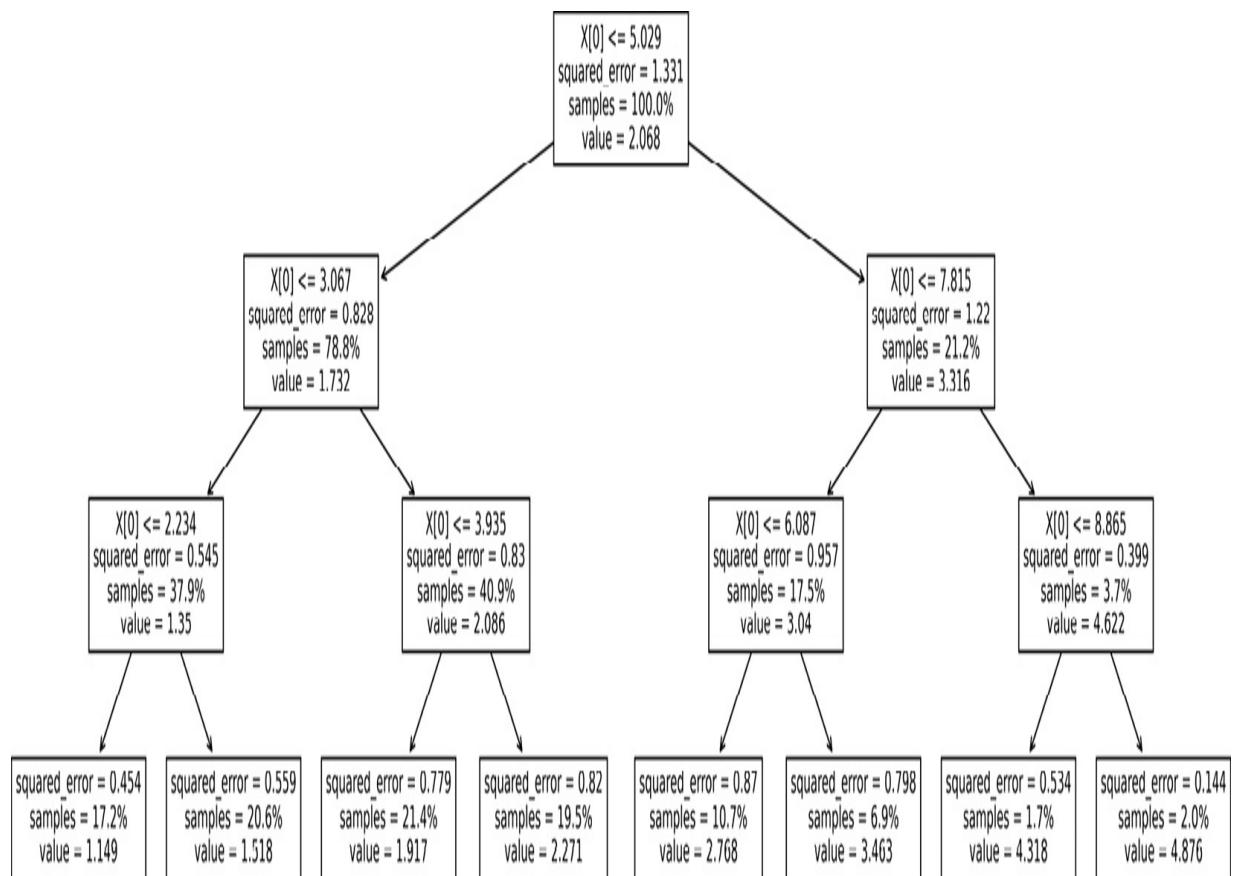


Figure 4.14 – A diagram of the decision tree

Now, let's implement decision tree discretization with **Feature-engine**. With **Feature-engine**, we can discretize multiple variables at the same time, while searching for the best hyperparameters for the decision trees using cross-validation.

First, we need to divide the dataset into train and test sets, just as we did in *step 3*.

11. Let's import the discretizer:

```
from feature_engine.discretisation import  
DecisionTreeDiscretiser
```

12. Let's capture all variables except the latitude and longitude in a list:

```
variables = list(X.columns)[:-2]
```

13. Now, let's set up the decision tree discretizer, which will optimize the hyperparameter's maximum depth and minimum samples per leaf based on the negative mean square error metric using three-fold cross-validation:

```
treeDisc = DecisionTreeDiscretiser(  
  
    cv=3,  
  
    scoring='neg_mean_squared_error',  
  
    variables=variables,  
  
    regression=True,  
  
    param_grid={'max_depth': [1, 2, 3],  
                "min_samples_leaf": [10, 20, 50]},  
  
)
```

14. Let's **fit()** the discretizer using the train set and the target so that it finds the best decision trees for each of the variables:

```
treeDisc.fit(X_train, y_train)
```

15. Let's inspect the best parameters for the tree trained for the **MedInc** variable:

```
treeDisc.binner_dict_["MedInc"].best_params_
```

The output of the preceding code shows that the optimal depth for the decision tree is **3** and the optimal samples per leaf is **10**:

```
{'max_depth': 3, 'min_samples_leaf': 10}
```

16. Let's transform the variables in the train and test sets:

```
train_t = treeDisc.transform(X_train)  
test_t = treeDisc.transform(X_test)
```

17. Finally, we can explore the number of observations per partition for three of the discretized variables:

```
plt.figure(figsize=(6, 12), constrained_layout=True)  
for i in range(3):  
    # location in figure  
    ax = plt.subplot(3, 1, i + 1)  
    # variable to plot  
    var = variables[i]  
    # determine proportion of observations per bin  
    t1 = train_t[var].value_counts(normalize=True)  
    t2 = test_t[var].value_counts(normalize=True)  
    # concatenate proportions  
    tmp = pd.concat([t1, t2], axis=1)
```

```
tmp.columns = ["train", "test"]

# order the intervals

tmp.sort_index(inplace=True)

tmp.plot.bar(ax=ax)

plt.xticks(rotation=45)

plt.ylabel('Observations per bin')

# add variable name as title

ax.set_title(var)

plt.show()
```

We can see the number of observations per bin in the output of the preceding code:

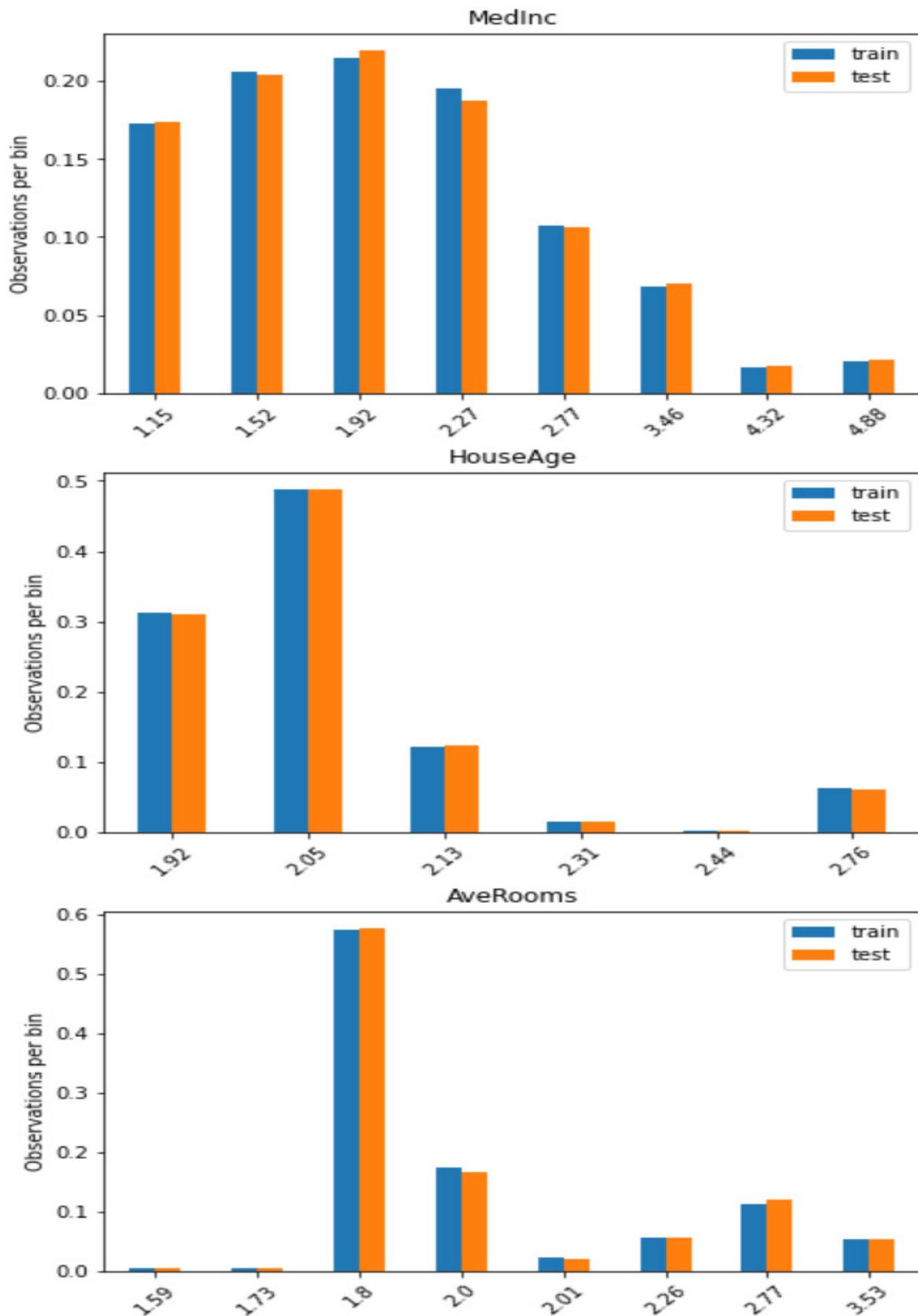


Figure 4.15 – The number of observations per bin

As evidenced in the plots, discretization with decision trees returns a different fraction of observations at each node or bin.

How it works...

To perform discretization with decision trees, first, we chose a variable, **MedInc**, and fit a decision tree for regression using **DecisionTreeRegressor()** from **scikit-learn**. We used **to_frame()** to transform the **pandas** series with the variable into a dataframe and make the data compatible with the **scikit-learn** predictors. We used the **fit()** method to make the tree learn how to predict the target from **MedInc**. With the **predict()** method, the tree estimated the target from **MedInc** in the train and test sets. The decision tree returned eight distinct values, which were its predictions. These outputs represented the bins or partitions of the discretized variable.

To visualize the monotonic relationship between the tree outputs and the target, we created a plot of the bins predicted by the tree versus the mean target value for each of these bins. We used the **pandas groupby()** method to group the observations per bin and then calculated the target mean in each bin. With **pandas plot()**, we plotted the relationship between the bins and **target**.

To perform decision tree discretization using **Feature-engine**, we used **DecisionTreeDiscretiser()** indicating the cross-validation fold, the performance metric, and the hyperparameter space to optimize each tree.

With the `fit()` method, the discretizer fit the best tree for each of the variables. With the `transform()` method, we obtained the discretized variables in the train and test sets.

There's more...

Feature-engine's implementation is inspired by the winning solution of the KDD 2009 data science competition. The winners created new features by obtaining predictions of decision trees based on continuous features. You can find more details in the *Winning the KDD Cup Orange Challenge with Ensemble Selection* article, on page 27 of the article series available at <http://www.mtome.com/Publications/CiML/CiML-v3-book.pdf>.

For a review of discretization techniques, you might find the following articles useful:

- Dougherty et al, *Supervised and Unsupervised Discretization of Continuous Features, Machine Learning: Proceedings of the 12th International Conference*, 1995,
(<https://ai.stanford.edu/~ronnyk/disc.pdf>).
- Lu et al, *Discretization: An Enabling Technique, Data Mining, and Knowledge Discovery*, 6, 393–423, 2002,
(https://www.researchgate.net/publication/220451974_Discretization_An_Enabling_Technique).
- Garcia et al, *A Survey of Discretization Techniques: Taxonomy and Empirical Analysis in Supervised Learning, IEEE Transactions on Knowledge in Data Engineering* 25 (4), 2013,
(<https://ieeexplore.ieee.org/document/6152258>).

5

Working with Outliers

An outlier is a data point that is significantly different from the remaining data. Statistical parameters such as the mean and variance are sensitive to outliers. Outliers may also affect the performance of some machine learning models, such as linear regression models. In these cases, we may want to remove or engineer the outliers in our variables.

How can we engineer outliers? One way to handle outliers is to perform variable discretization with any of the techniques we covered in [Chapter 4](#), *Performing Variable Discretization*. With discretization, the outliers will fall in the lower or upper intervals and, therefore, will be treated as the remaining lower or higher values of the variable. An alternative way to handle outliers is to assume that the information is missing, treat the outliers together with the remaining missing data, and carry out any of the missing imputation techniques described in [Chapter 1](#), *Imputing Missing Data*. We can also remove observations with outliers from the dataset or cap the maximum and minimum values of the variables, as we will discuss throughout this chapter.

NOTE

Sometimes, outliers can be very informative. In these cases, we don't want to remove them from the data. This chapter does not focus on these scenarios. The techniques discussed in this chapter are

tailored to the management of outliers in those cases where they negatively affect the performance of the machine learning models.

In this chapter, we will first discuss how to identify outliers in a dataset, and then we will describe how to remove outliers or how to cap variables at maximum or minimum values.

This chapter will cover the following recipes:

- Visualizing outliers with boxplots
- Finding outliers using the mean and standard deviation
- Finding outliers with the interquartile range proximity rule
- Removing outliers
- Capping or censoring outliers
- Capping outliers using quantiles

Technical requirements

In this chapter, we will use the Python libraries NumPy, pandas, Matplotlib, seaborn, and Feature-engine.

Visualizing outliers with boxplots

In this recipe, we will identify outliers using boxplots. Boxplots produce a box that encloses the observations within the 75th and 25th quantiles, or in other words, within the **Inter-Quartile Range (IQR)**. The IQR is given through the following equation:

$$IQR = 75th \text{ quantile} - 25th \text{ quantile}$$

According to the IQR proximity rule, a value is an outlier if it falls outside the following boundaries:

$$\text{Upper boundary} = 75th \text{ quantile} + (IQR * 1.5)$$

In a boxplot, these boundaries are indicated by their whiskers. Thus, values outside the whiskers are considered outliers. Outliers are highlighted with asterisks.

How to do it...

We will create the boxplots utilizing the **Seaborn** library. Let's begin the recipe by importing the Python libraries and loading the dataset as follows:

1. Import the required Python libraries as follows:

```
import pandas as pd  
  
import matplotlib.pyplot as plt  
  
import seaborn as sns  
  
sns.set(style="darkgrid")  
  
from sklearn.datasets import fetch_california_housing
```

2. Let's load the California house prices dataset from **scikit-learn** as follows:

```
x, y = fetch_california_housing()
```

```
return_X_y=True, as_frame=True)
```

3. Let's make a boxplot of the **MedInc** variable to visualize outliers as follows:

```
plt.figure(figsize=(3,6))  
sns.boxplot(y=X["MedInc"])  
plt.title("Boxplot")  
plt.show()
```

The outliers are the asterisks sitting outside the whiskers, which delimit IQR proximity rule boundaries.

Boxplot

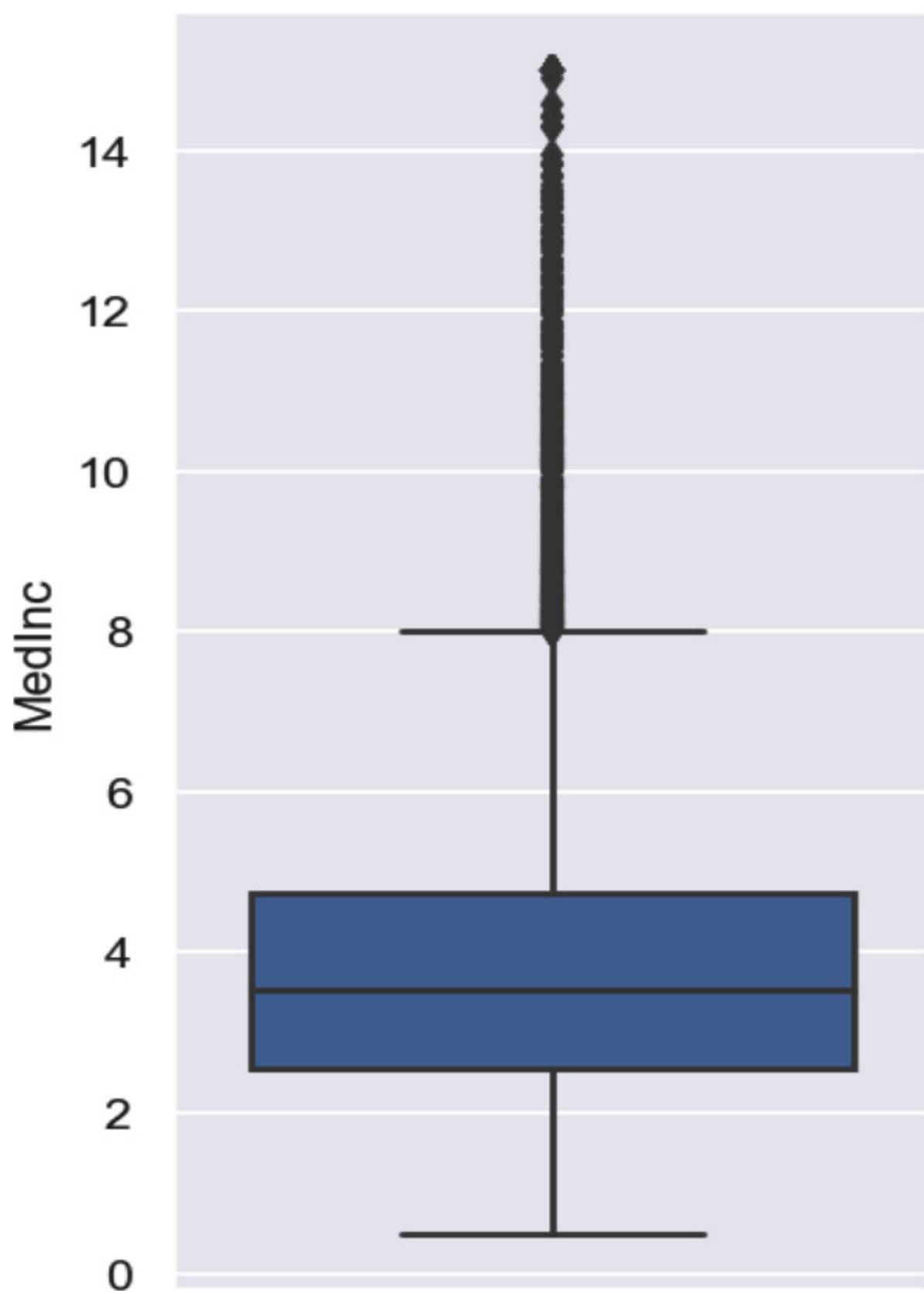


Figure 5.1 – A boxplot

4. Let's now create a function to plot a boxplot on top of a histogram:

```
def plot_boxplot_and_hist(data, variable):
    #figure composed of two matplotlib.Axes
    objects (ax_box and ax_hist)
    f, (ax_box, ax_hist) = plt.subplots(
        2, sharex=True,
        gridspec_kw={"height_ratios": (0.50, 0.85)}
    )
    # assigning a graph to each ax
    sns.boxplot(x=data[variable], ax=ax_box)
    sns.histplot(data=data, x=variable, ax=ax_hist)
    # Remove x axis name for the boxplot
    ax_box.set(xlabel='')
    plt.title(variable)
    plt.show()
```

5. Let's now use the previous function to create the plots for the **MedInc** variable:

```
plot_boxplot_and_hist(X, "MedInc")
```

We see in the following plot that most outliers lie on the right tail, or higher values, of the distribution as people with high-income salaries are unusual in this dataset:

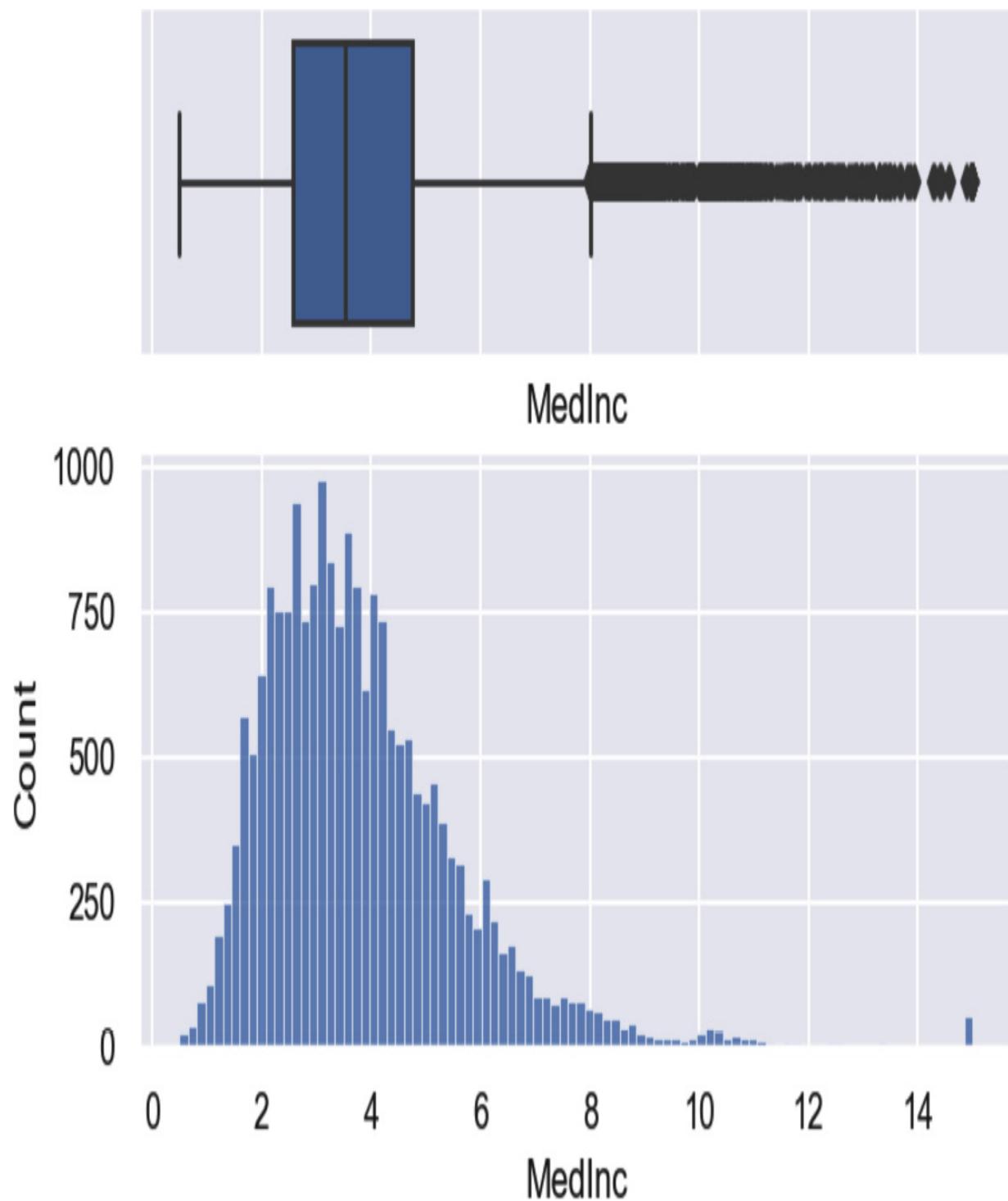


Figure 5.2 – A boxplot and a histogram

That's it! We now know how to identify outliers visually utilizing boxplots.

How it works...

In this recipe, we used the **boxplot** method from **seaborn** to create the boxplots. In the y-axis of the plot, we obtained the values of the variable. The asterisks highlighted extremely high values compared with the rest of the observations.

We then combined **seaborn's boxplot** with **seaborn's histplot** to create a histogram under the boxplot. The plots shared the x-axis with the values of the observations. The y-axis of the histogram contained the number of observations per interval. The variable was left-skewed, with most of the values in the lower range. The outliers lie only on the right tail of the distribution.

Finding outliers using the mean and standard deviation

In normally distributed variables, more than 99% of the observations lie within the interval comprising the mean plus or minus three times the standard deviation. Thus, any values beyond those limits can be considered outliers. In this recipe, we will identify outliers as those observations that lie outside of this interval.

How to do it...

Let's begin the recipe by importing the Python libraries and loading the dataset:

1. Import the required Python libraries:

```
import numpy as np  
import pandas as pd  
from sklearn.datasets import load_breast_cancer
```

2. Let's load the Breast Cancer dataset from **scikit-learn**:

```
breast_cancer = load_breast_cancer()  
  
X = pd.DataFrame(  
    breast_cancer.data,  
    columns=breast_cancer.feature_names  
)
```

3. Let's create a function that returns the mean plus and minus **fold** times the standard deviation, where **fold** is a parameter to the function:

```
def find_limits(df, variable, fold):  
    lower_limit = df[variable].mean() - fold *  
        df[variable].std()  
    upper_limit = df[variable].mean() + fold *  
        df[variable].std()  
    return lower_limit, upper_limit
```

4. Let's use the function to capture the extreme limits of the **mean smoothness** variable, which follows approximately a Gaussian distribution:

```
lower_limit, upper_limit = find_limits(  
    X, "mean smoothness", 3)
```

5. If we now execute `lower_limit` or `upper_limit`, we will see the values **0.0541** and **0.13855**, corresponding to the limits beyond which we can consider a value an outlier.
6. Let's create a Boolean vector that flags observations with values beyond the limits:

```
outliers = np.where(  
    (X["mean smoothness"] > upper_limit) |  
    (X["mean smoothness"] < lower_limit),  
    True,  
    False,  
)
```

If we now execute `outliers.sum()`, we will see the value **5**, indicating that there are five outliers or observations that are smaller or greater than the extreme values found with the mean and the standard deviation.

How it works...

With the **pandas** `mean()` and `std()` methods, we captured the mean and standard deviation of the variable. To find the outliers, we used **NumPy**'s `where()` method, which produced a Boolean vector with **True** if the value was an outlier. The `where()` function scanned the rows of the variable, and if the value was greater than the upper limit, or smaller than the lower limit, it was assigned **True**, and alternatively **False**. Finally, we used **pandas** `sum()` over this Boolean vector to calculate the total number of outliers.

Finding outliers with the interquartile range proximity rule

If the variables are not normally distributed variables, we can identify outliers utilizing the IQR proximity rule. According to the IQR rule, data points that fall below the 25th quantile - 1.5 times the IQR, or beyond the 75th quantile + 1.5 times the IQR, are outliers.

NOTE

We described the IQR in the *Visualizing outliers with boxplots* recipe.

In this recipe, we will identify outliers utilizing the IQR proximity rule.

How to do it...

Let's begin the recipe by importing the Python libraries and loading the dataset:

1. Import the required Python libraries:

```
import numpy as np  
import pandas as pd  
from sklearn.datasets import fetch_california_housing
```

2. Let's load the California housing dataset from **scikit-learn**:

```
x, y = fetch_california_housing(  
    return_X_y=True, as_frame=True)
```

3. Let's create a function that returns the 25th quantile - 1.5 times the IQR, or the 75th quantile + 1.5 times the IQR:

```
def find_limits(df, variable, fold):  
    IQR = df[variable].quantile(0.75) - df[variable].  
    quantile(0.25)  
    lower_limit = df[variable].quantile(0.25) - (IQR *  
    fold)  
    upper_limit = df[variable].quantile(0.75) + (IQR *  
    fold)  
    return lower_limit, upper_limit
```

4. Let's use the function to capture the extreme limits of the **MedInc** variable:

```
lower_limit, upper_limit = find_limits(X, "MedInc", 3)
```

If we now execute **lower_limit** or **upper_limit**, we will see the values **-3.9761** and **11.2828**, corresponding to the limits beyond which we can consider a value an outlier.

5. Let's create a Boolean vector with **True** for values beyond the limits:

```
outliers = np.where(  
    (X["MedInc"] > upper_limit) |  
    (X["MedInc"] < lower_limit),  
    True,  
    False,  
)
```

If we now execute `outliers.sum()`, we will see the value **140**, indicating that there are 140 outliers or observations that are smaller or greater than the extreme values found with the IQR proximity rule.

How it works...

The **pandas quantile()method** calculates any quantile of the variable. We calculated the 25th and 75th quantile by passing 0.25 and 0.75 as arguments to quantile. The rest of the recipe is exactly as we described in the previous recipe, *Finding outliers using the mean and standard deviation*.

Removing outliers

Trimming, or truncating, is the process of removing observations with outliers in one or more variables in the dataset. There are three commonly used methods to set the boundaries beyond which a value can be considered an outlier. If the variable is normally distributed, the boundaries are given by the mean plus or minus three times the standard deviation, as approximately 99% of the data will be distributed between those limits. For normally as well as not normally distributed variables, we can determine the limits using the IQR proximity rule or by directly setting the limits to the 5th and 95th quantiles. In this recipe, we are going to use the IQR proximity rule to identify and then remove outliers, using pandas, and then we will automate this process for multiple variables, utilizing Feature-engine.

How to do it...

Let's first import the Python libraries and load the data:

1. Let's import the Python libraries, functions, and classes:

```
import numpy as np  
  
import pandas as pd  
  
from sklearn.datasets import fetch_california_housing  
  
from sklearn.model_selection import train_test_split  
  
from feature_engine.outliers import OutlierTrimmer
```

2. Let's load the California housing dataset from **scikit-learn** and separate it into a training and a testing set:

```
X, y = fetch_california_housing(  
    return_X_y=True, as_frame=True)  
  
X_train, X_test, y_train, y_test = train_test_split(  
  
    X,  
  
    y,  
  
    test_size=0.3,  
  
    random_state=0,  
  
)
```

3. Let's create a function to find the boundaries of a variable distribution using the interquartile range proximity rule:

```
def find_limits(df, variable, fold):  
  
    IQR = df[variable].quantile(0.75) - df[variable].
```

```
quantile(0.25)

lower_limit = df[variable].quantile(0.25) - (IQR *
fold)

upper_limit = df[variable].quantile(0.75) + (IQR *
fold)

return lower_limit, upper_limit
```

NOTE

We can replace the code in the preceding function with that described in the *Finding outliers using the mean and standard deviation* recipe to find the boundaries using the mean and the standard deviation.

Alternatively, we can set the lower limit to

df[variable].quantile(0.05) and the upper limit to
df[variable].quantile(0.95).

4. Let's use the function from *step 3* to determine the limits of the **MedInc** variable:

```
lower_limit, upper_limit = find_limits(X_train,
"MedInc", 3)
```

5. Let's retain the observations whose value is greater than or equal to (**ge**) the lower limit in the train and test sets:

```
inliers = X_train["MedInc"].ge(lower_limit)

X_train = X_train.loc[inliers]

inliers = X_test["MedInc"].ge(lower_limit)

X_test = X_test.loc[inliers]
```

6. Let's retain the observations whose value is lower than or equal to (`le`) the upper limit:

```
inliers = X_train["MedInc"].le(upper_limit)  
X_train = X_train.loc[inliers]  
inliers = X_test["MedInc"].le(upper_limit)  
X_test = X_test.loc[inliers]
```

With `pd.shape`, we can corroborate that the training and testing sets are smaller in size after removing the outliers.

Now, we will remove outliers across multiple variables utilizing **Feature-engine**. First, we need to separate the data into a training and a testing set as we did in *step 2*.

7. Let's set up a transformer to remove outliers in three variables by determining the limits with the IQR rule:

```
trimmer = OutlierTrimmer(  
    variables = ["MedInc", "HouseAge", "Population"],  
    capping_method="iqr",  
    tail="both",  
    fold=1.5,  
)
```

8. Let's fit the transformer to the data so that it learns those limits:

```
trimmer.fit(X_train)
```

By executing `trimmer.left_tail_caps_`, we can visualize the lower limits for the three variables: `{'MedInc': -0.6776500000000012, 'HouseAge': -10.5, and 'Population': -626.0}`. By executing `trimmer.right_tail_caps_`, we can see the upper limits for those variables: `{'MedInc': 7.984350000000001, 'HouseAge': 65.5, and 'Population': 3134.0}`.

9. Finally, let's remove outliers from the train and test sets:

```
X_train_enc = trimmer.transform(X_train)  
X_test_enc = trimmer.transform(X_test)
```

To finish with the recipe, let's compare the distribution of a variable before and after removing outliers.

10. Let's import `matplotlib` and `seaborn` and create a function to display a boxplot on top of a histogram:

```
import matplotlib.pyplot as plt  
import seaborn as sns  
  
def plot_boxplot_and_hist(data, variable):  
    f, (ax_box, ax_hist) = plt.subplots(  
        2, sharex=True, gridspec_kw={"height_ratios":  
            (0.50, 0.85)}  
    )  
  
    sns.boxplot(x=data[variable], ax=ax_box)  
    sns.histplot(data=data, x=variable, ax=ax_hist)  
    ax_box.set(xlabel="")
```

```
plt.title(variable)  
plt.show()
```

NOTE

We discussed the code in *step 10* in the *Visualizing outliers with boxplots* recipe earlier in this chapter.

11. Let's now display the distribution of the **MedInc** variable before removing the outliers:

```
plot_boxplot_and_hist(X_train, "MedInc")
```

In the following plot, we see that **MedInc** is skewed and presents outliers at the maximum values:

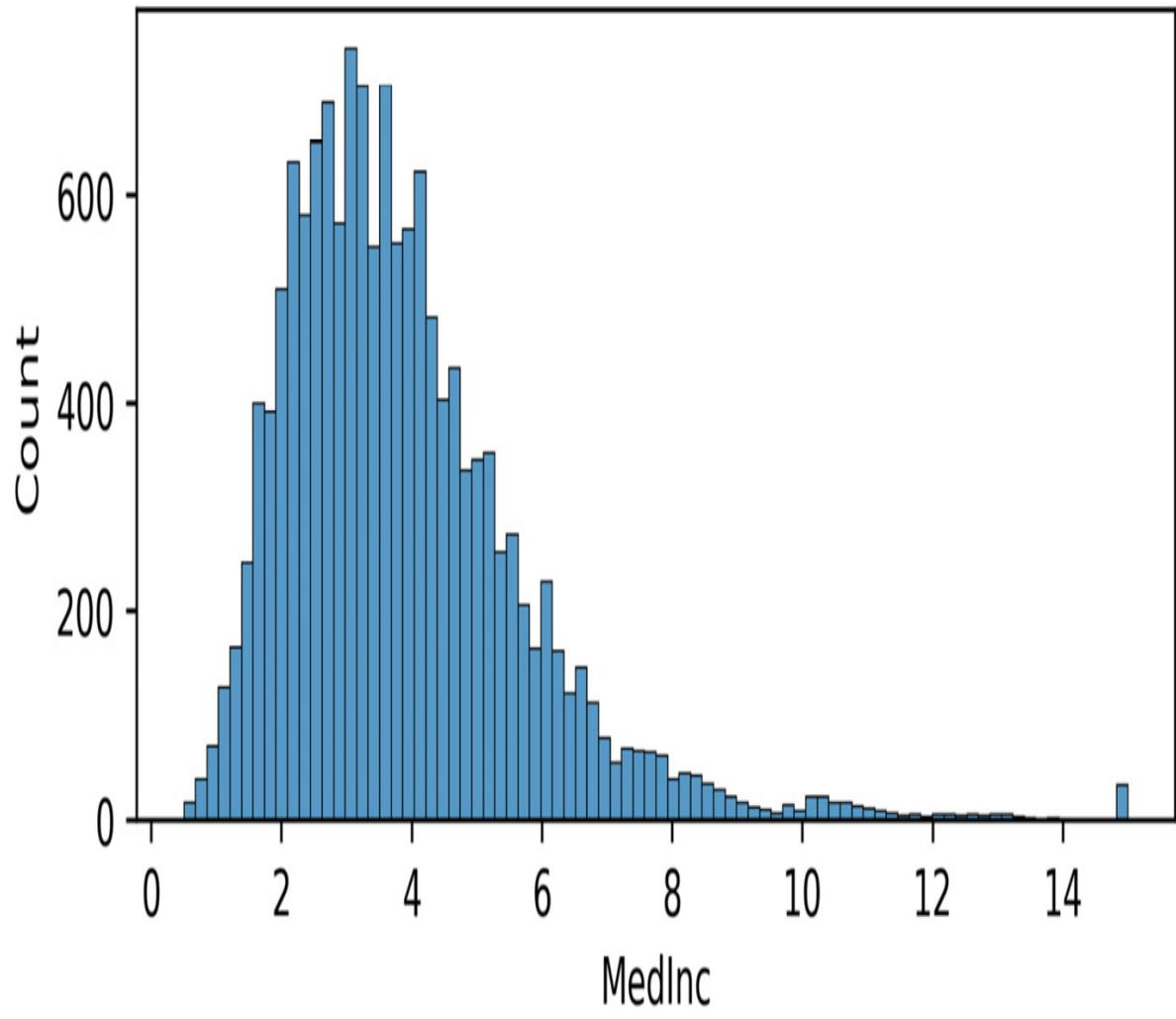
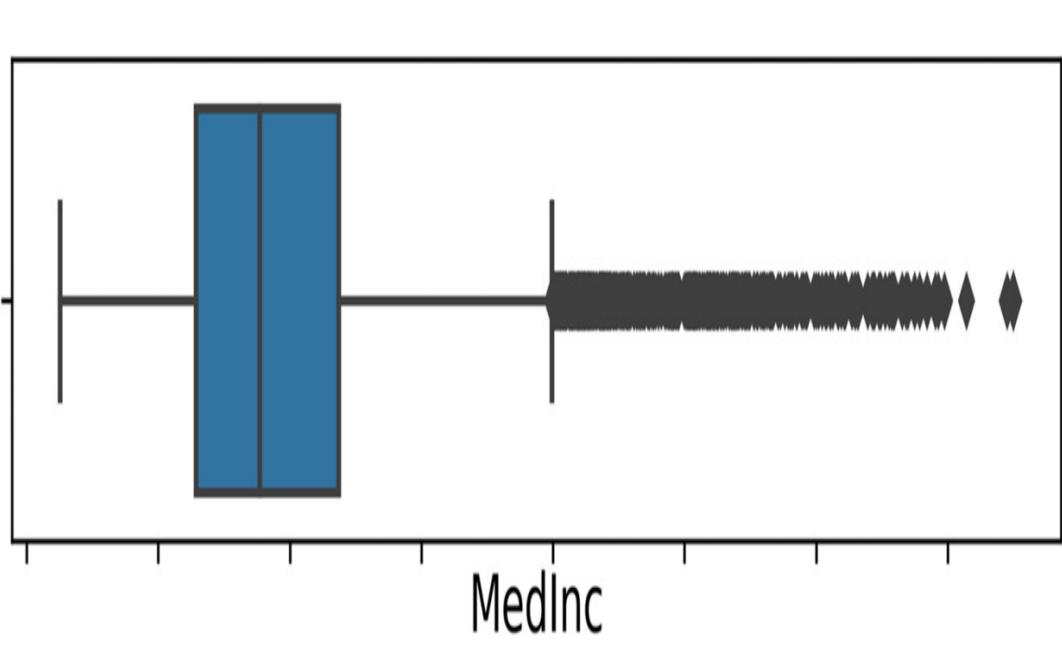


Figure 5.3 – The boxplot and the histogram before removing outliers

12. Finally, let's display the distribution of the same variable after removing the outliers:

```
plot_boxplot_and_hist(X_train_enc, "MedInc")
```

In the following plot, we see that we've removed the extreme maximum values of **MedInc**, and note that the histogram seems more evenly distributed:

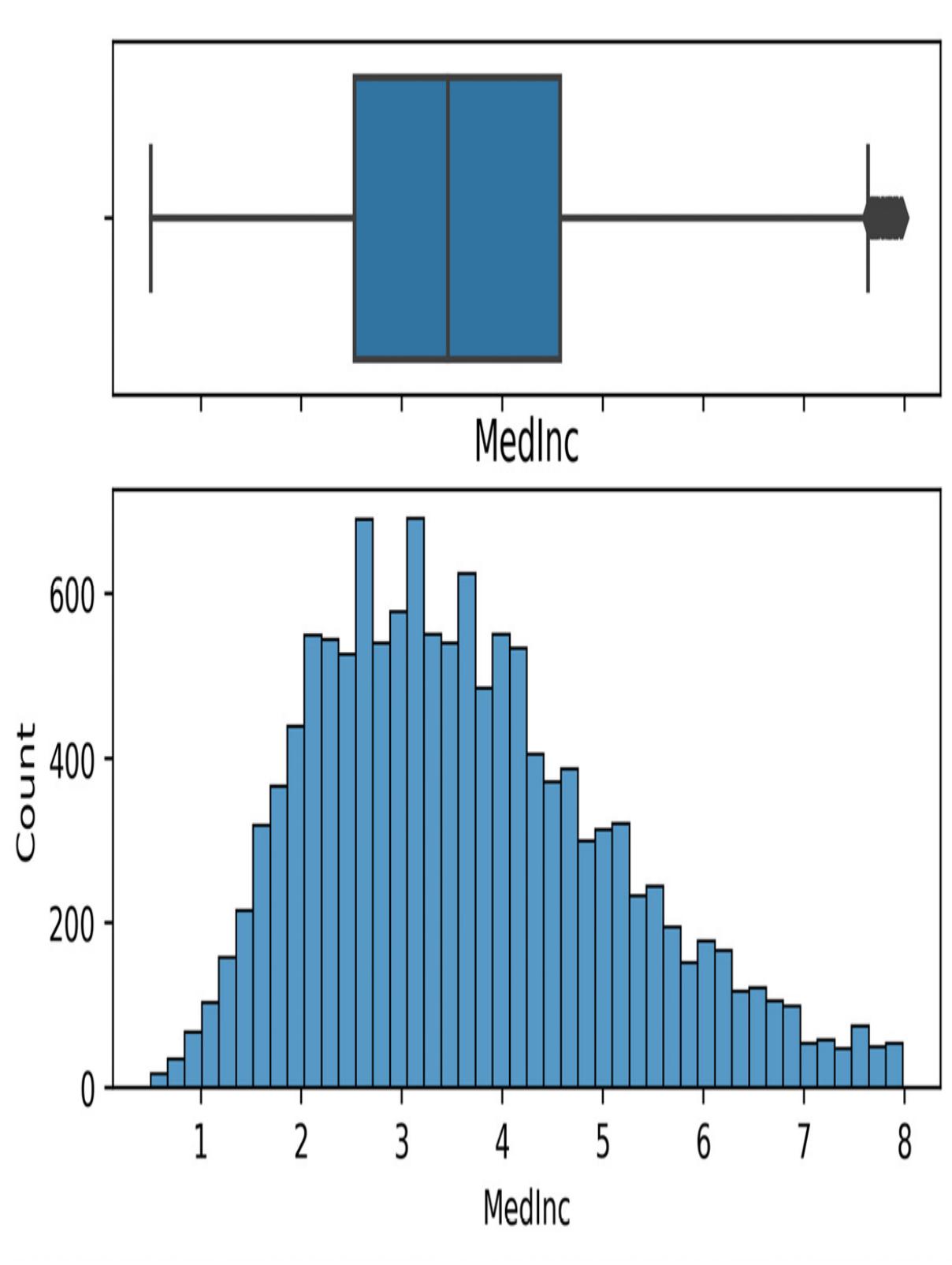


Figure 5.4 – The boxplot and the histogram after removing outliers

Note that the boxplot still highlights outliers in the variable. This is not uncommon. When we remove outliers, the values of the median and quantiles change and, therefore, values that were not flagged as outliers may now be beyond the whiskers.

How it works...

We created a function to find the boundaries according to the IQR proximity rule, as described in the *Finding outliers with the inter-quartile range proximity rule* recipe. With **pandas ge()**, we created a Boolean vector with **True** for the observations that are greater than or equal (that is, **ge**) to the lower limit. With **pandas le()**, we created another Boolean vector with **True** for those observations whose values are lower than or equal (that is, **le**) to the upper limit. With **pandas loc** and the Boolean vectors, we removed the outliers from the data.

We used **Feature-engine** to remove outliers simultaneously from multiple variables. The **OutlierTrimmer()** transformer can identify outliers based on the mean and standard deviation, IQR proximity rule, or quantiles. It can also change the width of the inliers interval through the **fold** parameter that multiplies the IQR or the standard deviation, and it can find outliers at both tails of the distribution or just at one end. By setting the **capping_method='iqr'** parameter, **OutlierTrimmer()** calculated the boundaries using the IQR proximity rule. With **fold** set to **3**, the IQR was multiplied by three to find the limits. With **tails** set to "**both**", outliers were flagged at both ends of the variable's distribution. To cap outliers at either end, we can pass "**left**" or "**right**" to the **tails** argument. With the **fit()** method, the transformer learned and stored the limits for each

variable. With the **transform()** method, it removed the outliers from the data.

Capping or censoring outliers

Capping or censoring is the process of transforming the data by limiting the extreme values, as in the outliers, to a certain maximum or minimum arbitrary value. With this procedure, the outliers are not removed but are instead replaced by other values. A typical strategy involves setting outliers to a specified percentile. For example, we can set all data below the 5th percentile to the value at the 5th percentile and all data greater than the 95th percentile to the value at the 95th percentile. Alternatively, we can cap the variable at the limits determined by the IQR proximity rule or at the mean plus and minus three times the standard deviation. In this recipe, we will cap variables at arbitrary values determined by the mean plus and minus three times the standard deviation using **pandas** and **Feature-engine**.

How to do it...

Let's first import the Python libraries and load the data:

1. Import the required Python libraries:

```
import numpy as np  
  
import pandas as pd  
  
from sklearn.datasets import load_breast_cancer  
  
from sklearn.model_selection import train_test_split  
  
from feature_engine.outliers import Winsorizer
```

2. Let's load the Breast Cancer dataset from **scikit-learn** and then separate it into a train and a test set:

```
X = pd.DataFrame(  
    breast_cancer.data,  
    columns=breast_cancer.feature_names)  
  
y = breast_cancer.target  
  
X_train, X_test, y_train, y_test = train_test_split(  
    X,  
    y,  
    test_size=0.3,  
    random_state=0,  
)
```

3. Let's create a function that returns the mean plus and minus **fold** times the standard deviation, where **fold** is a parameter to the function:

```
def find_limits(df, variable, fold):  
    lower_limit = df[variable].mean() - fold *  
        df[variable].std()  
    upper_limit = df[variable].mean() + fold *  
        df[variable].std()  
    return lower_limit, upper_limit
```

4. Let's use the function to capture the extreme limits of the **mean smoothness** variable, which follows approximately a Gaussian distribution:

```
lower_limit, upper_limit = find_limits(X_train, "worst  
smoothness", 3)  
  
lower_limit, upper_limit
```

5. Now let's cap the values at the lower and upper limits:

```
X_train["worst smoothness"].clip(lower=lower_limit,  
upper=upper_limit, inplace=True)  
  
X_test["worst smoothness"].clip(lower=lower_limit,  
upper=upper_limit, inplace=True)
```

6. Let's check that the values were censored:

```
X_train["worst smoothness"].min(), X_train["worst  
smoothness"].max()
```

The previous command returns the minimum and maximum values of the variable, which now coincide with the upper and lower limits we identified in *step 4: (0.07117, 0.20149734880520967)*.

Now, we will cap multiple variables simultaneously utilizing Feature-engine. First, we need to separate the data into a training and testing set as we did in *step 2*.

7. Let's set up a transformer to cap two variables at the limits determined with the mean and standard deviation:

```
capper = Winsorizer(  
  
    variables=["worst smoothness", "worst texture"],  
    capping_method="gaussian",  
    tail="both",
```

```
    fold=3,  
)  

```

8. Let's fit the transformer to the data so that it learns those limits:

```
capper.fit(X_train)
```

By executing `capper.left_tail_caps_`, we can visualize the lower limits for the three variables: `{'worst_smoothness': 0.06356074164705164, 'worst_texture': 7.092123638591499}`. By

executing `capper.right_tail_caps_`, we can see the upper limits for those variables: `{'worst_smoothness': 0.20149734880520967, 'worst_texture': 43.97692158753917}`.

9. Finally, let's remove outliers from the train and test sets:

```
X_train = capper.transform(X_train)  
X_test = capper.transform(X_test)
```

If we now execute `X_train[capper.variables_].max()`, we'll see the maximum values coincide with the upper limits:

```
worst_smoothness      0.201497  
worst_texture        43.976922  
dtype: float64
```

That's it! We've now capped variables at the minimum and maximum values.

How it works...

With the function described in the *Finding outliers using the mean and standard deviation* recipe, we identified the outliers. With **pandas clip()**, we capped the values of the variable at those values.

We used **Feature-engine** to cap outliers in multiple variables simultaneously. The **winsorizer()** transformer can identify outliers based on the mean and standard deviation, the IQR proximity rule, or by using percentiles. It can also change the width of the interval through the **fold** parameter that multiplies the IQR or the standard deviation, and it can find outliers at both tails of the distribution or just one.

By setting the **capping_method='gaussian'** parameter, **Winsorizer()** calculated the boundaries using the mean and the standard deviation. With **fold** set to **3**, the standard deviation was multiplied by three to find the limits. With **tails** set to "**both**", outliers were flagged at both ends of the variables distribution. With the **fit()** method, the transformer learned and stored the limits for each variable. With the **transform()** method, it capped the variable values at those limits. Finally, we used **pandas max()** to corroborate that the variables were capped.

There's more...

Winsorizer() caps the variables at values estimated using statistical parameters of the variables. Feature-engine also has the **ArbitraryOutlierCapper()** transformer, which allows us to cap variables at arbitrary values. To learn more about the arbitrary capper from Feature-engine, visit [https://feature-](https://feature-engine.readthedocs.io/en/latest/tutorials/capping_outliers.html)

engine.readthedocs.io/en/latest/api_doc/outliers/ArbitraryOutlierCapper.html.

Capping outliers using quantiles

When capping outliers, we clip the variable extreme values to a certain maximum or minimum value determined by some statistical parameter. A typical strategy involves setting outliers to a specified percentile. For example, we can set all data below the 5th percentile to the value at the 5th percentile and all data greater than the 95th percentile to the value at the 95th percentile. In this recipe, we will cap variables at arbitrary values determined by the percentiles using **pandas** and **Feature-engine**.

How to do it...

Let's first import the Python libraries and load the data:

1. Import the required Python libraries:

```
import pandas as pd

from sklearn.datasets import load_breast_cancer

from sklearn.model_selection import train_test_split

from feature_engine.outliers import Winsorizer
```

2. Let's load the Breast Cancer dataset from **scikit-learn**:

```
breast_cancer = load_breast_cancer()

X = pd.DataFrame(
    breast_cancer.data,
```

```
columns=breast_cancer.feature_names)

y = breast_cancer.target
```

3. Let's separate the data into a train set and a test set:

```
x_train, x_test, y_train, y_test = train_test_split(
    x,
    y,
    test_size=0.3,
    random_state=0,
)
```

4. Let's cap the variable's minimum and maximum values to the 5th and 95th percentile:

```
for variable in x_train.columns:
    lower_limit = x_train[variable].quantile(0.05)
    upper_limit = x_train[variable].quantile(0.95)
    x_train[variable].clip(
        lower=lower_limit, upper=upper_limit, inplace=True)
    x_test[variable].clip(
        lower=lower_limit, upper=upper_limit, inplace=True)
```

That's it, now the variables have been clipped to their respective 5th and 95th percentiles.

We will cap multiple variables simultaneously utilizing Feature-engine. First, we need to separate the data into a training set and a testing set as we

did in *step 2*.

5. Let's set up a transformer to cap two variables at the 5th and 95th percentiles:

```
capper = Winsorizer(  
    variables = ["worst smoothness", "worst texture"],  
    capping_method="quantiles",  
    tail="both",  
    fold=0.05,  
)
```

TIP

To cap variables at the 10th and 90th percentiles, we need to set **fold=0.1**.

6. Let's fit the transformer to the data so that it learns those limits:

```
capper.fit(X_train)
```

By executing **capper.left_tail_caps_**, we can visualize the lower limits for the three variables: **{'worst smoothness': 0.0960535, 'worst texture': 16.7975}**. By executing **capper.right_tail_caps_**, we can see the upper limits for those variables: **{'worst smoothness': 0.1732149999999998, 'worst texture': 36.2775}**.

7. Finally, let's remove outliers from the train and test sets:

```
X_train = capper.transform(X_train)  
X_test = capper.transform(X_test)
```

If we now execute `X_train[capper.variables_].max()`, we'll see the maximum values coincide with the upper limits:

```
worst smoothness      0.173215  
worst texture        36.277500  
dtype: float64
```

That's it! We've now capped variables at the minimum and maximum values.

How it works...

With `pandas quantile()`, we found the 5th and 95th percentiles of each variable. With `pandas clip()`, we capped the values of the variable at those values.

We used `Feature-engine` to cap outliers in multiple variables simultaneously based on quantiles by setting `capping_method` to `"quantiles"` and `fold` to `0.05`. With `tails` set to `"both"`, outliers were capped at both ends of the variables distribution. With the `fit()` method, the transformer learned and stored the limits for each variable. With the `transform()` method, it capped the variable values at those limits. Finally, we used `pandas max()` to corroborate that the variables were capped.

6

Extracting Features from Date and Time Variables

Date and time variables are those that contain information about dates, times, or both. In programming, we refer to these variables as **datetime** variables. Examples of **datetime** variables include date of birth, the time of an event, and date of last payment. The cardinality of **datetime** variables is usually very high. This means they contain a multitude of unique values, each corresponding to a specific combination of date and/or time. Therefore, we do not utilize **datetime** variables in their raw format in machine learning models. Instead, we enrich the dataset by extracting multiple features from these variables. In this chapter, we will learn how to extract new features from date and time by utilizing the pandas **dt** module. Later on, we will automate feature extraction over multiple variables with Feature-engine.

This chapter will cover the following recipes:

- Extracting features from dates with pandas
- Extracting features from time with pandas
- Capturing elapsed time between **datetime** variables
- Working with time in different time zones
- Automating feature extraction with Feature-engine

Technical requirements

In this chapter, we will use the pandas, NumPy, and Feature-engine Python libraries.

Extracting features from dates with pandas

datetime variables can take dates, time, or dates and time as values. They are not used in their raw format to build machine learning algorithms.

Instead, we create additional features from them, and we can enrich the dataset dramatically by extracting information from the date and time.

The **pandas** Python library contains a lot of capabilities for working with dates and time. pandas **dt** is the **accessor** object to the **datetime** properties of a pandas Series. To access the pandas **dt** functionality, the variables should be cast in a data type that supports these operations, such as **datetime** or **timedelta**.

TIP

Often, the **datetime** variables are cast as objects, particularly when the data is loaded from a CSV file. Therefore, to extract the date and time features that we will discuss throughout this chapter, it is necessary to recast the variables as **datetime**.

In this recipe, we will learn how to extract features from dates by utilizing pandas.

Getting ready

The following are some of the features that we can extract from the date part of the `datetime` variable off the shelf using pandas:

- `pandas.Series.dt.year`
- `pandas.Series.dt.quarter`
- `pandas.Series.dt.month`
- `pandas.Series.dt.isocalendar().week`
- `pandas.Series.dt.day`
- `pandas.Series.dt.day_of_week`
- `pandas.Series.dt.weekday`
- `pandas.Series.dt.dayofyear`
- `pandas.Series.dt.day_of_year`

We can use the features we've obtained with pandas to create even more features, such as the semester or whether it is the weekend. We will learn how to do this in the next section.

How to do it...

To proceed with the recipe, let's import pandas and NumPy, and create a toy DataFrame:

1. Import `pandas` and `numpy`:

```
import numpy as np  
import pandas as pd
```

2. Let's create 20 **datetime** values beginning from **2019-03-05** at midnight, followed by increments of 1 day. Then, we must capture the values in a DataFrame and display the top five rows:

```
rng_ = pd.date_range("2019-03-05", periods=20,  
freq="D")  
  
data = pd.DataFrame({"date": rng_})  
  
data.head()
```

Our variable contains only dates, as shown in the following output:

	date
0	2019-03-05
1	2019-03-06
2	2019-03-07
3	2019-03-08
4	2019-03-09

Figure 6.1 – First five rows of a toy DataFrame with a datetime variable

TIP

We can check the data format of the variable by executing **data["date"].dtypes**. If the variable is cast as an object, we can convert it into **datetime** format by executing **data["date_dt"] = pd.to_datetime(data["date"])**.

3. Let's extract the year part of the date in a new column and display the top five rows of the resulting DataFrame:

```
data["year"] = data["date"].dt.year  
data.head()
```

We can see the new variable, **year**, in the following screenshot:

	date	year
0	2019-03-05	2019
1	2019-03-06	2019
2	2019-03-07	2019
3	2019-03-08	2019
4	2019-03-09	2019

Figure 6.2 – First five rows of a toy DataFrame with the new year variable

4. Let's extract the quarter out of the date into a new column and display the top five rows:

```
data["quarter"] = data["date"].dt.quarter  
data[["date", "quarter"]].head()
```

We can see the new variable, **quarter**, in the following output:

	date	quarter
0	2019-03-05	1
1	2019-03-06	1
2	2019-03-07	1
3	2019-03-08	1
4	2019-03-09	1

Figure 6.3 – First five rows of a toy DataFrame with the new quarter variable

5. With **quarter**, we can now create the **semester** feature:

```
data["semester"] = np.where(data["quarter"]<3, 1, 2)
```

TIP

To familiarize yourself with the distinct values of the new variables, you can use pandas **unique()** – for example,

df["quarter"].unique() or **df["semester"].unique()**.

6. Let's extract the month part of the date in a new column and display the top five rows of the DataFrame:

```
data["month"] = data["date"].dt.month
data[["date", "month"]].head()
```

We can see **month** in the newly created variable in the following output:

	date	month
0	2019-03-05	3
1	2019-03-06	3
2	2019-03-07	3
3	2019-03-08	3
4	2019-03-09	3

Figure 6.4 – First five rows of a toy DataFrame with the new month variable

7. We can extract the week number (in a year) that the date corresponds to like so:

```
data["week"] = data["date"].dt.isocalendar().week
```

We can see the **week** variable in the following output:

	date	week
0	2019-03-05	10
1	2019-03-06	10
2	2019-03-07	10
3	2019-03-08	10
4	2019-03-09	10

Figure 6.5 – First five rows of a toy DataFrame with the new week variable

8. Let's extract the day of the month, which can take values between 1 and 31, and capture it in a new column. Then, we will display the top rows

of the DataFrame:

```
data["day_mo"] = data["date"].dt.day  
data[["date", "day_mo"]].head()
```

We can see the day of the month in the new variable in the following output:

	date	day_mo
0	2019-03-05	5
1	2019-03-06	6
2	2019-03-07	7
3	2019-03-08	8
4	2019-03-09	9

Figure 6.6 – First five rows of a toy DataFrame with the new variable

9. Let's extract the day of the week, with values between 0 and 6 (from Monday to Sunday), in a new column, then display the top rows:

```
data["day_week"] = data["date"].dt.dayofweek  
data[["date", "day_mo", "day_week"]].head()
```

We can see the day of the week in the new variable in the following output:

	date	day_mo	day_week
0	2019-03-05	5	1
1	2019-03-06	6	2
2	2019-03-07	7	3
3	2019-03-08	8	4
4	2019-03-09	9	5

Figure 6.7 – First five rows of a toy DataFrame with the new variables derived from days

10. Next, let's create a binary variable that indicates whether the date was a weekend. Then, we will display the DataFrame's top rows:

```
data["is_weekend"] = (
    data["date"].dt.dayofweek > 4).astype(int)
data[["date", "day_week", "is_weekend"]].head()
```

We can see the new **is_weekend** variable in the following output:

	date	day_week	is_weekend
0	2019-03-05	1	0
1	2019-03-06	2	0
2	2019-03-07	3	0
3	2019-03-08	4	0
4	2019-03-09	5	1

Figure 6.8 – First five rows of a toy DataFrame with the new is_weekend variable

With that, we have created many new features from the date part of a **datetime** variable using pandas. These features are useful for data analysis, visualization, and machine learning.

How it works...

In this recipe, we extracted the many date-related features from a **datetime** variable by using the **dt** module from pandas. First, we created a toy DataFrame with a variable that contained both the date and time in its values. To create the toy DataFrame, we used the pandas **date_range()** method to create a range of values starting from an arbitrary date and increasing this by intervals of 1 day. With the **periods** argument, we indicated the number of values to create in the range – that is, the number of dates. With the **freq** argument, we indicated the size of the steps between the dates. We used **D** for days in our example. Finally, we transformed the date range into a DataFrame with the pandas **DataFrame()** class.

To extract the different parts of the date, we used pandas **dt** module to access the **datetime** properties of a pandas Series, and then utilized the different properties required: **year**, **month**, and **quarter** to capture the year, month, and quarter in new columns of the DataFrame, respectively. To find the semester, we used the **where()** method from NumPy in combination with the newly created **quarter** variable. NumPy's **where()** method scanned the values of this **quarter** variable. If they were smaller than 3, it returned a value of 1 for the first semester; otherwise, it returned a value of 2.

To extract the different representations of days and weeks, we utilized the `isocalendar().week`, `day`, and `dayofweek` properties. To create a binary variable indicating whether the date was a weekend, we used the `where()` method from NumPy in combination with the newly created `day_week` variable. NumPy's `where()` method scanned the name of each day, and if they were smaller than 5, it assigned a value of `0`; otherwise, it assigned a value of `1`. With that, we created multiple features that we can use for data analysis and machine learning.

There's more...

pandas also offers the following methods out of the box to create Boolean vectors to highlight the beginning or end of a month, a quarter, or a year, and also whether it is a leap year:

- `pandas.Series.dt.is_month_start`
- `pandas.Series.dt.is_month_end`
- `pandas.Series.dt.is_quarter_start`
- `pandas.Series.dt.is_quarter_end`
- `pandas.Series.dt.is_year_start`
- `pandas.Series.dt.is_year_end`
- `pandas.Series.dt.is_leap_year`
- `pandas.Series.dt.days_in_month`

We can also return the number of days in a specific month with `pd.dt.days_in_month` and the day in a year (from 1 to 365) with

`pd.dt.dayofyear`.

For more details, visit the pandas `datetime` documentation:

https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.xhtml#time-date-components.

See also

To learn how to create different `datetime` ranges with pandas `date_ranges()`, visit https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.xhtml#offset-aliases.

To learn more about pandas `dt`, visit <https://pandas.pydata.org/pandas-docs/stable/reference/series.xhtml#datetime-properties>.

Extracting features from time with pandas

Some events occur more often at certain times of the day – for example, fraudulent activity is more likely to occur during the night or early morning. Air pollutant concentration also changes with the time of the day, with peaks at rush hour when there are more vehicles on the streets. Therefore, deriving time features is extremely useful. In this recipe, we will extract different time parts of a `datetime` variable by utilizing pandas and NumPy.

Getting ready

The following are some of the features that we can extract offtheshelf using pandas:

- `pandas.Series.dt.hour`

- **pandas.Series.dt.minute**
- **pandas.Series.dt.second**

How to do it...

To proceed with this recipe, we must import the necessary libraries and create a toy dataset:

1. Let's import **pandas** and **numpy**:

```
import numpy as np  
import pandas as pd
```

2. Let's create 20 **datetime** observations, beginning from **2019-03-05** at midnight followed by increments of 1 hour, 15 minutes, and 10 seconds. Next, we must capture the range in a DataFrame and display the top five rows:

```
rng_ = pd.date_range("2019-03-05", periods=20,  
freq="1h15min10s")  
  
df = pd.DataFrame({"date": rng_})  
  
df.head()
```

In the following screenshot, we can see the variable we just created, with a date part and a time part, and the values increasing by intervals of 1 hour, 15 minutes, and 10 seconds:

	date
0	2019-03-05 00:00:00
1	2019-03-05 01:15:10
2	2019-03-05 02:30:20
3	2019-03-05 03:45:30
4	2019-03-05 05:00:40

Figure 6.9 – First five rows of a toy DataFrame with a datetime variable

3. Let's extract the hour, minute, and second parts of the time into three new columns, then display the DataFrame's top five rows:

```
df["hour"] = df["date"].dt.hour  
df["min"] = df["date"].dt.minute  
df["sec"] = df["date"].dt.second  
df.head()
```

We can see the different time parts in the new columns of the DataFrame:

	date	hour	min	sec
0	2019-03-05 00:00:00	0	0	0
1	2019-03-05 01:15:10	1	15	10
2	2019-03-05 02:30:20	2	30	20
3	2019-03-05 03:45:30	3	45	30
4	2019-03-05 05:00:40	5	0	40

Figure 6.10 – First five rows of the toy DataFrame with the variables derived from time

NOTE

Remember that pandas `dt` needs a `datetime` object to work. You can change the data type of an `object` variable into `datetime` using pandas `to_datetime()`.

4. Let's perform the same operations that we did in *step 3* but now in one line of code:

```
df[["h", "m", "s"]] = pd.DataFrame([(x.hour, x.minute,
                                         x.second) for x in df["date"]])

df.head()
```

We can see the newly created variables in the following screenshot:

	date	hour	min	sec	h	m	s
0	2019-03-05 00:00:00	0	0	0	0	0	0
1	2019-03-05 01:15:10	1	15	10	1	15	10
2	2019-03-05 02:30:20	2	30	20	2	30	20
3	2019-03-05 03:45:30	3	45	30	3	45	30
4	2019-03-05 05:00:40	5	0	40	5	0	40

Figure 6.11 – First five rows of the toy DataFrame with the variables derived from time

NOTE

Remember that you can display the unique values of a variable with pandas `unique()`, for example, by executing
`df['hour'].unique()`.

- Finally, let's create a binary variable that flags whether the event occurred in the morning, between 6 A.M. and 12 P.M:

```
df["is_morning"] = np.where(
    (df["hour"] < 12) & (df["hour"] > 6), 1, 0 )
df.head()
```

We can see the `is_morning` variable in the following screenshot:

	date	hour	min	sec	h	m	s	is_morning
0	2019-03-05 00:00:00	0	0	0	0	0	0	0
1	2019-03-05 01:15:10	1	15	10	1	15	10	0
2	2019-03-05 02:30:20	2	30	20	2	30	20	0
3	2019-03-05 03:45:30	3	45	30	3	45	30	0
4	2019-03-05 05:00:40	5	0	40	5	0	40	0

Figure 6.12 – First five rows of the toy DataFrame with the new variables derived from time

With that, we have extracted multiple features from the time part of a **datetime** variable. These features can be used for data analysis as well as to build machine learning models.

How it works...

In this recipe, we created features that capture representations of time. First, we created a toy DataFrame with a **datetime** variable. We used the pandas **date_range()** method to create a range of 20 values starting from an arbitrary date and increased this by intervals of 1 hour, 15 minutes, and 10 seconds. We used the '**1h15min10s**' string as the frequency term for the **freq** argument to indicate the desired increments. Next, we transformed the date range into a DataFrame with the pandas **DataFrame()** method.

To extract the different time parts, we used pandas `dt` to access the time properties: `hour`, `minute`, and `second`. To create a binary variable to indicate whether the time was in the morning, we used the `where()` method from NumPy in combination with the `hour` variable. NumPy's `where()` method scanned the `hour` variable; if its values were smaller than 12 and bigger than 6, it assigned a value of `1`; otherwise, it assigned a value of `0`. With these operations, we added several features to the DataFrame that can be used for data analysis and to train machine learning models.

There's more...

We can also extract microseconds and nanoseconds with the following pandas properties:

- `pandas.Series.dt.microsecond`
- `pandas.Series.dt.nanosecond`

For more details, visit https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.xhtml#time-date-components.

Capturing the elapsed time between datetime variables

`datetime` variables offer value individually, though they may offer additional value collectively when used with other `datetime` variables to derive important insights. A common example consists of deriving the `age` at the time of an event from the `date of birth` and `date of event` variables. We can combine several `datetime` variables to derive the time that has

passed and create more meaningful features. In this recipe, we will learn how to capture the time between two **datetime** variables in different formats and the time between a **datetime** variable and the current time by utilizing pandas, NumPy, and the **datetime** library.

How to do it...

To proceed with this recipe, we must import the necessary libraries and create a toy dataset:

1. Let's begin by importing **pandas**, **numpy**, and **datetime**:

```
import datetime  
import numpy as np  
import pandas as pd
```

2. Let's create two **datetime** variables with 20 values each, in which values start from **2019-05-03** and increase in intervals of 1 hour and 1 month, respectively. Then, we must capture the variables in a DataFrame, add column names, and display the top rows:

```
rng_hr = pd.date_range("2019-03-05", periods=20,  
freq="H")  
  
rng_month = pd.date_range("2019-03-05", periods=20,  
freq="M")  
  
df = pd.DataFrame({"date1": rng_hr, "date2":  
rng_month})  
  
df.head()
```

We can see the first five rows of the created variables in the following output:

	date1	date2
0	2019-03-05 00:00:00	2019-03-31
1	2019-03-05 01:00:00	2019-04-30
2	2019-03-05 02:00:00	2019-05-31
3	2019-03-05 03:00:00	2019-06-30
4	2019-03-05 04:00:00	2019-07-31

Figure 6.13 – First five rows of the toy DataFrame

3. Let's capture the difference in days between the two variables in a new feature, and then display the DataFrame's top rows:

```
df["elapsed_days"] = (df["date2"] -  
df["date1"]).dt.days  
  
df.head()
```

We can see the difference in days in the following output:

	date1	date2	elapsed_days
0	2019-03-05 00:00:00	2019-03-31	26
1	2019-03-05 01:00:00	2019-04-30	55
2	2019-03-05 02:00:00	2019-05-31	86
3	2019-03-05 03:00:00	2019-06-30	116
4	2019-03-05 04:00:00	2019-07-31	147

Figure 6.14 – First five rows of the toy DataFrame with the new variable

4. Let's capture the difference in months between the two **datetime** variables in a new feature and then display the DataFrame's top rows:

```
df["months_passed"] = (
    (df["date2"] - df["date1"]) / np.timedelta64(1,
    "M"))

df["months_passed"] = np.round(df["months_passed"], 0)

df.head()
```

We can see the difference in months between the variables in the following screenshot:

	date1	date2	elapsed_days	months_passed
0	2019-03-05 00:00:00	2019-03-31	26	1.0
1	2019-03-05 01:00:00	2019-04-30	55	2.0
2	2019-03-05 02:00:00	2019-05-31	86	3.0
3	2019-03-05 03:00:00	2019-06-30	116	4.0
4	2019-03-05 04:00:00	2019-07-31	147	5.0

Figure 6.15 – DataFrame with the time difference in different units

5. Now, let's calculate the time in between the variables in minutes and seconds and then display the DataFrame's top rows:

```
df["diff_seconds"] = (
    df["date2"] - df["date1"])/np.timedelta64(1, "s")
df["diff_minutes"] = (
    df["date2"] - df["date1"])/ np.timedelta64(1, "m")
df.head()
```

We can see the new variables in the following output:

	date1	date2	elapsed_days	months_passed	diff_seconds	diff_minutes
0	2019-03-05 00:00:00	2019-03-31	26	1.0	2246400.0	37440.0
1	2019-03-05 01:00:00	2019-04-30	55	2.0	4834800.0	80580.0
2	2019-03-05 02:00:00	2019-05-31	86	3.0	7509600.0	125160.0
3	2019-03-05 03:00:00	2019-06-30	116	4.0	10098000.0	168300.0
4	2019-03-05 04:00:00	2019-07-31	147	5.0	12772800.0	212880.0

Figure 6.16 – DataFrame with the time difference in different units

6. Finally, let's calculate the difference between one variable and the current day, and then display the first five rows of the DataFrame:

```
df["to_today"] = (
    datetime.datetime.today() - df["date1"])
df.head()
```

We can see the new variable in the final column of the DataFrame in the following screenshot:

	date1	date2	elapsed_days	months_passed	diff_seconds	diff_minutes	to_today
0	2019-03-05 00:00:00	2019-03-31	26	1.0	2246400.0	37440.0	234 days 16:46:28.095694
1	2019-03-05 01:00:00	2019-04-30	55	2.0	4834800.0	80580.0	234 days 15:46:28.095694
2	2019-03-05 02:00:00	2019-05-31	86	3.0	7509600.0	125160.0	234 days 14:46:28.095694
3	2019-03-05 03:00:00	2019-06-30	116	4.0	10098000.0	168300.0	234 days 13:46:28.095694
4	2019-03-05 04:00:00	2019-07-31	147	5.0	12772800.0	212880.0	234 days 12:46:28.095694

Figure 6.17 – DataFrame with the new variables

NOTE

The **to_today** variable on your computer will be different from the one in this book, due to the difference between the current date (at the time of writing) and when you execute the code.

That's it! We've now enriched our dataset with new features that were created by comparing two **datetime** variables.

How it works...

In this recipe, we captured different representations of the time between two **datetime** variables. To proceed with this recipe, we created a toy DataFrame with two variables, each with 20 dates starting at an arbitrary date. The first variable increased in intervals of 1 hour, while the second variable increased in intervals of 1 month. We created the variables with

pandas `date_range()`, which we covered extensively in the previous recipes in this chapter.

To determine the difference between the variables – that is, to determine the time between them – we directly subtracted one `datetime` variable from the other – that is, one pandas Series from the other. The difference between the two pandas Series returns a new pandas Series. To capture the difference in days, we used pandas `dt`, followed by `days`. To convert the time difference into months, we used the `timedelta()` method from NumPy and indicated we wanted the difference in months, passing `M` in the second argument of the method. To capture the difference in seconds and minutes, we passed the `s` and `m` strings to `timedelta()`, respectively.

NOTE

NumPy's `timedelta()` method complements pandas `datetime`. The arguments for NumPy's `timedelta` method are a number – `1`, in our example – to represent the number of units, and a `datetime` unit, such as `(D)ay`, `(M)onth`, `(Y)ear`, `(h)ours`, `(m)inutes`, or `(s)econds`.

Finally, we captured the difference from one `datetime` variable to today's date. We obtained the date and time of today using the built-in Python library, `datetime`, while using the `datetime.today()` method. We subtracted one of the `datetime` variables from our DataFrame for today's date and captured the difference in days, hours, minutes, seconds, and nanoseconds, which is the default value of the operation.

See also

To learn more about NumPy's **timedelta**, visit
<https://numpy.org/devdocs/reference/arrays.datetime.xhtml#datetime-and-timedelta-arithmetic>.

Working with time in different time zones

Some organizations operate internationally; therefore, the information they collect about events may be recorded alongside the time zone of the area where the event took place. To be able to compare events that occurred across different time zones, we need to set all of the variables within the same zone. In this recipe, we will learn how to unify the time zones of a **datetime** variable and then learn how to reassign a variable to a different time zone using pandas.

How to do it...

To proceed with this recipe, we must import pandas and then create a toy DataFrame with two variables, each one containing a date and time in different time zones:

1. Import **pandas**:

```
import pandas as pd
```

2. Let's create a toy DataFrame with one variable with values in different time zones:

```
df = pd.DataFrame()  
df['time1'] = pd.concat([  
    pd.Series(  
        [
```

```
pd.date_range(  
    start='2015-06-10 09:00', freq='H',  
    periods=3,  
    tz='Europe/Berlin')),  
  
pd.Series(  
    pd.date_range(  
        start='2015-09-10 09:00', freq='H',  
        periods=3,  
        tz='US/Central'))  
, axis=0)
```

3. Now, let's add another **datetime** variable to the DataFrame, which also contains values in different time zones, and then display the resulting DataFrame:

```
df['time2'] = pd.concat([  
  
    pd.Series(  
        pd.date_range(  
            start='2015-07-01 09:00', freq='H',  
            periods=3,  
            tz='Europe/Berlin')),  
  
    pd.Series(  
        pd.date_range(  
            start='2015-08-01 09:00', freq='H',  
            periods=3,
```

```
tz='US/Central'))  
], axis=0)  
  
df
```

We can see the toy DataFrame with the variables in the different time zones in the following screenshot:

	time1	time2
0	2015-06-10 09:00:00+02:00	2015-07-01 09:00:00+02:00
1	2015-06-10 10:00:00+02:00	2015-07-01 10:00:00+02:00
2	2015-06-10 11:00:00+02:00	2015-07-01 11:00:00+02:00
0	2015-09-10 09:00:00-05:00	2015-08-01 09:00:00-05:00
1	2015-09-10 10:00:00-05:00	2015-08-01 10:00:00-05:00
2	2015-09-10 11:00:00-05:00	2015-08-01 11:00:00-05:00

Figure 6.18 – Toy DataFrame with two datetime variables in different time zones

NOTE

The time zone is indicated with the **+02** and **-05** values, respectively.

4. To work with different time zones, first, we must unify the time zone to the central time zone setting, **utc = True**:

```

df['time1_utc'] = pd.to_datetime(df['time1'], utc=True)
df['time2_utc'] = pd.to_datetime(df['time2'], utc=True)

df

```

Note how, in the new variables, **utc** is zero, whereas in the previous variables, it varies:

	time1	time2	time1_utc	time2_utc
0	2015-06-10 09:00:00+02:00	2015-07-01 09:00:00+02:00	2015-06-10 07:00:00+00:00	2015-07-01 07:00:00+00:00
1	2015-06-10 10:00:00+02:00	2015-07-01 10:00:00+02:00	2015-06-10 08:00:00+00:00	2015-07-01 08:00:00+00:00
2	2015-06-10 11:00:00+02:00	2015-07-01 11:00:00+02:00	2015-06-10 09:00:00+00:00	2015-07-01 09:00:00+00:00
0	2015-09-10 09:00:00-05:00	2015-08-01 09:00:00-05:00	2015-09-10 14:00:00+00:00	2015-08-01 14:00:00+00:00
1	2015-09-10 10:00:00-05:00	2015-08-01 10:00:00-05:00	2015-09-10 15:00:00+00:00	2015-08-01 15:00:00+00:00
2	2015-09-10 11:00:00-05:00	2015-08-01 11:00:00-05:00	2015-09-10 16:00:00+00:00	2015-08-01 16:00:00+00:00

Figure 6.19 – The variables have now been unified to the central time zone (UTC)

5. Now, let's calculate the difference in days between the variables and then display the DataFrame:

```
df['elapsed_days'] = (
```

```
df['time2_utc'] - df['time1_utc']).dt.days  
df['elapsed_days'].head()
```

We can see the time between the values of the variables in the DataFrame in the following output:

```
0    21  
1    21  
2    21  
0   -40  
1   -40  
  
Name: elapsed_days, dtype: int64
```

6. Finally, let's change the time zone of the **datetime** variables to alternative ones and display the new variables:

```
df['time1_london'] = df[  
    'time1_utc'].dt.tz_convert('Europe/London')  
df['time2_berlin'] = df[  
    'time1_utc'].dt.tz_convert('Europe/Berlin')  
df[['time1_london', 'time2_berlin']]
```

We can see the variables in their respective time zones in the following screenshot:

	time1_london	time2_berlin
0	2015-06-10 08:00:00+01:00	2015-06-10 09:00:00+02:00
1	2015-06-10 09:00:00+01:00	2015-06-10 10:00:00+02:00
2	2015-06-10 10:00:00+01:00	2015-06-10 11:00:00+02:00
0	2015-09-10 15:00:00+01:00	2015-09-10 16:00:00+02:00
1	2015-09-10 16:00:00+01:00	2015-09-10 17:00:00+02:00
2	2015-09-10 17:00:00+01:00	2015-09-10 18:00:00+02:00

Figure 6.20 – The variables reformatted into different time zones

Note how, when changing time zones, not only do the values of the zone change – that is, the **+01** and **+02** values in the preceding screenshot – but the value of the hour changes as well.

How it works...

In this recipe, we changed the time zone of the variables and performed operations with variables in different time zones. To begin, we created a DataFrame with two variables, the values of which started at an arbitrary date and increased hourly; these were set in different time zones. To combine the different time zone variables in one column within the

DataFrame, we concatenated the series returned by pandas `date_range()` by utilizing the pandas `concat()` method. We set the `axis` argument to `0`, to indicate we wanted to concatenate the series vertically in one column. We covered the arguments of pandas `date_range()` extensively in former recipes in this chapter; see, for example, the *Extracting features from dates with pandas* and *Extracting features from time with pandas* recipes for more details.

To reset the time zone of the variables to the central zone, we used the pandas `to_datetime()` method and passed `utc=True`. Finally, we determined the time difference between the variables by subtracting one series from the other and capturing the difference in days. To reassign a different time zone, we used the pandas `tz_convert()` method, indicating the new time zone as an argument.

See also

To learn more about the `_datetime()` method, visit
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.to_datetime.xhtml.

To learn more about the pandas `tz_convert()` method, visit
https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.dt.tz_convert.xhtml.

Automating feature extraction with Feature-engine

Feature-engine is a Python library dedicated to engineering and selecting features and is well-suited to working with pandas DataFrames. The **DatetimeFeatures()** class from Feature-engine can extract features from date and time automatically by using the pandas **dt** module under the hood. This transformer allows you to extract the following features:

- Month
- Quarter
- Semester
- Year
- Week
- Day of week
- Day of month
- Day of year
- Weekend
- Month start
- Month end
- Quarter start
- Quarter end
- Year start
- Year end
- Leap year

- Days in month
- Hour
- Minute
- Second

In this recipe, we will automatically create features from date and time by utilizing Feature-engine.

How to do it...

To proceed with this recipe, we must import the necessary libraries and create a toy dataset:

1. Let's begin by importing **pandas** and **DatetimeFeatures()**:

```
import pandas as pd
from feature_engine.datetime import DatetimeFeatures
```

2. Let's create 20 **datetime** values, with values beginning from **2019-03-05** at midnight followed by increments of 1 day. Then, we must capture the value range in a DataFrame:

```
rng_ = pd.date_range('2019-03-05', periods=20,
freq='D')
data = pd.DataFrame({'date': rng_})
```

3. Let's set up the transformer to create the default date and time features automatically:

```
dtfs = DatetimeFeatures()
```

```
    variables=None,  
    features_to_extract=None,  
)  
)
```

NOTE

DatetimeFeatures() automatically finds all the variables of the **datetime** type, or that could be parsed as **datetime** when the **parameter** variable is set to **None**. Alternatively, you can pass a variable list, with the names of the variables for which you want to create the date and time features.

4. Let's add the date and time features to the data:

```
dft = dtfs.fit_transform(data)
```

5. Let's capture the names of the new variables in a list:

```
vars_ = [v for v in dft.columns if "date" in v]
```

NOTE

DatetimeFeatures() names the new features with the original variable name plus the type of feature created, for example, **date_day_of_week**.

If we execute **dft[vars_].head()**, we will see the new variables, as shown in the following output:

	<code>date_month</code>	<code>date_year</code>	<code>date_day_of_week</code>	<code>date_day_of_month</code>	<code>date_hour</code>	<code>date_minute</code>	<code>date_second</code>
0	3	2019	1	5	0	0	0
1	3	2019	2	6	0	0	0
2	3	2019	3	7	0	0	0
3	3	2019	4	8	0	0	0
4	3	2019	5	9	0	0	0

Figure 6.21 – DataFrame returned by `DatetimeFeatures()` with the new variables

NOTE

We can create specific features by passing their names to `features_to_extract`, for example, `dts = DatetimeFeatures(features_to_extract=["week", "year"])`. We can also extract all possible features by setting that parameter to `"all"`. If `features_to_extract=None`, Feature-engine will return default features, predefined in the transformer.

`DatetimeFeatures()` can also create features from variable values in different time zones. Let's learn how to correctly set up the transformer in this situation.

6. Let's create a toy DataFrame with one variable with values in different time zones:

```
df = pd.DataFrame()
df["time"] = pd.concat([
    [
```

```
pd.Series(  
    pd.date_range(  
        start="2014-08-01 09:00", freq="H",  
        periods=3, tz="Europe/Berlin"  
    ),  
    pd.Series(  
        pd.date_range(  
            start="2014-08-01 09:00", freq="H",  
            periods=3, tz="US/Central"  
        ),  
    ],  
    axis=0,  
)
```

If we execute **df**, we will see the toy DataFrame, as shown in the following screenshot:

	time
0	2014-08-01 09:00:00+02:00
1	2014-08-01 10:00:00+02:00
2	2014-08-01 11:00:00+02:00
0	2014-08-01 09:00:00-05:00
1	2014-08-01 10:00:00-05:00
2	2014-08-01 11:00:00-05:00

Figure 6.22 – DataFrame with a variable with values in different time zones

7. Now, let's set up **DatetimeFeatures()** so that it extracts three specific features, indicating **utc=True** so that it transforms the variable into the central time zone:

```
dfts = DatetimeFeatures(
    features_to_extract=["day_of_week", "hour",
    "minute"],
    drop_original=False,
    utc=True,
)
```

8. Now, we can create the new features:

```
dft = dfts.fit_transform(df)
```

DatetimeFeatures() will put all timestamps in UTC before deriving the features. With **dft.head()**, we can see the resulting DataFrame:

	time	time_day_of_week	time_hour	time_minute
0	2014-08-01 09:00:00+02:00	4	7	0
1	2014-08-01 10:00:00+02:00	4	8	0
2	2014-08-01 11:00:00+02:00	4	9	0
0	2014-08-01 09:00:00-05:00	4	14	0
1	2014-08-01 10:00:00-05:00	4	15	0

Figure 6.23 – DataFrame with the original and new variables

With that, we've created multiple date and time-related features in a few lines of code. Feature-engine offers a great alternative to manually creating features per feature with pandas.

How it works...

DatetimeFeatures() extracts several date and time features from **datetime** variables automatically by utilizing pandas **dt** under the hood. It works with variables whose original data types are **datetime**, as well as with object-like and categorical variables, provided that they can be parsed into **datetime** format. **DatetimeFeatures()** can extract all features when we set the **features_to_extract** parameter to "**all**" or a specific subset predefined by the class when we leave the parameter set to **None**. Alternatively, we can pass the names of the features to extract into a list.

DatetimeFeatures() automatically finds the **datetime** variables, although we can define them in a list and pass them to the parameter **variables**.

With the **fit()** method, the transformer does not learn any parameters; instead, it checks that the variables that are entered by the user are or can be parsed into **datetime** format. If the user does not indicate the variables, it finds the **datetime** variables automatically. With **transform()**, **DatetimeFeatures()** adds the date and time-derived variables to the DataFrame.

7

Performing Feature Scaling

Many machine learning algorithms are sensitive to the scale of the features. In particular, the coefficients of linear models depend on the scale of the feature; that is, changing the feature scale will change the coefficient's value. In linear models, as well as and algorithms that depend on distance calculations, such as clustering and principal component analysis, features with bigger value ranges tend to dominate over features with smaller ranges. Therefore, having features within a similar scale allows us to compare feature importance and also helps algorithms converge faster, thus improving performance and training times.

Scaling techniques will divide the variables by some constant; therefore, it is important to highlight that no matter the scaling method, the shape of the variable distribution does not change. If what you want is to change the distribution shape, check out [Chapter 3, Transforming Numerical Variables](#).

In this chapter, we will describe different methods you can use to put the features on the same scale, highlighting the characteristics of each method and the variable distributions for which they are better suited.

This chapter will cover the following recipes:

- Standardizing the features
- Scaling to the maximum and minimum values
- Scaling with the median and quantiles

- Performing mean normalization
- Implementing maximum absolute scaling
- Scaling to vector unit length

Technical requirements

The main libraries that we use in this chapter are scikit-learn for scaling and pandas to handle the data and make a few plots.

Standardizing the features

Standardization is the process of centering the variable at 0 and standardizing the variance to 1. To standardize features, we subtract the mean from each observation and then divide the result by the standard deviation:

$$x_{scaled} = \frac{x - mean(x)}{std(x)}$$

The result of the preceding transformation is called the z-score and represents how many standard deviations a given observation *deviates* from the mean. In this recipe, we will implement standardization with scikit-learn.

How to do it...

To begin, we will import the required packages, load the dataset, and prepare the train and test sets:

1. Import the required Python packages, classes, and functions:

```
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

2. Let's load the California housing dataset from scikit-learn into a dataframe, and drop the **Latitude** and **Longitude** variables as we would not use them as such in predictive models:

```
X, y = fetch_california_housing(
    return_X_y=True, as_frame=True)
X.drop(labels=["Latitude", "Longitude"], axis=1,
inplace=True)
```

3. Now, let's divide the data into train and test sets:

```
X_train, X_test, y_train, y_test = train_test_split(X,
y, test_size=0.3, random_state=0)
```

4. Next, we'll set up a standard scaler using **StandardScaler()** from scikit-learn and fit it to the train set so that it learns each variable's mean and standard deviation:

```
scaler = StandardScaler() scaler.fit(X_train)
```

5. Now, let's standardize the train and test sets with the trained scaler:

```
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

StandardScaler() stores the mean and standard deviation needed to standardize the variables. Let's visualize the learned parameters.

6. First, we'll print the mean values that were learned by the **scaler**:

```
scaler.mean_
```

The mean values per variable can be seen in the following output:

```
array([3.86666741e+00, 2.86187016e+01, 5.42340368e+00,  
1.09477484e+00, 1.42515732e+03, 3.04051776e+00])
```

7. Now, let's print the standard deviation values that were learned by the **scaler**:

```
scaler.scale_
```

The standard deviation of each variable can be seen in the following output:

```
array([1.89109236e+00, 1.25962585e+01, 2.28754018e+00,  
4.52736275e-01, 1.14954037e+03, 6.86792905e+00])
```

Scikit-learn scalers, just like any scikit-learn transformer, return NumPy arrays.

8. Let's convert the arrays into dataframes:

```
X_train_scaled = pd.DataFrame(  
    X_train_scaled, columns=X_train.columns)  
X_test_scaled = pd.DataFrame(  
    X_test_scaled, columns=X_test.columns)
```

Now we can compare the transformed data with the original data to visualize the changes.

9. Let's print the main statistics from the original variables:

```
X_test.describe()
```

In the output of the preceding command, we can see that the data is not centered at 0 and the variance varies among the variables:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup
count	6192.000000	6192.000000	6192.000000	6192.000000	6192.000000	6192.000000
mean	3.880013	28.687984	5.442057	1.101109	1426.222061	3.140976
std	1.920007	12.560416	2.862733	0.519956	1091.567168	15.796292
min	0.499900	1.000000	1.465753	0.500000	8.000000	0.692308
25%	2.552150	18.000000	4.414452	1.006494	796.000000	2.436452
50%	3.529600	29.000000	5.227365	1.048741	1169.500000	2.825041
75%	4.768750	37.000000	6.064257	1.098434	1727.250000	3.285501
max	15.000100	52.000000	141.909091	25.636364	16305.000000	1243.333333

Figure 7.1 – Statistical parameters of the original variables

10. Let's now print the main statistics from the transformed variables:

```
x_test_scaled.describe()
```

In the output of the preceding command, we can see that the data is now centered at 0 and the variance is 1:

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup
count	6192.000000	6192.000000	6192.000000	6192.000000	6192.000000	6192.000000
mean	0.007057	0.005500	0.008154	0.013991	0.000926	0.014627
std	1.015290	0.997154	1.251446	1.148474	0.949568	2.300008
min	-1.780329	-2.192612	-1.730090	-1.313734	-1.232803	-0.341909
25%	-0.695110	-0.843004	-0.441064	-0.194995	-0.547312	-0.087955
50%	-0.178240	0.030271	-0.085698	-0.101679	-0.222400	-0.031374
75%	0.477017	0.665380	0.280150	0.008082	0.262794	0.035671
max	5.887302	1.856210	59.664826	54.207251	12.944167	180.591967

Figure 7.2 – Statistical parameters of the scaled variables

11. Finally, let's plot a histogram of the original distributions:

```
x_test.hist(bins=20, figsize=(20, 20))
plt.show()
```

In the following output, we see the distribution of each variable:

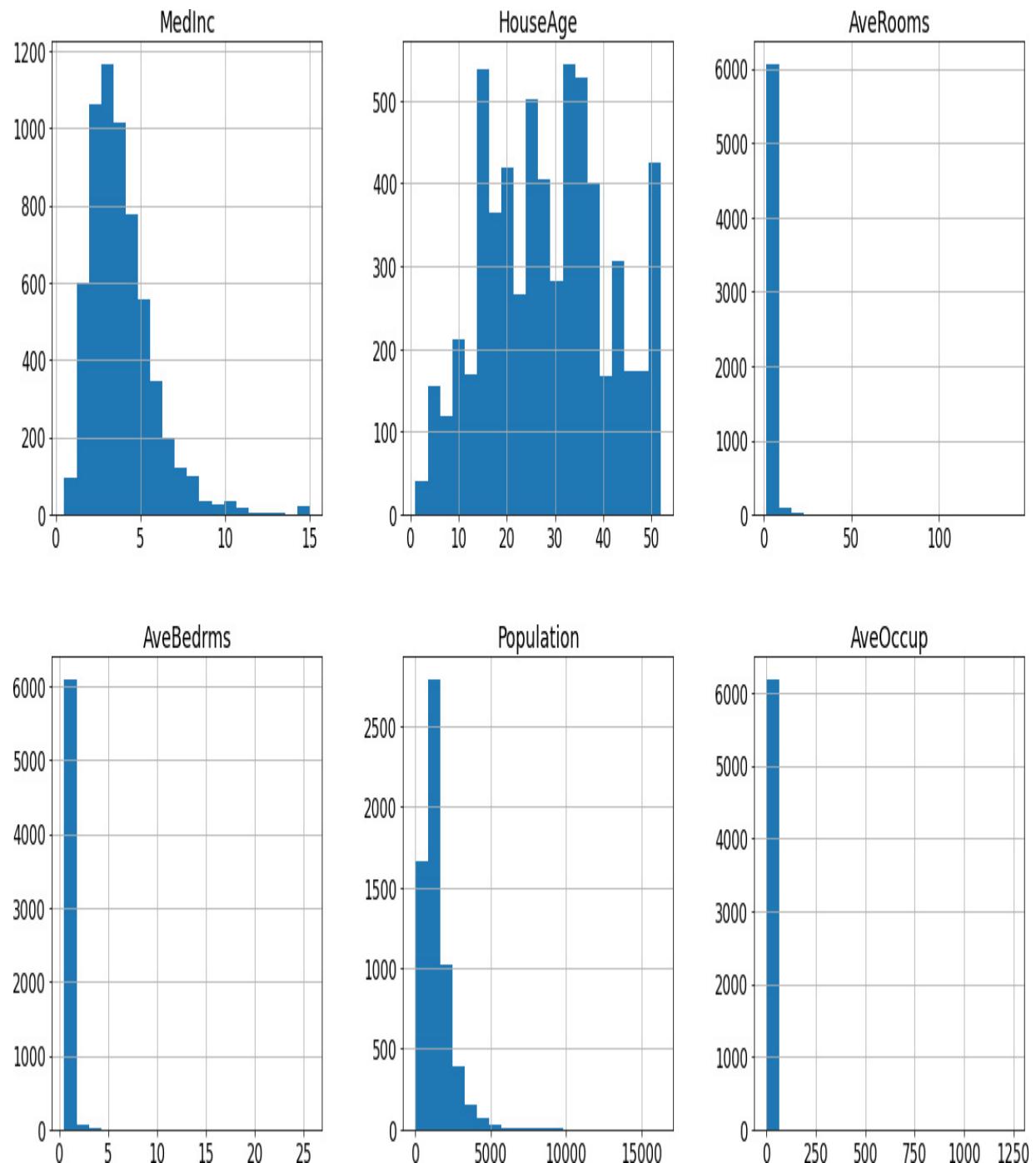


Figure 7.3 – Histograms of the original variables

12. And now, let's plot a histogram of the scaled variables:

```
x_test_scaled.hist(bins=20, figsize=(20, 20))
plt.show()
```

In the following output, we see that after the scaling, the shape of the variable's distribution remains the same:

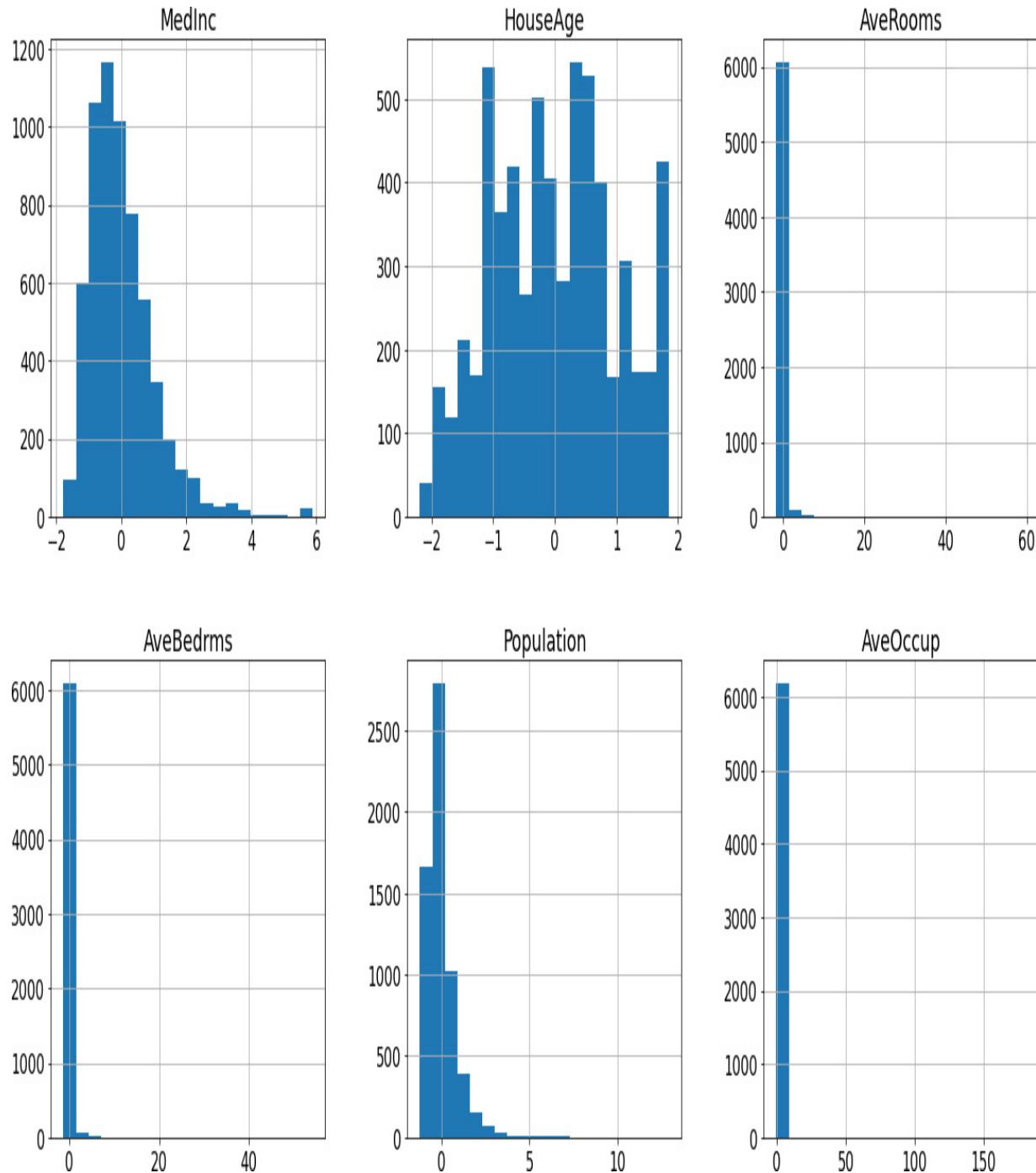


Figure 7.4 – Histograms of the scaled variables

We now have the variables of our data on a similar scale.

How it works...

In this recipe, we standardized the variables of the California housing dataset by utilizing scikit-learn. We split the data into train and test because the parameters for the standardization should be learned from the train set.

To standardize these features, we used **StandardScaler()**. Using the **fit()** method, the scaler learned each variable's mean and standard deviation and stored them in its **mean_** and **scale_** attributes. Using the **transform()** method, the scaler standardized the variables in the train and test sets, returning NumPy arrays.

Scaling to the maximum and minimum values

Scaling to the minimum and maximum values squeezes the values of the variables between 0 and 1. To implement this scaling technique, we subtract the minimum value from all the observations and divide the result by the value range, that is, the difference between the maximum and minimum values:

$$x_{scaled} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

In this recipe, we will implement scaling to the minimum and maximum values by utilizing scikit-learn.

How to do it...

To begin, we will import the required packages, load the dataset, and prepare the train and test sets:

1. Import pandas and the required scikit-learn classes and functions:

```
import pandas as pd
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
```

2. Let's load the California housing dataset from scikit-learn into a pandas dataframe:

```
X, y = fetch_california_housing(
    return_X_y=True, as_frame=True
)
X.drop(labels=["Latitude", "Longitude"], axis=1,
inplace=True)
```

3. Let's divide the data into train and test sets:

```
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.3,
    random_state=0,
)
```

4. Let's set up the scaler and then fit it to the train set so that it learns each variable's minimum and maximum values:

```
scaler = MinMaxScaler()
scaler.fit(X_train)
```

5. Finally, let's scale the variables in the train and test sets with the trained scaler:

```
X_train_scaled = scaler.transform(X_train)  
X_test_scaled = scaler.transform(X_test)
```

TIP

MinMaxScaler() stores the maximum and minimum values and the value ranges in its **data_max_**, **min_**, and **data_range_** attributes, respectively.

We can now transform these NumPy arrays into pandas dataframes and compare the variable distributions and their values before and after the transformation, as we did in *steps 8 to 12* of the *Standardizing the features* recipe.

How it works...

In this recipe, we scaled the numerical variables of the California housing dataset that comes with scikit-learn to their minimum and maximum values.

The **MinMaxScaler()** from scikit-learn learned the minimum and maximum values of each variable when we called the **fit()** method and stored these parameters in its **data_max_**, **min_**, and **data_range** attributes. With the **transform()** method, the scaler removed the minimum value from each variable in the train and test sets and divided the result by the value range, returning NumPy arrays.

After the transformation, the minimum and maximum values of each variable were 0 and 1, respectively.

TIP

If you want to scale only a subset of variables, you can use either `ColumnTransformer()` from scikit-learn or `SklearnTransformerWrapper()` from Feature-engine together with `MinMaxScaler()`.

Scaling with the median and quantiles

When scaling variables to the median and quantiles, the median value is removed from the observations, and the result is divided by the **Inter-Quartile Range (IQR)**. The IQR is the difference between the 1st quartile and the 3rd quartile, or, in other words, the difference between the 25th quantile and the 75th quantile:

$$x_{scaled} = \frac{x - \text{median}(x)}{\text{75th quantile}(x) - \text{25th quantile}(x)}$$

This method is known as *robust scaling* because it produces more robust estimates for the center and value range of the variable and is recommended if the data contains outliers. In this recipe, we will implement scaling with the median and IQR by utilizing scikit-learn.

How to do it...

To begin, we will import the required packages, load the dataset, and prepare the train and test sets:

1. Import pandas and the required scikit-learn classes and functions:

```
import pandas as pd
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import RobustScaler
```

2. Let's load the California housing dataset from scikit-learn into a pandas dataframe:

```
X, y = fetch_california_housing(
    return_X_y=True, as_frame=True)
X.drop(labels=["Latitude", "Longitude"], axis=1,
inplace=True)
```

3. Let's divide the data into train and test sets:

```
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.3,
    random_state=0,
)
```

4. To perform scaling to the median and quantiles, we use

RobustScaler() from scikit-learn and fit it to the train set so that it learns and stores the median and IQR:

```
scaler = RobustScaler()
scaler.fit(X_train)
```

5. Finally, let's scale the variables in the train and test sets with the trained scaler:

```
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

6. We can go ahead and print the variable median values stored by **RobustScaler()**:

```
scaler.center_
```

7. The medians that are stored in the **center_** attribute of **RobustScaler()** can be seen in the following output:

```
array([3.53910000e+00, 2.90000000e+01, 5.22931763e+00,  
1.04878049e+00, 1.16500000e+03, 2.81635506e+00])
```

8. Now, let's output the IQR stored in **RobustScaler()**:

```
scaler.scale_
```

We can see the IQR for each variable in the following output:

```
array([2.16550000e+00, 1.90000000e+01, 1.59537022e+00,  
9.41284380e-02, 9.40000000e+02, 8.53176853e-01])
```

We can now transform these NumPy arrays into pandas dataframes and compare the variable distributions and their values before and after the transformation, as we did in *steps 8 to 12* of the *Standardizing the features* recipe.

How it works...

In this recipe, we scaled the numerical variables of the California housing dataset from scikit-learn to the median and IQR. First, we divided it into train and test sets.

To scale the features, we created an instance of **RobustScaler()**. With the **fit()** method, the scaler learned the median and IQR for each variable from the train set. With the **transform()** method, the scaler subtracted the median from each variable in the train and test sets and divided the result by the IQR, returning NumPy arrays with the scaled variables.

After the transformation, the median values of the variables are centered at 0, but the overall shape of the distributions did not change.

You can check the distributions in our repository at

<https://github.com/PacktPublishing/Python-Feature-Engineering-Cookbook-Second-Edition/blob/main/ch07-scaling/Recipe-3-robust-scaling.ipynb>.

Performing mean normalization

In mean normalization, we center the variable at 0 and rescale the distribution to the value range. This procedure involves subtracting the mean from each observation and then dividing the result by the difference between the minimum and maximum values:

$$x_{scaled} = \frac{x - mean(x)}{\max(x) - \min(x)}$$

In this recipe, we will implement mean normalization with pandas and then with scikit-learn.

How to do it...

We'll begin by importing the required libraries, loading the dataset, and preparing the train and test sets:

1. Import pandas and the required scikit-learn class and function:

```
import pandas as pd
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
```

2. Let's load the California housing dataset from scikit-learn into a pandas dataframe:

```
X, y = fetch_california_housing(
    return_X_y=True, as_frame=True)
X.drop(labels=["Latitude", "Longitude"], axis=1,
inplace=True)
```

3. Let's divide the data into train and test sets:

```
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.3,
    random_state=0,
)
```

4. Let's learn the mean values from the variables in the train set using the following:

```
means = X_train.mean(axis=0)
```

NOTE

We set the axis to 0 to indicate we want the mean across all the rows, that is, across all the observations, which is the mean of each variable. If we set *axis* to 1 instead, pandas will calculate the mean value per observation across all the columns.

5. By executing **print(means)**, we can see the mean values per variable:

MedInc	3.866667
HouseAge	28.618702
AveRooms	5.423404
AveBedrms	1.094775

```
Population      1425.157323
AveOccup        3.040518
dtype: float64
```

- Now, let's capture the difference between the maximum and minimum values per variable:

```
ranges = X_train.max(axis=0)-X_train.min(axis=0)
```

- By executing **print(ranges)**, we can see the value ranges per variable:

```
MedInc          14.500200
HouseAge         51.000000
AveRooms        131.687179
AveBedrms       33.733333
Population      35679.000000
AveOccup        598.964286
dtype: float64
```

NOTE

The pandas **mean()**, **max()**, and **min()** methods return a pandas Series.

- Now, we'll implement the mean normalization of the train and test sets by utilizing the learned parameters:

```
X_train_scaled = (X_train - means) / ranges
X_test_scaled = (X_test - means) / ranges
```

TIP

In order to transform future data, you will need to store these parameters, for example, in a **.txt** or **.csv** file.

Note that this procedure returns pandas dataframes of the transformed train and test sets. You can go ahead and compare the variables before and after

the transformations using the commands from *steps 9 to 12* of the *Standardizing the features* recipe.

How it works...

In this recipe, we standardized the numerical variables of the California housing dataset from scikit-learn. We loaded the dataset and divided it into train and test sets using the `train_test_split()` function from scikit-learn. To implement mean normalization, we captured the mean values of the numerical variables in the train set using the pandas `mean()` method. Next, we determined the difference between the maximum and minimum values of the numerical variables in the train set by utilizing the pandas `max()` and `min()` methods. Finally, we used the pandas Series with the mean values and the value ranges to implement normalization. We subtracted the mean from each observation in our train and test sets and divided the result by the value ranges. This returned the normalized variables in a pandas dataframe.

There's more...

There is no dedicated scikit-learn transformer to implement mean normalization, but we combine the use of two transformers to do so. This is convenient if you want to implement this transformation within a pipeline.

1. To do this, we need to import pandas and load the data, just like we did in *steps 1 to 3* in the *How to do it...* section of this recipe. Then, follow these steps:
2. Import the scikit-learn transformers:

```
from sklearn.preprocessing import StandardScaler,  
RobustScaler
```

3. Let's set up **StandardScaler()** so that it learns and subtracts the mean but does not divide the result by the standard deviation:

```
scaler_mean = StandardScaler(  
    with_mean=True, with_std=False)
```

4. Now, let's set up **RobustScaler()** so that it does not remove the median from the values but divides them by the value range, that is, the difference between the maximum and minimum values:

```
scaler_minmax = RobustScaler(  
    with_centering=False,  
    with_scaling=True,  
    quantile_range=(0, 100)  
)
```

TIP

To divide by the difference between the minimum and maximum values, we need to specify $(0, 100)$ in the **quantile_range** argument of **RobustScaler()**.

5. Let's fit the scalers to the train set so that they learn and store the mean, maximum, and minimum values:

```
scaler_mean.fit(X_train)  
scaler_minmax.fit(X_train)
```

6. Finally, let's apply mean normalization to the train and test sets:

```
X_train_scaled = scaler_minmax.transform(  
    scaler_mean.transform(X_train))  
X_test_scaled = scaler_minmax.transform(
```

```
    scaler_mean.transform(X_test)
)
```

We transformed the data with **StandardScaler()** to remove the mean and then transformed the resulting NumPy array with **RobustScaler()** to divide the result by the range between the minimum and maximum values. We described the functionality of **StandardScaler()** in the *Standardizing the features* recipe of this chapter and **RobustScaler()** in the *Scaling with the median and quantiles* recipe of this chapter.

TIP

MinMaxScaler() stores the maximum and minimum values and the value ranges in its **data_max_**, **min_**, and **data_range_** attributes.

Implementing maximum absolute scaling

Maximum absolute scaling scales the data to its maximum value; that is, it divides every observation by the maximum value of the variable:

$$x_{scaled} = \frac{x}{\max(x)}$$

As a result, the maximum value of each feature will be 1.0. Note that maximum absolute scaling does not center the data. It was specifically designed for scaling sparse data. In this recipe, we will implement maximum absolute scaling with scikit-learn.

TIP

Scikit-learn recommends using this transformer on data that is centered at 0 or on sparse data.

Getting ready

Maximum absolute scaling was specifically designed to scale sparse data. Thus, we will use a bag-of-words dataset that contains sparse variables for the recipe. In this dataset, the variables are words, the observations are documents, and the values are the number of times each word appears in the document. Most entries in the data are 0.

For guidelines on how to download, prepare, and store the dataset, please check out the *Getting ready* section of the *Implementing feature binarization* recipe in [Chapter 4, Performing Variable Discretization](#).

Alternatively, you can download and run this notebook at
<https://github.com/solegalli/Python-Feature-Engineering-Cookbook-Second-Edition/blob/main/ch04-discretization/download-prepare-store-enron-data.ipynb>.

How to do it...

Let's begin by importing the required packages and loading the dataset:

1. Import the required libraries and the scaler:

```
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.preprocessing import MaxAbsScaler
```

2. Let's load the bag-of-words dataset:

```
data = pd.read_csv("bag_of_words.csv"))
```

NOTE

Although we omit this step in the recipe, remember that the maximum absolute values should be learned from a training dataset only. So, remember to split the dataset into train and test sets when carrying out your analysis.

3. Let's set up the maximum absolute scaler and fit it to the data so that it learns the variable's maximum values:

```
scaler = MaxAbsScaler()  
scaler.fit(data)
```

4. Now, let's scale the variables by utilizing the trained scaler:

```
X_train_scaled = scaler.transform(data)
```

TIP

MaxScaler() stores the maximum values in its **max_abs_** attribute.

5. Let's display the maximum values stored by the scaler:

```
scaler.max_abs_
```

In the output of the preceding command, we see the maximum number of times each word appeared in a document:

```
array([ 7.,  6.,  2.,  2., 11.,  4.,  3.,  6., 52.,  2.])
```

6. Let's convert the NumPy array into a dataframe:

```
data_scaled = pd.DataFrame(  
    data_scaled, columns=data.columns)
```

And now, let's plot the distributions of the original and scaled variables.

7. Let's make a histogram with the bag of words before carrying out the scaling:

```
data.hist(bins=20, figsize=(20, 20))  
plt.show()
```

In the following output, we see the number of times each word appears in a document:

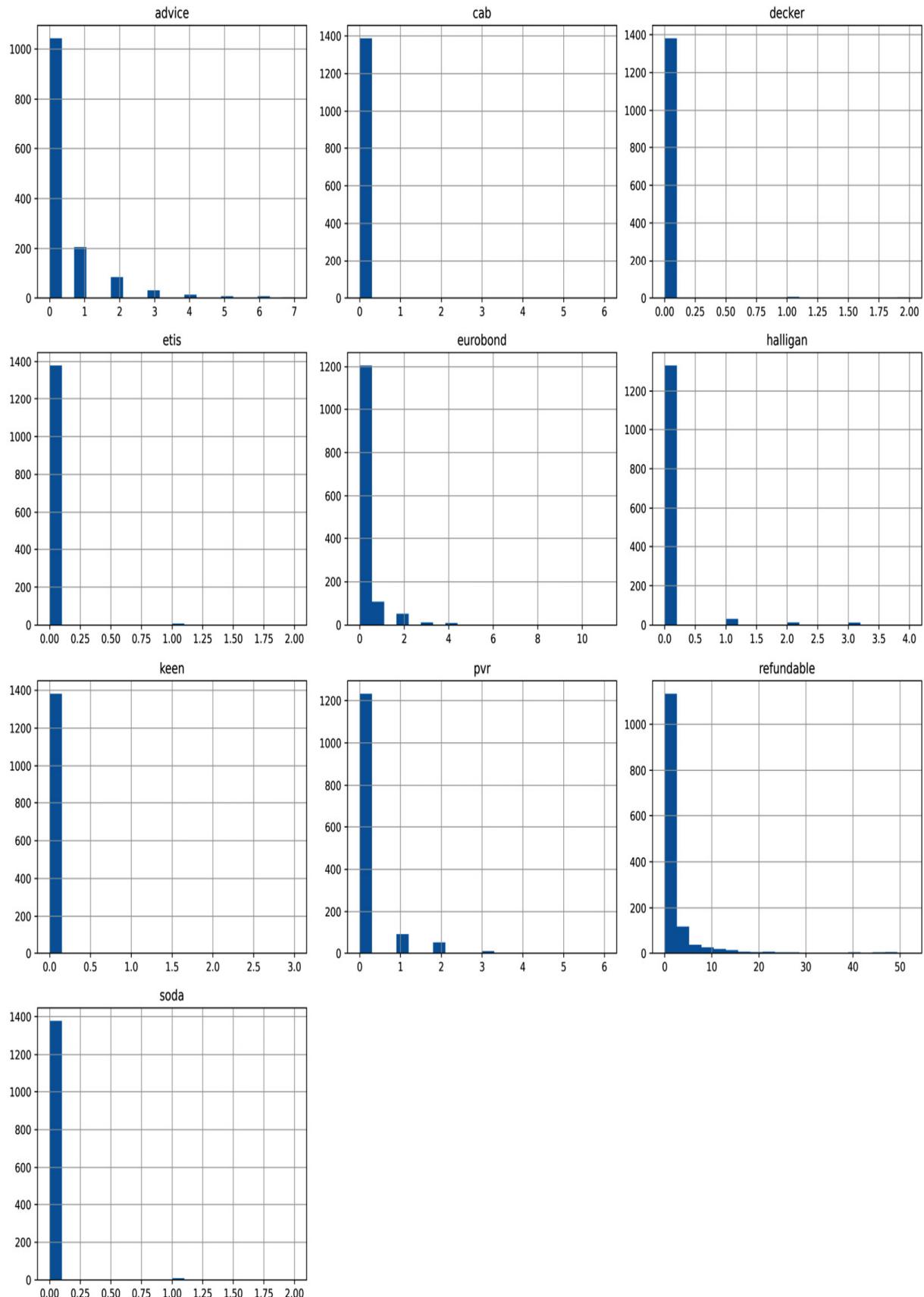


Figure 7.5 – Histograms with different word counts

8. Now, let's make a histogram with the scaled variables:

```
data_scaled.hist(bins=20, figsize=(20, 20))  
plt.show()
```

In the following output, we can corroborate the change of scale of the variables, but their distribution shape remains the same:

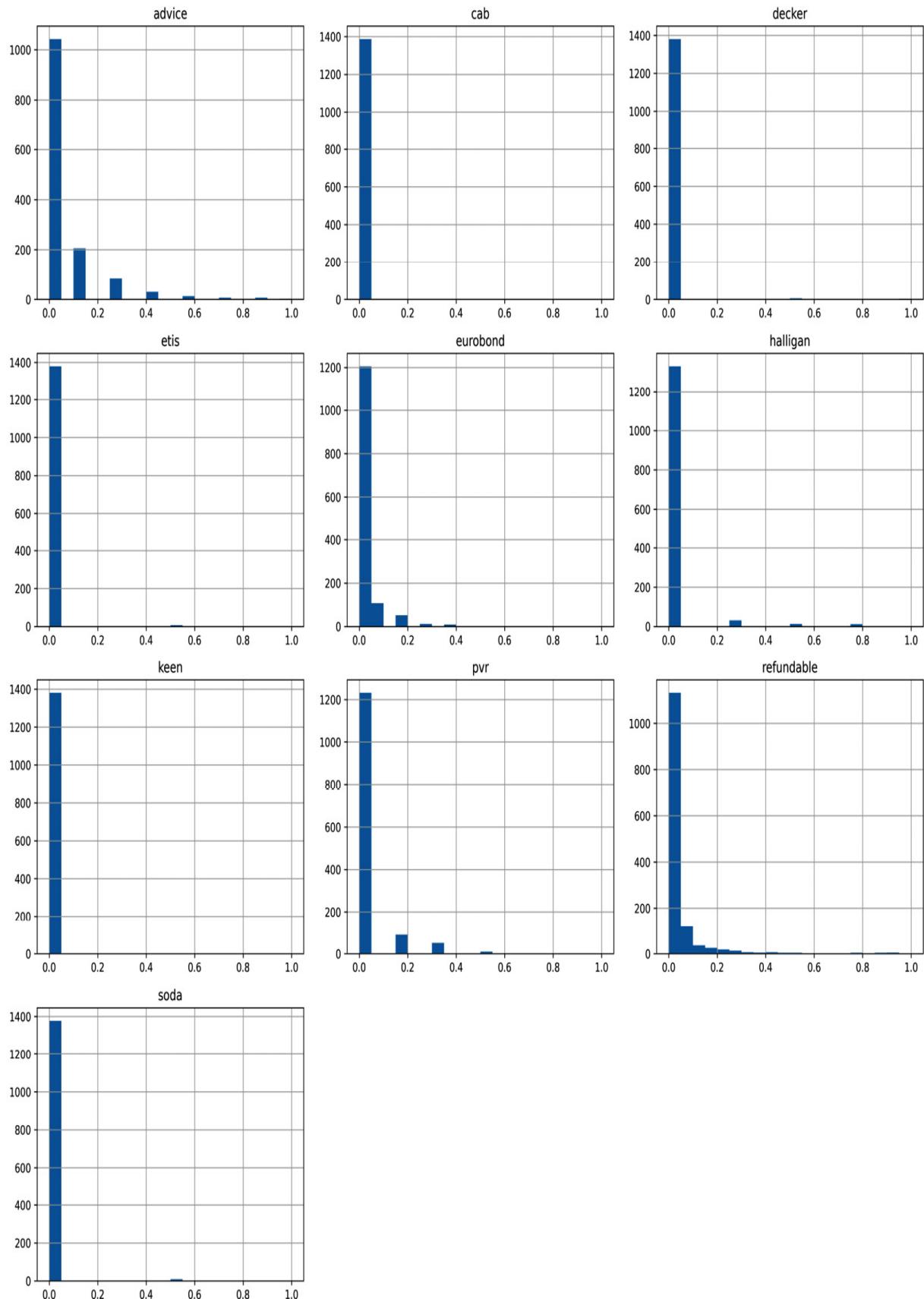


Figure 7.6 – Histograms of the word counts after the transformation

Let's move on to the next section!

How it works...

In this recipe, we scaled the sparse variables of a bag of words to their maximum values. To scale the features, we created an instance of **MaxAbsScaler()**. With the **fit()** method, the scaler learned the maximum values for each variable and stored them in its **max_abs_** attribute. With the **transform()** method, the scaler divided the variables by their maximum values, returning a NumPy array.

There's more...

If you want to center the variable distributions at **0** and then scale them to their absolute maximum, you can do so by combining the use of two scikit-learn transformers:

1. Let's import the required libraries, transformers, and functions:

```
import pandas as pd
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MaxAbsScaler,
StandardScaler
```

2. Let's load the California housing dataset and split it into train and test sets:

```
X, y = fetch_california_housing(
    return_X_y=True, as_frame=True)
```

```
x.drop(labels=["Latitude", "Longitude"], axis=1,  
inplace=True)  
X_train, X_test, y_train, y_test = train_test_split(x,  
y, test_size=0.3, random_state=0)
```

3. Let's set up **StandardScaler()** from scikit-learn so that it learns and subtracts the mean but does not divide the result by the standard deviation:

```
scaler_mean = StandardScaler(  
    with_mean=True, with_std=False)
```

4. Now, let's set up **MaxAbsScaler()** with its default parameters:

```
scaler_maxabs = MaxAbsScaler()
```

5. Let's fit the scalers to the train set so that they learn the required parameters:

```
scaler_mean.fit(X_train)  
scaler_maxabs.fit(scaler_mean.transform(X_train))
```

6. Finally, let's transform the train and test sets:

```
X_train_scaled = scaler_maxabs.transform(  
    scaler_mean.transform(X_train))  
X_test_scaled = scaler_maxabs.transform(  
    scaler_mean.transform(X_test))
```

We transformed the datasets with **StandardScaler()** to remove the mean and then transformed the returned NumPy arrays with **MaxAbsScaler()** to scale the variables to their maximum values.

Scaling to vector unit length

When scaling to vector unit length, we scale individual samples or observations so that the transformed vector has a length of 1, or in other

words, a norm of 1. Note that this scaling technique scales each individual observation and not each individual variable. To be clear, in all the scaling methods that we discussed so far in the chapter, the algorithms learned some parameters from each variable and then used those parameters to shift or rescale the distribution of the variables. On the contrary, when we scale to the unit length, we seek to normalize each observation individually, contemplating their values across all features.

Scaling to the unit norm is achieved by dividing each observation vector by either the Manhattan distance (l_1 norm) or the Euclidean distance (l_2 norm) of the vector. The Manhattan distance is given by the sum of the absolute components of the vector:

$$l_1(x) = |x_1| + |x_2| + \dots + |x_n|$$

The Euclidean distance is given by the square root of the square sum of the component of the vector:

$$l_2(x) = \sqrt{x_1^2 + x_2^2 + \dots + x_n^2}$$

Here, x_1 , x_2 , and x_n are the values of variables 1, 2, and n for each observation.

Scaling to the unit norm can be used when utilizing kernels to quantify similarity for text classification and clustering. In this recipe, we will implement scaling to vector unit length using scikit-learn.

How to do it...

To begin, we'll import the required packages, load the dataset, and prepare the train and test sets:

1. Import the required Python packages, classes, and functions:

```
import numpy as np import pandas as pd
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import Normalizer
```

2. Let's load the California housing dataset from scikit-learn into a pandas dataframe:

```
X, y = fetch_california_housing(
    return_X_y=True, as_frame=True)
X.drop(labels=["Latitude", "Longitude"], axis=1,
inplace=True)
```

3. Let's divide the data into train and test sets:

```
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.3,
    random_state=0,
)
```

4. Let's set up **Normalizer()** from scikit-learn to scale each observation to the Manhattan distance or **l1**:

```
scaler = Normalizer(norm='l1')
```

TIP

To normalize utilizing the Euclidean distance, you need to set the norm to **l2** using **scaler = Normalizer(norm='l2')**.

5. Let's transform the train and test sets; that is, we'll divide each observation vector by its norm:

```
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

6. We can calculate the length, that is, the Manhattan distance of each observation vector, using `linalg()` from NumPy:

```
np.round(np.linalg.norm(X_train_scaled, ord=1, axis=1), 1)
```

TIP

You need to set `ord=1` for the Manhattan distance or `ord=2` for the Euclidean distance as arguments of NumPy's `linalg()`, depending on whether you scaled the features to the `l1` or `l2` norm.

We can see the normalized observation vectors in the following output:

```
array([1., 1., 1., ..., 1., 1., 1.])
```

As expected, the feature length for each observation is **1**.

TIP

You can compare the output of step 7 with the distance of the unscaled data by executing `np.round(np.linalg.norm(X_train, ord=1, axis=1), 1)`.

How it works...

In this recipe, we scaled the numerical variables of the California housing dataset from scikit-learn to the vector unit norm by utilizing the Manhattan or Euclidean distance. To scale the features, we created an instance of

Normalizer() from scikit-learn and set the norm to **l1** for the Manhattan distance. For the Euclidean distance, we set the norm to **l2**. Then, we applied the **fit()** method, although there were no parameters to be learned, as this normalization procedure depends exclusively on the values of the features for each observation. Finally, with the **transform()** method, the scaler divided each observation by its norm. This returned a NumPy array with the scaled dataset.

8

Creating New Features

We can create valuable features by combining two or more variables. For example, in finance, the **disposable income**, which is the total income minus the **acquired debt** for any one month, might be more predictive of credit risk than just the **income**. Similarly, the **total acquired debt** of a person across financial products, such as a car loan, a mortgage, and credit cards, might be more predictive of credit risk than any debt considered individually. In these examples, we used domain knowledge of the data to craft the new variables, and the new variables were created by adding or subtracting existing features.

In some cases, a variable may not have a linear or monotonic relationship with the target, but a polynomial combination might. For example, if our variable has a quadratic relationship with the target, $y = x^2$, we can convert that into a linear relationship by squaring the original variable. We can also obtain better variable relationships with the target by transforming the variables through splines, or by using decision trees.

In this chapter, we will create new features using multiple mathematical functions that will be applied based on domain knowledge, and then automate feature creation by combining existing features with polynomial functions or decision trees or by creating new features using splines.

This chapter will cover the following recipes:

- Combining features with mathematical functions
- Comparing features to reference variables
- Performing polynomial expansion
- Combining features with decision trees
- Creating periodic features from cyclical variables
- Creating spline features

Technical requirements

In this chapter, we will use the Python libraries pandas, NumPy, Matplotlib, scikit-learn, and Feature-engine.

Combining features with mathematical functions

New features can be created by combining existing variables with mathematical and statistical functions. At the beginning of this chapter, we mentioned that we can calculate the total debt by summing up the debt across individual financial products, as follows:

$$\text{Total debt} = \text{car loan debt} + \text{credit card debt} + \text{mortgage debt}$$

We can also derive other insightful features using alternative statistical operations. For example, we can determine the maximum debt of a customer across financial products or the average time users have spent on a web page:

maximum debt = max(car loan balance, credit card balance, mortgage balance)

average time on page = mean(time spent user 1, time spent user 2, time spent user 3)

We can, in principle, use any mathematical or statistical operation to create new features, such as the product, mean, standard deviation, or maximum or minimum values, to name a few. In this recipe, we will implement these mathematical operations using pandas and Feature-engine.

Getting ready

In this recipe, we will use the Breast Cancer dataset that comes with scikit-learn, which contains information about the morphology of cell nuclei in tissue samples, and a target indicating whether they are cancerous cells. The features are computed from digitized images of breast cells, and they describe the characteristics of the cell nuclei in those images.

To become familiar with the dataset, run the following commands in a Jupyter Notebook or Python console:

```
from sklearn.datasets import load_breast_cancer data =  
load_breast_cancer()print(data.DESCR)
```

The preceding code block should print out a description of the dataset and an interpretation of its variables.

How to do it...

In this recipe, we will create new features by combining variables using multiple mathematical operations:

1. Let's begin by loading the necessary libraries, classes, and data:

```
import pandas as pd  
  
from feature_engine.creation import MathFeatures from  
sklearn.datasets import load_breast_cancer
```

2. Let's load the Breast Cancer dataset into a pandas DataFrame:

```
data = load_breast_cancer()df = pd.DataFrame(data.data,  
columns=data.feature_names)
```

In the following code lines, we will create new features by combining variables using multiple mathematical operations.

3. Let's begin by creating a list with the subset of features that we want to combine:

```
features = [  
  
    "mean smoothness",  
  
    "mean compactness",  
  
    "mean concavity",  
  
    "mean concave points",  
  
    "mean symmetry",  
  
]
```

The features in *step 3* represent the mean characteristics of cell nuclei in the images. It might be useful to obtain the mean across all examined characteristics.

4. Let's get the mean value of the features and then display the resulting feature:

```
df["mean_features"] = df[features].mean(axis=1)  
df["mean_features"].head()
```

The following output shows the mean value of the features:

```
0    0.21702  
1    0.10033  
2    0.16034  
3    0.20654  
4    0.14326  
  
Name: mean_features, dtype: float64
```

5. Similarly, to capture the general variability of the cell nuclei, let's determine the standard deviation of the mean characteristics, and then display the resulting feature:

```
df["std_features"] = df[features].std(axis=1)  
df["std_features"].head()
```

The following output shows the standard deviation of features:

```
0    0.080321  
1    0.045671  
2    0.042333  
3    0.078097  
4    0.044402
```

Name: std_features, dtype: float64

TIP

When we craft new features based on domain knowledge, we know exactly how we want to combine the variables. We could also combine features with multiple operations and then evaluate whether they are predictive using, for example, a feature selection algorithm or by deriving feature importance from the machine learning model.

6. Let's make a list containing mathematical functions that we want to use to combine the features:

```
math_func = ["sum", "prod", "mean", "std", "max",  
"min"]
```

7. Now, let's apply the previous functions to combine the features, and capture the resulting variables in a new DataFrame:

```
df_t = df[features].agg(math_func, axis="columns")
```

If we execute `df_t.head()`, we will see the DataFrame with the newly created features:

	sum	prod	mean	std	max	min
0	1.08510	0.000351	0.21702	0.080321	0.3001	0.11840
1	0.50165	0.000007	0.10033	0.045671	0.1812	0.07017
2	0.80170	0.000092	0.16034	0.042333	0.2069	0.10960
3	1.03270	0.000267	0.20654	0.078097	0.2839	0.10520
4	0.71630	0.000050	0.14326	0.044402	0.1980	0.10030

Figure 8.1 – DataFrame with the newly created features

NOTE

pandas **agg** can apply multiple functions to combine features. It can take a list of strings with the function names, as we did in *step 7*, a list of NumPy functions, such as **np.log**, and also Python functions that you create.

We can create the same features that we created with pandas automatically by using Feature-engine.

8. Let's create a list by using the name of the output features:

```
new_feature_names = ["sum_f", "prod_f", "mean_f",
"std_f", "max_f", "min_f"]
```

9. Let's set up the **MathFeatures()** transformer so that it applies the functions in *step 6* to the features from *step 3*, and names the new features with the strings from *step 8*:

```
create = MathFeatures(
    variables=features,
    func=math_func,
    new_variables_names=new_feature_names,
)
```

10. Now, let's create the new features, add them to the original DataFrame, and capture it in a new variable:

```
df_t = create.fit_transform(df)
```

We can display the input and output features by executing `df_t[features + new_feature_names].head()`:

	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	sum_f	prod_f	mean_f	std_f	max_f	min_f
0	0.11840	0.27760	0.3001	0.14710	0.2419	1.08510	0.000351	0.21702	0.080321	0.3001	0.11840
1	0.08474	0.07864	0.0869	0.07017	0.1812	0.50165	0.000007	0.10033	0.045671	0.1812	0.07017
2	0.10960	0.15990	0.1974	0.12790	0.2069	0.80170	0.000092	0.16034	0.042333	0.2069	0.10960
3	0.14250	0.28390	0.2414	0.10520	0.2597	1.03270	0.000267	0.20654	0.078097	0.2839	0.10520
4	0.10030	0.13280	0.1980	0.10430	0.1809	0.71630	0.000050	0.14326	0.044402	0.1980	0.10030

Figure 8.2 – DataFrame with the input features and the newly created variables

While pandas `agg` returns a DataFrame with the features resulting from the operation, Feature-engine goes one step further, by concatenating the new features to the original DataFrame.

How it works...

The `pandas` library has plenty of built-in operations to return the desired mathematical and statistical computations over the indicated axis – that is, across the rows or the columns of a DataFrame. In this recipe, we leveraged the power of pandas to create new features from existing ones. First, we made a list containing the names of the features we wanted to combine.

Next, we applied those features over a slice of the DataFrame using the pandas `mean()` and `std()` methods, though we could apply any of the `sum()`, `prod()`, `max()`, and `min()` methods. These methods return the mean, standard deviation, sum, product, and maximum and minimum values of those features, respectively. To perform these operations across the columns, we added the `axis=1` argument within the methods.

With pandas `agg()`, we applied several of the mathematical combinations simultaneously. pandas `agg()` takes, as arguments, a list of strings corresponding to the methods to apply and the `axis` property that the operations should be applied to, which can be either `1` for columns or `0` for rows. pandas `agg()` returned a pandas DataFrame with the feature combination as columns.

Finally, we created the same features by combining variables with Feature-engine. We used the `MathFeatures()` transformer, which takes the features to combine and the functions to apply as input; it also has the option to indicate the names of the new features. When we used `fit()`, the transformer did not learn parameters but checked that the variables were indeed numerical. `transform()` used `pandas.agg` under the hood to apply the mathematical functions to combine the variables.

See also

To find out more about the mathematical operations supported by pandas, go to <https://pandas.pydata.org/pandas-docs/stable/reference/frame.xhtml#computations-descriptive-stats>.

To learn more about pandas **aggregate**, go to
<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.aggregate.xhtml>.

Comparing features to reference variables

In the previous recipe, *Combining features with mathematical functions*, we created new features by applying mathematical or statistical functions, such as the sum or the mean, to a group of variables. Some mathematical operations, however, such as subtraction or division, make more sense when performed between two features, or when considering multiple features against one reference variable. These operations are very useful to derive ratios, such as the **debt-to-income ratio**:

$$\text{debt-to-income ratio} = \text{total debt} / \text{total income}$$

Alternatively, we can use them for differences, for example, to calculate **disposable income**:

$$\text{disposable income} = \text{income} - \text{total debt}$$

In this recipe, we will learn how to create new features via subtraction or division while utilizing pandas, and then automate the procedure for multiple variables by using Feature-engine.

How to do it...

Let's begin by loading the necessary Python libraries and the Breast Cancer dataset from scikit-learn:

1. Let's load the necessary libraries, classes, and data:

```
import pandas as pd  
  
from feature_engine.creation import RelativeFeatures  
from sklearn.datasets import load_breast_cancer
```

2. Let's load the Breast Cancer dataset into a pandas DataFrame:

```
data = load_breast_cancer()  
df = pd.DataFrame(data.data,  
columns=data.feature_names)
```

In the Breast Cancer dataset, there are features for capturing the worst and mean characteristics of the cell nuclei of breast cells. For example, for each image, we have the worst compactness observed in all nuclei and the mean compactness of all nuclei. A feature that captures the difference between the worst and the mean value could be predictive of tumor malignancy.

3. Let's capture the difference between two features, the **worst compactness** and **mean compactness** of cell nuclei, in a new variable and display its values:

```
df["difference"] = df["worst compactness"].sub(df["mean compactness"])  
  
df["difference"].head()
```

In the following code block, we can see the values of the difference between the features:

```
0    0.38800  
1    0.10796  
2    0.26460
```

```
3    0.58240
4    0.07220
Name: difference, dtype: float64
```

NOTE

We can perform the same calculation by executing

```
df["difference"] = df["worst compactness"] -  
(df["mean compactness"]).
```

Similarly, the ratio between the worst and the average characteristic of the cell nuclei might be indicative of malignancy.

4. Now, let's create a new feature with the ratio between the worst and mean radius of the nuclei, and then display its values:

```
df["quotient"] = df["worst radius"].div(  
    df["mean radius"])  
df["quotient"].head()
```

In the following code block, we can see the values of the ratio between the features:

```
0    1.410784
1    1.214876
2    1.197054
3    1.305604
4    1.110892
Name: quotient, dtype: float64
```

NOTE

We can calculate the ratio by executing an alternative command – that is, `df["quotient"] = df["worst radius"] / (df["mean radius"])`.

We can also capture the ratio of every nuclei morphology characteristic and the mean radius or mean area of the nuclei. Let's begin by capturing these subsets of variables into lists.

5. Make a list of the features in the numerator:

```
features = ["mean smoothness",
            "mean compactness", "mean concavity",
            "mean symmetry"]
```

6. Make a list of the features in the denominator:

```
reference = ["mean radius", "mean area"]
```

NOTE

We can create features by dividing all of the features in *step 5* with one of the features in *step 6* with pandas by executing `df[features].sub(df["mean radius"])`.

7. Let's set up Feature-engine's **RelativeFeatures()** so that it subtracts or divides every feature from *step 5* against the features from *step 6*:

```
creator = RelativeFeatures(
    variables=features,
    reference=reference,
    func=["sub", "div"],
```

)

NOTE

Subtracting the features from *step 5* and *step 6* does not make biological sense, but we will do it anyway to demonstrate the power of the **RelativeFeatures()** transformer.

8. Let's add the new features to the DataFrame and capture it in a new variable:

```
df_t = creator.fit_transform(df)
```

9. Let's capture the names of the new features in a list:

```
new_features = [  
    f for f in df_t.columns if f not in  
    creator.feature_names_in_]
```

TIP

The **feature_names_in_** attribute is a fairly new attribute that's now available in most scikit-learn transformers and all Feature-engine transformers and stores the name of the variables in the DataFrame that were used to fit the transformer. When using **transform()** over a DataFrame, the transformers will check that the features match those used in the training set while using this attribute.

If we execute **print(new_features)**, we will see a list containing the names of the features after the combinations. Note that the features contain the variables on the left- and right-hand sides of the equation, plus the function that was applied to them to create the new feature:

```
['mean_smoothness_sub_mean_radius',
```

```
'mean compactness_sub_mean radius',
'mean concavity_sub_mean radius',
'mean symmetry_sub_mean radius',
'mean smoothness_sub_mean area',
'mean compactness_sub_mean area',
'mean concavity_sub_mean area',
'mean symmetry_sub_mean area',
'mean smoothness_div_mean radius',
'mean compactness_div_mean radius',
'mean concavity_div_mean radius',
'mean symmetry_div_mean radius',
'mean smoothness_div_mean area',
'mean compactness_div_mean area',
'mean concavity_div_mean area',
'mean symmetry_div_mean area']
```

Finally, we can display the first five rows of the resulting variables by executing `df_t[new_features].head()`:

	mean smoothness_sub_mean radius	mean compactness_sub_mean radius	mean concavity_sub_mean radius	mean symmetry_sub_mean radius	mean smoothness_sub_mean area	mean compactness_sub_mean area
0	-17.87160	-17.71240	-17.6899	-17.7481	-1000.88160	-1000.72240
1	-20.48526	-20.49136	-20.4831	-20.3888	-1325.91526	-1325.92136
2	-19.58040	-19.53010	-19.4926	-19.4831	-1202.89040	-1202.84010
3	-11.27750	-11.13610	-11.1786	-11.1603	-385.95750	-385.81610
4	-20.18970	-20.15720	-20.0920	-20.1091	-1296.89970	-1296.86720

Figure 8.3 – DataFrame with the newly created features

Feature-engine adds new features at the back of the original DataFrame and automatically adds variable names to those features. By doing so, Feature-engine automates much of the manual work that we would do with pandas.

How it works...

The **pandas** library has plenty of built-in operations to compare one feature or a subset of features to a single reference variable. In this recipe, we used the pandas **sub()** and **div()** methods to determine the difference or the ratio between two variables or a subset of variables and one reference feature.

To subtract one variable from another, we applied the **sub()** method to a pandas Series with the first variable, passing the second pandas Series with the second variable within the method, which returned a third pandas Series

with the second variable subtracted from the first one. To divide one variable from another, we used the `div()` method, which works identically – that is, it divides the variable on the left by the variable passed as an argument of `div()`.

Next, we combined several variables with two reference variables automatically via subtraction or division by utilizing the `ReferenceFeatures()` transformer from Feature-engine.

`ReferenceFeatures()` takes the variables to combine, the reference variables, and the functions to use to combine the former with the latter. When using `fit()`, the transformer did not learn parameters but checked that the variables were numerical. `transform()` added the new features to the DataFrame.

NOTE

`ReferenceFeatures()` can also add, multiply, get the module, or get the power of a group of variables by a group of reference variables. You can find out more in its documentation: https://feature-engine.readthedocs.io/en/latest/api_doc/creation/RelativeFeatures.xhtml.

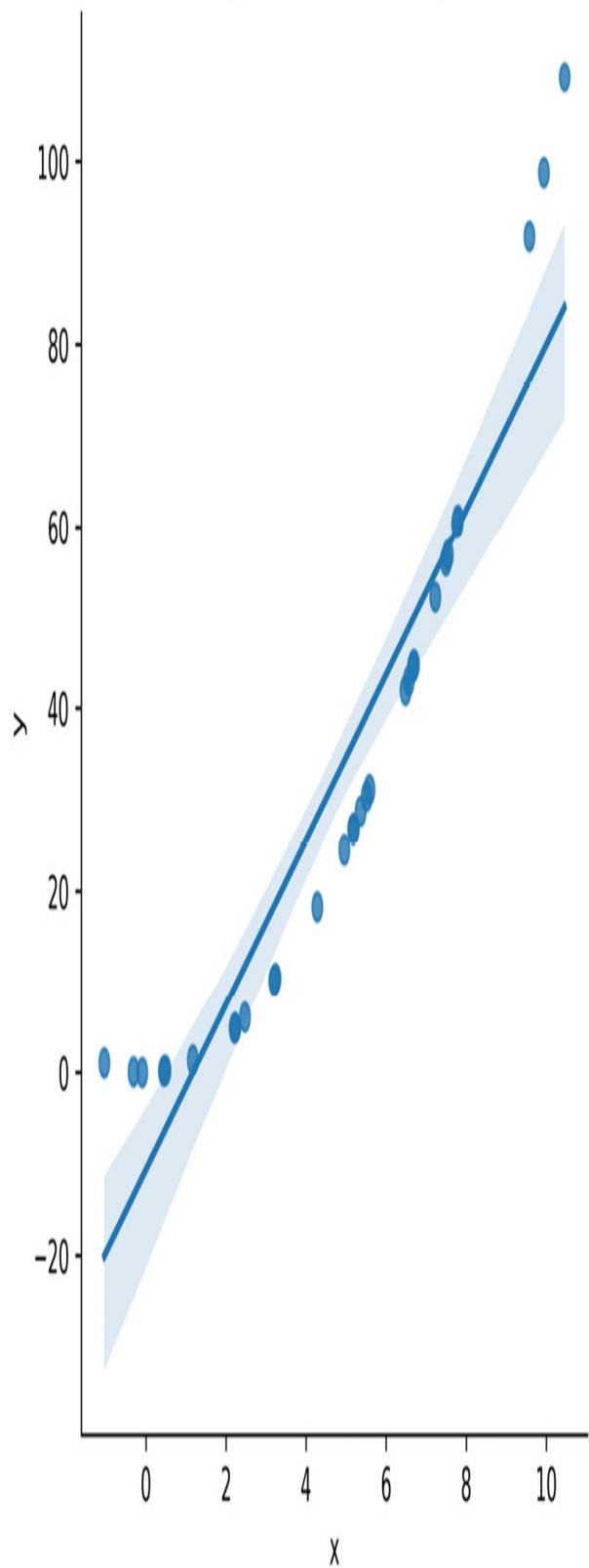
See also

To learn more about the binary operations supported by pandas, go to <https://pandas.pydata.org/pandas-docs/stable/reference/frame.xhtml#binary-operator-functions>.

Performing polynomial expansion

Existing variables can be combined to create new insightful features. We discussed how to combine variables using mathematical and statistical operations in the previous two recipes, *Combining features with mathematical functions* and *Combining features to reference variables*. A combination of one feature with itself – that is, a polynomial combination of the same feature – can also return more predictive features. For example, in cases where the target has a quadratic relation with a variable, creating a second-degree polynomial of the feature allows us to use it in a linear model, as shown in the following figure:

Quadratic relationship



Linear relationship with quadratic feature

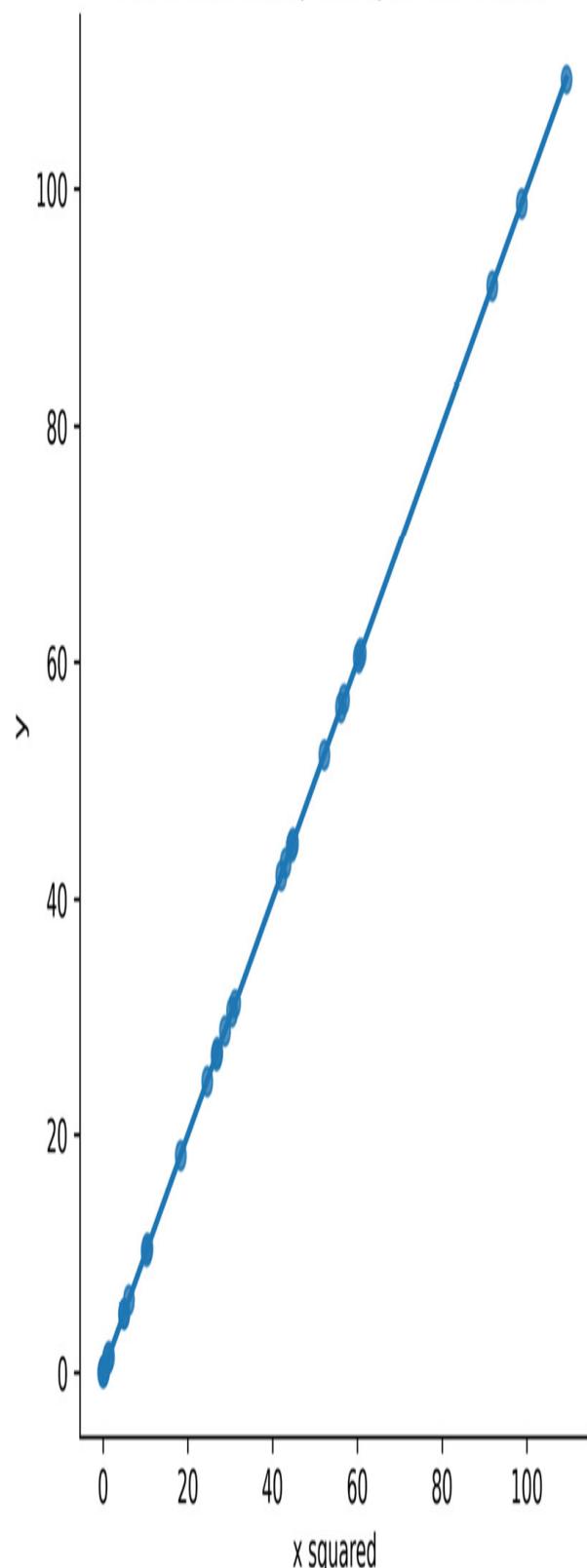


Figure 8.4 – Change in the relationship between a target and a predictor variable after squaring the values of the latter

In the plot on the left, due to the quadratic relationship between the target, y , and the variable, x , there is a poor linear fit. Yet, in the plot on the right, we appreciate how the **$x_squared$** variable, which is a quadratic combination of x , shows a linear relationship with the target, y , and therefore improves the performance of the linear model, which predicts y from **$x_squared$** .

With similar logic, polynomial combinations of the same or different variables can return new variables that convey additional information and capture feature interaction and can, therefore, result in useful inputs for our linear models. We can create polynomial variables automatically using scikit-learn, and, in this recipe, we will learn how to do so.

Getting ready

Polynomial expansion serves to automate the creation of new features, capture feature interaction, and capture potential non-linear relationships between the original variables and the target. To create polynomial features, we need to determine which features to combine and which polynomial degree to use.

NOTE

High polynomial degrees or a large number of features to combine will return an enormous number of new features. High polynomial degrees may also result in overfitting.

The **PolynomialFeatures()** transformer from scikit-learn creates polynomial combinations of the features with a degree less than or equal to the specified degree, automatically. To follow up easily with this recipe, let's understand the output of the **PolynomialFeatures()** transformer from scikit-learn, when used to create second- and third-degree polynomial combinations of three variables.

Second-degree polynomial combinations of three variables – a , b , and c – will return the following new features:

$$1, a, b, c, ab, ac, bc, a^2, b^2, c^2$$

From the previous features, a , b , and c are the original variables; ab , ac , and bc are the products of those features; and a^2 , b^2 , and c^2 are the squared values of the original features. The **PolynomialFeatures()** transformer also returns the bias term 1 , which we would probably exclude when creating features.

TIP

The resulting features – ab , ac , and bc – are called interactions or feature interactions of degree 2. The degree reflects the number of variables combined. The result combines a maximum of two variables because we indicated a second-degree polynomial as the maximum allowed combination.

Third-degree polynomial combinations of the three variables – a , b , and c – will return the following new features:

$$1, a, b, c, ab, ac, bc, abc, a^2b, a^2c, b^2a, b^2c, c^2a, c^2b, a^3, b^3, c^3$$

Among the returned features, in addition to those returned by the second-degree polynomial combination, we now have the third-degree combinations of the features with themselves (a^3 , b^3 , and c^3); the squared values of every feature combined linearly with a second feature (a^2b , a^2c , b^2a , b^2c , c^2a , and c^2b); and the product of the three features (abc). Note how we have all possible interactions of degrees 1, 2, and 3 and the *bias term 1*.

Now that we understand the output of the polynomial expansion, let's jump into the recipe.

How to do it...

In this recipe, we will create features via polynomial expansion using a toy dataset so that we become familiar with the returned variables. Creating features via polynomial expansion of a real dataset is identical to what we will discuss in this recipe. Follow these steps:

1. Let's import the required libraries, classes, and data:

```
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
from sklearn.preprocessing import PolynomialFeatures
```

2. Let's create a DataFrame containing one variable with values from 1 to 10:

```
df = pd.DataFrame(np.linspace(0, 10, 11), columns=["var"])
```

3. Let's set up the transformer to create all possible combinations up to a third-degree polynomial of the single variable and exclude the bias term from the result – that is, we will exclude the value 1:

```
poly = PolynomialFeatures(  
    degree=3,  
    interaction_only=False,  
    include_bias=False)
```

4. Now, let's create the polynomial combinations and display the result:

```
dft = poly.fit_transform(df)  
dft
```

The new features are returned in a NumPy array, where we can see the original feature followed by its values squared and then its values to the power of three:

```
array([[  0.,    0.,    0.],  
       [  1.,    1.,    1.],  
       [  2.,    4.,    8.],  
       [  3.,    9.,   27.],  
       [  4.,   16.,   64.],  
       [  5.,   25.,  125.],  
       [  6.,   36.,  216.],  
       [  7.,   49.,  343.],  
       [  8.,   64.,  512.],  
       [  9.,   81.,  729.]])
```

```
[ 10., 100., 1000.]])
```

5. We can obtain the name of the new features as follows:

```
poly.get_feature_names_out()
```

The previous command returns the names of the new features, where \wedge indicates the degree of the polynomial combination:

```
array(['var', 'var^2', 'var^3'], dtype=object)
```

6. Now, let's capture the array from *step 4* in a pandas DataFrame, add the feature names from *step 5* as column names, and plot the new feature values against the original variable:

```
dft = pd.DataFrame(  
    dft, columns=poly.get_feature_names_out())  
  
plt.plot(df["var"], dft)  
plt.legend(dft.columns)  
plt.xlabel("original variable")  
plt.ylabel("new variables")  
plt.show()
```

Here, we can see the relationship that the polynomial features have with the original feature values:

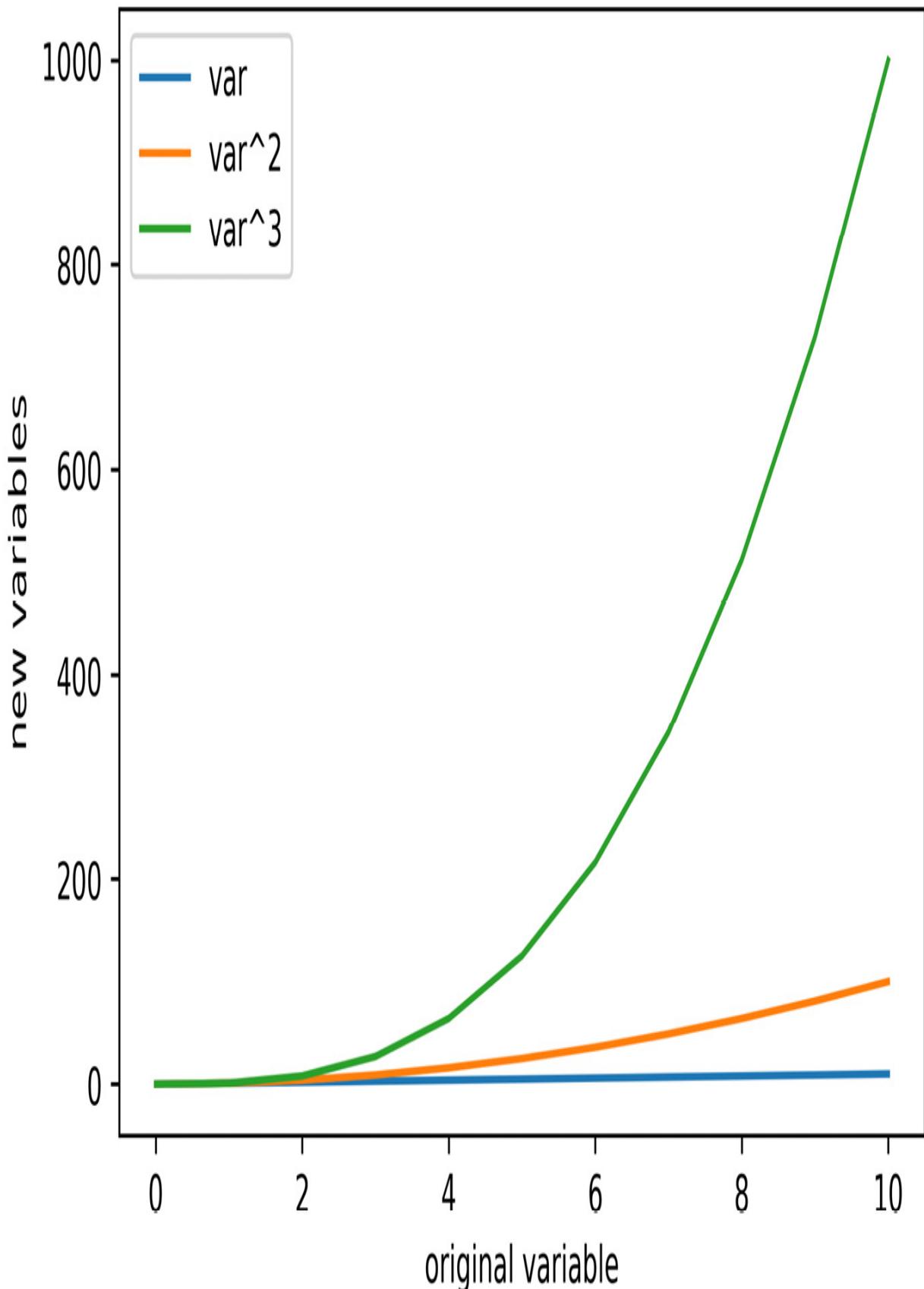


Figure 8.5 – Relationship between the polynomial expansion features and the original variable values

7. Let's add two additional variables to our toy dataset, with values from 1 to 10:

```
df["col"] = np.linspace(0, 5, 11)  
df["feat"] = np.linspace(0, 5, 11)
```

8. Next, let's combine the three features in the dataset with polynomial expansion up to the second degree, but this time, we will only return features produced by combining at least two different variables:

```
poly = PolynomialFeatures(  
    degree=2, interaction_only=True,  
    include_bias=False)  
  
dft = poly.fit_transform(df)
```

9. Let's capture the feature combinations in a DataFrame and add the feature names:

```
dft = pd.DataFrame(  
    dft, columns=poly.get_feature_names_out()  
)
```

If we execute **df**, we will see the features resulting from the polynomial expansion, which contain the original features plus all possible combinations of the three variables:

	var	col	feat	var col	var feat	col feat
0	0.0	0.0	0.0	0.0	0.0	0.00
1	1.0	0.5	0.5	0.5	0.5	0.25
2	2.0	1.0	1.0	2.0	2.0	1.00
3	3.0	1.5	1.5	4.5	4.5	2.25
4	4.0	2.0	2.0	8.0	8.0	4.00
5	5.0	2.5	2.5	12.5	12.5	6.25
6	6.0	3.0	3.0	18.0	18.0	9.00
7	7.0	3.5	3.5	24.5	24.5	12.25
8	8.0	4.0	4.0	32.0	32.0	16.00
9	9.0	4.5	4.5	40.5	40.5	20.25
10	10.0	5.0	5.0	50.0	50.0	25.00

Figure 8.6 – DataFrame with the original features plus all possible combinations of two features

TIP

Go ahead and create third-degree polynomial combinations of the features, returning only the interactions or all possible features to get a better sense of the output of **PolynomialFeatures()**.

With that, we've learned how to expand the feature space by combining existing features either with themselves or with other features in the data. Creating features via polynomial expansion using a real dataset is, in essence, identical. If we want to combine only a subset of the features, we can select the features to combine by utilizing **ColumnTransformer()**.

How it works...

In this recipe, we created new features by using polynomial combinations of three variables in a toy dataset. To create these polynomial features, we used the **PolynomialFeatures()** transformer from scikit-learn, which generates a new feature matrix consisting of all polynomial combinations of the indicated features with a degree less than or equal to the specified **degree**, by default. By setting **degree** to **3**, we were able to create all possible polynomial combinations of degree 3 or smaller. To retain all of the terms of the expansion, we set **interaction_only** to **False**. To avoid returning the bias term, we set the **include_bias** parameter to **False**.

NOTE

Setting the **interaction_only** term to **True** only returns the interaction terms – that is, the variables that contain combinations of two or more variables.

The **fit()** method learned all of the possible feature combinations based on the parameters specified. At this stage, the transformer did not perform actual mathematical computations. The **transform()** method performed the mathematical computations with the features to create the new variables. With the **get_feature_names()** method, we could identify the terms of the expansion – that is, how each new feature was calculated.

There's more...

Now, let's create features by performing polynomial expansion on a subset of variables in the Breast Cancer dataset:

1. Let's import the necessary libraries, classes, and data:

```
import pandas as pd

from sklearn.datasets import load_breast_cancer

from sklearn.compose import ColumnTransformer

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import PolynomialFeatures
```

2. Let's load the data and separate it into train and test sets:

```
data = load_breast_cancer()

df = pd.DataFrame(data.data,
columns=data.feature_names)

X_train, X_test, y_train, y_test = train_test_split(
    df, data.target, test_size=0.3, random_state=0

)
```

3. Let's make a list with the features to combine:

```
features = ["mean smoothness", "mean compactness",
"mean concavity"]
```

4. Let's set up the transformer to create all possible combinations up to the third degree:

```
poly = PolynomialFeatures(
    degree=3, interaction_only=False,
    include_bias=False)
```

5. Let's set up the column transformer to create features only from those specified in *step 3*:

```
ct = ColumnTransformer([("poly", poly, features)])
```

6. Let's create the polynomial features:

```
train_t = ct.transform(X_train)  
test_t = ct.transform(X_test)
```

And that's it. By executing `ct.get_feature_names_out()`, we obtain the names of the new features. By executing `test_t = pd.DataFrame(test_t, columns=ct.get_feature_names_out())` we capture the features in a DataFrame; then, we can display their values with `pandas.head()`.

See also

To learn more about `PolynomialFeatures()` from scikit-learn, go to <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.xhtml>.

You can use the Python `gplearn` package to automatically map out other relationships between the variables and the target:

<https://gplearn.readthedocs.io/en/stable/intro.xhtml>.

Combining features with decision trees

In the winning solution of the KDD competition in 2009, the authors created new features by combining two or more variables using decision trees. When examining the variables, they noticed that some features had a high level of mutual information with the target yet low correlation,

indicating that the relationship with the target was not monotonic. While these features were predictive when used in tree-based algorithms, linear models could not take advantage of them. Hence, to use these features in linear models, they replaced the features with the outputs of decision trees trained on the individual features, or combinations of two or three variables, to return new features with a monotonic relationship with the target.

So, in short, combining features with decision trees is particularly useful for deriving features that are monotonic with the target, which is convenient for linear models. The procedure consists of training a decision tree using a subset of the features – typically one, two, or three at a time – and then using the prediction of the tree as a new feature.

In this recipe, we will learn how to create new features with decision trees using pandas and scikit-learn.

Getting ready

You can find more details about this procedure and the overall winning solution of the 2009 KDD data competition in this article:

<http://proceedings.mlr.press/v7/niculescu09/niculescu09.pdf>.

In this recipe, we will combine two features of the California Housing dataset, and then evaluate the relationship between the original and new features with the target to understand more about the intended output of this feature engineering technique.

How to do it...

Let's begin by importing the required libraries and getting the dataset ready:

1. Import pandas and the required functions, classes, and datasets from scikit-learn:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import GridSearchCV
```

2. Let's load the California Housing dataset and its target into a pandas DataFrame:

```
X, y = fetch_california_housing(
    return_X_y=True, as_frame=True)
```

3. Let's separate the dataset into train and test sets:

```
X_train, X_test, y_train, y_test = train_test_split(X,
    y, test_size=0.3, random_state=0)
```

NOTE

Remember to set **random_state**, as indicated in step 3, for reproducibility.

In the following lines, we are going to create a new feature from two variables in the dataset using a decision tree. We are going to build this

decision tree within **GridSearch()** so that we can optimize the tree's depth.

4. Let's create a dictionary with the parameter to optimize and a few of the potential values it can take:

```
param_grid = {"max_depth": [2, 3, 4, None]}
```

NOTE

To find out which other parameters you can optimize, go to

<https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.xhtml>.

5. Let's set up the decision tree within a scikit-learn **GridSearch()** with five-fold cross-validation, add the dictionary with the parameter to optimize that we created in *step 4*, and indicate that we want to optimize the mean squared error of the model:

```
tree_model = GridSearchCV(  
    DecisionTreeRegressor(random_state=0),  
    cv=5,  
    scoring="neg_mean_squared_error",  
    param_grid=param_grid,  
)
```

NOTE

We are using **DecisionTreeRegressor()** from scikit-learn because the target in this dataset is continuous. If you have a binary target or are performing classification, use **DecisionTreeClassifier()**

instead. Note that you will have to change the scoring metric to one permitted for classification.

6. Let's make a list containing the variables we want to combine:

```
variables = ["AveRooms", "AveBedrms"]
```

7. Now, let's train the decision tree using the selected features:

```
tree_model.fit(X_train[variables], y_train)
```

8. Let's add the new feature that's returned by the decision tree in the train and test sets:

```
X_train["new_feat"] = tree_model.predict(  
    X_train[variables])  
  
X_test["new_feat"] = tree_model.predict(  
    X_test[variables])
```

With that, we have created a new feature by combining the information of two existing features using a decision tree. Let's explore the relationship between the original and new features with the target.

9. Let's make a scatter plot of each feature individually against the target, and then the new feature against the target, and display them side by side. The y -axis, which will contain the values of the target variable, will be the same for all plots:

```
fig, axs = plt.subplots(1, 3, figsize=(12, 4),  
sharey=True)  
  
axs[0].scatter(X_test["AveRooms"], y_test)  
axs[0].set_ylabel("House price")
```

```
axs[0].set_xlabel("AveRooms")
axs[1].scatter(X_test["AveBedrms"], y_test)
axs[1].set_xlabel("AveBedrms")
axs[2].scatter(X_test["new_feat"], y_test)
axs[2].set_xlabel("new_feat")
plt.show()
```

We can begin to appreciate the monotonic relationship between the new variable and the target in the right-hand plot, where there isn't a clear relationship between the original features and the target:

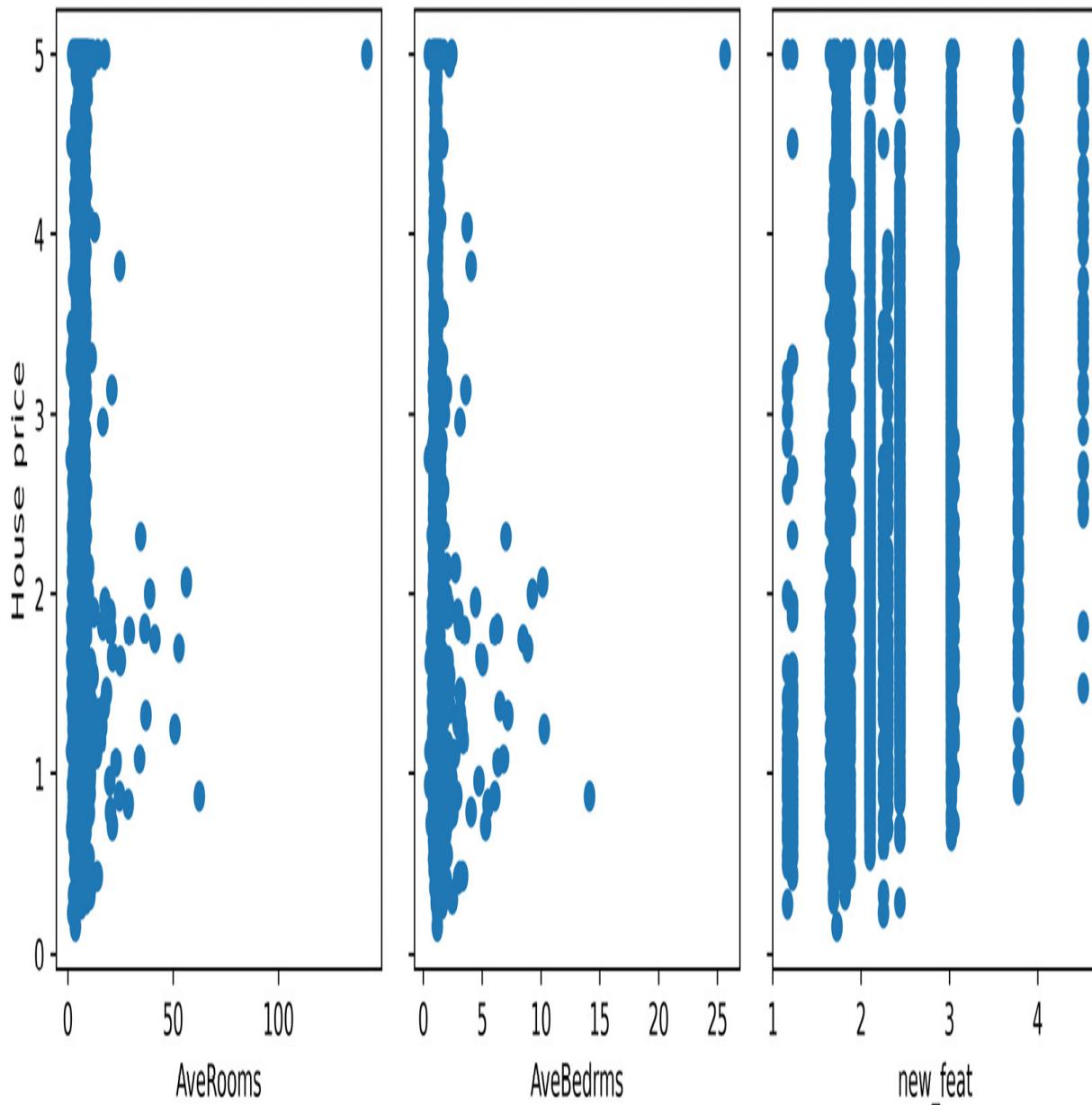


Figure 8.7 – Scatter plots between the original or new variables and the target

10. Finally, let's print out the correlation between the original features and the new variable with the target:

```
for var in variables + ["new_feat"]:  
    corr = np.corrcoef(X_test[var], y_test)[0, 1]
```

```
corr = np.round(corr, 3)

print(f"Correlation between {var} and the target:
{corr}")
```

As shown in the following output, the new feature has a higher correlation with the target, so it should be more predictive when used in linear models:

Correlation between AveRooms and the target: 0.141

Correlation between AveBedrms and the target: -0.033

Correlation between new_feat and the target: 0.466

I hope I have given you a flavor of the power of combining features with decision trees and how to do so using Python and scikit-learn.

How it works...

In this recipe, we combined two variables into a new feature while utilizing a decision tree. We loaded the dataset from scikit-learn and then separated the data into train and test sets using the **train_test_split()** function. Next, we created a dictionary with the decision tree parameter to optimize as the key, and a list of the values to examine as values.

Next, we created an instance of a decision tree for regression by using **DecisionTreeRegressor()** from scikit-learn inside **GridSearch()**, indicating the cross-validation fold, the metric to optimize, and the dictionary with the parameters and values to examine. Next, we fit the decision tree using the two variables that we wanted to combine, and, finally, with the **predict()** method, we obtained the predictions derived

by the tree from those two features, which we captured as a new feature in the DataFrame.

Then, we compared the relationship of the original or new feature with the target through scatter plots and by evaluating their correlation.

Creating periodic features from cyclical variables

Some features are periodic, for example, the hours in a day, the months in a year, and the days in a week. They all start at a certain value, say January, go up to a certain other value, say December, and then start over from the beginning. Some features are numeric, such as the hours, and some can be represented with numbers, such as the months, with values of 1 to 12. Yet, this numeric representation does not capture the periodicity or cyclical nature of the variable. For example, December (12) is closer to January (1) than June (6); however, this relationship is not captured by the numerical representation of the feature. But we could change it if we transformed these variables with sine and cosine, two naturally periodic functions.

Encoding cyclical features with the sine and cosine functions allows linear models to leverage the cyclical nature of features and reduce their modeling error. In this recipe, we will create new features from periodic variables that capture the cyclical nature of the feature.

Getting ready

Trigonometric functions, such as sine and cosine, are periodic, with values cycling between -1 and 1 every 2π cycles:

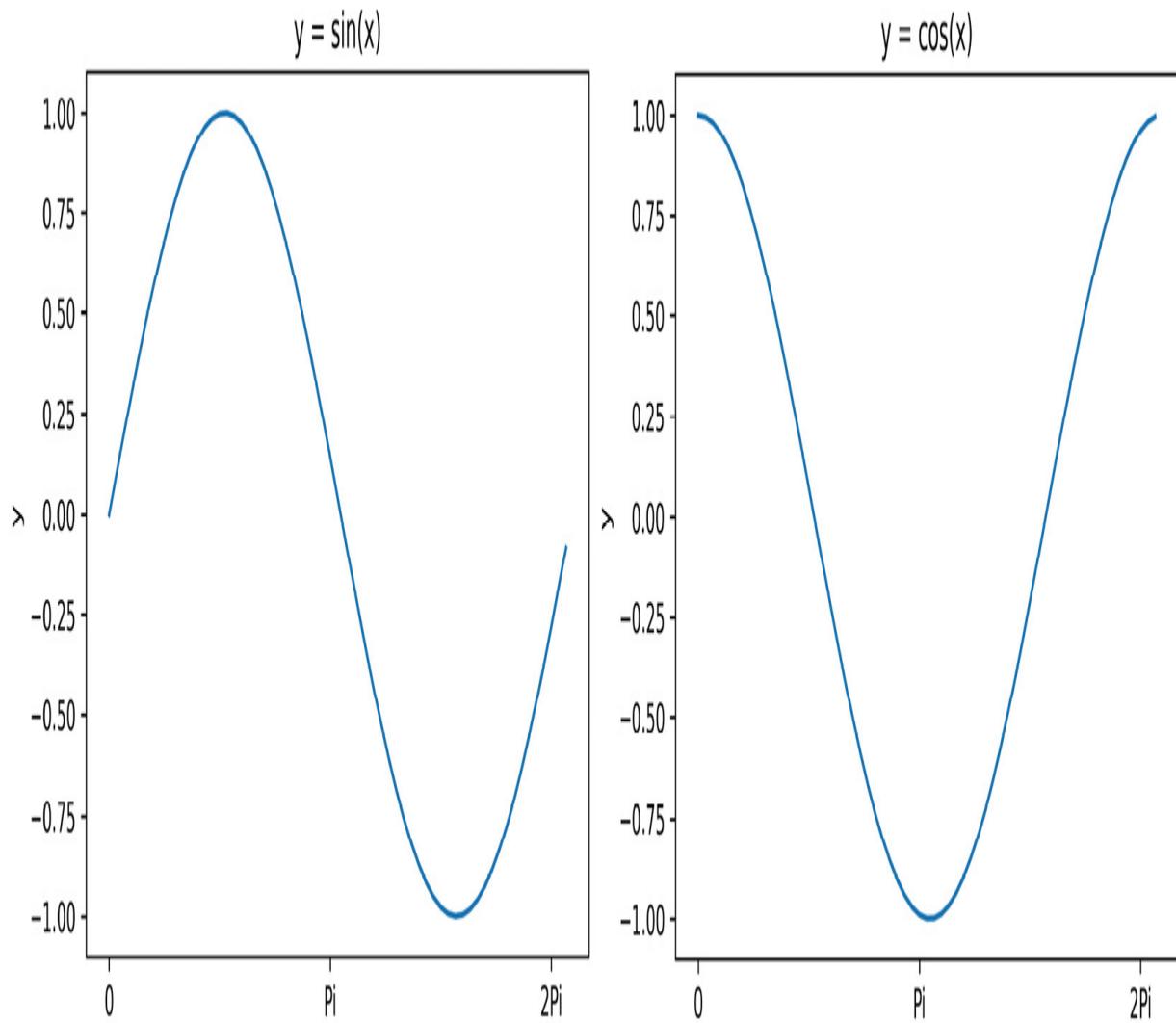


Figure 8.8 – Sine and cosine functions

We can capture the periodicity of a cyclical variable by applying a trigonometric transformation after normalizing the variable values between 0 and 2π :

$$\sin(x) = \sin\left(2\pi \frac{x}{X_{max}}\right)$$

Dividing the variable's values by its maximum will normalize it between 0 and 1, whereas multiplying it by 2π will rescale the variable between 0 and 2π .

Should we use sine? Or should we use cosine? The thing is, we need to use both to encode all the values of the variables unequivocally. Since sine and cosine circle between 0 and 1, they will take a value of 0 for more than 1 value of x . For example, the sine of 0 returns 0, and so does the sine of π . So, if we encode a variable with just the sine, we wouldn't be able to distinguish between the values 0 and π anymore. However, because the sine and the cosine are out of phase, the cosine of 0 returns 1, whereas the cosine of π returns -1. Hence, by encoding the variable with the two functions, we are now able to distinguish between 0 and 1, which would take (0,1) and (0,-1) as values for the sine and cosine functions, respectively.

How to do it...

In this recipe, first, we will transform the **hour** variable in a toy DataFrame with the sine and the cosine to get a sense of the new variable representation. Then, we will automate feature creation from multiple cyclical variables using Feature-engine:

1. Let's begin by importing the necessary libraries:

```
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt
```

2. Let's create a toy DataFrame with one variable – **hour** – with values between 0 and 23:

```
df = pd.DataFrame([i for i in range(24)], columns=["hour"])
```

3. Next, we will create two features using the sine and cosine transformations, after normalizing the variable values between 0 and 2π :

```
df["hour_sin"] = np.sin(  
    df["hour"] / df["hour"].max() * 2 * np.pi)  
  
df["hour_cos"] = np.cos(  
    df["hour"] / df["hour"].max() * 2 * np.pi)
```

4. If we execute **df.head()**, we will see the original and new features:

	hour	hour_sin	hour_cos
0	0	0.000000	1.000000
1	1	0.269797	0.962917
2	2	0.519584	0.854419
3	3	0.730836	0.682553
4	4	0.887885	0.460065

Figure 8.9 – DataFrame with the hour variable and the new features obtained by the sine and cosine transformations

5. Let's make a scatter plot between the hour and its sine-transformed values:

```
plt.scatter(df["hour"], df["hour_sin"])
plt.ylabel("Sine of hour")
plt.xlabel("Hour")
plt.title("Sine transformation")
```

In the following plot, we can see how the values of the hour circle between -1 and 1, just like the sine function after the transformation:

Sine transformation

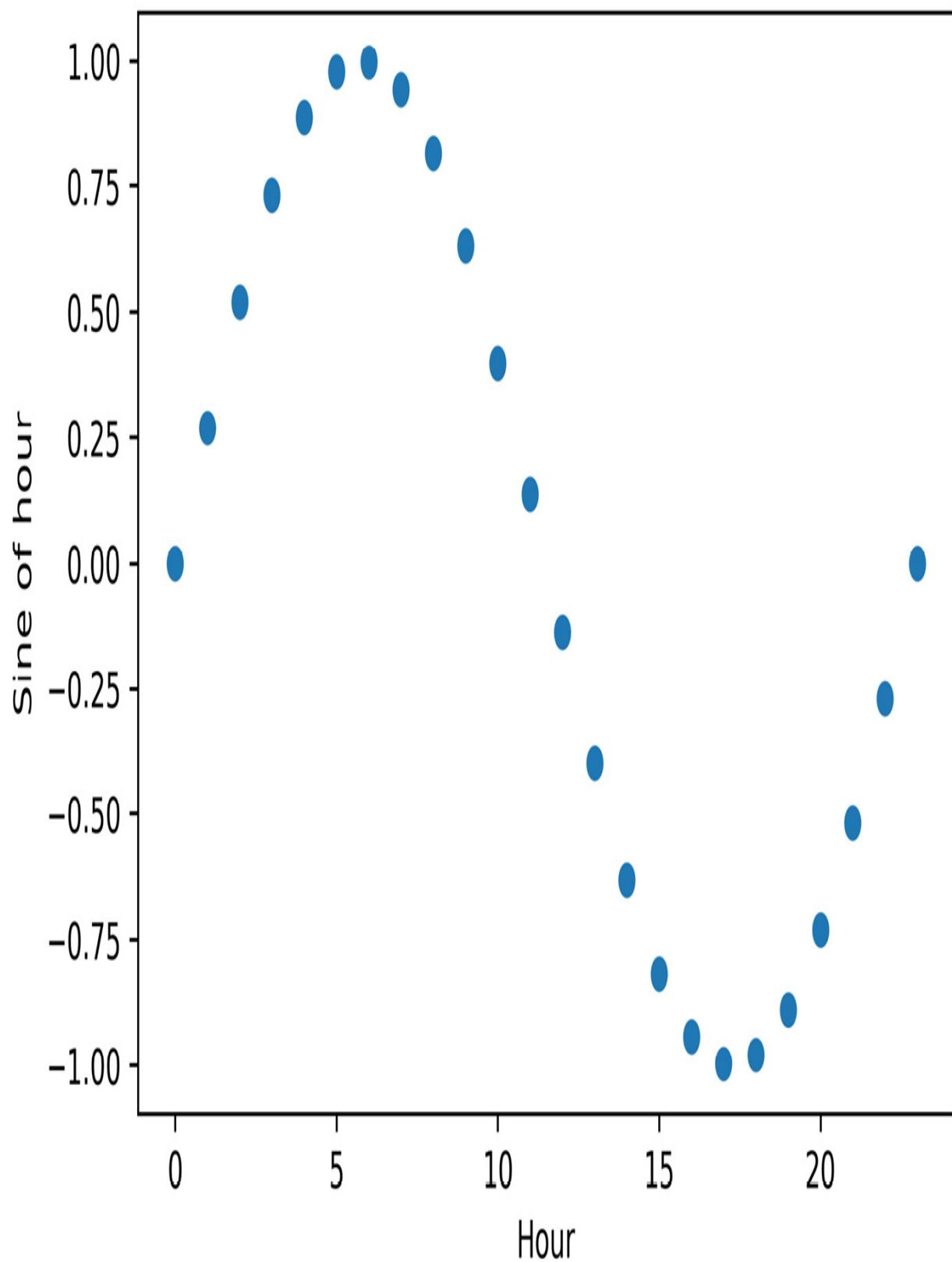


Figure 8.10 – Scatter plot of the hour versus its sine transformed values

6. Now, let's make a scatter plot between the hour and its cosine transformation:

```
plt.scatter(df["hour"], df["hour_cos"])
plt.ylabel("Cosine of hour")
plt.xlabel("Hour")
plt.title("Cosine transformation")
```

In the following plot, we can see how the values of the hour circle between -1 and 1, just like the cosine function after the transformation:

Cosine transformation

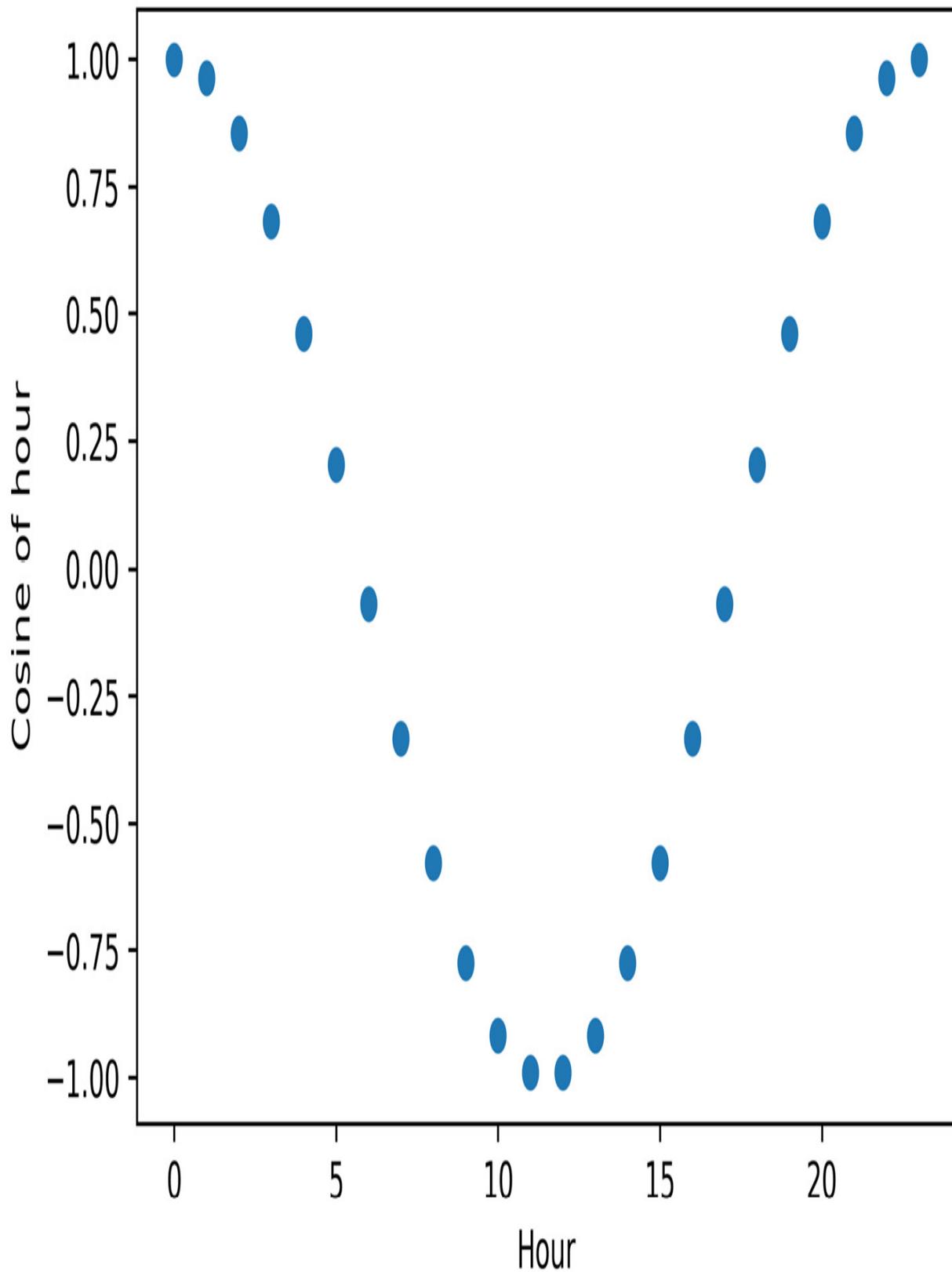


Figure 8.11 – Scatter plot of the hour versus its cosine-transformed values

Finally, we can reconstitute the cyclical nature of the hour, which is now captured by the two new features.

7. Let's plot the values of the sine versus the cosine of the hour and overlay the original values of the hour on a color map:

```
fig, ax = plt.subplots(figsize=(7, 5))

sp = ax.scatter(df["hour_sin"], df["hour_cos"],
c=df["hour"])

ax.set(
    xlabel="sin(hour)",
    ylabel="cos(hour)",
)

_ = fig.colorbar(sp)
```

In the following plot, we can see how the two trigonometric transformations of the hour reflect the cyclical nature of the hour, in a plot that reminds us of a clock:

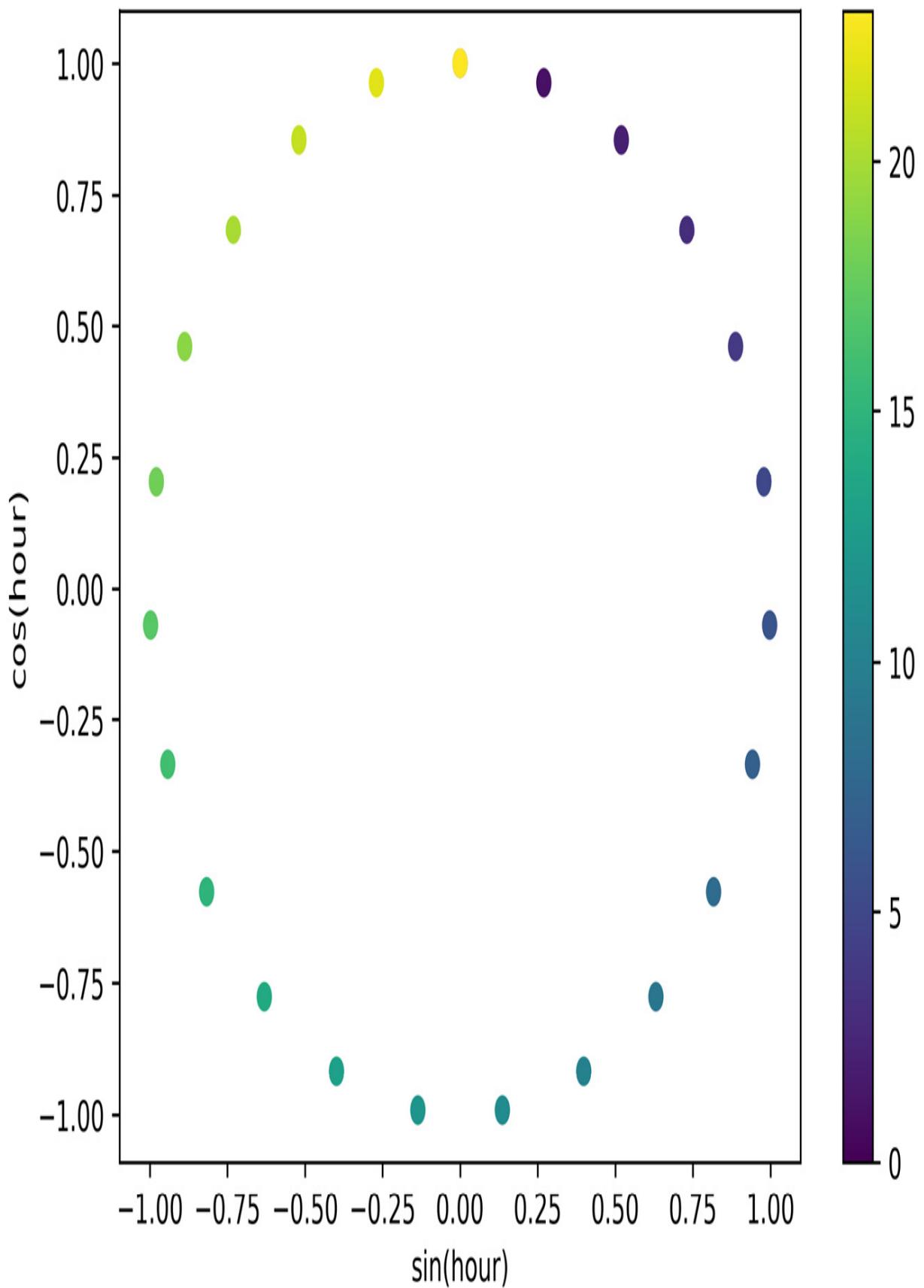


Figure 8.12 – Scatter plot of the trigonometric transformation of the hour

NOTE

The code implementation and idea for this plot were taken from scikit-learn's documentation: https://scikit-learn.org/stable/auto_examples/applications/plot_cyclical_feature_engineering.xhtml#trigonometric-features.

Now that we understand the nature and effect of the transformation, let's create new features using the sine and cosine transformations from multiple variables automatically. We will use Feature-engine's **CyclicalFeatures**.

8. Let's import **CyclicalFeatures**:

```
from feature_engine.creation import CyclicalFeatures
```

9. Let's create a toy DataFrame that contains the **hour**, **month**, and **week** variables, whose values vary between 0 and 24, 1 and 12, and 0 and 6, respectively:

```
df = pd.DataFrame()
df["hour"] = pd.Series([i for i in range(24)])
df["month"] = pd.Series([i for i in range(1, 13)]*2)
df["week"] = pd.Series([i for i in range(7)]*4)
```

Now, if we execute **df.head()**, we will see the first five rows of the toy DataFrame:

	hour	month	week
0	0	1	0
1	1	2	1
2	2	3	2
3	3	4	3
4	4	5	4

Figure 8.13 – Toy DataFrame with three cyclical features

- Let's set up the transformer so that it creates the sine and cosine features from these variables:

```
cyclic = CyclicalFeatures(  
    variables=None,  
    drop_original=False,  
)
```

NOTE

By setting **variables** to **None**, the transformer will create the trigonometric features from all numerical variables. Alternatively, we can pass the subset of cyclical variables in a list. We can retain or drop the original variables after creating the cyclical features using the **drop_original** parameter.

- To finish, let's add the features to the DataFrame and capture it in a new variable:

```
dft = cyclic.fit_transform(df)
```

Now, if we execute `dft.head()`, we will see the original and new features:

	hour	month	week	hour_sin	hour_cos	month_sin	month_cos	week_sin	week_cos
0	0	1	0	0.000000	1.000000	0.500000	8.660254e-01	0.000000e+00	1.0
1	1	2	1	0.269797	0.962917	0.866025	5.000000e-01	8.660254e-01	0.5
2	2	3	2	0.519584	0.854419	1.000000	6.123234e-17	8.660254e-01	-0.5
3	3	4	3	0.730836	0.682553	0.866025	-5.000000e-01	1.224647e-16	-1.0
4	4	5	4	0.887885	0.460065	0.500000	-8.660254e-01	-8.660254e-01	-0.5

Figure 8.14 – DataFrame with cyclical features plus the features created through the sine and cosine functions

And that's it – we've created features by using the sine and cosine transformation automatically from multiple variables and added them directly to the original DataFrame.

How it works...

In this recipe, we encoded cyclical features by values obtained from the sine and cosine functions applied to the normalized values of the variable. First, we normalized the variable values between 0 and 2π . To do this, we

divided the variable values by the variable maximum value, which we obtained with `pandas.max()`, to scale the variables between 0 and 1. Next, we multiplied those values by 2π , using `numpy.pi`. Finally, we used `np.sin` and `np.cos` to apply the sine and cosine transformations, respectively.

To automate these procedures for multiple variables, we used Feature-engine's `CyclicalFeatures`. With `fit()`, the transformer learned the maximum values of each variable, and with `transform()`, it added the features resulting from the sine and cosine transformations to the DataFrame.

See also

More details about cyclical features can be found in Feature-engine's documentation: <https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html>.

Creating spline features

Linear models expect a linear relationship between the predictor variables and the target. However, we can use linear models to model non-linear effects if we first transform the features. In the *Performing polynomial expansion* recipe, we saw how we can unmask linear patterns by creating features with polynomial functions. In this recipe, we will discuss the use of splines.

Splines are used to mathematically reproduce flexible shapes. They consist of piecewise low-degree polynomial functions. To create splines, we must place knots at several values of x within its value range. These knots indicate where the pieces of the function join together. Then, we fit low-degree polynomials to the data between two consecutive knots.

There are several types of splines, such as smoothing splines, regression splines, and B-splines. scikit-learn supports the use of B-splines to create features. The procedure to fit and therefore return the spline values for a certain variable based on a polynomial degree and the number of knots exceeds the scope of this recipe. For more details, check out the resources in the *See also* section of this recipe. In this recipe, we'll get a sense of what splines are and how we can use them to improve the performance of linear models.

Getting ready

Let's get a sense of what splines are. In the following figure, on the left, we can see a spline of degree 1. It consists of two linear pieces – one from 2.5 to 5 and the other from 5 to 7.5. There are three knots: 2.5, 5, and 7.5. And outside the interval of 2.5 to 7.5, the spline takes a value of 0. The latter is characteristic of splines; they are only non-negative between certain values. On the right-hand side of the figure, we can see three splines of degree 1. We can construct as many splines as we want by introducing more knots:

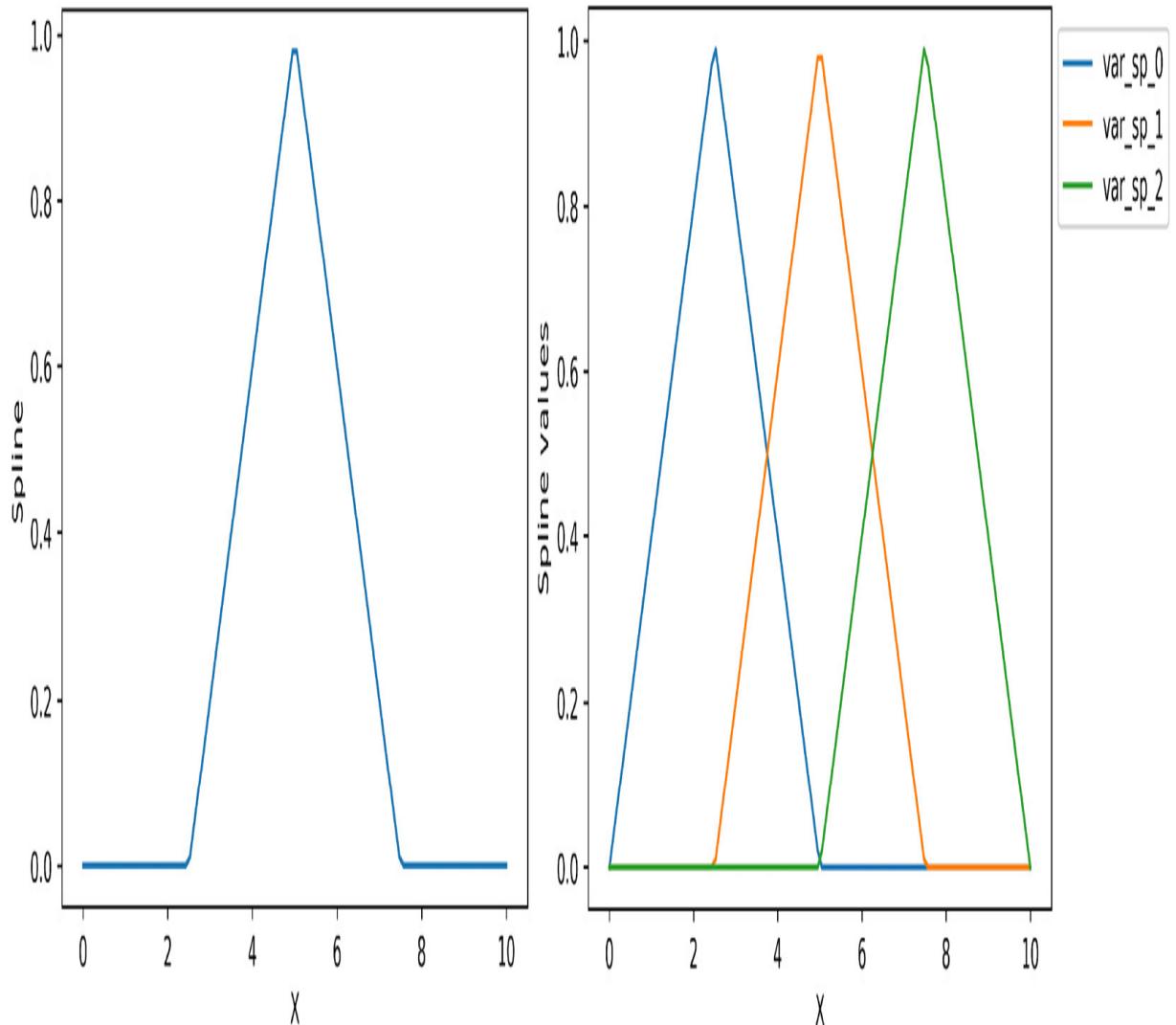


Figure 8.15 – Splines of degree 1

In the following figure, on the left, we can see a quadratic spline, also known as a spline of degree 2. It is based on four adjacent knots: 0, 2.5, 5, and 7.5. On the right-hand side of the figure, we can see several splines of degree 2:

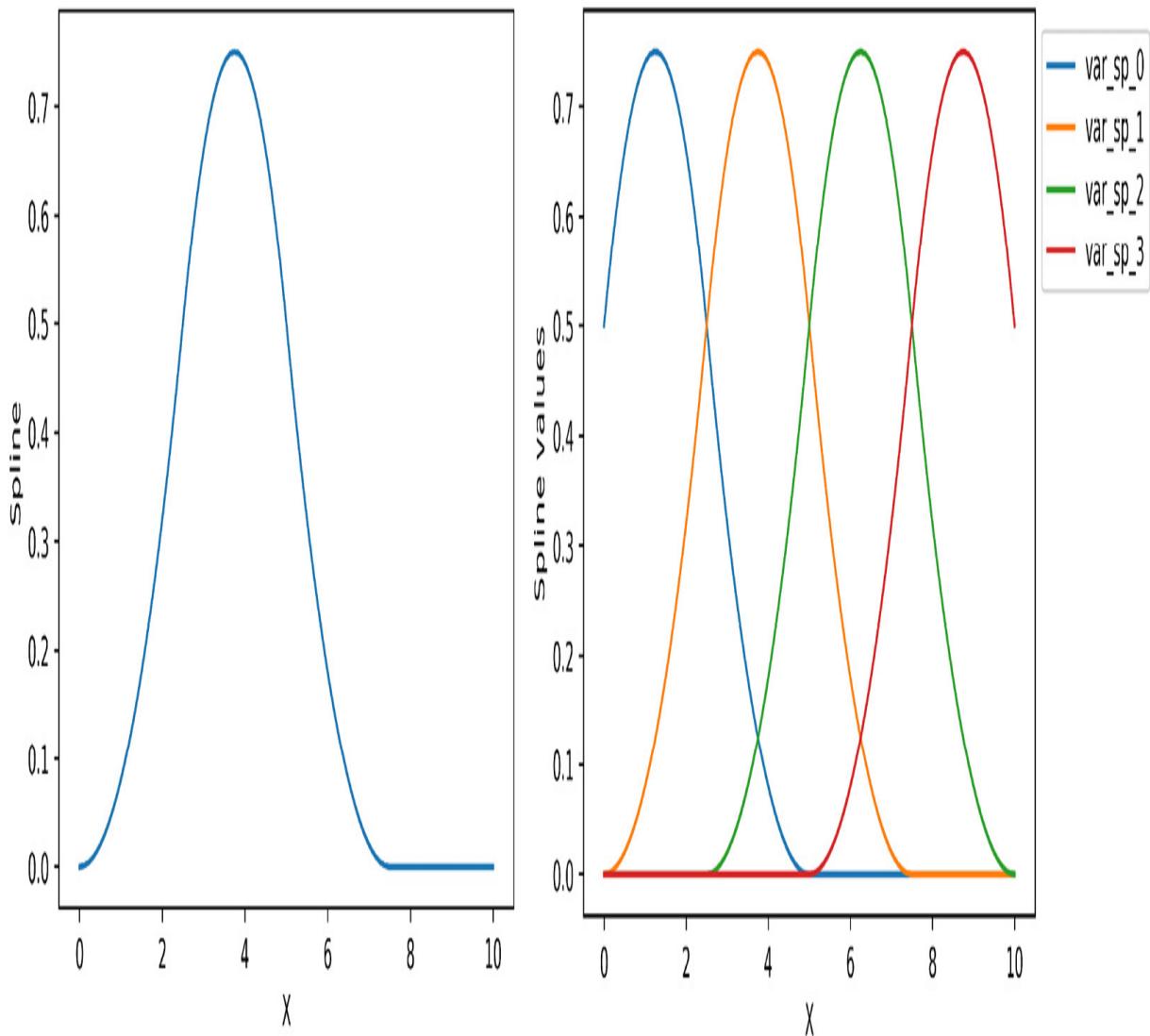


Figure 8.16 – Splines of degree 2

We can use splines to model non-linear functions and we will learn how to do this in the next section.

How to do it...

In this recipe, we will use splines to model the sine function. Once we have gotten a sense of what splines are and how we can use them to fit non-linear

covariates through a linear model, we will use splines for regression in a real dataset:

NOTE

The idea to model the sine function with splines was taken from scikit-learn's documentation: https://scikit-learn.org/stable/auto_examples/linear_model/plot_polynomial_interpolation.xhtml.

1. Let's begin by importing the necessary libraries and classes:

```
import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt  
from sklearn.linear_model import Ridge  
from sklearn.preprocessing import SplineTransformer
```

2. Let's create a training set, **X**, with 20 values between -1 and 11, and the target variable, **y**, which is the sine of **X**:

```
x = np.linspace(-1, 11, 20)  
y = np.sin(x)
```

3. Let's plot the relationship between **X** and **y**:

```
plt.plot(x, y)  
plt.ylabel("y")  
plt.xlabel("X")
```

In the following plot, we can see the sine function of **X**:

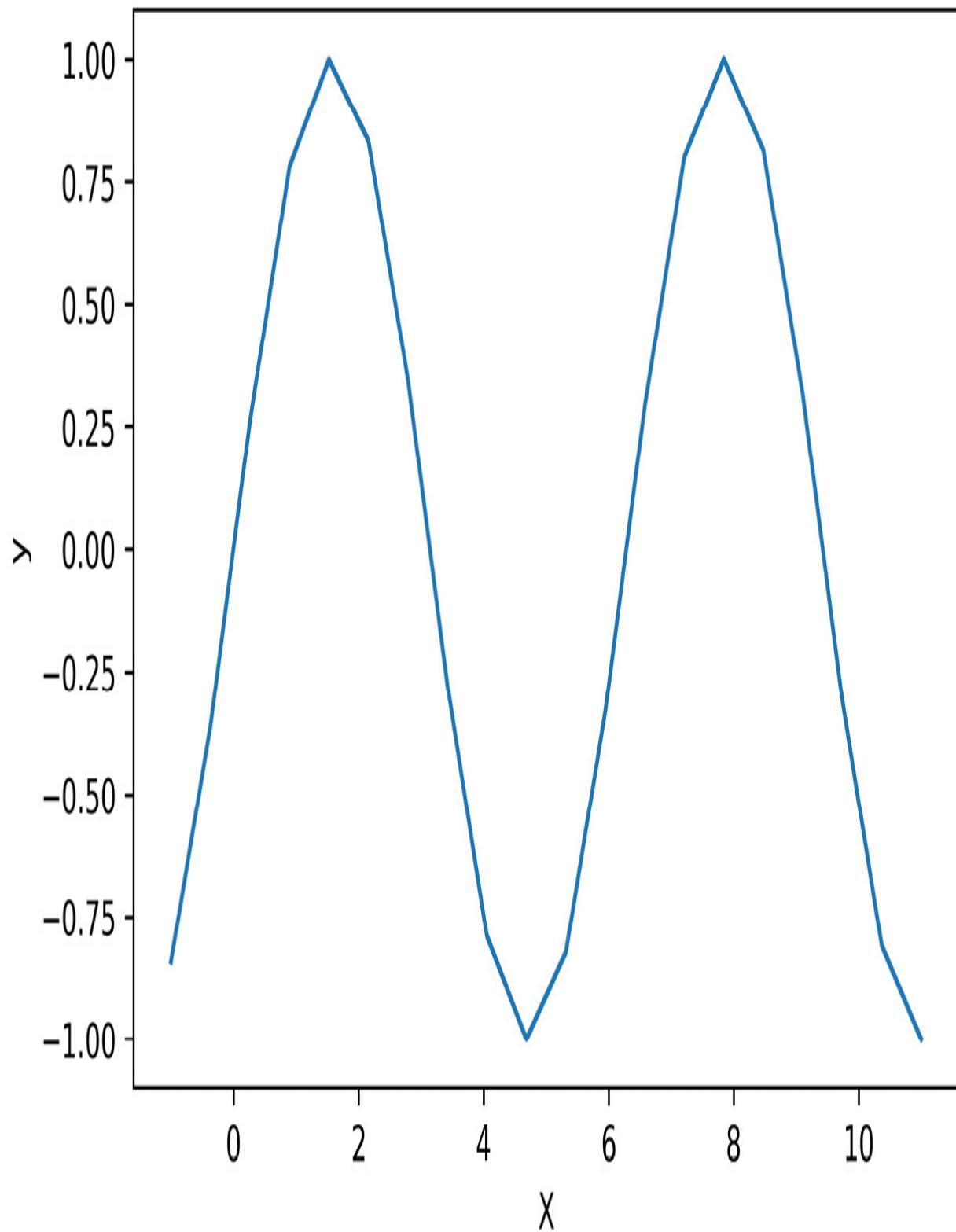


Figure 8.17 – Relationship between the predictor and the target variable

4. Let's fit a linear model to predict **y** from **X** by utilizing **Ridge** regression, and then obtain the predictions of the model:

```
linmod = Ridge(random_state=10)  
linmod.fit(X.reshape(-1, 1), y)  
pred = linmod.predict(X.reshape(-1, 1))
```

5. Now, let's plot the relationship between **X** and **y**, and overlay the predictions:

```
plt.plot(X, y)  
plt.plot(X, pred)  
plt.ylabel("y")  
plt.xlabel("X")  
plt.legend(["y", "linear"], bbox_to_anchor=(1, 1),  
loc="upper left")
```

In the following figure, we can see that the linear model returned a very poor fit of the non-linear relationship between **X** and **y**:

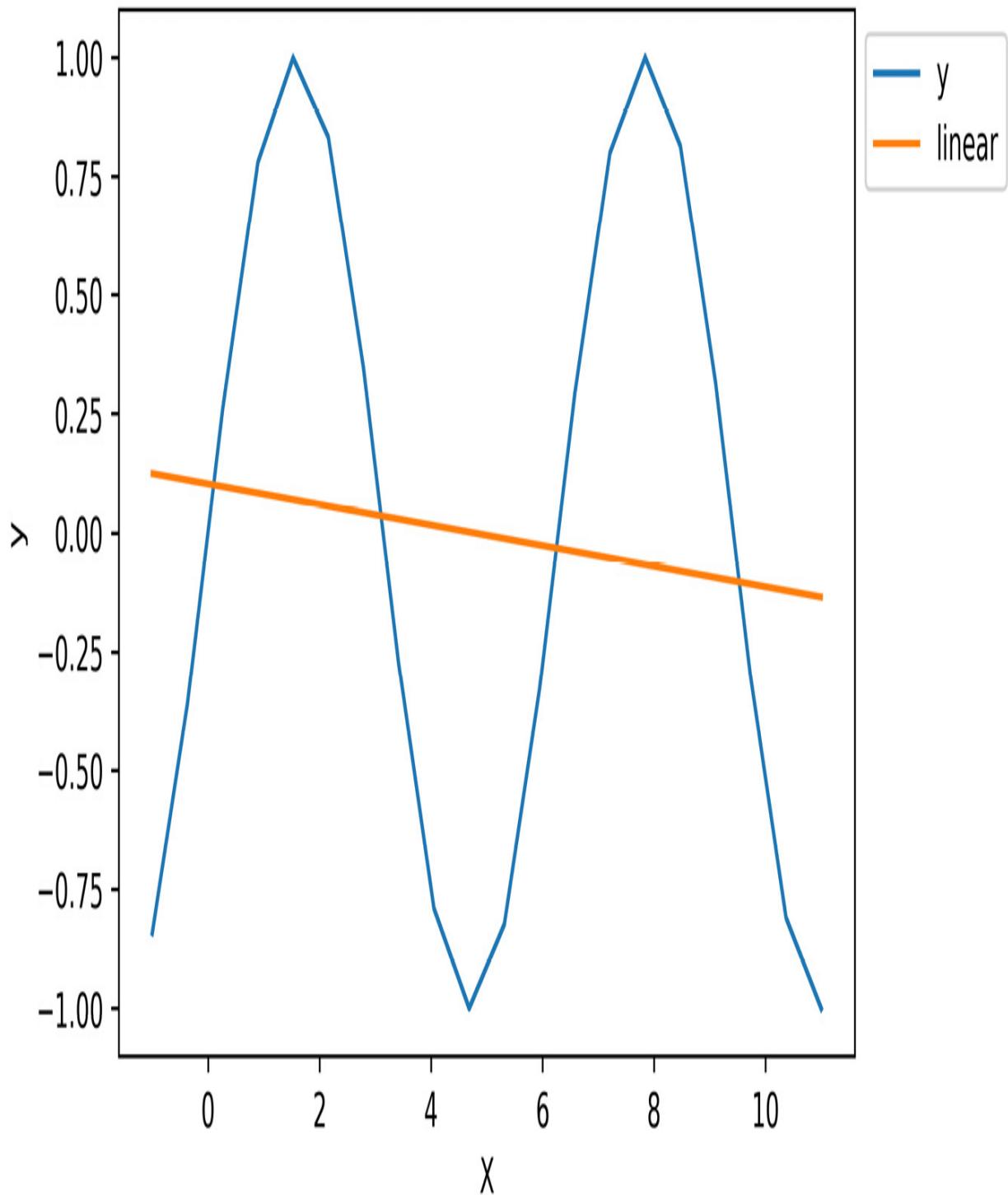


Figure 8.18 – The linear fit between X and y

6. Now, let's set up **SplineTransformer** to obtain spline features from **X** by utilizing third-degree polynomials and five knots at equidistant places within the values of **X**:

```
spl = SplineTransformer(degree=3, n_knots=5)
```

7. Let's obtain the spline features and convert the NumPy array into a pandas DataFrame, adding the names of the spline basis functions:

```
X_t = spl.fit_transform(X.reshape(-1, 1))

X_df = pd.DataFrame(
    X_t, columns = spl.get_feature_names_out(
        ["var"]
    )
)
```

By executing **X_df.head()**, we can see the spline features:

	var_sp_0	var_sp_1	var_sp_2	var_sp_3	var_sp_4	var_sp_5	var_sp_6
0	0.166667	0.666667	0.166667	0.000000	0.0	0.0	0.0
1	0.082009	0.627011	0.289425	0.001555	0.0	0.0	0.0
2	0.032342	0.526705	0.428512	0.012441	0.0	0.0	0.0
3	0.008335	0.393741	0.555936	0.041989	0.0	0.0	0.0
4	0.000656	0.256111	0.643704	0.099529	0.0	0.0	0.0

Figure 8.19 – DataFrame with the splines

NOTE

SplineTransformer returns a new feature matrix of features consisting of **n_splines = n_knots + degree - 1**.

8. Now, let's plot the splines against the values of **X**:

```
plt.plot(X, X_t)

plt.legend(
    spl.get_feature_names_out(["var"]),
    bbox_to_anchor=(1, 1),
    loc="upper left")

plt.xlabel("X")
plt.ylabel("Splines values")
plt.title("Splines")
plt.show()
```

In the following figure, we can see the relationship between the different splines and the values of the predictor variable, **X**:

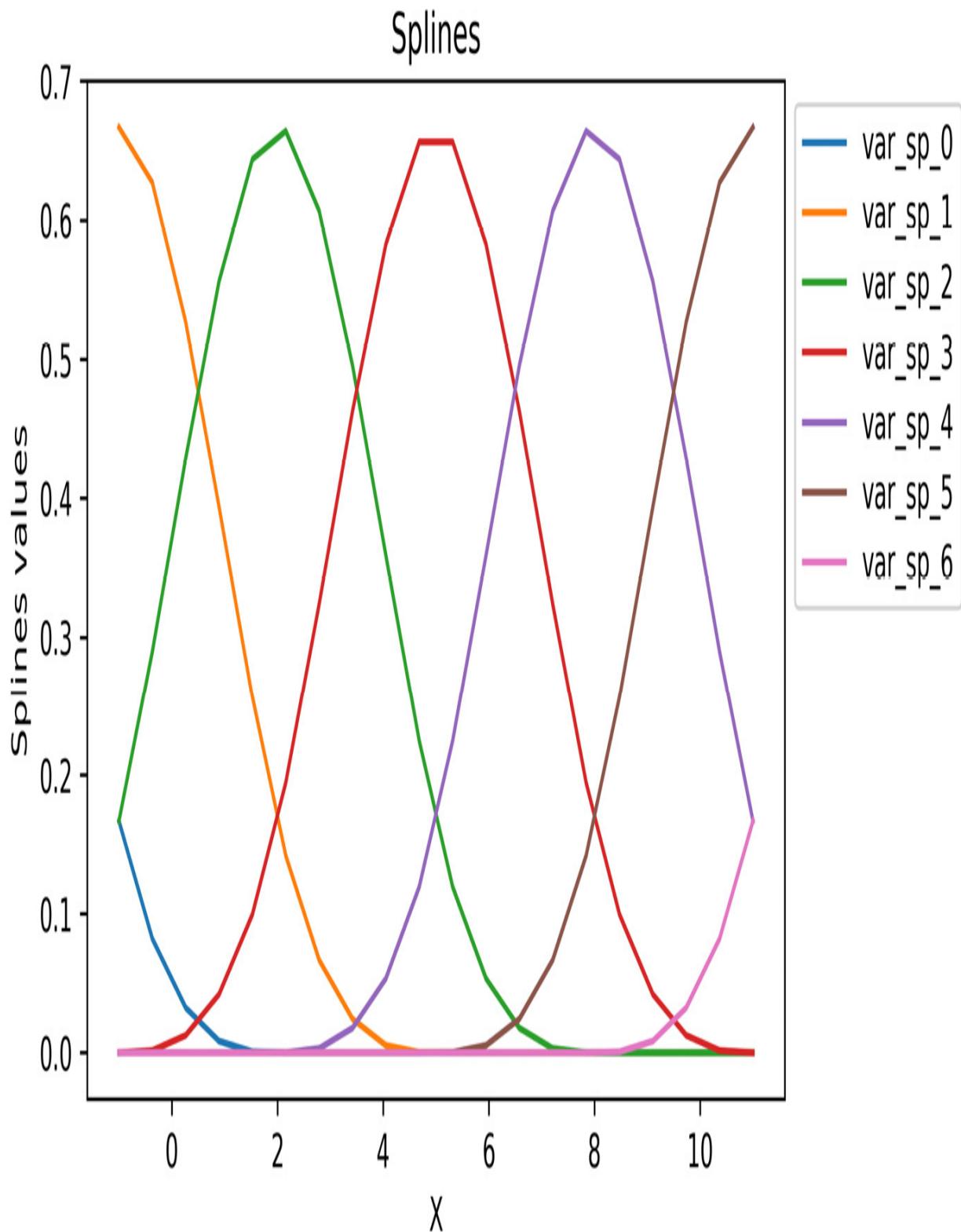


Figure 8.20 – Splines plotted against the values of the predictor variable, X

9. Now, let's fit a linear model to predict **y** from the spline features obtained from **X** by utilizing **Ridge** regression, and then obtain the predictions of the model:

```
linmod = Ridge(random_state=10)  
linmod.fit(X_t, y)  
pred = linmod.predict(X_t)
```

10. Now, let's plot the relationship between **X** and **y**, and overlay the predictions:

```
plt.plot(X, y)  
plt.plot(X, pred)  
plt.ylabel("y")  
plt.xlabel("X")  
plt.legend(["y", "splines"], bbox_to_anchor=(1, 1),  
loc="upper left")
```

In the following figure, we can see that by utilizing spline features as input, **Ridge** regression can better predict the shape of **y**:

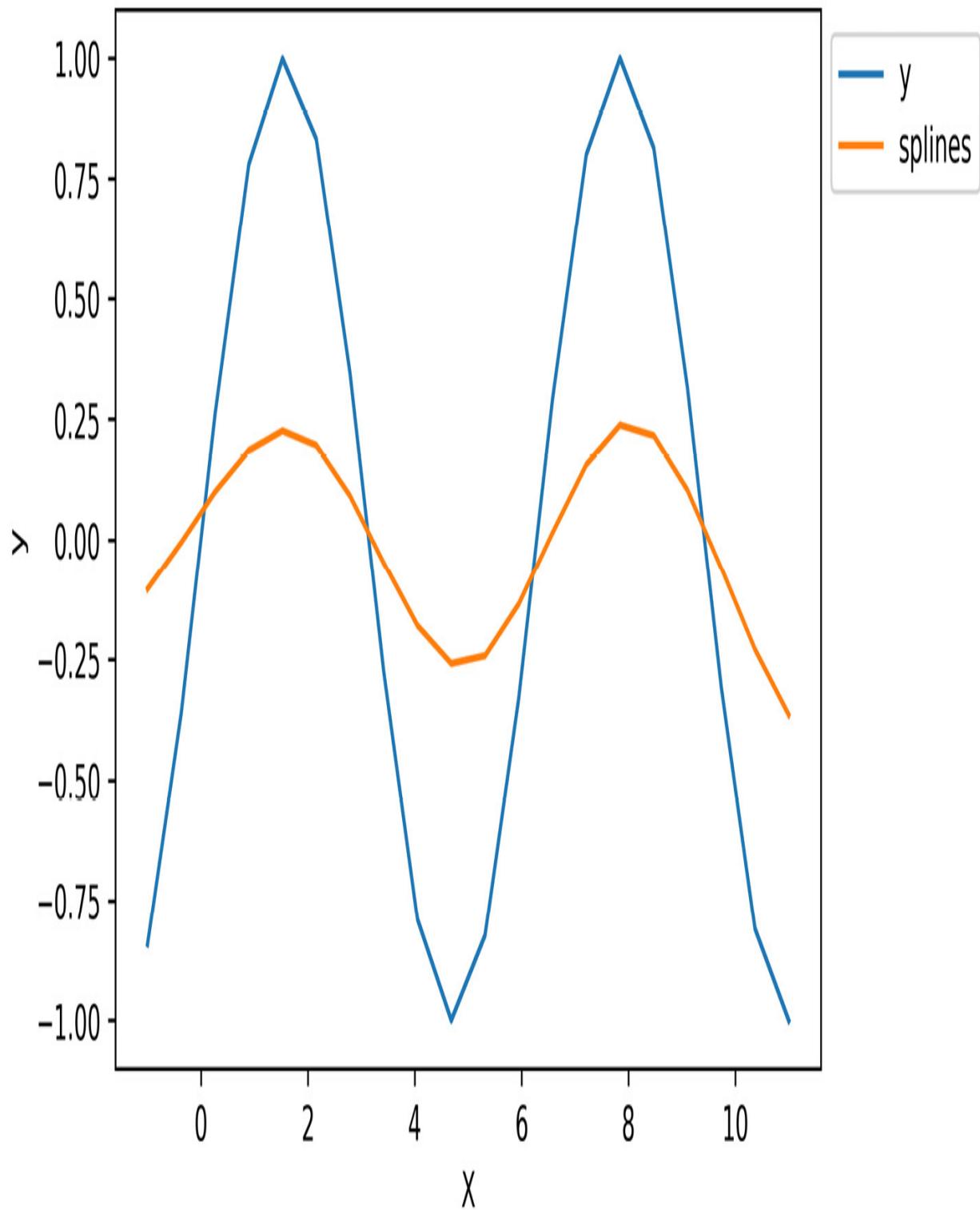


Figure 8.21 – The linear fit between the splines from X and y

TIP

Increasing the number of knots of the degree of the polynomial increases the flexibility of the spline curves. Try creating splines from higher polynomial degrees and see how the predictions of the Ridge regression change.

Now that we understand what the splines features are and how we can use them to predict non-linear effects, let's try them out on a real dataset.

11. Let's import some additional classes and functions from scikit-learn:

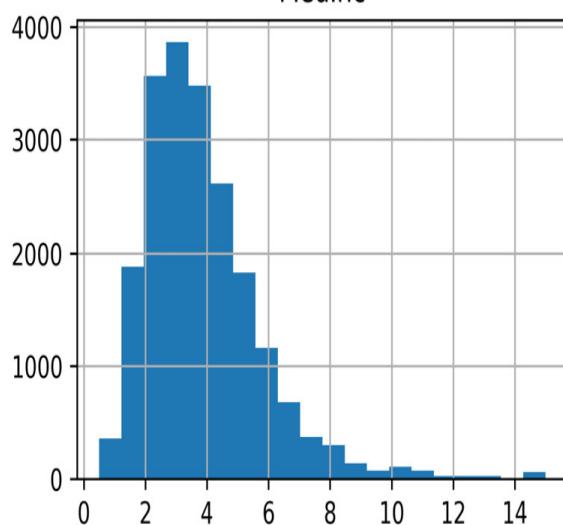
```
from sklearn.datasets import fetch_california_housing  
from sklearn.compose import ColumnTransformer  
from sklearn.model_selection import cross_validate
```

12. Let's load the California Housing dataset, drop two variables that we won't use for modeling, and plot histograms of the remaining predictors:

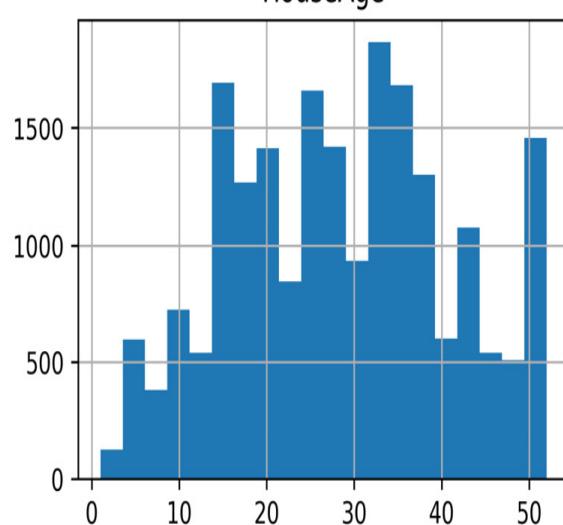
```
x, y = fetch_california_housing(  
    return_X_y=True, as_frame=True)  
x.drop(["Latitude", "Longitude"], axis=1, inplace=True)  
x.hist(bins=20, figsize=(10,10))  
plt.show()
```

In the following figure, we can see the distribution of the predictors:

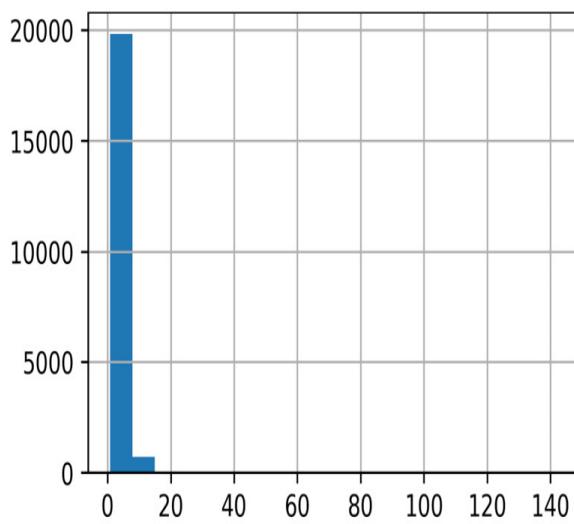
MedInc



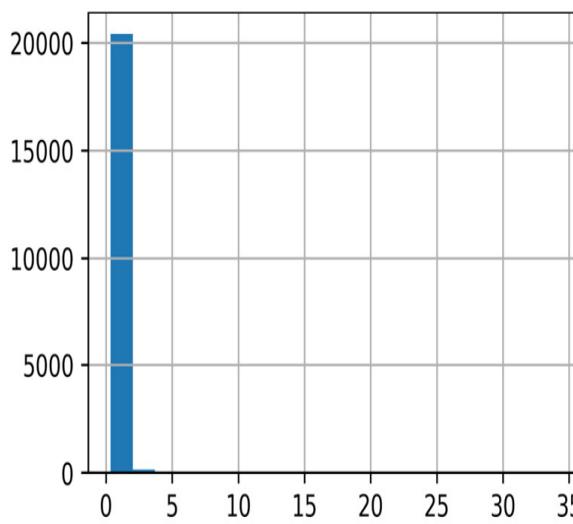
HouseAge



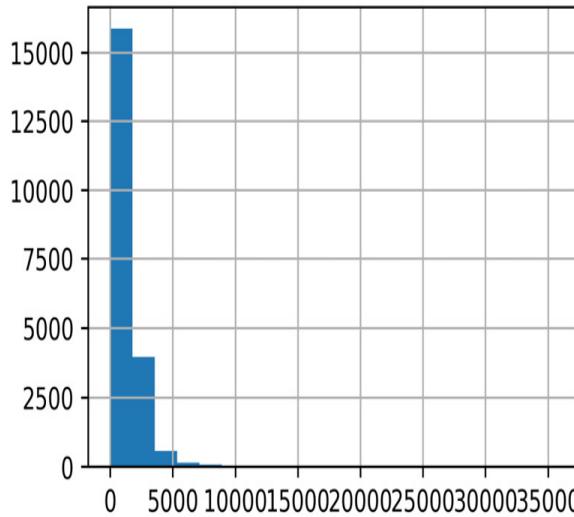
AveRooms



AveBedrms



Population



AveOccup

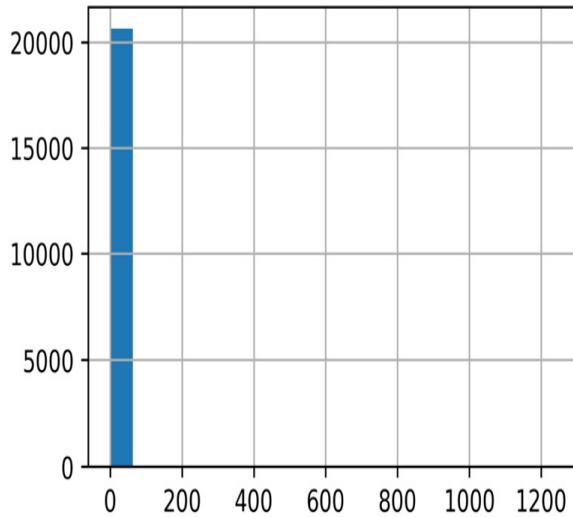


Figure 8.22 – Histograms of the predictor variables

Here, we used splines to model frequencies. The distributions of the last four variables resemble frequencies. So, let's create spline features from them and see whether they improve the performance of a linear model.

13. First, we will fit a **Ridge** regression to predict house prices based on the existing variables by utilizing cross-validation and obtain the performance of the model to set up the benchmark:

```
linmod = Ridge(random_state=10)
cv = cross_validate(linmod, X, y)
mean_, std_ = np.mean(
    cv["test_score"]),
    np.std(cv["test_score"])
print(f"Model score: {mean_} +- {std_}")
```

The previous commands return the following model performance, where the values are R^2 :

```
Model score: 0.490618615960575 +- 0.03617289854929812
```

14. Now, let's set up the transformers to obtain spline features from four variables by utilizing third-degree polynomials and 50 knots, and then fit them to the data:

```
spl = SplineTransformer(degree=3, n_knots=50)
ct = ColumnTransformer(
    [("splines", spl, ["AveRooms", "AveBedrms",
    "Population", "AveOccup"])],
```

```
        remainder="passthrough",  
    )  
    ct.fit(X, y)
```

15. Now, let's fit a **Ridge** regression to predict house prices based on the existing variables plus the polynomial features by using cross-validation, and then obtain the performance of the model:

```
cv = cross_validate(linmod, ct.transform(X), y)  
mean_, std_ = np.mean(  
    cv["test_score"]), np.std(cv["test_score"])  
print(f"Model score: {mean_} +- {std_}")
```

The previous commands return the following model performance, where the values are R^2 :

```
Model score: 0.5553526813919297 +- 0.02244513992785257
```

As we can see, by using splines instead of the original variables, we can improve the performance of the linear regression model.

How it works...

In this recipe, we created new features based on splines. First, we used a toy variable, with values from -1 to 11, and then we obtained splines from a real dataset. The procedure in both cases is identical: we used **SplineTransformer** from scikit-learn. **SplineTransformer** takes the **degree** property of the polynomial and the number of knots (**n_knots**) as input and returns the splines that better fit the data. The knots are placed at

equidistant values of **X** by default, but through the **knots** parameter, we can choose to place them uniformly distributed at the quantiles of **X** instead, or pass an array with the specific values of **X** that should be used as knots.

NOTE

The number, spacing, and position of the knots are arbitrarily set by the user and are the parameters that influence the shape of the splines the most. When using splines in regression models, we can optimize these parameters in a randomized search with cross-validation.

With **fit()**, the transformer computes the knots of the splines. With **transform()**, it returns the array of B-splines. The transformer returns **n_splines=n_knots + degree - 1**.

Finally, we used **ColumnTransformer** to derive splines from a subset of features. **ColumnTransformer** applies **SplineTransformer** to the subset of variables. Because we set **remainder** to "passthrough", **ColumnTransformer** concatenated the features that were not used to obtain splines to the resulting matrix of splines. By doing this, we fitted a **Ridge** regression with the splines plus the **MedInc** and **HouseAge** variables and managed to improve the linear model's performance.

See also

To find out more about the math underlying B-splines, check out the following articles:

- Perperoglou, et al. *A review of spline function procedures in R* (<https://bmcmedresmethodol.biomedcentral.com/articles/10.1186/s1287>

[4-019-0666-3](#)). BMC Med Res Methodol 19, 46 (2019).

- Eilers and Marx. *Flexible Smoothing with B-splines and Penalties* (<https://projecteuclid.org/journals/statistical-science/volume-11/issue-2/Flexible-smoothing-with-B-splines-and-penalties/10.1214/ss/1038425655.full>).
- For an example of how to use B-splines to model time series data, check out the following page in scikit-learn's documentation: https://scikit-learn.org/stable/auto_examples/applications/plot_cyclical_feature_engineering.xhtml#periodic-spline-features.

9

Extracting Features from Relational Data with Featuretools

In previous chapters, we worked with data organized in rows and columns, where the columns are the variables, the rows are the observations, and each observation is independent. In this chapter, we will focus on creating features from relational datasets. In relational datasets, data is structured across various tables, which can be joined together via unique identifiers. These unique identifiers indicate the relationships that exist between the different tables.

A classic example of relational data is that held by retail companies. One table can contain information about customers, such as names and addresses. A second table can contain information about the purchases made by the customers, such as the type and number of items bought per purchase. A third table can contain information about the customers' interactions with the company's website, with variables such as session duration, the mobile device used, and the pages visited. The customers, the purchases, and the sessions are identified with unique identifiers. These unique identifiers allow us to put these tables together, and in this way, we can get information about the customers' purchases or sessions.

If we want to know more about the types of customers we have (that is, customer segmentation) or make predictions about whether they would buy a product, we can create features that aggregate or summarize the

information across the different tables at the customer level. For example, we can create features that capture the maximum amount spent by the customer on a purchase, the number of purchases they have made, the time between sessions, or the average session duration. The number of features we can create and the ways in which we can aggregate this information are plentiful.

In this chapter, we will discuss some common ways of creating aggregated views of relational data utilizing the Python library Featuretools. We will begin by setting up various data tables and their relationships and automatically creating features, and next, we will follow up with more detail on the different features that we can create.

In this chapter, we will cover the following recipes:

- Setting up an entity set and creating features automatically
- Creating features with general and cumulative operations
- Combining numerical features
- Extracting features from date and time
- Extracting features from text
- Creating features with aggregation primitives

Technical requirements

In this chapter, we will use **pandas**, **matplotlib**, and the open source Python library Featuretools. You can install Featuretools with **pip**:

```
pip install featuretools
```

Otherwise, you can do so with **conda**:

```
conda install -c conda-forge featuretools
```

Make sure you have Featuretools version 1.14.0 or greater to run this notebook. The code was tested using versions 1.14.0 and 1.15.0.

NOTE

We will work with the **Online Retail II dataset** from the UCI Machine Learning Repository:

<https://archive.ics.uci.edu/ml/datasets/Online+Retail+II>. Dua, D. and Graff, C. (2019). *UCI Machine Learning Repository* (<http://archive.ics.uci.edu/ml>). Irvine, CA: The University of California, School of Information and Computer Science.

To download the **Online Retail II** dataset, follow these steps:

1. Go to <https://archive.ics.uci.edu/ml/machine-learning-databases/00502/>.
2. Click on **online_retail_II.xlsx** to download the data.
3. Save **online_retail_II.xlsx** to the folder from which you will run the following commands.

After you've downloaded the dataset, open a Jupyter notebook and execute the following:

1. Import **pandas**:

```
import pandas as pd
```

2. The data is spread across two Excel sheets. Load them into a single dataframe:

```
file = "online_retail_II.xlsx"
```

```
df_1 = pd.read_excel(file, sheet_name="Year 2009-2010")
df_2 = pd.read_excel(file, sheet_name="Year 2010-2011")
df = pd.concat([df_1, df_2])
```

3. Step 2 may raise an error if you don't have the **openpyxl** library installed. Go ahead and install it to proceed with the data preparation. Select the data corresponding to the United Kingdom and then drop the **Country** column:

```
df = df[df["Country"]=="United Kingdom"]
df.drop("Country", axis=1, inplace=True)
```

4. Remove any rows without a given **Customer ID**:

```
df.dropna(subset=["Customer ID"], inplace=True)
```

5. Rename the dataframe's columns:

```
df.columns = [
    "invoice",
    "stock_code",
    "description",
    "quantity",
    "invoice_date",
    "price",
    "customer_id",
]
```

6. Reset the index of the dataframe:

```
df.reset_index(inplace=True, drop=True)
```

7. Re-order the columns and save the data:

```
ordered_cols = [  
    "customer_id",  
    "invoice",  
    "invoice_date",  
    "stock_code",  
    "description",  
    "quantity",  
    "price",  
]  
  
df[ordered_cols].to_csv("retail.csv", index=False)
```

Make sure you install Featuretools and download and prepare the dataset before proceeding with this chapter since we will be using both throughout.

Setting up an entity set and creating features automatically

Relational datasets or databases contain data spread across multiple tables and the relationships between tables are dictated by a unique identifier that tells us how we can join those tables. To automate feature creation with Featuretools, first, we need to enter the different data tables and establish their relationships within what is called an **entity set**. The entity set then informs Featuretools how these tables are connected so that the library can automatically create features based on those relationships.

We will work with a dataset containing information about customers, invoices, and products. First, we will set up an entity set highlighting the relationships between these three items. This entity set will be the starting point for the remaining recipes in this chapter. Next, we will create features automatically by aggregating the data at the customer, invoice, and product levels, utilizing the default parameters from Featuretools.

In this recipe, you will learn how to correctly set up an entity set and extract a bunch of features automatically for each entity. In the upcoming recipes, we will delve deeper into the different types of features that we can create with Featuretools.

Getting ready

In this recipe, we will use the **Online Retail II** dataset from the UCI Machine Learning repository. In this table, there are customers, which are businesses that buy in bulk from the retail company. The customers are identified with the unique identifier **customer_id**. Each customer makes one or more purchases, which are flagged by the unique identifier **invoice**, containing the invoice number. In each invoice, there are one or more items that have been bought by the customer. Each item or product sold by the company is also identified with a unique **stock code**.

Thus, the data has the following relations:

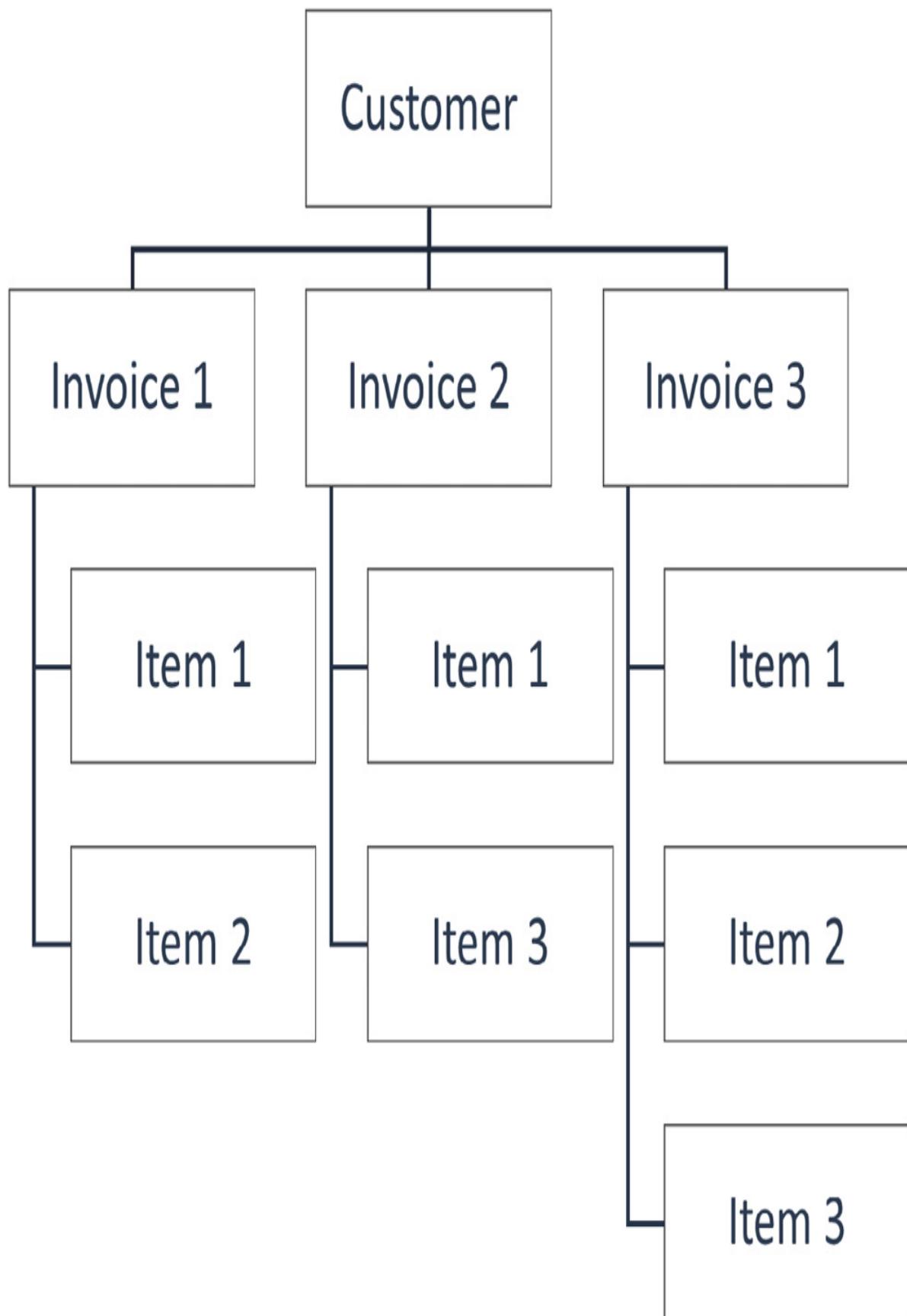


Figure 9.1 – Diagram showing the relationships between the data

Each customer made one or more purchases, identified by the invoice number. Each invoice contains one or more items, identified by the stock code. Each item can be bought by one or more customers and is therefore present in several invoices. With these relationships in mind, let's proceed to the recipe.

How to do it...

In this recipe, we will set up an entity set with the data, and then highlight the different relationships in the dataset. Finally, we will create features by aggregating the information in the dataset at the customer, invoice, and product levels:

1. Let's import the required libraries:

```
import matplotlib.pyplot as plt  
import pandas as pd  
import featuretools as ft  
from woodwork.logical_types import Categorical
```

2. Let's load the dataset that we prepared in the *Technical requirements* section and display the first five rows of it:

```
df = pd.read_csv(  
    "retail.csv", parse_dates=["invoice_date"])  
df.head()
```

In the following screenshot, we can see the unique identifiers for customers (**customer_id**) and invoices (**invoice**), and additional information about the items bought in each invoice, such as the item's code, description, quantity, and unit price, as well as the date of the invoice:

	customer_id	invoice	invoice_date	stock_code	description	quantity	price
0	13085.0	489434	2009-12-01 07:45:00	85048	15CM CHRISTMAS GLASS BALL 20 LIGHTS	12	6.95
1	13085.0	489434	2009-12-01 07:45:00	79323P	PINK CHERRY LIGHTS	12	6.75
2	13085.0	489434	2009-12-01 07:45:00	79323W	WHITE CHERRY LIGHTS	12	6.75
3	13085.0	489434	2009-12-01 07:45:00	22041	RECORD FRAME 7" SINGLE SIZE	48	2.10
4	13085.0	489434	2009-12-01 07:45:00	21232	STRAWBERRY CERAMIC TRINKET BOX	24	1.25

Figure 9.2 – Online Retail dataframe

TIP

Use pandas **unique()** to identify the number of unique items, customers, and invoices – for example, by executing
df["customer_id"].nunique().

3. Let's initialize an entity set with an arbitrary name, such as **data**:

```
es = ft.EntitySet(id="data")
```

4. Let's add the dataframe to the entity set. Note that we give the dataframe a name (**data**), we need to add a unique identifier for each

row, which we call **rows**, and since we do not have a unique row identifier, we will create this additional column when setting the entity set. Finally, we indicate that **invoice_date** is of the datetime type and **customer_id** is **Categorical**:

```
es = es.add_dataframe(  
    dataframe=df,  
    dataframe_name="data",  
    index="rows",  
    make_index=True,  
    time_index="invoice_date",  
    logical_types={"customer_id": Categorical},  
)
```

5. Next, we add the relationship between the original **data** and **invoices**. To do this, we indicate the original or base dataframe, which we called **data** in step 4, we give the new dataframe a name, **invoices**, we add the unique identifier for invoices, and we add the column containing **customer_id** to this dataframe:

```
es.normalize_dataframe(  
    base_dataframe_name="data",  
    new_dataframe_name="invoices",  
    index="invoice",  
    copy_columns=["customer_id"],  
)
```

TIP

We copy the **customer_id** variable to the **invoices** data because we want to create a subsequent relationship between customers and invoices.

6. Now, we add the second relationship, which is between customers and invoices. To do this, we indicate the base dataframe, which we named **invoices** in step 5, we give the new dataframe a name, **customers**, and add the unique customer identifier:

```
es.normalize_dataframe(  
    base_dataframe_name="invoices",  
    new_dataframe_name="customers",  
    index="customer_id",  
)
```

7. We can add a third relationship between the original data and the products:

```
es.normalize_dataframe(  
    base_dataframe_name="data",  
    new_dataframe_name="items",  
    index="stock_code",  
)
```

8. Let's display the information in the entity set:

```
es
```

The entity set contains four dataframes: the original data, the **invoices** dataframe, the **customers** dataframe, and the product or **items** dataframe. The entity also contains the relationships between invoices or items with the original data, as well as between customers and invoices:

Entityset: data

DataFrames:

```
data [Rows: 741301, Columns: 8]  
invoices [Rows: 40505, Columns: 3]  
customers [Rows: 5410, Columns: 2]  
items [Rows: 4631, Columns: 2]
```

Relationships:

```
data.invoice -> invoices.invoice  
invoices.customer_id -> customers.customer_id  
data.stock_code -> items.stock_code
```

9. Let's display the **invoices** dataframe:

```
es["invoices"].head()
```

We see in the following output that Featuretools automatically created a dataframe containing the invoice's unique identifier, followed by the customer's unique identifier and the first date registered for each invoice:

	invoice	customer_id	first_data_time
489434	489434	13085.0	2009-12-01 07:45:00
489435	489435	13085.0	2009-12-01 07:46:00
489436	489436	13078.0	2009-12-01 09:06:00
489437	489437	15362.0	2009-12-01 09:08:00
489438	489438	18102.0	2009-12-01 09:24:00

Figure 9.3 – Dataframe with information at the invoice level

10. Let's now display the **customers** dataframe:

```
es["customers"].head()
```

We can see in the following output that Featuretools automatically created a dataframe containing the customer's unique identifier, followed by the date of the first invoice for this customer:

customer_id	first_data_time
13085.0	13085.0 2009-12-01 07:45:00
13078.0	13078.0 2009-12-01 09:06:00
15362.0	15362.0 2009-12-01 09:08:00
18102.0	18102.0 2009-12-01 09:24:00
18087.0	18087.0 2009-12-01 09:43:00

Figure 9.4 – Dataframe with information at the customer level

TIP

Go ahead and display the dataframe containing the products by executing `es["items"].head()`. You can also evaluate the size of the different dataframes using pandas `shape`. You will notice that the number of rows in each dataframe coincides with the number of unique invoices, customers, and products.

11. We can also display the relationships between these data tables:

```
es.plot()
```

NOTE

To visualize the data relationships, you need to have Graphviz installed. If you don't, follow the instructions in the Featuretools documentation to install it per your operating system:

<https://featuretools.alteryx.com/en/stable/install.xhtml#installing-graphviz>.

In the following output, we can see the relational datasets and their relationships:

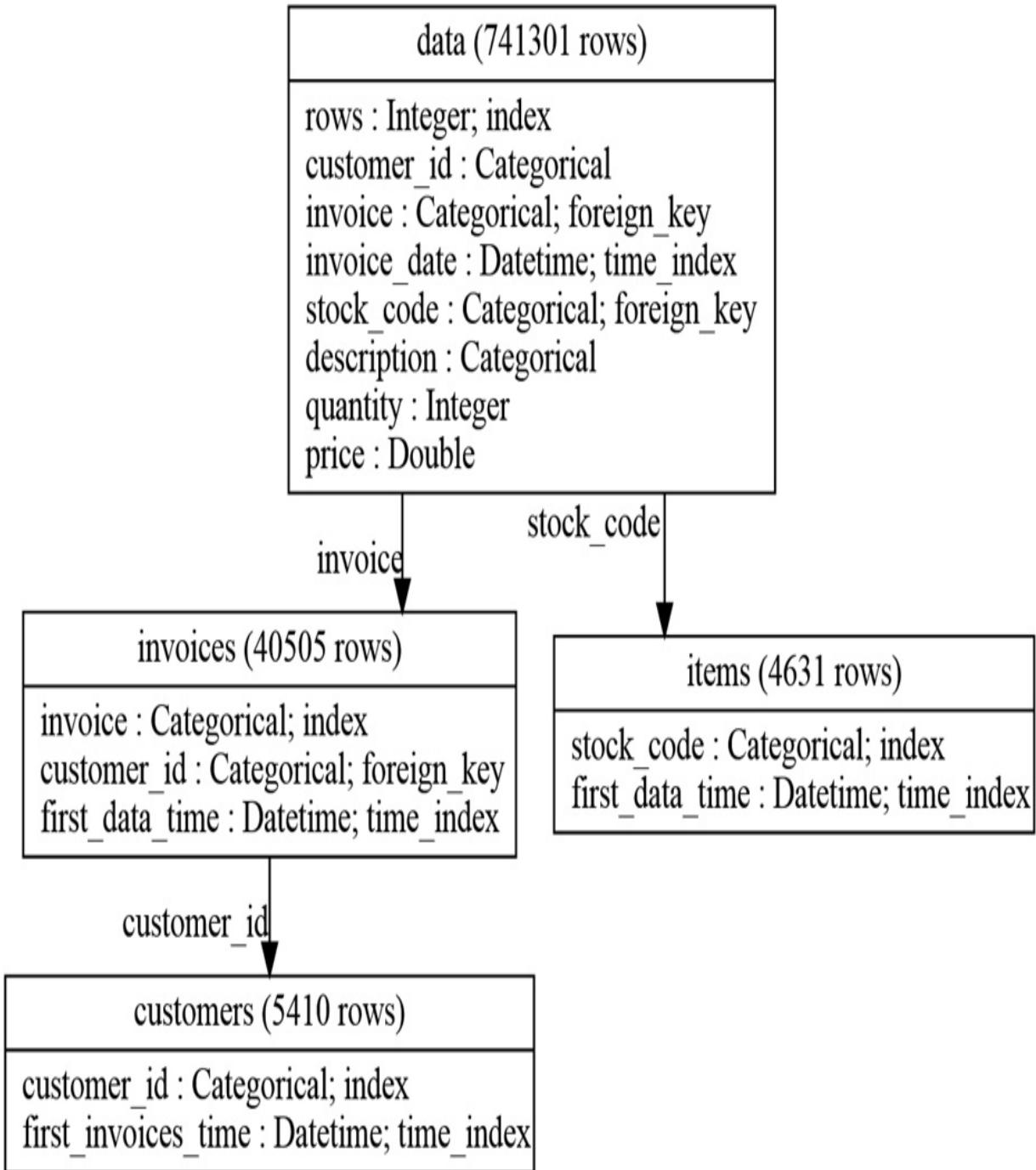


Figure 9.5 – Relationships between invoices, customers, and products within our dataset

After entering the data and their relationships, we can start automatically creating features for each one of our new dataframes, that is, customers, invoices, and products, using the default parameters from Featuretools.

12. Let's create features by aggregating the data at the customer level. To do this, we set up the **deep feature synthesis (dfs)** class from Featuretools, indicating **customers** as the target dataframe. When creating features, we want to ignore the two columns with unique identifiers:

```
feature_matrix, feature_defs = ft.dfs(  
    entityset=es,  
    target_dataframe_name="customers",  
    ignore_columns={  
        "invoices": ["invoice"],  
        "invoices": ["customer_id"],  
    }  
)
```

NOTE

The former block of code triggered the creation of 114 features with different aggregations of data at the customer level. The **feature_matrix** variable is a dataframe with the feature values and **feature_defs** is a list of the names of the new features. Go ahead and execute **feature_defs** or visit our accompanying GitHub repository (<https://github.com/PacktPublishing/Python-Feature-Engineering-Cookbook-Second-Edition/blob/main/ch09-featuretools/Recipe1-Setting-up-an-entity-set.ipynb>) to check the

names of the created features. You will find more details about these features in the *How it works...* section.

13. Let's display the name of five of the created features:

```
feature_defs[5:10]
```

In the following output, we can see 5 of the 114 features created by Featuretools:

```
[<Feature: MIN(data.price)>,
<Feature: MIN(data.quantity)>,
<Feature: MODE(data.description)>,
<Feature: MODE(data.stock_code)>,
<Feature: NUM_UNIQUE(data.description)>]
```

NOTE

Featuretools names the new features with the function used to create them, followed by the dataframe that was used to perform the aggregation, followed by the aggregated variable name. Thus, **MIN(data.quantity)** is equivalent to **df.groupby(["customer_id"])["quantity"].min()**. We will give more details in the *How it works...* section.

14. Let's display the first five rows of the dataframe containing five of the created features:

```
feature_matrix[feature_matrix.columns[5:10]].head()
```

In the following output, we can see the first five rows containing the values of the five new features:

	<code>MIN(data.price)</code>	<code>MIN(data.quantity)</code>		<code>MODE(data.description)</code>	<code>MODE(data.stock_code)</code>	<code>NUM_UNIQUE(data.description)</code>
<u>customer_id</u>						
13085.0	0.55	-48.0	RECORD FRAME 7" SINGLE SIZE	22041	52	
13078.0	0.19	-14.0	AREA PATROLLED METAL SIGN	82582	165	
15362.0	0.21	1.0	BLUE PADDED SOFT MOBILE	20703	38	
18102.0	0.27	-324.0	CREAM HEART CARD HOLDER	22189	415	
18087.0	0.36	-96.0	WHITE HANGING HEART T-LIGHT HOLDER	85123A	48	

Figure 9.6 – Dataframe with five features created by aggregating the data at the customer level

15. Similarly, we can create features automatically by aggregating information at the invoice level:

```
feature_matrix, feature_defs = ft.dfs(
    entityset=es,
    target_dataframe_name="invoices",
    ignore_columns = {"data": ["customer_id"]},
    max_depth = 1,
)
```

16. The previous step returns 24 features – let's display their names:

```
feature_defs
```

We can see the names of the features in the following output:

```
[<Feature: customer_id>,
 <Feature: COUNT(data)>,
 <Feature: MAX(data.price)>,
 <Feature: MAX(data.quantity)>,
 <Feature: MEAN(data.price)>,
 <Feature: MEAN(data.quantity)>,
 <Feature: MIN(data.price)>,
 <Feature: MIN(data.quantity)>,
 <Feature: MODE(data.description)>,
 <Feature: MODE(data.stock_code)>,
 <Feature: NUM_UNIQUE(data.description)>,
 <Feature: NUM_UNIQUE(data.stock_code)>,
 <Feature: SKEW(data.price)>,
 <Feature: SKEW(data.quantity)>,
 <Feature: STD(data.price)>,
 <Feature: STD(data.quantity)>,
 <Feature: SUM(data.price)>,
 <Feature: SUM(data.quantity)>,
 <Feature: DAY(first_data_time)>,
 <Feature: MONTH(first_data_time)>,
 <Feature: WEEKDAY(first_data_time)>,
 <Feature: YEAR(first_data_time)>]
```

TIP

Go ahead and display the dataframe containing the new features by executing `feature_matrix.head()` or check our accompanying GitHub repository for the result.

To wrap up, by using the code from *step 16* and changing the target dataframe name from `invoices` to `items`, go ahead and create features automatically at the product level.

How it works...

In this recipe, we set up an entity set containing the data and the relationships between some of its variables (unique identifiers). After that, we automatically created features by aggregating the information in the dataset for each of the unique identifiers. We used two main classes from Featuretools, `EntitySet` and `dfs`, to create the features. Let's discuss each of these in more detail.

`EntitySet` stores the data, the logical types of the variables, and the relationships between the variables. The variable types (whether numeric or categorical) are automatically assigned by Featuretools. We can also set up specific variable types when adding a dataframe to the entity set. In *step 4*, we added the data to the entity set and set the logical type of `customer_id` to `Categorical`.

`EntitySet` has the `add_dataframe` method, which we used in *step 4* to add a new dataframe. When using this method, we need to specify the unique identifier, and if there is none, then we need to create one, as we did in *step 4*, by setting `make_index` to `True`. Note that in the `index` parameter

from **add_dataframe**, we passed the "**rows**" string. With this configuration, **EntitySet** added the "**rows**" column containing the unique identifier for each row to the dataframe, which is a new sequence of integers starting with **0**.

NOTE

Instead of using the **add_dataframe** method to add a dataframe to an entity set, we can add it by executing **es["df_name"] = df**, where "**df_name**" is the name we want to give to the dataframe and **df** is the dataframe we want to add.

EntitySet has the **normalize_dataframe** method, which is used to create a new dataframe and relationship from unique values of an existing column. The method takes the name of the dataframe to which the new dataframe will be related and a name for the new dataframe. We also need to indicate the unique identifier for the new dataframe in the **index** parameter. By default, this method creates a new dataframe containing the unique identifier, followed by a datetime column containing the first date each unique identifier was registered. We can add more columns to this dataframe by using the **copy_columns** parameters as we did in *step 5*. Adding more columns to the new dataframe is useful if we want to follow up with relationships to this new dataframe, as we did in *step 6*.

EntitySet has also the **plot()** method, which displays the existing relationships in the entity set. In *Figure 9.5*, we saw the relationships between our data tables; the **invoices** and **items** (products) tables were related to the original data, whereas the **customers** table was related to the **invoices** table, which was in turn, related to the original data.

NOTE

The relationship between the tables dictates how the features will be created. The `invoices` and `items` tables are related to the original data. Thus, we can only create features with **depth 1**. The `customers` table is, on the other hand, related to `invoices`, which is related to data. Thus, we can create features with **depth 2**. That means that new features will consist of aggregations from the entire dataset, or aggregations for invoices first, which will then be subsequently aggregated for customers. We can regulate the features to create with the `max_depth` parameter in `dfs`.

After setting up the data and the relationships, we used `dfs` from Featuretools to automatically create features. When creating features with `dfs`, we need to set the target dataframe, that is, the data table for which the features should be created. The `dfs` class creates features by **transforming** and **aggregating** existing variables, through what are called **transform** and **aggregate primitives**.

A transform primitive transforms variables. For example, from datetime variables, using a transform primitive, `dfs` extracts the `month`, `year`, `day`, and `week` values.

An aggregate primitive aggregates information for a unique identifier. Aggregate primitives use mathematical operations such as the mean, standard deviation, maximum and minimum values, the sum, and the skew coefficient for numerical variables. For categorical variables, aggregate primitives use the mode and the count of unique items. For unique identifiers, aggregate primitives count the number of occurrences.

With the functionality of transform and aggregate primitives in mind, let's try to understand the features that we created in this recipe. We used the default parameters of **dfs** to create the default features.

NOTE

For more details on the default features offered by Featuretools, visit <https://featuretools.alteryx.com/en/stable/generated/featuretools.dfs.html#featuretools.dfs>.

We first created features for each customer. Featuretools returned 114 features for each customer. Because the **customers** data is related to the **invoices** data, which is related to the entire dataset, the features were created by aggregating data at two levels. First, the data was aggregated for each customer using the entire dataset. Next, the data was aggregated for each invoice first, and then the pre-aggregated data was aggregated again for each customer.

Featuretools names the new features with the function used to aggregate the data, for example, **COUNT**, **MEAN**, **STD**, and **SKEW**, among others. Next, it uses the data that was used for the aggregation and follows it with the variable that was aggregated. For example, the **MEAN(data.quantity)** feature contains the mean quantity of items bought by the customer calculated from the entire dataset, which is the equivalent of **df.groupby("customer_id")["quantity"].mean()**, if you are familiar with pandas. On the other hand, the **MEAN(invoices.MEAN(data.quantity))** feature first takes the mean quantity of items for each invoice, that is, **df.groupby("invoice")**

`["quantity"].mean()`, and from the resulting series, it takes the mean value, considering the invoices for a particular customer.

For categorical features, Featuretools determines the mode and the unique values. For example, from the **description** variable, we've got the **NUM_UNIQUE(data.description)** and **MODE(data.description)** features. The description is just the name of the item. Thus, these features highlight the number of unique items the customer bought and the item the customer bought the most times.

NOTE

Note something interesting: the **NUM_UNIQUE(data.description)** and **MODE(data.description)** variables are numeric after the aggregation. Thus, Featuretools creates more features by using numerical aggregations of these variables. In this way, the **MAX(invoices.NUM_UNIQUE(data.description))** feature first finds the number of unique items per invoice, and then returns the maximum from those values for a particular customer, considering all the customer's invoices.

From datetime features, Featuretools extracts date components by default. Remember that the **customers** dataframe contains the **customer_id** variable and the date of the first invoice for each customer, as we saw in the output of *step 10*. From this datetime feature, Featuretools created the **DAY(first_invoices_time)**, **MONTH(first_invoices_time)**, **WEEKDAY(first_invoices_time)**, and **YEAR(first_invoices_time)** features containing the different date parts.

Finally, Featuretools also returned the total number of invoices per customer (**COUNT(invoices)**) and the total number of rows

(**COUNT(data)**) per customer.

See also

For more details into what inspired Featuretools, check the original article *Deep Feature Synthesis: Towards Automating Data Science Endeavors* by Kanter and Veeramachanen at

https://www.jmaxkanter.com/papers/DSAA_DSM_2015.pdf.

Creating features with general and cumulative operations

Featuretools uses what are called **transform primitives** to create features. Transform primitives take one or more columns in a dataset as input and return one or more columns as output. They are applied to a single dataframe.

Featuretools divides its transform primitives into various categories depending on the type of operation they perform or the type of variable they modify. For example, **general transform primitives** apply mathematical operations, such as the square root, the sine, and the cosine. **Cumulative transform primitives** create new features by comparing a row's value to the previous row's value. For example, the cumulative sum, cumulative mean, and cumulative minimum and maximum values belong to this category, as well as the difference between row values. There is another cumulative transformation that can be applied to datetime variables, which is the **time since previous**, which determines the time passed between two consecutive timestamps.

In this recipe, we will create features using the general and cumulative transform primitives in Featuretools.

Getting ready

General variable transformations are useful when we want to change the distribution of a variable, as we saw in [Chapter 3, Transforming Numerical Variables](#), or in the *Creating periodic features from cyclical variables* recipe in [Chapter 8, Creating New Features](#). From the transformations described in those chapters, Featuretools supports the square root transformation and the sine and cosine (albeit without the normalization between 0 and 2 π).

With a cumulative transformation, we can, for example, get the total number of items bought per invoice by adding up the item's quantity in each row at the invoice level. To understand the features that we will create in this recipe, let's create them with pandas first:

1. Let's import **pandas**:

```
Import numpy as np  
import pandas as pd
```

2. Let's load the dataset that we prepared in the *Technical requirements* section:

```
df = pd.read_csv(  
    "retail.csv", parse_dates=["invoice_date"])
```

3. Let's capture the two numerical variables, **price** and **quantity**, in a list:

```
numeric_vars = ["quantity", "price"]
```

4. Let's capture the names of the cumulative functions in a list:

```
func = ["cumsum", "cummax", "diff"]
```

5. Let's create a list with new names for the variables that we will create:

```
new_names = [f"{var}_{function}"  
            for function in func for var in numeric_vars]
```

6. Let's create new variables using the cumulative functions from *step 4*, applied to the variables from *step 3*, and add them to the dataframe:

```
df[new_names] = df.groupby("invoice")  
[numeric_vars].agg(func)
```

The previous step returns the cumulative sum or cumulative maximum value, or the difference between rows, within each invoice. As soon as it encounters a new invoice number, it starts afresh. To better understand the result, execute **df[df["invoice"] == "489434"] [numeric_vars + new_names]** and compare the result to that of **df[df["invoice"] == "489435"] [numeric_vars + new_names]**.

Let's now apply the sine and cosine transformation to the entire dataframe.

7. Let's create a list with names for the new variables:

```
new_names = [f"{var}_{function}" for function in  
            ["sin", "cos"] for var in numeric_vars]
```

8. Let's transform the price and quantity with the sine and cosine:

```
df[new_names] = df[numeric_vars].agg([np.sin, np.cos])
```

The transformation in *step 8* was applied to the entire dataset, irrespective of the invoice number. You can inspect the result by executing

```
df[new_names].head().
```

With these feature creation procedures in mind, let's automate the process with Featuretools.

How to do it...

We will apply cumulative transformations within each invoice and general transformations to the entire dataset:

1. First, we'll import `pandas`, `featuretools`, and the `Categorical` logical type:

```
import pandas as pd
import featuretools as ft
from woodwork.logical_types import Categorical
```

2. Let's load the dataset that we prepared in the *Technical requirements* section:

```
df = pd.read_csv(
    "retail.csv", parse_dates=["invoice_date"])
```

3. Let's set up an entity set:

```
es = ft.EntitySet(id="data")
```

4. Let's add the dataframe to the entity set:

```
es = es.add_dataframe(
    dataframe=df,
    dataframe_name="data",
```

```
    index="rows",
    make_index=True,
    time_index="invoice_date",
    logical_types={"customer_id": Categorical},
)
)
```

5. Let's create a new dataframe with a relationship to the dataframe from *step 4*:

```
es.normalize_dataframe(
    base_dataframe_name="data",
    new_dataframe_name="invoices",
    index="invoice",
    copy_columns=["customer_id"],
)
)
```

NOTE

For more details about steps 4 and 5, visit the *Setting up an entity set and creating features automatically* recipe.

6. Let's make a list with the cumulative transformations to perform:

```
cum_primitives = ["cum_sum",
"cum_max", "diff", "time_since_previous"]
```

NOTE

You can find Featuretools-supported cumulative transformations at this link:

https://featuretools.alteryx.com/en/stable/api_reference.xhtml#cumulative-transform-primitives.

7. Let's make a list of the general transformations to carry out:

```
general_primitives = ["sine"]
```

NOTE

You can find Featuretools-supported general transformations at this link:

https://featuretools.alteryx.com/en/stable/api_reference.xhtml#general-transform-primitives.

8. Finally, let's create the features. We use the **dfs** class, setting the original dataframe as the target dataframe, that is, the one whose variables we want to modify. Note that we pass an empty list to the **agg_primitives** parameter; this is to avoid returning the default aggregation primitives. We pass the general primitive from *step 7* to the **trans_primitives** parameter and the cumulative primitives from *step 6* to the **groupby_trans_primitives** parameter:

```
feature_matrix, feature_defs = ft.dfs(  
    entityset=es,  
    target_dataframe_name="data",  
    agg_primitives=[],  
    trans_primitives=general_primitives,  
    groupby_trans_primitives = cum_primitives,  
    ignore_dataframes = ["invoices"],  
)
```

TIP

Featuretools will automatically apply the cumulative primitive transformations after grouping by invoice because of the relationship we set up in step 5.

9. Let's now display the name of the created features:

```
feature_defs
```

In the following output, we can see the name of the features that we created, including the sine and cosine of the price and quantity and the cumulative transformations of these variables after grouping them by invoice number:

```
[<Feature: customer_id>,
 <Feature: invoice>,
 <Feature: stock_code>,
 <Feature: description>,
 <Feature: quantity>,
 <Feature: price>,
 <Feature: SINE(price)>,
 <Feature: SINE(quantity)>,
 <Feature: CUM_MAX(price) by invoice>,
 <Feature: CUM_MAX(quantity) by invoice>,
 <Feature: CUM_SUM(price) by invoice>,
 <Feature: CUM_SUM(quantity) by invoice>,
 <Feature: DIFF(price) by invoice>,
 <Feature: DIFF(quantity) by invoice>,
```

```
<Feature: TIME_SINCE_PREVIOUS(invoice_date) by invoice>]
```

As you can see from the previous list, the new features were appended as new columns to the original dataframe and returned in a new variable. You can display the final dataframe by executing **feature_matrix.head()**, which will return the following dataframe:

	customer_id	invoice	stock_code	description	quantity	price	SINE(price)	SINE(quantity)	CUM_MAX(price) by invoice	CUM_MAX(quantity) by invoice
rows										
0	13085.0	489434	85048	15CM CHRISTMAS GLASS BALL 20 LIGHTS	12	6.95	0.618486	-0.536573	6.95	12.0
1	13085.0	489434	79323P	PINK CHERRY LIGHTS	12	6.75	0.450044	-0.536573	6.95	12.0
2	13085.0	489434	79323W	WHITE CHERRY LIGHTS	12	6.75	0.450044	-0.536573	6.95	12.0
3	13085.0	489434	22041	RECORD FRAME 7" SINGLE SIZE	48	2.10	0.863209	-0.768255	6.95	48.0
4	13085.0	489434	21232	STRAWBERRY CERAMIC TRINKET BOX	24	1.25	0.948985	-0.905578	6.95	48.0
					CUM_SUM(price) by invoice	CUM_SUM(quantity) by invoice	DIFF(price) by invoice	DIFF(quantity) by invoice	TIME_SINCE_PREVIOUS(invoice_date) by invoice	
					6.95	12.0	NaN	NaN	NaN	NaN
					13.70	24.0	-0.20	0.0		0.0
					20.45	36.0	0.00	0.0		0.0
					22.55	84.0	-4.65	36.0		0.0
					23.80	108.0	-0.85	-24.0		0.0

Figure 9.7 – Dataframe with the original and new features

For more details about the created features, check the *How it works...* section.

How it works...

To create features using general and cumulative transformations with Featuretools, first, we need to set up an entity set with the data and define the relationships between its variables. We described how to set up an entity set in the *Setting up and entity set and creating features automatically* recipe.

To apply cumulative and general transforms, we used the **dfs** class from Featuretools. We applied general transformation to the entire dataframe without grouping anything by a specific variable. To do this, we passed a list of strings that identify each transformation to the **trans_primitives** parameter from **dfs**.

We applied cumulative transformation after grouping by invoice. To do this, we passed a list of strings that identify each transformation to the **groupby_trans_primitives** parameter from **dfs**. Featuretools knows it should group by invoice because we established this unique identifier by using the **normalize_dataframe** method from **EntitySet** in step 5.

Finally, we did not want features created from the variables in the **invoices** dataframe; thus, we set **dfs** to ignore this dataframe.

The **dfs** class returned two variables, the dataframe with the original and new features and the name of the features in a list. The new features are named with the operations applied to create them, such as **SINE**, **COSINE**, **CUM_MAX**, or **DIFF**, followed by the variable to which the transformation was applied, and when corresponding, the variable that was used for grouping.

Note that Featuretools automatically recognizes and selects the variables over which the transformations should be applied. The sine, cosine, cumulative sum, maximum, and difference were applied to numerical variables, whereas the **time_since_previous** transformation was applied to the datetime variable.

Combining numerical features

In [*Chapter 8, Creating New Features*](#), we saw that we can create new features by combining variables with mathematical operations. Featuretools supports a number of operations to combine variables, including addition, division, modulo, and multiplication. In this recipe, we will learn how to combine these features with Featuretools.

How to do it...

Let's begin by importing the libraries and getting the dataset ready:

1. First, we'll import **pandas**, **featuretools**, and the **Categorical** logical type:

```
import pandas as pd
```

```
import featuretools as ft
from woodwork.logical_types import Categorical
```

2. Let's load the dataset that we prepared in the *Technical requirements* section:

```
df = pd.read_csv(
    "retail.csv", parse_dates=["invoice_date"])
```

3. Let's set up an entity set:

```
es = ft.EntitySet(id="data")
```

4. Let's add the dataframe to the entity set:

```
es = es.add_dataframe(
    dataframe=df,
    dataframe_name="data",
    index="rows",
    make_index=True,
    time_index="invoice_date",
    logical_types={"customer_id": Categorical},
)
```

5. Let's create a new dataframe with a relationship to the dataframe from *step 4*:

```
es.normalize_dataframe(
    base_dataframe_name="data",
    new_dataframe_name="invoices",
```

```
    index="invoice",
    copy_columns=["customer_id"],
)
```

NOTE

For more details about *steps 4 and 5*, visit the *Setting up an entity set and creating features automatically* recipe.

6. We will multiply the **quantity** and **price** variables, which reflect the number of items bought and the unit price, respectively, to obtain the total amount paid:

```
feature_matrix, feature_defs = ft.dfs(
    entityset=es,
    target_dataframe_name="data",
    agg_primitives=[],
    trans_primitives=["multiply_numeric"],
    primitive_options={
        ("multiply_numeric"): {
            'include_columns': {
                'data': ["quantity", "price"]
            }
        }
    },
    ignore_dataframes=["invoices"],
)
```

7. Let's now display the name of the new features:

```
feature_defs
```

In the following output, we can see the feature names, the last one of which corresponds to the combination of the **price** and **quantity** variables:

```
[<Feature: customer_id>,
<Feature: invoice>,
<Feature: stock_code>,
<Feature: description>,
<Feature: quantity>,
<Feature: price>,
<Feature: price * quantity>]
```

8. To finish off, let's inspect the new dataframe created in *step 6*:

```
feature_matrix.head()
```

In the following output, we can see that the new feature was appended to the end of the original dataframe:

rows	customer_id	invoice	stock_code	description	quantity	price	price * quantity
0	13085.0	489434	85048	15CM CHRISTMAS GLASS BALL 20 LIGHTS	12	6.95	83.4
1	13085.0	489434	79323P	PINK CHERRY LIGHTS	12	6.75	81.0
2	13085.0	489434	79323W	WHITE CHERRY LIGHTS	12	6.75	81.0
3	13085.0	489434	22041	RECORD FRAME 7" SINGLE SIZE	48	2.10	100.8
4	13085.0	489434	21232	STRAWBERRY CERAMIC TRINKET BOX	24	1.25	30.0

Figure 9.8 – Dataframe with the new feature from the combination of price with quantity

Combining features with Featuretools may seem like a lot of work, compared to the `(df["price"].mul(df["quantity"]))` pandas functionality. The real power comes in when we create new features in this way and follow it up with aggregations at the invoice or customer level.

How it works...

To multiply features, we used the **MultiplyNumeric** primitive from Featuretools (<https://primitives.featurelabs.com/#MultiplyNumeric>), which can be accessed from `dfs` using the `multiply_numeric` string. We set the former string to the `trans_primitive` parameter and then used the `primitive_options` parameter to specify which variables to multiply. Note that in addition, we passed an empty list to the `agg_primitives` parameter to override the default aggregation primitives, and we ignored the features coming from the `invoices` dataframe.

Extracting features from date and time

In *Chapter 6, Extracting Features from Date and Time*, we created various features from the date and time parts of datetime variables using pandas and Feature-engine. We can also extract multiple features from datetime variables, automatically utilizing Featuretools.

Featuretools supports the creation of various features from datetime variables using its **datetime transform primitives**. These primitives

include the creation of features such as year, month, and day, and also other features such as **is it lunch time** or **is it weekday**?

NOTE

For more details on the features that can be created using datetime variables, visit

https://featuretools.alteryx.com/en/stable/api_reference.xhtml#datetime-transform-primitives.

In this recipe, we will automatically create multiple features from a datetime variable with Featuretools.

How to do it...

Let's begin by importing the libraries and getting the dataset ready:

1. First, we'll import **pandas**, **featuretools**, and the **Categorical** logical type:

```
import pandas as pd  
import featuretools as ft  
from woodwork.logical_types import Categorical
```

2. Let's load the dataset that we prepared in the *Technical requirements* section:

```
df = pd.read_csv(  
    "retail.csv", parse_dates=["invoice_date"])
```

3. Let's set up an entity set:

```
es = ft.EntitySet(id="data")
```

4. Let's add the dataframe to the entity set:

```
es = es.add_dataframe(  
    dataframe=df,  
    dataframe_name="data",  
    index="rows",  
    make_index=True,  
    time_index="invoice_date",  
    logical_types={"customer_id": Categorical},  
)
```

5. Let's create a new dataframe with a relationship to the dataframe from

step 4:

```
es.normalize_dataframe(  
    base_dataframe_name="data",  
    new_dataframe_name="invoices",  
    index="invoice",  
    copy_columns=["customer_id"],  
)
```

NOTE

For more details about steps 4 and 5, visit the *Setting up an entity set and creating features automatically* recipe.

6. Let's make a list with string names that identify the features we want to create:

```
date_primitives = ["day", "year", "month", "weekday",
                    "days_in_month", "part_of_day",
                    "is_federal_holiday",
                    "hour", "minute"]
```

7. Let's now create date and time features from the invoice date variable:

```
feature_matrix, feature_defs = ft.dfs(
    entityset=es,
    target_dataframe_name="invoices",
    agg_primitives=[],
    trans_primitives=date_primitives,
)
```

NOTE

In *step 3*, we entered the invoice date variable as a time variable. Thus, Featuretools will use this variable to create the date- and time-related features.

8. Let's display the name of the created features:

```
feature_defs
```

In the following output, we can see the name of the original and time features:

```
[<Feature: customer_id>,
<Feature: DAY(first_data_time)>,
<Feature: DAYS_IN_MONTH(first_data_time)>,
```

```
<Feature: HOUR(first_data_time)>,
<Feature: IS_FEDERAL_HOLIDAY(first_data_time)>,
<Feature: MINUTE(first_data_time)>,
<Feature: MONTH(first_data_time)>,
<Feature: PART_OF_DAY(first_data_time)>,
<Feature: WEEKDAY(first_data_time)>,
<Feature: YEAR(first_data_time)>]
```

9. Let's now display the resulting dataframe containing the original and new features:

```
feature_matrix.head()
```

In the following output, we can see the dataframe with the original and time features:

	customer_id	DAY(first_data_time)	DAYS_IN_MONTH(first_data_time)	HOUR(first_data_time)	IS_FEDERAL_HOLIDAY(first_data_time)	
invoice						
489434	13085.0	1	31	7	False	
489435	13085.0	1	31	7	False	
489436	13078.0	1	31	9	False	
489437	15362.0	1	31	9	False	
489438	18102.0	1	31	9	False	
	MINUTE(first_data_time)	MONTH(first_data_time)	PART_OF_DAY(first_data_time)	WEEKDAY(first_data_time)	YEAR(first_data_time)	
	45	12	early morning	1	2009	
	46	12	early morning	1	2009	
	6	12	late morning	1	2009	
	8	12	late morning	1	2009	
	24	12	late morning	1	2009	

Figure 9.9 – Dataframe with the original and time features

Note that some of the created features are numeric, such as **HOUR** or **DAY**, some are Booleans, such as **IS_FEDERAL_HOLIDAY**, and some are categorical, such as **PART_OF_DAY**.

How it works...

To create features from datetime variables, we used **datetime transform primitives** from Featuretools

(https://featuretools.alteryx.com/en/stable/api_reference.xhtml#datetime-transform-primitives). These primitives can be accessed from **dfs** using the strings we specified in *step 6* with the **trans_primitive** parameter. Note that in addition, we passed an empty list to the **agg_primitives** parameter to override the default aggregation primitives, and we ignored the features coming from the **invoices** dataframe.

TIP

We override the aggregation primitives in order to focus on the features we want to discuss in the recipe. However, in practice, when creating features with Featuretools, we will most likely use a combination of transform and aggregate primitives.

There's more...

Featuretools supports additional time-related features, which you can find by carefully inspecting the primitive list at

<https://primitives.featurelabs.com/#MultiplyNumeric>. One example is the **DistanceToHoliday** primitive

(<https://primitives.featurelabs.com/#DistanceToHoliday>), which returns the time between a date and a specified holiday. Let's see how we can use this cool primitive.

First, we need to import pandas and Featuretools and set up the entity set as we did in *steps 1 to 5* in the *How to do it...* section:

1. Let's import the primitive to calculate the distance to a holiday:

```
from featuretools.primitives import DistanceToHoliday
```

2. Let's prepare the primitive by specifying that we are interested in the UK's bank holidays and that in particular, we want to know how long it is until Boxing Day:

```
distance_to_boxing_day = DistanceToHoliday(  
    holiday="Boxing Day", country="UK")
```

3. Let's now add that feature to the dataframe:

```
feature_matrix, feature_defs = ft.dfs(  
    entityset=es,  
    target_dataframe_name="invoices",  
    agg_primitives=[],  
    trans_primitives=[distance_to_boxing_day],  
    verbose=True,  
)
```

4. Let's print the name of the returned features:

```
feature_defs
```

In the following output, we can see the names of the original and new features in the resulting dataframe:

```
[<Feature: customer_id>,  
<Feature: DISTANCE_TO_HOLIDAY(first_data_time,  
holiday=Boxing Day, country=UK)>]
```

TIP

Note that we do not have the **price** or **description** variables, among other things, because we transformed the **invoices** dataset in *step 2* and not the entire dataframe.

5. Finally, let's display the resulting data:

```
feature_matrix.head()
```

We can see the resulting dataframe with the distance to Boxing Day in the following output:

	customer_id	DISTANCE_TO_HOLIDAY(first_data_time, holiday=Boxing Day, country=UK)
invoice		
489434	13085.0	25.0
489435	13085.0	25.0
489436	13078.0	25.0
489437	15362.0	25.0
489438	18102.0	25.0

Figure 9.10 – Dataframe with the distance to Boxing Day

6. To find out which holidays exist in the UK and are supported by Featuretools, we need to import **HolidayUtil**:

```
from featuretools.primitives.utils import HolidayUtil
```

7. Next, we need to specify that we are interested in the UK:

```
holidayUtil = HolidayUtil("UK")
```

8. Finally, we will capture the holidays in a list and then display them:

```
available_holidays = list(  
    set(holidayUtil.federal_holidays.values()))  
  
available_holidays
```

We can see the following holidays for the UK:

```
['Good Friday',  
'Boxing Day',  
"St. Patrick's Day [Northern Ireland]",  
"New Year's Day",  
'Golden Jubilee of Elizabeth II',  
'Millennium Celebrations',  
'Christmas Day',  
'Late Summer Bank Holiday [England/Wales/Northern  
Ireland]',  
'Platinum Jubilee of Elizabeth II',  
'New Year Holiday [Scotland]',  
'Battle of the Boyne [Northern Ireland]',  
'May Day',  
'Boxing Day (Observed)',  
'Silver Jubilee of Elizabeth II',  
'Summer Bank Holiday [Scotland]',  
'Christmas Day (Observed)'
```

'Wedding of Charles and Diana',
'Diamond Jubilee of Elizabeth II',
"St. Patrick's Day [Northern Ireland] (Observed)",
'Easter Monday [England/Wales/Northern Ireland]',
'State Funeral of Queen Elizabeth II',
'New Year Holiday [Scotland] (Observed)',
'Wedding of William and Catherine',
"St. Andrew's Day [Scotland]",
"New Year Holiday [Scotland], New Year's Day
(Observed)",
'Spring Bank Holiday',
"New Year's Day (Observed)"]

With this list in mind, you can pick and choose the holidays that make sense for your business.

Extracting features from text

In [*Chapter 11, Extracting Features from Text Variables*](#), we will discuss various features that we can extract from pieces of text utilizing pandas and scikit-learn. We can also extract multiple features from text automatically by utilizing Featuretools.

Featuretools supports the creation of two basic features from text as part of its default functionality, which are the **number of characters** and the **number of words** in a piece of text. In addition, there is an accompanying

Python library, **Natural Language Processing (NLP) primitives**, which contains a lot more functionality to create more advanced features for use with text. Among these functions, we find primitives for counting the number of stop words and punctuation and calculating the amount of whitespace and uppercase, as well as more complex functions for deriving the diversity score or the polarity score.

NOTE

For a complete list of the functions available, visit

https://featuretools.alteryx.com/en/stable/api_reference.xhtml#natural-language-processing-primitives.

In this recipe, we will automatically create multiple features from a text variable, utilizing Featuretools default functionality first, and then utilizing primitives from the **NLP Primitives** library.

Getting ready

To follow along with this recipe, you need to install the **NLP Primitives** library, which you can do with **pip**:

```
pip install nlp_primitives
```

Otherwise, you can use **conda**:

```
conda install -c conda-forge nlp-primitives
```

NOTE

For more details, visit the **NLP Primitives** GitHub repository:

https://github.com/alteryx/nlp_primitives.

How to do it...

Let's begin by importing the libraries and getting the dataset ready:

1. First, we'll import **pandas**, **featuretools**, and the logical types:

```
import pandas as pd  
  
import featuretools as ft  
  
from woodwork.logical_types import (  
    Categorical,  
    NaturalLanguage,  
)
```

2. Let's load the dataset that we prepared in the *Technical requirements* section:

```
df = pd.read_csv(  
    "retail.csv", parse_dates=["invoice_date"])
```

3. Let's set up an entity set:

```
es = ft.EntitySet(id="data")
```

4. Let's add the dataframe to the entity set, highlighting that the **description** variable is a text variable:

```
es = es.add_dataframe(  
    dataframe=df,  
    dataframe_name="data",  
    index="rows",  
    make_index=True,
```

```
    time_index="invoice_date",  
    logical_types={  
        "customer_id": Categorical,  
        "description    }  
)  
)
```

NOTE

For Featuretools text primitives to work, we need to indicate which variables are text by using the **NaturalLanguage** logical type.

5. Let's create a new dataframe with a relationship to the dataframe from

step 4:

```
es.normalize_dataframe(  
    base_dataframe_name="data",  
    new_dataframe_name="invoices",  
    index="invoice",  
    copy_columns=["customer_id"],  
)
```

NOTE

For more details about steps 4 and 5, visit the *Setting up an entity set and creating features automatically* recipe.

6. Let's make a list with string names that identify the text features we want to create:

```
text_primitives = ["num_words", "num_characters"]
```

7. Let's now extract the text features from the **description** variable:

```
feature_matrix, feature_defs = ft.dfs(  
    entityset=es,  
    target_dataframe_name="data",  
    agg_primitives=[],  
    trans_primitives=text_primitives,  
    ignore_dataframes=["invoices"],  
)
```

8. Let's display the name of the created features:

```
feature_defs
```

In the following output, we can see the name of the original features, followed by those created from the **description** variable:

```
[<Feature: customer_id>,  
<Feature: invoice>,  
<Feature: stock_code>,  
<Feature: quantity>,  
<Feature: price>,  
<Feature: NUM_CHARACTERS(description)>,  
<Feature: NUM_WORDS(description)>]
```

9. Let's now display the resulting dataframe:

```
feature_matrix.head()
```

In the following output, we can see the dataframe with the original data and the features created from the text:

	customer_id	invoice	stock_code	quantity	price	NUM_CHARACTERS(description)	NUM_WORDS(description)
rows							
0	13085.0	489434	85048	12	6.95	35.0	6.0
1	13085.0	489434	79323P	12	6.75	18.0	3.0
2	13085.0	489434	79323W	12	6.75	20.0	4.0
3	13085.0	489434	22041	48	2.10	28.0	6.0
4	13085.0	489434	21232	24	1.25	30.0	4.0

Figure 9.11 – Dataframe with the original data and the date and time features

Note that Featuretools removes the original text variable, **description**, and in its place, it returns the new features.

To follow up, let's create more interesting features. To do this, we need to have the **NLP Primitives** library installed.

10. Let's import a few text primitives:

```
from nlp_primitives import (
    DiversityScore,
    MeanCharactersPerWord,
    PunctuationCount,
)
```

11. Let's now add the text primitives from *step 10* to our list of primitives from *step 6*:

```
text_primitives = text_primitives + [  
    DiversityScore,  
    MeanCharactersPerWord,  
    PunctuationCount,  
]
```

To create the text primitives, you need to execute the code from *step 7* after executing *steps 10* and *11*. After that, go ahead and display the names of the new features by executing **feature_defs**.

12. To finish off the recipe, let's display the new features created after executing the code from *step 7* with the primitives from *step 11*:

```
new_vars = feature_matrix.columns[5:10]  
feature_matrix[new_vars].head()
```

In the following output, we can see the dataframe with new features created from the **description** variable:

DIVERSITY_SCORE(description) MEAN_CHARACTERS_PER_WORD(description)

rows

	DIVERSITY_SCORE(description)	MEAN_CHARACTERS_PER_WORD(description)
0	0.833333	5.000000
1	1.000000	5.333333
2	1.000000	5.666667
3	1.000000	4.600000
4	1.000000	6.750000

NUM_CHARACTERS(description) NUM_WORDS(description) PUNCTUATION_COUNT(description)

35.0	6.0	0
18.0	3.0	0
20.0	4.0	0
28.0	6.0	1
30.0	4.0	0

Figure 9.12 – Dataframe with the new text features

Go ahead and compare the new features with the original variable to understand their output better.

How it works...

To create features from text variables, first, we used the Featuretools built-in functionality for counting the number of words and characters, and then we used additional NLP primitives available in the **NLP Primitives** library.

The default primitives can be accessed from **dfs** with the strings we specified in *step 6* using the **trans_primitives** parameter. The additional primitives need to be imported from **NLP Primitives** and then passed on to the **trans_primitives** parameter from **dfs**. With this, **dfs** is able to tap into the functionality of these primitives to create new features from the text. The **NLP Primitives** library uses the **nltk** Python library under the hood.

Creating features with aggregation primitives

Throughout this chapter, we've created features automatically by transforming existing variables into new features. For example, we extracted date and time parts from datetime variables, we counted the number of words, characters, and punctuation in a text, and we combined

numerical features into new variables. To create these features, we worked with **transform primitives**.

Featuretools also supports **aggregation primitives**. These primitives take related observations as input and return a single value. For example, if we have a numerical variable, **price**, related to an **invoice**, an aggregation primitive would take all the price observations for a single invoice and return a single value, such as the mean price or the sum (that is, the total amount paid), for that particular invoice.

TIP

The Featuretools aggregation functionality is the equivalent of **groupby** in pandas, followed by pandas functions such as **mean**, **sum**, **std**, and **count**, among others.

Some aggregation primitives work with numerical variables, such as the mean, sum, or maximum and minimum values. Other aggregation primitives are specific to categorical variables, such as the number of unique values and the most frequent value (mode).

NOTE

For a complete list of supported aggregation primitives, visit https://featuretools.alteryx.com/en/stable/api_reference.xhtml#aggregation-primitives.

In this recipe, first, we will create multiple features by aggregating existing variables, and then we will combine creating new features with transform primitives and subsequently aggregating them with aggregation primitives.

Getting ready

In this recipe, we will use the **Online Retail II** dataset from the UCI Machine Learning Repository. This dataset has information about items, invoices, and customers. To follow along with this recipe, it is important to understand the nature of and the relationships between these entities and how to correctly set up an entity set with Featuretools, which we described in the *Setting up an entity set and creating features automatically* recipe.

How to do it...

Let's begin by importing the libraries and getting the dataset ready:

1. First, we'll import **pandas**, **featuretools**, and the logical types:

```
import pandas as pd  
  
import featuretools as ft  
  
from woodwork.logical_types import (  
    Categorical, NaturalLanguage)
```

2. Let's load the dataset that we prepared in the *Technical requirements* section:

```
df = pd.read_csv(  
    "retail.csv", parse_dates=["invoice_date"])
```

3. Let's set up an entity set:

```
es = ft.EntitySet(id="data")
```

4. Let's add the dataframe to the entity set, highlighting that the **description** variable is a text variable, the **customer_id** is

Categorical, and the invoice date is a datetime feature:

```
es = es.add_dataframe(  
    dataframe=df,  
    dataframe_name="data",  
    index="rows",  
    make_index=True,  
    time_index="invoice_date",  
    logical_types={  
        "customer_id": Categorical,  
        "description": NaturalLanguage,  
    }  
)
```

5. Let's create a new dataframe with a relationship to the dataframe from *step 4*:

```
es.normalize_dataframe(  
    base_dataframe_name="data",  
    new_dataframe_name="invoices",  
    index="invoice",  
    copy_columns=["customer_id"],  
)
```

6. Now, we add the second relationship, which is between customers and invoices. To do this, we indicate the base dataframe, which we called

invoices in step 5, we give the new dataframe a name, **customers**, and add the unique customer identifier:

```
es.normalize_dataframe(  
    base_dataframe_name="invoices",  
    new_dataframe_name="customers",  
    index="customer_id",  
)
```

NOTE

For more details about steps 4 to 5, visit the *Setting up an entity set and creating features automatically* recipe.

7. Let's make a list with string names that identify the aggregation primitives we want to use:

```
agg_primitives = ["mean", "max", "min", "sum"]
```

8. Let's create features by aggregating the data at the customer level. To do this, we set up the **dfs** class from Featuretools, indicating **customers** as the target dataframe and passing the aggregation primitives from step 7 and an empty list to the **trans_primitives** parameter to prevent **dfs** from returning the default transformations:

```
feature_matrix, feature_defs = ft.dfs(  
    entityset=es,  
    target_dataframe_name="customers",  
    agg_primitives=agg_primitives,  
    trans_primitives=[],
```

)

9. Let's display the name of the created features:

```
feature_defs
```

In the following output, we can see the names of the features that were aggregated at the customer level:

```
[<Feature: MAX(data.price)>,
 <Feature: MAX(data.quantity)>,
 <Feature: MEAN(data.price)>,
 <Feature: MEAN(data.quantity)>,
 <Feature: MIN(data.price)>,
 <Feature: MIN(data.quantity)>,
 <Feature: SUM(data.price)>,
 <Feature: SUM(data.quantity)>,
 <Feature: MAX(invoices.MEAN(data.price))>,
 <Feature: MAX(invoices.MEAN(data.quantity))>,
 <Feature: MAX(invoices.MIN(data.price))>,
 <Feature: MAX(invoices.MIN(data.quantity))>,
 <Feature: MAX(invoices.SUM(data.price))>,
 <Feature: MAX(invoices.SUM(data.quantity))>,
 <Feature: MEAN(invoices.MAX(data.price))>,
 <Feature: MEAN(invoices.MAX(data.quantity))>,
 <Feature: MEAN(invoices.MEAN(data.price))>,
```

```
<Feature: MEAN(invoices.MEAN(data.quantity))>,
<Feature: MEAN(invoices.MIN(data.price))>,
<Feature: MEAN(invoices.MIN(data.quantity))>,
<Feature: MEAN(invoices.SUM(data.price))>,
<Feature: MEAN(invoices.SUM(data.quantity))>,
<Feature: MIN(invoices.MAX(data.price))>,
<Feature: MIN(invoices.MAX(data.quantity))>,
<Feature: MIN(invoices.MEAN(data.price))>,
<Feature: MIN(invoices.MEAN(data.quantity))>,
<Feature: MIN(invoices.SUM(data.price))>,
<Feature: MIN(invoices.SUM(data.quantity))>,
<Feature: SUM(invoices.MAX(data.price))>,
<Feature: SUM(invoices.MAX(data.quantity))>,
<Feature: SUM(invoices.MEAN(data.price))>,
<Feature: SUM(invoices.MEAN(data.quantity))>,
<Feature: SUM(invoices.MIN(data.price))>,
<Feature: SUM(invoices.MIN(data.quantity))>]
```

NOTE

Remember that Featuretools names the features with the function used to create them, followed by the dataframe that was used in the computation, followed by the variable that was used in the computation. Thus, **MAX(data.price)** is the maximum price seen in the dataset for each customer. On the other hand,

MEAN(invoices.MAX(data.price)) is the mean value of all maximum prices observed in each invoice for a particular customer. That is, if a customer has six invoices, first, we find the maximum price for each of the six invoices, and then take the average of those values.

10. Let's now display the resulting dataframe containing the original data and new features:

```
feature_matrix.head()
```

In the following output, we can see *some* of the variables in the dataframe returned by **dfs**:

	$\text{MAX}(\text{data.price})$	$\text{MAX}(\text{data.quantity})$	$\text{MEAN}(\text{data.price})$	$\text{MEAN}(\text{data.quantity})$	$\text{MIN}(\text{data.price})$	$\text{MIN}(\text{data.quantity})$	$\text{SUM}(\text{data.price})$
<i>customer_id</i>							
13085.0	830.12	48.0	12.413587	9.076087	0.55	-48.0	1142.05
13078.0	12.75	300.0	3.961193	14.061988	0.19	-14.0	3386.82
15362.0	9.95	48.0	3.612000	9.200000	0.21	1.0	144.48
18102.0	3580.80	1008.0	10.831367	175.196629	0.27	-324.0	11567.90
18087.0	852.80	3906.0	11.971368	78.189474	0.36	-96.0	1137.28

	$\text{SUM}(\text{data.quantity})$	$\text{MAX}(\text{invoices.MEAN}(\text{data.price}))$	$\text{MAX}(\text{invoices.MEAN}(\text{data.quantity}))$...	$\text{MIN}(\text{invoices.MEAN}(\text{data.price}))$	$\text{MIN}(\text{invoices.MEAN}(\text{data.quantity}))$
<i>customer_id</i>						
835.0	830.120000	20.750000	...		1.828571	-15.428571
12023.0	12.750000	61.333333	...		0.190000	-14.000000
368.0	3.628261	13.117647	...		3.590000	6.304348
187110.0	3580.800000	624.000000	...		0.480000	-324.000000
7428.0	852.800000	3906.000000	...		0.820000	-96.000000

Figure 9.13 – Dataframe with some of the original and new features for each customer

NOTE

Due to space limitations, we can't display the entire output of step 9, so make sure you execute it on your computer or visit our accompanying GitHub repository for more details:

<https://github.com/PacktPublishing/Python-Feature-Engineering->

[Cookbook-Second-Edition/blob/main/ch09-featuretools/Recipe6-Creating-features-with-aggregation-primitives.ipynb](#).

To follow up, let's combine the recipes that we've discussed throughout the chapter with the aggregation functions from this recipe. First, we will create new features from existing variables and then aggregate those features, along with the existing variables, at the customer level:

1. Let's make lists with date and text primitives:

```
trans_primitives = ["month", "weekday", "num_words"]
```

2. Let's make a list with an aggregation primitive:

```
agg_primitives = ["mean"]
```

3. Let's now automatically create features by transforming and aggregating the variables:

```
feature_matrix, feature_defs = ft.dfs(  
    entityset=es,  
    target_dataframe_name="customers",  
    agg_primitives=agg_primitives,  
    trans_primitives=trans_primitives,  
    max_depth=3,  
)
```

The code from *step 13* triggers the creation of the features and their subsequent aggregation at the customer level.

4. Let's display the names of the new features:

```
feature_defs
```

In the following output, we can see the names of the created variables:

```
[<Feature: MEAN(data.price)>,
 <Feature: MEAN(data.quantity)>,
 <Feature: MONTH(first_invoices_time)>,
 <Feature: WEEKDAY(first_invoices_time)>,
 <Feature: MEAN(invoices.MEAN(data.price))>,
 <Feature: MEAN(invoices.MEAN(data.quantity))>,
 <Feature: MEAN(data.NUM_WORDS(description))>,
 <Feature: MEAN(invoices.MEAN(data.NUM_
 WORDS(description))))>] WORDS(description)))>]
```

Note that in our recipes, we keep the creation of features to a minimum due to space limitations, but you can create as many features as you want and enrich your datasets dramatically with the functionality built into Featuretools.

How it works...

In this recipe, we brought together the creation of features using transform primitives, which we discussed throughout the chapter, with the creation of features using aggregation primitives.

To create features with Featuretools automatically, first, we need to enter the data into an entity set and establish the relationships between the data. We discussed how to set up an entity set in the *Setting up an entity set and creating features automatically* recipe.

To aggregate existing features, we used the `dfs` class. We created a list with string names that can be used to access the aggregation primitives from `dfs` using the `agg_primitives` parameter. To aggregate existing variables without creating new features, we passed an empty list to the `trans_primitives` parameter of `dfs`.

The `customers` dataframe is the child of the `invoice` dataframe, which is in turn, the child of the original data. Thus, `dfs` created aggregations from the original data and the pre-aggregated data for each invoice. Thus, the `MEAN(data.price)` feature consists of the mean price for an item bought by a customer calculated from the entire data, whereas `MEAN(invoices.MEAN(data.price))` calculates the mean price per invoice first and then takes the mean of those values for a customer. Thus, if a customer has five invoices, first, Featuretools calculates the mean price paid for each of those invoices and then takes the mean of those values. As such, `MEAN(data.price)` and `MEAN(invoices.MEAN(data.price))` are not the same feature.

NOTE

An aggregate primitive aggregates information for a unique identifier. Aggregate primitives use mathematical operations such as the mean, standard deviation, maximum and minimum values, the sum, and the skew coefficient for numerical variables. For categorical variables, aggregate primitives use the mode and the count of unique items. For unique identifiers, aggregate primitives count the number of occurrences.

Next, we combined the creation of new features from date and text variables with aggregation. First, we defined a list of strings that can be

used to access the transform primitives using the `trans_primitives` parameter of `dfs`. Then, we set up `dfs` with this list and passed a list with a string for an aggregation primitive to the `agg_primitives` parameter.

In the output of *step 13*, we got a list of the new features. From these, we can identify the features created from the first invoice date for each customer, such as `MONTH(first_invoices_time)` and `WEEKDAY(first_invoices_time)`. We can also see features that were aggregated from features created from text, such as `MEAN(data.NUM_WORDS(description))` and `MEAN(invoices.MEAN(data.NUM_WORDS(description)))`. Finally, we can see the aggregations of existing numerical variables, such as `MEAN(data.price)` and `MEAN(invoices.MEAN(data.price))`.

10

Creating Features from a Time Series with tsfresh

Throughout this book, we've discussed feature engineering methods and tools suitable for tabular data and relational datasets. In this chapter, we will focus on time series data. A time series is a sequence of observations taken sequentially over time. Examples of time series are energy generation and demand, temperature, air pollutant concentration, stock prices, and sales revenue. Each of these examples constitutes a variable, and their values change over time.

The availability of cheap sensors that can measure motion, movement, humidity, or temperature, among several other things, has dramatically increased the availability of temporally annotated data. These time series can then be used in classification tasks. For example, based on the electricity fingerprint of a household at a given time interval, we can infer whether a certain appliance was being used. Based on the signal of an ultrasound sensor, we can determine the probability of a failure in a pipeline. And based on the characteristics of a sound wavelength, we can anticipate whether a listener will like the song. We can also use time series in regression tasks. For example, we can use the signal returned by machinery sensors to predict the remaining useful life of the device.

To use time series in traditional supervised machine learning models, each time series needs to be mapped into a well-defined feature vector that

captures the characteristics of the time series. These characteristics, which are based on the distribution of the time series data points, can be captured by simple mathematical operations such as the mean and the standard deviation. We can also describe time series based on their correlation properties, stationarity, entropy, and non-linear time series analysis functions that decompose the time series signal through, for example, Fourier or wavelet transformations.

Creating features from time series can be very time-consuming because we may need to consider the many algorithms that can be used for signal processing and time series analysis to identify and extract meaningful features. The **tsfresh** Python package, which stands for **Time Series FeatuRe Extraction on the Basis of Scalable Hypothesis** tests, accelerates this process by combining 63 time series characterization methods, which compute more than 750 features automatically. The **tsfresh** library also provides a feature selection algorithm that identifies the most predictive features for a given time series. Thus, tsfresh narrows the gap between signal processing experts and machine learning practitioners by providing a tool that automatically applies complex time series methods to extract features that describe the time series.

In this chapter, we will learn how to automatically create hundreds of features from time series data by utilizing tsfresh. Then, we will discuss how to select the most relevant features, how to optimize feature creation, and how to embed the creation of features from time series within a scikit-learn pipeline to classify time series.

In this chapter, we will cover the following recipes:

- Extracting features automatically from a time series
- Creating and selecting features from a time series
- Tailoring feature creation to a different time series
- Creating pre-selected features
- Embedding feature creation into a scikit-learn pipeline

Technical requirements

In this chapter, we will use the open source **tsfresh** Python library. You can install tsfresh with **pip** by executing **pip install tsfresh**.

NOTE

If you have an old Microsoft operating system, you may need to update Microsoft C++ Build Tools to proceed with tsfresh's installation. Follow the steps in this thread to do so:

<https://stackoverflow.com/questions/64261546/how-to-solve-error-microsoft-visual-c-14-0-or-greater-is-required-when-inst>.

Throughout the recipes in this chapter, we will work with the **Occupancy Detection** dataset, available in the UCI Machine Learning Repository:
<http://archive.ics.uci.edu/ml/datasets/Occupancy+Detection+#>.

NOTE

The dataset was described in *Accurate occupancy detection of an office room from light, temperature, humidity, and CO₂ measurements using statistical learning models. Luis M. Candanedo, Veronique Feldheim. Energy and Buildings. Volume 112, 15 January 2016. Pages 28-39.*

To download the **Occupancy Detection** dataset, follow these steps:

1. Go to <http://archive.ics.uci.edu/ml/machine-learning-databases/00357/>.
2. Click on **occupancy_data.zip** to download the data.
3. Unzip the file from *step 2*.
4. Save the **datatraining.txt** file, which is inside the folder from *step 3* to the folder where you will run the commands in the coming paragraph.

After you've downloaded the dataset, open a Jupyter Notebook and perform the following steps:

1. Import **pandas**:

```
import pandas as pd
```

2. Load the data into a DataFrame:

```
df = pd.read_table(  
    "datatraining.txt", sep=",", parse_dates=["date"])
```

3. Select the data between the dates indicated in the following code snippet:

```
df = df[(df["date"] >= "2015-02-04 18:00:00") &  
        (df["date"] < "2015-02-10 09:00:00")]
```

4. Rename the DataFrame's columns:

```
df.columns = [  
    "date",  
    "temperature",
```

```
"humidity",
"light",
"co2",
"humidity_ratio",
"occupancy"

]
```

5. We will add a unique identifier to highlight each hour of data:

```
ls = []
c = 0
for i in range(len(df)):
    if i % 60 == 0:
        c = c+1
    ls.append(c)
df['id']=ls
```

NOTE

In this dataset, there are 135 hours of records at 1-minute intervals. Each hour of records is marked with a unique integer in step 5. Thus, the first hour of records takes the value 1, the records in the second hour take the value 2, and so on.

6. Re-order the columns and save the data:

```
df[["id"]+columns[0:-1]].to_csv("occupancy.csv",
index=False)
```

7. The **occupancy** variable takes a value of 1 when someone was in the office at any given minute. Let's create a new target variable that indicates that the office was occupied if people were in the office for more than 30 minutes:

```
occ = (df.groupby(  
    "id") ["occupancy"].mean()>0.5).astype (int)
```

8. Save the target variable:

```
occ.to_csv("occupancy_target.csv", index=True)
```

The Occupancy Detection dataset contains time series data taken at 1-minute intervals. These variables measure the temperature, humidity, CO₂ level, and light consumption within an office. The dataset also contains a target variable, indicating whether someone was in the office at any given minute. We modified this target in *step 7*, to indicate whether the office was occupied for more than 30 minutes at any given hour. We also added an additional variable in *step 5*, with a unique identifier for each hour in the data.

You probably noticed that the dataset and the target variable have different numbers of rows. The dataset contains 135 hours of records at 1-minute intervals – that is, 8,100 rows. The target has only 135 rows, with a label indicating whether the office was occupied in each of the 135 hours.

TIP

Check out the notebook in this book's GitHub repository for plots of the different time series in this dataset:

<https://github.com/PacktPublishing/Python-Feature-Engineering->

[Cookbook-Second-Edition/blob/main/ch10-tsfresh/prepare-occupancy-dataset.ipynb](#).

Extracting features automatically from a time series

Time series are data points indexed in time order. Based on a time series sequence, we can predict several things. For example, we can predict whether a pipeline will fail, a music genre, whether a person is sick, or, as we will do in this recipe, whether an office is occupied.

To train regression or classification models with traditional machine learning algorithms, we require a dataset of size $M \times N$, where M is the number of rows and N is the number of features or columns. However, with time series data, what we have is a collection of M time series. And each time series has multiple rows indexed in time. To use time series in supervised learning models, each time series needs to be mapped into a well-defined feature vector, N , as shown in the following diagram:

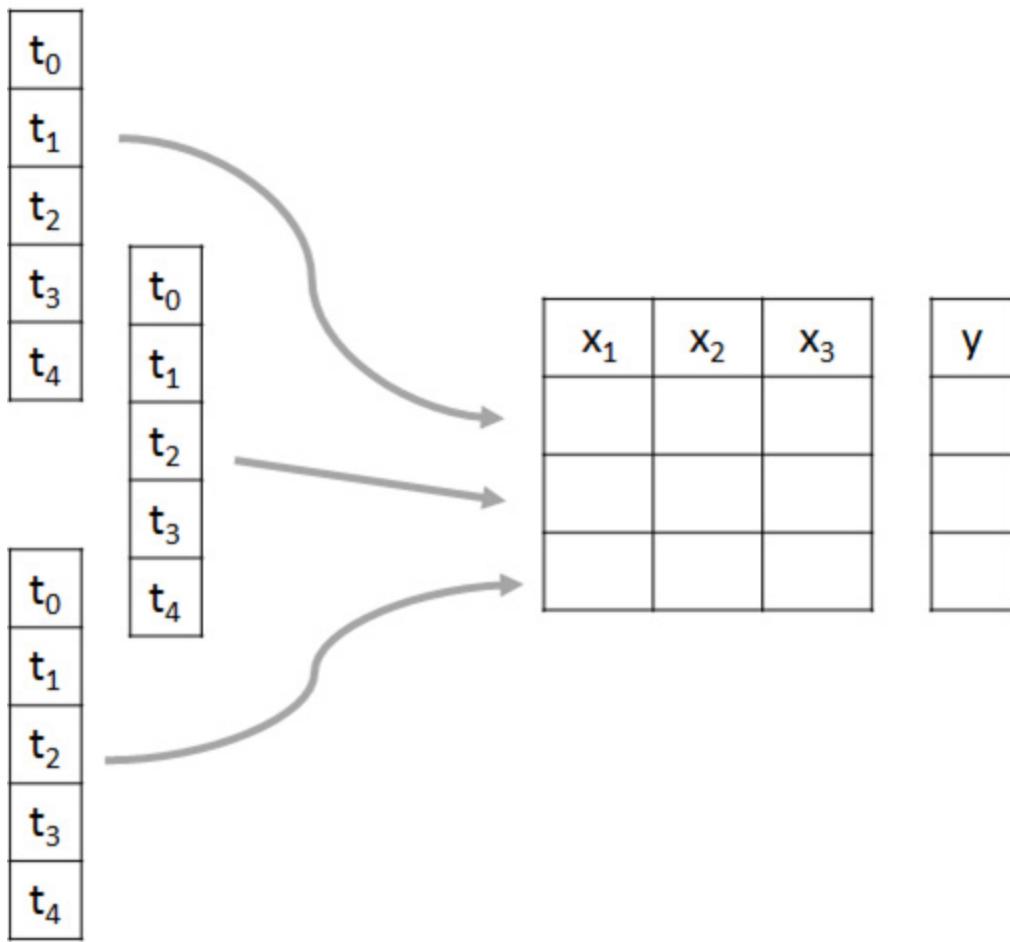


Figure 10.1 – Diagram showing the process of feature creation from a time series for classification or regression

These feature vectors, which are represented as x_1 , x_2 , and x_3 in *Figure 10.1*, should capture the characteristics of the time series. For example, x_1 could be the mean value of the time series and x_2 its variance. We can create many features to characterize the time series concerning the distribution of data points, correlation properties, stationarity, or entropy, among others. Therefore, the feature vector, N , can be constructed by applying a series of **characterization methods** that take a time series as input and return one or more scalars as output. The mean (or the sum operations) takes the time series sequence and returns a single scalar, with

the mean value of the time series or the sum of its values. We can also fit a linear trend to the time series sequence, which will return two scalars – one with the slope and one with the intercept.

The tsfresh library applies 63 characterization methods to a time series, each of which returns one or more scalars, therefore resulting in more than 750 features for any given time series. In this recipe, we will use tsfresh to transform time series data into an M x N feature table, which we will then use to predict office occupancy.

Getting ready

In this recipe, we will use the **Occupancy Detection** dataset that we prepared in the *Technical requirements* section. This dataset contains time series data taken at 1-minute intervals. The variables measure the temperature, humidity, CO₂ level, and light consumption within an office. There are 135 hours of measurements, and each hour is flagged with a unique identifier. We also created a target variable that indicates in which of these 135 hours the office was occupied. Let's load the data and make some plots to understand its structure:

1. Let's load **pandas** and **matplotlib**:

```
import matplotlib.pyplot as plt  
import pandas as pd
```

2. Let's load the dataset and display the first five rows:

```
x = pd.read_csv("occupancy.csv", parse_dates=["date"])  
x.head()
```

In the following screenshot, you can see the dataset, which contains a unique identifier for the first hour of records, followed by the date and time of the measurements, and values for five time series capturing temperature, humidity, lights, and CO₂ levels in the office:

	id	date	temperature	humidity	light	co2	humidity_ratio
0	1	2015-02-04 18:00:00	23.075	27.175000	419.0	688.00	0.004745
1	1	2015-02-04 18:01:00	23.075	27.150000	419.0	690.25	0.004741
2	1	2015-02-04 18:02:00	23.100	27.100000	419.0	691.00	0.004739
3	1	2015-02-04 18:03:00	23.100	27.166667	419.0	683.50	0.004751
4	1	2015-02-04 18:04:00	23.050	27.150000	419.0	687.50	0.004734

Figure 10.2 – DataFrame with the time series data

3. Let's create a function to plot the time series from *step 2* for a given hour:

```
def plot_timeseries(n_id):  
    fig, axes = plt.subplots(nrows=2, ncols=3,  
                           figsize=(20, 10))  
    X[X["id"] == n_id]["temperature"].plot(ax=axes[0, 0],  
                                             title="temperature")  
    X[X["id"] == n_id]["humidity"].plot(ax=axes[0, 1],  
                                         title="humidity")  
    X[X["id"] == n_id]["light"].plot(ax=axes[0, 2],  
                                      title="light")  
    X[X["id"] == n_id]["co2"].plot(ax=axes[1, 0],  
                                   title="CO2")  
    X[X["id"] == n_id]["humidity_ratio"].plot(ax=axes[1, 1],  
                                              title="humidity_ratio")  
    X[X["id"] == n_id]["date"].plot(ax=axes[1, 2],  
                                    title="date")
```

```
X[X["id"] == n_id]["light"].plot(ax=axes[0, 2],  
    title="light")  
  
X[X["id"] == n_id]["co2"].plot(ax=axes[1, 0],  
    title="co2")  
  
X[X["id"] == n_id]  
["humidity_ratio"].plot(ax=axes[1,  
    1], title="humidity_ratio")  
  
plt.show()
```

4. Let's plot the time series corresponding to an hour when the office was not occupied:

```
plot_timeseries(2)
```

In the following screenshot, we can see the time series values during the second hour of records, when the office was empty:

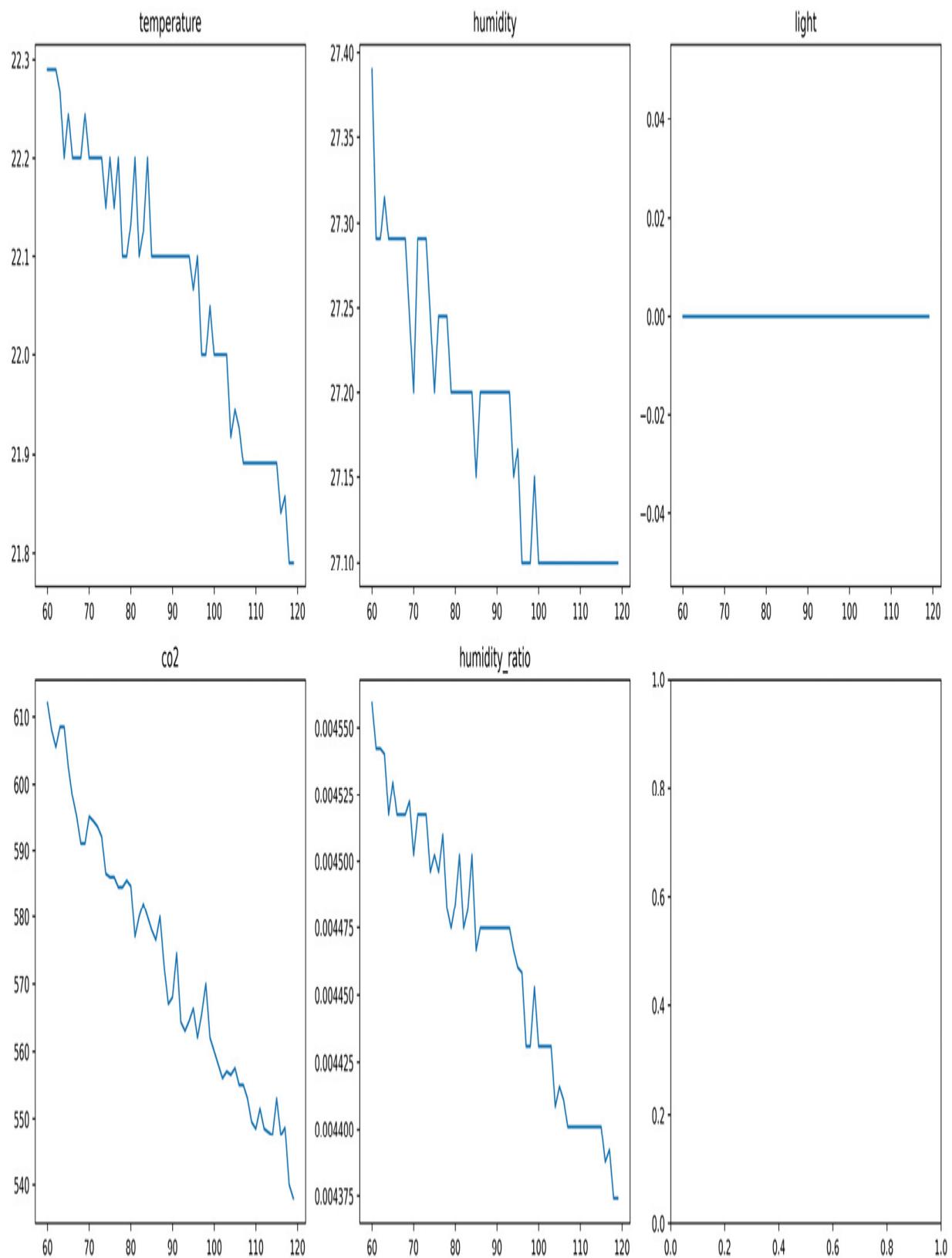


Figure 10.3 – Time series values during the second hour of data

Note that the lights were off, and that is why we can see the flat line at 0 in the plot of **light** consumption in the top-right corner.

5. Now, let's plot the time series data corresponding to an hour when the office was occupied:

```
plot_timeseries(15)
```

In the following screenshot, we can see the time series values during the fifteenth hour of records, when the office was occupied:

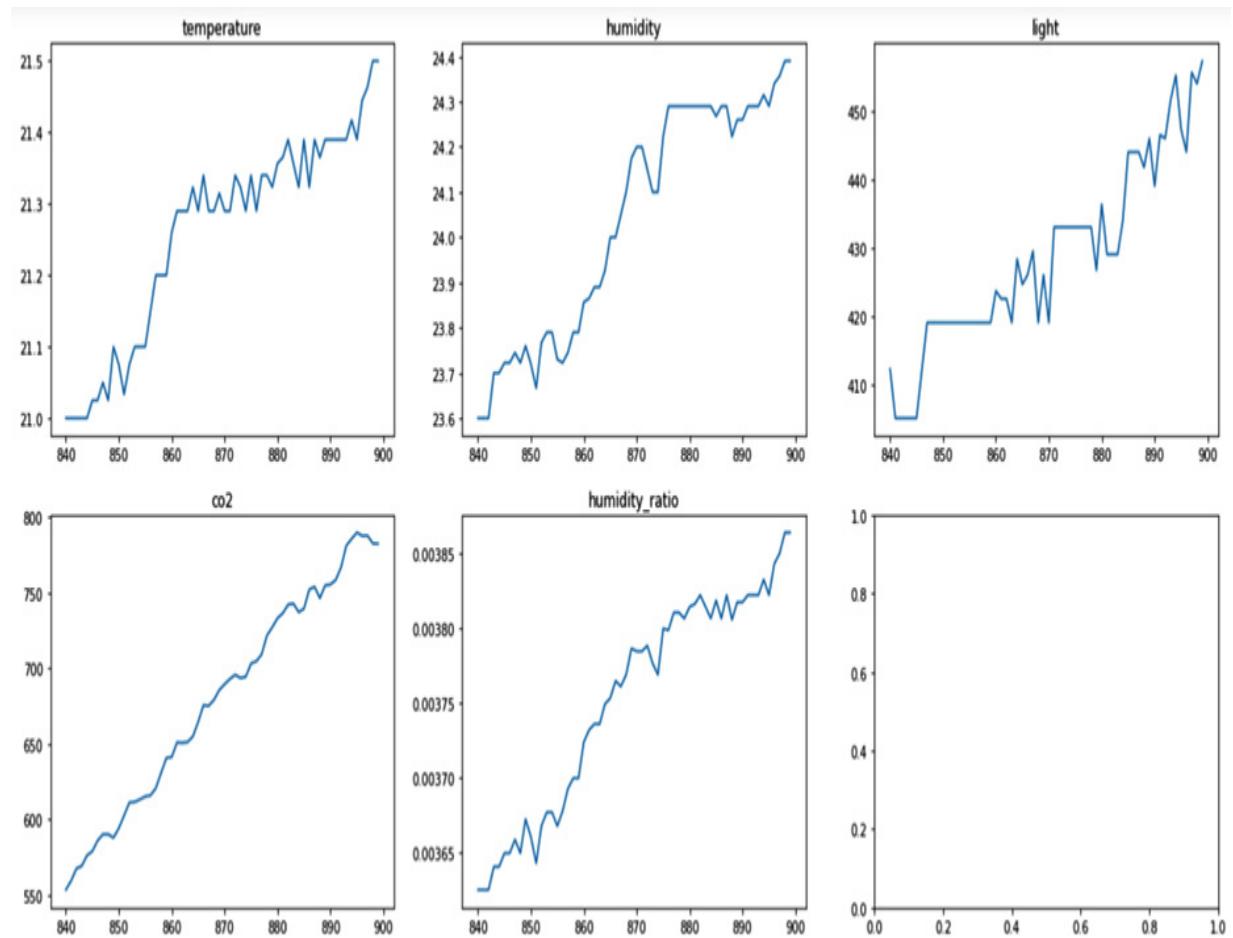


Figure 10.3 – Time series values during the fifteenth hour of data

Notice that the lights were on this time.

I hope these plots make it easier to understand what we will do in the recipe. From each of these 60-minute time series, we will create more than 750 features that capture their characteristics. We will create these features automatically, using tsfresh.

How to do it...

We will begin by automatically creating hundreds of features from one time series, lights, and then use those features to predict whether the office was occupied at any given hour:

1. Let's import the required Python libraries and functions:

```
import pandas as pd

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import classification_report

from sklearn.model_selection import train_test_split

from tsfresh import extract_features

from tsfresh.utilities.dataframe_functions import impute
```

2. Let's load the dataset that we prepared in the *Technical requirements* section:

```
x = pd.read_csv("occupancy.csv", parse_dates=["date"])
```

3. Let's load the target variable and then convert it into a pandas series:

```
y = pd.read_csv("occupancy_target.csv", index_col="id")

y = pd.Series(y["occupancy"])
```

NOTE

We convert the target into a pandas series because tsfresh only accepts pandas series or NumPy arrays as target variables.

4. Let's create hundreds of features automatically for each hour of records using tsfresh, and then display the shape of the resulting matrix. To create features from the **light** variable, we need to pass the DataFrame containing this variable and the unique identifier for each time series:

```
features = extract_features(  
    X[["id", "light"]], column_id="id")  
  
features.shape
```

The output of the previous step is **(135, 789)**. Each row represents an hour of records. The tsfresh library created **789** features that characterize light consumption at any given hour. Go ahead and execute **features.head()** to get a view of the resulting DataFrame. For space reasons, we can't display the entire DataFrame in the book. So, instead, we will explore some of the features.

5. Let's display five of the created features:

```
feats = features.columns[10:15]  
  
feats
```

In the output of *step 5*, we can see the names of five features that were created by simple mathematical operations such as the mean, variance, or standard deviation of the light consumption at any given hour, the length of the time series, or the coefficient of variation:

```
Index(['light_mean', 'light_length',
       'light_standard_deviation',
       'light_variation_coefficient', 'light_variance'],
      dtype='object')
```

6. Now, let's display the values of the features from *step 5* for the first 5 hours of records:

```
features[feats].head()
```

We can see the new features that were created for the first 5 hours of light consumption in the following DataFrame:

	light_mean	light_length	light_standard_deviation	light_variation_coefficient	light_variance
1	48.875	60.0	134.485582	2.751623	18086.371875
2	0.000	60.0	0.000000	NaN	0.000000
3	0.000	60.0	0.000000	NaN	0.000000
4	0.000	60.0	0.000000	NaN	0.000000
5	0.000	60.0	0.000000	NaN	0.000000

Figure 10.4 – Features created for each hour of light consumption

In *Figure 10.4*, we can see that the lights were on in the first hour of records, and then off in the following 4 hours, by inspecting the mean value of light consumption. The length of the time series is 60 because we have 60 minutes of records – one for each hour.

NOTE

The tsfresh library applies 63 feature creation methods to a time series. Based on the characteristics of the time series, such as its length or its variability, some of the methods will return missing values or infinite values. For example, in *Figure 10.4*, we can see that the **variation coefficient** could not be calculated for those hours where the light consumption is constant. And the variance is also 0 in those cases. In fact, for our particular dataset, many of the resulting features contain only NAN values and are therefore unsuitable for training machine learning models.

7. The tsfresh library offers an imputation function to quickly impute features, should they have NAN values. Let's go ahead and impute our features:

```
impute(features)
```

8. Now, let's use these features to train a logistic regression model and predict whether the office was occupied. Let's begin by separating the dataset into train and test sets:

```
x_train, x_test, y_train, y_test = train_test_split(  
    features,  
    y,  
    test_size=0.1,  
    random_state=42,  
)
```

9. Now, let's set up and train a logistic regression model, and then evaluate its performance:

```
cls = LogisticRegression(random_state=10, C=0.01)
```

```

cls.fit(X_train, y_train)

print(classification_report(y_test,
cls.predict(X_test)))

```

In the following output, we can see the evaluation metrics that are commonly used for classification analysis, which suggests that the created features are useful for predicting office occupancy:

	precision	recall	f1-score	support
0	1.00	0.91	0.95	11
1	0.75	1.00	0.86	3
accuracy			0.93	14
macro avg	0.88	0.95	0.90	14
weighted avg	0.95	0.93	0.93	14

10. To finish off, let's create features automatically for each of the time series in our dataset and then display the shape of the resulting DataFrame. This time, we will perform the imputation automatically after extracting the features:

```

features = extract_features(
    X,
    column_id="id",
    impute_function=impute,
    column_sort="date",
)
features.shape

```

NOTE

In *step 10*, we indicated that we want to sort our time series based on the timestamp containing the time and date of the measurement, by passing the `datetime` variable to the `column_sort` parameter. This is useful if our time series are not equidistant or not ordered in time. If we leave this parameter set to `None`, tsfresh assumes that the time series are ordered and equidistant.

The output of *step 10* consists of a DataFrame with 135 rows, containing 3,945 features that characterize the five original time series – temperature, light, humidity and its ratio, and CO₂ in the office. These features were imputed in *step 10*, so you can go ahead and use these DataFrames to train another logistic regression model to predict office occupancy.

How it works...

In this recipe, we used tsfresh to automatically create hundreds of features for each of the five time series, and then used those features to train a logistic regression model to predict whether an office was occupied.

To create features with tsfresh, each time series that we want to extract features from must be marked with a unique identifier. In our dataset, that was the `id` variable.

To create features for each time series, in *step 4*, we used the `extract_features` function from tsfresh and passed the DataFrame containing the time series that we want to extract the features from – that is, `lights` – and the unique identifier for each hour of light consumption records. Note that we had to pass `id` to the `column_id` parameter.

The `extract_features` function has many parameters, some of which we will discuss in upcoming recipes. In this recipe, we used the additional `column_sort` parameter. In *step 4*, we left it set to `None`, so tsfresh assumed that the data was ordered in time and that the time stamps were equidistant. In *step 10*, we passed the `date` variable as the sorting variable.

TIP

In our dataset, we could have either left `column_sort` set to `None` or passed the `date` variable because our time series were indeed ordered in time and the timestamps were equidistant. Otherwise, be mindful of this parameter.

The `extract_features` function also accepts the `impute` function through the `impute_function` parameter, to automatically remove infinite and NAN values from the created features.

NOTE

For more details about the `extract_features` function, visit https://tsfresh.readthedocs.io/en/latest/api/tsfresh.feature_extraction.xhtml#module-tsfresh.feature_extraction.

The `impute` method, which can be used independently, as we did in *step 7*, or within the `extract_features` function, as we did in *step 10*, will replace NAN values and infinite values introduced in the features automatically. It will replace `NAN`, `-Inf`, and `Inf` values with the variable's median, minimum, or maximum values, respectively. If the feature contains only NAN values, they will be replaced by zeroes. The imputation occurs in place – that is, in the same DataFrame that is being imputed.

Now, let's discuss the output of the feature creation process. The **extract features** function returns a DataFrame containing as many rows as there are unique identifiers in the data. In our case, it returned a DataFrame with 135 rows. The columns of the resulting DataFrame correspond to the 789 values that were returned by 63 characterization methods, applied to each of the 135 60-minute time series.

In *step 5*, we explored some of the resulting features, which captured the time serie mean, variance, and coefficient of variation, as well as their length. Let's explore a few more of the resulting features.

Some of the created variables are very self-explanatory. For example, the '**light_skewness**' and '**light_kurtosis**' variables contain the skewness and kurtosis coefficients, which characterize the data distribution. The '**light_has_duplicate_max**', '**light_has_duplicate_min**', and '**light_has_duplicate**' variables indicate whether the time series has duplicated values or duplicated minimum and maximum values. The '**light_quantile_q_0.1**', '**light_quantile_q_0.2**', and '**light_quantile_q_0.3**' variable return the different quantile values of the time series. Finally, the '**light_autocorrelation_lag_0**', '**light_autocorrelation_lag_1**', and '**light_autocorrelation_lag_2**' variables return the autocorrelation of the time series with its past values lagged by 0, 1, or 2 steps, which may be useful in forecasting.

Other characterization methods return features obtained from signal processing algorithms, such as the **continuous wavelet** transform for the Ricker wavelet, which returns the

'**light_cwt_coefficients_coeff_0_w_2_widths_(2, 5, 10,**

```
20)', 'light_cwt_coefficients_coeff_0_w_5_widths_(2, 5,  
10, 20)',  
'light_cwt_coefficients_coeff_0_w_10_widths_(2, 5, 10,  
20)', and  
'light_cwt_coefficients_coeff_0_w_20_widths_(2, 5, 10,  
20)' features, among others.
```

NOTE

We can't discuss each of these feature characterization methods or their outputs in detail in the book because they are too many. You can find more details about the transformations supported by tsfresh and their formulation at

https://tsfresh.readthedocs.io/en/latest/api/tsfresh.feature_extraction.xhtml.

Some of the features that are automatically created by tsfresh may not make sense or even be possible to calculate for some time series because they require a certain length or data variability, or the time series must meet certain distribution assumptions. Therefore, the suitability of the features will depend on the nature of the time series.

TIP

We can decide which features to extract from our time series based on domain knowledge, or by creating all possible features and then applying feature selection algorithms.

See also

For more details on the tsfresh library, check out its original article: *Christ, Braun, Neuffer, and Kempa-Liehr (2018). Time Series FeatURe Extraction on basis of Scalable Hypothesis tests (tsfresh – A Python package).* *Neurocomputing* 307 (2018). Pages 72-77.

<https://dl.acm.org/doi/10.1016/j.neucom.2018.03.067>.

Creating and selecting features for a time series

In the previous recipe, we automatically extracted several hundred features from a time series variable using tsfresh. If we have more than one time series variable, we can easily end up with a dataset that contains thousands of features.

When we create classification and regression models to solve real-life problems, we often want our models to take a small number of relevant features as input to produce their predictions. Simpler models have many advantages. First, their output is easier to interpret for the end users of the models. Second, simpler models are cheaper to store and faster to train. They also return their outputs faster.

The tsfresh library provides a highly parallel feature selection algorithm based on non-parametric statistical hypothesis tests, which can be executed at the back of the feature creation procedure to quickly remove irrelevant features. The feature selection procedure utilizes different tests for different features.

Under the hood, tsfresh uses the following tests:

- Fisher's exact test of independence if both the feature and the target are binary
- Kolmogorov-Smirnov test if either the feature or the target is binary
- Kendall rank test, if neither the feature nor target is binary

The advantage of these tests is that they are non-parametric, and thus make no assumption on the underlying distribution of the variables being tested.

The result of these tests is a vector of p-values that measures the significance of the association between each feature and the target. These p-values are then evaluated based on the Benjamini-Yekutieli procedure to decide which features to keep.

NOTE

For more details about tsfresh's feature selection procedure, check the original article: *Christ, Kempa-Liehr, Feindt: Distributed and parallel time series feature extraction for industrial big data applications. Asian Machine Learning Conference (ACML) 2016, Workshop on Learning on Big Data (WLBD)*, Hamilton (New Zealand), ArXiv, <https://arxiv.org/abs/1610.07717v1>.

In this recipe, we will automatically create hundreds of features from various time series, and then select the most relevant features by utilizing tsfresh.

How to do it...

We will begin by automatically creating and selecting features from one time series, **lights**, and then we will automate the procedure for multiple

time series. We will use the extracted features to predict whether the office was occupied at any given hour:

1. Let's import the required Python libraries and functions:

```
import pandas as pd

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import classification_report

from sklearn.model_selection import train_test_split

from tsfresh import (

    extract_features,

    extract_relevant_features,

    select_features,

)

from tsfresh.utilities.dataframe_functions import impute
```

2. Let's load the dataset and the target variable that we prepared in the *Technical requirements* section into a DataFrame and a pandas series, respectively:

```
X = pd.read_csv("occupancy.csv", parse_dates=["date"])

y = pd.read_csv("occupancy_target.csv", index_col="id")

y = pd.Series(y["occupancy"])
```

3. Let's create hundreds of features automatically for each hour of records of **light** use, and impute the resulting features:

```
features = extract_features(
```

```
X[["id", "light"]],  
    column_id="id",  
    impute_function=impute,  
)
```

The output of the previous step is a DataFrame that consists of 135 rows and 789 columns called **features**, which means that 789 features were created from each hour of light consumption.

NOTE

For more details about *step 3*, or the Occupancy Detection dataset, check out the *Extracting features automatically from a time series* recipe of this chapter.

4. Now, let's select the features based on the non-parametric tests that we mentioned in the introduction of this recipe, and then display the number of selected features:

```
features = select_features(features, y)  
len(features)
```

The output of *step 4* is **135**, which means that from the 789 features created in *step 3*, only **135** are statistically significant. Go ahead and execute **features.head()** to display the first five rows of the resulting DataFrame.

For space reasons, we will only display the first five of the selected features:

```
feats = features.columns[0:5]
```

```
features[feats].head()
```

We can see the values of the first five features for the first 5 hours of light consumption in the following DataFrame:

	light_minimum	light_agg_linear_trend_attr_intercept	_chunk_len_50_f_agg_min
1	0.0		0.0
2	0.0		0.0
3	0.0		0.0
4	0.0		0.0
5	0.0		0.0

	light_quantile_q_0.1	light_quantile_q_0.3	light_quantile_q_0.4
1	0.0	0.0	0.0
2	0.0	0.0	0.0
3	0.0	0.0	0.0
4	0.0	0.0	0.0
5	0.0	0.0	0.0

Figure 10.5 – DataFrame with five of the selected features created for each hour of light consumption

Check the discussion in the *How it works...* section for a more detailed analysis of the DataFrame from step 5.

5. Now, we will use the features from step 4 to train a logistic regression model and predict whether the office was occupied. Let's begin by separating the dataset into train and test sets:

```
x_train, x_test, y_train, y_test = train_test_split(  
    features,  
    y,  
    test_size=0.1,  
    random_state=42,  
)
```

6. Now, let's set up and train a logistic regression model, and then evaluate its performance:

```
cls = LogisticRegression(  
    random_state=10, C=0.1, max_iter=1000)  
  
cls.fit(X_train, y_train)  
  
print(classification_report(y_test,  
    cls.predict(X_test)))
```

In the following output, we can see the evaluation metrics that are commonly used for classification analysis. These suggest that the selected features are useful for predicting office occupancy:

	precision	recall	f1-score	support
0	1.00	0.91	0.95	11
1	0.75	1.00	0.86	3
accuracy			0.93	14

macro avg	0.88	0.95	0.90	14
weighted avg	0.95	0.93	0.93	14

TIP

Go ahead and compare the results from *step 7* with those of *step 9* in the *Extracting features automatically from a time series* recipe of this chapter. You will see that we obtained similar performance, with only a fraction of the features.

7. We can trigger the feature creation and feature selection procedures by using one single function, **extract_relevant_features**, and therefore combine *steps 3* and *4*. Now, let's create features automatically for the five time series in our dataset, then select the most relevant features, and finally display the shape of the resulting DataFrame:

```
features = extract_relevant_features(
    X,
    y,
    column_id="id",
    column_sort="date",
)
features.shape
```

NOTE

The parameters of **extract_relevant_features** are very similar to those of **extract_features**. Note, however, that the former will automatically perform imputation to be able to proceed with the

feature selection. We discussed the parameters of **extract_features** in the *Extracting features automatically from time series* recipe of this chapter.

The output of *step 7* consists of a DataFrame with 135 rows, containing only 969 features from the original 3,945 that are returned by default by tsfresh. Go ahead and use this DataFrame to train another logistic regression model to predict office occupancy.

How it works...

In this recipe, we created hundreds of features from a time series and then selected the most relevant features based on non-parametric statistical tests. The feature creation and selection procedures were carried out automatically by tsfresh.

To create the features, we used tsfresh's **extract_features** function, which we described in detail in the *Extracting features automatically from a time series* recipe of this chapter.

To select features, we used the **select_features** function, also from tsfresh. This function applies different statistical tests, depending on the nature of the feature and the target. Briefly, if the feature and target are binary, it tests their relationship with Fisher's exact test. If either the feature or the target is binary, and the other variable is continuous, it tests their relationship by using the Kolmogorov-Smirnov test. If neither the features nor the target is binary, it uses the Kendall rank test.

The result of these tests is a vector with one p-value per feature. Next, tsfresh applies the Benjamini-Yekutieli procedure, which aims to reduce the

false discovery rate, to select which features to keep based on the p-values. This feature selection procedure has some advantages, the main one being that statistical tests are fast to compute, and therefore the selection algorithm is scalable and can be parallelized.

From *Figure 10.5*, however, we can immediately detect one of the limitations of tsfresh's selection procedure: it is unable to remove redundant features. We know from the *Extracting features automatically from a time series* recipe that if the lights are off, then most likely the office is empty. Therefore, many of the features that capture one aspect of this pattern (lights off) will be important to predict occupancy, and redundant at the same time. Note that in *Figure 10.5*, the different quantiles of light consumption contain identical values, so they are identical features. We only need one of them to train a machine learning model; we can get rid of the rest.

NOTE

The limitations of the tsfresh algorithm are shared by all filter selection methods. Because these methods evaluate each feature individually, they can't assess their interaction with other features, and as such, return redundant features as part of their selection procedure.

Finally, in *step 8*, we combined the feature creation step (*step 3*) and the feature selection step (*step 4*) into one function. The **extract_relevant_features** function applies the **extract_features** function to create the features from each time series and imputes them. Next, it applies the **select_features** function to return a DataFrame containing one row per unique identifier, and the features that were selected

for each time series. Note that different features can be selected for different time series.

See also

The selection algorithm from tsfresh offers a quick method to remove irrelevant features. However, it does not find the best feature subset for the classification or regression task. Other feature selection methods can be applied at the back of tsfresh that can reduce the feature space further.

NOTE

For more details on feature selection algorithms, check out the book *Feature Selection in Machine Learning with Python*, by Soledad Galli on Leanpub: <https://leanpub.com/feature-selection-in-machine-learning/>.

Tailoring feature creation to different time series

The tsfresh library extracts many features based on the time series characteristics and distribution, such as their correlation properties, stationarity, entropy, and non-linear time series analysis functions, which decompose the time series signal through, for example, Fourier or wavelet transformations. Depending on the nature of the time series, some of these transformations make more sense than others. For example, wavelength decomposition methods can make sense for time series resulting from signals or sensors but are unsuitable for time series representing sales or stock prices.

In this recipe, we will discuss how to optimize the tsfresh feature extraction procedure to create specific features for each time series, and then use these features to predict office occupancy.

How to do it...

The tsfresh library accesses the methods that will be used to create features through a dictionary that contains the method's names as keys, and if they are based on a parameter, it has the parameter as a value. The library comes with some predefined dictionaries. We'll explore these predefined dictionaries first, which can be accessed through the **settings** module:

1. Let's import the required Python libraries and functions and the **settings** module:

```
import pandas as pd

from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split
from tsfresh.feature_extraction import extract_features
from tsfresh.feature_extraction import settings
```

2. Let's load the dataset and the target variable that we prepared in the *Technical requirements* section into a DataFrame and a pandas series, respectively:

```
X = pd.read_csv("occupancy.csv", parse_dates=["date"])
y = pd.read_csv("occupancy_target.csv", index_col="id")
y = pd.Series(y["occupancy"])
```

The tsfresh library has three main dictionaries that control the feature creation output: **settings.ComprehensiveFCParameters**, **settings.EfficientFCParameters**, and **settings.MinimalFCParameters**. Let's explore the dictionary that returns the smaller number of features.

3. Let's instantiate the dictionary that returns the smaller number of features and then display the feature creation methods that will be applied by this configuration:

```
minimal_feat = settings.MinimalFCParameters()  
minimal_feat.items()
```

In the output of *step 2*, we can see a dictionary with the feature extraction method names as keys, and the parameters used by those methods, if any, as values:

```
ItemsView({'sum_values': None, 'median': None, 'mean':  
None, 'length': None, 'standard_deviation': None,  
'variance': None, 'root_mean_square': None, 'maximum':  
None, 'absolute_maximum': None, 'minimum': None})
```

NOTE

Go ahead and explore the other two predefined dictionaries, **settings.ComprehensiveFCParameters** and **settings.EfficientFCParameters**, by adapting the code from *step 3*.

4. Now, let's use the dictionary from *step 3* to extract only those features from the **light** time series and then display the shape of the resulting DataFrame:

```
features = extract_features(  
    X[["id", "light"]],  
    column_id="id",  
    default_fc_parameters=minimal_feat,  
)  
  
features.shape
```

The output of *step 4* is **(135, 10)**, which means that only 10 features were created for each of the 135 hours of light consumption data.

5. Let's display the resulting DataFrame:

```
features.head()
```

We can see the values of the features for the first 5 hours of light consumption in the following DataFrame:

	light_sum_values	light_median	light_mean	light_length	light_standard_deviation	light_variance
1	2932.5	0.0	48.875	60.0	134.485582	18086.371875
2	0.0	0.0	0.000	60.0	0.000000	0.000000
3	0.0	0.0	0.000	60.0	0.000000	0.000000
4	0.0	0.0	0.000	60.0	0.000000	0.000000
5	0.0	0.0	0.000	60.0	0.000000	0.000000
	light_variance	light_root_mean_square	light_maximum	light_absolute_maximum	light_minimum	
	18086.371875	143.091361	419.0	419.0	0.0	
	0.000000	0.000000	0.0	0.0	0.0	
	0.000000	0.000000	0.0	0.0	0.0	
	0.000000	0.000000	0.0	0.0	0.0	
	0.000000	0.000000	0.0	0.0	0.0	

Figure 10.6 – DataFrame with the features created for each hour of light consumption

Now, we will use these features to train a logistic regression model and predict whether the office was occupied.

6. Let's begin by separating the dataset into train and test sets:

```
x_train, x_test, y_train, y_test = train_test_split(
    features,
    y,
    test_size=0.1,
    random_state=42,
)
```

7. Now, let's set up and train a logistic regression model, and then evaluate its performance:

```
cls = LogisticRegression(random_state=10, C=0.01)

cls.fit(X_train, y_train)

print(classification_report(y_test,
cls.predict(X_test)))
```

In the following output, we can see the evaluation metrics that are commonly used for classification analysis. These suggest that the selected features are useful for predicting office occupancy:

	precision	recall	f1-score	support
0	1.00	0.91	0.95	11
1	0.75	1.00	0.86	3
accuracy			0.93	14
macro avg	0.88	0.95	0.90	14
weighted avg	0.95	0.93	0.93	14

TIP

Because light consumption is a very good indicator of office occupancy, with very simple features, we can obtain a predictive logistic regression model.

Now, let's learn how to specify the creation of different features for different time series.

8. Let's create a dictionary with the names of the methods that we want to use to create features from the **light** time series. We enter the method's

names as keys, and if the methods take a parameter, we pass it as an additional dictionary to the corresponding key; otherwise, we pass **None** as the values:

```
light_feat = {  
    "sum_values": None,  
    "median": None,  
    "standard_deviation": None,  
    "quantile": [{"q": 0.2}, {"q": 0.7}],  
}
```

9. Now, let's create a dictionary with the features that we want to create from the **co2** time series:

```
co2_feat = {  
    "root_mean_square": None,  
    "number_peaks": [{"n": 1}, {"n": 2}],  
}
```

10. Let's combine these dictionaries into a new dictionary:

```
kind_to_fc_parameters = {  
    "light": light_feat,  
    "co2": co2_feat,  
}
```

11. Finally, let's use the dictionary from *step 10* to create the features from both time series:

```
features = extract_features(
```

```
X[["id", "light", "co2"]],  
    column_id="id",  
    kind_to_fc_parameters=kind_to_fc_parameters,  
)
```

The output of *step 11* consists of a DataFrame with 135 rows and 8 features. If we execute `features.columns`, we will see the names of the created features:

```
Index(['light__sum_values', 'light__median',  
       'light__standard_deviation',  
       'light__quantile__q_0.2',  
       'light__quantile__q_0.7',  
       'co2__root_mean_square',  
       'co2__number_peaks__n_1',  
       'co2__number_peaks__n_2'],  
      dtype='object')
```

Note that in the output from *step 11*, different variables have been created from each of the **light** and **co2** time series.

How it works...

In this recipe, we created a subset of the possible features for our time series data. First, we created features specified in a predefined dictionary that comes with tsfresh. Next, we created our own dictionary, specifying the creation of different features for different time series.

The tsfresh package comes with some predefined dictionaries that can be accessed through the **settings** module. The **MinimalFCParameters** directory triggers the creation of 10 simple features based on basic statistical parameters of the time series distribution. In *step 3*, we displayed the names of the methods that are used to create each one of these 10 variables. Among these, the "**sum_values**" method will return the sum of the time series values, whereas the "**mean**" method will return its mean value. These methods do not need additional parameters to create the features, so for each of the keys corresponding to their names, we used **None** as the values.

The predefined **EfficientFCParameters** dictionary applies the methods that are fast to compute, whereas the **ComprehensiveFCParameters** dictionary returns all possible features.

NOTE

For more details about these dictionaries, check out the tsfresh documentation at

https://tsfresh.readthedocs.io/en/latest/text/feature_extraction_settings.xhtml.

By using these predefined dictionaries in the **default_fc_parameters** parameter of tsfresh's **extract_features** function, we can create specific features from one or more time series, as we did in *step 4*. Note that **default_fc_parameters** instructs **extract_features** to create the same features from all the time series.

To create different features for different time series, we can use the **kind_to_fc_parameters** parameter of tsfresh's **extract_features**

function. This parameter takes a dictionary with dictionaries, where each dictionary specifies parameters for one time series.

In *step 8*, we created a dictionary to specify the creation of features from the **light** time series. Note that the "**sum_values**" and "**mean**" methods take **None** as values, but the **quantile** method needs additional parameters corresponding to the quantiles that should be returned from the time series.

In *step 9*, we created a dictionary to specify the creation of features from the **co2** time series. In *step 10*, we combined both dictionaries into one that takes the name of the time series as keys and the feature creation dictionaries as values. Then, we used this dictionary in **kind_to_fc_parameters** of tsfresh's **extract_features** function.

This way of specifying features is suitable if we use domain knowledge to create the features, or if we only create a small number of features. But do we need to type each method by hand into a dictionary if we want to create multiple features for various time series? Not really. In the following recipe, we will learn how to specify which features to create based on features selected by Lasso.

Creating pre-selected features

In the *Creating and selecting features for a time series* recipe, we learned how to select relevant features using tsfresh. We also discussed how we can use additional feature selection procedures to further reduce the number of features created from our time series.

In this recipe, we will create and select features using tsfresh. Next, we will reduce the feature space by utilizing Lasso regularization. Then, we will

learn how to create a dictionary from the selected feature names to trigger the creation of those features from future time series.

How to do it...

Let's begin by importing the necessary libraries and getting the dataset ready:

1. Let's import the required libraries and functions:

```
import pandas as pd

from sklearn.feature_selection import SelectFromModel
from sklearn.linear_model import LogisticRegression
from tsfresh import (
    extract_features,
    extract_relevant_features,
)
from tsfresh.feature_extraction import settings
```

2. Let's load the Occupancy Detection dataset that we prepared in the *Technical requirements* section and its target variable:

```
x = pd.read_csv("occupancy.csv", parse_dates=["date"])
y = pd.read_csv("occupancy_target.csv", index_col="id")
y = pd.Series(y["occupancy"])
```

3. Let's create and select features for our five time series and then display the shape of the resulting DataFrame:

```
features = extract_relevant_features(
```

```
x,  
y,  
column_id="id",  
column_sort="date",  
)  
features.shape
```

The output of *step 2* is **(135, 969)**, indicating that 969 features were returned from the five original time series, for each hour of records.

NOTE

We discussed the function from *step 3* in the *Extracting features automatically from a time series* and *Creating and selecting features for a time series* recipes.

4. Now, let's select features using Lasso regularization. First, let's set up a logistic regression with Lasso regularization, which is the "**l1**" penalty, and some additional parameters that I set arbitrarily:

```
cls = LogisticRegression(  
    penalty="l1",  
    solver="liblinear",  
    random_state=10,  
    C=0.05,  
    max_iter=1000,  
)
```

5. Next, we must set up a transformer that will select the features based on the coefficients from the logistic regression model from *step 4*:

```
selector = SelectFromModel(cls)
```

6. Let's train the logistic regression model and select the features:

```
selector.fit(features, y)
```

7. Now, let's capture the selected features in a new variable and display their names:

```
features = selector.get_feature_names_out()
```

```
features
```

In the following output, we can see the names of the selected features:

```
array(['light_sum_of_reoccurring_data_points',
       'co2_spkt_welch_density_coeff_2', 'co2_variance',
       'temperature_c3_lag_1', 'temperature_abs_energy',
       'temperature_c3_lag_2', 'temperature_c3_lag_3',
       'co2_sum_of_reoccurring_data_points',
       'light_spkt_welch_density_coeff_8', 'light_variance',
       'light_agg_linear_trend_attr_"slope"_chunk_len_50_f_agg_"var"',
       'light_mean_f_agg_"var"])
```

```
'light__agg_linear_trend__attr_"intercept"__chunk_
len_10__f_agg_"var"' ],
dtype=object)
```

Note that Lasso regularization selected features created by different feature characterization methods from the time series – that is, **light**, **co2**, and **temperature**.

8. To create the features from *step 6* from our dataset, we need to capture their names and the parameters they use in a dictionary. We can do this automatically from the feature names with tsfresh:

```
kind_to_fc_parameters = settings.from_columns(
    selector.get_feature_names_out(),
)
```

9. Let's display the dictionary that we just created:

```
kind_to_fc_parameters
```

In the output of *step 9*, we can see the dictionary that was created from the names of the features from *step 6*:

```
{'light':
    {'sum_of_reoccurring_data_points': None,
     'spkt_welch_density': [{coeff': 8}],
     'variance': None,
     'agg_linear_trend': [
         {'attr': 'slope', 'chunk_len': 50, 'f_agg': 'var'},
```

```

        {'attr': 'intercept', 'chunk_len': 10, 'f_agg':'var'}
    ],
},
'co2':
{
    'spkt_welch_density': [{ 'coeff': 2}],
    'variance': None,
    'sum_of_reoccurring_data_points': None
},
'temperature': {'c3': [ { 'lag': 1}, { 'lag': 2}, { 'lag':3}], 'abs_energy': None}
}

```

10. Now, we can use the dictionary from *step 7* together with the **extract_features** function to create only those features from our dataset:

```

features = extract_features(
    X,
    column_id="id",
    column_sort="date",
    kind_to_fc_parameters=kind_to_fc_parameters,
)

```

The new DataFrame, which can be displayed by executing `features.head()`, only contains the 12 selected features. Go ahead and corroborate the result on your computer, or check this book's GitHub repository.

How it works...

In this recipe, we created 969 features from five time series. Next, we reduced the feature space to 12 features by using Lasso regularization. Finally, we captured the specifications of the selected features in a dictionary so that, looking forward, we only create those features from our time series.

To automatically create and select features with tsfresh, we used the `extract_relevant_features` function, which we described in detail in the *Creating and selecting features for a time series* recipe.

Lasso regularization has the intrinsic ability to reduce some of the coefficients of the logistic regression model to zero. The contribution of the features whose coefficient is zero to the prediction of office occupancy is null and can therefore be removed. The `SelectFromModel()` class can identify and remove those features. We set up an instance of `SelectFromModel()` with a logistic regression model that used Lasso regularization to find the model coefficients. With the `fit()` method, `SelectFromModel()` trained the logistic regression model using the 969 features created from our time series and identified those whose coefficients were different from zero. Then, with the `get_feature_names_out()` method, we captured the names of the selected features in a new variable.

To create only the 12 features selected by Lasso regularization, we created a dictionary from the variable names by using the `from_columns()` function from tsfresh. This function returned a dictionary with the variables from which features were selected as keys. The values were additional dictionaries, containing the methods used to create features as keys, and the parameters used, if any, as values. To create the new features, we used this dictionary together with the `extract_features` function.

NOTE

Note that in step 8, we passed the entire dataset to the `extract_features` function. The resulting features only contained features extracted from three of the five time series.

Embedding feature creation in a scikit-learn pipeline

Throughout this chapter, we discussed how to automatically create and select features from time series data by utilizing tsfresh. Then, we used these features to train a classification model to predict whether an office was occupied at any given hour.

The tsfresh library offers wrapper classes around its main functions, `extract_features` and `extract_relevant_features`, to make features that have been created from time series compatible with the scikit-learn pipeline.

In this recipe, we will line up the process of creating features with tsfresh and training a logistic regression model in a scikit-learn pipeline.

How to do it...

Let's begin by importing the necessary libraries and getting the dataset ready:

1. Let's import the required libraries and functions:

```
import pandas as pd

from sklearn.pipeline import Pipeline

from sklearn.linear_model import LogisticRegression

from sklearn.model_selection import train_test_split

from sklearn.metrics import classification_report

from tsfresh.transformers import

RelevantFeatureAugmenter
```

2. Let's load the Occupancy Detection dataset that we prepared in the *Technical requirements* section and its target variable:

```
X = pd.read_csv("occupancy.csv", parse_dates=["date"])

y = pd.read_csv("occupancy_target.csv", index_col="id")

y = pd.Series(y["occupancy"])
```

3. Let's create an empty DataFrame that contains the index of the target variable:

```
tmp = pd.DataFrame(index=y.index)
```

4. Now, let's split the DataFrame from *step 3* and the target from *step 2* into train and test sets:

```
X_train, X_test, y_train, y_test = train_test_split(tmp, y)
```

NOTE

X_train and **X_test** are containers where the features created by tsfresh will be added. They are crucial for the functionality of **RelevantFeatureAugmenter()**.

5. Let's create a dictionary containing the specifications of the features that should be created from each time series. I have defined the features arbitrarily this time:

```
kind_to_fc_parameters = {

    "light": {

        "c3": [{"lag": 3}, {"lag": 2}, {"lag": 1}],

        "abs_energy": None,
        "sum_values": None,
        "fft_coefficient": [
            {"attr": "real", "coeff": 0},
            {"attr": "abs", "coeff": 0}],
        "spkt_welch_density": [
            {"coeff": 2}, {"coeff": 5}, {"coeff": 8}]

    },
    "agg_linear_trend": [
        {"attr": "intercept",
         "chunk_len": 50, "f_agg": "var"},

        {"attr": "slope",
         "chunk_len": 50, "f_agg": "var"}]
}
```

```
],
  "change_quantiles": [
    {"f_agg": "var", "isabs": False,
     "qh": 1.0, "ql": 0.8},
    {"f_agg": "var", "isabs": True,
     "qh": 1.0, "ql": 0.8},
  ],
},
"co2": {
  "fft_coefficient": [
    {"attr": "real", "coeff": 0},
    {"attr": "abs", "coeff": 0}],
  "c3": [{"lag": 3}, {"lag": 2}, {"lag": 1}],
  "sum_values": None,
  "abs_energy": None,
  "sum_of_reoccurring_data_points": None,
  "sum_of_reoccurring_values": None,
},
"temperature": {"c3": [{"lag": 1},
 {"lag": 2}, {"lag": 3}], "abs_energy": None},
}
```

NOTE

We discussed the parameters of the dictionary from *step 5* in the *Tailoring feature creation to a different time series* recipe.

6. Let's set up **RelevantFeatureAugmenter()**, which is a wrapper around the **extract_relevant_features** function, to create the features specified in *step 5*:

```
augmenter = RelevantFeatureAugmenter(  
    column_id="id",  
    column_sort="date",  
    kind_to_fc_parameters=kind_to_fc_parameters,  
)
```

NOTE

To create all possible features, use the **FeatureAugmenter()** class instead in *step 6*.

7. Let's combine the feature creation instance from *step 6* with a logistic regression model in a scikit-learn pipeline:

```
pipe = Pipeline(  
    [  
        ("augmenter", augmenter),  
        ("classifier", LogisticRegression(  
            random_state=10, C=0.01)),  
    ]  
)
```

8. Now, let's tell **RelevantFeatureAugmenter()** which dataset it needs to use to create the features:

```
pipe.set_params(augmenter__timeseries_container=X)
```

9. Now, let's fit the pipeline, which will trigger the feature creation process, followed by the training of the logistic regression model:

```
pipe.fit(X_train, y_train)
```

10. Now, let's obtain predictions using the time series in the test set, and evaluate the model's performance through a classification report:

```
print(classification_report(  
    y_test, pipe.predict(X_test)))
```

We can see the output of *step 10* here:

	precision	recall	f1-score	support
0	0.96	1.00	0.98	27
1	1.00	0.86	0.92	7
accuracy			0.97	34
macro avg	0.98	0.93	0.95	34
weighted avg	0.97	0.97	0.97	34

The values of the classification report suggest that the extracted features are suitable for predicting whether the office is occupied at any given hour.

How it works...

In this recipe, we combined creating features from a time series with tsfresh with training a machine learning algorithm from the scikit-learn library in a pipeline.

The tsfresh library offers two wrappers around its main functions to make the feature creation process compatible with the scikit-learn pipeline. In this recipe, we used the **RelevantFeatureAugmenter()** class, which wraps the **extract_relevant_features** function to create and then select features from a time series.

The default functionality of **RelevantFeatureAugmenter()** is as follows. With the **fit()** method, **RelevantFeatureAugmenter()** creates and selects features. The name of the selected features are then stored internally in the transformer. With the **transform()** method, **RelevantFeatureAugmenter()** creates the selected features from the time series.

We overrode the default functionality of **RelevantFeatureAugmenter()** by passing a dictionary with the features we wanted to create to its **kind_to_fc_parameters** parameter. Therefore, with **transform()**, **RelevantFeatureAugmenter()** created the indicated features from the time series.

To create all features from the time series, tsfresh offers the **FeatureAugmenter()** class, which has the same functionality as **RelevantFeatureAugmenter()**, but without the feature selection step.

Both **RelevantFeatureAugmenter()** and **FeatureAugmenter()** need two DataFrames to work. The first DataFrame contains the time series data with the unique identifiers (we loaded this DataFrame in *step 2*). The

second DataFrame should be empty and contain the unique identifiers in its index (we created this DataFrame in *step 3*). After the features were created by the `transform()` method, they were added to this second DataFrame.

Note how, in *step 9*, we fit the pipeline by using the empty DataFrame, but in *step 8*, we set up `RelevantFeatureAugmenter()` to extract features from the raw time series data.

When we `fit()` the pipeline, we created features from our raw time series and trained a logistic regression model with the resulting features. With the `predict()` method, we created features from the test set, and the subsequent logistic regression predictions based on those features.

See also

For more details about the classes and procedures used in this recipe, visit the following links:

- The tsfresh documentation:
https://tsfresh.readthedocs.io/en/latest/api/tsfresh.transformers.xhtml#tsfresh.transformers.relevant_feature_augmenter.RelevantFeatureAugmenter
- A Jupyter Notebook with a demo: <https://github.com/blue-yonder/tsfresh/blob/main/notebooks/examples/02%20sklearn%20Pipeline.ipynb>

11

Extracting Features from Text Variables

Text can be part of the variables in our datasets. For example, in insurance, information about an incident may come from free text fields in a form. On a website that gathers customer reviews, some information may come from short text descriptions provided by the users. Text data does not show the **tabular** pattern of the datasets we have worked with throughout this book. Instead, information in texts can vary in length and content, and the writing style may be different. We can still extract a lot of information from text variables to use as predictive features in machine learning models. The techniques we will cover in this chapter belong to the realm of **Natural Language Processing (NLP)**. NLP is a subfield of linguistics and computer science, concerned with the interactions between computer and human language, or, in other words, how to program computers to understand human language. NLP includes a multitude of techniques to understand the syntax, semantics, and discourse of text, and therefore to do this field justice, it would require a book in itself.

In this chapter, we will discuss those techniques that will allow us to quickly extract features from short pieces of text, to complement our predictive models. Specifically, we will discuss how to capture a piece of text's complexity by looking at some statistical parameters of the text, such as the word length and count, the number of words and unique words used, the number of sentences, and so on. We will use the pandas and scikit-learn

libraries, and we will make a shallow dive into a very useful Python NLP toolkit called **Natural Language Toolkit (NLTK)**.

This chapter will cover the following recipes:

- Counting characters, words, and vocabulary
- Estimating text complexity by counting sentences
- Creating features with bag-of-words and n-grams
- Implementing term frequency-inverse document frequency
- Cleaning and stemming text variables

Technical requirements

In this chapter, we will use the pandas, Matplotlib, and scikit-learn Python libraries. We will also use NLTK from Python, a comprehensive library for NLP and text analysis. You can find the instructions to install NLTK here:

<http://www.nltk.org/install.xhtml>. If you are using the Python Anaconda distribution, follow these instructions to install NLTK:

<https://anaconda.org/anaconda/nltk>.

After you have installed NLTK, open up a Python console and execute the following:

```
import nltk  
nltk.download('punkt')  
nltk.download('stopwords')
```

These commands will download the necessary data for you to be able to run the recipes in this chapter successfully.

NOTE

If you haven't downloaded these or other data sources necessary for NLTK functionality, NLTK will raise an error. Read the error message carefully because it will direct you to download the data required to run the command you are trying to execute.

Counting characters, words, and vocabulary

One of the salient characteristics of text is its complexity. Long descriptions are more likely to contain more information than short descriptions. Texts rich in different, unique words are more likely to be richer in detail than texts that repeat the same words over and over. In the same way, when we speak, we use many short words such as articles and prepositions to build the sentence structure, yet the main concept is often derived from the nouns and adjectives we use, which tend to be longer words. So, as you can see, even without reading the text, we can start inferring how much information the text provides by determining the number of words, the number of unique words, the lexical diversity, and the length of those words. In this recipe, we will learn how to extract these features from a text variable using pandas.

Getting ready

We are going to use the **20 Newsgroup** dataset that comes with scikit-learn, which comprises around 1,800 news posts on 20 different topics. More details about this dataset can be found in these links:

- Scikit-learn dataset website: [z](#)
- Home page for the **20 Newsgroup** dataset:
<http://qwone.com/~jason/20Newsgroups/>

Before jumping into the recipe, let's become familiar with the features we are going to derive from these text pieces. We mentioned that longer descriptions, more words in the article, a greater variety of unique words, and longer words tend to correlate with the amount of information the article provides. Hence, we can capture text complexity by extracting the following information:

- The total number of characters in the text
- The total number of words
- The total number of unique words
- Lexical diversity = total number of words / number of unique words
- Word average length = number of characters / number of words

In this recipe, we will extract these numerical features using pandas, which is equipped with multiple pieces of string processing functionality that can be accessed via the **str** vectorized string functions for series.

How to do it...

Let's begin by loading pandas and getting the dataset ready to use:

1. Load **pandas** and the dataset from scikit-learn:

```
import pandas as pd  
from sklearn.datasets import fetch_20newsgroups
```

2. Let's load the train set part of the **20 Newsgroup** dataset into a pandas DataFrame:

```
data = fetch_20newsgroups(subset='train')  
df = pd.DataFrame(data.data, columns=['text'])
```

TIP

You can print out an example of a **text** variable in the DataFrame by executing `print(df['text'][1])`. Change the number between `[]` to navigate through different texts. Note how every text description is a single string composed of letters, numbers, punctuation, and spaces.

Now that we have the text in a pandas DataFrame, we are ready to extract the features.

3. Let's capture the number of characters in each text piece in a new column:

```
df['num_char'] = df['text'].str.len()
```

TIP

You can remove trailing whitespaces, including new lines, in a string before counting the number of characters by adding the **strip()** method before the **len()** method: `df['num_char'] = df['text'].str.strip().str.len()`.

4. Let's capture the number of words in each text in a new column:

```
df['num_words'] = df['text'].str.split().str.len()
```

5. Let's capture the number of **unique** words in each text in a new column:

```
df['num_vocab'] = df['text'].str.lower().str.split().apply(  
    set).str.len()
```

NOTE

Python will interpret the same word as two different words if one has a capital letter. To avoid this behavior, we can introduce the **lower()** method before the **split()** method.

6. Let's create a feature that captures the **lexical diversity** – that is, the total number of words to the number of unique words:

```
df['lexical_div'] = df['num_words'] / df['num_vocab']
```

7. Let's calculate the average word length by dividing the number of characters by the number of words:

```
df['ave_word_length'] = df['num_char'] / df['num_words']
```

If we execute **df.head()**, we will see the first five rows of data with the text and the newly created features:

		text	num_char	num_words	num_vocab	lexical_div	ave_word_length
0		From: lerxst@wam.umd.edu (where's my thing)\nS...	716	123	93	1.322581	5.821138
1		From: guykuo@carson.u.washington.edu (Guy Kuo)...	857	123	99	1.242424	6.967480
2		From: twillis@ec.ecn.purdue.edu (Thomas E Will...	1980	339	219	1.547945	5.840708
3		From: jgreen@amber (Joe Green)\nSubject: Re: W...	814	113	96	1.177083	7.203540
4		From: jcm@head-cfa.harvard.edu (Jonathan McDow...	1117	171	139	1.230216	6.532164

Figure 11.1 – DataFrame view with the text variable and the new features

With that, we have extracted five different features that capture the text complexity, which we can use as inputs for our machine learning algorithms. With **df.head()**, you can peek at the values of the first five rows of the created features.

NOTE

In this recipe, we created new features straight away from the raw data, without doing any data cleaning, removing punctuation, or even stemming words. Note that these are steps that are performed ahead of most NLP standard procedures. To learn more about this, visit the *Cleaning and stemming text variables* recipe at the end of this chapter.

How it works...

In this recipe, we created five new features that capture text complexity by utilizing pandas `str` to access built-in pandas functionality to work with strings. We worked with the text column of the train subset of the **20 Newsgroup** dataset that comes with scikit-learn. Each row in this dataset is composed of a string with text.

We used pandas `str`, followed by `len()`, to count the number of characters in each string – that is, the total number of letters, numbers, symbols, and spaces. We also combined `str.len()` with `str.strip()` to remove trailing whitespaces at the beginning and end of the string and in new lines, before counting the number of characters.

To count the number of words, we used pandas `str`, followed by `split()`, to divide the string into a list of words. The `split()` method creates a list of words by breaking the string at the whitespaces between words. Next, we counted those words with pandas `str.len()`, obtaining the number of words per string.

NOTE

We can alter the behavior of `str.split()` by passing the string or character that we would like to have the strings divided by. For

example, `df['text'].str.split(';')` divides a string at each occurrence of ;.

To determine the number of unique words, we used pandas `str.split()` to divide the string into a list of words. Next, we applied the built-in Python `set()` method within pandas `apply()` to return a set of words; remember that a set contains **unique occurrences** of the elements in a list – that is, unique words. Next, we counted those words with pandas `str.len()` to return the vocabulary, or, in other words, the number of unique words in the string. Python interprets words that are written in uppercase differently from those in lowercase; therefore, we introduced the pandas `lower()` method to set all the characters in lowercase before splitting the string and counting the number of unique words.

To create the **lexical diversity** and **average word length** features, we simply performed a vectorized division of two pandas Series. And that's it: we created five new features with information about the complexity of the text.

There's more...

We can get a glimpse of the distribution of the newly created features in each of the 20 different news topics present in the dataset by introducing some simple visualizations. To make histogram plots of the newly created features, after you have run all of the steps in the *How it works...* section of this recipe, follow these steps:

1. Import `matplotlib`:

```
import matplotlib.pyplot as plt
```

2. Add the target with the news topics to the **20 Newsgroup** DataFrame:

```
df['target'] = data.target
```

3. Create a function that displays a histogram of a feature of your choice for each of the news topics:

```
def plot_features(df, text_var):  
    nb_rows = 5  
    nb_cols = 4  
    fig, axs = plt.subplots(  
        nb_rows, nb_cols, figsize=(12, 12))  
    plt.subplots_adjust(wspace=None, hspace=0.4)  
    n = 0  
    for i in range(0, nb_rows):  
        for j in range(0, nb_cols):  
            axs[i, j].hist(df[df.target==n][text_var], bins=30)  
            axs[i, j].set_title(text_var + ' | ' +  
                str(n))  
            n += 1  
    plt.show()
```

4. Run the function for the **number of words** feature:

```
plot_features(df, 'num_words')
```

The preceding code block returns a plot where you can see the distribution of the number of words in each of the 20 news topics, numbered from 0 to 19 in the plot title:

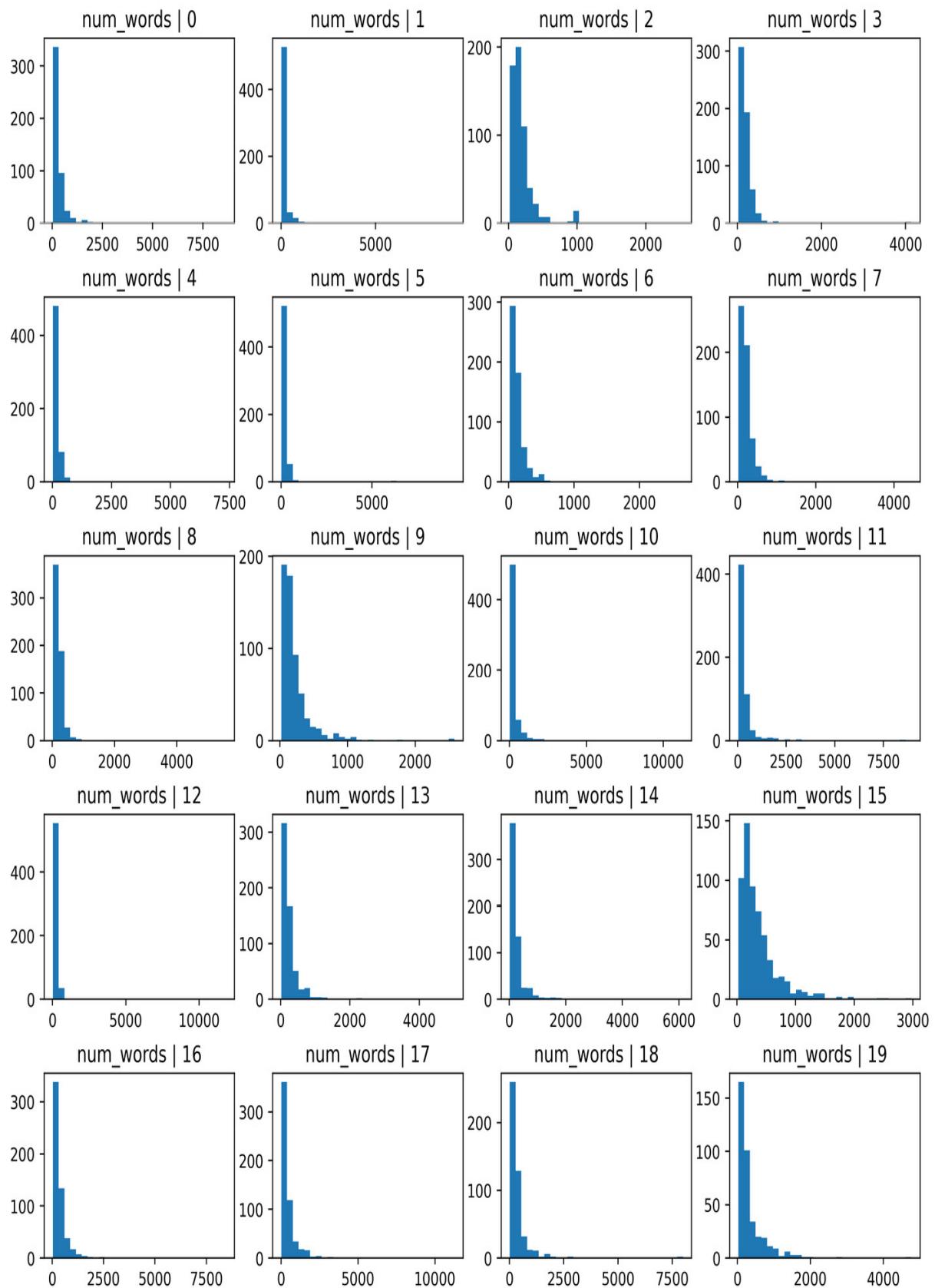


Figure 11.2 – Histograms of the features extracted from text

The number of words shows a different distribution, depending on the news topics. Therefore, this feature is likely useful in a classification algorithm to predict the topic of the text.

See also

To learn more about the built-in string processing functionality from pandas `str`, go to https://pandas.pydata.org/pandas-docs/stable/user_guide/text.xhtml#method-summary.

Estimating text complexity by counting sentences

One aspect of a piece of text we can capture in features is its complexity. Usually, longer descriptions that contain multiple sentences spread over several paragraphs tend to provide more information than descriptions with very few sentences. Therefore, capturing the number of sentences may provide some insight into the amount of information provided by the text. This process is called sentence tokenization. Tokenization is the process of splitting a string into a list of pieces or tokens. In the *Counting characters, words, and vocabulary* recipe, we did word tokenization – that is, we divided the string into words. In this recipe, we will divide the string into sentences, and then we will count them. We will use the NLTK Python library, which provides this functionality.

Getting ready

In this recipe, we will use the NLTK Python library. For guidelines on how to install NLTK, check out the *Technical requirements* section of this chapter.

How to do it...

Let's begin by importing the required libraries and dataset:

1. Load **pandas**, the sentence tokenizer from NLTK, and the dataset from scikit-learn:

```
import pandas as pd
from nltk.tokenize import sent_tokenize
from sklearn.datasets import fetch_20newsgroups
```

2. To understand the functionality of the sentence tokenizer from NLTK, let's create a variable that contains a string with multiple sentences:

```
text = """
The alarm rang at 7 in the morning as it usually did on
Tuesdays. She rolled over, stretched her arm, and
stumbled to the button till she finally managed to
switch it off. Reluctantly, she got up and went for a
shower. The water was cold as the day before the
engineers did not manage to get the boiler working. Good
thing it was still summer.

Upstairs, her cat waited eagerly for his morning snack.
Miaow! He voiced with excitement as he saw her climb the
stairs.

"""
```

3. Now, let's separate the string we created in step 2 into sentences using the NLTK sentence tokenizer:

```
sent_tokenize(text)
```

TIP

If you encounter an error in step 3, read the error message carefully and download the data source required by NLTK, as described in the error message. For more details, check out the *Technical requirements* section.

The sentence tokenizer will return the list of sentences shown in the following output:

```
[ '\nThe alarm rang at 7 in the morning as it usually did on
Tuesdays.',
 'She rolled over,\nstretched her arm, and stumbled to the
button till she finally managed to switch it off.',
 'Reluctantly, she got up and went for a shower.',
 'The water was cold as the day before the engineers\ndid
not manage to get the boiler working.',
 'Good thing it was still summer.',
 'Upstairs, her cat waited eagerly for his morning snack.',
 'Miaow!',
 'He voiced with excitement\nas he saw her climb the
stairs.]
```

NOTE

The escape character followed by the letter `\n` indicates a new line.

4. Let's count the number of sentences in the `text` variable:

```
len(sent_tokenize(text))
```

The code in the preceding line returns **8**, which is the number of sentences in our `text` variable. Now, let's determine the number of sentences in an entire DataFrame.

5. Let's load the train subset of the **20 Newsgroup** dataset into a pandas DataFrame:

```
data = fetch_20newsgroups(subset='train')
df = pd.DataFrame(data.data, columns=['text'])
```

6. To speed up the following steps, we will only work with the first **10** rows of the DataFrame:

```
df = df.loc[1:10]
```

7. Let's also remove the first part of the text, which contains information about the email sender, subject, and other details we are not interested in. Most of this information comes before the word **Lines** followed by :, so let's split the string at **Lines:** and capture the second part of the string:

```
df['text'] = df['text'].str.split('Lines:').apply(  
    lambda x: x[1])
```

8. Finally, let's create a variable that captures the number of sentences per **text** variable:

```
df['num_sent'] = df['text'].apply(  
    sent_tokenize).apply(len)
```

With the **df** command, you can display the entire DataFrame with the **text** variable and the new feature containing the number of sentences per text:

		text	num_sent
1	11\nNNTP-Posting-Host: carson.u.washington.ed...		6
2	36\n\nwell folks, my mac plus finally gave up...		9
3	14\nDistribution: world\nNNTP-Posting-Host: a...		7
4	23\n\nFrom article <C5owCB.n3p@world.std.com>...		10
5	58\n\nIn article <1r1eu1\$4t@transfer.stratus....		21
6	12\n\nThere were a few people who responded t...		8
7	44\nDistribution: world\nNNTP-Posting-Host: d...		15
8	10\n\nI have win 3.0 and downloaded several i...		3
9	29\n\njap10@po.CWRU.Edu (Joseph A. Pellettier...		12
10	13\n\nI have a line on a Ducati 900GTS 1978 m...		11

Figure 11.3 – DataFrame with the text variable and the number of sentences per text

Now, we can use this new feature in machine learning algorithms.

How it works...

In this recipe, we separated a string with text into sentences using **sent_tokenizer** from the NLTK library. **sent_tokenizer** has been pre-trained to recognize capitalization and different types of punctuation that signal the beginning and the end of a sentence.

First, we applied **sent_tokenizer** to a manually created string to become familiar with its functionality. The tokenizer divided the text into a list of

seven sentences. We combined the tokenizer with the built-in Python `len()` method to count the number of sentences in the string.

Next, we loaded a dataset with text and, to speed up the computation, we only retained the first 10 rows of the DataFrame using pandas `loc[]`. Next, we removed the first part of the text, which contained information about the email sender and subject. To do this, we split the string at `Line:` using pandas `str.split()`, which returned a list with two elements – the strings before and after `Line:`. Utilizing a lambda function within pandas `apply()`, we retained the second part of the text – that is, the second string in the list returned by pandas `split()`.

Finally, we applied `sent_tokenizer` to each row in the DataFrame with the pandas `apply()` method, to separate the strings into sentences, and then applied the built-in Python `len()` method to the list of sentences to return the number of sentences per string. This way, we created a new feature that contains the number of sentences per text.

There's more...

NLTK has functionality for word tokenization among other useful features, which we can use instead of pandas to count and return the number of words. You can find out more about NLTK's functionality here:

- *Python 3 Text Processing with NLTK 3 Cookbook*, by Jacob Perkins, Packt Publishing
- NLTK documentation: <http://www.nltk.org/>

Creating features with bag-of-words and n-grams

A **Bag-of-Words (BoW)** is a simplified representation of a piece of text that captures the words that are present in the text and the number of times each word appears in the text. So, for the text string *Dogs like cats, but cats do not like dogs*, the derived BoW is as follows:

dogs	like	cats	but	do	not
2	2	2	1	1	1

Figure 11.4 – BoW derived from the sentence “Dogs like cats, but cats do not like dogs”

Here, each word becomes a variable, and the value of the variable represents the number of times the word appears in the string. As you can see, BoW captures multiplicity but does not retain word order or grammar. That is why it is a simple, yet useful, way of extracting features and capturing some information about the texts we are working with.

To capture some syntax, BoW can be used together with n-grams. An n-gram is a contiguous sequence of n items in a given text. Continuing with the sentence *Dogs like cats, but cats do not like dogs*, the derived 2-grams are as follows:

- Dogs like
- like cats
- cats but
- but do

- do not
- like dogs

We can create, together with a BoW, a bag of n-grams, where the additional variables are given by the 2-grams and the values for each 2-grams are the number of times they appear in each string; for this particular example, the value is 1. So, our final BoW with 2-grams would look like this:

dogs	like	cats	but	do	not	dogs	like	cats	but	do	not	like
2	2	2	1	1	1	1	1	1	1	1	1	1

Figure 11.5 – BoW with 2-grams

In this recipe, we will learn how to create BoWs with or without n-grams using scikit-learn.

Getting ready

Before jumping into this recipe, let's get familiar with some of the parameters of a BoW that we can adjust to make the BoW more or less comprehensive. When creating a BoW over several pieces of text, a new feature is created for each unique word that appears at least once in any of the text pieces we are analyzing. If the word appears only in one piece of text, it will show a value of 1 for that particular text and 0 for all of the others.

Therefore, BoWs tend to be sparse matrices, where most of the values are zeros. Also, the number of columns – that is, the number of words – can be

quite large if we work with huge text corpora, and even bigger if we also include n-grams. To limit the number of columns created and the sparsity of the returned matrix, we can choose to retain words that appear across multiple texts; or, in other words, we can retain words that appear in, at least, a certain percentage of texts.

To reduce the number of columns and sparsity of the BoW, we should also work with words in the same case – for example, lowercase – as Python will identify words in a different case as different words. We can also reduce the number of columns and sparsity by removing **stop words**. Stop words are very frequently used words that make sentences flow, but that, per se, do not carry any useful information. Examples of stop words are pronouns such as **I**, **you**, and **he**, as well as prepositions and articles.

In this recipe, we will learn how to set words in lowercase, remove stop words, retain words with a minimum acceptable frequency, and capture n-grams all together with a single transformer from scikit-learn:
CountVectorizer().

How to do it...

Let's begin by loading the necessary libraries and getting the dataset ready:

1. Load **pandas**, **CountVectorizer**, and the dataset from scikit-learn:

```
import pandas as pd
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import
CountVectorizer
```

2. Let's load the train set part of the **20 Newsgroup** dataset into a pandas DataFrame:

```
data = fetch_20newsgroups(subset='train')
df = pd.DataFrame(data.data, columns=['text'])
```

3. To make interpreting the results easier, let's remove punctuation and numbers from the text variable:

```
df['text'] = df['text'].str.replace(
    '[^\w\s]', '', regex=True).str.replace(
    '\d+', '', regex=True)
```

4. Now, let's set up **CountVectorizer()** so that, before creating the BoW, it puts the text in lowercase, removes stop words, and retains words that appear at least in 5% of the text pieces:

```
vectorizer = CountVectorizer(
    lowercase=True,
    stop_words='english',
    ngram_range=(1, 1),
    min_df=0.05)
```

NOTE

To introduce n-grams as part of the returned columns, we can change the value of **ngrams_range** to, for example, **(1, 2)**. The tuple provides the lower and upper boundaries of the range of n-values for different n-grams to be extracted. In the case of **(1, 2)**, **CountVectorizer()** will return single words and arrays of two consecutive words.

5. Let's fit **CountVectorizer()** so that it learns which words should be used in the BoW:

```
vectorizer.fit(df['text'])
```

6. Now, let's create the BoW:

```
X = vectorizer.transform(df['text'])
```

7. Finally, let's capture the BoW in a DataFrame with the corresponding feature names:

```
bagofwords = pd.DataFrame(  
    X.toarray(),  
    columns = vectorizer.get_feature_names_out()  
)
```

With that, we have created a pandas DataFrame that contains words as columns and the number of times they appeared in each text as values. You can inspect the result by executing **bagofwords.head()**:

	able	access	actually	ago	apr	article	articleid	ask	available	away	...	works	world	writes	wrong	wrote	xnewsreader	year	years	yes	you're
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	1	0	0
1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0	1	1	0	0	0	0
4	0	0	0	0	0	0	2	0	0	0	0	0	0	0	1	0	0	0	0	1	0

5 rows × 191 columns

Figure 11.6 – DataFrame with the BoW

We can use this BoW as input for a machine learning model.

How it works...

CountVectorizer() from scikit-learn converts a collection of text documents into a matrix of token counts. These tokens can be individual

words or arrays of two or more consecutive words – that is, n-grams. In this recipe, we created a BoW from a text variable in a DataFrame.

We loaded the **20 Newsgroup** text dataset from scikit-learn and removed punctuation and numbers from the text rows using pandas **replace()**, which can be accessed through pandas **str**, to replace digits, '\d+', or symbols, '[^\w\s]', with empty strings, ''. Then, we used **CountVectorizer()** to create the BoW. We set the **lowercase** parameter to **True**, to put the words in lowercase before extracting the BoW. We set the **stop_words** argument to **english** to ignore stop words – that is, to avoid stop words in the BoW. We set **ngram_range** to the **(1, 1)** tuple to return only single words as columns. Finally, we set **min_df** to **0.05** to return words that appear in at least 5% of the texts, or, in other words, in 5 % of the rows in the DataFrame.

After setting up the transformer, we used the **fit()** method to allow the transformer to find the words that fulfill the preceding criteria. Finally, using the **transform()** method, the transformer returned an object containing the BoW with its feature names, which we captured in a pandas DataFrame.

See also

For more details about **CountVectorizer()**, visit the scikit-learn documentation at https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.xhtml.

Implementing term frequency-inverse document frequency

Term Frequency-Inverse Document Frequency (TF-IDF) is a numerical statistic that captures how relevant a word is in a document concerning the entire collection of documents. What does this mean? Some words will appear a lot within a text document as well as across documents, for example, the English words **the**, **a**, and **is**. These words generally convey little information about the actual content of the document and don't make it stand out from the crowd. TF-IDF provides a way to weigh the importance of a word by contemplating how many times it appears in a document, concerning how often it appears across documents. Hence, commonly occurring words such as **the**, **a**, and **is** will have a low weight, and words more specific to a topic, such as **leopard**, will have a higher weight.

TF-IDF is the product of two statistics: **term frequency** and **inverse document frequency**. Term frequency is, in its simplest form, the count of the word in an individual text. So, for term t , the term frequency is calculated as $tf(t) = \text{count}(t)$ and is determined text by text. The inverse document frequency is a measure of how common the word is *across* all documents and is usually calculated on a logarithmic scale. A common implementation is given by the following:

$$idf(t) = \log \left(\frac{n}{1 + df(t)} \right)$$

Here, n is the total number of documents, and $df(t)$ is the number of documents in which the term t appears. The bigger the value of $df(t)$, the lower the weighting for the term.

TF-IDF can be also used together with n-grams. Similarly, to weight an n-gram, we compound the n-gram frequency in a certain document by the times the n-gram appears across all documents.

In this recipe, we will learn how to extract features using TF-IDF with or without n-grams using scikit-learn.

Getting ready

The scikit-learn implementation of TF-IDF uses a slightly different way to calculate the **IDF** statistic. For more details on the exact formula, visit the scikit-learn documentation: https://scikit-learn.org/stable/modules/feature_extraction.xhtml#tfidf-term-weighting.

TF-IDF shares the characteristics of BoW when creating the term matrix – that is, high feature space and sparsity. To reduce the number of features and sparsity, we can remove stop words, set the characters to lowercase, and retain words that appear in a minimum percentage of observations. If you are unfamiliar with these terms, visit the *Creating features with bag-of-words and n-grams* recipe in this chapter for a recap.

In this recipe, we will learn how to set words in lowercase, remove stop words, retain words with a minimum acceptable frequency, capture n-grams, and then return the TF-IDF statistic of words, all using a single transformer from scikit-learn: **TfidfVectorizer()**.

How to do it...

Let's begin by loading the necessary libraries and getting the dataset ready:

1. Load **pandas**, **TfidfVectorizer**, and the dataset from scikit-learn:

```
import pandas as pd
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import
TfidfVectorizer
```

2. Let's load the train set part of the **20 Newsgroup** dataset into a pandas DataFrame:

```
data = fetch_20newsgroups(subset='train')
df = pd.DataFrame(data.data, columns=['text'])
```

3. To make interpreting the results easier, let's remove punctuation and numbers from the text variable:

```
df['text'] = df['text'].str.replace(
    '[^\w\s]', '', regex=True).str.replace(
    '\d+', '', regex=True)
```

4. Now, let's set up **TfidfVectorizer()** from scikit-learn so that, before creating the TF-IDF metrics, it puts all text in lowercase, removes stop words, and retains words that appear in at least 5% of the text pieces:

```
vectorizer = TfidfVectorizer(
    lowercase=True,
    stop_words='english',
    ngram_range=(1, 1),
    min_df=0.05)
```

NOTE

To introduce n-grams as part of the returned columns, we can change the value of **ngrams_range** to, for example, **(1, 2)**. The tuple provides the lower and upper boundaries of the range of n-values for different n-grams to be extracted. In the case of **(1, 2)**, **TfidfVectorizer()** will return single words and arrays of two consecutive words as columns.

5. Let's fit **TfidfVectorizer()** so that it learns which words should be introduced as columns of the TF-IDF matrix:

```
vectorizer.fit(df['text'])
```

6. Now, let's create the TF-IDF matrix:

```
X = vectorizer.transform(df['text'])
```

7. Finally, let's capture the TF-IDF matrix in a DataFrame with the corresponding feature names:

```
tfidf = pd.DataFrame(  
    X.toarray(),  
    columns = vectorizer.get_feature_names_out()  
)
```

With that, we have created a pandas DataFrame that contains words as columns and the TF-IDF as values. You can inspect the result by executing **tfidf.head()**:

	able	access	actually	ago	apr	article	articleid	ask	available	away	...
0	0.0	0.000000	0.000000	0.0	0.0	0.000000	0.000000	0.0	0.0	0.0	...
works	world	writes	wrong	wrote	xnewsreader	year	years	yes	you're		
1	0.0	0.000000	0.000000	0.0	0.0	0.000000	0.356469	0.0	0.0	0.0	...
2	0.0	0.135765	0.123914	0.0	0.0	0.000000	0.000000	0.0	0.0	0.0	...
3	0.0	0.000000	0.000000	0.0	0.0	0.110035	0.000000	0.0	0.0	0.0	...
4	0.0	0.000000	0.000000	0.0	0.0	0.262692	0.000000	0.0	0.0	0.0	...

Figure 11.7 – DataFrame with features resulting from TF-IDF

Now, we can use this term frequency DataFrame to train machine learning models.

How it works...

In this recipe, we extracted the TF-IDF values of words present in at least 5% of the documents by utilizing `TfidfVectorizer()` from scikit-learn.

We loaded the **20 Newsgroup** text dataset from scikit-learn and then removed punctuation and numbers from the text rows using pandas `replace()`, which can be accessed through pandas `str`, to replace digits, '\d+', or symbols, '[^\w\s]', with empty strings, ''. Then, we used `TfidfVectorizer()` to create TF-IDF statistics for words. We set the `lowercase` parameter to `True` to put words in lowercase before making the calculations. We set the `stop_words` argument to `english` to avoid stop words in the returned matrix. We set `ngram_range` to the `(1, 1)` tuple to return single words as features. Finally, we set the `min_df` argument to `0.05` to return words that appear at least in 5% of the texts or, in other words, in 5% of the rows.

After setting up the transformer, we applied the `fit()` method to let the transformer find the words to retain in the final term matrix. With the `transform()` method, the transformer returned an object with the words and their TF-IDF values, which we then captured in a pandas DataFrame with the appropriate feature names. Now, we can use these features in machine learning algorithms.

See also

For more details on `TfidfVectorizer()`, visit the scikit-learn documentation at https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.xhtml.

Cleaning and stemming text variables

Some variables in our dataset can be created based on free text fields, which are manually completed by users. People have different writing styles, and we use a variety of punctuation marks, capitalization patterns, and verb conjugations to convey the content, as well as the emotions around it. We can extract information from text without taking the trouble to read it by creating statistical parameters that summarize the text's complexity, keywords, and relevance of words in a document. We discussed these methods in the preceding recipes of this chapter. Yet, to derive these statistics and aggregated features, we should clean the text variables first.

Text cleaning or text preprocessing involves punctuation removal, the elimination of stop words, character case setting, and word stemming. Punctuation removal consists of deleting characters that are not letters, numbers, or spaces; in some cases, we also remove numbers. The elimination of stop words refers to removing common words that are used in our language to allow for the sentence structure and flow, but that individually convey little or no information. Examples of stop words include articles such as **the** and **a** for the English language, as well as pronouns such as **I**, **you**, and **they**, and commonly used verbs in their various conjugations, such as the verbs **to be** and **to have**, as well as the auxiliary verbs **would** and **do**.

To allow computers to identify words correctly, it is also necessary to set all the words in the same case, since the words **Toy** and **toy** would be identified as different by a computer, due to the capital T in the first one. Finally, to focus on the **message** of the text and not count them as different words that convey similar meanings if it weren't for their conjugation, we may also want to introduce word stemming as part of the preprocessing pipeline. Word

stemming refers to reducing each word to its root or base so that the words **playing**, **plays**, and **played** become **play**, which, in essence, convey the same or very similar meaning.

In this recipe, we will learn how to remove punctuation and stop words, set words in lowercase, and perform word stemming with pandas and NLTK.

Getting ready

We are going to use the NLTK stem package to perform word stemming, which incorporates different algorithms to stem words from English and other languages. Each method differs in the algorithm it uses to find the root of the word; therefore, they may output slightly different results. I recommend that you read more about it, try different methods, and choose the one that serves the project you are working on.

More information about NLTK stemmers can be found here: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.xhtml.

How to do it...

Let's begin by loading the necessary libraries and getting the dataset ready:

1. Load **pandas**, **stopwords**, and **SnowballStemmer** from NLTK and the dataset from scikit-learn:

```
import pandas as pd
from nltk.corpus import stopwords
from nltk.stem.snowball import SnowballStemmer
from sklearn.datasets import fetch_20newsgroups
```

2. Let's load the train set part of the **20 Newsgroup** dataset into a pandas DataFrame:

```
data = fetch_20newsgroups(subset='train')
df = pd.DataFrame(data.data, columns=['text'])
```

Now, let's begin with the text cleaning.

NOTE

Print some example text with the **print(df['text'][10])** command, right after executing each line of code in this recipe, so that you can visualize the changes introduced to the text.

3. First, let's remove the punctuation:

```
df["text"] = df['text'].str.replace('[^\w\s]', '')
```

TIP

You can also remove the punctuation using the built-in **string** module from Python. First, import the module by executing **import string** and then execute **df['text'] = df['text'].str.replace('{}'.format(string.punctuation), '')**.

4. We can also remove characters that are numbers, leaving only letters, as follows:

```
df['text'] = df['text'].str.replace('\d+', '', regex=True)
```

5. Now, let's set all cases in lowercase:

```
df['text'] = df['text'].str.lower()
```

Now, let's remove stop words.

NOTE

Step 6 may fail if you did not download the **nltk** library **stopwords**. Visit the *Technical requirements* section for more details.

6. Let's create a function that splits a string into a list of words, then removes the stop words from the list if the words are within NLTK's English stop words list, and finally, concatenates the remaining words back into a string:

```
def remove_stopwords(text):
    stop = set(stopwords.words('english'))
    text = [word
        for word in text.split() if word not in stop]
    text = ' '.join(x for x in text)
    return text
```

NOTE

To be able to process the data with scikit-learn's **CountVectorizer()** or **TfidfVectorizer()**, we need the text to be in string format. Therefore, after removing the stop words, we need to return the words as a single string. We have transformed NLTK's stop words list into a set because sets are faster to scan than lists. This improves the computation time.

7. Now, let's use the function we created in *step 6* to remove stop words from the **text** variable:

```
df['text'] = df['text'].apply(remove_stopwords)
```

Finally, let's stem the words in our data. We will use **SnowballStemmer** from NLTK to do so.

8. Let's create an instance of **SnowballStemmer** for the English language:

```
stemmer = SnowballStemmer("english")
```

TIP

Try the stemmer in a single word to see how it works; for example, run `stemmer.stem('running')`. You should see `run` as the result of that command. Try different words!

9. Let's create a function that splits a string into a list of words, then applies `stemmer` to each word, and finally concatenates the stemmed word list back into a string:

```
def stemm_words(text):  
    text = [stemmer.stem(word) for word in  
text.split()]  
    text = ' '.join(x for x in text)  
    return text
```

10. Let's use the function we created in *step 9* to stem the words in our data:

```
df['text'] = df['text'].apply(stemm_words)
```

Now, our text is ready to create features based on character and word counts, as well as create BoWs or TF-IDF matrices, as described in the previous recipes of this chapter.

If we execute `print(df['text'][10])`, we will see a text example after cleaning:

```
irwincmptrclonestarorg irwin arnstein subject recommend duc  
summari what worth distribut usa expir sat may gmt organ  
computrac inc richardson tx keyword ducati gts much line  
line ducati gts model k clock run well paint  
bronzebrownorang fade leak bit oil pop st hard accel shop  
fix tran oil leak sold bike owner want think like k opinion  
pleas email thank would nice stabl mate beemer ill get jap  
bike call axi motor tuba irwin honk therefor  
computracrichardsontx irwincmptrclonestarorg dod r
```

NOTE

Note that the only feature that needs to be derived *before* removing punctuation is the count of the sentences, as punctuation and capitalization are needed to define the boundaries of each sentence.

How it works...

In this recipe, we removed punctuation, numbers, and stop words from a text variable, set the words in lowercase, and finally, stemmed the words to their root. We removed punctuation and numbers from the text variable using pandas **replace()**, which can be accessed through pandas **str**, to replace digits, '\d+', or symbols, '[^\w\s]', with empty strings, ''. Alternatively, we can use the **punctuation** module from the built-in **string** package.

TIP

Run **string.punctuation** in your Python console after importing **string** to visualize the symbols that will be replaced by empty strings.

Next, utilizing pandas string processing functionality through **str**, we set all of the words to lowercase with the **lower()** method. To remove stop words from the text, we used the **stopwords** module from NLTK, which contains a list of words that are considered frequent – that is, the stop words. We created a function that takes a string and splits it into a list of words using pandas **str.split()**, and then, with list comprehension, we looped over the words in the list and retained the non-stop words. Finally, with the **join()** method, we concatenated the retained words back into a string. We used the built-in Python **set()** method over the NLTK stop words list to improve computation efficiency since it is faster to iterate over sets than over lists. Finally, with pandas **apply()**, we applied the function to each row of our text data.

TIP

Run `stopwords.words('english')` in your Python console after importing `stopwords` from NLTK to visualize the list with the stop words that will be removed.

Finally, we stemmed the words using `SnowballStemmer` from NLTK. `stemmer` works one word at a time. Therefore, we created a function that takes a string and splits it into a list of words using pandas `str.split()`. In a list comprehension, we applied `stemmer` word per word, and then concatenated the list of stemmed words back into a string, using the `join()` method. With pandas `apply()`, we applied the function to stem words to each row of the DataFrame.

The cleaning steps we performed in this recipe resulted in strings containing the original text, without punctuation or numbers, in lowercase, without common words, and with the root of the word instead of its conjugated form. The data, as it is returned, can be used to derive features, as described in the *Counting characters, words, and vocabulary* recipe, or to create BoWs and TI-IDF matrices, as described in the *Creating features with bag-of-words and n-grams* and *Implementing term frequency-inverse document frequency* recipes.

Index

As this ebook edition doesn't have fixed pagination, the page numbers below are hyperlinked for reference only, based on the printed edition of this book.

Symbols

20 Newsgroups

reference link [332](#)

-Inf value [308](#)

A

aggregate primitive [267](#)

aggregation primitives

features, creating [289-295](#)

reference link [289](#)

arbitrary capper

reference link [168](#)

arbitrary intervals

variable, discretizing [128-132](#)

arbitrary number

used, for replacing missing values [16-19](#)

Asian Machine Learning Conference (ACML) [310](#)

automatic feature extraction

from time series [300-309](#)

AveOccup variable [86](#)

average occupancy [86](#)

average word length feature [335](#)

B

Bag-of-Words (BoW) [341](#)

reference link [139](#)

used, for creating features [341-344](#)

BinaryEncoder()

reference link [75](#)

binary encoding

performing [72-75](#)

reference link [75](#)

binary operations

reference link [227](#)

binary variable

creating, through one-hot encoding [36-43](#)

Box-Cox transformation [98](#)

performing 97-104

BoxCoxTransformer() 103

boxplots

used, for visualizing outliers 154-156

C

categorical encoding 33

categorical encoding techniques, for neural network classifiers

reference link 75

categorical variables

imputing 11-15

categories

replacing, with counts or frequency encoding 48

characterization methods 301

characters

counting 332-335

Complete Case Analysis (CCA) 3

continuous wavelet 309

counts or frequency encoding

used, for replacing categories 48

CountVectorizer() 351

reference link [345](#)

cumulative operations

features, creating [269-274](#)

cumulative transformation

reference link [272](#)

cumulative transform primitives [269](#)

cyclical features

reference link [245](#)

cyclical variables

used, for creating periodic features [238-245](#)

D

datetime properties

reference link [178](#)

datetime transform primitives [278](#)

reference link [281](#)

datetime variable

elapsed time, capturing between [182-185](#)

features, extracting [278-283](#)

date variable

features, extracting with pandas [172-177](#)

debt-to-income ratio [222](#)

decision tree

 features, combining [234-238](#)

 using, for discretization [144-151](#)

deep feature synthesis (dfs) [264](#)

discretization [109](#)

 decision tree, using [144-151](#)

 performing, with k-means clustering [133-138](#)

E

elbow method [138](#)

 reference link [138](#)

entity set [258](#)

 features automatically, creating [258-268](#)

 setting up [258-268](#)

EqualFrequencyDiscretiser() class [124](#)

equal-frequency discretization

 implementing [120-127](#)

EqualWidthDiscretiser() transformer [115](#)

equal-width discretization

 performing [110-120](#)

extreme values

finding, for imputation [19-22](#)

F

feature binarization

implementing [139-144](#)

feature creation

embedding, in scikit-learn pipeline [325-328](#)

tailoring, to different time series [315-320](#)

feature-engine

implementing [151](#)

used, for automating feature extraction [190-194](#)

feature extraction

automating, with feature-engine [190-194](#)

feature_names_in_ attribute

reference link [225](#)

features

creating, with Bag-of-Words (BoW) [341-344](#)

creating, with n-grams [34-344](#)

standardizing [196-200](#)

Featuretools

reference link [263](#)

G

general operations

features, creating [269-274](#)

general transformation

reference link [272](#)

general transform primitives [269](#)

gplearn package

reference link [234](#)

H

histograms

features, extracting from text [336, 337](#)

I

imputation [1](#)

imputed values

marking [22-26](#)

infrequent categories

grouping [69-72](#)

Inf value [308](#)

Inter-Quartile Range (IQR) [154](#), [202](#)

interquartile range proximity rule

used, for finding outliers [158-160](#)

K

k-means clustering

used, for performing discretization [133-138](#)

K-Nearest Neighbors (KNN) [30](#)

used, for estimating missing data [30](#), [31](#)

L

lexical diversity [334](#), [335](#)

list-wise deletion [3](#)

logarithm function

used, for transforming variables [78-84](#)

M

mathematical functions

features, combining [218-222](#)

reference link [222](#)

maximum absolute scaling

implementing [208-214](#)

maximum values

scaling [200-202](#)

mean compactness [223](#)

mean deviation

used, for finding outliers [157, 158](#)

mean imputation

performing [6-10](#)

mean normalization

performing [204-208](#)

median

scaling with [202-204](#)

median imputation

performing [6-10](#)

method summary

reference link [337](#)

minimum values

scaling [200-202](#)

missing data

estimating, with K-Nearest Neighbors (KNN) [30, 31](#)

used, for removing observations [3-6](#)

missing values

replacing, with arbitrary number [16-19](#)

Multivariate Imputation by Chained Equations (MICE) [27](#)

performing [27-29](#)

reference link [29](#)

N

NAN value [308](#)

Natural Language Processing (NLP) [284](#)

Natural Language Toolkit (NLTK) [341](#)

URL [341](#)

n-grams

used, for creating features [341-344](#)

NLTK stemmers

reference link [349](#)

number of characters [284](#)

number of words [284](#)

numerical features

combining [275-277](#)

O

observations

removing, with missing data [3-6](#)

one-hot encoding

performing [44-47](#)

used, for creating binary variable [36-43](#)

Online Retail II dataset [256](#)

ordinal encoding

performing, on target value [59-61](#)

outliers

capping [165-167](#)

capping, with quantiles [168-170](#)

censoring [165-167](#)

finding, with interquartile range proximity rule [158-160](#)

finding, with mean deviation [157, 158](#)

finding, with standard deviation [157](#)

removing [160-164](#)

visualizing, with boxplots [154-156](#)

P

pandas

methods [178](#)

used, for extracting date variable features [172-177](#)

used, for extracting time variable features [178-181](#)

pandas aggregate

reference link [222](#)

periodic features

creating, from cyclical variables [238-245](#)

polynomial expansion

performing [227-234](#)

`PolynomialFeatures()`

reference link [234](#)

power transformations

using [93-97](#)

`PowerTransformer()` [103](#)

pre-selected features

creating [320-324](#)

Q

Quantile-Quantile (Q-Q) plot [78](#)

quantiles

scaling with [202-204](#)

used, for capping outliers [168-170](#)

R

rare categories [69](#)
grouping [69-72](#)
reciprocal function
used, for transforming variables [85-89](#)
reference variables
features, comparing [222-226](#)
robust scaling [203](#)

S

scikit-learn dataset
reference link [332](#)
scikit-learn pipeline
feature creation, embedding [325-328](#)
sentence tokenization [338](#)
text complexity, estimating [338-341](#)
spline features
creating [245-254](#)
square root
using, for transforming variables [89-93](#)
standard deviation
used, for finding outliers [157, 158](#)

stop words [342](#), [350](#)

T

target mean encoding

implementing [61](#)-[66](#)

Term Frequency-Inverse Document Frequency (TF-IDF)

implementing [345](#)-[348](#)

text

used, for extracting text [284](#)-[288](#)

text complexity

estimating, by sentence tokenization [338](#)-[341](#)

text variables

cleaning [348](#)-[353](#)

stemming [348](#)-[353](#)

Tfidf term weighting

reference link [345](#)

TfidfVectorizer() [351](#)

reference link [348](#)

time/date components

reference link [178](#)

time series

automatic feature extraction [301-309](#)

features, creating [310-314](#)

features, selecting [310-314](#)

specific features, creating [315-320](#)

time since previous [269](#)

time variable

features, extracting with pandas [178-181](#)

working with, in time zones [186-189](#)

`to_datetime()` method

reference link [189](#)

transform primitives [267, 269, 289](#)

tsfresh documentation

reference link [320](#)

tsfresh library [301, 310](#)

tests, using [310](#)

`tz_convert()` method

reference link [189](#)

V

variable

discretizing, into arbitrary intervals [128-132](#)

transforming, with logarithm function [78-84](#)
transforming, with reciprocal function [85-89](#)
transforming, with square root [90-93](#)
variance stabilizing transformations [77](#)
variation coefficient [306](#)
vector unit length
scaling [214-216](#)
vocabulary
counting [332-335](#)

W

Weight of Evidence (WoE) [66](#)
encoding [66-69](#)
words
counting [332-335](#)
Workshop on Learning on Big Data (WLBD) [310](#)
worst compactness [223](#)

Y

Yeo-Johnson transformation
performing [104-107](#)
reference link [107](#)



[Packt.com](https://www.packt.com)

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePUB files available? You can upgrade to the eBook version at [packt.com](https://www.packt.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



packt

1ST EDITION

Data Cleaning and Exploration with Machine Learning

Get to grips with machine learning techniques to
achieve sparkling-clean data quickly



MICHAEL WALKER

Data Cleaning and Exploration with Machine Learning

Michael Walker

ISBN: 978-1-80324-167-8

- Explore essential data cleaning and exploration techniques to be used before running the most popular machine learning algorithms
- Understand how to perform preprocessing and feature selection, and how to set up the data for testing and validation
- Model continuous targets with supervised learning algorithms
- Model binary and multiclass targets with supervised learning algorithms
- Execute clustering and dimension reduction with unsupervised learning algorithms
- Understand how to use regression trees to model a continuous target

A large, abstract graphic at the top of the cover features a complex pattern of light-colored, faceted hexagonal shapes. Some of these facets are transparent or semi-transparent, revealing a vibrant, multi-colored interior composed of orange, yellow, green, and blue hues.

packt

1ST EDITION

Production-Ready Applied Deep Learning

Learn how to construct and deploy complex models in
PyTorch and TensorFlow deep learning frameworks



TOMASZ PALCZEWSKI
JAEJUN (BRANDON) LEE | LENIN MOOKIAH

Production-Ready Applied Deep Learning

Tomasz Palczewski, Jaejun (Brandon) Lee, Lenin Mookiah

ISBN: 978-1-80324-366-5

- Understand how to develop a deep learning model using PyTorch and TensorFlow
- Convert a proof-of-concept model into a production-ready application
- Discover how to set up a deep learning pipeline in an efficient way using AWS
- Explore different ways to compress a model for various deployment requirements
- Develop Android and iOS applications that run deep learning on mobile devices
- Monitor a system with a deep learning model in production
- Choose the right system architecture for developing and deploying a model

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *Python Feature Engineering Cookbook*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here to go straight to the Amazon review page](#) for this book and share your feedback or leave a review on the site that you purchased it from.

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

Download a Free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781804611302>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly