

Data Science Projects with Python

A case study approach to successful data science projects using Python, pandas, and scikit-learn



Stephen Klosterman

Packt

www.packt.com

Data Science Projects with Python

A case study approach to successful data science projects using Python, pandas, and scikit-learn

Stephen Klosterman

Packt

Data Science Projects with Python

Copyright © 2019 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, and its dealers and distributors will be held liable for any damages caused or alleged to be 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.

Author: Stephen Klosterman

Technical Reviewer: Richard Ackon

Managing Editor: Mahesh Dhyani

Acquisitions Editor: Koushik Sen

Production Editor: Samita Warang

Editorial Board: David Barnes, Ewan Buckingham, Simon Cox, Manasa Kumar, Alex Mazonowicz, Douglas Paterson, Dominic Pereira, Shiny Poojary, Erol Staveley, Ankita Thakur, Mohita Vyas, and Jonathan Wray

First Published: April 2019

Production Reference: 1300419

ISBN: 978-1-83855-102-5

Published by Packt Publishing Ltd.

Livery Place, 35 Livery Street

Birmingham B3 2PB, UK

Table of Contents

Preface	i
---------	---

Data Exploration and Cleaning	1
Introduction	2
Python and the Anaconda Package Management System	2
Indexing and the Slice Operator	3
Exercise 1: Examining Anaconda and Getting Familiar with Python	5
Different Types of Data Science Problems	9
Loading the Case Study Data with Jupyter and pandas	11
Exercise 2: Loading the Case Study Data in a Jupyter Notebook	14
Getting Familiar with Data and Performing Data Cleaning	17
The Business Problem	18
Data Exploration Steps	19
Exercise 3: Verifying Basic Data Integrity	20
Boolean Masks	25
Exercise 4: Continuing Verification of Data Integrity	27
Exercise 5: Exploring and Cleaning the Data	32
Data Quality Assurance and Exploration	38
Exercise 6: Exploring the Credit Limit and Demographic Features	39
Deep Dive: Categorical Features	44
Exercise 7: Implementing OHE for a Categorical Feature	47
Exploring the Financial History Features in the Dataset	51
Activity 1: Exploring Remaining Financial Features in the Dataset	61
Summary	62

Introduction to Scikit-Learn and Model Evaluation

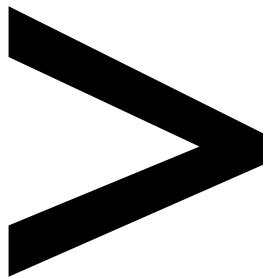
Introduction	66
Exploring the Response Variable and Concluding the Initial Exploration	66
Introduction to Scikit-Learn	69
Generating Synthetic Data	75
Data for a Linear Regression	76
Exercise 8: Linear Regression in Scikit-Learn	78
Model Performance Metrics for Binary Classification	81
Splitting the Data: Training and Testing sets	82
Classification Accuracy	85
True Positive Rate, False Positive Rate, and Confusion Matrix	88
Exercise 9: Calculating the True and False Positive and Negative Rates and Confusion Matrix in Python	90
Discovering Predicted Probabilities: How Does Logistic Regression Make Predictions?	94
Exercise 10: Obtaining Predicted Probabilities from a Trained Logistic Regression Model	94
The Receiver Operating Characteristic (ROC) Curve	100
Precision	104
Activity 2: Performing Logistic Regression with a New Feature and Creating a Precision-Recall Curve	105
Summary	106
Details of Logistic Regression and Feature Exploration	109
Introduction	110
Examining the Relationships between Features and the Response	110
Pearson Correlation	114
F-test	118

Exercise 11: F-test and Univariate Feature Selection	119
Finer Points of the F-test: Equivalence to t-test for Two Classes and Cautions	124
Hypotheses and Next Steps	125
Exercise 12: Visualizing the Relationship between Features and Response	126
Univariate Feature Selection: What It Does and Doesn't Do	134
Understanding Logistic Regression with function Syntax in Python and the Sigmoid Function	134
Exercise 13: Plotting the Sigmoid Function	138
Scope of Functions	142
Why is Logistic Regression Considered a Linear Model?	144
Exercise 14: Examining the Appropriateness of Features for Logistic Regression	147
From Logistic Regression Coefficients to Predictions Using the Sigmoid	151
Exercise 15: Linear Decision Boundary of Logistic Regression	153
Activity 3: Fitting a Logistic Regression Model and Directly Using the Coefficients	162
Summary	163
The Bias-Variance Trade-off	165

Introduction	166
Estimating the Coefficients and Intercepts of Logistic Regression	166
Gradient Descent to Find Optimal Parameter Values	168
Exercise 16: Using Gradient Descent to Minimize a Cost Function	172
Assumptions of Logistic Regression	177
The Motivation for Regularization: The Bias-Variance Trade-off	181
Exercise 17: Generating and Modeling Synthetic Classification Data	184
Lasso (L1) and Ridge (L2) Regularization	188

Cross Validation: Choosing the Regularization Parameter and Other Hyperparameters	194
Exercise 18: Reducing Overfitting on the Synthetic Data Classification Problem	200
Options for Logistic Regression in Scikit-Learn	212
Scaling Data, Pipelines, and Interaction Features in Scikit-Learn	215
Activity 4: Cross-Validation and Feature Engineering with the Case Study Data	217
Summary	219
Decision Trees and Random Forests	223
Introduction	224
Decision trees	224
The Terminology of Decision Trees and Connections to Machine Learning	225
Exercise 19: A Decision Tree in scikit-learn	227
Training Decision Trees: Node Impurity	236
Features Used for the First splits: Connections to Univariate Feature Selection and Interactions	240
Training Decision Trees: A Greedy Algorithm	241
Training Decision Trees: Different Stopping Criteria	241
Using Decision Trees: Advantages and Predicted Probabilities	246
A More Convenient Approach to Cross-Validation	248
Exercise 20: Finding Optimal Hyperparameters for a Decision Tree	251
Random Forests: Ensembles of Decision Trees	258
Random Forest: Predictions and Interpretability	261
Exercise 21: Fitting a Random Forest	262
Checkerboard Graph	268
Activity 5: Cross-Validation Grid Search with Random Forest	270
Summary	271

Imputation of Missing Data, Financial Analysis, and Delivery to Client	275
Introduction	276
Review of Modeling Results	276
Dealing with Missing Data: Imputation Strategies	278
Preparing Samples with Missing Data	281
Exercise 22: Cleaning the Dataset	281
Exercise 23: Mode and Random Imputation of PAY_1	286
A Predictive Model for PAY_1	294
Exercise 24: Building a Multiclass Classification Model for Imputation ...	296
Using the Imputation Model and Comparing it to Other Methods	302
Confirming Model Performance on the Unseen Test Set	307
Financial Analysis	308
Financial Conversation with the Client	310
Exercise 25: Characterizing Costs and Savings	311
Activity 6: Deriving Financial Insights	317
Final Thoughts on Delivering the Predictive Model to the Client	318
Summary	321
Appendix	323
Index	355



Preface

About

This section briefly introduces the author, the coverage of this book, the technical skills you'll need to get started, and the hardware and software required to complete all of the included activities and exercises.

About the Book

Data Science Projects with Python is designed to give you practical guidance on industry-standard data analysis and machine learning tools in Python, with the help of realistic data. The book will help you understand how you can use pandas and Matplotlib to critically examine a dataset with summary statistics and graphs, and extract meaningful insights. You will continue to build on your knowledge as you learn how to prepare data and feed it to machine learning algorithms, such as regularized logistic regression and random forest, using the scikit-learn package. You'll discover how to tune algorithms to provide the best predictions on new, unseen data. As you delve into later chapters, you'll gain an understanding of the workings and output of these algorithms, and gain insight into not only the predictive capabilities of the models but also the way they make predictions.

About the Author

Stephen Klosterman is a machine learning data scientist at CVS Health. He enjoys helping to frame problems in a data science context and delivering machine learning solutions that business stakeholders understand and value. His education includes a Ph.D. in biology from Harvard University, where he was an assistant teacher of the data science course.

Objectives

- Install the required packages to set up a data science coding environment
- Load data into a Jupyter Notebook running Python
- Use Matplotlib to create data visualizations
- Fit a model using scikit-learn
- Use lasso and ridge regression to reduce overfitting
- Fit and tune a random forest model and compare performance with logistic regression
- Create visuals using the output of the Jupyter Notebook

Audience

If you are a data analyst, data scientist, or a business analyst who wants to get started with using Python and machine learning techniques to analyze data and predict outcomes, this book is for you. Basic knowledge of computer programming and data analytics is a must. Familiarity with mathematical concepts such as algebra and basic statistics will be useful.

Approach

Data Science Projects with Python takes a case study approach to simulate the working conditions you will experience when applying data science and machine learning concepts. You will be presented with a problem and a dataset, and walked through the steps of defining an answerable question, deciding what analysis methods to use, and implementing all of this in Python to create a deliverable.

Hardware Requirements

For the optimal student experience, we recommend the following hardware configuration:

- Processor: Intel Core i5 or equivalent
- Memory: 4 GB RAM
- Storage: 35 GB available space

Software Requirements

You'll also need the following software installed in advance:

- OS: Windows 7 SP1 64-bit, Windows 8.1 64-bit or Windows 10 64-bit, Ubuntu Linux, or the latest version of OS X
- Browser: Google Chrome/Mozilla Firefox Latest Version
- Notepad++/Sublime Text as IDE (this is optional, as you can practice everything using the Jupyter Notebook on your browser)
- Python 3.4+ (latest is Python 3.7) installed (from <https://python.org>)
- Python libraries as needed (Jupyter, NumPy, Pandas, Matplotlib, and so on)

Installation and Setup

Before you start this book, make sure you have installed the Anaconda environment as we will be using the Anaconda distribution of Python.

Installing Anaconda

Install Anaconda by following the instructions at this link: <https://www.anaconda.com/distribution/>

Additional Resources

The code bundle for this book is hosted on GitHub at <https://github.com/TrainingByPackt/Data-Science-Projects-with-Python>.

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

Conventions

Code words in the text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "By typing `conda list` at the command line, you can see all the packages installed in your environment."

A block of code is set as follows:

```
import numpy as np #numerical computation
import pandas as pd #data wrangling
import matplotlib.pyplot as plt #plotting package
#Next line helps with rendering plots
%matplotlib inline
import matplotlib as mpl #add'l p-lotting functionality
mpl.rcParams['figure.dpi'] = 400 #high res figures
import graphviz #to visualize decision trees
```

New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Create a new Python 3 notebook from the **New** menu as shown."

1

Data Exploration and Cleaning

Learning Objectives

By the end of this chapter, you will be able to:

- Perform basic operations in Python
- Describe the business context of the case study data and its suitability for the task
- Perform data cleaning operations
- Examine statistical summaries and visualize the case study data
- Implement one-hot encoding on categorical variables

This chapter will get you started with basic operations in Python and shows you how to perform data-related operations such as data verification, data cleaning, datatype conversion, examining statistical summaries, and more.

Introduction

Most businesses possess a wealth of data on their operations and customers. Reporting on these data in the form of descriptive charts, graphs, and tables is a good way to understand the current state of the business. However, in order to provide quantitative guidance on future business strategies and operations, it is necessary to go a step further. This is where the practices of machine learning and predictive modeling become involved. In this book, we will show how to go from descriptive analyses to concrete guidance for future operations using predictive models.

To accomplish this goal, we'll introduce some of the most widely-used machine learning tools via Python and many of its packages. You will also get a sense of the practical skills necessary to execute successful projects: inquisitiveness when examining data and communication with the client. Time spent looking in detail at a dataset and critically examining whether it accurately meets its intended purpose is time well spent. You will learn several techniques for assessing data quality here.

In this chapter, after getting familiar with the basic tools for data exploration, we will discuss a few typical working scenarios for how you may receive data. Then, we will begin a thorough exploration of the case study dataset and help you learn how you can uncover possible issues, so that when you are ready for modeling, you may proceed with confidence.

Python and the Anaconda Package Management System

In this book, we will use the Python programming language. Python is a top language for data science and is one of the fastest growing programming languages. A commonly cited reason for Python's popularity is that it is easy to learn. If you have Python experience, that's great; however, if you have experience with other languages, such as C, Matlab, or R, you shouldn't have much trouble using Python. You should be familiar with the general constructs of computer programming to get the most out of this book. Examples of such constructs are **for** loops and **if** statements that guide the **control flow** of a program. No matter what language you have used, you are likely familiar with these constructs, which you will also find in Python.

A key feature of Python, that is different from some other languages, is that it is zero-indexed; in other words, the first element of an ordered collection has an index of 0. Python also supports negative indexing, where index-1 refers to the last element of an ordered collection and negative indices count backwards from the end. The slice operator, `:`, can be used to select multiple elements of an ordered collection from within a range, starting from the beginning, or going to the end of the collection.

Indexing and the Slice Operator

Here, we demonstrate how indexing and the slice operator work. To have something to index, we will create a **list**, which is a **mutable** ordered collection that can contain any type of data, including numerical and string types. "Mutable" just means the elements of the list can be changed after they are first assigned. To create the numbers for our list, which will be consecutive integers, we'll use the built-in **range()** Python function. The **range()** function technically creates an **iterator** that we'll convert to a list using the **list()** function, although you need not be concerned with that detail here. The following screenshot shows a basic list being printed on the console:

```
>>> example_list = list(range(1,6))
>>> example_list
[1, 2, 3, 4, 5]
>>> example_list[0]
1
>>> example_list[-1]
5
>>> example_list[-2]
4
>>> example_list[:3]
[1, 2, 3]
>>> example_list[0] = 'a string'
>>> example_list
['a string', 2, 3, 4, 5]
>>> █
```

Figure 1.1: List creation and indexing

A few things to notice about Figure 1.1: the endpoint of an interval is open for both slice indexing and the **range()** function, while the starting point is closed. In other words, notice how when we specify the start and end of **range()**, endpoint 6 is not included in the result but starting point 1 is. Similarly, when indexing the list with the slice **[:3]**, this includes all elements of the list with indices up to, but not including, 3.

We've referred to ordered collections, but Python also includes unordered collections. An important one of these is called a **dictionary**. A dictionary is an unordered collection of **key:value** pairs. Instead of looking up the values of a dictionary by integer indices, you look them up by keys, which could be numbers or strings. A dictionary can be created using curly braces {} and with the **key:value** pairs separated by commas. The following screenshot is an example of how we can create a dictionary with counts of fruit – examine the number of apples, then add a new type of fruit and its count:

```
>>> example_dict = {'apples':5, 'oranges':8}
>>> example_dict['apples']
5
>>> example_dict['bananas'] = 13
>>> example_dict
{'apples': 5, 'oranges': 8, 'bananas': 13}
>>>
```

Figure 1.2: An example dictionary

There are many other distinctive features of Python and we just want to give you a flavor here, without getting into too much detail. In fact, you will probably use packages such as **pandas** (**pandas**) and **NumPy** (**numpy**) for most of your data handling in Python. NumPy provides fast numerical computation on arrays and matrices, while pandas provides a wealth of data wrangling and exploration capabilities on tables of data called **DataFrames**. However, it's good to be familiar with some of the basics of Python—the language that sits at the foundation of all of this. For example, indexing works the same in NumPy and pandas as it does in Python.

One of the strengths of Python is that it is open source and has an active community of developers creating amazing tools. We will use several of these tools in this book. A potential pitfall of having open source packages from different contributors is the dependencies between various packages. For example, if you want to install pandas, it may rely on a certain version of NumPy, which you may or may not have installed. Package management systems make life easier in this respect. When you install a new package through the package management system, it will ensure that all the dependencies are met. If they aren't, you will be prompted to upgrade or install new packages as necessary.

For this book, we will use the **Anaconda** package management system, which you should already have installed. While we will only use Python here, it is also possible to run R with Anaconda.

Note

Environments

If you previously had Anaconda installed and were using it prior to this book, you may wish to create a new Python 3.x environment for the book. Environments are like separate installations of Python, where the set of packages you have installed can be different, as well as the version of Python. Environments are useful for developing projects that need to be deployed in different versions of Python. For more information, see <https://conda.io/docs/user-guide/tasks/manage-environments.html>.

Exercise 1: Examining Anaconda and Getting Familiar with Python

In this exercise, you will examine the packages in your Anaconda installation and practice some basic Python control flow and data structures, including a **for** loop, **dict**, and **list**. This will confirm that you have completed the installation steps in the preface and show you how Python syntax and data structures may be a little different from other programming languages you may be familiar with. Perform the following steps to complete the exercise:

Note

The code file for this exercise can be found here: <http://bit.ly/2Oyag1h>.

1. Open up a Terminal, if you're using macOS or Linux, or a Command Prompt window in Windows. Type `conda list` at the command line. You should observe an output similar to the following:

```
requests           2.21.0          py37_0
rope              0.11.0          py37_0
ruamel_yaml       0.15.46         py37h1de35cc_0
scikit-image      0.14.1          py37h0a44026_0
scikit-learn      0.20.1          py37h27c97d8_0
scipy              1.1.0           py37h1410ff5_2
seaborn            0.9.0           py37_0
send2trash         1.5.0           py37_0
setuptools         40.6.3          py37_0
simplegeneric     0.8.1           py37_2
singledispatch    3.4.0.3         py37_0
sip                4.19.8          py37h0a44026_0
six                1.12.0          py37_0
snappy             1.1.7           he62c110_3
snowballstemmer   1.2.1           py37_0
```

Figure 1.3: Selection of packages from conda list

You can see all the packages installed in your environment. Look how many packages already come with the default Anaconda installation! These include all the packages we will need for the book. However, installing new packages and upgrading existing ones is easy and straightforward with Anaconda; this is one of the main advantages of a package management system.

2. Type `python` in the Terminal to open a command-line Python interpreter. You should obtain an output similar to the following:

```
Python 3.7.1 (default, Dec 14 2018, 13:28:58)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

Figure 1.4: Command-line Python

You should also see some information about your version of Python, as well as the Python command prompt (`>>>`). When you type after this prompt, you are writing Python code.

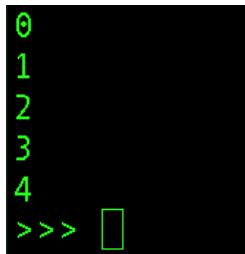
Note

Although we will be using the Jupyter Notebook in this book, one of the aims of this exercise is to go through the basic steps of writing and running Python programs on the command prompt.

3. Write a **for** loop at the command prompt to print values from 0 to 4 using the following code:

```
for counter in range(5):  
...     print(counter)  
...
```

Once you hit Enter when you see (...) on the prompt, you should obtain this output:



A terminal window showing the output of a Python script. The output consists of five lines of green text on a black background, each containing a number from 0 to 4. After the numbers, there is a prompt '>>>'. The entire terminal window is enclosed in a black border.

```
0  
1  
2  
3  
4  
>>> □
```

Figure 1.5: Output of a for loop at the command line

Notice that in Python, the opening of the **for** loop is followed by a colon, and **the body of the loop requires indentation**. It's typical to use four spaces to indent a code block. Here, the **for** loop prints the values returned by the **range()** iterator, having repeatedly accessed them using the **counter** variable with the **in** keyword.

Note

For many more details on Python code conventions, refer to the following: <https://www.python.org/dev/peps/pep-0008/>.

Now, we will return to our dictionary example. The first step here is to create the dictionary.

4. Create a dictionary of fruits (**apples**, **oranges**, and **bananas**) using the following code:

```
example_dict = {'apples':5, 'oranges':8, 'bananas':13}
```

5. Convert the dictionary to a list using the **list()** function, as shown in the following snippet:

```
dict_to_list = list(example_dict)  
dict_to_list
```

Once you run the preceding code, you should obtain the following output:

```
['apples', 'oranges', 'bananas']
```

Figure 1.6: Dictionary keys converted to a list

Notice that when this is done and we examine the contents, only the keys of the dictionary have been captured in the list. If we wanted the values, we would have had to specify that with the `.values()` method of the list. Also, notice that the list of dictionary keys happens to be in the same order that we wrote them in when creating the dictionary. This is not guaranteed, however, as dictionaries are unordered collection types.

One convenient thing you can do with lists is to append other lists to them with the `+` operator. As an example, in the next step we will combine the existing list of fruit with a list that contains just one more type of fruit, overwriting the variable containing the original list.

6. Use the `+` operator to combine the existing list of fruits with a new list containing only one fruit (`pears`):

```
dict_to_list = dict_to_list + ['pears']  
dict_to_list
```

```
['apples', 'oranges', 'bananas', 'pears']
```

Figure 1.7: Appending to a list

What if we wanted to sort our list of fruit types?

Python provides a built-in `sorted()` function that can be used for this; it will return a sorted version of the input. In our case, this means the list of fruit types will be sorted alphabetically.

7. Sort the list of fruits in alphabetical order using the `sorted()` function, as shown in the following snippet:

```
sorted(dict_to_list)
```

Once you run the preceding code, you should see the following output:

```
['apples', 'bananas', 'oranges', 'pears']
```

Figure 1.8: Sorting a list

That's enough Python for now. We will show you how to execute the code for this book, so your Python knowledge should improve along the way.

Note

As you learn more and inevitably want to try new things, you will want to consult the documentation: <https://docs.python.org/3/>.

Different Types of Data Science Problems

Much of your time as a data scientist is likely to be spent wrangling data: figuring out how to get it, getting it, examining it, making sure it's correct and complete, and joining it with other types of data. pandas will facilitate this process for you. However, if you aspire to be a machine learning data scientist, you will need to master the art and science of **predictive modeling**. This means using a mathematical model, or idealized mathematical formulation, to learn the relationships within the data, in the hope of making accurate and useful predictions when new data comes in.

For this purpose, data is typically organized in a tabular structure, with **features** and a **response variable**. For example, if you want to predict the price of a house based on some characteristics about it, such as **area** and **number of bedrooms**, these attributes would be considered the features and the **price of the house** would be the response variable. The response variable is sometimes called the **target variable** or **dependent variable**, while the features may also be called the **independent variables**.

If you have a dataset of 1,000 houses including the values of these features and the prices of the houses, you can say you have 1,000 **samples of labeled** data, where the labels are the known values of the response variable: the prices of different houses. Most commonly, the tabular data structure is organized so that different rows are different samples, while features and the response occupy different columns, along with other metadata such as sample IDs, as shown in *Figure 1.9*.

House ID	Area (i.e. m ²)	Number of bedrooms	Price (\$)
1	1,500	3	200,000
2	2,500	5	600,000
3	2,000	3	500,000

Figure 1.9: Labeled data (the house prices are the known target variable)

Regression Problem

Once you have trained a model to learn the relationship between the features and response using your labeled data, you can then use it to make predictions for houses where you don't know the price, based on the information contained in the features. The goal of predictive modeling in this case is to be able to make a prediction that is close to the true value of the house. Since we are predicting a numerical value on a continuous scale, this is called a **regression problem**.

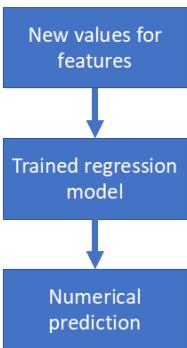
Classification Problem

On the other hand, if we were trying to make a qualitative prediction about the house, to answer a **yes** or **no** question such as "will this house go on sale within the next five years?" or "will the owner default on the mortgage?", we would be solving what is known as a **classification problem**. Here, we would hope to answer the yes or no question correctly. The following figure is a schematic illustrating how model training works, and what the outcomes of regression or classification models might be:

Model training phase

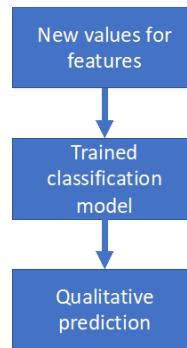


Prediction phase: regression



For example,
predict the price
of a house.
Hopefully get
close to the
actual price.

Prediction phase: classification



For example,
predict the
answer of a
yes/no question.
Hopefully get it
right.

Figure 1.10: Schematic of model training and prediction for regression and classification

Classification and regression tasks are called **supervised learning**, which is a class of problems that relies on labeled data. These problems can be thought of as needing "supervision" by the known values of the target variable. By contrast, there is also **unsupervised learning**, which relates to more open-ended questions of trying to find some sort of structure in a dataset that does not necessarily have labels. Taking a broader view, any kind of applied math problem, including fields as varied as **optimization**, **statistical inference**, and **time series modeling**, may potentially be considered an appropriate responsibility for a data scientist.

Loading the Case Study Data with Jupyter and pandas

Now it's time to take a first look at the data we will use in our case study. We won't do anything in this section other than ensure that we can load the data into a **Jupyter Notebook** correctly. Examining the data, and understanding the problem you will solve with it, will come later.

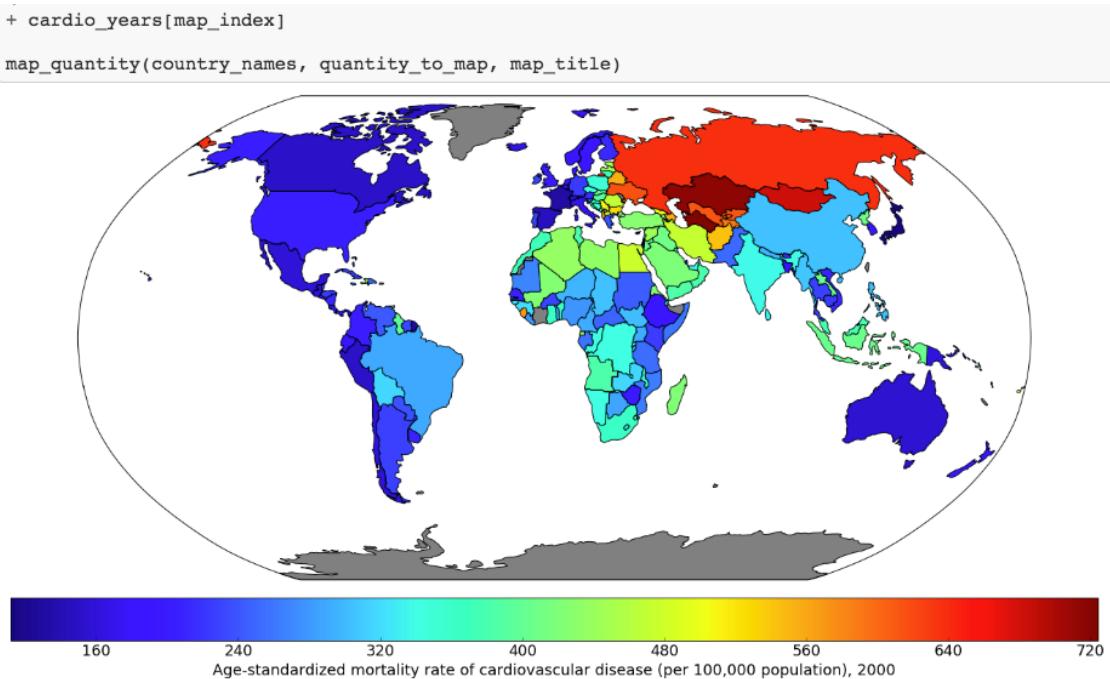
The data file is an Excel spreadsheet called **default_of_credit_card_clients** **courseware_version_1_13_19.xls**. We recommend you first open the spreadsheet in Excel, or the spreadsheet program of your choice. Note the number of rows and columns, and look at some example values. This will help you know whether or not you have loaded it correctly in the Jupyter Notebook.

Note

The dataset can be obtained from the following link: <http://bit.ly/2Hlk5t3>. This is a modified version of the original dataset, which has been sourced from the UCI Machine Learning Repository [<http://archive.ics.uci.edu/ml>]. Irvine, CA: University of California, School of Information and Computer Science.

What is a Jupyter Notebook?

Jupyter Notebooks are interactive coding environments that allow for in-line text and graphics. They are great tools for data scientists to communicate and preserve their results, since both the methods (code) and the message (text and graphics) are integrated. You can think of the environment as a kind of webpage where you can write and execute code. Jupyter Notebooks can, in fact, be rendered as web pages and are done so on GitHub. Here is one of our example notebooks: <http://bit.ly/2OvndJg>. Look it over and get a sense of what you can do. An excerpt from this notebook is displayed here, showing code, graphics, and prose, known as **markdown** in this context:



Northern Central Asia, Russia, and parts of Eastern Europe seem to have the highest incidences. The Middle East and much of Africa have moderately high but variable rates, while Western Europe, the Americas, Australia, and Eastern Asian countries have relatively low rates of cardiovascular disease.

Figure 1.11: Example of a Jupyter Notebook showing code, graphics, and markdown text

One of the first things to learn about Jupyter Notebooks is how to navigate around and make edits. There are two modes available to you. If you select a cell and press **Enter**, you are in **edit mode** and you can edit the text in that cell. If you press **Esc**, you are in **command mode** and you can navigate around the notebook.

When you are in command mode, there are many useful hotkeys you can use. The Up and Down arrows will help you select different cells and scroll through the notebook. If you press `y` on a selected cell in command mode, it changes it to a **code cell**, in which the text is interpreted as code. Pressing `m` changes it to a **markdown cell**. `Shift + Enter` evaluates the cell, rendering the markdown or executing the code, as the case may be.

Our first task in our first Jupyter Notebook will be to load the case study data. To do this, we will use a tool called **pandas**. It is probably not a stretch to say that pandas is the pre-eminent data-wrangling tool in Python.

A DataFrame is a foundational class in pandas. We'll talk more about what a class is later, but you can think of it as a template for a data structure, where a data structure is something like the lists or dictionaries we discussed earlier. However, a DataFrame is much richer in functionality than either of these. A DataFrame is similar to spreadsheets in many ways. There are rows, which are labeled by a row index, and columns, which are usually given column header-like labels that can be thought of as a column index. **Index** is, in fact, a data type in pandas used to store indices for a DataFrame, and columns have their own data type called **Series**.

You can do a lot of the same things with a DataFrame that you can do with Excel sheets, such as creating pivot tables and filtering rows. pandas also includes SQL-like functionality. You can join different DataFrames together, for example. Another advantage of DataFrames is that once your data is contained in one of them, you have the capabilities of a wealth of pandas functionality at your fingertips. The following figure is an example of a pandas DataFrame:

	ID	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_1
0	b730d678-5446	20000	2	2	1	24	2
1	99a3ae70-57a8	120000	2	2	2	26	-1
2	90194006-f94a	90000	2	2	2	34	0
3	19acf9a3-2b01	50000	2	2	1	37	0
4	70d7bc16-a6a4	50000	1	2	1	57	-1

Figure 1.12: Example of a pandas DataFrame with an integer row index at the left and a column index of strings

The example in *Figure 1.12* is in fact the data for the case study. As a first step with Jupyter and pandas, we will now see how to create a Jupyter Notebook and load data with pandas. There are several convenient functions you can use in pandas to explore your data, including `.head()` to see the first few rows of the DataFrame, `.info()` to see all columns with datatypes, `.columns` to return a list of column names as strings, and others we will learn about in the following exercises.

Exercise 2: Loading the Case Study Data in a Jupyter Notebook

Now that you've learned about Jupyter Notebooks, the environment in which we'll write code, and pandas, the data wrangling package, let's create our first Jupyter Notebook. We'll use pandas within this notebook to load the case study data and briefly examine it. Perform the following steps to complete the exercise:

Note

For Exercises 2–5 and Activity 1, the code and the resulting output have been loaded in a Jupyter Notebook that can be found at <http://bit.ly/2W9cwPH>. You can scroll to the appropriate section within the Jupyter Notebook to locate the exercise or activity of choice.

1. Open a Terminal (macOS or Linux) or a Command Prompt window (Windows) and type `jupyter notebook`.
You will be presented with the Jupyter interface in your web browser. If the browser does not open automatically, copy and paste the URL from the terminal in to your browser. In this interface, you can navigate around your directories starting from the directory you were in when you launched the notebook server.
2. Navigate to a convenient location where you will store the materials for this book, and create a new Python 3 notebook from the **New** menu, as shown here:



Figure 1.13: Jupyter home screen

3. Make your very first cell a markdown cell by typing `m` while in command mode (press Esc to enter command mode), then type a number sign, `#`, at the beginning of the first line, followed by a space, for a heading. Make a title for your notebook here. On the next few lines, place a description.

Here is a screenshot of an example, including other kinds of markdown such as bold, italics, and the way to write code-style text in a markdown cell:

First Jupyter notebook

Welcome to your first jupyter notebook! The first thing to know about Jupyter notebooks is that there are two kinds of cells. This is a markdown cell.

There are a lot of different ways to mark up the text in markdown cells, including `bold` and `*italics*`.

The next one will be a ``code`` cell.

Figure 1.14: Unrendered markdown cell

Note that it is good practice to make a title and brief description of your notebook, to identify its purpose to readers.

4. Press Shift + Enter to render the markdown cell.

This should also create a new cell, which will be a code cell. You can change it to a markdown cell, as you now know how to do, by pressing `m`, and back to a code cell by pressing `y`. You will know it's a code cell because of the `In []:` next to it.

5. Type `import pandas as pd` in the new cell, as shown in the following screenshot:

First Jupyter notebook

Welcome to your first jupyter notebook! The first thing to know about Jupyter notebooks is that there are two kinds of cells. This is a markdown cell.

There are a lot of different ways to mark up the text in markdown cells, including `bold` and `italics`.

The next one will be a `code` cell.

In []: `import pandas as pd`

Figure 1.15: Rendered markdown cell and code cell

After you execute this cell, the pandas library will be loaded into your computing environment. It's common to import libraries with "as" to create a short alias for the library. Now, we are going to use pandas to load the data file. It's in Microsoft Excel format, so we can use `pd.read_excel`.

Note

For more information on all the possible options for `pd.read_excel`, refer to the following documentation: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_excel.html.

6. Import the dataset, which is in the Excel format, as a DataFrame using the `pd.read_excel()` method, as shown in the following snippet:

```
df = pd.read_excel('..../Data/default_of_credit_card_clients_courseware_version_1_21_19.xls')
```

Note that you need to point the Excel reader to wherever the file is located. If it's in the same directory as your notebook, you could just enter the filename. The `pd.read_excel` method will load the Excel file into a **DataFrame**, which we've called `df`. The power of pandas is now available to us.

Let's do some quick checks in the next few steps. First, do the numbers of rows and columns match what we know from looking at the file in Excel?

7. Use the `.shape` method to review the number of rows and columns, as shown in the following snippet:

```
df.shape
```

Once you run the cell, you will obtain the following output:

```
In [3]: df.shape  
Out[3]: (30000, 25)
```

Figure 1.16: Checking the shape of a DataFrame

This should match your observations from the spreadsheet. If it doesn't, you would then need to look into the various options of `pd.read_excel` to see if you needed to adjust something.

With this exercise, we have successfully loaded our dataset into the Jupyter Notebook. You can also have a look at the `.info()` and `.head()` methods, which will tell you information about all the columns, and show you the first few rows of the `DataFrame`, respectively. Now you're up and running with your data in pandas.

As a final note, while this may already be clear, observe that if you define a variable in one code cell, it is available to you in other code cells within the notebook. The code cells within a notebook share scope as long as the kernel is running, as shown in the following screenshot:

The screenshot shows three code cells in a Jupyter Notebook. Cell In [4] contains the assignment `a = 5`. Cell In [5] contains the variable `a`. Cell Out[5] shows the output `5`, indicating that the variable `a` defined in cell [4] is accessible in cell [5].

```
In [4]: a = 5
In [5]: a
Out[5]: 5
```

Figure 1.17: Variable in scope between cells

Getting Familiar with Data and Performing Data Cleaning

Now let's imagine we are taking our first look at this data. In your work as a data scientist, there are several possible scenarios in which you may receive such a dataset. These include the following:

1. You created the SQL query that generated the data.
2. A colleague wrote a SQL query for you, with your input.
3. A colleague who knows about the data gave it to you, but without your input.
4. You are given a dataset about which little is known.

In cases 1 and 2, your input was involved in generating/extracting the data. In these scenarios, you probably understood the business problem and then either found the data you needed with the help of a data engineer, or did your own research and designed the SQL query that generated the data. Often, especially as you gain more experience in your data science role, the first step will be to meet with the business partner to understand, and refine the mathematical definition of, the business problem. Then, you would play a key role in defining what is in the dataset.

Even if you have a relatively high level of familiarity with the data, doing data exploration and looking at **summary statistics** of different variables is still an important first step. This step will help you select good features, or give you ideas about how you can engineer new features. However, in the third and fourth cases, where your input was not involved or you have little knowledge about the data, data exploration is even more important.

Another important initial step in the data science process is examining the **data dictionary**. The data dictionary, as the term implies, is a document that explains what the data owner thinks should be in the data, such as definitions of the column labels. It is the data scientist's job to go through the data carefully to make sure that these impressions match the reality of what is in the data. In cases 1 and 2, you will probably need to create the data dictionary yourself, which should be considered essential project documentation. In cases 3 and 4, you should seek out the dictionary if at all possible.

The case study data we'll use in this book is basically similar to case 3 here.

The Business Problem

Our client is a credit card company. They have brought us a dataset that includes some demographics and recent financial data (the past six months) for a sample of 30,000 of their account holders. This data is at the credit account level; in other words, there is one row for each account (you should always clarify what the definition of a row is, in a dataset). Rows are labeled by whether in the next month after the six month historical data period, an account owner has defaulted, or in other words, failed to make the minimum payment.

Goal

Your goal is to develop a predictive model for whether an account will default next month, given demographics and historical data. Later in the book, we'll discuss the practical application of the model.

The data is already prepared, and a data dictionary is available. The dataset supplied with the book, **default of credit card clients courseware version 1 21 19.xls**, is a modified version of this dataset in the UCI Machine Learning Repository: <https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients>. Have a look at that web page, which includes the data dictionary.

Note

The original dataset has been obtained from UCI Machine Learning Repository [<http://archive.ics.uci.edu/ml>]. Irvine, CA: University of California, School of Information and Computer Science. In this book, we have modified the dataset to suit the book objectives. The modified dataset can be found here: <http://bit.ly/2HIk5t3>.

Data Exploration Steps

Now that we've understood the business problem and have an idea of what is supposed to be in the data, we can compare these impressions to what we actually see in the data. Your job in data exploration is to not only look through the data both directly and using numerical and graphical summaries, but also to think critically about whether the data make sense and match what you have been told about them. These are helpful steps in data exploration:

1. How many columns are there in the data?
These may be features, response, or metadata.
2. How many rows (samples)?
3. What kind of features are there? Which are **categorical** and which are **numerical**?

Categorical features have values in discrete classes such as "Yes," "No," or "maybe." Numerical features are typically on a continuous numerical scale, such as dollar amounts.

4. What does the data look like in these features?

To see this, you can examine the range of values in numeric features, or the frequency of different classes in categorical features, for example.

5. Is there any missing data?

We have already answered questions 1 and 2 in the previous section; there are 30,000 rows and 25 columns. As we start to explore the rest of these questions in the following exercise, pandas will be our go-to tool. We begin by verifying basic data integrity in the next exercise.

Note

Note that compared to the website's description of the data dictionary, X6-X11 are called PAY_1-PAY_6 in our data. Similarly, X12-X17 are BILL_AMT1-BILL_AMT6, and X18-X23 are PAY_AMT1-PAY_AMT6.

Exercise 3: Verifying Basic Data Integrity

In this exercise, we will perform a basic check on whether our dataset contains what we expect and verify whether there are the correct number of samples.

The data are supposed to have observations for 30,000 credit accounts. While there are 30,000 rows, we should also check whether there are 30,000 unique account IDs. It's possible that, if the SQL query used to generate the data was run on an unfamiliar schema, values that are supposed to be unique are in fact not unique.

To examine this, we can check if the number of unique account IDs is the same as the number of rows. Perform the following steps to complete the exercise:

Note

The code and the resulting output graphics for this exercise have been loaded in a Jupyter Notebook that can be found here: <http://bit.ly/2W9cwPH>.

1. Examine the column names by running the following command in the cell:

```
df.columns
```

The `.columns` method of the DataFrame is employed to examine all the column names. You will obtain the following output once you run the cell:

```
df.columns  
  
Index(['ID', 'LIMIT_BAL', 'SEX', 'EDUCATION', 'MARRIAGE', 'AGE', 'PAY_1',  
       'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6', 'BILL_AMT1', 'BILL_AMT2',  
       'BILL_AMT3', 'BILL_AMT4', 'BILL_AMT5', 'BILL_AMT6', 'PAY_AMT1',  
       'PAY_AMT2', 'PAY_AMT3', 'PAY_AMT4', 'PAY_AMT5', 'PAY_AMT6',  
       'default payment next month'],  
      dtype='object')
```

Figure 1.18: Columns of the dataset

As can be observed, all column names are listed in the output. The account ID column is referenced as **ID**. The remaining columns appear to be our features, with the last column being the response variable. Let's quickly review the dataset information that was given to us by the client:

LIMIT_BAL: Amount of the credit provided (in New Taiwanese (NT) dollar) including individual consumer credit and the family (supplementary) credit.

SEX: Gender (1 = male; 2 = female).

Note

We will not be using the gender data to decide credit-worthiness owing to ethical considerations.

EDUCATION: Education (1 = graduate school; 2 = university; 3 = high school; 4 = others).

MARRIAGE: Marital status (1 = married; 2 = single; 3 = others).

AGE: Age (year).

PAY_1–Pay_6: A record of past payments. Past monthly payments, recorded from April to September, are stored in these columns.

PAY_1 represents the repayment status in September; **PAY_2** = repayment status in August; and so on up to **PAY_6**, which represents the repayment status in April.

The measurement scale for the repayment status is as follows: -1 = pay duly; 1 = payment delay for one month; 2 = payment delay for two months; and so on up to 8 = payment delay for eight months; 9 = payment delay for nine months and above.

BILL_AMT1–BILL_AMT6: Bill statement amount (in NT dollar).

BILL_AMT1 represents the bill statement amount in September; **BILL_AMT2** represents the bill statement amount in August; and so on up to **BILL_AMT7**, which represents the bill statement amount in April.

PAY_AMT1–PAY_AMT6: Amount of previous payment (NT dollar). **PAY_AMT1** represents the amount paid in September; **PAY_AMT2** represents the amount paid in August; and so on up to **PAY_AMT6**, which represents the amount paid in April.

Let's now use the `.head()` method in the next step to observe the first few rows of data.

2. Type the following command in the subsequent cell and run it using *Shift + Enter*:

```
df.head()
```

You will observe the following output:

```
df.head()
```

	ID	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_1
0	798fc410-45c1	20000	2	2	1	24	2
1	8a8c8f3b-8eb4	120000	2	2	2	26	-1
2	85698822-43f5	90000	2	2	2	34	0
3	0737c11b-be42	50000	2	2	1	37	0
4	3b7f77cc-dbc0	50000	1	2	1	57	-1

5 rows × 25 columns

Figure 1.19: `.head()` of a DataFrame

The ID column seems like it contains unique identifiers. Now, to verify if they are in fact unique throughout the whole dataset, we can count the number of unique values using the `.nunique()` method on the **Series** (aka column) **ID**. We first select the column using square brackets.

3. Select the target column (**ID**) and count unique values using the following command:

```
df['ID'].nunique()
```

You will see in the following output that the number of unique entries is 29,687:

```
df[ 'ID' ].nunique()
```

```
29687
```

Figure 1.20: Finding a data quality issue

4. Run the following command to obtain the number of rows in the dataset:

```
df.shape
```

As can be observed in the following output, the total number of rows in the dataset is 30,000:

```
df.shape
```

```
(30000, 25)
```

Figure 1.21: Dimensions of the dataset

We see here that the number of unique IDs is less than the number of rows. This implies that the ID is not a unique identifier for the rows of the data, as we thought. So we know that there is some duplication of IDs. But how much? Is one ID duplicated many times? How many IDs are duplicated?

We can use the `.value_counts()` method on the ID series to start to answer these questions. This is similar to a **group by/count** procedure in SQL. It will list the unique IDs and how often they occur. We will perform this operation in the next step and store the value counts in a variable **id_counts**.

5. Store the value counts in a variable defined as `id_counts` and then display the stored values using the `.head()` method, as shown:

```
id_counts = df['ID'].value_counts()  
id_counts.head()
```

You will obtain the following output:

```
id_counts = df['ID'].value_counts()  
id_counts.head()
```

```
e50d8395-da32      2  
4534975d-bf92      2  
a3a5c0fc-fdd6      2  
0d66d575-c461      2  
fd6033f4-cc72      2  
Name: ID, dtype: int64
```

Figure 1.22: Getting value counts of the account IDs

Note that `.head()` returns the first five rows by default. You can specify the number of items to be displayed by passing the required number in the parentheses, `()`.

6. Display the number of grouped duplicated entries by running another value count:

```
id_counts.value_counts()
```

You will obtain the following output:

```
id_counts.value_counts()
```

```
1      29374  
2      313  
Name: ID, dtype: int64
```

Figure 1.23: Getting value counts of the account IDs

In the preceding output and from the initial value count, we can see that most IDs occur exactly once, as expected. However, 313 IDs occur twice. So, no ID occurs more than twice. Armed with this information, we are ready to begin taking a closer look at this data quality issue and fixing it. We will be creating Boolean masks to further clean the data.

Boolean Masks

To help clean the case study data, we introduce the concept of a **logical mask**, also known as a **Boolean mask**. A logical mask is a way to filter an array, or series, by some condition. For example, we can use the "is equal to" operator in Python, `==`, to find all locations of an array that contain a certain value. Other comparisons, such as "greater than" (`>`), "less than" (`<`), "greater than or equal to" (`>=`), and "less than or equal to" (`<=`), can be used similarly. The output of such a comparison is an array or series of True/False values, also known as **Boolean** values. Each element of the output corresponds to an element of the input, is **True** if the condition is met, and is **False** otherwise. To illustrate how this works, we will use **synthetic data**. Synthetic data is data that is created to explore or illustrate a concept. First, we are going to import the NumPy package, which has many capabilities for generating random numbers, and give it the alias `np`:

```
import numpy as np
```

Now we use what's called a **seed** for the random number generator. If you set the seed, you will get the same results from the random number generator across runs. Otherwise this is not guaranteed. This can be a helpful option if you use random numbers in some way in your work and want to have consistent results every time you run a notebook:

```
np.random.seed(seed=24)
```

Next, we generate 100 random integers, chosen from between 1 and 5 (inclusive). For this we can use `numpy.random.randint`, with the appropriate arguments.

```
random_integers = np.random.randint(low=1, high=5, size=100)
```

Let's look at the first five elements of this array, with `random_integers[:5]`. The output should appear as follows:

```
array([3, 4, 1, 4, 2])
```

Figure 1.24: Random integers

Suppose we wanted to know the locations of all elements of `random_integers` equal to 3. We could create a Boolean mask to do this.

```
is_equal_to_3 = random_integers == 3
```

From examining the first 5 elements, we know the first element is equal to 3, but none of the rest are. So in our Boolean mask, we expect **True** in the first position and **False** in the next 4 positions. Is this the case?

```
is_equal_to_3[:5]
```

The preceding code should give this output:

```
array([ True, False, False, False, False])
```

Figure 1.25: Boolean mask for the random integers

This is what we expected. This shows the creation of a Boolean mask. But what else can we do with them? Suppose we wanted to know how many elements were equal to 3. To know this, you can take the sum of a Boolean mask, which interprets **True** as 1 and **False** as 0:

```
sum(is_equal_to_3)
```

This should give us the following output:

22

Figure 1.26: Sum of the Boolean mask

This makes sense, as with a random, equally likely choice of 5 possible values, we would expect each value to appear about 20% of the time. In addition to seeing how many values in the array meet the Boolean condition, we can also use the Boolean mask to select the elements of the original array that meet that condition. Boolean masks can be used directly to index arrays, as shown here:

```
random_integers[is_equal_to_3]
```

This outputs the elements of **random_integers** meeting the Boolean condition we specified. In this case, the 22 elements equal to 3:

```
array([3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3])
```

Figure 1.27: Using the Boolean mask to index an array

Now you know the basics of Boolean arrays, which are useful in many situations. In particular, you can use the **.loc** method of **DataFrames** to index the rows of the **DataFrames** by a Boolean mask, and the columns by label. Let's continue exploring the case study data with these skills.

Exercise 4: Continuing Verification of Data Integrity

In this exercise, with our knowledge of Boolean arrays, we will examine some of the duplicate IDs we discovered. In Exercise 3, we learned that no ID appears more than twice. We can use this learning to locate the duplicate IDs and examine them. Then we take action to remove rows of dubious quality from the dataset. Perform the following steps to complete the exercise:

Note

The code and the output graphics for this exercise have been loaded in a Jupyter Notebook that can be found here: <http://bit.ly/2W9cwPH>.

1. Continuing where we left off in Exercise 3, we want the indices of the `id_counts` series, where the count is 2, to locate the duplicates. We assign the indices of the duplicated IDs to a variable called `dupe_mask` and display the first 5 duplicated IDs using the following commands:

```
dupe_mask = id_counts == 2  
dupe_mask[0:5]
```

You will obtain the following output:

```
47d9ee33-0df0      True  
db903e22-a55a      True  
9878723a-0b58      True  
8d3a2576-a958      True  
956cbf4a-d24e      True  
Name: ID, dtype: bool
```

Figure 1.28: A Boolean mask to locate duplicate IDs

Here, `dupe_mask` is the logical mask that we have created for storing the Boolean values.

Note that in the preceding output, we are displaying only the first five entries using `dupe_mask` to illustrate the contents of this array. As always, you can edit the indices in the square brackets (`[]`) to change the number of entries displayed.

Our next step is to use this logical mask to select the IDs that are duplicated. The IDs themselves are contained as the index of the `id_count` series. We can access the index in order to use our logical mask for selection purposes.

2. Access the index of **id_count** and display the first five rows as context using the following command:

```
id_counts.index[0:5]
```

With this, you will obtain the following output:

```
Index(['47d9ee33-0df0', 'db903e22-a55a', '9878723a-0b58', '8d3a2576-a958',
       '956cbf4a-d24e'],
      dtype='object')
```

Figure 1.29: Duplicated IDs

3. Select and store the duplicated IDs in a new variable called **dupe_ids** using the following command:

```
dupe_ids = id_counts.index[dupe_mask]
```

4. Convert **dupe_ids** to a list and then obtain the length of the list using the following commands:

```
dupe_ids = list(dupe_ids)
len(dupe_ids)
```

You should obtain the following output:

313

Figure 1.30: Output displaying the list length

We changed the **dupe_ids** variable to a **list**, as we will need it in this form for future steps. The list has a length of 313, as can be seen in the preceding output, which matches our knowledge of the number of duplicate IDs from the value count.

5. We verify the data in **dupe_ids** by displaying the first five entries using the following command:

```
dupe_ids[0:5]
```

We obtain the following output:

```
dupe_ids[0:5]
```

```
[ '47d9ee33-0df0' ,  
  'db903e22-a55a' ,  
  '9878723a-0b58' ,  
  '8d3a2576-a958' ,  
  '956cbf4a-d24e' ]
```

Figure 1.31: Making a list of duplicate IDs

We can observe from the preceding output that the list contains the required entries of duplicate IDs. We're now in a position to examine the data for the IDs in our list of duplicates. In particular, we'd like to look at the values of the features, to see what, if anything, might be different between these duplicate entries. We will use the `.isin` and `.loc` methods for this purpose.

Using the first three IDs on our list of dupes, `dupe_ids[0:3]`, we will plan to first find the rows containing these IDs. If we pass this list of IDs to the `.isin` method of the ID series, this will create another logical mask we can use on the larger DataFrame to display the rows that have these IDs. The `.isin` method is nested in a `.loc` statement indexing the DataFrame in order to select the location of all rows containing "True" in the Boolean mask. The second argument of the `.loc` indexing statement is `:`, which implies that all columns will be selected. By performing the following steps, we are essentially filtering the DataFrame in order to view all the columns for the first three duplicate IDs.

6. Run the following command in your Notebook to execute the plan we formulated in the previous step:

```
df.loc[df['ID'].isin(dupe_ids[0:3]), :].head(10)
```

	ID	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_1	PAY_2	PAY_3	PAY_4	...	BILL_AMT4	BILL_AMT5
11769	9878723a-0b58	130000	1	5	1	44	0	0	0	0	...	54668	22780
11869	9878723a-0b58	0	0	0	0	0	0	0	0	0	...	0	0
14979	db903e22-a55a	50000	1	2	2	55	2	0	0	0	...	15848	16026
15079	db903e22-a55a	0	0	0	0	0	0	0	0	0	...	0	0
23377	47d9ee33-0df0	550000	2	2	1	32	2	0	0	0	...	548020	530672
23477	47d9ee33-0df0	0	0	0	0	0	0	0	0	0	...	0	0

6 rows × 25 columns

Figure 1.32: Examining the data for duplicate IDs

What we observe here is that each duplicate ID appears to have one row with what seems like valid data, and one row of entirely zeros. Take a moment and think to yourself what you would do with this knowledge.

After some reflection, it should be clear that you ought to delete the rows with all zeros. Perhaps these arose through a faulty join condition in the SQL query that generated the data? Regardless, a row of all zeros is definitely invalid data as it makes no sense for someone to have an age of 0, a credit limit of 0, and so on.

One approach to deal with this issue would be to find rows that have all zeros, except for the first column, which has the IDs. These would be invalid data in any case, and it may be that if we get rid of all of these, we would also solve our problem of duplicate IDs. We can find the entries of the DataFrame that are equal to zero by creating a Boolean matrix that is the same size as the whole DataFrame, based on the "is equal to zero" condition.

7. Create a Boolean matrix of the same size as the entire DataFrame using `==`, as shown:

```
df_zero_mask = df == 0
```

In the next steps, we'll use `df_zero_mask`, which is another DataFrame containing Boolean values. The goal will be to create a Boolean series, `feature_zero_mask`, that identifies every row where all the elements starting from the second column (the features and response, but not the IDs) are 0. To do so, we first need to index `df_zero_mask` using the integer indexing (`.iloc`) method. In this method, we pass `(:)` to examine all rows and `(1:)` to examine all columns starting with the second one (index 1). Finally, we will apply the `all()` method along the column axis (`axis=1`), which will return `True` if and only if every column in that row is `True`. This is a lot to think about, but it's pretty simple to code, as will be observed in the following step.

8. Create the Boolean series `feature_zero_mask`, as shown in the following:

```
feature_zero_mask = df_zero_mask.iloc[:,1: ].all(axis=1)
```

9. Calculate the sum of the Boolean series using the following command:

```
sum(feature_zero_mask)
```

You should obtain the following output:

315

Figure 1.33: The number of rows with all zeros except for the ID

The preceding output tells us that 315 rows have zeros for every column but the first one. This is greater than the number of duplicate IDs (313), so if we delete all the "zero rows," we may get rid of the duplicate ID problem.

10. Clean the DataFrame by eliminating the rows with all zeros, except for the ID, using the following code:

```
df_clean_1 = df.loc[~feature_zero_mask,: ].copy()
```

While performing the cleaning operation in the preceding step, we return a new DataFrame called `df_clean_1`. Notice that here we've used the `.copy()` method after the `.loc` indexing operation to create a copy of this output, as opposed to a view on the original DataFrame. You can think of this as creating a new DataFrame, as opposed to referencing the original one. Within the `.loc` method, we used the logical not operator, `~`, to select all the rows that don't have zeros for all the features and response, and `:` to select all columns. These are the valid data we wish to keep. After doing this, we now want to know if the number of remaining rows is equal to the number of unique IDs.

11. Verify the number of rows and columns in `df_clean_1` by running the following code:

```
df_clean_1.shape
```

You will obtain the following output:

```
df_clean_1 = df.loc[~feature_zero_mask,:].copy()
```

```
df_clean_1.shape
```

```
(29685, 25)
```

Figure 1.34: Dimensions of the cleaned DataFrame

12. Obtain the number of unique IDs by running the following code:

```
df_clean_1['ID'].nunique()
```

```
df_clean_1['ID'].nunique()
```

```
29685
```

Figure 1.35: Number of unique IDs in the cleaned DataFrame

From the preceding output, we can see that we have successfully eliminated duplicates, as the number of unique IDs is equal to the number of rows. Now take a breath and pat yourself on the back. That was a whirlwind introduction to quite a few pandas techniques for indexing and characterizing data. Now that we've filtered out the duplicate IDs, we're in a position to start looking at the actual data itself: the features, and eventually, the response. We'll walk you through this process.

Exercise 5: Exploring and Cleaning the Data

Thus far, we have identified a data quality issue related to the metadata: we had been told that every sample from our dataset corresponded to a unique account ID, but found that this was not the case. We were able to use logical indexing and pandas to correct this issue. This was a fundamental data quality issue, having to do simply with what samples were present, based on the metadata. Aside from this, we are not really interested in the metadata column of account IDs: for the time being these will not help us develop a predictive model for credit default.

Now, we are ready to start examining the values of the features and response, the data we will use to develop our predictive model. Perform the following steps to complete this exercise:

Note

The code and the resulting output for this exercise have been loaded in a Jupyter Notebook that can be found here: <http://bit.ly/2W9cwPH>.

1. Obtain the data type of the columns in the data by using the `.info()` method as shown:

```
df_clean_1.info()
```

You should see the following output:

```
df_clean_1.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 29685 entries, 0 to 29999
Data columns (total 25 columns):
ID                      29685 non-null object
LIMIT_BAL                29685 non-null int64
SEX                      29685 non-null int64
EDUCATION                29685 non-null int64
MARRIAGE                 29685 non-null int64
AGE                      29685 non-null int64
PAY_1                     29685 non-null object
PAY_2                     29685 non-null int64
PAY_3                     29685 non-null int64
PAY_4                     29685 non-null int64
PAY_5                     29685 non-null int64
PAY_6                     29685 non-null int64
BILL_AMT1                 29685 non-null int64
BILL_AMT2                 29685 non-null int64
BILL_AMT3                 29685 non-null int64
BILL_AMT4                 29685 non-null int64
BILL_AMT5                 29685 non-null int64
BILL_AMT6                 29685 non-null int64
PAY_AMT1                  29685 non-null int64
PAY_AMT2                  29685 non-null int64
PAY_AMT3                  29685 non-null int64
PAY_AMT4                  29685 non-null int64
PAY_AMT5                  29685 non-null int64
PAY_AMT6                  29685 non-null int64
default payment next month 29685 non-null int64
dtypes: int64(23), object(2)
memory usage: 5.9+ MB
```

Figure 1.36: Getting column metadata

We can see in *Figure 1.34* that there are 25 columns. Each row has 29,685 **non-null** values, according to this summary, which is the number of rows in the DataFrame. This would indicate that there is no missing data, in the sense that each cell contains some value. However, if there is a fill value to represent missing data, that would not be evident here.

We also see that most columns say **int64** next to them, indicating they are an **integer** data type, that is, numbers such as ..., -2, -1, 0, 1, 2,... . The exceptions are **ID** and **PAY_1**. We are already familiar with **ID**; this contains strings, which are account IDs. What about **PAY_1**? According to the values in the data dictionary, we'd expect this to contain integers, like all the other features. Let's take a closer look at this column.

2. Use the **.head(n)** pandas method to view the top **n** rows of the **PAY_1** series:

```
df_clean_1['PAY_1'].head(5)
```

You should obtain the following output:

```
df_clean_1['PAY_1'].head(5)
```

```
0      2
1     -1
2      0
3      0
4     -1
Name: PAY_1, dtype: object
```

Figure 1.37: Examine a few columns' contents

The integers on the left of the output are the index, which are simply consecutive integers starting with 0. The data from the **PAY_1** column is shown on the left. This is supposed to be the payment status of the most recent month's bill, using values -1, 1, 2, 3, and so on. However, we can see that there are values of 0 here, which are not documented in the data dictionary. According to the data dictionary, "The measurement scale for the repayment status is: -1 = pay duly; 1 = payment delay for one month; 2 = payment delay for two months; . . . ; 8 = payment delay for eight months; 9 = payment delay for nine months and above" (<https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients>). Let's take a closer look, using the value counts of this column.

3. Obtain the value counts for the **PAY_1** column by using `.value_counts()` method:

```
df_clean1['PAY_1'].value_counts()
```

You should see the following output:

```
df_clean_1['PAY_1'].value_counts()

0                13087
-1               5047
1                3261
Not available   3021
-2               2476
2                2378
3                292
4                 63
5                 23
8                 17
6                 11
7                  9
Name: PAY_1, dtype: int64
```

Figure 1.38: Value counts of the **PAY_1** column

The preceding output reveals the presence of two undocumented values: 0 and -2, as well as the reason this column was imported by pandas as an **object** data type, instead of **int64** as we would expect for integer data. There is a '**Not available**' string present in this column, symbolizing missing data. Later on in the book, we'll come back to this when we consider how to deal with missing data. For now, we'll remove rows of the dataset, for which this feature has a missing value.

4. Use a logical mask with the `!=` operator (which means "does not equal" in Python) to find all the rows that don't have missing data for the **PAY_1** feature:

```
valid_pay_1_mask = df_clean_1['PAY_1'] != 'Not available'
valid_pay_1_mask[0:5]
```

By running the preceding code, you will obtain the following output:

```
valid_pay_1_mask = df_clean_1['PAY_1'] != 'Not available'  
  
valid_pay_1_mask[0:5]  
  
0    True  
1    True  
2    True  
3    True  
4    True  
Name: PAY_1, dtype: bool
```

Figure 1.39: Creating a Boolean mask

5. Check how many rows have no missing data by calculating the sum of the mask:

```
sum(valid_pay_1_mask)
```

You will obtain the following output:

```
sum(valid_pay_1_mask)
```

26664

Figure 1.40: Sum of the Boolean mask for non-missing data

We see that 26,664 rows do not have the value '**Not available**' in the **PAY_1** column. We saw from the value count that 3,021 rows do have this value, and $29,685 - 3,021 = 26,664$, so this checks out.

6. Clean the data by eliminating the rows with the missing values of **PAY_1** as shown:

```
df_clean_2 = df_clean_1.loc[valid_pay_1_mask, :].copy()
```

7. Obtain the shape of the cleaned data using the following command:

```
df_clean_2.shape
```

You will obtain the following output:

```
df_clean_2 = df_clean_1.loc[valid_pay_1_mask,:].copy()

df_clean_2.shape

(26664, 25)
```

Figure 1.41: Shape of the cleaned data

After removing these rows, we check that the resulting DataFrame has the expected shape. You can also check for yourself whether the value counts indicate the desired values have been removed like this: `df_clean_2['PAY_1'].value_counts()`.

Lastly, so this column's data type can be consistent with the others, we will cast it from the generic `object` type to `int64` like all the other features, using the `.astype` method. Then we select a couple columns, including `PAY_1`, to examine the data types and make sure it worked.

- Run the following command to convert the data type for `PAY_1` from `object` to `int64` and show the column metadata for `PAY_1` and `PAY_2`:

```
df_clean_2['PAY_1'] = df_clean_2['PAY_1'].astype('int64')
df_clean_2[['PAY_1', 'PAY_2']].info()
```

```
df_clean_2['PAY_1'] = df_clean_2['PAY_1'].astype('int64')

df_clean_2[['PAY_1', 'PAY_2']].info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 26664 entries, 0 to 29999
Data columns (total 2 columns):
PAY_1    26664 non-null int64
PAY_2    26664 non-null int64
dtypes: int64(2)
memory usage: 624.9 KB
```

Figure 1.42: Check the data type of the cleaned column

Congratulations, you have completed your second data cleaning operation! However, if you recall, during this process we also noticed the undocumented values of -2 and 0 in **PAY_1**. Now, let's imagine we got back in touch with our business partner and learned the following information:

- -2 means the account started that month with a zero balance, and never used any credit
- -1 means the account had a balance that was paid in full
- 0 means that at least the minimum payment was made, but the entire balance wasn't paid (that is, a positive balance was carried to the next month)

We thank our business partner since this answers our questions, for now. Maintaining a good line of communication and working relationship with the business partner is important, as you can see here, and may determine the success or failure of a project.

Data Quality Assurance and Exploration

So far, we remedied two data quality issues just by asking basic questions or by looking at the `.info()` summary. Let's now take a look at the first few columns. Before we get to the historical bill payments, we have the credit limits of the accounts of **LIMIT_BAL**, and the demographic features **SEX**, **EDUCATION**, **MARRIAGE**, and **AGE**. Our business partner has reached out to us, to let us know that gender should not be used to predict credit-worthiness, as this is unethical by their standards. So we keep this in mind for future reference. Now we explore the rest of these columns, making any corrections that are necessary.

In order to further explore the data, we will use histograms. Histograms are a good way to visualize data that is on a continuous scale, such as currency amounts and ages. A histogram groups similar values into bins, and shows the number of data points in these bins as a bar graph.

To plot histograms, we will start to get familiar with the graphical capabilities of pandas. pandas relies on another library called Matplotlib to create graphics, so we'll also set some options using matplotlib. Using these tools, we'll also learn how to get quick statistical summaries of data in pandas.

Exercise 6: Exploring the Credit Limit and Demographic Features

In this exercise, we start our exploration of data with the credit limit and age features. We will visualize them and get summary statistics to check that the data contained in these features is sensible. Then we will look at the education and marriage categorical features to see if the values there make sense, and correct them as necessary. `LIMIT_BAL` and `AGE` are numerical features, meaning they are measured on a continuous scale. Consequently, we'll use histograms to visualize them. Perform the following steps to complete the exercise:

Note

The code and the resulting output for this exercise have been loaded in a Jupyter Notebook that can be found here: <http://bit.ly/2W9cwPH>.

1. Import `matplotlib` and set up some plotting options with this code snippet:

```
import matplotlib.pyplot as plt #import plotting package

#render plotting automatically
%matplotlib inline

import matplotlib as mpl #additional plotting functionality

mpl.rcParams['figure.dpi'] = 400 #high resolution figures
```

This imports `matplotlib` and uses `.rcParams` to set the resolution (`dpi` = dots per inch) for a nice crisp image; you may not want to worry about this last part unless you are preparing things for presentation, as it could make the images quite large in your notebook.

2. Run `df_clean_2[['LIMIT_BAL', 'AGE']].hist()` and you should see the following histograms:

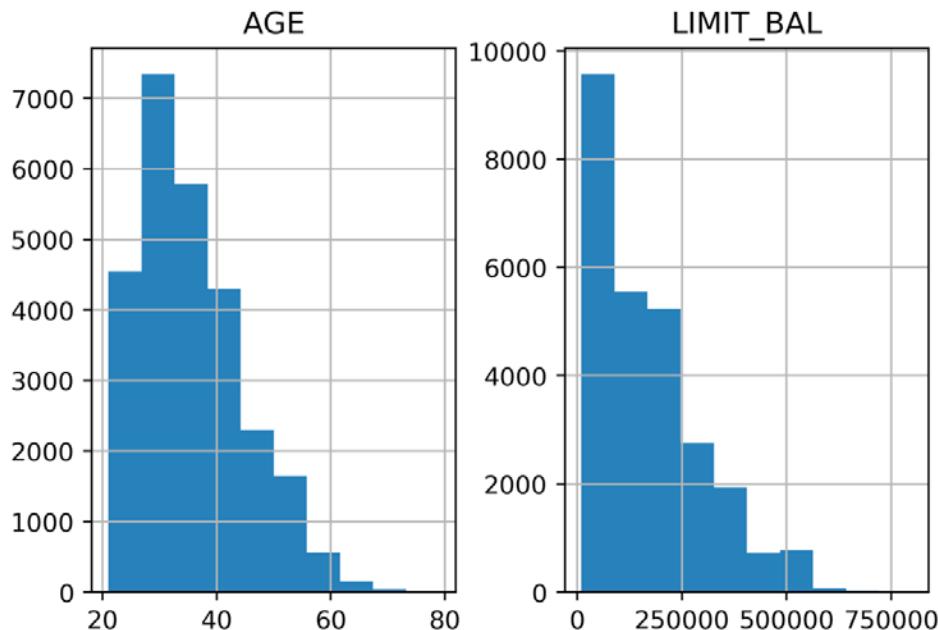


Figure 1.43: Histograms of the credit limit and age data

This is nice visual snapshot of these features. We can get a quick, approximate look at all of the data in this way. In order to see statistics such as mean and median (that is, the 50th percentile), there is another helpful pandas function.

3. Generate a tabular report of summary statistics using the following command:

```
df_clean_2[['LIMIT_BAL', 'AGE']].describe()
```

You should see the following output:

```
df_clean_2[['LIMIT_BAL', 'AGE']].describe()
```

	LIMIT_BAL	AGE
count	26664.000000	26664.000000
mean	167919.054905	35.505213
std	129839.453081	9.227442
min	10000.000000	21.000000
25%	50000.000000	28.000000
50%	140000.000000	34.000000
75%	240000.000000	41.000000
max	800000.000000	79.000000

Figure 1.44: Statistical summaries of credit limit and age data

Based on the histograms and the convenient statistics computed by `.describe()`, which include a count of non-nulls, the mean and standard deviation, minimum, maximum, and quartiles, we can make a few judgements.

LIMIT_BAL, the credit limit, seems to make sense. The credit limits have a minimum of 10,000. This dataset is from Taiwan; the exact unit of currency (NT dollar) may not be familiar, but intuitively, a credit limit should be above zero. You are encouraged to look up the conversion to your local currency and consider these credit limits. For example, 1 US dollar is about 30 NT dollars.

The **AGE** feature also looks reasonably distributed, with no one under the age of 21 having a credit account.

For the categorical features, a look at the value counts is useful, since there are relatively few unique values.

4. Obtain the value counts for the **EDUCATION** feature using the following code:

```
df_clean_2['EDUCATION'].value_counts()
```

You should see this output:

```
df_clean_2['EDUCATION'].value_counts()

2    12458
1    9412
3    4380
5    245
4    115
6     43
0     11
Name: EDUCATION, dtype: int64
```

Figure 1.45: Value counts of the **EDUCATION** feature

Here, we see undocumented education levels 0, 5, and 6, as the data dictionary describes only "Education (1 = graduate school; 2 = university; 3 = high school; 4 = others)". Our business partner tells us they don't know about the others. Since they are not very prevalent, we will lump them in with the "others" category, which seems appropriate, with our client's blessing, of course.

5. Run this code to combine the undocumented levels of the **EDUCATION** feature into the level for "others" and then examine the results:

```
df_clean_2['EDUCATION'].replace(to_replace=[0, 5, 6], value=4,
                                inplace=True)
df_clean_2['EDUCATION'].value_counts()
```

The pandas `.replace` method makes doing the replacements described in the preceding step pretty quick. Once you run the code, you should see this output:

```
df_clean_2['EDUCATION'].replace(to_replace=[0, 5, 6], value=4, inplace=True)

df_clean_2['EDUCATION'].value_counts()

2    12458
1    9412
3    4380
4    414
Name: EDUCATION, dtype: int64
```

Figure 1.46: Cleaning the **EDUCATION** feature

Note that here we make this change **in place** (`inplace=True`). This means that, instead of returning a new DataFrame, this operation will make the change on the existing DataFrame.

- Obtain the value counts for the **MARRIAGE** feature using the following code:

```
df_clean_2['MARRIAGE'].value_counts()
```

You should obtain the following output:

```
df_clean_2['MARRIAGE'].value_counts()
```

```
2    14158
1    12172
3     286
0      48
Name: MARRIAGE, dtype: int64
```

Figure 1.47: Value counts of raw **MARRIAGE** feature

The issue here is similar to that encountered for the **EDUCATION** feature; there is a value, 0, which is not documented in the data dictionary: "1 = married; 2 = single; 3 = others". So we'll lump it in with "others".

- Change the values of 0 in the **MARRIAGE** feature to 3 and examine the result with this code:

```
df_clean_2['MARRIAGE'].replace(to_replace=0, value=3, inplace=True)
df_clean_2['MARRIAGE'].value_counts()
```

The output should be:

```
2    14158
1    12172
3     334
Name: MARRIAGE, dtype: int64
```

Figure 1.48: Value counts of cleaned **MARRIAGE** feature

We've now accomplished a lot of exploration and cleaning of the data. We will do some more advanced visualization and exploration of the financial history features, that come after this in the DataFrame, later.

Deep Dive: Categorical Features

Machine learning algorithms only work with numbers. If your data contains text features, for example, these would require transformation to numbers in some way. We learned above that the data for our case study is, in fact, entirely numerical. However, it's worth thinking about how it got to be that way. In particular, consider the **EDUCATION** feature.

This is an example of what is called a **categorical feature**: you can imagine that as raw data, this column consisted of the text labels '**graduate school**', '**university**', '**high school**', and '**others**'. These are called the **levels** of the categorical feature; here, there are four levels. It is only through a mapping, which has already been chosen for us, that these data exist as the numbers 1, 2, 3, and 4 in our dataset. This particular assignment of categories to numbers creates what is known as an **ordinal feature**, since the levels are mapped to numbers in order. As a data scientist, at a minimum you need to be aware of such mappings, if you are not choosing them yourself.

What are the implications of this mapping?

It makes some sense that the education levels are ranked, with 1 corresponding to the highest level of education in our dataset, 2 to the next highest, 3 to the next, and 4 presumably including the lowest levels. However, when you use this encoding as a numerical feature in a machine learning model, it will be treated just like any other numerical feature. For some models, this effect may not be desired.

What if a model seeks to find a straight-line relationship between the features and response?

Whether or not this works in practice depends on the actual relationship between different levels of education and the outcome we are trying to predict.

Here, we examine two hypothetical cases of ordinal categorical variables, each with 10 levels. The levels measure the self-reported satisfaction levels from customers visiting a website. The average number of minutes spent on the website for customers reporting each level is plotted on the y-axis. We've also plotted the line of best fit in each case to illustrate how a linear model would deal with these data, as shown in the following figure:

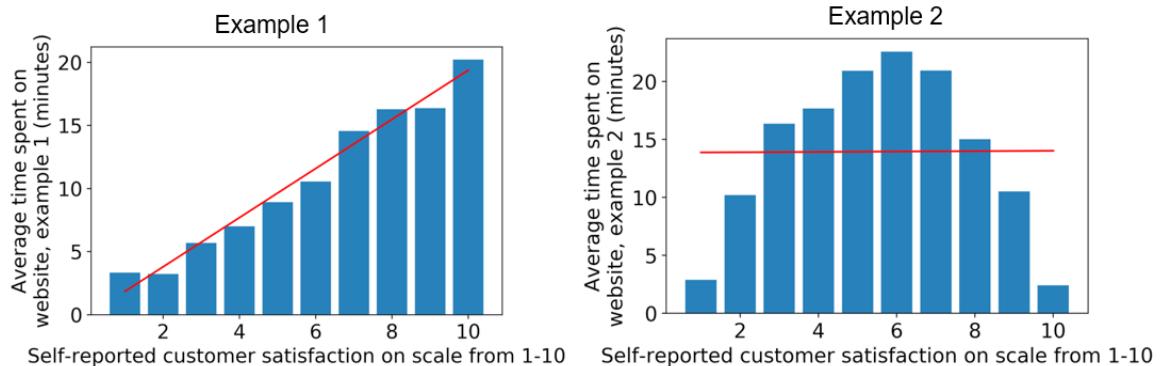


Figure 1.49: Ordinal features may or may not work well in a linear model

We can see that if an algorithm assumes a linear (straight-line) relationship between features and response, this may or may not work well depending on the actual relationship between this feature and the response variable. Notice that in the preceding example, we are modeling a regression problem: the response variable takes on a continuous range of numbers. However, some classification algorithms such as **logistic regression** also assume a linear effect of the features. We will discuss this in greater detail later when we get into modeling the data for our case study.

Roughly speaking, for a binary classification model, you can look at the different levels of a categorical feature in terms of the average values of the response variable, which represent the "rates" of the positive class (i.e., the samples where the response variable = 1) for each level. This can give you an idea of whether an ordinal encoding will work well with a linear model. Assuming you've imported the same packages in your Jupyter notebook as in the previous sections, you can quickly look at this using a **groupby/agg** and bar plot in pandas:

```
f_clean_2.groupby('EDUCATION').agg({'default payment next month':'mean'}).
plot.bar(legend=False)
plt.ylabel('Default rate')
plt.xlabel('Education level: ordinal encoding')
```

Once you run the code, you should obtain the following output:

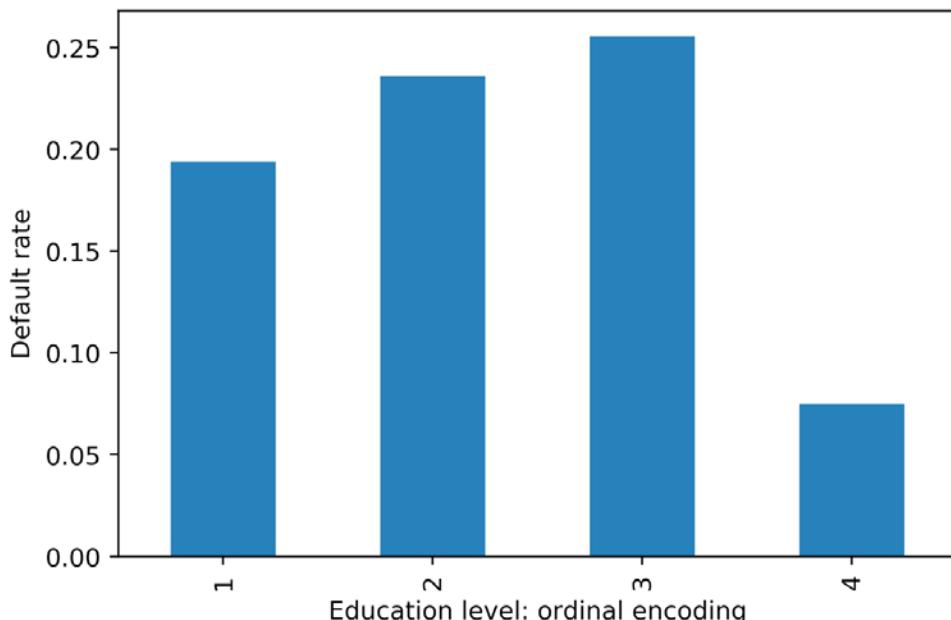


Figure 1.50: Default rate within education levels

Similar to example 2 in Figure 1.49, it looks like a straight-line fit would probably not be the best description of the data here. In case a feature has a non-linear effect like this, it may be better to use a more complex algorithm such as a decision tree or random forest. Or, if a simpler and more interpretable linear model such as logistic regression is desired, we could avoid an ordinal encoding and use a different way of encoding categorical variables. A popular way of doing this is called one-hot encoding (OHE).

OHE is a way to transform a categorical feature, which may consist of text labels in the raw data, into a numerical feature that can be used in mathematical models.

Let's learn about this in an exercise. And if you are wondering why a logistic regression is more interpretable and a random forest is more complex, we will be learning about these concepts in detail during the rest of the book.

Note: Categorical variables in different machine learning packages

Some machine learning packages, for instance, certain R packages or newer versions of the Spark platform for big data, will handle categorical variables without assuming they are ordinal. Always be sure to carefully read the documentation to learn what the model will assume about the features, and how to specify whether a variable is categorical, if that option is available.

Exercise 7: Implementing OHE for a Categorical Feature

In this exercise, we will "reverse engineer" the **EDUCATION** feature in the dataset to obtain the text labels that represent the different education levels, then show how to use pandas to create an OHE.

First, let's consider our **EDUCATION** feature, before it was encoded as an ordinal. From the data dictionary, we know that 1 = graduate school, 2 = university, 3 = high school, 4 = others. We would like to recreate a column that has these strings, instead of numbers. Perform the following steps to complete the exercise:

Note

The code and the resulting output for this exercise have been loaded in a Jupyter Notebook that can be found here: <http://bit.ly/2W9cwPH>.

1. Create an empty column for the categorical labels called **EDUCATION_CAT**. Using the following command:

```
df_clean_2['EDUCATION_CAT'] = 'none'
```

2. Examine the first few rows of the DataFrame for the **EDUCATION** and **EDUCATION_CAT** columns:

```
df_clean_2[['EDUCATION', 'EDUCATION_CAT']].head(10)
```

The output should appear as follows:

```
df_clean_2[['EDUCATION', 'EDUCATION_CAT']].head(10)
```

	EDUCATION	EDUCATION_CAT
0	2	none
1	2	none
2	2	none
3	2	none
4	2	none
5	1	none
6	1	none
7	2	none
8	3	none
9	3	none

Figure 1.51: Selecting columns and viewing the first 10 rows

We need to populate this new column with the appropriate strings. pandas provides a convenient functionality for mapping values of a **Series** on to new values. This function is in fact called **.map** and relies on a dictionary to establish the correspondence between the old values and the new values. Our goal here is to map the numbers in EDUCATION on to the strings they represent. For example, where the EDUCATION column equals the number 1, we'll assign the 'graduate school' string to the EDUCATION_CAT column, and so on for the other education levels.

3. Create a dictionary that describes the mapping for education categories using the following code:

```
cat_mapping = {  
    1: "graduate school",  
    2: "university",  
    3: "high school",  
    4: "others"  
}
```

4. Apply the mapping to the original EDUCATION column using **.map** and assign the result to the new EDUCATION_CAT column:

```
df_clean_2['EDUCATION_CAT'] = df_clean_2['EDUCATION'].map(cat_mapping)  
df_clean_2[['EDUCATION', 'EDUCATION_CAT']].head(10)
```

After running those lines, you should see the following output:

```
df_clean_2[['EDUCATION', 'EDUCATION_CAT']].head(10)
```

	EDUCATION	EDUCATION_CAT
0	2	university
1	2	university
2	2	university
3	2	university
4	2	university
5	1	graduate school
6	1	graduate school
7	2	university
8	3	high school
9	3	high school

Figure 1.52: Examining the string values corresponding to the ordinal encoding of EDUCATION

Excellent! Note that we could have skipped Step 1, where we assigned the new column with '`none`', and gone straight to Steps 3 and 4 to create the new column. However, sometimes it's useful to create a new column initialized with a single value, so it's worth knowing how to do that.

Now we are ready to one-hot encode. We can do this by passing a `Series` of a `DataFrame` to the pandas `get_dummies()` function. The function got this name because one-hot encoded columns are also referred to as **dummy variables**. The result will be a new DataFrame, with as many columns as there are levels of the categorical variable.

- Run this code to create a one-hot encoded DataFrame of the EDUCATION_CAT column. Examine the first 10 rows:

```
edu_ohe = pd.get_dummies(df_clean_2['EDUCATION_CAT'])
edu_ohe.head(10)
```

This should produce the following output:

```
edu_ohe = pd.get_dummies(df_clean_2[ 'EDUCATION_CAT' ])
edu_ohe.head(10)
```

	graduate school	high school	none	others	university
0	0	0	0	0	1
1	0	0	0	0	1
2	0	0	0	0	1
3	0	0	0	0	1
4	0	0	0	0	1
5	1	0	0	0	0
6	1	0	0	0	0
7	0	0	0	0	1
8	0	1	0	0	0
9	0	1	0	0	0

Figure 1.53: Data frame of one-hot encoding

You can now see why this is called "one-hot encoding": across all these columns, any particular row will have a 1 in exactly 1 column, and 0s in the rest. For a given row, the column with the 1 should match up to the level of the original categorical variable. To check this, we need to concatenate this new DataFrame with the original one and examine the results side by side. We will use the pandas `concat` function, to which we pass the list of DataFrames we wish to concatenate, and the `axis=1` keyword saying to concatenate them horizontally; that is, along the column axis. This basically means we are combining these two DataFrames "side by side", which we know we can do because we just created this new DataFrame from the original one: we know it will have the same number of rows, which will be in the same order as the original DataFrame.

6. Concatenate the one-hot encoded DataFrame to the original DataFrame as follows:

```
df_with_ohe = pd.concat([df_clean_2, edu_ohe], axis=1)
df_with_ohe[['EDUCATION_CAT', 'graduate school',
             'high school', 'university', 'others']].head(10)
```

You should see this output:

```
df_with_ohe = pd.concat([df_clean_2, edu_ohe], axis=1)
df_with_ohe[['EDUCATION_CAT', 'graduate school',
             'high school', 'university', 'others']].head(10)
```

	EDUCATION_CAT	graduate school	high school	university	others
0	university	0	0	1	0
1	university	0	0	1	0
2	university	0	0	1	0
3	university	0	0	1	0
4	university	0	0	1	0
5	graduate school	1	0	0	0
6	graduate school	1	0	0	0
7	university	0	0	1	0
8	high school	0	1	0	0
9	high school	0	1	0	0

Figure 1.54: Checking the one-hot encoded columns

Alright, looks like this has worked as intended. OHE is another way to encode categorical features that avoids the implied numerical structure of an ordinal encoding. However, notice what has happened here: we have taken a single column, **EDUCATION**, and exploded it out into as many columns as there were levels in the feature. In this case, since there are only four levels, this is not such a big deal. However, if your categorical variable had a very large number of levels, you may want to consider an alternate strategy, such as grouping some levels together into single categories.

This is a good time to save the DataFrame we've created here, which encapsulates our efforts at cleaning the data and adding an OHE column.

Choose a filename, and write the latest DataFrame to a CSV (comma-separated value) file like this: `df_with_ohe.to_csv('..../Data/Chapter_1_cleaned_data.csv', index=False)`, where we don't include the index, as this is not necessary and can create extra columns when we load it later.

Exploring the Financial History Features in the Dataset

We are ready to explore the rest of the features in the case study dataset. We will first practice loading a DataFrame from the **CSV** file we saved at the end of the last section. This can be done using the following snippet:

```
df = pd.read_csv('..../Data/Chapter_1_cleaned_data.csv')
```

Note that if you are continuing to write code in the same notebook, this overwrites the value held by the **df** variable previously, which was the DataFrame of raw data. We encourage you to check the `.head()`, `.columns`, and `.shape` of the DataFrame. These are good things to check whenever loading a DataFrame. We don't do this here for the sake of space, but it's done in the companion notebook.

Note

The path to your CSV file may be different depending on where you saved it.

The remaining features to be examined are the financial history features. They fall naturally into three groups: the status of the monthly payments for the last six months, and the billed and paid amounts for the same period. First, let's look at the payment statuses. It is convenient to break these out as a list so we can study them together. You can do this using the following code:

```
pay_feats = ['PAY_1', 'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6']
```

We can use the `.describe` method on these six **Series** to examine summary statistics:

```
df[pay_feats].describe()
```

This should produce the following output:

```
df[pay_feats].describe()
```

	PAY_1	PAY_2	PAY_3	PAY_4	PAY_5	PAY_6
count	26664.000000	26664.000000	26664.000000	26664.000000	26664.000000	26664.000000
mean	-0.017777	-0.133363	-0.167679	-0.225023	-0.269764	-0.293579
std	1.126769	1.198640	1.199165	1.167897	1.131735	1.150229
min	-2.000000	-2.000000	-2.000000	-2.000000	-2.000000	-2.000000
25%	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000
50%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
75%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
max	8.000000	8.000000	8.000000	8.000000	8.000000	8.000000

Figure 1.55: Summary statistics of payment status features

Here, we observe that the range of values is the same for all of these features: -2, -1, 0, ... 8. It appears that the value of 9, described in the data dictionary as "payment delay for nine months and above", is never observed.

We have already clarified the meaning of all of these levels, some of which were not in the original data dictionary. Now let's look again at the `value_counts()` of PAY_1, now sorted by the values we are counting, which are the `index` of this **Series**:

```
df[pay_feats[0]].value_counts().sort_index()
```

This should produce the following output:

```
df[pay_feats[0]].value_counts().sort_index()

-2      2476
-1      5047
0      13087
1      3261
2      2378
3       292
4        63
5        23
6        11
7         9
8        17
Name: PAY_1, dtype: int64
```

Figure 1.56: Value counts of the payment status for the previous month

Compared to the positive integer values, most of the values are either -2, -1, or 0, which correspond to an account that was in good standing last month: not used, paid in full, or made at least the minimum payment.

Notice that, because of the definition of the other values of this variable (1 = payment delay for one month; 2 = payment delay for two months, and so on), this feature is sort of a hybrid of categorical and numerical features. Why should no credit usage correspond to a value of -2, while a value of 2 means a 2-month late payment, and so forth? We should acknowledge that the numerical coding of payment statuses -2, -1, and 0 constitute a decision made by the creator of the dataset on how to encode certain categorical features, which were then lumped in with a feature that is truly numerical: the number of months of payment delay (values of 1 and larger). Later on, we will consider the potential effects of this way of doing things on the predictive capability of this feature.

For now, we will simply continue to explore the data. This dataset is small enough, with 18 of these financial features and a handful of others, that we can afford to individually examine every feature. If the dataset had thousands of features, we would likely forgo this and instead explore **dimensionality reduction** techniques, which are ways to condense the information in a large number of features down to a smaller number of derived features, or, alternatively, methods of **feature selection**, which can be used to isolate the important features from a candidate field of many. We will demonstrate and explain some feature selection techniques later. But on this dataset, it's feasible to visualize every feature. As we know from the last chapter, a histogram is a good way to get a quick visual interpretation of the same kind of information we would get from tables of value counts. You can try this on the most recent month's payment status features with `df[pay_feats[0]].hist()`, to produce this:

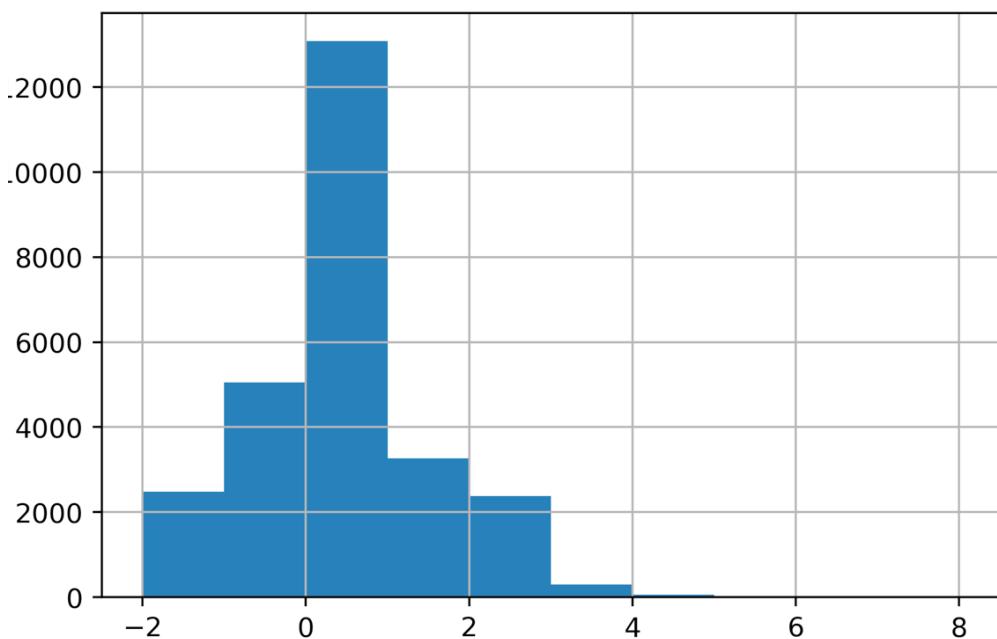


Figure 1.57: Histogram of PAY_1 using default arguments

Now we're going to take an in-depth look at how this graphic is produced and consider whether it is as informative as it should be. A key point about the graphical functionality of pandas is that **pandas plotting actually calls matplotlib under the hood**. Notice that the last available argument to the pandas `.hist()` method is `**kwds`, which the documentation indicates are **matplotlib** keyword arguments.

Note

For more information, refer to the following: <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.hist.html>.

Looking at the `matplotlib` documentation for `matplotlib.pyplot.hist` shows additional arguments you can use with the pandas `.hist()` method, such as the type of histogram to plot (see https://matplotlib.org/api/_as_gen/matplotlib.pyplot.hist.html for more details). In general, to get more details about plotting functionality, it's important to be aware of `matplotlib`, and in some scenarios, you will want to use `matplotlib` directly, instead of pandas, to have more control over the appearance of plots.

You should be aware that pandas uses `matplotlib`, which in turn uses NumPy. When plotting histograms with `matplotlib`, the numerical calculation for the values that make up the histogram is actually carried out by the NumPy `.histogram` function. This is a key example of code reuse, or "not reinventing the wheel". If a standard functionality, such as plotting a histogram, already has a good implementation in Python, there is no reason to create it anew. And the if mathematical operation to create the histogram data for the plot is already implemented, this should be leveraged as well. This shows the interconnectedness of the Python ecosystem.

We'll now address a couple of key issues that arise when calculating and plotting histograms.

Number of bins

Histograms work by grouping together values into what are called **bins**. The number of bins is the number of vertical bars that make up the discrete histogram plot we see. If there are a large number of unique values on a continuous scale, such as the histogram of ages we viewed earlier, histogram plotting works relatively well "out of the box", with default arguments. However, when the number of unique values is close to the number of bins, the results may be a little misleading. The default number of bins is 10, while in the `PAY_1` feature, there are 11 unique values. In cases like this, it's better to manually set the number of histogram bins to the number of unique values.

In our current example, since there are very few values in the higher bins of `PAY_1`, the plot may not look much different. But in general, this is important to keep in mind when plotting histograms.

Bin edges

The locations of the edges of the bins determine how the values get grouped in the histogram. Instead of indicating the number of bins to the plotting function, you could alternatively supply a list or array of numbers for the `bins` keyword argument. This input would be interpreted as the bin edge locations on the x axis. The way values are grouped into bins in `matplotlib`, using the edge locations, is important to understand. All bins, except the last one, group together values as low as the left edge, and up to **but not including** values as high as the right edge. In other words, the left edge is closed but the right edge is open for these bins. However, the last bin includes both edges; it has a closed left and right edge. This is of more practical importance when you are binning a relatively small number of unique values that may land on the bin edges.

For control over plot appearance, it's usually better to specify the bin edge locations. We'll create an array of 12 numbers, which will result in 11 bins, each one centered around one of the unique values of `PAY_1`:

```
pay_1_bins = np.array(range(-2,10)) - 0.5  
pay_1_bins
```

The output shows the bin edge locations:

```
pay_1_bins = np.array(range(-2,10)) - 0.5  
pay_1_bins  
  
array([-2.5, -1.5, -0.5,  0.5,  1.5,  2.5,  3.5,  4.5,  5.5,  6.5,  7.5,  
      8.5])
```

Figure 1.58: Specifying histogram bin edges

As a final point of style, it is important to always *label your plots* so that they are interpretable. We haven't yet done this manually, because in some cases, pandas does it automatically, and in other cases, we simply left the plots unlabeled. From now on, we will follow best practice and label all plots. We use the `xlabel` and `ylabel` functions in `matplotlib` to add axis labels to this plot. The code is as follows:

```
df[pay_feats[0]].hist(bins=pay_1_bins)  
plt.xlabel('PAY_1')  
plt.ylabel('Number of accounts')
```

The output should look like this:

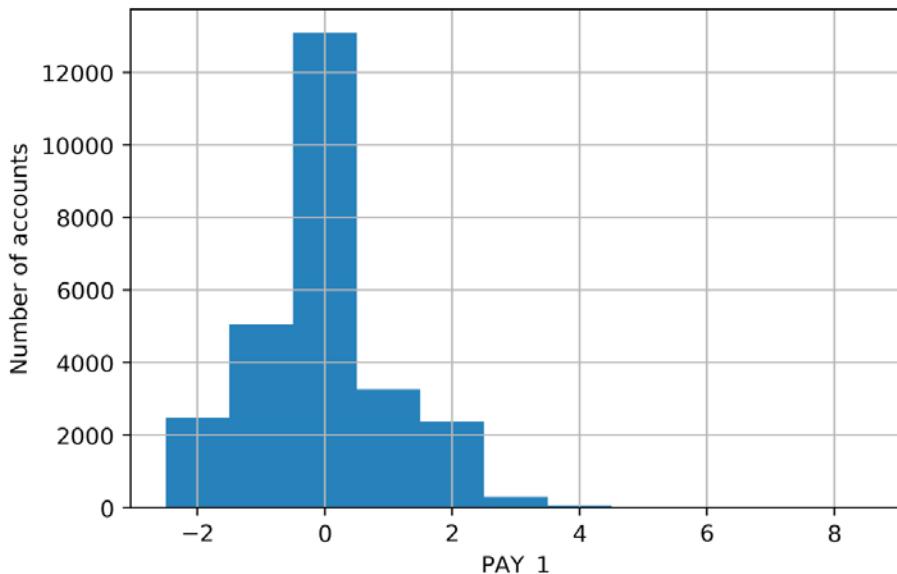


Figure 1.59: A better histogram of PAY_1

While it's tempting, and often sufficient, to just call plotting functions with the default arguments, one of your jobs as a data scientist is to create *accurate and representative data visualizations*. To do that, sometimes you need to dig into the details of plotting code, as we've done here.

What have we learned from this data visualization?

Since we already looked at the value counts, this confirms for us that most accounts are in good standing (values -2, -1, and 0). For those that aren't, it's more common for the "months late" to be a smaller number. This makes sense; likely, most people are paying off their balances before too long. Otherwise, their account may be closed or sold to a collection agency. Examining the distribution of your features and making sure it seems reasonable is a good thing to confirm with your client, as the quality of these data underlie the predictive modeling you seek to do.

Now that we've established some good plotting style for histograms, let's use pandas to plot multiple histograms together, and visualize the payment status features for each of the last six months. We can pass our list of column names `pay_feats` to access multiple columns to plot with the `.hist()` method, specifying the bin edges we've already determined, and indicating we'd like a 2 by 3 grid of plots. First, we set the font size small enough to fit between these subplots. Here is the code for this:

```
mpl.rcParams['font.size'] = 4  
df[pay_feats].hist(bins=pay_1_bins, layout=(2,3))
```

The plot titles have been created automatically for us based on the column names. The y axes are understood to be counts. The resulting visualizations are as follows:

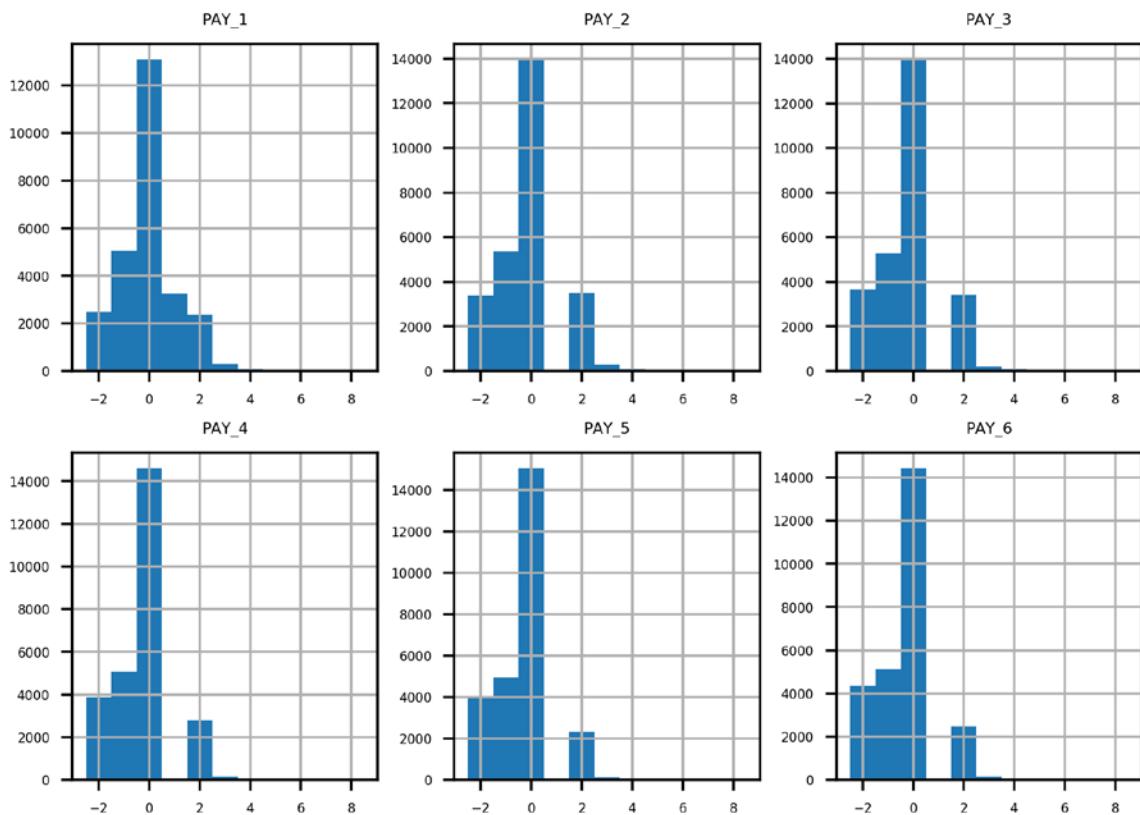


Figure 1.60: Grid of histogram subplots

We've already seen the first of these, and it makes sense. What about the rest of them? Remember the definitions of the positive integer values of these features, and what each feature means. For example, **PAY_2** is the repayment status in August, **PAY_3** is the repayment status in July, and the others go further back in time. A value of 1 means payment delay for 1 month, while a value of 2 means payment delay for 2 months, and so forth.

Did you notice that something doesn't seem right? Consider the values between July (**PAY_3**) and August (**PAY_2**). In July, there are very few accounts that had a 1-month payment delay; this bar is not really visible in the histogram. However, in August, there are suddenly thousands of accounts with a 2-month payment delay. This does not make sense: the number of accounts with a 2-month delay in a given month should be less than or equal to the number of accounts with a 1-month delay in the previous month. Let's take a closer look at accounts with a 2-month delay in August and see what the payment status was in July. We can do this with the following code, using a Boolean mask and `.loc`, as shown in the following snippet:

```
df.loc[df['PAY_2']==2, ['PAY_2', 'PAY_3']].head()
```

The output of this should appear as follows:

```
df.loc[df['PAY_2']==2, ['PAY_2', 'PAY_3']].head()
```

	PAY_2	PAY_3
0	2	-1
1	2	0
13	2	2
15	2	0
47	2	2

Figure 1.61: Payment status in July (PAY_3) of accounts with a 2-month payment delay in August (PAY_2)

From Figure 1.61, it's clear that accounts with a 2-month delay in August have nonsensical values for the July payment status. The only way to progress to a 2-month delay should be from a 1-month delay the previous month, yet none of these accounts indicate that.

When you see something like this in the data, you need to either check the logic in the query used to create the dataset or contact the person who gave you the dataset. After double-checking these results, for example using `.value_counts()` to view the numbers directly, we contact our client to inquire about this issue.

The client lets us know that they had been having problems with pulling the most recent month of data, leading to faulty reporting for accounts that had a 1-month delay in payment. In September, they had mostly fixed these problems (although not entirely; that is why there were missing values in the **PAY_1** feature, as we found). So, in our dataset, the value of 1 is underreported in all months except for September (the **PAY_1** feature). In theory, the client could create a query to look back into their database and determine the correct values for **PAY_2**, **PAY_3**, and so on up to **PAY_6**. However, for practical reasons, they won't be able to complete this retrospective analysis in time for us to receive it and include it in our analysis.

Because of this, only the most recent month of our payment status data is correct. This means that, of all the payment status features, only **PAY_1** is representative of future data, those that will be used to make predictions with the model we develop. This is a key point: *a predictive model relies on getting the same kind of data to make predictions that it was trained on*. This means we can use **PAY_1** as a feature in our model, but not **PAY_2** or the other payment status features from previous months.

This episode shows the importance of a thorough examination of data quality. Only by carefully combing through the data did we discover this issue. It would have been nice if the client had told us up front that they were having reporting issues over the last few months, when our dataset was collected, and that the reporting procedure was not **consistent** during that time period. However, ultimately it is our responsibility to build a credible model, so we need to be sure we believe the data is correct, by making this kind of careful examination. We explain to the client that we can't use the older features since they are not representative of the future data the model will be **scored** on (that is, make predictions on future months), and politely ask them to let us know of any further data issues they are aware of. There are none at this time.

Activity 1: Exploring Remaining Financial Features in the Dataset

In this activity, you will examine the remaining financial features in a similar way to how we examined PAY_1, PAY_2, PAY_3, and so on. In order to better visualize some of these data, we'll use a mathematical function that should be familiar: the logarithm. You'll use pandas to **apply**, which serves to apply any function to an entire column or DataFrame in the process. Once you complete the activity, you should have the following set of histograms of logarithmic transformations of non-zero payments:

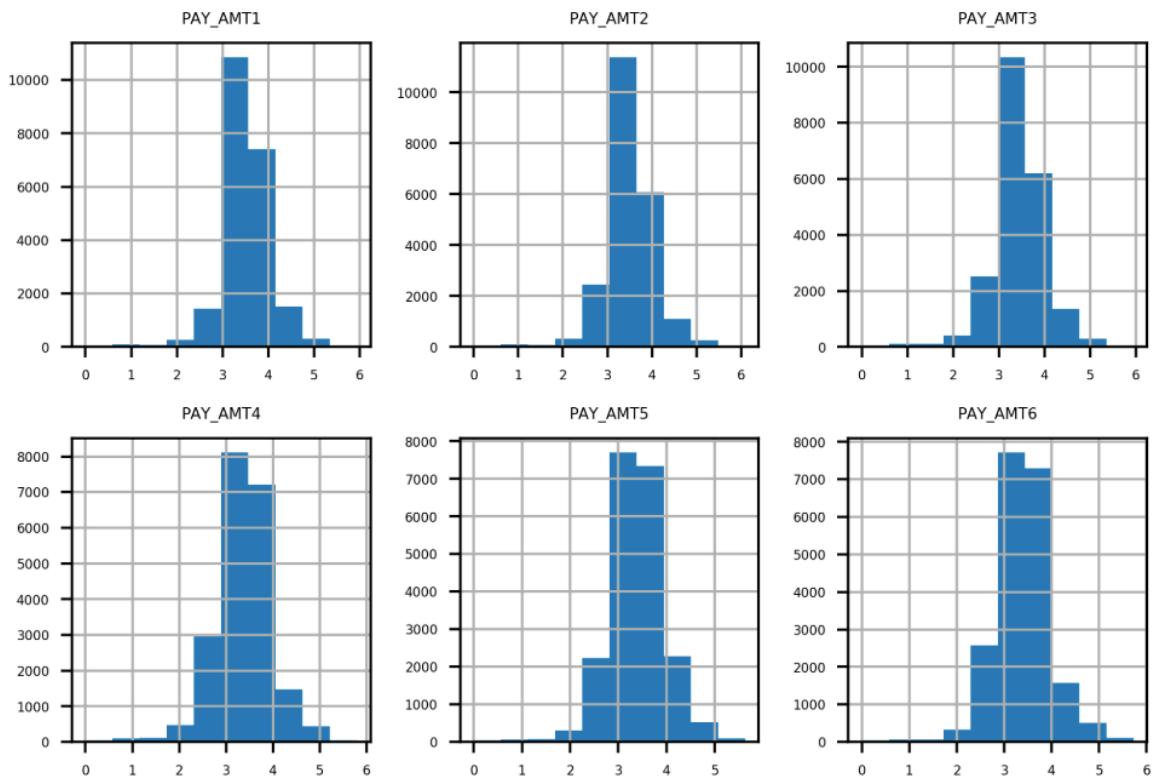


Figure 1.62: Expected set of histograms

Perform the following steps to complete the activity:

Note

The code and the resulting output graphics for this exercise have been loaded into a Jupyter Notebook that can be found here: <http://bit.ly/2TXZmrA>.

1. Create lists of feature names for the remaining financial features.
2. Use `.describe()` to examine statistical summaries of the bill amount features. Reflect on what you see. Does it make sense?
3. Visualize the bill amount features using a 2 by 3 grid of histogram plots.
Hint: You can use 20 bins for this visualization.
4. Obtain the `.describe()` summary of the payment amount features. Does it make sense?
5. Plot a histogram of the bill payment features similar to the bill amount features, but also apply some rotation to the x-axis labels with the `xrot` keyword argument so that they don't overlap. In any plotting function, you can include the `xrot=<angle>` keyword argument to rotate x-axis labels by a given angle in degrees. Consider the results.
6. Use a Boolean mask to see how many of the payment amount data are exactly equal to 0. Does this make sense given the histogram in the previous step?
7. Ignoring the payments of 0 using the mask you created in the previous step, use pandas `.apply()` and NumPy's `np.log10()` to plot histograms of logarithmic transformations of the non-zero payments. Consider the results.

Hint: You can use `.apply()` to apply any function, including `log10`, to all the elements of a DataFrame or a column using the following syntax:
`.apply(<function_name>).`

Note

The solution to this activity can be found on page 324.

Summary

This was the first chapter in our book, *Data Science Projects with Python*. Here, we made extensive use of pandas to load and explore the case study data. We learned how to check for basic consistency and correctness by using a combination of statistical summaries and visualizations. We answered such questions as "Are the unique account IDs truly unique?", "Is there any missing data that has been given a fill value?", and "Do the values of the features make sense given their definition?"

You may notice that we spent nearly all of this chapter identifying and correcting issues with our dataset. This is often the most time consuming stage of a data science project. While it is not always the most exciting part of the job, it gives you the raw materials necessary to build exciting models and insights. These will be the subjects of most of the rest of this book.

Mastery of software tools and mathematical concepts is what allows you execute data science projects, at a technical level. However, managing your relationships with clients, who are relying on your services to generate insights from their data, is just as important to a successful project. You must make as much use as you can of your business partner's understanding of the data. They are likely going to be more familiar with it than you, unless you are already a subject matter expert on the data for the project you are completing. However, even in that case, your first step should be a thorough and critical review of the data you are using.

In our data exploration, we discovered an issue that could have undermined our project: the data we had received was not internally consistent. Most of the months of the payment status features were plagued by a data reporting issue, included nonsensical values, and were not representative of the most recent month of data, or the data that would be available to the model going forward. We only uncovered this issue by taking a careful look at all of the features. While this is not always possible in different projects, especially when there is a very large number of features, you should always take the time to spot check as many features as you can. If you can't examine every feature, it's useful to check a few of every category of feature (if the features fall into categories, such as financial or demographic).

When discussing data issues like this with your client, make sure you are respectful and professional. The client may simply have forgotten about the issue when presenting you with the data. Or, they may have known about it but assumed it wouldn't affect your analysis for some reason. In any case, you are doing them an essential service by bringing it to their attention and explaining why it would be a problem to use flawed data to build a model. You should back up your claims with results if possible, showing that using the incorrect data either leads to decreased, or unchanged, model performance. Or, alternatively, you could explain that if only a different kind of data would be available in the future, compared to what's available now for training a model, the model built now will not be useful. Be as specific as you can, presenting the kinds of graphs and tables that we used to discover the data issue here.

In the next chapter, we will examine the response variable for our case study problem, which completes the initial data exploration. Then we will start to get some hands-on experience with machine learning models and learn how we can decide whether a model is useful or not.

2

Introduction to Scikit-Learn and Model Evaluation

Learning Objectives

By the end of this chapter, you will be able to:

- Explain the response variable
- Describe the implications of imbalanced data in binary classification
- Split data into training and testing sets
- Describe model fitting in scikit-learn
- Derive several metrics for binary classification
- Create an ROC curve and a precision-recall curve

This chapter will conclude the initial exploratory analysis and present new tools to perform model evaluation.

Introduction

The first chapter got you started with some basic Python, and then progressed to equipping you with tools for data exploration. Specifically, we performed operations such as loading the dataset and verifying data integrity, and we performed our first exploratory analysis on our case study dataset.

In this chapter, we finish our exploration of the data by examining the response variable. After we've concluded that the data is of high quality and makes sense, we will be ready to move forward with the practical concerns of developing machine learning models. We will take our first steps with scikit-learn, one of the most popular machine learning packages available in the Python language. Before learning the details of how mathematical models work in the next chapter, here we'll start to get comfortable with the syntax for using them in scikit-learn.

We will also learn some common techniques for how to answer the question, "Is this model good or not?" There are many possible ways to approach model evaluation. For business applications, some kind of financial analysis to determine the value that could be created by the model is usually necessary. However, we will reserve this for the end of the book.

There are several important model evaluation criteria that are considered standard knowledge in data science and machine learning. We will cover a few of the most widely used classification model performance metrics here, to give you a strong foundation.

Exploring the Response Variable and Concluding the Initial Exploration

We have now looked through all the **features** to see whether any data is missing, as well as to generally examine them. The features are important because they constitute the **inputs** to our machine learning algorithm. On the other side of the model lies the **output**, which is a prediction of the **response variable**. For our problem, this is a binary flag indicating whether or not an account will default the next month, which would have been October for our historical dataset.

Our overall job on this project is to come up with a predictive model for this target. Since the response variable is a yes/no flag, this problem is called a **binary classification** task. In our labeled data, the samples (accounts) that defaulted (that is, '`default payment next month`' = 1) are said to belong to the **positive class**, while those that didn't belong to the **negative class**. The key piece of information to examine regarding the response of a binary classification problem is: what is the proportion of the positive class? This is an easy check.

Before we perform this check, it is essential to load the required packages we will use in this chapter. This can be done with the following code:

```
import numpy as np #numerical computation
import pandas as pd #data wrangling
import matplotlib.pyplot as plt #plotting package
#Next line helps with rendering plots
%matplotlib inline
import matplotlib as mpl #add'l plotting functionality
mpl.rcParams['figure.dpi'] = 400 #high res figures
```

We will also require the cleaned version of the case study data. You can load it using the following code:

```
df = pd.read_csv('../Data/Chapter_1_cleaned_data.csv')
```

Note

The cleaned dataset should have been saved as a result of your work in *Chapter 1, Data Exploration and Cleaning*. The path to the cleaned data in the preceding code snippet may be different depending on your environment.

Now, to find the proportion of the positive class, all we need to do is get the average of the response variable over the whole dataset. This has the interpretation of the default rate. It's also worthwhile to check the number of samples in each class. This is presented in the following screenshot:

```
df['default payment next month'].mean()
```

```
0.2217971797179718
```

```
df.groupby('default payment next month')['ID'].count()
```

```
default payment next month
0      20750
1      5914
Name: ID, dtype: int64
```

Figure 2.1: Class balance of response variable

Since the target variable is **1** or **0**, taking the mean of this column indicates the fraction of accounts that defaulted: 22%. We also confirm the number of accounts in each class by doing a **groupby/count** operation. The proportion of samples in the positive class (default = 1), also called the **class fraction** for this class, is an important statistic. In binary classification, datasets are described in terms of being **balanced** or **imbalanced**: are the proportions of positive and negative samples equal, or not? Most machine learning classification models are designed to work with balanced data: a 50/50 split between the classes.

However, in practice, real data is rarely balanced. Consequently, there are several methods geared toward dealing with imbalanced data. These include the following:

- **Undersampling** the majority class: randomly throwing out samples from the majority class until the class fractions are equal, or at least less imbalanced.
- **Oversampling** the minority class: randomly adding duplicate samples of the minority class to achieve the same goal.
- **Weighting samples**: This method is performed as part of the training step, so the minority class collectively has as much "emphasis" as the majority class in the fitted model. The effect of this is similar to oversampling.

There are other more sophisticated methods, as well.

While our data is not, strictly speaking, balanced, we also note that a positive class fraction of 22% is not particularly imbalanced, either. Some domains, such as fraud detection, typically deal with much smaller positive class fractions: on the order of 1% or less. This is because the proportion of "bad actors" is quite small compared to the total population of transactions; at the same time, it is important to be able to identify them if possible. For problems like this, it is likely that using a method to address class imbalance will lead to substantially better results.

Now that we've explored the response variable, we have concluded our initial data exploration. However, data exploration should be considered an ongoing task that you should continually have in mind during any project. As you create models and generate new results, it's always good to think about what those results imply about the data, which usually requires a quick iteration back in to the exploration phase. A particularly helpful kind of exploration, which is also typically done before model building, is examining the relationship between features and the response. We gave a preview of that in *Chapter 1, Data Exploration and Cleaning*, when we were grouping by the **EDUCATION** feature and examining the mean of the response variable. We will also do more of this later. However, this has more to do with building a model than checking the inherent quality of the data.

The initial perusal through all the data, as we've done here, is an important foundation to lay at the beginning of a project. As you do this, you should ask yourself the following questions:

- Are the data **complete**?
Are there missing values or other anomalies?
- Are the data **consistent**?
Does the distribution change over time, and if so, is this expected?
- Do the data **make sense**?
Do the values of the features fit with their definition in the data dictionary?

The latter two questions help you determine whether you think the data are **correct**. If the answer to any of these questions is "no," this should be addressed before continuing with the project.

Also, if you think of any alternative, or additional data that might be helpful to have, and possible to get, now would be a good point in the project life cycle to augment your data set with them. Examples of this may include postal-code-level demographic data, which you could **join** to your data set, if you had the addresses associated with accounts. We don't have these for the case study data and have decided to proceed on this project with the data we have now.

Introduction to Scikit-Learn

While pandas will save you a lot of time in loading, examining, and cleaning data, the machine learning algorithms that will enable you to do predictive modeling are located in other packages. We consider scikit-learn to be the premier machine learning package for Python, outside of deep learning. While it's impossible for any one package to offer "everything," scikit-learn comes pretty close in terms of accommodating a wide range of approaches for classification and regression, and unsupervised learning. That being said, a few other packages you should also be aware of are as follows:

SciPy:

- Most of the packages we've used so far are actually part of the SciPy ecosystem.
- SciPy itself offers lightweight functions for classical approaches such as linear regression and linear programming.

StatsModels:

- More oriented toward statistics and more comfortable for users familiar with R
- Can get p-values and confidence intervals on regression coefficients
- Capability for time series models such as ARIMA

XGBoost:

- Offers a state-of-the art ensemble model that often outperforms random forests

TensorFlow, Keras, and PyTorch:

- Deep learning capabilities

There are many other Python packages that may come in handy, but this gives you an idea. Now, on to scikit-learn. Scikit-learn offers a wealth of different models for regression and classification (as well as unsupervised learning), but, conveniently, the syntax for using them is consistent. In this section, we will illustrate model syntax using a **logistic regression** model. Logistic regression, despite its name, is actually a classification model. This is one of the simplest (and therefore most important) classification models. We will go through the mathematical details of how logistic regression works, later. Until then, you can simply think of it as a black box that can learn from labeled data, then make predictions.

Earlier, we became familiar with the concept of training an algorithm on data, so that you might use this trained model to then make predictions on new data. Scikit-learn encapsulates these core functionalities in the `.fit` method for training models, and the `.predict` method for making predictions. Because of the consistent syntax, you can call `.fit` and `.predict` on any scikit-learn model from linear regression to classification trees.

The first step is to choose some model, in this example a logistic regression, and instantiate it from the **class** provided by scikit-learn. This means you are taking the blueprint of the model that scikit-learn makes available to you and creating a useful **object** out of it. You can train this object on your data, and then save it to disk for later use. The following snippets can be used to perform this task. The first step is to import the class:

```
from sklearn.linear_model import LogisticRegression
```

The code to instantiate the class in to an object is as follows:

```
my_lr = LogisticRegression()
```

The object is now a variable in our workspace. We can examine it using the following code:

```
my_lr
```

This should give the following output:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, max_iter=100, multi_class='warn',
                   n_jobs=None, penalty='l2', random_state=None, solver='warn',
                   tol=0.0001, verbose=0, warm_start=False)
```

Figure 2.2: Instantiating a model in scikit-learn

Notice that after we've created the model object, we've examined it and can see a lot of output. This output represents the **default options** that go along with the logistic regression model. In effect, these are choices we have made regarding the details of model implementation, without having been aware of it. The danger in an easy-to-use package such as scikit-learn is that it has the potential to obscure these choices from you; we were able to instantiate the model without specifying any of them. However, any time you use a machine learning model that has been prepared for you as scikit-learn models have been, your first job is to understand each and every option that is available. A best practice in such cases is to explicitly provide every keyword parameter to the model when you create the object. Even if you are just selecting all the default options, this will help increase your awareness of the decisions that are being made.

Here is the code for instantiating a logistic regression model with all default options:

```
my_new_lr = LogisticRegression(C=1.0, class_weight=None, dual=False, fit_
                               intercept=True,
                               intercept_scaling=1, max_iter=100, multi_class='warn',
                               n_jobs=None, penalty='l2', random_state=None, solver='warn',
                               tol=0.0001, verbose=0, warm_start=False)
```

Even though the object we've created here in `my_new_lr` is identical to `my_lr`, being explicit like this is especially helpful as you are starting out and learning about different kinds of models. Once you're more comfortable, you may wish to just instantiate with the default options and make changes later as necessary. Here, we show how this may be done. The following code sets two options and displays the current state of the model object:

```
my_new_lr.C = 0.1
my_new_lr.solver = 'liblinear'
my_new_lr
```

This should produce the following:

```
LogisticRegression(C=0.1, class_weight=None, dual=False, fit_intercept=True,
    intercept_scaling=1, max_iter=100, multi_class='warn',
    n_jobs=None, penalty='l2', random_state=None, solver='liblinear',
    tol=0.0001, verbose=0, warm_start=False)
```

Figure 2.3: Updating model hyperparameter and solver

Here, we've taken what is called a **hyperparameter** of the model, **C**, and updated it from its default value of **1** to **0.1**. We've also specified a solver. At this point you may be curious about what a hyperparameter is, what **C** means here, and for that matter what do all these other options mean? That is good; you should be curious about these things. Suffice to say now that a hyperparameter is like an option that you supply to the model, before fitting it to the data. Later, we will explain in detail what all the options here are and how you can effectively choose values for them.

For now, simply to illustrate the core functionality, we will blindly forge ahead and fit this nearly-default logistic regression to some data. As you know by now, supervised learning algorithms rely on labeled data. That means we need both the features, customarily contained in a variable called **X**, and the corresponding responses, in a variable called **y**. We will borrow the first 10 samples of one feature, and the response, from our data set to illustrate:

```
X = df['EDUCATION'][0:10].values.reshape(-1,1)
X
```

That should show the values of the EDUCATION feature for the first 10 samples:

```
array([[2],
       [2],
       [2],
       [2],
       [2],
       [1],
       [1],
       [2],
       [3],
       [3]])
```

Figure 2.4: First 10 values of a feature

The corresponding first 10 values of the response variable can be obtained as follows:

```
y = df['default payment next month'][0:10].values  
y  
  
array([1, 1, 0, 0, 0, 0, 0, 0, 0, 0])
```

Figure 2.5: First 10 values of the response variable

Here, we have selected a couple of series (that is, columns) from our DataFrame: the **EDUCATION** feature we've been discussing, and the response variable. Then we selected the first 10 elements of each, and finally used the `.values` method to return NumPy arrays. Also notice that we used the `.reshape` method to reshape the features. Scikit-learn expects that the first dimension (that is, the number of rows) of the array of features will be the number of samples, so we needed to make that reshaping for `X`, but not for `y`. The `-1` in the first positional argument of `.reshape` means to make the output array shape flexible in that dimension, according to how much data goes in. Since we just have a single feature in this example, we specified the number of columns as the second argument, `1`, and let the `-1` argument indicate that the array should "fill up" along the first dimension with as many elements as necessary to accommodate the data, in this case 10 elements. Note that while we've extracted the data in to NumPy arrays to show how this can be done, it's also possible to use pandas series as input to scikit-learn.

Let's now use this data to fit our logistic regression. This is accomplished with just one line:

```
my_new_lr.fit(X, y)
```

```
my_new_lr.fit(X, y)  
  
LogisticRegression(C=0.1, class_weight=None, dual=False, fit_intercept=True,  
                   intercept_scaling=1, max_iter=100, multi_class='warn',  
                   n_jobs=None, penalty='l2', random_state=None, solver='liblinear',  
                   tol=0.0001, verbose=0, warm_start=False)
```

Figure 2.6: Fitting a model in scikit-learn

That's all there is to it! Once your data is prepared, fitting a model almost seems like an afterthought here. Of course, we are ignoring all the important options and what they mean right now. But, technically speaking, fitting a model is very easy in terms of the code. You can see that the output of this cell just prints the options again. While we have not returned anything from the fitting procedure aside from this output, a very important change has taken place. The `my_new_lr` model object is now a trained model. We say that this change happened **in place** since no new object was created; the existing object, `my_new_lr`, has been modified. This is similar to modifying a DataFrame in place. We can now use our trained model to make predictions on new features, that the model has never "seen" before. Let's try the next 10 rows from the `EDUCATION` feature.

We can select and view these features using a new variable, `new_X`:

```
new_X = df['EDUCATION'][10:20].values.reshape(-1,1)  
new_X
```

```
array([[3],  
       [1],  
       [2],  
       [2],  
       [1],  
       [3],  
       [1],  
       [1],  
       [1],  
       [3]])
```

Figure 2.7: New features to make predictions for

Making predictions is done like this:

```
my_new_lr.predict(new_X)  
  
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Figure 2.8: Predictions on the new features

We can also view what the true values corresponding to these predictions are:

```
df['default payment next month'][10:20].values  
  
array([0, 0, 0, 1, 0, 0, 1, 0, 0, 0])
```

Figure 2.9: The true labels of the predictions

Here, we've illustrated several things. After getting our new feature values, we've called the `.predict` method on the trained model. Notice that the only argument to this method is a set of features, that is, an "X" that we've called `new_X`. There is no "y" supplied in this case. In fact, that is the whole point of predictive modeling: you don't necessarily know the true value of the response variable, so you need to be able to predict it. We see that the output of the `.predict` method is the model's predictions for those samples. We then compare these predictions to the actual true values of the response variable, which we pull from the DataFrame.

So, how did our blindly constructed model do? We may naively observe that since the model predicted all 0s, and 80% of the true labels were 0s, we were right 80% of the time, which seems pretty good. On the other hand, we entirely failed to successfully predict any 1s. So, if those were important, we did not actually do that well at all. While this is just a toy example to get you familiar with how scikit-learn works, it's worth considering what a "good" prediction might look like for this problem. We will get into the details of assessing model predictive capability shortly. For now, congratulate yourself on having gotten your hands dirty with some real data, and fitting your first machine learning model.

Generating Synthetic Data

You are now familiar with a variety of Python packages. In fact, these are enough to complete a huge range of data science projects, from classification and regression, to unsupervised learning. And you have seen how fitting a model works in scikit-learn, one of the most useful machine learning packages in Python. In the following exercise, you will walk through the model fitting process on your own.

In order to have data to fit, you will actually generate your own **synthetic data**. Synthetic data is a valuable learning tool for illustrating mathematical concepts. In order to make synthetic data, we will again illustrate here how to use NumPy's `random` library to generate random numbers, as well as matplotlib's `scatter` and `plot` functions to create scatter and line plots, respectively. In the exercise, we'll use scikit-learn for the linear regression part. We don't go in to the details of linear regression here, although you are doubtless familiar with the concept of a **line of best fit**, which is the same thing.

To get started, we use NumPy to make a one-dimensional array of feature values, X , consisting of 1,000 random real numbers (in other words not just integers but decimals as well) between 0 and 10. We again use a **seed** for the random number generator. Next, we use **numpy.random.uniform**, which draws from the uniform distribution: it's equally likely to choose any number between **low** (inclusive) and **high** (exclusive), and will return an array of whatever **size** you specify. We create a one-dimensional array (that is, a vector) with 1,000 elements, then examine the first 10. All of this can be done using the following code:

```
np.random.seed(seed=1)  
X = np.random.uniform(low=0.0, high=10.0, size=(1000,))  
X[0:10]
```

The output should appear as follows:

```
array([4.17022005e+00, 7.20324493e+00, 1.14374817e-03, 3.02332573e+00,  
       1.46755891e+00, 9.23385948e-01, 1.86260211e+00, 3.45560727e+00,  
       3.96767474e+00, 5.38816734e+00])
```

Figure 2.10: Creating random, uniformly distributed numbers with NumPy

Data for a Linear Regression

Now we need a response variable. For this example, we'll generate data that follows the assumptions of linear regression: data that follows a linear trend but has normally distributed errors. It is unlikely that a real-world data set would satisfy the formal statistical assumptions of a model like this, but you can create a synthetic data set that essentially does. Linear regression theoretically should only be used to model data where the response is a linear transformation of the features, with normally distributed (also called Gaussian) noise:

$$y = ax + b + N(\mu, \sigma)$$

Figure 2.11: Linear equation with Gaussian noise

Here a is the slope, b is the intercept, and the Gaussian noise has a mean of μ with a standard deviation of σ . In order to write code to implement this, we need to make a corresponding vector of responses, y , which are calculated as a slope times the feature array, X , plus some Gaussian noise (again using NumPy), and an intercept.

To create the linear regression data, we first set the random seed. Then we declare variables for the slope and intercept of our linear data. Finally, we create the response variable using the familiar equation for a line, with the addition of some Gaussian noise: an array of 1,000 data points the same shape (`size`) as the feature array, `X`, where the mean of the noise (`loc`) is 0 and the standard deviation (`scale`) is 1. This will add a little "spread" to our linear data. The code to create linear data with Gaussian noise is as follows:

```
np.random.seed(seed=1)  
slope = 0.25  
intercept = -1.25  
y = slope * X + np.random.normal(loc=0.0, scale=1.0, size=(1000,)) +  
    intercept
```

Now we'd like to visualize these data. We will use matplotlib to plot `y` against the feature `X` as a scatter plot. First we use `.rcParams` to set the resolution (`dpi` = dots per inch) for a nice crisp image. Then we create the scatter plot with `plt.scatter`, where the `X` and `y` data are the first two arguments, respectively, and the `s` argument takes a size for the dots.

This code can be used for plotting:

```
mpl.rcParams['figure.dpi'] = 400  
plt.scatter(X,y,s=1)
```

After executing these cells, you should see something like this in your notebook:

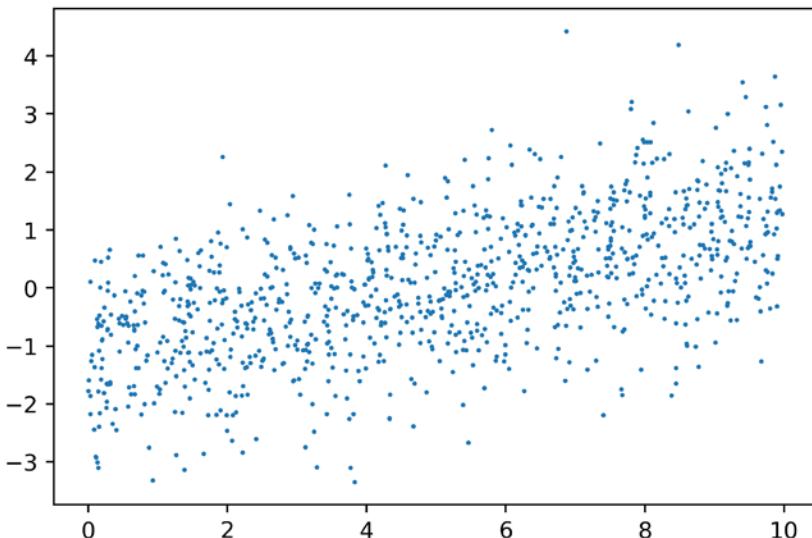


Figure 2.12: Plot the noisy linear relationship

Looks like some noisy linear data, just like we hoped! Now we need to model it.

Exercise 8: Linear Regression in Scikit-Learn

In this exercise, we will take the synthetic data we just generated and determine a line of best fit, or linear regression, using scikit-learn. The first step is to import a linear regression model class from scikit-learn and create an object from it. The import is similar to the `LogisticRegression` class we imported previously. As with any model class, you should observe what all the default options are. Notice that here for a linear regression, there are not that many options to specify: you may use the defaults for this exercise. The default settings include `fit_intercept=True`, meaning the regression model will include an intercept term. This is certainly appropriate since we added an intercept to the synthetic data. Perform the following steps to complete the exercise:

Note

For Exercises 8–10 and Activity 2, the code and the resulting output have been loaded in a Jupyter Notebook that can be found here: <http://bit.ly/2UWPgYo>. You can scroll to the appropriate section within the Jupyter Notebook to locate the exercise or activity of choice.

1. Execute this code to import the linear regression model class and instantiate it:

```
from sklearn.linear_model import LinearRegression  
lin_reg = LinearRegression()  
lin_reg
```

You should see the following output:

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,  
normalize=False)
```

Figure 2.13: Working with the linear regression class

Now we can fit the model using our synthetic data, remembering to reshape the feature array, as we did earlier, so that that samples are along the first dimension. After fitting the linear regression model, we examine `lin_reg.intercept_`, which contains the intercept of the fitted model, just as one would think, as well as `lin_reg.coef_`, which contains the slope.

- Run this code to fit the model and examine the coefficients:

```
lin_reg.fit(X.reshape(-1,1), y)
print(lin_reg.intercept_)
print(lin_reg.coef_)
```

You should see this output for the intercept and slope:

```
-1.161256600282589
[0.24002588]
```

Figure 2.14: Fitting the regression and examining the parameters

We again see that actually fitting a model in scikit-learn, once the data is prepared and the options for the model are decided, is a trivial process. This is because all the algorithmic work of determining the model parameters is abstracted away from the user. We will discuss some of these details later in the case of the logistic regression model we'll use on the case study data.

What about the slope and intercept of our fitted model?

These numbers are fairly close to the slope and intercept we indicated when creating the model. However, because of the random noise, they are only approximations.

Finally, we can use the model to make predictions on feature values. Here, we do this using the same data used to fit the model: the array of features, `X`. We capture the output of this as a variable, `y_pred`: This is very similar to our previous case, only here we put the output of the `.predict` method into a variable.

- Run this code to make predictions:

```
y_pred = lin_reg.predict(X.reshape(-1,1))
```

We can plot the predictions, `y_pred`, against feature `X` as a line plot, over the scatter plot of the feature and response data like we made in Step 4. Here we repeat Step 4, with the addition of `plt.plot`, which produces a line plot by default, to plot the feature and the model-predicted response values. Notice that we follow the `X` and `y` data with '`r`' in our call to `plt.plot`. This keyword argument causes the line to have a red color and is part of a shorthand syntax for plot formatting.

4. This code can be used to plot the raw data, as well as the fitted model predictions on these data:

```
plt.scatter(X,y,s=1)  
plt.plot(X,y_pred,'r')
```

After executing this cell, you should see something like this:

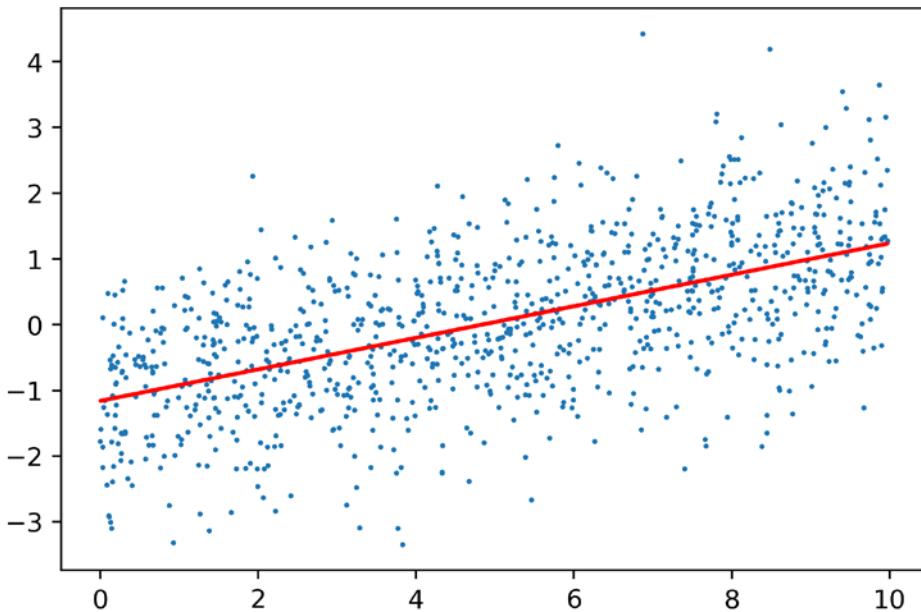


Figure 2.15: Plotting the data and the regression line

The plot shows what the line of best fit to the data looks like, just as we'd expect.

In this exercise, as opposed to when we called `.predict` with logistic regression, we made predictions on the same data `X` that we used to train the model. This is an important distinction. While here, we are seeing how the model "fits" the same data that it was trained on, we previously examined model predictions on new, unseen data. In machine learning, we are usually concerned with predictive capabilities: we want models that can help us know the likely outcomes of future scenarios. However, it turns out that model predictions on both the **training data** used to fit the model, and the **testing data**, which was not used to fit the model, are important for understanding the workings of the model. We will formalize these notions later in Chapter 4, *The Bias-Variance Trade-off* when we discuss the **bias-variance tradeoff**.

Model Performance Metrics for Binary Classification

Before we start building predictive models in earnest, we would like to know how we can determine, once we've created a model, whether it is "good" in some sense of the word. As you may imagine, this question has received a lot of attention from researchers and practitioners. Consequently, there is a wide variety of model performance metrics to choose from.

Note

For an idea of the range of options, have a look at the scikit-learn model evaluation page: https://scikit-learn.org/stable/modules/model_evaluation.html#model-evaluation.

When selecting a model performance metric to assess the predictive quality of a model, it's important to keep two things in mind.

Appropriateness of the metric for the problem

Metrics are typically only defined for a specific class of problems, such as classification or regression. For a binary classification problem, several metrics characterize the correctness of the yes or no question that the model answers. An additional level of detail here is how often the model is correct for each class, the positive and negative classes. We will go in to detail on these metrics here. On the other hand, regression metrics are aimed at measuring how close a prediction is to the target quantity. If we are trying to predict the price of a house, how close did we come? Are we systematically over- or under-estimating? Are we getting the more expensive houses wrong but the cheaper ones, right? There are many possible ways to look at regression metrics.

Does the metric answer the business question?

Whatever class of problem you are working on, there will be many choices for the metric. Which one is the right one? And even then, how do you know if a model is "good enough" in terms of the metric? At some level, this is a subjective question. However, we can be objective when we consider what the goal of the model is. In a business context, typical goals are to create profit or reduce loss. Ultimately, you need to unify your business question, which is often related to money in some way, and the metric you will use to judge your model. For example, in our credit default problem, is there a particularly high cost associated with not correctly identifying accounts that will default? Is this more important than correctly identifying accounts that won't default?

Later in the book, we'll incorporate the concept of relative costs and benefits of correct and incorrect classifications in our problem and conduct a financial analysis. First, we'll introduce you to the most common metrics used to assess predictive quality of binary classification models, the kind of models we need to build for our case study.

Splitting the Data: Training and Testing sets

In the scikit-learn introduction of this chapter, we introduced the concept of using a trained model to make predictions on new data that the model had never "seen" before. It turns out this is a foundational concept in predictive modeling. In our quest to create a model that has predictive capability, we need some kind of measure of how well the model can make predictions on data that were not used to fit the model. This is because in fitting a model, the model becomes "specialized" at learning the relationship between features and response, on the specific set of labeled data that were used for fitting. While this is nice, in the end we want to be able to use the model to make accurate predictions on new, unseen data, for which we don't know the true value of the labels.

For example, in our case study, once we deliver the trained model to our client, they will then generate a new data set of features like those we have now, except instead of spanning the period from April to September, they will span from May to October. And our client will be using the model with these features, to predict whether accounts will default in November.

In order to have knowledge of how well we can expect our model to predict which accounts will actually default in November (which won't be known until December), we can take our current data set, and reserve some of the data we have, with known labels, from the model training process. This data is referred to as **testing data** and may also be called **out-of-sample data** since it consists of samples that were not used in training the model. Those samples used to train the model are called **training data**. The practice of holding out a set of testing data gives us an idea of how the model will perform when it is used for its intended purpose, to make predictions on samples that were not included during model training. In this chapter we'll create an example train/test split set to illustrate different binary classification metrics.

We will use the convenient `train_test_split` functionality of scikit-learn to split the data so that 80% will be used for training, holding 20% back for testing. These percentages are a common way to make the split; in general, you want enough training data to allow the algorithm to adequately "learn" from a representative sample of data. However, these percentages are not set in stone. If you have a very large number of samples, you may not need as large a percentage of training data, since you will be able to achieve a pretty large, representative training set with a lower percentage. We encourage you to experiment with different sizes and see the effect. Also, be aware that every problem is different, with respect to how much data is needed to effectively train a model. There is no hard and fast rule for sizing your training and testing sets. Consider the code shown in the following snippet:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    df['EDUCATION'].values.reshape(-1,1), df['default payment next month'].values,
    test_size=0.2, random_state=24)
```

Notice in the preceding snippet that we've indicated the `test_size` as 0.2, or 20%. The size of the training data will be automatically set to the remainder, 80%. Let's examine the shapes of our training and testing data, to see whether they are as expected as shown in the following output:

```
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)

(21331, 1)
(5333, 1)
(21331,)
(5333,)
```

Figure 2.16: Shape of training and testing sets

You should confirm for yourself that the number of samples (rows) in the training and testing sets is consistent with an 80/20 split.

In making the train/test split, we've also set the `random_state` parameter, which is a random number seed. Using this parameter allows for a consistent train/test split across runs of this notebook. Otherwise, the random splitting procedure would select a different 20% of the data for testing each time the code was run. Finally, observe that similar to our initial illustration of model fitting, we are still only going to use a single feature here: `EDUCATION`. We are doing this just to show how to split data and calculate different model performance metrics. We will try other features later when we go in to detail on how different machine learning models work, and seek to create the best model we can to deliver to our client.

The first argument to `train_test_split` is the features, in this case just `EDUCATION`, and the second argument is the response. There are four outputs: the features of the samples in the training and testing sets, respectively, and the corresponding response variables that go with these sets of features. All this function has done is randomly select 20% of the row indices from the data set and subset out these features and responses as testing data, leaving the rest as training. Now that we have our training and testing data, it's good to make sure the nature of the data is the same between these sets. In particular, is the fraction of the positive class similar? You can observe this in the following output:

```
np.mean(y_train)
```

```
0.223102526838873
```

```
np.mean(y_test)
```

```
0.21657603600225014
```

Figure 2.17: Class fractions in training and testing data

The positive class fractions in the training and testing data are both about 22%. This is good, as these are the same as the overall data, and we can say that the training set is representative of the testing set. In this case, since we have a pretty large data set with tens of thousands of samples, and the classes are not too imbalanced, we don't have to take precautions to ensure this happens.

However, you can imagine that if the data set is smaller, and the positive class is very rare, it may happen that the class fractions are noticeably different between the training and testing sets, or worse yet, there are no positive samples at all in the testing set! In order to guard against such scenarios, you could use **stratified sampling**, with the **stratify** keyword argument of **train_test_split**. This procedure also makes a random split of the data into training and testing sets but guarantees that the class fractions will be equal or very similar between the two.

Note

Out-of-time testing

If your data contains both features and responses that span a substantial period of time, it's a good practice to try making your train/test split over time. For example, if you have two years of data with features and responses from every month, you may wish to try sequentially training the model on 12 months of data and testing on the next month, or the month after that, depending on what is operationally feasible when the model will be used. You could repeat this until you've exhausted your data, to get a few different testing scores. This will give you useful insight into model performance, because it simulates the actual conditions the model will face when it is deployed: a model trained on older features and responses will be used to make predictions on newer data. In the case study, the responses only come from one point in time (credit defaults within one month), so this is not an option here.

Classification Accuracy

Now we proceed to fit an example model, to illustrate binary classification metrics. We will continue to use a logistic regression with near-default options, choosing the same options we demonstrated in *Chapter 1, Data Exploration and Cleaning*:

```
from sklearn.linear_model import LogisticRegression

example_lr = LogisticRegression(C=0.1, class_weight=None, dual=False, fit_intercept=True,
                                intercept_scaling=1, max_iter=100, multi_class='warn',
                                n_jobs=None, penalty='l2', random_state=None, solver='liblinear',
                                tol=0.0001, verbose=0, warm_start=False)
```

Figure 2.18: Loading the model class and creating a model object

Now we proceed to train the model, as you might imagine, using the labeled data from our training set. We proceed immediately to use the trained model to make predictions on the features of the samples from the held-out test set:

```
example_lr.fit(X_train, y_train)

LogisticRegression(C=0.1, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, max_iter=100, multi_class='warn',
                   n_jobs=None, penalty='l2', random_state=None, solver='liblinear',
                   tol=0.0001, verbose=0, warm_start=False)

y_pred = example_lr.predict(X_test)
```

Figure 2.19: Training a model and making predictions on the test set

We've stored the model-predicted labels of the test set in a variable called `y_pred`. How should we now assess the quality of these predictions? We have the true labels, in the `y_test` variable. First, we will compute what is probably the simplest of all binary classification metrics: **accuracy**. Accuracy is defined as the proportion of samples that were correctly classified.

One way to calculate accuracy is to create a logical mask that is `True` whenever the predicted label is equal to the actual label, and `False` otherwise. We can then take the average of this mask, which will interpret True as 1 and False as 0, giving us the proportion of correct classifications:

```
is_correct = y_pred == y_test

np.mean(is_correct)

0.7834239639977498
```

Figure 2.20: Calculating classification accuracy with a logical mask

This indicates the model is correct 78% of the time. While this is a pretty straightforward calculation, there are actually easier ways to calculate accuracy using the convenience of scikit-learn. One way is to use the trained model's `.score` method, passing the features of the testing data to make predictions on, and the testing labels. This method makes the predictions and then does the same calculation we performed previously. Or, we could import scikit-learn's `metrics` library, which includes many model performance metrics including `accuracy_score`. For this, we pass the true labels and the predicted labels:

```
example_lr.score(X_test, y_test)
```

```
0.7834239639977498
```

```
from sklearn import metrics
```

```
metrics.accuracy_score(y_test, y_pred)
```

```
0.7834239639977498
```

Figure 2.21: Calculating classification accuracy with scikit-learn

These all give the same result, as they should. Now that we know how accurate the model is, how do we interpret this metric? On the surface, an accuracy of 78% may sound good. We are getting most of the predictions right. However, an important test for the accuracy of binary classification is to compare things to a very simple hypothetical model that only makes one prediction: this hypothetical model predicts the majority class for every sample, no matter what the features are. While in practice this model is useless, it provides an important extreme case with which to compare the accuracy of our trained model. Such extreme cases are sometimes referred to as null models.

Think about what the accuracy of such a null model would be. In our data set, we know that about 22% of the samples are positive. So, the negative class is the majority class, with the remaining 78% of the samples. Therefore, a null model for this data set, that always predicts the majority negative class, will be right 78% of the time. Now when we compare our trained model here to such a null model, it becomes clear that an accuracy of 78% is actually not very useful. We can get the same accuracy with a model that doesn't pay any attention to the features!

While we can interpret accuracy in terms of a majority-class null model, there are other binary classification metrics that delve a little deeper in to how the model is performing for negative, versus positive samples.

True Positive Rate, False Positive Rate, and Confusion Matrix

In binary classification, there are just two labels to consider: positive and negative. As a more descriptive way to look at model performance than the accuracy of prediction across all samples, we can also look at the accuracy of only those samples that have a positive label. The proportion of these that we successfully predict as positive, is called the **true positive rate (TPR)**. If we say that **P** is the number of samples in the **positive class** in the testing data, and **TP** is the number of **true positives**, defined as the number of positive samples that were predicted to be positive by the model, then the TPR is as follows:

$$TPR = \frac{TP}{P}$$

Figure 2.22: True positive rate equation

The flip side of the true positive rate is the **false negative rate (FNR)**. This is the proportion of positive testing samples that we incorrectly predicted as negative. Such errors are called **false negatives (FN)** and the false negative rate (FNR) is calculated as follows:

$$FNR = \frac{FN}{P}$$

Figure 2.23: False negative rate equation

Since all the positive samples are either correctly or incorrectly predicted, the sum of the number of true positives and the number of false negatives equals the total number of positive samples. Mathematically, $P = TP + FN$ and therefore, using the definitions of TPR and FNR, we have the following:

$$TPR + FNR = 1$$

Figure 2.24: The relation between TPR and FNR

Since the TPR and FNR sum to 1, it's usually sufficient to just calculate one of them.

Similar to the TPR and FNR, there is the **true negative rate (TNR)** and the **false positive rate (FPR)**. If N is the number of **negative** samples, the sum of **true negative** samples (**TN**) is the number of these that are correctly predicted, and the sum of **false positive** (**FP**) samples is the number incorrectly predicted as positive:

$$TNR = \frac{TN}{N}$$

Figure 2.25: True negative rate equation

$$FPR = \frac{FP}{N}$$

Figure 2.26: False positive rate equation

$$TNR + FPR = 1$$

Figure 2.27: Relation between true negative rate and false positive rate

True and false positives and negatives can be conveniently summarized in a table called a **confusion matrix**. A confusion matrix for a binary classification problem is a 2×2 matrix where the true class is along one axis and the predicted class is along the other. The confusion matrix gives a quick summary of how many true and false positives and negatives there are:

		Predicted class	
		N	P
True class	N	TN	FP
	P	FN	TP

Figure 2.28: The confusion matrix for binary classification

Since we hope to make correct classifications, we hope that the **diagonal** (that is, the entries along a diagonal line from the top left to the bottom right: TN and TP) of the confusion matrix are relatively large, while the off-diagonals are relatively small, as these represent incorrect classifications.

Exercise 9: Calculating the True and False Positive and Negative Rates and Confusion Matrix in Python

In this exercise, we'll use the testing data and model predictions from the logistic regression model we created previously, using only the **EDUCATION** feature. We will illustrate how to manually calculate the true and false positive and negative rates, as well as the numbers of true and false positives and negatives needed for the confusion matrix. Then we show a quick way to calculate a confusion matrix with scikit-learn. Perform the following steps to complete the exercise:

Note

The code and the resulting output for this exercise have been loaded in a Jupyter Notebook that can be found here: <http://bit.ly/2UWPgYo>.

1. Run this code to calculate the number of positive samples:

```
P = sum(y_test)  
P
```

The output should appear like this:

1155

Figure 2.29: Calculating the number of positives

Now we need the number of true positives. These are samples where the true label is 1 and the prediction is also 1. We can identify these with a logical mask for the samples that are positive (`y_test==1`) AND (& is the logical **AND** operator in Python) have a positive prediction (`y_pred==1`).

2. Use this code to calculate the number of true positives:

```
TP = sum( (y_test==1) & (y_pred==1) )  
TP
```

0

Figure 2.30: Calculating the number of true positives

The true positive rate is the proportion of true positives to positives.

- Run the following code to obtain the true positive rate:

```
TPR = TP/P  
TPR
```

You will obtain the following output:

0.0

Figure 2.31: Calculating the true positive rate

Similarly, we can identify the false negatives.

- Calculate the number of false negatives with this code:

```
FN = sum( (y_test==1) & (y_pred==0) )  
FN
```

This should output the following:

1155

Figure 2.32: Calculating the number of false negatives

We'd also like the false negative rate.

- Calculate the false negative rate with this code:

```
FNR = FN/P  
FNR
```

This should output the following:

1.0

Figure 2.33: The false negative rate

What have we learned from the true positive and false negative rates?

First, we can confirm that they sum to 1. This fact is easy to see because the TPR = 0 and the FPR = 1. What does this tell us about our model? On the testing set, at least for the positive samples, the model has in fact acted as a majority class null model. Every positive sample was predicted to be negative, so none of them were correctly predicted.

6. Let's find the TNR and FPR of our testing data. Since these calculations are very similar to those we looked at previously, we show them all at once and illustrate a new Python function:

```
N = sum(y_test==0)
N
4178

TN = sum( (y_test==0) & (y_pred==0))
TN
4178

FP = sum( (y_test==0) & (y_pred==1))
FP
0

TNR = TN/N
FPR = FP/N
print('The true negative rate is {} and the false positive rate is {}'.format(TNR, FPR))

The true negative rate is 1.0 and the false positive rate is 0.0
```

Figure 2.34: Calculating true negative and false positive rates and printing them

In addition to calculating the TNR and FPR in a similar way that we had previously with the TPR and FNR, we demonstrate the `print` function in Python along with the `.format` method for strings, which allows substitution of variables in locations marked by curly braces `{}`. There are a range of options for formatting numbers, such as including a certain number of decimal places.

Note

For additional details, refer to <https://docs.python.org/3/tutorial/inputoutput.html>.

Now, what have we learned here? In fact, our model behaves exactly as the majority class null model for all samples, both positive and negative. It's clear we're going to need a better model.

While we have manually calculated all the entries of the confusion matrix in this exercise, in scikit-learn there is a quick way to do this. Note that in scikit-learn, the true class is along the vertical axis and the predicted class is along the horizontal axis of the confusion matrix, as we presented it earlier.

7. Create a confusion matrix in scikit-learn with this code:

```
metrics.confusion_matrix(y_test, y_pred)
```

You will obtain the following output:

```
metrics.confusion_matrix(y_test, y_pred)  
  
array([[4178,      0],  
       [1155,      0]])
```

Figure 2.35: The confusion matrix for our example model

All the information we need to calculate the TPR, FNR, TNR, and FPR is contained in the confusion matrix. We also note that there are many more classification metrics that can be derived from the confusion matrix. In fact, some of these are actually synonyms for ones we've already examined here. For example, the TPR is also called **recall** and **sensitivity**. Along with recall, another metric that is often used for binary classification is **precision**: this is the proportion of positive predictions that are correct (as opposed to the proportion of positive samples that are correctly predicted). We'll get more experience with precision in the activity for this chapter.

Note

Multiclass classification

Our case study involves a binary classification problem, with only two possible outcomes: the account does or does not default. Another important type of machine learning classification problem is multiclass classification. In multiclass classification, there are several possible mutually exclusive outcomes. A classic example is image recognition of handwritten digits; a handwritten digit should be only one of 0, 1, 2, ... 9. Although multiclass classification is outside the scope of this book, the metrics we are learning now for binary classification can be extended to the multiclass setting.

Discovering Predicted Probabilities: How Does Logistic Regression Make Predictions?

Now that we're familiar with accuracy, true and false positives and negatives, and the confusion matrix, it's time to learn a little bit more about logistic regression, in order to further our knowledge of binary classification metrics. So far, we've only considered logistic regression as a "black box" that can learn from labeled training data and then make binary predictions on new features. While we will learn about the workings of logistic regression in detail later in the book, we can begin to peek inside the black box now.

One thing to understand about how logistic regression works is that the raw predictions – in other words, the direct outputs from the mathematical equation that defines logistic regression – are not binary labels. They are actually **probabilities** on a scale from 0 to 1 (although, technically, the equation never allows for the probabilities to be exactly equal to 0 or 1, as we'll see later). These probabilities are only transformed into binary predictions through the use of a **threshold**. The threshold is the probability above which a prediction is declared to be positive, and below which it is negative. The threshold in scikit-learn is 0.5. This means any sample with a predicted probability of at least 0.5 is identified as positive, and any with a predicted probability < 0.5 is decided to be negative. However, we are not required to use the default threshold. In fact, choosing the threshold is one of the key flexibilities of logistic regression, as well as other machine learning classification algorithms that estimate probabilities of class membership.

Exercise 10: Obtaining Predicted Probabilities from a Trained Logistic Regression Model

In the following exercise, we will get familiar with the predicted probabilities of logistic regression and how to obtain them from a scikit-learn model.

We can begin to discover predicted probabilities by further examining the methods available to us on the logistic regression model object that we trained already in this chapter. Recall that before, once we trained the model, we could then make binary predictions using the values of features from new samples, by passing these values to the `.predict` method of the trained model. These are predictions made on the assumption of a threshold of 0.5.

However, we can directly access the predicted probabilities of these samples, using the `.predict_proba` method. Perform the following steps to complete the exercise:

Note

The code and the resulting output for this exercise have been loaded in a Jupyter Notebook that can be found here: <http://bit.ly/2UWPgYo>.

1. Obtain predicted probabilities for the testing samples using this code:

```
y_pred_proba = example_lr.predict_proba(X_test)  
y_pred_proba
```

The output should be as follows:

```
array([[0.77423402, 0.22576598],  
       [0.77423402, 0.22576598],  
       [0.78792915, 0.21207085],  
       ...,  
       [0.78792915, 0.21207085],  
       [0.78792915, 0.21207085],  
       [0.78792915, 0.21207085]])
```

Figure 2.36: Predicted probabilities of the testing data

We see in the output of this, which we've stored in `y_pred_proba`, that there are two columns. This is because there are two classes in our classification problem: negative and positive. Assuming the negative labels are coded as 0 and the positives as 1, as they are in our data, scikit-learn will report the probability of negative class membership as the first column, and positive class membership as the second.

From your discussion of probability, you should conclude that the sum of class probabilities over the two classes is equal to 1 for every sample. Let's confirm this.

First, we can use `np.sum` over the first dimension (columns) to calculate the sum of probabilities for each sample.

2. Calculate the sum of predicted probabilities for each sample with this code:

```
prob_sum = np.sum(y_pred_proba, 1)  
prob_sum
```

The output is as follows:

```
array([1., 1., 1., ..., 1., 1., 1.])
```

Figure 2.37: Calculating sums of predicted probabilities

It certainly looks like all 1s! We should check to see that the result is the same shape as the array of testing data labels.

3. Check the array shape with this code:

```
prob_sum.shape
```

This should output the following:

```
(533,)
```

Figure 2.38: The shape of the probability sum array

Good; this is the expected shape. Now, to check that each and every value is 1. We use `np.unique` to show all the unique elements of this array. This is similar to `DISTINCT` in SQL. If all the probability sums are indeed 1, there should only be 1 unique element of the probability array: 1.

4. Show all unique array elements with this code:

```
np.unique(prob_sum)
```

This should output the following:

```
array([1.])
```

Figure 2.39: The unique value of the probability sum array

After confirming our belief in the predicted probabilities, we note that since class probabilities sum to 1, it's sufficient to just consider the second column, the predicted probability of positive class membership. Let's capture these in an array.

5. Run this code to put the second column of the predicted probabilities array (predicted probability of membership in positive class) in an array:

```
pos_proba = y_pred_proba[:, 1]
pos_proba
```

The output should be as follows:

```
array([0.22576598, 0.22576598, 0.21207085, ..., 0.21207085, 0.21207085,
       0.21207085])
```

Figure 2.40: Predicted probabilities of positive class membership

What do these probabilities look like? One way to find out, and a good diagnostic for model output, is to plot the predicted probabilities. A histogram is a natural way to do this. We can directly use the matplotlib histogram function, `hist()`, to do this. Note that if you execute a cell with only the histogram function, you will get the output of the NumPy histogram function returned before the plot. This includes the number of samples in each bin, and the locations of the bin edges.

6. Execute this code to see histogram output and an unformatted plot (not shown):

```
plt.hist(pos_proba)
```

The output is as follows:

```
(array([1883.,      0.,      0., 2519.,      0.,      0.,   849.,      0.,
       0.,      0.,      82.]),
array([0.21207085, 0.21636321, 0.22065556, 0.22494792, 0.22924027,
       0.23353263, 0.23782498, 0.24211734, 0.24640969, 0.25070205,
       0.2549944 ]),
<a list of 10 Patch objects>)
```

Figure 2.41: Details of histogram calculation

This may be useful information for you, and could also be obtained directly from the `np.histogram()` function. However, here we're mainly interested in the plot, so we adjust the font size and add some axis labels.

7. Run this code for a formatted histogram plot of predicted probabilities:

```
mpl.rcParams['font.size'] = 12
plt.hist(pos_proba)
plt.xlabel('Predicted probability of positive class for testing data')
plt.ylabel('Number of samples')
```

The plot should look like this:

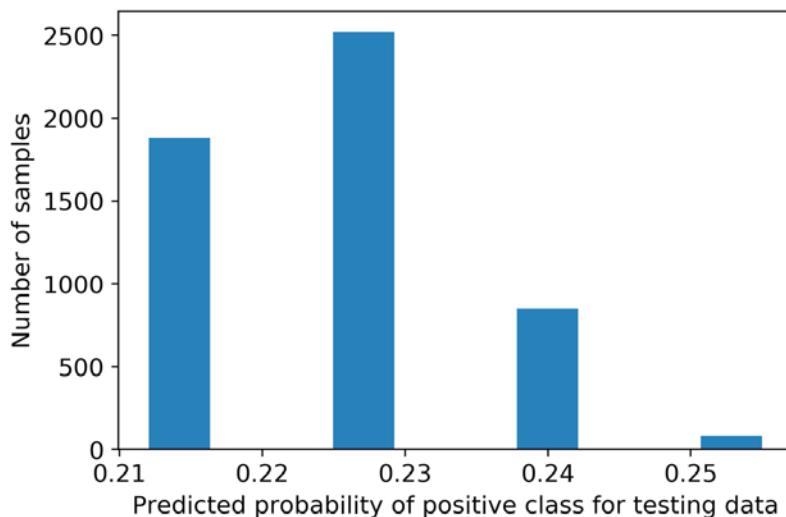


Figure 2.42: Histogram plot of predicted probabilities

Notice that in the histogram of probabilities, there are only four bins that actually have samples in them, and they are spaced fairly far apart. This is because there are only four unique values for the **EDUCATION** feature, which is the only feature in our example model.

Also, notice that all the predicted probabilities are below 0.5. This is the reason every sample was predicted to be negative, using the 0.5 threshold. Now we can imagine that if we set our threshold below 0.5, we would get different results.

For example, if we set the threshold at 0.25, all of the samples in the smallest bin to the far right of Figure 2.42 would be classified as positive, since the predicted probability for all of these is above 0.25. It would be informative for us if we could visually see how many of these samples actually had positive labels. Then we could see whether moving our threshold down to 0.25 would improve the performance of our classifier by classifying the samples in the far-right bin as positive.

In fact, we can visualize this easily, using a **stacked histogram**. This will look a lot like the histogram in Figure 2.42, except that the negative and positive samples will be colored differently. First, we need to distinguish between positive and negative samples in the predicted probabilities. We can do this by indexing our array of predicted probabilities with logical masks; first to get positive samples, where `y_test == 1`, and then to get negative samples, where `y_test == 0`.

8. Isolate the predicted probabilities for positive and negative samples with this code:

```
pos_sample_pos_proba = pos_proba[y_test==1]  
neg_sample_pos_proba = pos_proba[y_test==0]
```

Now we want to plot these as a stacked histogram. The code is similar to the histogram we already created, except that we will pass a list of arrays to be plotted, which are the arrays of probabilities for positive and negative samples we just created, and a keyword indicating we'd like the bars to be stacked, as opposed to plotted side by side. We'll also create a legend so that the colors are clearly identifiable on the plot.

9. Plot a stacked histogram using this code:

```
plt.hist([pos_sample_pos_proba, neg_sample_pos_proba],  
        histtype='barstacked')  
plt.legend(['Positive samples', 'Negative samples'])  
plt.xlabel('Predicted probability of positive class')  
plt.ylabel('Number of samples')
```

The plot should look like this:

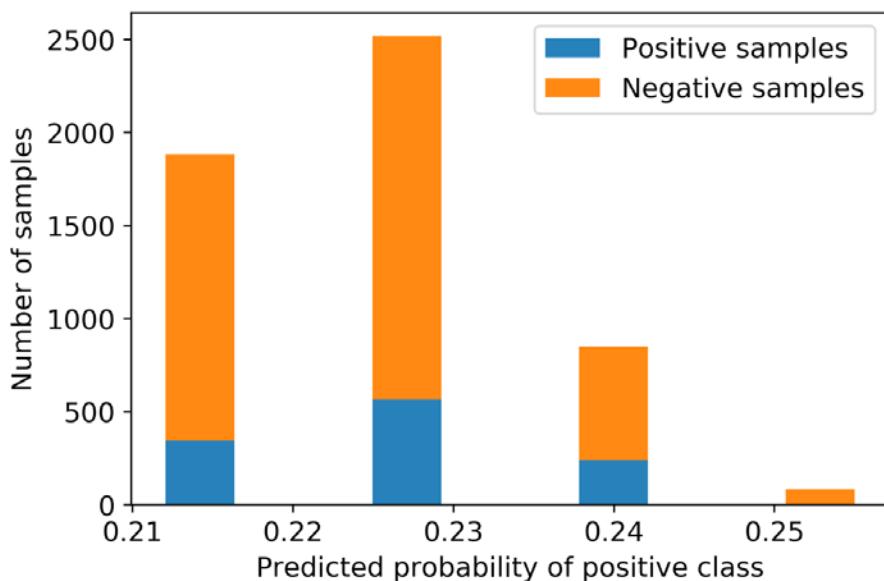


Figure 2.43: Stacked histogram of predicted probabilities by class

The plot shows us the true label of the samples for each predicted probability. Now we can consider what the effect would be of lowering the threshold to 0.25. Take a moment and think about what this would mean, keeping in mind that any sample with a predicted probability at or above the threshold would be classified as positive.

Since nearly all the samples in the small bin to the right on Figure 2.42 are negative samples, if we were to decrease the threshold to 0.25, we would erroneously classify these as positive samples and increase our false positive rate. At the same time, we still wouldn't have managed to classify many, if any, positive samples correctly, so our true positive rate wouldn't increase very much at all. Making this change would appear to decrease the accuracy of the model.

The Receiver Operating Characteristic (ROC) Curve

Deciding on a threshold for a classifier is a question of finding the "sweet spot" where we are successfully recovering enough true positives, without incurring too many false positives. As the threshold is lowered more and more, there will be more of both. A good classifier will be able to capture more true positives without the expense of a large number of false positives. What would be the effect of lowering the threshold even more, with the predicted probabilities from the previous exercise? It turns out there is a classic method of visualization in machine learning, with a corresponding metric, that can help answer these kinds of questions.

The **receiver operating characteristic (ROC)** curve is a plot of the pairs of true positive rates (y-axis) and false positive rates (x-axis) that result from lowering the threshold down from 1, all the way to 0. You can imagine that if the threshold is 1, there are no positive predictions since a logistic regression only predicts probabilities strictly between 0 and 1 (endpoints not included). Since there are no positive predictions, the TPR and the FPR are both 0, so the ROC curve starts out at (0, 0). As the threshold is lowered, the TPR will start to increase, hopefully faster than the FPR if it's a good classifier. Eventually, when the threshold is lowered all the way to 0, every sample is predicted to be positive, including all the ones that are, in fact, positive, but also all the ones that are actually negative. This means the TPR is 1 but the FPR is also 1. In between these two extremes are the reasonable options for where you may want to set the threshold, depending on the relative costs and benefits of true and false positives and negatives for the specific problem being considered. In this way, it is possible to get a complete picture of the performance of the classifier at all different thresholds to decide which one to use.

We could write the code to determine the TPRs and FPRs of the ROC curve by using the predicted probabilities and varying the threshold from 1 to 0. Instead, we will use scikit-learn's convenient functionality, which will take the true labels and predicted probabilities as inputs, and return arrays of TPRs, FPRs, and the thresholds that lead to them. We will then plot the TPRs against the FPRs to show the ROC curve. Run this code to use scikit-learn to generate the arrays of TPRs and FPRs for the ROC curve.

```
fpr, tpr, thresholds = metrics.roc_curve(y_test, pos_proba)
```

Now we need to produce a plot. We'll use `plt.plot`, which will make a line plot using the first argument as the *x* values (FPRs), the second argument as the *y* values (TPRs), and the shorthand '`*-`' to indicate a line plot with star symbols where the data points are located. We add a straight-line plot from the point (0, 0) to (1, 1), which will appear in red ('`r`') and as a dashed line ('`--`'). We've also given the plot a legend (which we'll explain shortly), as well as axis labels and a title. This code produces the ROC plot:

```
plt.plot(fpr, tpr, '*-')
plt.plot([0, 1], [0, 1], 'r--')
plt.legend(['Logistic regression', 'Random chance'])
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.title('ROC curve')
```

And the plot should look like this:

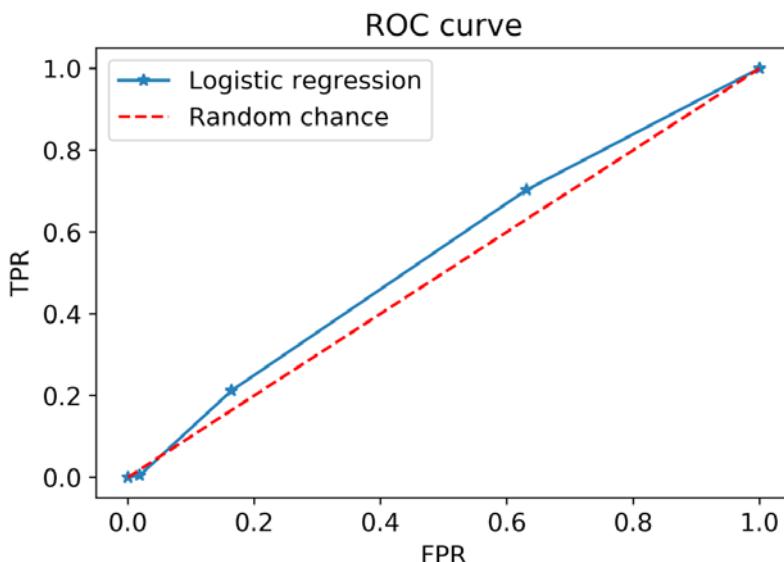


Figure 2.44: ROC curve for our logistic regression, with a line of random chance shown for comparison

What have we learned from our ROC curve? We can see that it starts at (0,0) with a threshold high enough so that there are no positive classifications. Then the first thing that happens, as we imagined previously when lowering the threshold to about 0.25, is that we get an increase in the false positive rate, but very little increase in the true positive rate. The effects of continuing to lower the threshold so that the other bars from our stacked histogram plot would be included as positive classifications are shown by the subsequent points on the line. We can see the thresholds that lead to these rates by examining the threshold array, which is not part of the plot. View the thresholds used to calculate the ROC curve using this code:

```
thresholds
```

The output should be as follows:

```
thresholds  
array([1.2549944 , 0.2549944 , 0.24007604, 0.22576598, 0.21207085])
```

Figure 2.45: Thresholds for the ROC curve

Notice that the first threshold is actually above 1; practically speaking, it just needs to be a threshold that's high enough that there are no positive classifications.

Now consider what a "good" ROC curve would look like. As we lower the threshold, we want to see the TPR increase, which means our classifier is doing a good job of correctly identifying positive samples. At the same time, ideally the FPR should not increase that much. The ROC curve of an effective classifier would "hug" the upper left corner of the plot: high true positive rate, low false positive rate. You can imagine that a perfect classifier would get a TPR of 1 (recovers all the positive samples) with an FPR of 0 and appear as a sort of square starting at (0,0), going up to (0,1), and finishing at (1,1). While in practice this kind of performance is highly unlikely, it gives us a limiting case.

Further consider what the area under the curve (AUC) of such a classifier would be, remembering integrals from calculus if you have taken it. The AUC of a perfect classifier would be 1, because the shape of the curve would be a square on the unit interval [0, 1].

On the other hand, the line labeled as "Random chance" in our plot is the ROC curve that theoretically results from flipping an unbiased coin as a classifier: it's just as likely to get a true positive as a false positive, so lowering the threshold introduces more of each in equal proportion and the TPR and FPR increase at the same rate. The AUC under this ROC would be half of the perfect classifier's, as you can see graphically, and would be 0.5.

So, in general, the ROC AUC is going to be between 0.5 and 1 (although values below 0.5 are technically possible). Values close to 0.5 indicate the model can do little better than random chance (coin flip) as a classifier, while values closer to 1 indicate better performance. The **ROC AUC** is a key metric for the quality of a classifier and is widely used in machine learning. The ROC AUC may also be referred to as the **C-statistic**.

As such an important metric, naturally scikit-learn has a convenient way to calculate it. Let's see what the ROC AUC of the logistic regression classifier is, where we can pass the same information that we did to the **roc_curve** function. Calculate the area under the ROC curve with this code:

```
metrics.roc_auc_score(y_test, pos_proba)
```

And observe the output:

0.5434650477972642

Figure 2.46: ROC AUC for the logistic regression

The ROC AUC for the logistic regression is pretty close to 0.5, meaning it's not a very effective classifier. This may not be surprising, considering we have expended no effort to determine which features out of the candidate pool are actually useful, at this point. We're just getting used to model fitting syntax and learning the way to calculate model quality metrics using a simple model containing only the **EDUCATION** feature. Later on, by considering other features, hopefully we'll get a higher ROC AUC.

Note

ROC curve: How did it get that name?

During World War II, radar receiver operators were evaluated on their ability to judge whether something that appeared on their radar screen was in fact an enemy aircraft or not. These decisions involved the same concepts of true and false positives and negatives that we are interested in for binary classification. The ROC curve was devised as a way to measure the effectiveness of operators of radar receiver equipment.

Precision

Before embarking on the activity, we consider the classification metric briefly introduced previously: **precision**. Like the ROC curve, this diagnostic is useful over a range of thresholds. Precision is defined as follows:

$$\text{precision} = \frac{TP}{TP + FP}$$

Figure 2.47: Precision equation

Consider the interpretation of this, in the sense of varying the threshold across the range of predicted probabilities, as we did for the ROC curve. At a high threshold, there will be relatively few samples predicted as positive. As we lower the threshold, more and more will be predicted as positive. Our hope is that as we do this, the number of true positives increases more quickly than the number of false positives, as we saw on the ROC curve. Precision looks at the ratio of the number of true positives to the sum of true and false positives. Think about the denominator here: what is the sum of true and false positives?

This sum is in fact the total number of positive predictions. So, precision measures the ratio of positive predictions that are correct, to all positive predictions. For this reason, it is also called **positive predictive value**. Precision is often used when the classes are very imbalanced and is more focused on getting good model performance in the sense of the correctness of positive predictions. It's easy to get a high true positive rate by predicting many samples as positive, but this also increases the number of false positives relative to the number of positive predictions. If there are very few positive samples, precision gives a more critical assessment of the quality of a classifier than the ROC AUC. As with the ROC curve, there is a convenient function in scikit-learn to calculate precision, together with recall (also known as the true positive rate), over a range of thresholds: `metrics.precision_recall_curve`.

Why might precision be a useful measure of classifier performance? Imagine that for every positive model prediction, you are going to take some expensive course of action, such as offering a valuable coupon to a customer to retain their loyalty. False positives would be customers to whom you offered a coupon but didn't need to, representing wasted money. You would want to be sure that you were making the right decisions in whom to offer coupons to. Precision would be a good metric to use in this situation.

Activity 2: Performing Logistic Regression with a New Feature and Creating a Precision-Recall Curve

In this activity, you'll train a logistic regression using a feature besides **EDUCATION**. Then you will graphically assess the tradeoff between precision and recall, as well as calculate the area underneath a precision-recall curve. You will also calculate the ROC AUC on both the training and testing sets and compare them.

Perform the following steps to complete the activity:

Note

The code and the resulting output for this activity have been loaded in a Jupyter Notebook that can be found here: <http://bit.ly/2UWPgYo>.

1. Use scikit-learn's **train_test_split** to make a new set of training and testing data. This time, instead of **EDUCATION**, use **LIMIT_BAL**: the account's credit limit.
2. Train a logistic regression model using the training data from your split.
3. Create the array of predicted probabilities for the testing data.
4. Calculate the ROC AUC using the predicted probabilities and the true labels of the testing data. Compare this to the ROC AUC from using the **EDUCATION** feature.
5. Plot the ROC curve.
6. Calculate the data for the **precision-recall curve** on the testing data using scikit-learn functionality.
7. Plot the precision-recall curve using matplotlib.
8. Use scikit-learn to calculate the area under the precision-recall curve
9. Now recalculate the ROC AUC, except this time do it for the training data. How is this different, conceptually and quantitatively, from your earlier calculation?

Note

The solution to this activity can be found on page 331.

Summary

In this chapter, we finished the initial exploration of the case study data by examining the response variable. Once we became confident in the completeness and correctness of the data set, we became prepared to explore the relation between features and response and build models.

We spent much of this chapter getting used to model fitting in scikit-learn at a technical, coding level, and learning about metrics we could use with the binary classification problem of the case study. When trying different feature sets and different kinds of models, you will need some way to tell if one approach is working better than another. Consequently, you'll need to use model performance metrics.

While accuracy is a familiar and intuitive metric, we learned why it may not give a useful assessment of the performance of a classifier. We learned how to use a majority class null model to tell whether an accuracy rate is truly good, or no better than what would result from prediction of simply the most common class for all samples. When the data are imbalanced, accuracy is usually not the best way to judge a classifier.

In order to have a more nuanced view of how a model is performing, it's necessary to separate out the positive and negative classes and assess the accuracy on these independently. From the resulting counts of true and false positive and negative classifications, which can be summarized in a confusion matrix, we can derive several other metrics: true and false positive and negative rates. Combining true and false positives and negatives with the concept of predicted probabilities and a variable threshold of prediction, we can further characterize the usefulness of a classifier using the ROC curve, the precision-recall curve, and the areas under these curves.

With these tools, you are well equipped to answer general questions about the performance of a binary classifier, in any domain you may be working in. Later in the book, we will learn about application-specific ways to assess model performance, by attaching costs and benefits to true and false positives and negatives. Before that, starting in the next chapter, we will begin learning the details behind what is possibly the most popular and simplest classification model: **logistic regression**.

3

Details of Logistic Regression and Feature Exploration

Learning Objectives

By the end of this chapter, you will be able to:

- Write list comprehensions in Python
- Describe the workings of logistic regression
- Formulate the sigmoid and logit versions of logistic regression
- Utilize univariate feature selection to find important features
- Customize plots with the Matplotlib API
- Characterize the linear decision boundary of a logistic regression

This chapter presents the basics of logistic regression along with various other methods for examining the relationship between features and a response variable.

Introduction

In the previous chapter, we concluded our examination of the response variable, and developed a few example machine learning models using scikit-learn. However, the features we used, **EDUCATION** and **LIMIT_BAL**, were not chosen in a systematic way.

In this chapter, we will start to develop techniques that can be used to assess features one by one. This will enable making a quick pass over all the features to see which ones could be expected to be useful for predictive modeling. For the most promising features, we will see how to create visual summaries that serve as useful communication tools.

Next, we will begin our detailed examination of logistic regression. We'll learn why logistic regression is considered to be a linear model, even if the formulation involves some non-linear functions. As a key consequence of this linearity, we will see why the decision boundary of logistic regression could make it difficult to accurately classify data. Along the way, we'll learn how to write functions in Python.

Examining the Relationships between Features and the Response

In order to make accurate predictions of the response variable, good features are necessary. We need features that are clearly linked to the response variable in some way. Thus far, we've examined the relationship between a couple features and the response variable, either by calculating a **groupby/mean** of the response variable, or by trying models directly, which is another way to make this kind of exploration. However, we have not yet made a systematic exploration of how all the features relate to the response variable. We will do that now and capitalize on all the hard work we put in when we were exploring the features and making sure the data quality was good.

A popular way of getting a quick look at how all the features relate to the response variable, as well as how the features are related to each other, is by using a **correlation plot**. We will first create the correlation plot for the case study data, then discuss how to interpret it, along with some mathematical details.

In order to create a correlation plot, the necessary inputs include all features that we plan to explore, as well as the response variable. Since we are going to use most of the column names from the DataFrame for this, a quick way to get the appropriate list in Python is to start with all the column names and remove those that we don't want from the list. As a preliminary step, we start a new notebook for this chapter and load packages and the cleaned data from *Chapter 1, Data Exploration and Cleaning*, with this code:

```
import numpy as np #numerical computation
import pandas as pd #data wrangling
import matplotlib.pyplot as plt #plotting package
#Next line helps with rendering plots
%matplotlib inline
import matplotlib as mpl #add'l plotting functionality
import seaborn as sns #a fancy plotting package
mpl.rcParams['figure.dpi'] = 400 #high res figures
df = pd.read_csv('../Data/Chapter_1_cleaned_data.csv')
```

Note

The path to your file of cleaned data may be different, depending on where you saved it in *Chapter 1, Data Exploration and Cleaning*.

Notice that this notebook starts out in a very similar way to the previous chapter's notebook, except we also import the **Seaborn** package, which has many convenient plotting features that build on **Matplotlib**. Now let's make a list of all the columns of the DataFrame and look at the first and last five:

```
features_response = df.columns.tolist()
features_response[:5]
['ID', 'LIMIT_BAL', 'SEX', 'EDUCATION', 'MARRIAGE']
features_response[-5:]
['graduate school', 'high school', 'none', 'others', 'university']
```

Figure 3.1: Get a list of column names

Recall that we are not to use the gender variable due to ethical concerns, and we learned that **PAY_2**, **PAY_3**, ..., **PAY_6** are incorrect and should be ignored. Also, we are not going to examine the one-hot encoding we created from the **EDUCATION** variable, since the information from those columns is already included in the original feature, at least in some form. We will just use the **EDUCATION** feature directly. Finally, it makes no sense to use **ID** as a feature, since this is simply a unique account identifier and has nothing to do with the response variable. Let's make another list of column names that are neither features nor the response. We want to exclude these from our analysis:

```
items_to_remove = ['ID', 'SEX', 'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6',
                   'EDUCATION_CAT', 'graduate school', 'high school',
                   'none',
                   'others', 'university']
```

To have a list of column names that consists only of the features and response we will use, we want to remove the names in **items_to_remove** from the current list contained in **features_response**. There are several ways to do this in Python. We will use this opportunity to learn about a particular way of building a list in Python, called a **list comprehension**. When people talk about certain constructions as being **Pythonic**, or idiomatic to the Python language, list comprehensions are often one of the things that are mentioned.

What is a list comprehension? Conceptually, it is basically the same as a **for** loop. However, list comprehensions enable the creation of lists, which may be spread across several lines in an actual **for** loop, to be written in one line. They are also slightly faster than **for** loops, due to optimizations within Python. While this likely won't save us much time here, this is a good chance to become familiar with them. Here is an example list comprehension:

```
example_list_comp = [item for item in range(5)]
example_list_comp

[0, 1, 2, 3, 4]
```

Figure 3.2: Example of a list comprehension

That's all there is to it!

We can also use additional clauses to make the list comprehensions flexible. For example, we can use them to reassign the `features_response` variable with a list containing everything that's not in the list of strings we wish to remove:

```
features_response = [item for item in features_response if item not in items_to_remove]
features_response

['LIMIT_BAL',
'EDUCATION',
'MARRIAGE',
'AGE',
'PAY_1',
'BILL_AMT1',
'BILL_AMT2',
'BILL_AMT3',
'BILL_AMT4',
'BILL_AMT5',
'BILL_AMT6',
'PAY_AMT1',
'PAY_AMT2',
'PAY_AMT3',
'PAY_AMT4',
'PAY_AMT5',
'PAY_AMT6',
'default payment next month']
```

Figure 3.3: Using a list comprehension to prune down the column names

The use of `if` and `not in` within the list comprehension is fairly self-explanatory. Easy readability in structures like list comprehensions is one of the reasons for the popularity of Python.

Note

The Python documentation (<https://docs.python.org/3/tutorial/datastructures.html>) defines list comprehensions as the following:

"A list comprehension consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses."

Thus, list comprehensions can enable you to do things with less code, in a way that is usually pretty readable and understandable.

Pearson Correlation

Now we are ready to create our correlation plot. Underlying a correlation plot is a **correlation matrix**, which we must calculate first. pandas makes this easy. We just need to select our columns of features and response values using the list we just created and call the `.corr()` method on these columns. As we calculate this, note that the type of correlation available to us in pandas is **linear correlation**, also known as **Pearson correlation**. Pearson correlation is used to measure the strength and direction (that is, positive or negative) of the linear relationship between two variables:

```
corr = df[features_response].corr()
corr.iloc[0:5,0:5]
```

	LIMIT_BAL	EDUCATION	MARRIAGE	AGE	PAY_1
LIMIT_BAL	1.000000	-0.232688	-0.111873	0.149157	-0.273396
EDUCATION	-0.232688	1.000000	-0.137097	0.179035	0.112653
MARRIAGE	-0.111873	-0.137097	1.000000	-0.412828	0.019759
AGE	0.149157	0.179035	-0.412828	1.000000	-0.044277
PAY_1	-0.273396	0.112653	0.019759	-0.044277	1.000000

Figure 3.4: First five rows and columns of the correlation matrix

After creating the correlation matrix, you should notice that the row and column names are the same. Then, for each possible comparison between all pairs of features, as well as all features and the response, which we can't see yet here in the first five rows and columns, there is a number. This number is called the **correlation** between these two columns. You should notice that all the correlations are between -1 and 1; a column has a correlation of 1 with itself (the diagonal of the correlation matrix), and there is repetition: each comparison appears twice since each column name from the original DataFrame appears as both a row and column in the correlation matrix. Before saying more about correlation, we'll use Seaborn to make a nice plot of it. Here is the plotting code, followed by the output:

```
sns.heatmap(corr,
            xticklabels=corr.columns.values,
            yticklabels=corr.columns.values,
            center=0)
```

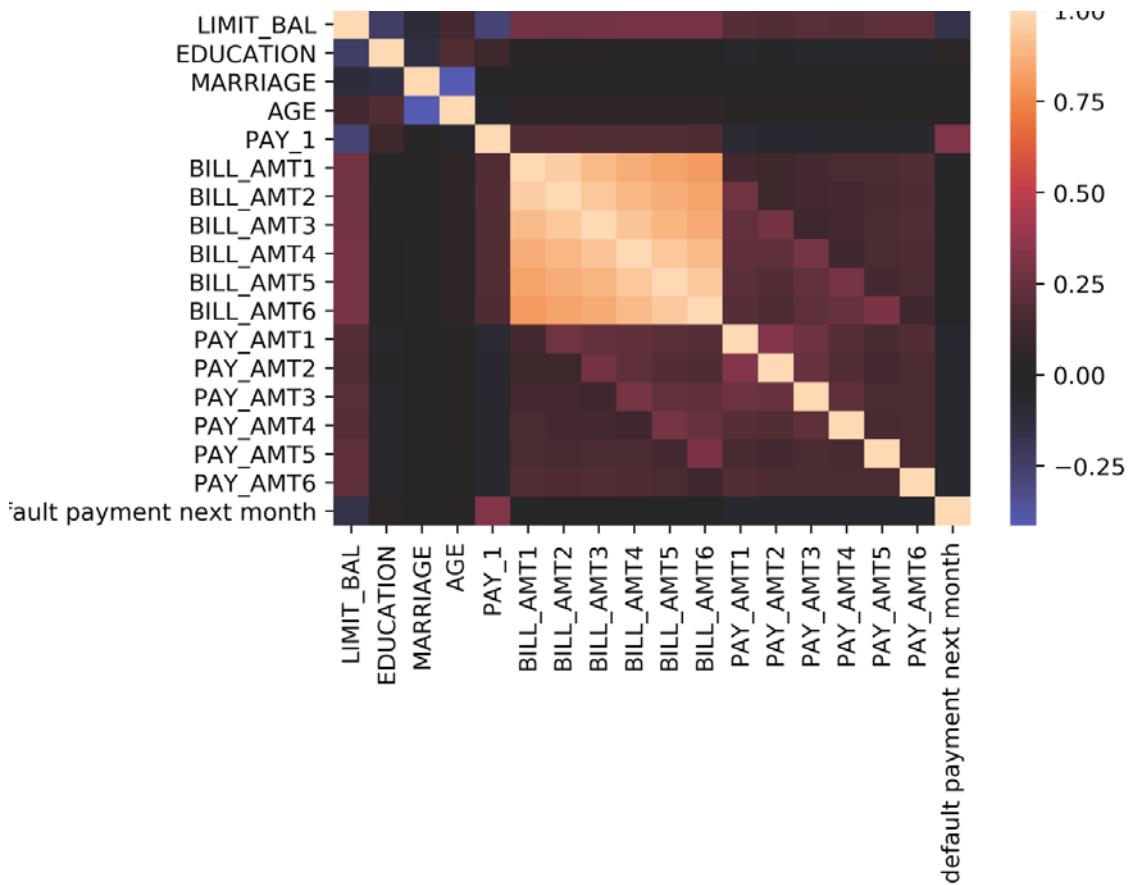


Figure 3.5: Heatmap of the correlation plot in Seaborn

The Seaborn **heatmap** makes an obvious visualization of the correlation matrix, according to the color scale on the right of Figure 3.5, which is called a **colorbar**. Notice that when calling `sns.heatmap`, in addition to the matrix, we supplied the **tick labels** for the x and y axes, which are the features and response names, and indicated that the center of **colorbar** should be 0, so that positive and negative correlation are distinguishable as red and blue, respectively.

What does this plot tell us? At a high level, if two features, or a feature and the response, are **highly correlated** with each other, you can say there is a strong association between them. Features that are highly correlated to the response will be good features to use for prediction. This high correlation could be positive or negative; we'll explain the difference shortly.

The correlation plot shows us, along the bottom row (or last column), that the **PAY_1** feature is probably the most strongly correlated feature to the response variable. We can also see that a number of features are highly correlated to each other, in particular the **BILL_AMT** features. We will talk in the next chapter about the importance of features that are correlated with each other; this is important to know about for certain models such as logistic regression, that make assumptions about the correlations between features. For now, we take the observation that **PAY_1** is likely going to be the best, most predictive feature for our model. The other feature that looks like it may be important is **LIMIT_BAL**, which is negatively correlated. Depending on how astute your vision is, only these two really appear to be any color other than black (meaning 0 correlation), in the bottom row of Figure 3.5.

What is linear correlation, mathematically speaking? If you've taken basic statistics, you are likely familiar with linear correlation already. For two columns, X and Y, linear correlation ρ (lowercase Greek letter "rho") is defined as the following:

$$\rho = \frac{E[(X - \mu_X)(Y - \mu_Y)]}{\sigma_X \sigma_Y}$$

Figure 3.6: Linear correlation equation

This equation describes the **expected value** (E , which you can think of as the average) of the difference between the elements of X, and their average, μ_x , multiplied by the difference between the corresponding elements of Y, and their average, μ_y . The average is taken over pairs of X, Y values. You can imagine that if when X is relatively large (compared to its mean), Y also tends to be large, and similarly Y tends to be small when X is small, then the terms of the multiplication in the numerator will tend to either be both positive or both negative, leading to a positive product and **positive correlation** after the expected value is taken. Conversely, if Y tends to decrease as X increases, they will have **negative correlation**. The denominator (product of **standard deviations** of X and Y) serves to normalize linear correlation to the scale of $[-1, 1]$. Because Pearson correlation is adjusted for the mean and standard deviation of the data, the actual values of the data are not so important as the relationship between X and Y. Stronger linear correlations are closer to 1 or -1. If there is no linear relation between X and Y, the correlation will be close to zero.

It's worth noting that, while it is regularly used in this context by data science practitioners, Pearson correlation is not strictly appropriate for a binary response variable, as we have in our problem. Technically speaking, among other restrictions, Pearson correlation is only valid for **continuous data**, such as the data we used for our linear regression exercise in *Chapter 2, Introduction to Scikit-Learn and model evaluation*. However, Pearson correlation can still accomplish the purpose of giving a quick idea of the potential usefulness of features. It is also conveniently available in software libraries such as pandas.

In data science in general, you will find that certain widely used techniques may be applied to data that violates their formal statistical assumptions. It is important to be aware of the formal assumptions underlying methods. In fact, knowledge of these assumptions may be tested during interviews for data science jobs. However, in practice, as long as a technique can help us on our way to understanding the problem and finding an effective solution, it can still be a valuable tool.

That being said, linear correlation will not be an effective measure of the predictive power of all features. In particular, it only picks up on linear relationships. Shifting our focus momentarily to a hypothetical regression problem, have a look at the following examples and discuss what you expect the linear correlations to be. Notice that the values of the data on the x - and y -axes are not labeled; this is because the location (mean) and standard deviation (scale) of data does not affect Pearson correlation, only the relationship between the variables, which can be discerned by plotting them together.

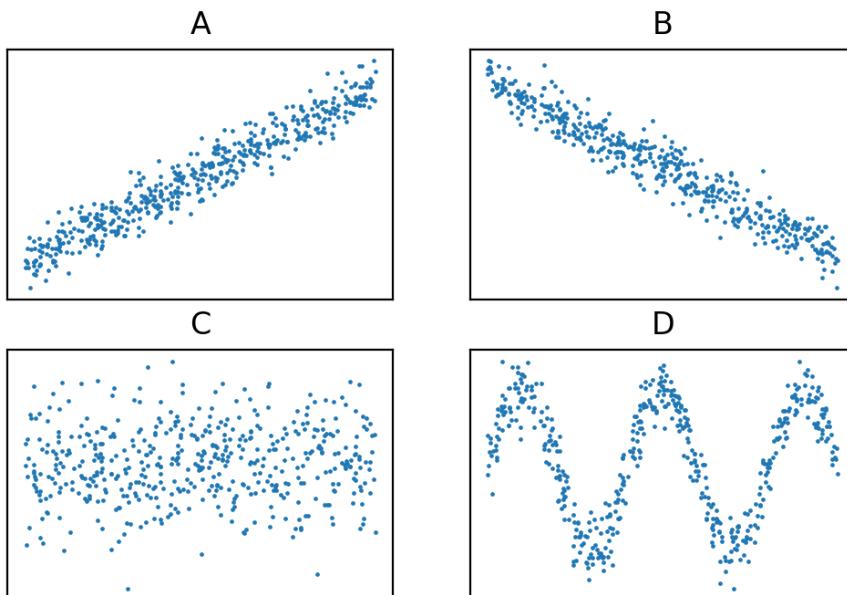


Figure 3.7: Scatterplots of the relation between example variables

For examples A and B, the actual Pearson correlations of these datasets are 0.96 and -0.97, respectively, according to the formula given previously. From looking at the plots, it's pretty clear that a correlation close to 1 or -1 has provided useful insight on the relationship between these variables. For example C, the correlation is 0.06. Correlation closer to 0 looks like an effective indication of the lack of an association here: the value of Y doesn't really seem to have much to do with the value of X. However, in example D, there is clearly some relationship between the variables. But the linear correlation is actually lower than the previous example, at 0.02! Here, X and Y tend to "move together" over smaller scales, but this is averaged out over all samples when linear correlation is calculated.

Ultimately, any summary statistic such as correlation that you may choose is only that: a summary. It could hide important details. For this reason, it is usually a good idea to visually examine the relationship between the features and response. This potentially takes up a lot of space on the page, so we won't demonstrate it here for all features in the case study. However, both pandas and Seaborn offer functions to create what's called a **scatterplot matrix**. A scatterplot matrix is similar to a correlation plot, but it actually shows all the data as a grid of scatter plots of all features and the response variable. This allows you to examine the data directly in a concise format. Since this could potentially be a lot of data and plots, you may need to downsample your data and look at a reduced number of features for the function to run efficiently.

F-test

While Pearson correlation is theoretically valid for continuous response variables, the binary response variable for the case study data could be considered categorical data, with only two categories: 0 and 1. Among the different kinds of tests we can run, to see whether features are associated with a categorical response, is the **ANOVA F-test**, available in scikit-learn as `f_classif`. ANOVA stands for "analysis of variance". The ANOVA F-test can be contrasted with the **regression F-test**, which is very similar to Pearson correlation, also available in scikit-learn as `f_regression`.

We will do an ANOVA F-test using the candidate features for the case study data in the following exercise. You will see that the output consists of F-statistics, as well as p-values. How can we interpret this output? We will focus on the p-value, for reasons that will become clear in the exercise. The p-value is a useful concept across a wide variety of statistical measures. For instance, although we didn't examine them, each of the Pearson correlations calculated for the preceding correlation matrix has a corresponding p-value. There is a similar concept of p-value corresponding to linear regression coefficients, logistic regression coefficients, and other measures.

In the context of the F-test, the p-value answers the question: "For the samples in the positive class, how likely is it that the average value of this feature is the same as that of samples in the negative class?" If a feature has very different average values between the positive and negative classes, it will:

- Be very unlikely that those average values are the same (low p-value)
- Probably be a good feature in our model because it will help us discriminate between positive and negative classes

Keep these points in mind during the following exercise.

Exercise 11: F-test and Univariate Feature Selection

In this exercise, we'll use the F-test to examine the relation between features and response. We will examine this method as part of what is called **univariate feature selection**: the practice of testing features one by one against the response variable, to see which ones have predictive power. Perform the following steps to complete the exercise:

Note

For Exercises 11–15 and Activity 3, the code and the resulting output have been loaded in a Jupyter notebook that can be found here: <http://bit.ly/2Dz5INA>. You can scroll to the appropriate section within the Jupyter notebook to locate the exercise or activity of choice.

1. Our first step in doing the ANOVA F-test is to separate out the features and response as NumPy arrays, taking advantage of the list we created, as well as integer indexing in pandas:

```
X = df[features_response].iloc[:, :-1].values  
y = df[features_response].iloc[:, -1].values  
print(X.shape, y.shape)
```

The output should show the shapes of the features and response:

```
x = df[features_response].iloc[:, :-1].values  
y = df[features_response].iloc[:, -1].values  
print(x.shape, y.shape)
```

```
(26664, 17) (26664,)
```

```
from sklearn.feature_selection import f_classif
```

```
[f_stat, f_p_value] = f_classif(X, y)
```

Figure 3.8: Shape of feature and response arrays

There are 17 features, and both the features and response arrays have the same number of samples as expected.

2. Import the **f_classif** function and feed in the features and response:

```
from sklearn.feature_selection import f_classif  
[f_stat, f_p_value] = f_classif(X, y)
```

There are two outputs from **f_classif**: the **F-statistic** and the **p-value**, for the comparison of each feature to the response variable. Let's create a new DataFrame containing the feature names and these outputs, to facilitate our inspection. One way to specify a new DataFrame is by using a **dictionary**, with **key:value** pairs of column names and the data to be contained in each column. We show the DataFrame sorted (ascending) on p-value.

3. Use this code to create a **DataFrame** of feature names, F-statistics, and p-values, and show it sorted on p-value:

```
f_test_df = pd.DataFrame({'Feature':features_response[:-1],  
                           'F statistic':f_stat,  
                           'p value':f_p_value})  
f_test_df.sort_values('p value')
```

The output should look like this:

```
f_test_df = pd.DataFrame({'Feature':features_response[:-1],
                           'F statistic':f_stat,
                           'p value':f_p_value})
f_test_df.sort_values('p value')
```

	Feature	F statistic	p value
4	PAY_1	3156.672300	0.000000e+00
0	LIMIT_BAL	651.324071	5.838366e-142
11	PAY_AMT1	140.612679	2.358354e-32
12	PAY_AMT2	101.408321	8.256124e-24
13	PAY_AMT3	90.023873	2.542641e-21
15	PAY_AMT5	85.843295	2.090120e-20
16	PAY_AMT6	80.420784	3.219565e-19
14	PAY_AMT4	79.640021	4.774112e-19
1	EDUCATION	32.637768	1.122175e-08
2	MARRIAGE	18.078027	2.127555e-05
5	BILL_AMT1	11.218406	8.110226e-04
7	BILL_AMT3	5.722938	1.675157e-02
6	BILL_AMT2	5.668454	1.727965e-02
3	AGE	5.479140	1.925206e-02
8	BILL_AMT4	3.434740	6.384965e-02
9	BILL_AMT5	1.216082	2.701409e-01
10	BILL_AMT6	1.049561	3.056176e-01

Figure 3.9: Results of the ANOVA F-test

Note that for every decrease in p-value, there is an increase in the F-statistic, so the information in these columns is essentially the same in terms of ranking features.

The conclusions we can draw from the DataFrame of F-statistics and p-values are similar to what we observed in the correlation plot: `PAY_1` and `LIMIT_BAL` appear to be the most useful features. They have the smallest p-values, indicating the average values of these features between the positive and negative classes are **significantly different**, and these features will help predict which class a sample belongs to.

In scikit-learn, one of the uses of measures such as the F-test is to perform **univariate feature selection**. This may be helpful if you have a very large number of features, many of which may be totally useless, and would like a quick way to get a "short list" of which ones might be most useful. For example, if we wanted to retrieve only the 20% of features with the highest F-statistics, we can do this easily with the `SelectPercentile` class. Also note there is a similar class for selecting the top "k" features (where k is any number you specify), called `SelectKBest`. Here we demonstrate how to select the top 20%.

4. To select the top 20% of features according to the F-test, first import the `SelectPercentile` class:

```
from sklearn.feature_selection import SelectPercentile
```

5. Instantiate an object of this class, indicating we'd like to use the same feature selection criteria, ANOVA F-test, that we've already been considering in this exercise, and that we'd like to select the top 20% of features.

```
selector = SelectPercentile(f_classif, percentile=20)
```

6. Use the `.fit` method to fit the object on our features and response data, similar to how a model would be fit:

```
selector.fit(X, y)
```

The output should appear like this:

```
from sklearn.feature_selection import SelectPercentile

selector = SelectPercentile(f_classif, percentile=20)

selector.fit(X, y)

SelectPercentile(percentile=20,
                 score_func=<function f_classif at 0x1a218ba730>)
```

Figure 3.10: Univariate feature selection in scikit-learn for the top 20% of features

There are several ways to access the selected features directly, which you may learn about in the scikit-learn documentation (that is, the `.transform` method, or in the same step as fitting with `.fit_transform`). However, these methods will return NumPy arrays, which don't tell you the names of the features that were selected, just the values. For that, you can use the `.get_support` method of the feature selector object, which will give you the column indices of the feature array that were selected.

- Capture the indices of the selected features in an array named `best_feature_ix`:

```
best_feature_ix = selector.get_support()
best_feature_ix
```

The output should appear as follows, indicating a logical index that can be used with an array of feature names, as well as values, assuming they're in the same order as the features array supplied to `SelectPercentile`:

```
best_feature_ix = selector.get_support()
best_feature_ix
array([ True, False, False, False,  True, False, False, False,
       False, False,  True,  True, False, False, False])

features = features_response[:-1]

best_features = [features[counter] for counter in range(len(features))
                 if best_feature_ix[counter]]

best_features
['LIMIT_BAL', 'PAY_1', 'PAY_AMT1', 'PAY_AMT2']
```

Figure 3.11: Logical index of selected features

- The feature names can be obtained using all but the last element (the response variable name) of our `features_response` list by indexing with `:-1`:

```
features = features_response[:-1]
```

- Use the index array we created in Step 7 with a list comprehension and the `features` list, to find the selected feature names, as follows:

```
best_features = [features[counter] for counter in range(len(features))
                  if best_feature_ix[counter]]
best_features
```

The output should be as follows:

```
best_feature_ix = selector.get_support()
best_feature_ix
array([ True, False, False, False,  True, False, False, False, False,
       False, False,  True,  True, False, False, False])

features = features_response[:-1]

best_features = [features[counter] for counter in range(len(features))
                 if best_feature_ix[counter]]

best_features
['LIMIT_BAL', 'PAY_1', 'PAY_AMT1', 'PAY_AMT2']
```

Figure 3.12: Examining the labels of the top 20% of features

In this code, the list comprehension has looped through the number of elements in the `features` array (`len(features)`) with loop increment `counter`, using the Boolean array `best_feature_ix`, representing selected features, in the `if` statement to test whether each feature was selected and capturing the name if so.

The selected features agree with the top four rows of our DataFrame of F-test results, so the feature selection has worked as expected. While it's not strictly necessary to do things both ways, since they both lead to the same result, it's good to check your work, especially as you are learning new concepts. You should be aware that with convenience methods such as `SelectPercentile`, you don't get visibility into the F-statistics or p-values. However, in some situations it may be more convenient to use these methods, as the p-values may not necessarily be important, outside of their utility in ranking features.

Finer Points of the F-test: Equivalence to t-test for Two Classes and Cautions

When we use an F-test to look at the difference in means between just two groups, as we've done here for the binary classification problem of the case study, the test we are performing actually reduces to what's called a **t-test**. An F-test is extensible to three or more groups and so is useful for multiclass classification. A t-test just compares the means between two groups of samples, to see if the difference in those means is **statistically significant**.

While the F-test served our purposes here of univariate feature selection, there are a few cautions to keep in mind. Going back to the concept of formal statistical assumptions, for the F-test these include that the data is **normally distributed**. We have not checked this. Also, in comparing the same response variable, y , to many potential features from the matrix, X , we have performed what are known in statistics as **multiple comparisons**. In short, this means that by examining multiple features in comparison to the same response over and over, the odds increase that we'll find what we think is a "good feature" just by random chance. However, such features may not generalize to new data. There are statistical **corrections for multiple comparisons** that amount to adjusting the p-values to account for this.

Even if we have not followed all the statistical "rules" that go along with these methods, we can still get useful results from them. Many methods that assume a normal distribution are regularly used with non-normal data, often with acceptable results. And the multiple comparison correction is more of a concern when p-values are the ultimate quantity of interest, for example, when making statistical inferences. Here, p-values are just a means to an end of ranking the feature list. The order of this ranking would not change, if the p-values were corrected for multiple comparisons.

In addition to knowing which features are likely to be useful for modeling, it is good to have a deeper understanding of the important features. Consequently, we will make a detailed graphical exploration of these in the next exercise. We will also look at other methods for feature selection later, that don't make the same assumptions as those we've introduced here and are more directly integrated with the predictive models that we will build.

Hypotheses and Next Steps

According to our univariate feature exploration, the feature with the strongest association with the response variable is **PAY_1**. Does this make sense? What is the interpretation of **PAY_1**? **PAY_1** is the payment status of the account, in the most recent month. As we learned in initial data exploration, there are some values that indicate that the account was in good standing: -2 means no account usage, -1 means balance paid in full, and 0 means at least the minimum payment was made. On the other hand, positive integer values indicate a delay of payment by that many months. Accounts with delayed payments last month, were accounts that could be considered in default. This means that, essentially, this feature captures historical values of the response variable. Features such as this are extremely important as *one of the best predictors for just about any machine learning problem is historical data on the same thing you are trying to predict (that is, the response variable)*. This should make sense: people who have defaulted before are probably at the highest risk of defaulting again.

How about **LIMIT_BAL**, the credit limit of accounts? Thinking about how credit limits are assigned, it is likely that our client has assessed how risky a borrower is when deciding their credit limit. Riskier clients should be given lower limits, so the creditor is less exposed. Therefore, we may expect to see a higher probability of default, for accounts with lower values of **LIMIT_BAL**.

What have we learned from our univariate feature selection exercise? We have an idea of what the most important features in our model are likely to be. And, from the correlation matrix, we have some idea of how they are related to the response variable. However, knowing the limitations of the tests we used, it is a good idea to visualize these features for a closer look at the relation between features and response. We have also started to develop **hypotheses** about these features: why do we think they are important? Now, by visualizing the relationships between the features and the response variable, we can determine whether our ideas are compatible with what we can see in the data.

Such hypotheses and visualizations are often a key part of presenting your results to a client, who may be interested in how a model works, not just the fact that it does work.

Exercise 12: Visualizing the Relationship between Features and Response

In this exercise, you will further your knowledge of plotting functions from Matplotlib that you used earlier in this book. You'll learn how to customize graphics to better answer specific questions with the data. As you pursue these analyses, you will create insightful visualizations of how the **PAY_1** and **LIMIT_BAL** features relate to the response variable, that may possibly provide support for the hypotheses you formed about these features. This will be done by becoming more familiar with the Matplotlib **API**. Perform the following steps to complete the exercise:

Note

The code and the resulting output for this exercise have been loaded in a Jupyter notebook that can be found here: <http://bit.ly/2Dz5iNA>.

1. Calculate a baseline for the response variable of the default rate across the whole dataset using pandas' `.mean()`:

```
overall_default_rate = df['default payment next month'].mean()  
overall_default_rate
```

The output of this should be:

0.2217971797179718

Figure 3.13: Default rate over the whole dataset

What would be a good way to visualize default rates for different values of the PAY_1 feature?

Recall our observation that this feature is sort of like a hybrid categorial and numerical feature. We'll choose to plot it in a way that is typical for categorical features, due to the relatively small number of unique values. In *Chapter 1, Data Exploration and Cleaning* we did `value_counts` of this feature as part of data exploration, then later we learned about `groupby/mean` when looking at the **EDUCATION** feature. `groupby/mean` would be a good way to visualize the default rate again here, for different payment statuses.

2. Use this code to create a `groupby/mean` aggregation:

```
group_by_pay_mean_y = df.groupby('PAY_1').agg({'default payment next month':np.mean})
group_by_pay_mean_y
```

The output should look as follows:

```
group_by_pay_mean_y = df.groupby('PAY_1').agg({'default payment next month':np.mean})
group_by_pay_mean_y
```

default payment next month	
PAY_1	
-2	0.131664
-1	0.170002
0	0.128295
1	0.336400
2	0.694701
3	0.773973
4	0.682540
5	0.434783
6	0.545455
7	0.777778
8	0.588235

Figure 3.14: Mean of the response variable by groups of the PAY_1 feature

Looking at these values, you may already be able to discern the trend. Let's go straight to plotting them. We'll take it step by step and introduce some new concepts. You should put all the code from steps 3 through 6 in a single code cell.

In Matplotlib, every plot exists on an **axes**, and within a **figure** window. By creating objects for **axes** and **figures**, you can directly access and change their properties, including axis labels, tick marks, and other things on the axes, or the dimensions of the figure.

3. Create an **axes** object in a variable also called **axes**, using the following code:

```
axes = plt.axes()
```

4. Plot the overall default rate as a red horizontal line.

Matplotlib makes this easy; you just have to indicate the y-intercept of this line with the **axhline** function. Notice that instead of calling this function from **plt**, now we are calling it as a method on our **axes** object:

```
axes.axhline(overall_default_rate, color='red')
```

Now, over this line, we want to plot the default rate within each group of **PAY_1** values.

5. Use the **plot** method of the **DataFrame** of grouped data we created. Specify to include an '**x**' marker along the line plot, to not have a **legend** instance, which we'll create later, and that the **parent axis** of this plot should be the **axes** we are already working with (otherwise, pandas would erase what was already there and create new **axes**):

```
group_by_pay_mean_y.plot(marker='x', legend=False, ax=axes)
```

This is all the data we want to plot.

6. Set the y-axis label and create a **legend** instance (there are many possible options for controlling legend appearance, but a simple way is to provide a list of strings, indicating the labels for the graphical elements in the order they were added to the **axes**):

```
axes.set_ylabel('Proportion of credit defaults')
axes.legend(['Entire dataset', 'Groups of PAY_1'])
```

7. Executing all the code from Steps 3 through 6 in a single code cell should result in the following plot:

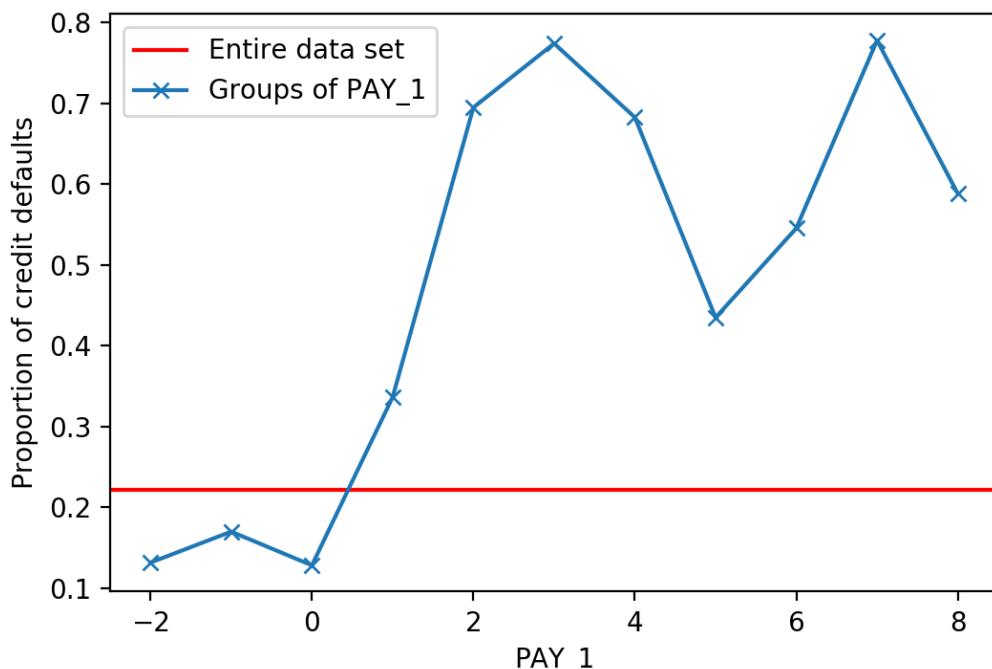


Figure 3.15: Credit default rates across all the data

Our visualization of payment statuses has revealed a clear, and probably expected, story: those who defaulted before, are in fact more likely to default again. The default rate of accounts in good standing is well below the overall default rate, which we know from before is about 22%. However, at least 30% of the accounts that were in default last month will be in default again next month, according to this. This is a good visual to share with our business partner as it shows the effect of what may be the most important feature in our model.

Now we turn our attention to the feature ranked as having the second strongest association with the target variable: **LIMIT_BAL**. This is a numerical feature with many unique values. A good way to visualize features such as this, for a classification problem, is to plot multiple histograms on the same axis, with different colors for the different classes. As a way to separate the classes, we can index them from the DataFrame using logical arrays.

8. Use this code to create logical masks for positive and negative samples:

```
pos_mask = y == 1
neg_mask = y == 0
```

To create our dual histogram plot, we'll make another `axes` object, then call the `.hist` method on it twice for the positive and negative class histograms. We supply a few additional keyword arguments: `alpha` creates transparency in the histograms, so that if they overlap we can still see each of them, and we specify the colors. The blue and red colors, with transparency, will show a purple color in places where the histograms overlap. Once we have the histograms, we rotate the x-axis tick labels to make them more legible and create several other annotations that should be self-explanatory.

9. Use the following code to create the dual histogram plot with the aforementioned properties:

```
axes = plt.axes()
axes.hist(df.loc[neg_mask, 'LIMIT_BAL'], alpha=0.5, color='blue')
axes.hist(df.loc[pos_mask, 'LIMIT_BAL'], alpha=0.5, color='red')
axes.tick_params(axis='x', labelrotation=45)
axes.set_xlabel('Credit limit (NT$)')
axes.set_ylabel('Number of accounts')
axes.legend(['Not defaulted', 'Defaulted'])
axes.set_title('Credit limits by response variable')
```

The plot should appear like this:

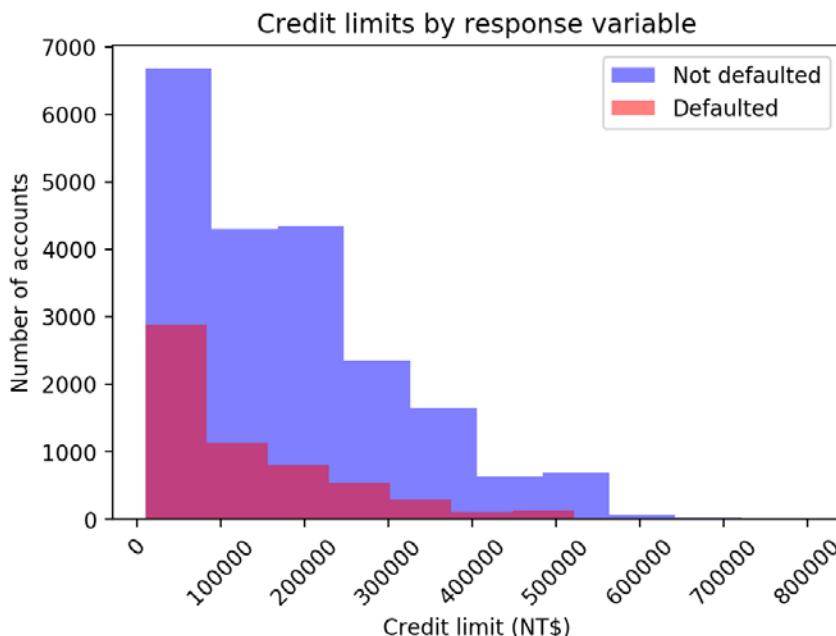


Figure 3.16: Dual histograms of credit limits

While this plot has accomplished all the formatting we wished to present, it's not quite as interpretable as it could be. What we hope to gain from looking at it, is some knowledge of how the credit limit may be a good way to distinguish between accounts that default, and those that do not. However, the primary visual takeaway here is that the blue histogram is bigger than the red one. This is due to the fact that fewer accounts default, than don't default. We already know this from examining the class fractions.

It would be more informative to show something about how the shapes of these histograms are different, not just their sizes. To emphasize this, we can make the total plotted area of the two histograms the same, by **normalizing** them. Matplotlib provides a keyword argument that makes this easy, creating what might be considered an empirical version of a **probability mass function**. This means that the integral, or area contained within each histogram, will be equal to 1 after normalization, since probabilities sum to 1.

After some experimentation, we decide to make a histogram with 16 bins. Since the maximum credit limit is NT\$800,000, we use `range` with an increment of NT\$50000.

10. Create the histogram bin edges with this code, which also prints the final bin edge as a check:

```
bin_edges = list(range(0,850000,50000))
print(bin_edges[-1])
```

The output should be:

```
df['LIMIT_BAL'].max()
```

```
800000
```

```
bin_edges = list(range(0,850000,50000))
print(bin_edges[-1])
```

```
800000
```

Figure 3.17: Observing the maximum credit limit and creating bin edges for normalized histograms

The plotting code for the normalized histograms is similar to before, with a few key changes: the use of the `bins` keyword to define bin edge locations, `density=True` to normalize the histograms, and changes to the plot annotations. The most complex part is that we need to adjust the `y tick labels`, so that the heights of the histogram bins have the interpretation of proportions, which is more intuitive than the default output.

Y tick labels are the text labels displayed next to the ticks on the y-axis and are usually simply the values of the ticks at those locations. However, you are able to manually change this if you want.

Note

According to the Matplotlib documentation, for a normalized histogram the bin heights are calculated by "dividing the count by the number of observations times the bin width" (https://matplotlib.org/api/_as_gen/matplotlib.pyplot.hist.html). So, we need to multiply the y-axis tick labels by the bin width of NT\$50,000, for the bin heights to represent the proportion of the total number of samples in each bin. Notice the two lines where we get the tick locations of the y-axis, then set the labels to a modified version. The rounding to two decimal places with `np.round` is needed due to slight errors of floating point arithmetic.

11. Run this code to produce normalized histograms:

```
mpl.rcParams['figure.dpi'] = 400
axes = plt.axes()
axes.hist(df.loc[neg_mask, 'LIMIT_BAL'], bins=bin_edges, alpha=0.5,
density=True, color='blue')
axes.hist(df.loc[pos_mask, 'LIMIT_BAL'], bins=bin_edges, alpha=0.5,
density=True, color='red')
axes.tick_params(axis='x', labelrotation=45)
axes.set_xlabel('Credit limit (NT$)')
axes.set_ylabel('Proportion of accounts')
y_ticks = axes.get_yticks()
axes.set_yticklabels(np.round(y_ticks*50000,2))
axes.legend(['Not defaulted', 'Defaulted'])
axes.set_title('Normalized distributions of credit limits by response
variable')
```

The plot should look like this:

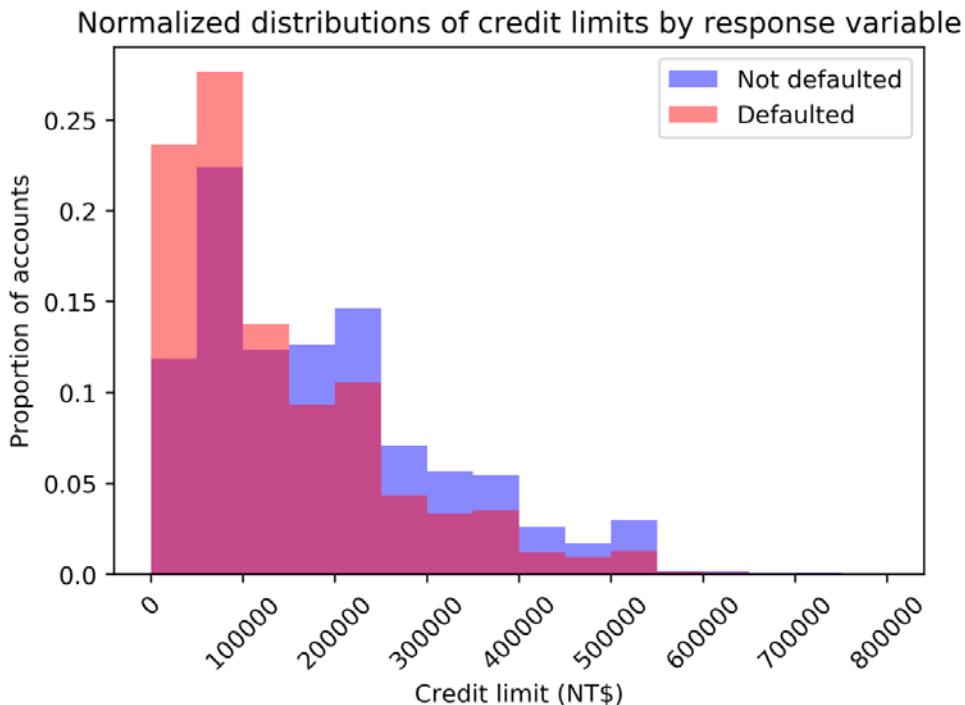


Figure 3.18: Normalized dual histograms

You can see that plots in Matplotlib are highly customizable. In order to view all the different things you can **get** from, and **set** on Matplotlib axes, have a look here: https://matplotlib.org/api/axes_api.html.

What can we learn from this plot? It looks like the accounts that default tend to have a higher proportion of lower credit limits. Accounts with credit limits less than NT\$150,000 are relatively more likely to default, while the opposite is true for accounts with limits higher than this. We should ask ourselves, does this make sense? Our hypothesis was that the client would give riskier accounts lower limits. This intuition is compatible with the higher proportions of defaulters with lower credit limits that we observed here.

Depending on how the model building goes, if the features we examined in this exercise turn out to be important for predictive modeling as we might expect, it would be good to show these graphs to our client, as part of a presentation of our work. This would give the client insight into how the model works.

A key learning from this section is that effective visual presentations take substantial time to produce. It is good to budget some time in your project workflow for this. Convincing visuals are worth the effort since they should be able to quickly and effectively communicate important findings to the client. They are usually a better choice than adding lots of text to the materials that you create. Visual communication of quantitative concepts is a core data science skill.

Univariate Feature Selection: What It Does and Doesn't Do

In this chapter, we have learned techniques for going through features one by one to see whether they have predictive power. This is a good first step, and if you already have features that are very predictive of the outcome variable, you may not need to spend much more time considering features before modeling. However, there are drawbacks to univariate feature selection. In particular, it does not consider the **interactions** between features. For example, what if the credit default rate is very high specifically for people with a certain education level and a certain range of credit limit?

Also, with the methods we used here, only the linear effects of features are captured. If a feature is more predictive when it's undergone some type of **transformation**, such as a **polynomial** or **logarithmic** transformation, or **binning (discretization)**, linear techniques of univariate feature selection may not be effective. Interactions and transformations are examples of **feature engineering**, or creating new features, in these cases from existing features. The shortcomings of linear feature selection methods can be remedied by non-linear modeling techniques including random forest, which we will examine later. But there is still value in looking for simple relationships that can be found by linear methods for univariate feature selection, and it is quick to do.

Understanding Logistic Regression with function Syntax in Python and the Sigmoid Function

In this section, we will open the black box all the way: we will gain a comprehensive understanding of how logistic regression works. We start off by introducing a new programming concept: **functions**. At the same time, we'll learn about a mathematical function that plays a key role in logistic regression.

In the most basic sense, a function in computer programming is a piece of code that takes inputs and produces outputs. You have been using functions throughout the book: functions that were written by someone else. Anytime that you use syntax such as this: `output = do_something_to(input)`, you have used a function. For example, NumPy has a function you can use to calculate the mean of the input:

```
np.mean([1, 2, 3, 4, 5])
```

```
3.0
```

Figure 3.19: The mean function in NumPy

Functions **abstract** away the operations being performed so that, in our example, you don't need to see all the lines of code that it takes to calculate a mean, every time you need to do this. For many common mathematical functions, there are already pre-defined versions available in packages such as NumPy. You do not need to "reinvent the wheel". The implementations in popular packages are likely popular for a reason: people have spent time thinking about how to create them in the most efficient way. So, it would be wise to use them. However, since all the packages we are using are **open source**, if you are interested to see how the functions in the libraries we use are implemented, you are able to look at the code within any of them.

Now, for the sake of illustration, let's learn Python function syntax by writing our own function for the arithmetic mean. Function syntax in Python is similar to **for** or **if** blocks, in that the body of a function is indented and the declaration of the function is followed by a colon. Here is the code for a function to compute the mean:

```
def my_mean(input_argument):
    output = sum(input_argument)/len(input_argument)
    return(output)
```

After you execute the code cell with this definition, the function is available to you in other code cells in the notebook. For example:

```
def my_mean(input_argument):
    output = sum(input_argument)/len(input_argument)
    return(output)
```

```
my_mean([1, 2, 3, 4, 5])
```

```
3.0
```

Figure 3.20: Calculating the mean with a user-defined function

The first part to defining a function, as shown here, is to start a line of code with **def**, followed by a space, followed by the name you'd like to call the function. After this come parentheses, inside which the names of the **parameters** of the function are specified. Parameters are names of the input variables, where these names are internal to the body of the function: the variable names defined as parameters are available within the function when it is **called** (used), but not outside the function. There can be more than one parameter; they would be comma-separated. After the parentheses comes a colon.

The body of the function is indented and can contain any code that operates on the inputs. Once these operations are done, the last line should start with **return** and contain the output variable(s), comma-separated if there are more than one. We are leaving out many fine points in this very simple introduction to functions, but those are the essential parts you need to get started.

The power of a function comes when you use it. Notice how after we define the function, in a separate code block we can **call** it by the name we've given it, and it operates on whatever inputs we **pass** it. It's as if we've copied and pasted all the code to this new location. But it looks much nicer than actually doing that. And if you are going to use the same code many times, a function can greatly reduce the overall length of your code.

As a brief additional note, you can specify the inputs using the parameter names explicitly, which can be clearer when there are many inputs.

```
my_mean(input_argument=[1, 2, 3])
```

```
2.0
```

Figure 3.21: Using a function with a parameter name

Now that we're familiar with the basics of Python functions, we are going to consider a mathematical function that's important to logistic regression, called the **sigmoid**. This function may also be called the **logistic function**. The definition of the sigmoid is:

$$f(X) = \text{sigmoid}(X) = \frac{1}{1 + e^{-X}}$$

Figure 3.22: The sigmoid function

We will break down the different parts of this function. As you can see, the sigmoid function involves the **irrational number e**, which is also known as the base of the **natural logarithm**, in contrast to the base-10 logarithms we used earlier for data exploration. In order to compute e^{-x} using Python, we don't actually need to perform the exponentiation manually. NumPy has a convenient function **exp** that takes e to the input exponent automatically. If you look at the documentation, you will see this process is called taking the "exponential", which sounds vague. But it is assumed to be understood that the base of the exponent is e in this case. In general, if you want to take an exponent in Python, such as 2³ ("two to the third power"), the syntax is 2 asterisks: **2**3**, which equals 8, for example.

Consider how inputs may be passed to the **np.exp** function. Since NumPy's implementation is **vectorized**, this function can take individual numbers as well as arrays or matrices as input. First, we compute the exponential of 1, which shows the approximate value of e, as well as e^0 , which of course equals 1, as does the zeroth power of any base:

```
np.exp(1)  
2.718281828459045
```

```
np.exp(0)  
1.0
```

Figure 3.23: Exp(1) and exp(0) with NumPy

To illustrate the vectorized implementation of `np.exp`, we create an array of numbers using NumPy's `linspace` function. This function takes as input the starting and stopping points of a range, both inclusive, and the number of values you'd like within that range, to create an array of that many linearly spaced values. This function performs a somewhat similar role as Python's `range`, but can also produce decimal values:

```
X_exp = np.linspace(-4, 4, 81)
print(X_exp[:5])
print(X_exp[-5:])
```

```
[ -4. -3.9 -3.8 -3.7 -3.6]
[ 3.6  3.7  3.8  3.9  4. ]
```

```
Y_exp = np.exp(X_exp)
plt.plot(X_exp, Y_exp)
plt.title('Plot of $e^x$')
```

Figure 3.24: Using `np.linspace` to make an array

Since `np.exp` is vectorized, it will compute the exponential of the whole array at once, in an efficient manner. Here is the code with output, to calculate the exponential of our array `X_exp` and examine the first five values:

```
Y_exp = np.exp(X_exp)
Y_exp[:5]
```

```
array([0.01831564, 0.02024191, 0.02237077, 0.02472353, 0.02732372])
```

Figure 3.25: NumPy's `exp` function

Exercise 13: Plotting the Sigmoid Function

In this exercise, we will use `X_exp` and `Y_exp`, created previously, to create a plot of what the exponential function looks like over the interval $[-4, 4]$. You need to have run all the code in Figures 3.23 and 3.24 to have these variables available for this exercise. Then we will define a function for the sigmoid, create a plot of that, and consider how it is related to the exponential function. Perform the following steps to complete the exercise:

Note

The code and the resulting output for this exercise have been loaded in a Jupyter Notebook that can be found here: <http://bit.ly/2Dz5iNA>.

1. Use this code to plot the exponential function:

```
plt.plot(X_exp, Y_exp)  
plt.title('Plot of $e^X$')
```

The plot should look like this:

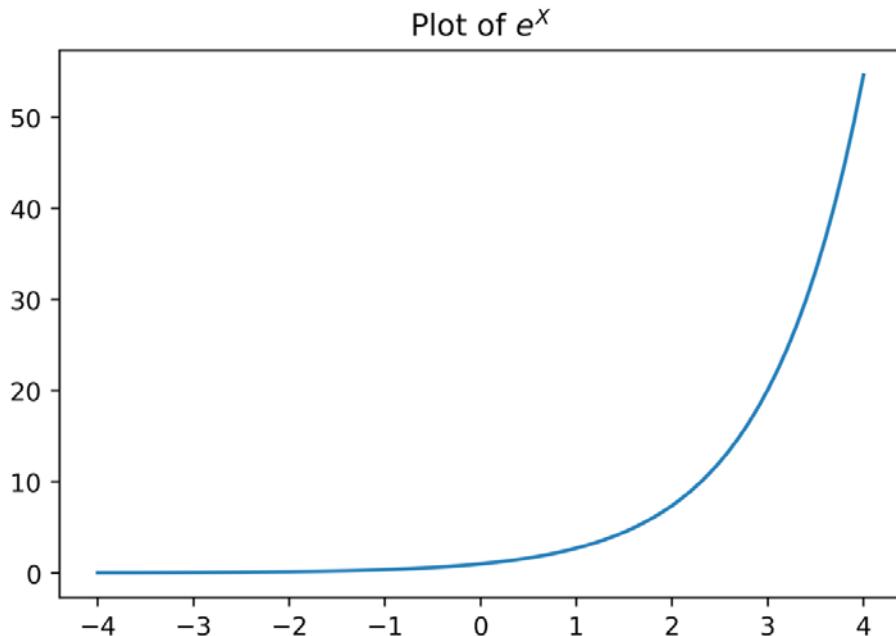


Figure 3.26: Plotting the exponential function

Notice that in titling the plot, we've taken advantage of a kind of syntax called **LaTeX**, which enables formatting of mathematical notation. We won't go in to the details of LaTeX here, but suffice to say that it is very flexible. Note that enclosing part of the title string in dollar signs causes it to be rendered using LaTeX, and that superscript can be created using ^.

Also note in figure 3.26 that many points spaced close together create the appearance of a smooth curve, but in fact it is a graph of discrete points connected by line segments.

What can we observe about the exponential function?

It is never negative: as X approaches negative infinity, Y approaches 0.

As X increases, Y increases slowly at first, but very quickly "blows up". This is what is meant when people say "exponential growth" to signify a rapid increase.

How can you think about the sigmoid in terms of the exponential?

First, the sigmoid involves e^{-x} , as opposed to e^x . The graph of e^{-x} is just the reflection of e^x about the y -axis. This can be plotted easily, using curly braces for multiple-character superscript in the plot title.

2. Run this code to see the plot of e^{-x} :

```
Y_exp = np.exp(-X_exp)
plt.plot(X_exp, Y_exp)
plt.title('Plot of $e^{-X}$')
```

The output should appear like this:

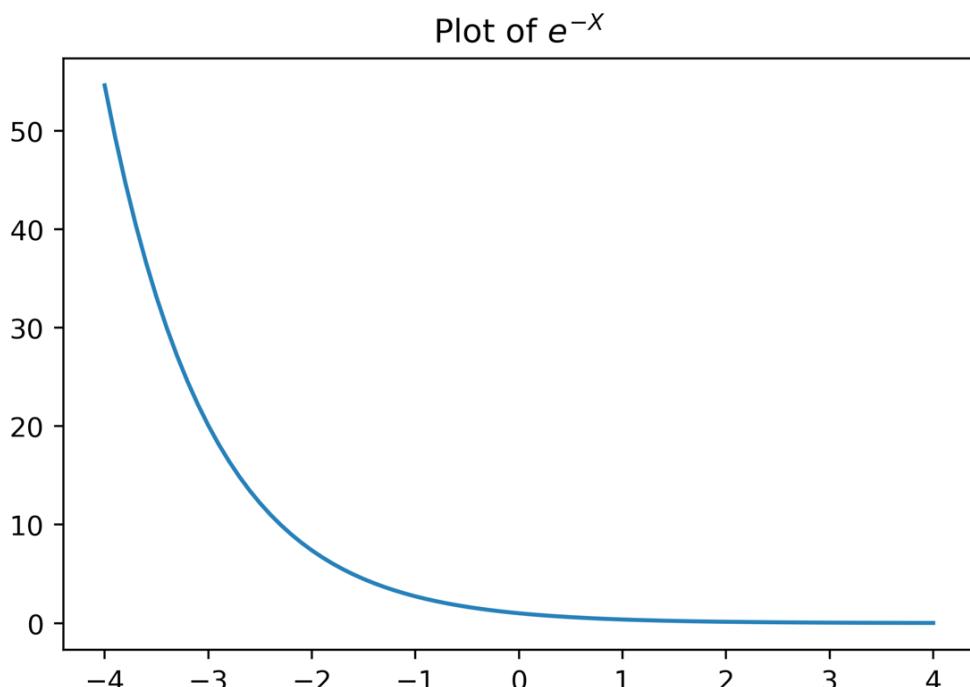


Figure 3.27: Plot of $\exp(-X)$

Now, in the sigmoid function, e^{-X} is in the denominator, with 1 added to it. The numerator is 1. So, what happens to the sigmoid, as X approaches negative infinity? We know that e^{-X} "blows up", becoming very large. Overall the denominator becomes very large and the fraction approaches 0. What about when X increases toward positive infinity? We can see that e^{-X} becomes very close to zero. So, in this case, the sigmoid function would be approximately $1/1 = 1$. This should give you an intuition that the sigmoid function stays between 0 and 1. Let's now implement a sigmoid function in Python and use it to create a plot to see how reality matches this intuition.

3. Define a sigmoid function like this:

```
def sigmoid(X):
    Y = 1 / (1 + np.exp(-X))
    return Y
```

4. Make a larger range of x-values to plot over and plot the sigmoid. Use this code:

```
X_sig = np.linspace(-7, 7, 141)
Y_sig = sigmoid(X_sig)
plt.plot(X_sig, Y_sig)
plt.yticks(np.linspace(0, 1, 11))
plt.grid()
plt.title('The sigmoid function')
```

The plot should look like this:

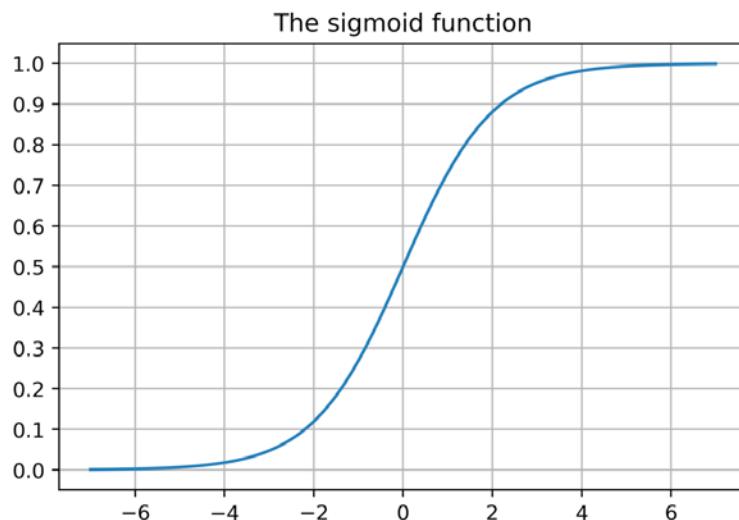


Figure 3.28: A sigmoid function plot

This image matches what we expected. Further, we can see that $\text{sigmoid}(0) = 0.5$. What is special about the sigmoid function? The output of this function is strictly bounded between 0 and 1. This is a good property for a function that should predict probabilities, which are also required to be between 0 and 1. Technically probabilities can be exactly equal to 0 and 1, while the sigmoid never is. But the sigmoid can be close enough that this is not a practical limitation.

Recall that we have described logistic regression as producing **predicted probabilities** of class membership, as opposed to directly predicting class membership. This enables a more flexible implementation of logistic regression, allowing selection of the threshold probability. The sigmoid function is the source of these predicted probabilities. Shortly, we will see how the different features are used in the calculation of the predicted probabilities.

Scope of Functions

As you begin to use functions, you should develop an awareness of the concept of **scope**. Notice that when we wrote the **sigmoid** function, we created a variable, **Y**, inside the function. Variables created inside functions are different from those created outside functions. They are effectively created and destroyed within the function itself, when it is called. These variables are said to be **local** in scope: local to the function. If you have been running all the code as written in this chapter in a single notebook in sequence, notice that you are not able to access the **Y** variable after using the **sigmoid** function:

```
Y

-----
NameError                                 Traceback (most recent call last)
<ipython-input-46-c881daf1af41> in <module>
----> 1 Y

NameError: name 'Y' is not defined
```

Figure 3.29: The **Y** variable not in the scope of the notebook

The `Y` variable is not in the **global** scope of the notebook. However, global variables created outside of functions are available within the local scope of functions, even if they are not input as parameters to the function. Here we demonstrate creating a variable outside of a function, which is global in scope, and then accessing it within a function. The function actually doesn't take any parameters at all, but as you can see it can work with the value of the global variable to create an output:

```
example_global_variable = 1

def example_function():
    output = example_global_variable + 1
    return(output)

example_function()

2
```

Figure 3.30: Global variable available within local scope of function

Note

More details on scope

The scope of variables can potentially be confusing but is good to know when you start making more advanced use of functions. While this knowledge isn't required for the book, you may wish to get a more in-depth perspective on variable scope in Python here: https://nbviewer.jupyter.org/github/rasbt/python_reference/blob/master/tutorials/scope_resolution_legb_rule.ipynb.

Sigmoid curves in scientific applications

Besides being fundamental to logistic regression, sigmoid curves are used in a variety of applications. In biology, they can be used to describe the growth of an organism, that starts slowly, then has a rapid phase, followed by a smooth tapering off as the final size is reached. Sigmoid can also be used to describe population growth, which has a similar trajectory, increasing rapidly but then slowing as the carrying capacity of the environment is reached.

Why is Logistic Regression Considered a Linear Model?

We mentioned previously that logistic regression is considered a **linear model**, while we were exploring whether the relationship between features and response resembled a linear relationship. Recall that we plotted a `groupby/mean` analysis for the **EDUCATION** feature in *Chapter 1, Data Exploration and Cleaning* as well as for the **PAY_1** feature in this chapter, to see whether the default rates across values of these features exhibited a linear trend. While this is a good way to get a quick approximation of how "linear or not" these features may be, here we formalize the notion of why logistic regression is a linear model.

A model is considered linear if the transformation of features that is used to calculate the prediction is a **linear combination** of the features. The possibilities for a linear combination are that each feature can be multiplied by a numerical constant, these terms can be added together, and an additional constant can be added. For example, in a simple model with two features, X_1 and X_2 , a linear combination would take the form:

$$\text{Linear combination of } X_1 \text{ and } X_2 = \theta_0 + \theta_1 X_1 + \theta_2 X_2$$

Figure 3.31: Linear combination of X_1 and X_2

The constants θ_i , can be any number, positive, negative, or zero, for $i = 0, 1, \text{ and } 2$ (although if a coefficient is 0, this removes a feature from the linear combination). A familiar example of a linear transformation of one variable is a straight line with the equation $y = mx + b$. In this case, $\theta_0 = b$ and $\theta_1 = m$. θ_0 is called the **intercept** of a linear combination, which should make sense when thinking about the equation of a straight line in slope-intercept form like this.

What kinds of things are "not allowed" in linear transformations? Any other mathematical expression besides what was just described, such as the following:

- Multiplying a feature by itself; for example, X_1^2 or X_1^3 . These are called polynomial terms.
- Multiplying features together; for example, $X_1 X_2$. These are called interactions.
- Applying non-linear transformations to features; for example, log and square root.
- Other complex mathematical functions.
- "If then" type of statements. For example, "if $X_1 > a$, then $y = b$."

However, while these transformations are not part of the basic formulation of a linear combination, they could be added to a linear model by **engineering features**, for example defining a new feature, $X_3 = X_1^2$.

Earlier, we learned that the predictions of logistic regression, which take the form of probabilities, are made using the sigmoid function. This function is clearly non-linear and is given by the following:

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

Figure 3.32: Non-linear sigmoid function

Why, then, is logistic regression considered a linear model? It turns out that the answer to this question lies in a different formulation of the sigmoid equation, called the **logit** function. We can derive the logit function by solving the sigmoid function for X; in other words, finding the inverse of the sigmoid function. First, we set the sigmoid equal to p , the probability of observing the positive class, then solve for X as shown in the following:

$$p = \frac{1}{1 + e^{-X}}$$

$$1 + e^{-X} = \frac{1}{p}$$

$$e^{-X} = \frac{1}{p} - 1$$

$$e^{-X} = \frac{1-p}{p}$$

$$e^X = \frac{p}{1-p}$$

$$X = \log\left(\frac{p}{1-p}\right)$$

Figure 3.33: Solving for X

Here, we've used some laws of exponents and logs to solve for X. You may also see the logit expressed as:

$$X = \log\left(\frac{p}{q}\right)$$

Figure 3.34: Logit function

The **probability of failure**, q , is expressed in terms of the **probability of success**, p : $q = 1 - p$, because probabilities sum to 1. Even though in our case, credit default would probably be considered a failure in the sense of real-world outcomes, the positive outcome (response variable = 1 in a binary problem) is conventionally considered "success" in mathematical terminology. The logit function is also called the **log odds**, because it is the natural logarithm of the **odds ratio**, p/q . Odds ratios may be familiar from the world of gambling, via phrases such as "the odds are 2 to 1 that team a defeats team b."

In general, what we've called capital X in these manipulations can stand for a linear combination of all the features. For example, this would be $X = \theta_0 + \theta_1X_1 + \theta_2X_2$ in our simple case of two features. Logistic regression is considered a linear model because the features included in X are, in fact, only subject to a linear combination when the response variable is considered to be the log odds. This is an alternative way of formulating the problem, as compared to the sigmoid equation.

In summary, the features X_1, X_2, \dots, X_j look like this in the sigmoid equation version of logistic regression:

$$p = \frac{1}{1 + e^{-(\theta_0 + \theta_1X_1 + \theta_2X_2 + \dots + \theta_jX_j)}}$$

Figure 3.35: Sigmoid version of logistic regression

But they look like this in the log odds version, which is why logistic regression is called a linear model:

$$\theta_0 + \theta_1X_1 + \theta_2X_2 + \dots + \theta_jX_j = \log\left(\frac{p}{q}\right)$$

Figure 3.36: Log odds version of logistic regression

Because of this way of looking at logistic regression, ideally the features of a logistic regression model would be **linear in the log odds** of the response variable. We will see what is meant by this in the following exercise.

Logistic regression is part of a broader class of statistical models called **Generalized Linear Models (GLMs)**. GLMs are connected to the fundamental concept of ordinary linear regression, which may have one feature (that is, the **line of best fit**, $y = mx + b$, for a single feature, x) or more than one in **multiple linear regression**. The mathematical connection between GLMs and linear regression is the **link function**. The link function of logistic regression is the logit function we just learned about.

Exercise 14: Examining the Appropriateness of Features for Logistic Regression

In Exercise 12, *Visualizing the Relationship Between Features and Response*, we plotted a **groupby/mean** of what might be the most important feature of the model, according to our exploration so far: the **PAY_1** feature. By grouping samples by the values of **PAY_1**, and then looking at the mean of the response variable, we are effectively looking at the probability p of default within each of these groups.

In this exercise, we will evaluate the appropriateness of **PAY_1** for logistic regression. We will do this by examining the log odds of default within these groups to see whether the response variable is linear in the log odds, as logistic regression formally assumes. Perform the following steps to complete the exercise:

Note

The code and the resulting output have for this exercise been loaded in a Jupyter Notebook that can be found here: <http://bit.ly/2Dz5iNA>.

1. Confirm you still have access to the variables from *Exercise 12, Visualizing the Relationship between Features and Response*, in your notebook by reviewing the DataFrame of the average value of the response variable for different values of **PAY_1** with this code:

```
group_by_pay_mean_y
```

The output should be:

group_by_pay_mean_y	
	default payment next month
PAY_1	
-2	0.131664
-1	0.170002
0	0.128295
1	0.336400
2	0.694701
3	0.773973
4	0.682540
5	0.434783
6	0.545455
7	0.777778
8	0.588235

Figure 3.37: Rates of default within groups of PAY_1 values as probabilities of default

- Extract the mean values of the response variable from these groups and put them in a variable, **p**, representing the probability of default:

```
p = group_by_pay_mean_y['default payment next month'].values
```

- Create a probability, **q**, of not defaulting. Since there are only two possible outcomes in this binary problem, and probabilities of all outcomes sum to 1, it is easy to calculate **q**. Also print the values of **p** and **q** to confirm:

```
q = 1-p
print(p)
print(q)
```

The output should be:

```
p = group_by_pay_mean_y['default payment next month'].values
q = 1-p
print(p)
print(q)

[0.13166397 0.17000198 0.12829525 0.33639988 0.69470143 0.7739726
 0.68253968 0.43478261 0.54545455 0.77777778 0.58823529]
[0.86833603 0.82999802 0.87170475 0.66360012 0.30529857 0.2260274
 0.31746032 0.56521739 0.45454545 0.22222222 0.41176471]
```

Figure 3.38: Calculating q from

4. Calculate the odds ratio from **p** and **q**, as well as the log odds, using the natural logarithm function from NumPy:

```
odds_ratio = p/q
log_odds = np.log(odds_ratio)
log_odds
```

The output should look like this:

```
odds_ratio = p/q
odds_ratio

array([0.15162791, 0.20482215, 0.14717742, 0.50693161, 2.27548209,
       3.42424242, 2.15      , 0.76923077, 1.2      , 3.5      ,
       1.42857143])

log_odds = np.log(odds_ratio)
log_odds

array([-1.88632574, -1.58561322, -1.91611649, -0.67937918,  0.82219194,
       1.23088026,  0.76546784, -0.26236426,  0.18232156,  1.25276297,
       0.35667494])
```

Figure 3.39: Odds ratio and log odds

5. In order to plot the log odds against the values of the feature, we can get the feature values from the index of the DataFrame containing the **groupby/mean**. You can show the index like this:

```
group_by_pay_mean_y.index
```

This should produce the following output:

```
group_by_pay_mean_y.index
```

```
Int64Index([-2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8], dtype='int64', name='PAY_1')
```

Figure 3.40: How to get values from the index of a pandas DataFrame

6. Create a similar plot to what we have already done, to show the log odds against the values of the feature. Here is the code:

```
plt.plot(group_by_pay_mean_y.index, log_odds, '-x')
plt.ylabel('Log odds of default')
plt.xlabel('Values of PAY_1')
```

The plot should look like this:

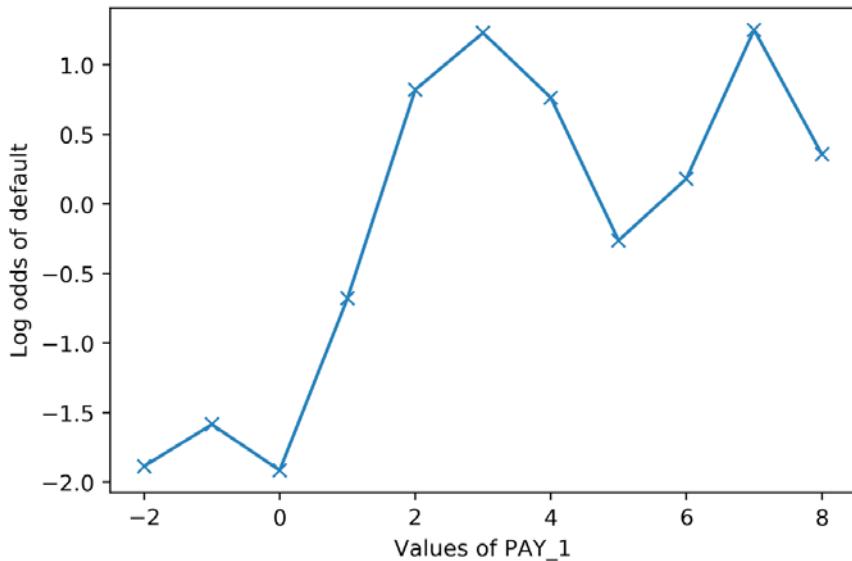


Figure 3.41: Log odds of default for values of PAY_1

We can see in this plot that the relationship between the log odds of the response variable and the **PAY_1** feature is not all that different from the relationship of the rate of default and this feature that we plotted in Exercise 12, *Visualizing the Relationship Between Features and Response*. For this reason, if the "rate of default" is a simpler concept for you to communicate to the business partner, it may be preferable. However, in terms of understanding the workings of logistic regression, this plot shows exactly what is assumed to be linear.

Is a straight line fit a good model for this data?

It certainly seems like a "line of best fit" drawn on this plot would go up from left to right. At the same time, these data don't seem like they would result from a truly linear process. One way to look at these data is that the values -2, -1, and 0, seem like they lie in a different regime of log odds than the others. `PAY_1 = 1` is sort of intermediate, and the rest are mostly larger. It may be that engineered features based on this variable, or different ways of encoding the categories represented by -2, -1, and 0 would be more effective for modeling. Keep this in mind as we proceed to model these data with a logistic regression, and then other approaches later in the book.

From Logistic Regression Coefficients to Predictions Using the Sigmoid

Before the next exercise, let's take a look at how the coefficients for a logistic regression are used to calculate predicted probabilities, and ultimately make predictions for the class of the response variable.

Recall that logistic regression predicts the probability of class membership, according to the sigmoid equation. In the case of two features with an intercept, the equation is:

$$p = \frac{1}{1 + e^{-(\theta_0 + \theta_1 X_1 + \theta_2 X_2)}}$$

Figure 3.42: Sigmoid function to predict probability of class membership for two features

When you call the `.fit` method on a logistic regression model object in scikit-learn using the training data, as we have demonstrated several times, the parameters (intercept and coefficients) θ_0 , θ_1 , and θ_2 are estimated from this labeled training data. Effectively, scikit-learn figures out how to choose values for θ_0 , θ_1 , and θ_2 , so that it would classify as many training data points correctly as possible. We'll gain some insight in to how this process works in the next chapter.

When you call `.predict`, scikit-learn calculates predicted probabilities according to the fitted parameter values and the sigmoid equation. A given sample will then be classified as positive if $p \geq 0.5$, and negative otherwise.

We know that the plot of the sigmoid equation looks like the following, where the x -axis shows the linear combination of the features $X = \theta_0 + \theta_1 X_1 + \theta_2 X_2$:

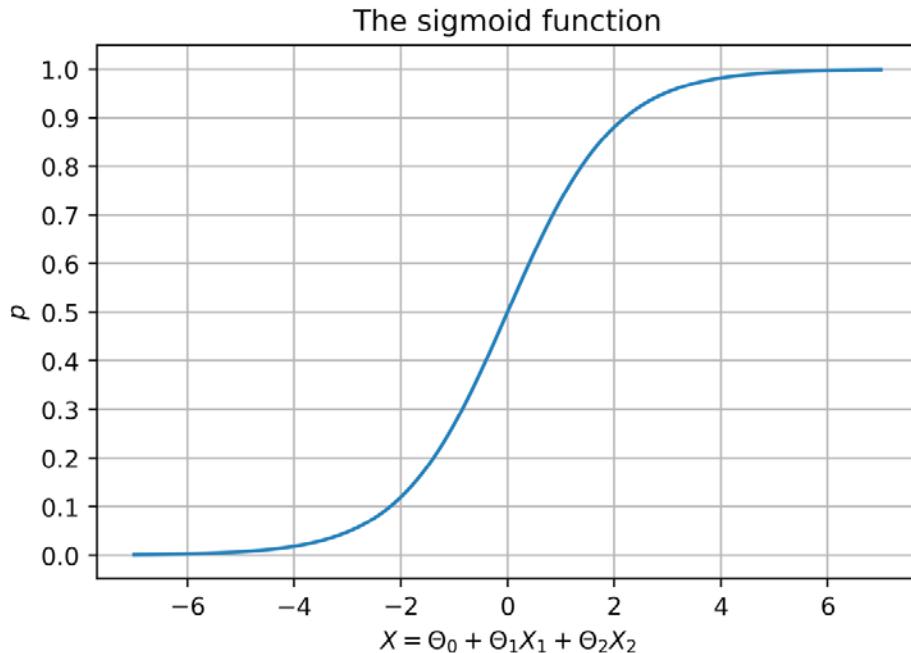


Figure 3.43: Predictions and true classes plotted together

Notice here that if $X = \theta_0 + \theta_1 X_1 + \theta_2 X_2 \geq 0$ on the x-axis, then the predicted probability would be $p \geq 0.5$ on the y-axis and a sample would be classified as positive. Otherwise, $p < 0.5$ and the sample would be classified as negative. We can use this observation to calculate a linear condition for positive prediction, in terms of the features X_1 and X_2 , using the coefficients and intercept. Solving the inequality for positive prediction $X = \theta_0 + \theta_1 X_1 + \theta_2 X_2 \geq 0$ for X_2 , we can obtain a linear inequality similar to a linear equation in $y = mx + b$ form: $X_2 \geq -(\theta_1/\theta_2)X_1 - (\theta_0/\theta_2)$

This will help to see the linear decision boundary of logistic regression in the X_1 - X_2 **feature space** in the following exercise.

We have now learned, from a theoretical and mathematical perspective, why logistic regression is considered a linear model. We also examined a single feature and considered whether the assumption of linearity was appropriate. It is also important to understand the assumption of linearity, in terms of how flexible and powerful we can expect logistic regression to be, when used as a classifier with multiple features.

Exercise 15: Linear Decision Boundary of Logistic Regression

In this exercise, we illustrate the concept of a **decision boundary** for a binary classification problem. We use synthetic data to create a clear example of how the decision boundary of logistic regression looks in comparison to the training samples. We start by generating two features, X_1 and X_2 , at random. Since there are two features, we can say that the data for this problem are two-dimensional. This makes it easy to visualize. The concepts we illustrate here generalize to cases of more than two features, such as the real-world datasets you're likely to see in your work; however, the decision boundary is harder to visualize in higher-dimensional spaces.

Perform the following steps to complete the exercise:

Note

The code and the resulting output for this exercise have been loaded in a Jupyter Notebook that can be found here: <http://bit.ly/2VnbXRL>.

1. Generate the features using the following code:

```
np.random.seed(seed=6)
X_1_pos = np.random.uniform(low=1, high=7, size=(20,1))
print(X_1_pos[0:3])
X_1_neg = np.random.uniform(low=3, high=10, size=(20,1))
print(X_1_neg[0:3])
X_2_pos = np.random.uniform(low=1, high=7, size=(20,1))
print(X_2_pos[0:3])
X_2_neg = np.random.uniform(low=3, high=10, size=(20,1))
print(X_2_neg[0:3])
```

You don't need to worry too much about why we selected the values we did; the plotting we do later should make it clear. Notice, however, that we are also going to assign the true class at the same time. The result of this is that we have 20 samples each in the positive and negative classes, for a total of 40 samples, and that we have two features for each sample. We show the first three values of each feature for both positive and negative classes.

The output should be the following:

```
np.random.seed(seed=6)
X_1_pos = np.random.uniform(low=1, high=7, size=(20,1))
print(X_1_pos[0:3])
X_1_neg = np.random.uniform(low=3, high=10, size=(20,1))
print(X_1_neg[0:3])
X_2_pos = np.random.uniform(low=1, high=7, size=(20,1))
print(X_2_pos[0:3])
X_2_neg = np.random.uniform(low=3, high=10, size=(20,1))
print(X_2_neg[0:3])
```



```
[[6.35716091]
 [2.99187883]
 [5.92737474]]
 [[3.38132155]
 [8.03046066]
 [8.61519394]]
 [[6.35716091]
 [2.99187883]
 [5.92737474]]
 [[3.38132155]
 [8.03046066]
 [8.61519394]]
```

Figure 3.44: Generating synthetic data for a binary classification problem

2. Plot these data, coloring the positive samples in red and the negative samples in blue. The plotting code is as follows:

```
plt.scatter(X_1_pos, X_2_pos, color='red', marker='x')
plt.scatter(X_1_neg, X_2_neg, color='blue', marker='x')
plt.xlabel('$X_1$')
plt.ylabel('$X_2$')
plt.legend(['Positive class', 'Negative class'])
```

The result should look like this:

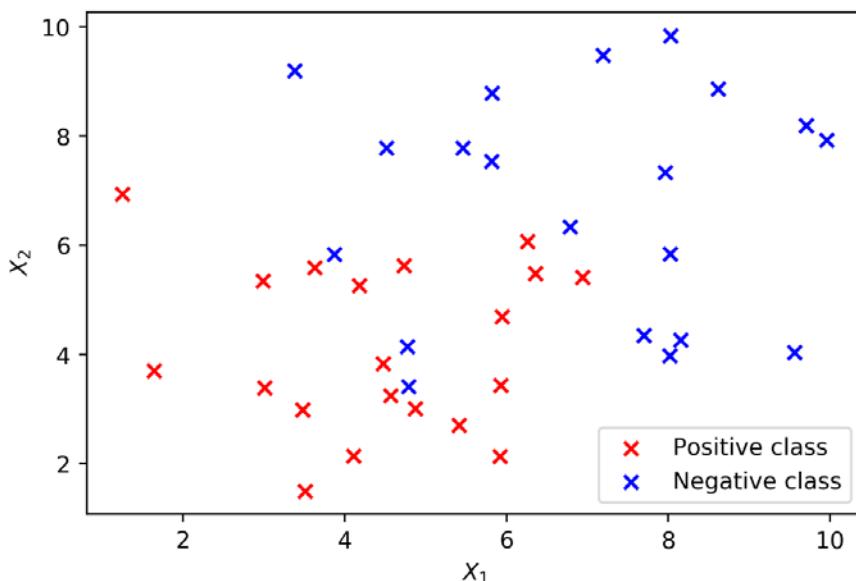


Figure 3.45: Generating synthetic data for a binary classification problem

In order to use our synthetic features with scikit-learn, we need to assemble them into a matrix. We use NumPy's `block` function for this to create a 40 by 2 matrix. There will be 40 rows because there are 40 total samples, and 2 columns because there are 2 features. We will arrange things so that the features for the positive samples come in the first 20 rows, and those for the negative samples after that.

3. Create a 40 by 2 matrix and then show the shape and the first 3 rows:

```
X = np.block([[X_1_pos, X_2_pos], [X_1_neg, X_2_neg]])
print(X.shape)
print(X[0:3])
```

The output should be:

```
(40, 2)
[[ 6.35716091  5.4790643 ]
 [ 2.99187883  5.3444234 ]
 [ 5.92737474  3.43664678 ]]
```

Figure 3.46: Combining synthetic features in to a matrix

We also need a response variable to go with these features. We know how we defined them, but we need an array of `y` values to let scikit-learn know.

4. Create a vertical stack (`vstack`) of 20 1s and then 20 0s to match our arrangement of the features and reshape to the way that scikit-learn expects. Here is the code:

```
y = np.vstack((np.ones((20,1)), np.zeros((20,1)))).reshape(40,)  
print(y[0:5])  
print(y[-5:])
```

You will obtain the following output:

```
[1. 1. 1. 1. 1.]  
[0. 0. 0. 0. 0.]
```

Figure 3.47: Create the response variable for the synthetic data

At this point, we are ready to fit a logistic regression model to these data with scikit-learn. We will use all of the data as training data and examine how well a linear model is able to fit the data. The next few steps should be familiar from your work in earlier chapters on how to instantiate a model class and fit the model.

5. First, import the model class using the following code:

```
from sklearn.linear_model import LogisticRegression
```

6. Now instantiate, indicating the liblinear solver, and show the model object using the following code:

```
example_lr = LogisticRegression(solver='liblinear')  
example_lr
```

The output should be as follows:

```
from sklearn.linear_model import LogisticRegression  
  
example_lr = LogisticRegression(solver='liblinear')  
  
example_lr  
  
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,  
                   intercept_scaling=1, max_iter=100, multi_class='warn',  
                   n_jobs=None, penalty='l2', random_state=None, solver='liblinear',  
                   tol=0.0001, verbose=0, warm_start=False)  
  
example_lr.fit(X, y)  
  
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,  
                   intercept_scaling=1, max_iter=100, multi_class='warn',  
                   n_jobs=None, penalty='l2', random_state=None, solver='liblinear',  
                   tol=0.0001, verbose=0, warm_start=False)
```

Figure 3.48: Fit a logistic regression model to the synthetic data in scikit-learn

- Now train the model on the synthetic data:

```
example_lr.fit(X, y)
```

How do the predictions from our fitted model look?

We first need to obtain these predictions, by using the trained model's `.predict` method on the same samples we used for model training. Then, in order to add these predictions to the plot, using the color scheme of red = positive class and blue = negative class, we will create two lists of indices to use with the arrays, according to whether the prediction is 1 or 0. See whether you can understand how we've used a list comprehension, including an `if` statement, to accomplish this.

- Use this code to get predictions and separate them into indices of positive and negative class predictions. Show the indices of positive class predictions as a check:

```
y_pred = example_lr.predict(X)
positive_indices = [counter for counter in range(len(y_pred)) if y_
pred[counter]==1]
negative_indices = [counter for counter in range(len(y_pred)) if y_
pred[counter]==0]
positive_indices
```

The output should be:

```
y_pred = example_lr.predict(X)

positive_indices = [counter for counter in range(len(y_pred)) if y_pred[counter]==1]
negative_indices = [counter for counter in range(len(y_pred)) if y_pred[counter]==0]

positive_indices
[1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 16, 17, 32, 38]
```

Figure 3.49: Positive class prediction indices

From the indices of positive predictions, we can already tell that not every sample in the training data was classified correctly: the positive samples were the first 20 samples, but there are indices outside of that range here. You may have already guessed that a linear decision boundary would not be able to perfectly classify these data, based on your discussion. Now let's put these predictions on the plot, in the form of circles around each data point, colored according to the prediction. You can compare the color of the `X` symbols, the true labels of the data, to the color of the circles (predictions), to see which points were classified correctly and incorrectly.

9. Here is the plotting code:

```
plt.scatter(X_1_pos, X_2_pos, color='red', marker='x')
plt.scatter(X_1_neg, X_2_neg, color='blue', marker='x')
plt.scatter(X[positive_indices,0], X[positive_indices,1], s=150,
marker='o',
    edgecolors='red', facecolors='none')
plt.scatter(X[negative_indices,0], X[negative_indices,1], s=150,
marker='o',
    edgecolors='blue', facecolors='none')
plt.xlabel('$X_1$')
plt.ylabel('$X_2$')
plt.legend(['Positive class', 'Negative class', 'Positive predictions',
'Negative predictions'])
```

The plot should appear as follows:

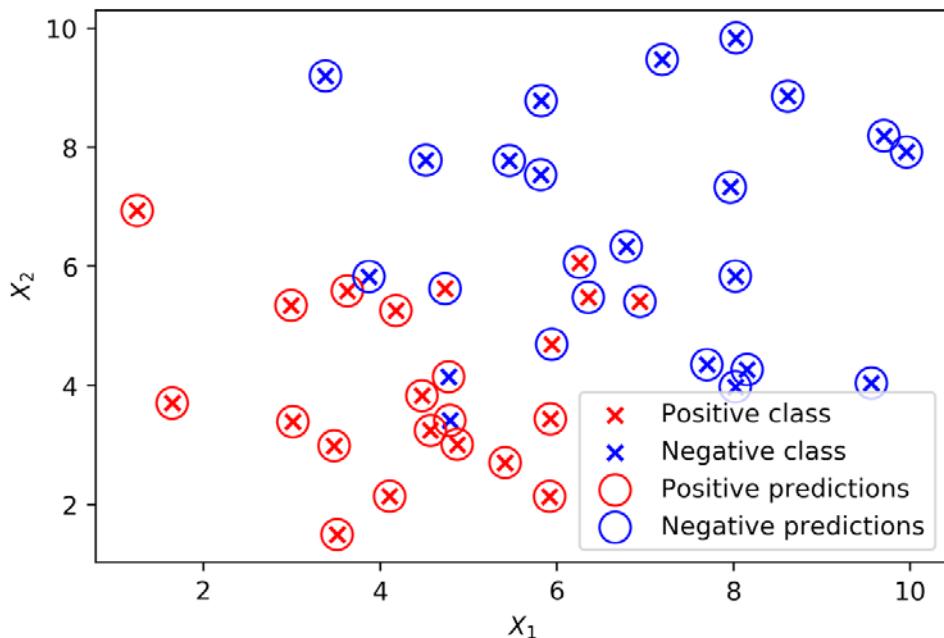


Figure 3.50: Predictions and true classes plotted together

From the plot, it's apparent that the classifier struggles with data points that are close to where you may imagine the linear decision boundary to be; some of these may end up on the wrong side of that boundary. How might we figure out, and visualize, the actual location of the decision boundary? From the previous section, we know we can obtain the decision boundary of a logistic regression, in two-dimensional feature space, using the inequality $X_2 \geq -(\theta_1/\theta_2)X_1 - (\theta_0/\theta_2)$. Since we've fitted the model here, we can retrieve the coefficients θ_1 and θ_2 , as well as the intercept θ_0 , to plug in to this equation and create the plot.

10. Use this code to get the coefficients from the fitted model and print them:

```
theta_1 = example_lr.coef_[0][0]
theta_2 = example_lr.coef_[0][1]
print(theta_1, theta_2)
```

The output should look like this:

-0.20245058016285858 -0.253364236267732

Figure 3.51: Coefficients from the fitted model

11. Use this code to get the intercept:

```
theta_0 = example_lr.intercept_
```

Now use the coefficients and intercept to define the linear decision boundary. This captures the dividing line of the inequality, $X_2 \geq -(\theta_1/\theta_2)X_1 - (\theta_0/\theta_2)$:

```
X_1_decision_boundary = np.array([0, 10])
X_2_decision_boundary = -(theta_1/theta_2)*X_1_decision_boundary -
(theta_0/theta_2)
```

To summarize the last few steps, after using the `.coef_` and `.intercept_` methods to retrieve the model coefficients θ_1 , θ_2 and the intercept θ_0 , we then used these to create a line defined by two points, according to the equation we described for the decision boundary.

12. Plot the decision boundary using the following code, with some adjustments to assign the correct labels for the legend, and to move the legend to a location (**loc**) outside a plot that is getting crowded:

```
pos_true = plt.scatter(X_1_pos, X_2_pos, color='red', marker='x',
label='Positive class')
neg_true = plt.scatter(X_1_neg, X_2_neg, color='blue', marker='x',
label='Negative class')
pos_pred = plt.scatter(X[positive_indices,0], X[positive_indices,1],
s=150, marker='o',
edgecolors='red', facecolors='none', label='Positive
predictions')
neg_pred = plt.scatter(X[negative_indices,0], X[negative_indices,1],
s=150, marker='o',
edgecolors='blue', facecolors='none', label='Negative
predictions')
dec = plt.plot(X_1_decision_boundary, X_2_decision_boundary, 'k-',
label='Decision boundary')
plt.xlabel('$X_1$')
plt.ylabel('$X_2$')
plt.legend(loc=[0.25, 1.05])
```

You will obtain the following plot:

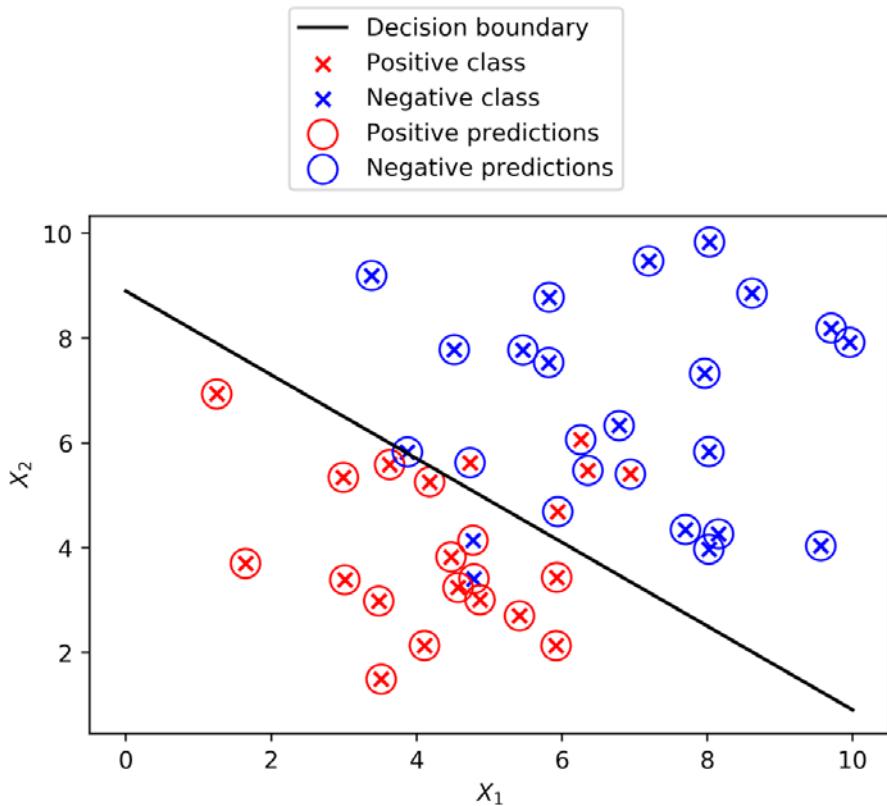


Figure 3.52: True classes, predicted classes, and the decision boundary of a logistic regression

How does the location of the decision boundary compare with where you thought it would be?

Can you see how a linear decision boundary will never perfectly classify these data?

As a way around this, we could create **engineered features** from existing features here, such as polynomials or interactions, to allow for more complex, non-linear decision boundaries in logistic regression. Or, we could use non-linear models such as random forest, which can also accomplish this, as we'll see later.

As a final note here, this example was easily visualized in two dimensions since there are only two features. In general, the decision boundary can be described by a **hyperplane**, which is the generalization of a straight line to multi-dimensional spaces. However, the restrictive nature of the linear decision boundary is still a factor for hyperplanes.

Activity 3: Fitting a Logistic Regression Model and Directly Using the Coefficients

In this activity, we're going to train a logistic regression model on the two most important features we discovered in univariate feature exploration, as well as learn how to manually implement logistic regression using coefficients from the fitted model. This will show you how you could use logistic regression in a computing environment where scikit-learn may not be available, but the mathematical functions necessary to compute the sigmoid function are. On successful completion of the activity, you should observe that the calculated ROC AUC values using scikit-learn predictions and those obtained from manual predictions should be the same: approximately 0.63.

Perform the following steps to complete the activity:

Note

The code and the resulting output for this activity have been loaded in a Jupyter Notebook that can be found here: <http://bit.ly/2Dz5iNA>.

1. Create a train/test split (80/20) with **PAY_1** and **LIMIT_BAL** as features.
2. Import **LogisticRegression**, with the default options, but set the solver to '**liblinear**'.
3. Train on the training data and obtain predicted classes, as well as class probabilities, using the testing data.

4. Pull out the coefficients and intercept from the trained model and manually calculate predicted probabilities. You'll need to add a column of 1s to your features, to multiply by the intercept.
5. Using a threshold of 0.5, manually calculate predicted classes. Compare this to the class predictions output by scikit-learn.
6. Calculate ROC AUC using both scikit-learn's predicted probabilities, and your manually predicted probabilities, and compare.

Note

The solution for this activity can be found on page 336.

Summary

In this chapter, we have learned how to explore features one at a time, using univariate feature selection methods including Pearson correlation and an ANOVA F-test. While looking at features in this way does not always tell the whole story, since you are potentially missing out on important interactions between features, it is a necessary step. Understanding the relationships between the most predictive features and the response variable, and creating effective visualizations around them, is a great way to communicate your findings to your client. We used customized plots, such as overlapping histograms created with Matplotlib, to create visualizations of the most important features.

Then we began an in-depth description of how logistic regression works, exploring such topics as the sigmoid function, log odds, and the linear decision boundary. While logistic regression is one of the simplest classification models, and often is not as powerful as other methods, it is one of the most widely used, and is the basis for more sophisticated models such as deep neural networks for classification. So, a detailed understanding of logistic regression can serve you well as you explore more advanced topics in machine learning. And, in some cases, a simple logistic regression may be all that's needed. All other things considered, the simplest model that satisfies the requirements is probably the best model.

If you master the materials in this and the next chapter, you will be well prepared to use logistic regression in your work. In the next chapter, we'll build on the fundamentals we learned here, to see how coefficients are estimated for a logistic regression, as well as how logistic regression can be used effectively with large numbers of features, and can also be used for feature selection.

4

The Bias-Variance Trade-off

Learning Objectives

By the end of this chapter, you will be able to:

- Describe the log-loss cost function of logistic regression.
- Implement the gradient descent procedure for estimating model parameters.
- Articulate the formal statistical assumptions of the logistic regression model.
- Characterize the bias-variance trade-off and use it to improve models.
- Formulate lasso and ridge regularization and use them in scikit-learn.
- Design a function to choose regularization hyperparameters by cross-validation.
- Engineer interaction features to improve an underfit model

This chapter presents the final details of logistic regression and equips you with the tools for improving underfitting and overfitting by employing regularization and simple feature engineering.

Introduction

In this chapter, we will introduce the final details of logistic regression. In addition to being able to use scikit-learn to fit logistic regression models, you will gain insight into the gradient descent procedure, which is similar to the processes that are used "under the hood" to accomplish model fitting. Finally, we'll complete our discussion of the basic logistic regression model by familiarizing ourselves with the formal statistical assumptions of this method.

We begin our exploration of the foundational machine learning concepts of overfitting, underfitting, and the bias-variance trade-off by examining how the logistic regression model can be extended to address the overfitting problem. After reviewing the mathematical details of the regularization methods that are used to alleviate overfitting, you will learn a useful practice for tuning the hyperparameters of regularization: cross-validation. Through the methods of regularization and some simple feature engineering, you will gain an understanding of how to improve both overfit and underfit models.

Estimating the Coefficients and Intercepts of Logistic Regression

In the previous chapter, we learned that the coefficients of a logistic regression (each of which goes with a particular feature), and the intercept, are determined when the `.fit` method is called on a logistic regression model in scikit-learn using the training data. These numbers are called the **parameters** of the model, and the process of finding the best values for them is called parameter **estimation**. Once the parameters are found, the logistic regression model is essentially a finished product; therefore, with just these numbers, we can use the trained logistic regression in any environment where we can perform common mathematical functions.

It is clear that the process of parameter estimation is important, since this is how we can make a functional model using our data. So, how does parameter estimation work? To understand this, the first step is to familiarize ourselves with the concept of a **cost function**. A cost function is a way of telling how far away the model predictions are from perfectly describing the data; that is to say, the larger the errors between the model predictions and the actual data, then the larger the "cost" returned by the cost function. This is a straightforward concept for regression problems: the difference between predictions and true values can be used for the cost, after going through a transformation (such as absolute value or squaring) which makes the value of the cost positive, and then averaging this over all the training samples.

For classification problems, especially in fitting logistic regression models, a typical cost function is the **log-loss** function, also called cross entropy loss. This is the cost function that scikit-learn uses, in a modified form, to fit logistic regression. Here is the definition of the log-loss function:

$$\text{log loss} = \frac{1}{n} \sum_{i=1}^n -(y_i \log(p_i) + (1 - y_i) \log(1 - p_i))$$

Figure 4.1: The log-loss function

Here, there are n training samples, y_i is the true label (0 or 1) of the i^{th} sample, p_i is the predicted probability that the label of the i^{th} sample equals 1, and \log is the natural logarithm. The summation notation (that is, the uppercase Greek letter, Sigma) over all the training samples and division by n serve to take the average of this cost function over all training samples. With this in mind, take a look at the following graph of the natural logarithm function, and consider what the interpretation of this cost function is:

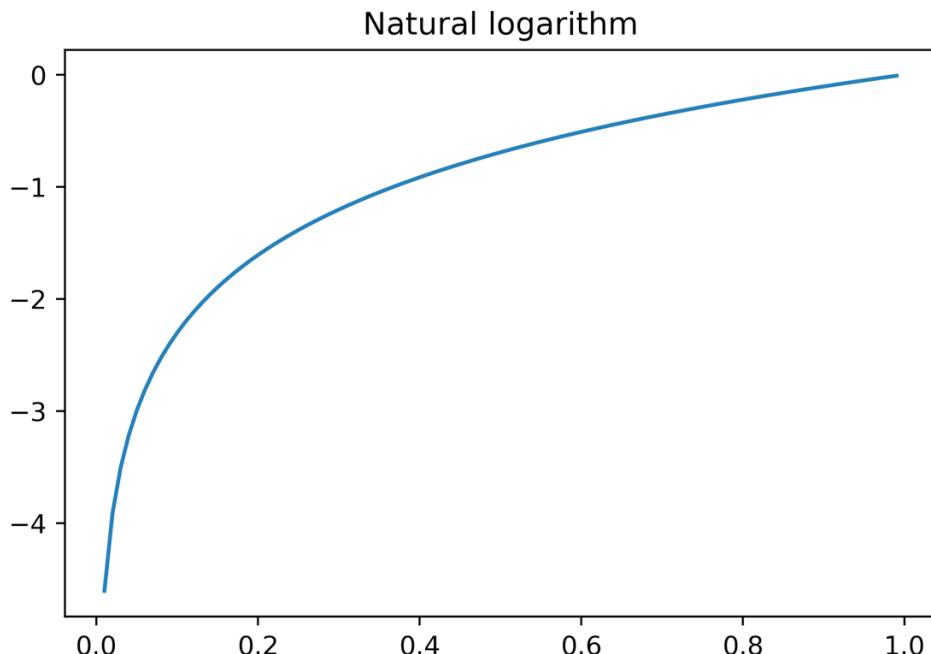


Figure 4.2: Natural logarithm on the interval (0, 1)

To see how the log-loss cost function works, consider its value for a positive sample; $y = 1$ for a positive sample, so this part of the cost function – that is, $(1 - y_i)\log(1 - p_i)$ – will be exactly equal to 0 and will not affect the value. In this case, the value of the cost function is $-y_i\log(p_i) = -\log(p_i)$ since $y_i = 1$. So, the cost for this sample is simply the negative of the natural logarithm of the predicted probability. Since the sample is in fact positive, how should the cost function behave? We expect that for predicted probabilities that are close to 1, the cost function will be small, representing a small error for predictions that are closer to the true value. For predictions that are closer to 0, it will be larger, since the cost function is supposed to take on larger values the more "wrong" the prediction is.

From the graph of the natural logarithm in Figure 4.1 we can see that for values of p that are closer to 0, the natural logarithm takes on increasingly negative values. This means the cost function will take on increasingly positive values, so that the cost of classifying a positive sample with a very low probability is relatively high, as it should be. However, if the predicted probability is closer to 1, then the graph indicates the cost will be closer to 0 – again this is as expected for a prediction that is "more correct". Therefore, the cost function behaves as expected for a positive sample. A similar observation can be made for negative samples.

Now we understand how the log-loss cost function works for logistic regression. But what does this have to do with how the coefficients and the intercept are determined? This will be subject of the next section.

Gradient Descent to Find Optimal Parameter Values

The problem of finding the parameter values (coefficients and intercept) for a logistic regression model using a log-loss cost boils down to a problem of **optimization**: we would like to find the set of parameters that results in the **minimum** cost, since the lowest possible theoretical cost is 0 and costs are higher for worse predictions. In other words, we want the set of parameters that is the "least wrong" on average over all of the training samples. This process is done for you automatically by the `.fit` method of the logistic regression model in scikit-learn. There are different solution techniques for finding the set of parameters with the lowest cost and you can choose which one you would like to use with the `solver` keyword when you are instantiating the model class. All of these methods work somewhat differently. However, they are all based on the concept of **gradient descent**.

Our task is to find the best set of parameters (that is, coefficients and the intercept). The "best" parameters are those which minimize the cost function. Practically speaking, this process starts with an **initial guess**. The choice of the initial guess is not that important for logistic regression and you don't need to make it manually; this is handled by the **solver** keyword. However, for more advanced machine learning algorithms such as deep neural networks, selection of the initial guesses for parameters requires more attention.

For the sake of illustration, we will consider a problem where there is only one parameter to estimate. We'll look at the value of a hypothetical cost function ($y = f(x) = x^2 - 2x$) and devise a gradient descent procedure to find the value of the parameter, x , for which the cost, y , is the lowest. Here, we choose some x values, create a function that returns the value of the cost function, and look at the value of the cost function over this range of parameters.

The code to do this is as follows:

```
X_poly = np.linspace(-3,5,81)  
print(X_poly[:5], '...', X_poly[-5:])
```

Here is the output of the print statement:

```
[-3. -2.9 -2.8 -2.7 -2.6] ... [4.6 4.7 4.8 4.9 5. ]
```

The remaining code snippet is as follows:

```
def cost_function(X):  
    return X * (X-2)  
  
y_poly = cost_function(X_poly)  
plt.plot(X_poly, y_poly)  
plt.xlabel('Parameter value')  
plt.ylabel('Cost function')  
plt.title('Error surface')
```

The resulting plot should appear as follows:

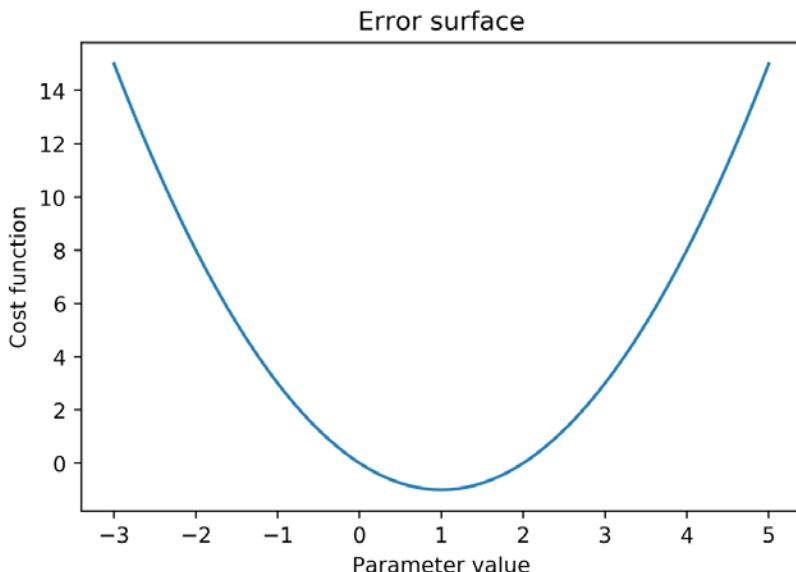


Figure 4.3: A cost function plot

Looking at the **error surface** in Figure 4.3, which is the plot of the cost function over a range of parameter values, it's pretty evident what parameter value will result in the lowest value of the cost function: $x = 1$. In fact, with some calculus, you could easily calculate the minimum by setting the derivative equal to zero and then solving for x , confirming that $x = 1$ is the minimum. However, generally speaking, it is not always feasible to solve the problem so simply. In cases where it is necessary to use gradient descent, we don't always have knowledge of how the entire error surface looks. Rather, after we've chosen the initial guess for the parameter, all we're able to know is the direction of the error surface in the immediate vicinity of that point.

Gradient descent is an iterative algorithm; starting from the initial guess, we try to find a new guess that lowers the cost function and continue with this until we've found a good solution. We are trying to move "downhill" on the error surface, but we only know which direction to move in and how far to move in that direction, based on the shape of the error surface in the immediate neighborhood of our current guess. In mathematical terms, we only know the value of the **derivative** (which is called the **gradient** in more than one dimension) at the parameter value of the current guess. If you have not studied calculus, you can think of the gradient as telling you which direction is downhill, and how steep the hill is from where you're standing. We use this information to "take a step" in the direction of decreasing error. How big a step we decide to take, depends on the **learning rate**. Since the gradient declines toward the direction of decreasing error, we want to take a step in the direction that is the negative of the gradient.

These notions can be formalized in this equation, to get to the new guess, x_{new} , from the current guess, x_{old} , where $f'(x_{old})$ is the derivative (that is, the gradient) of the cost function at the current guess:

$$x_{new} = x_{old} - f'(x_{old}) \times \text{learning rate}$$

Figure 4.4: Equation to obtain the new guess from the current guess

In the following figure, we can see the results of starting a gradient descent procedure from $x = 4.5$, with a learning rate of 0.75, and then optimizing x to attain the least value of the cost function:

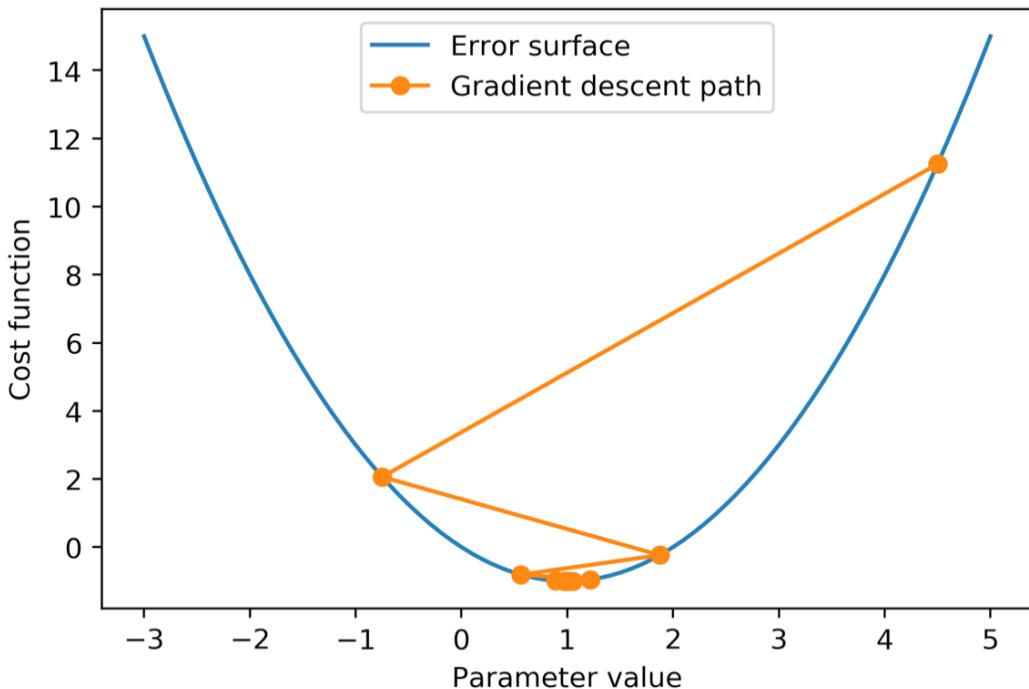


Figure 4.5: The gradient descent path

Gradient descent also works in higher-dimensional spaces; in other words, with more than one parameter. However, you can only visualize up to a two-dimensional error surface (that is, two parameters at a time on a three-dimensional plot) on a single graph.

Having described the workings of gradient descent, let's perform an exercise to implement the gradient descent algorithm, expanding on the example of this section.

Exercise 16: Using Gradient Descent to Minimize a Cost Function

In this exercise, our task is to find the best set of parameters in order to minimize the following hypothetical cost function: $y = f(x) = x^2 - 2x$. To do this, we will employ gradient descent, which was described in the preceding section. Perform the following steps to complete the exercise:

Note

For Exercises 16 to 18 and Activity 4, the code and the resulting output have been loaded in a Jupyter Notebook that can be found at <http://bit.ly/2ZAy2Pr>. You can scroll to the appropriate section within the Jupyter Notebook to locate the exercise or activity of choice.

1. Create a function that returns the value of the cost function and look at the value of the cost function over a range of parameters. You can use the following code to do this (note this repeats code from the preceding section):

```
X_poly = np.linspace(-3,5,81)
print(X_poly[:5], '...', X_poly[-5:])
def cost_function(X):
    return X * (X-2)
y_poly = cost_function(X_poly)
plt.plot(X_poly, y_poly)
plt.xlabel('Parameter value')
plt.ylabel('Cost function')
plt.title('Error surface')
```

You will obtain the following plot of the cost function:

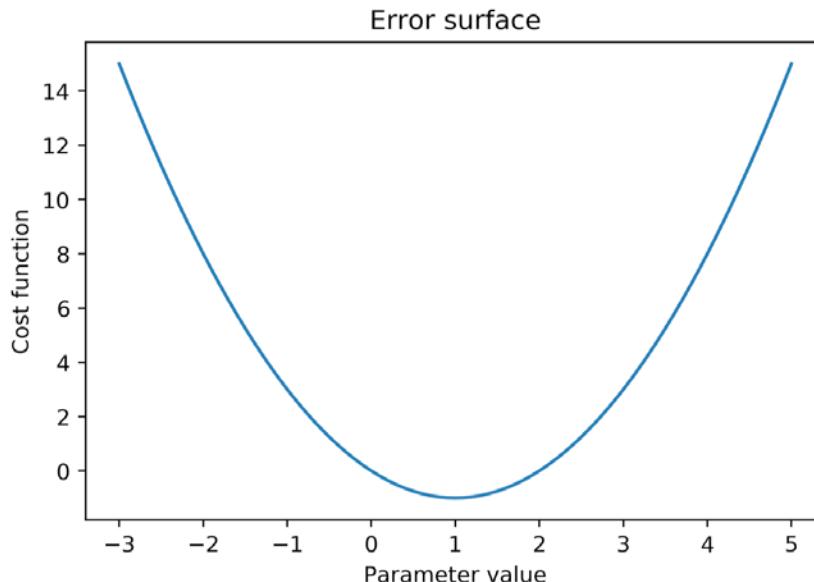


Figure 4.6: A cost function plot

2. Create a function for the value of the gradient. This is the analytical derivative of the cost function. Use this function to evaluate the gradient at the point $x = 4.5$, then use this in combination with the learning rate to find the next step of the gradient descent process.

```
def gradient(X):
    return (2*X) - 2
x_start = 4.5
learning_rate = 0.75
x_next = x_start - gradient(x_start)*learning_rate
x_next
```

Note

It doesn't matter if you haven't studied calculus and don't understand this part; you can just take it as a given that this is the function for the gradient. In some applications it's not actually possible to calculate an analytical derivative, so this may need to be numerically approximated.

After running the cell with `x_next`, you will obtain the following output:

-0.75

This is the next gradient descent step after $x = 4.5$.

3. Plot the gradient descent path, from the starting point to the next point, using the following code:

```
plt.plot(X_poly, y_poly)
plt.plot([x_start, x_next], [cost_function(x_start), cost_function(x_
next)], '-o')
plt.xlabel('Parameter value')
plt.ylabel('Cost function')
plt.legend(['Error surface', 'Gradient descent path'])
```

You will obtain the following output:

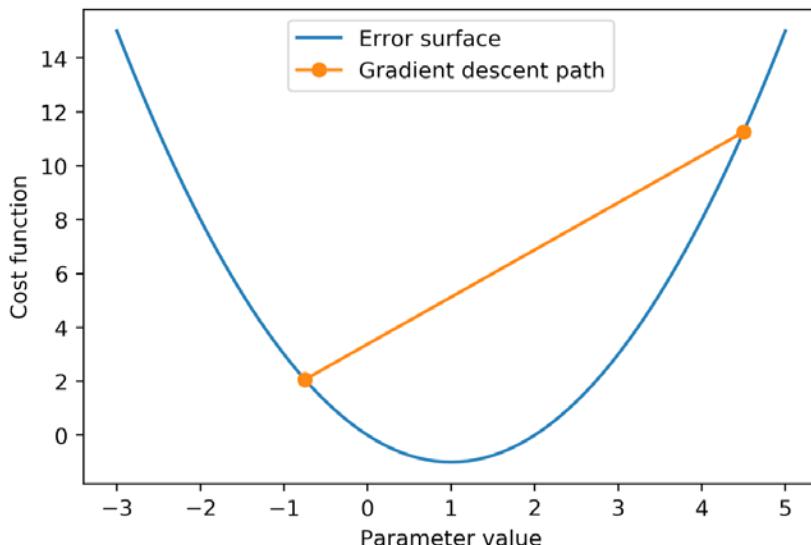


Figure 4.7: The first gradient descent path step

Here, it appears as though we've taken a step in the right direction. However, it's clear that we've "overshot" where we want to be. It may be that our learning rate is too large, and consequently, we are taking steps that are too big. While tuning the learning rate will be a good idea to converge toward an optimal solution more quickly, in this example, we can just continue illustrating the remainder of the process. Here, it looks like we may need to take a few more steps. In practice, gradient descent continues until the size of the steps become very small, or the change in the cost function becomes very small (you can specify how small by using the `tol` argument in the scikit-learn logistic regression), indicating that we're "close enough" to a good solution – that is, a **local minimum** of the cost function. For this example, we'll just take a total of 14 steps, or **iterations**, beyond the initial guess (note that you can also set the maximum number of iterations in scikit-learn with `max_iter`).

4. Perform 14 iterations to converge toward the local minimum of the cost function by using the following code snippet (note that `iterations = 15` but the endpoint is not included in the call to `range()`):

```
iterations = 15
x_path = np.empty(iterations,)
x_path[0] = x_start
for iteration_count in range(1,iterations):
    derivative = gradient(x_path[iteration_count-1])
    x_path[iteration_count] = x_path[iteration_count-1] -
(derivative*learning_rate)
x_path
```

You will obtain the following output:

```
array([ 4.5        , -0.75       ,  1.875      ,  0.5625     ,
       0.890625   ,  1.0546875 ,  0.97265625,  1.01367188,  0.99316406,
       1.00341797,  0.99829102,  1.00085449,  0.99957275,  1.00021362])
```

From the resulting values of the gradient descent process, it looks like (by the end) we've gotten very close (**1.00021362**) to the optimal solution (**1**).

5. Plot the gradient descent path using the following code:

```
plt.plot(X_poly, y_poly)
plt.plot(x_path, cost_function(x_path), '-o')
plt.xlabel('Parameter value')
plt.ylabel('Cost function')
plt.legend(['Error surface', 'Gradient descent path'])
```

You will obtain the following output:

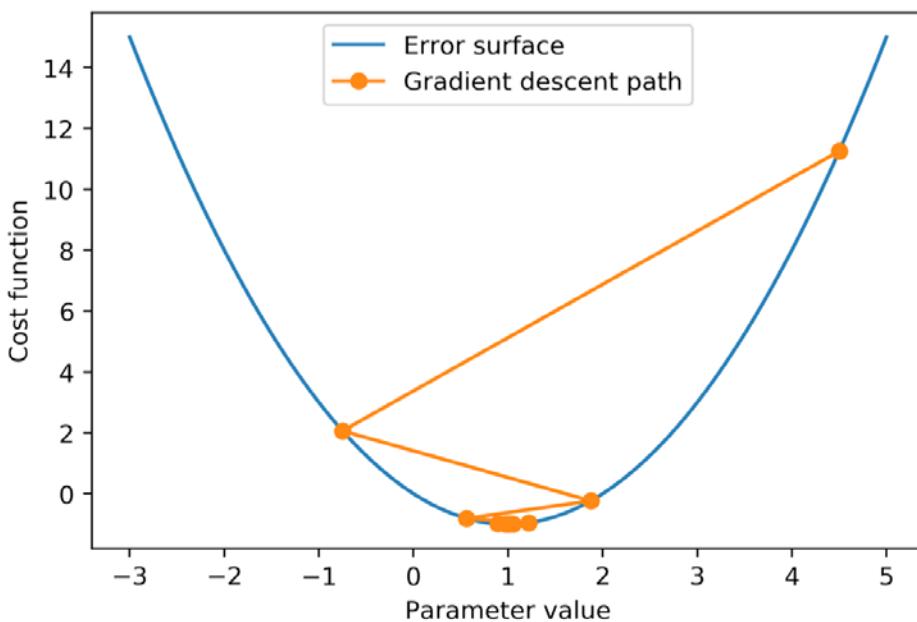


Figure 4.8: The gradient descent path

We encourage you to repeat the previous procedure with different learning rates in order to see how they affect the gradient descent path. With the right learning rate, it's possible to converge on a highly accurate solution very quickly. While the choice of learning rate can be important in different machine learning applications, for logistic regression, the problem is usually pretty easy to solve and you don't need to select a learning rate in scikit-learn.

As you experimented with different learning rates, did you notice what happened when the learning rate was greater than one? In this case, the step that we take in the direction of the decreasing error is too large and we actually wind up with a higher error. This problem can compound itself and actually lead the gradient descent process away from the region of minimum error. On the other hand, if the step size is too small, it can take a very long time to find the desired solution.

Assumptions of Logistic Regression

Since it is a classical statistical model, similar to the F-test and Pearson correlation we already examined, logistic regression makes certain assumptions about the data. While it's not necessary to follow every one of these assumptions in the strictest possible sense, it's good to be aware of them. That way, if a logistic regression model is not performing very well, you can try to investigate and figure out why, using your knowledge of the ideal situation in which a logistic regression would work well. You may find slightly different lists of the specific assumptions from different resources, however those that are listed here are widely accepted.

Features Are Linear in the Log Odds

We learned about this assumption in the previous chapter, *Chapter 3, Details of Logistic Regression and Feature Exploration*. Logistic regression is a linear model, so it will only work well as long as the features are effective at describing a linear trend in the log odds. In particular, logistic regression won't capture interactions, polynomial features, or the discretization of features, on its own. You can, however, specify all of these as "new features" – even though they are engineered from existing features.

Remember from the previous chapter that the most important feature from univariate feature exploration, **PAY_1**, was not found to be linear in the log odds.

No Multicollinearity of Features

Multicollinearity means that features are correlated with each other. The worst violation of this assumption is when features are perfectly correlated with each other, such as one feature being identical to another, or when one feature equals another multiplied by a constant. We can investigate the correlation of features using the correlation plot that we're already familiar with from univariate feature selection. Here, we load the data and recreate the plot, in the same way that we did previously:

Note

Adjust the path in the following code to the location where you saved the cleaned data from *Chapter 1, Data Exploration and Cleaning*.

```

df = pd.read_csv('../Data/Chapter_1_cleaned_data.csv')
features_response = df.columns.tolist()
items_to_remove = ['ID', 'SEX', 'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6',
                    'EDUCATION_CAT', 'graduate school', 'high school',
                    'none',
                    'others', 'university']

features_response = [item for item in features_response if item not in
                     items_to_remove]

corr = df[features_response].corr()

mpl.rcParams['figure.dpi'] = 400 #high res figures

sns.heatmap(corr,
             xticklabels=corr.columns.values,
             yticklabels=corr.columns.values,
             center=0)

```

The resulting plot is as follows:

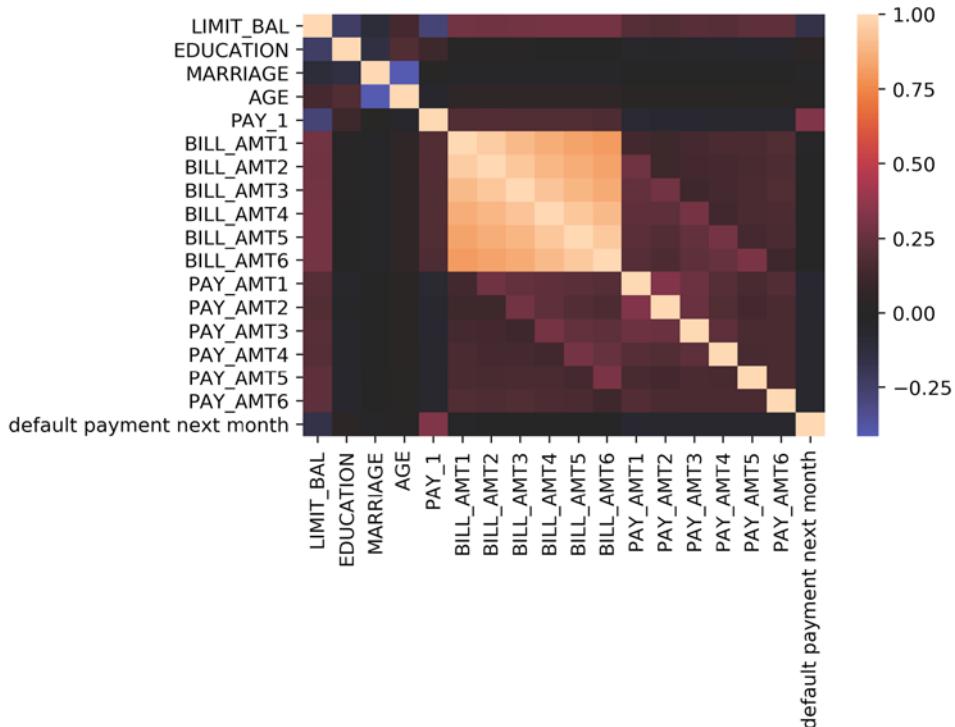


Figure 4.9: A correlation plot of features and the response

We can see from the correlation plot what perfect correlation (that is, $\rho = 1$) looks like: since every feature and the response variable has a correlation of 1 with itself, we can see that a correlation of 1 is a light, cream color. From the color bar, which doesn't include -1, we know there are no correlations with that value.

The clearest examples of correlated predictors in our case study data are the **BILL_AMT** features. It makes intuitive sense that bills are similar from month to month, for instance, in both of these types of accounts: an account that typically carries a balance of zero, or an account that has a large balance that is taking a while to pay off. Are any of the **BILL_AMT** features perfectly correlated? The answer is no, as we can confirm from *Figure 4.9*. So, while these features may not contribute much independent information, we don't see a reason to remove them at this point.

The Independence of Observations

This is a common assumption in classical statistical models, including linear regression. Here, the observations (or samples) are assumed to be independent. Does this make sense with the case study data? We'd want to confirm with our client whether the same individual can hold multiple credit accounts across the dataset and consider what to do depending on how common it was. Let's assume we've been told that in our data each credit account belongs to a unique person, so we may assume independence of observations in this respect.

Across different domains of data, some common violations of independence of observations are as follows:

- **Spatial autocorrelation** of observations; for example, in natural phenomena such as soil types, where observations that are geographically closer to each other may be similar to each other.
- **Temporal autocorrelation** of observations, which may occur in time series data. Observations at the current point in time are usually assumed to be correlated to the most recent point(s) in time.

However, these issues are not relevant to our case study data.

No Outliers

Outliers are observations where the value of the feature(s) or response are very far from most of the data, or are different in some other way. A more appropriate term for an outlier observation of a feature value is a high leverage point, as the term "outlier" is usually applied to the response variable. However, in our binary classification problem, it's not possible to have an outlier value of the response variable, since it can only take on the values between 0 and 1. In practice, you may see both of these terms used to refer to features.

To see why these kinds of points can have an adverse effect on linear models in general, take a look at this synthetic linear data and the line of best fit that results from linear regression:

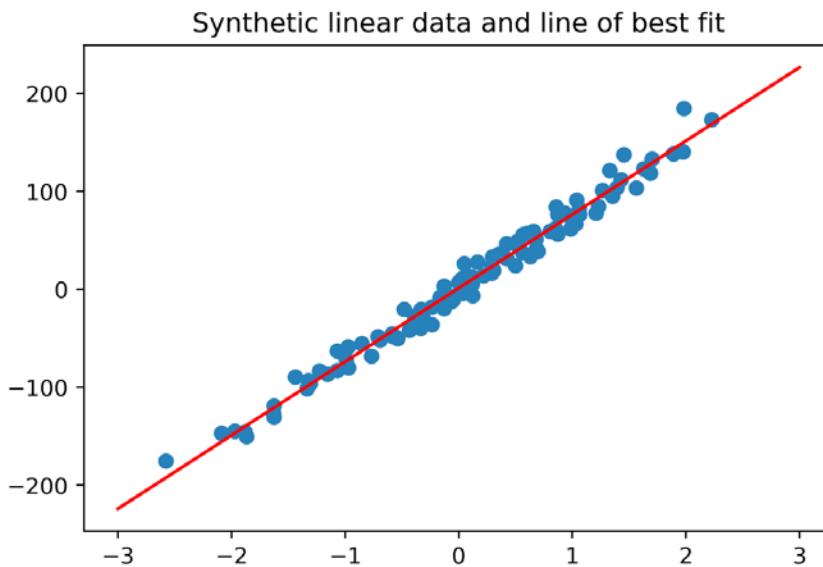


Figure 4.10: "Well-behaved" linear data and a regression fit

Here, the model intuitively appears to be a good fit for the data. However, what if an outlier feature value is added? In this case, we add a point with an x value that is very different from most of the observations and a y value that is in a similar range to the other observations. We can then see the resulting regression line:

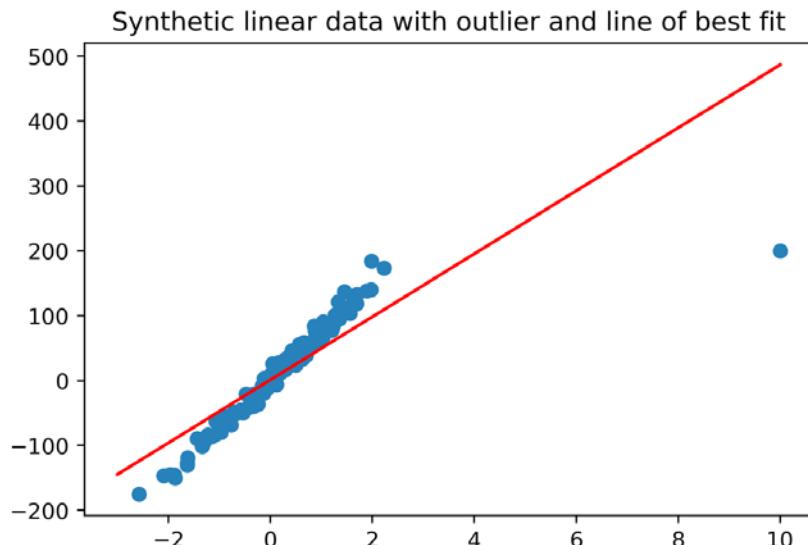


Figure 4.11: A plot showing what happens when an outlier is included

Due to the presence of a single high leverage point, the regression model fit for all the data is no longer a very good representation of much of the data. This shows the potential effect of just a single data point on linear models, especially if that point doesn't appear to follow the same trend as the rest of the data.

There are methods to deal with outliers. But a more fundamental question to ask is "Is data like this realistic?". If the data doesn't seem right, it is a good idea to ask the client whether the outliers are believable. If not, they should be excluded. However, if they do represent valid data, then non-linear models or other methods should be used.

With our case study data, we did not observe outliers in the histograms that we plotted during feature exploration. Therefore, we don't have this concern.

How Many Features Should You Include?

This is not so much an assumption as it is guidance on model building. There is no clear-cut law that states how many features to include in a logistic regression model. However, a common rule of thumb is the "rule of 10," which states that for every 10 occurrences of the rarest outcome class, 1 feature may be added to the model. So, for example, in a binary logistic regression problem with 100 samples, if the class balance has 20% positive outcomes and 80% negative outcomes, then there are only 20 positive outcomes in total, and so only 2 features should be used in the model. A "rule of 20" has also been suggested, which would be a more stringent limit on the number of features to include (1 feature in our example).

Another point to consider in the case of binary features, such as those that result from one-hot encoding, is how many samples will have a positive value for that feature. If the feature is very imbalanced, in other words, with very few samples containing either a 1 or a 0, it may not make sense to include it in the model.

For the case study data, we are fortunate to have a relatively large number of samples and relatively balanced features, so these are not concerns.

The Motivation for Regularization: The Bias-Variance Trade-off

We can extend the basic logistic regression model that we have learned about by using a powerful concept known as **shrinkage** or **regularization**. In fact, every logistic regression that you have fit so far in scikit-learn has used some amount of regularization. This is because this is a default option that is set in the logistic regression model object; however, until now, we have ignored it.

As you learn about these concepts in greater depth, you will also become familiar with a few foundational concepts in machine learning: **overfitting**, **underfitting**, and the **bias-variance trade-off**. A model is said to overfit the training data if the performance of the model on the training data (for example, the ROC AUC) is substantially better than the performance on a held-out test set. In other words, good performance on the training set does not generalize to the unseen test set. We started to discuss these concepts in *Chapter 2, Introduction to Scikit-Learn and Model Evaluation*, when we distinguished between model training and testing scores.

When a model is overfitting the training data, it is said to have high **variance**. That is to say, whatever variability exists in the training data, the model has learned this very well – in fact, too well. This will, therefore, be reflected in a high model training score. However, when such a model is then used to make predictions on new and unseen data, the performance is lower in the model testing phase. Overfitting is more likely in the following circumstances:

1. There are a large number of features available in relation to the number of samples. In particular, there may be so many possible features that it is cumbersome to directly inspect all of them, like we were able to do with the case study data.
2. A complex model, that is, more complex than logistic regression, is used. These include models such as random forests, other ensemble models, or neural networks.

Under these circumstances, the model has an opportunity develop more complex **hypotheses** about the relationship between features and the response variable in the training data during model fitting, making overfitting more likely.

In contrast, if a model is not fitting the training data very well, this is known as underfitting, and the model is said to have high **bias**.

We can examine the differences between underfitting, overfitting, and the ideal that sits in between, by fitting polynomial models on some hypothetical data:

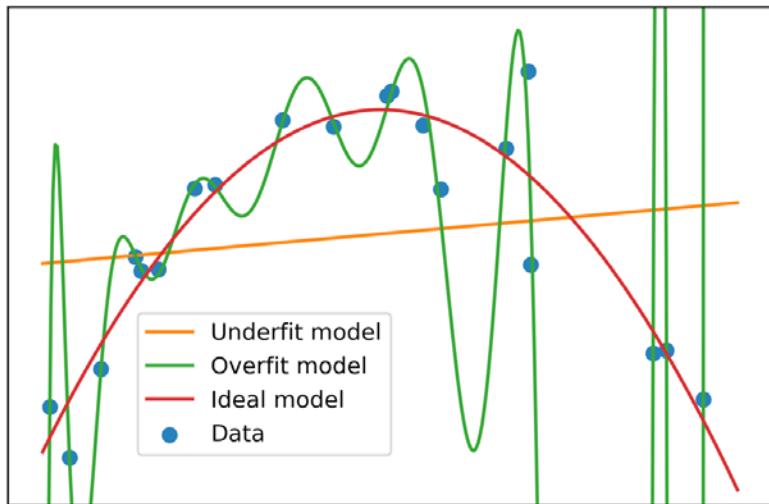


Figure 4.12: Quadratic data with underfit, overfit, and ideal models

In Figure 4.12 we can see that including fewer features, in this case a linear model of y with just two features, a slope, and the intercept, is clearly not a good representation of the data. This is known as an underfit model. However, if we include too many features, that is, with many high-degree polynomial terms, such as $x^2, x^3, x^4, \dots, x^{10}$, we can fit the training data almost perfectly. However, this is not necessarily a good thing. When we look at the results of this model in between the training data points, where new predictions may need to be made, we can see that the model is unstable and may not provide reliable predictions for data that was not in the training set. We can tell this just based on an intuitive understanding of the relationship between the features and the response variable, which we can get from examining the raw data.

Since we know that this data was generated by a second-degree (that is, quadratic) polynomial, we can easily find the ideal model by fitting a second-degree polynomial to the training data, as shown in Figure 4.12. In general, however, we won't know what the ideal model formulation is ahead of time. For this reason, we need to compare training and testing scores to assess whether a model may be overfitting or underfitting.

In some cases, it may be desirable to introduce some bias into the model training process, especially if this decreases overfitting and increases the performance of new, unseen data. In this way, it may be possible to leverage the bias-variance trade-off to improve a model. We can use **regularization** methods to accomplish this. Additionally, we may also be able to use these methods for **variable selection** as part of the modeling process. Using a predictive model to select variables is an alternative to the univariate feature selection methods that we've already explored. We begin to experiment with these concepts in the following exercise.

Exercise 17: Generating and Modeling Synthetic Classification Data

In this exercise, we'll observe overfitting in practice by using a synthetic dataset. Consider yourself in the situation of having been given a binary classification dataset with many candidate features (200), where you don't have time to look through all of them individually. It's possible that some of these features are highly correlated or related in some other way. However, with these many variables, it can be difficult to effectively explore all of them. Additionally, the dataset has relatively few samples: only 1,000. We are going to generate this challenging dataset by using a feature of scikit-learn that allows you to create synthetic datasets for making conceptual explorations such as this. Perform the following steps to complete the exercise:

Note

The code and the resulting output for this exercise have been loaded in a Jupyter Notebook. They can be found at <http://bit.ly/2ZAy2Pr>.

1. Import the `make_classification`, `train_test_split`, `LogisticRegression`, and `roc_auc_score` classes using the following code:

```
from sklearn.datasets import make_classification  
from sklearn.model_selection import train_test_split  
from sklearn.linear_model import LogisticRegression  
from sklearn.metrics import roc_auc_score
```

Notice that we've first imported several familiar classes from scikit-learn, in addition to a new one that we haven't seen before: `make_classification`. This class does just what its name indicates – it makes data for a classification problem. Using the various keyword arguments, you can specify how many samples and features to include, and how many classes the response variable will have. There is also a range of other options that effectively control how "easy" the problem will be to solve.

Note

For more information, refer to https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html. Suffice to say that we've selected options here that make a reasonably easy-to-solve problem, with some curveballs thrown in. In other words, we expect high model performance, but we'll have to work a little bit to get it.

2. Generate a dataset with two variables, `x_synthetic` and `y_synthetic`, 200 candidate features, and 1,000 samples using the following code:

```
X_synthetic, y_synthetic = \ make_classification(n_samples=1000, n_features=200, n_informative=3, n_redundant=10, n_repeated=0, n_classes=2, n_clusters_per_class=2, weights=None, flip_y=0.01, class_sep=0.8, hypercube=True, shift=0.0, scale=1.0, shuffle=True, random_state=24)
```

3. Examine the shape of the dataset using the following code:

```
print(X_synthetic.shape, y_synthetic.shape)
```

You will obtain the following output:

```
(1000, 200) (1000,)
```

Note that we've generated an almost perfectly-balanced dataset: close to a 50/50 class balance. It is also important to note that we've generated all the features so that they have the same `scale` – that is, a mean of 0 with a standard deviation of 1. Making sure that the features are on the same scale, or have roughly the same range of values, is a key point for using regularization methods – and we'll see why later. If the features in a raw dataset are on widely different scales, it is advisable to normalize them so that they are on the same scale. Scikit-learn has functionality to make this easy, which we'll learn about in the activity at the end of this chapter.

4. Plot the first few features as histograms to show that the range of values is the same using the following code:

```
for plot_index in range(4):
    plt.subplot(2,2,plot_index+1)
    plt.hist(X_synthetic[:,plot_index])
    plt.title('Histogram for feature {}'.format(plot_index+1))
plt.tight_layout()
```

You will obtain the following output:

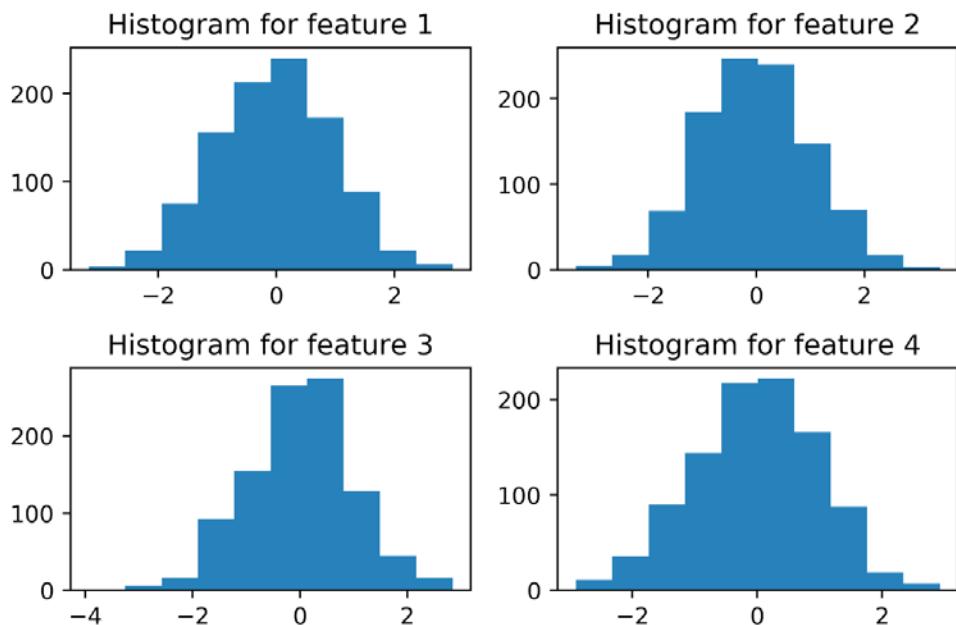


Figure 4.13: Histograms for the first 4 of 200 synthetic features

Because we generated this dataset, we don't need to directly examine all 200 features to make sure that they're on the same scale. So, what are the possible concerns with this dataset? The data is balanced in terms of the class fractions of the response variable, so we don't need to undersample, oversample, or use other methods that are helpful for imbalanced data. What about relationships among the features themselves, and the features and response variable? There are a lot of these relationships and it is a challenge to investigate them all directly. Based on our "rule of thumb", 200 features is too many (especially going by the "rule of 10"). We have 500 observations in the rarest class, so by that rule we shouldn't have more than 50 features. It's possible that with so many features, which we don't have a good idea of the quality of, the model training procedure will overfit.

5. Split the data into training and testing sets using an 80/20 split, and then instantiate a logistic regression model object using the following code:

```
X_syn_train, X_syn_test, y_syn_train, y_syn_test = train_test_split(  
    X_synthetic, y_synthetic,  
    test_size=0.2, random_state=24)  
lr_syn = LogisticRegression(solver='liblinear', penalty='l1', C=1000,  
    random_state=1)  
lr_syn.fit(X_syn_train, y_syn_train)
```

Notice here that we are specifying some new options in the logistic regression model, which so far, we have not paid attention to. First, we specify the **penalty** argument to be **l1**. This means we are going to use **L1 regularization**, which is also known as **lasso regularization**. We'll discuss the mathematical definition of this shortly. Second, notice that we have set the **C** parameter to be equal to 1,000. **C** is the "inverse of regularization strength," according to the scikit-learn documentation (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html). This means that higher values of **C** correspond to less regularization. By choosing a relatively large number such as 1,000, we are using relatively little regularization. The default value of **C** is 1. So, we are not really using much regularization here, rather, we are simply becoming familiar with the options to do so. Finally, we are using the **liblinear** solver, which we have used in the past. Although we happen to be using scaled data here, it's worth noting at this point that among the various options we have available for solvers, **liblinear** is "robust to unscaled data." Also note that **liblinear** is one of only two solver options that support the L1 penalty – the other option being **saga**.

Note

You can find out more information on available solvers at https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression.

6. Fit the logistic regression model on the training data using the following code:

Note

This process reports back all the options that we've indicated when instantiating the model and the defaults that we did not set. We have included these outputs at the relevant instances in the code snippet.

```
lr_syn.fit(X_syn_train, y_syn_train)
```

Here is the output:

```
LogisticRegression(C=1000, class_weight=None, dual=False, fit_
intercept=True,
    intercept_scaling=1, max_iter=100, multi_class='warn',
    n_jobs=None, penalty='l1', random_state=1, solver='liblinear',
    tol=0.0001, verbose=0, warm_start=False)
```

7. Calculate the training score using this code, by first getting predicted probabilities, then obtaining the ROC AUC:

```
y_syn_train_predict_proba = lr_syn.predict_proba(X_syn_train)
roc_auc_score(y_syn_train, y_syn_train_predict_proba[:,1])
```

The output should be:

0.9420000000000001

8. Calculate the testing score similar to how the training score was obtained:

```
y_syn_test_predict_proba = lr_syn.predict_proba(X_syn_test)
roc_auc_score(y_syn_test, y_syn_test_predict_proba[:,1])
```

The output should be:

0.8074807480748075

From these results, it's apparent that the logistic regression model has overfit the data. That is, the ROC AUC score on the training data is substantially higher than that of the testing data.

Lasso (L1) and Ridge (L2) Regularization

Before applying regularization to a logistic regression model, let's take a moment to understand what regularization is and how it works. The two ways of regularizing logistic regression models in scikit-learn are called **lasso** (also known as **L1** regularization) and **ridge** (also known as **L2** regularization). When instantiating the model object from the scikit-learn class, you can choose either **penalty = 'l1'** or **'l2'**. These are called "penalties" because the effect of regularization is to add a penalty, or a cost, for having larger values of the coefficients in a fitted logistic regression model.

As we've already learned, coefficients in a logistic regression model describe the relationship between the log-odds of the response and each of the features. Therefore, if a coefficient value is particularly large, then a small change in that feature will have a large effect on the prediction. When a model is being fit, and is learning the relationship between features and the response variable, the model can start to learn the "noise" in the data. We saw this previously in *Figure 4.12*: if there are many features available when fitting a model, and there are no "guardrails" on the values that their coefficients can take, then the model fitting process may try to discover relationships between the features and the response variable that won't generalize to new data. This is how the model becomes tuned to the unpredictable, random noise that accompanies real-world, imperfect data. Unfortunately, this only serves to increase the model's skill at predicting the training data, which is not our ultimate goal. Therefore, we should seek to root out such spurious relationships from the model.

Lasso and ridge regularization use different mathematical formulations to accomplish this goal. These methods work by making changes to the cost function that is used for model fitting, which we introduced previously as the log-loss function. Lasso regularization uses what is called the **1-norm** (hence the term L1):

$$\text{log loss with lasso penalty} = \sum_{j=1}^m |\sigma_j| + \frac{C}{n} \sum_{i=1}^n -(y_i \log(p_i) + (1 - y_i) \log(1 - p_i))$$

Figure 4.14: Log-loss equation with lasso penalty

The 1-norm $\sum_{j=1}^m |\sigma_j|$ is just the sum of the absolute values of all the coefficients of the m different features. The absolute value is used because having a coefficient be large in either the positive or negative directions can contribute to overfitting. So, what else is different about this cost function from the log-loss function that we saw earlier? Well, now there is a C factor that is multiplied by the numerator of the fraction in front of the sum of the log-loss function over all the n training samples. This is the "inverse of regularization strength" as described in the scikit-learn documentation (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html). Since this factor is in front of the term of the cost function that calculates the prediction error, as opposed to the term that does regularization, then making it larger makes the prediction error more important in the cost function, while regularization is made less important. In short, *larger values of C lead to less regularization* in the scikit-learn implementation.

L2, or ridge regularization, is similar to L1, except that instead of the sum of absolute values of coefficients, ridge uses the sum of their squares, called the **2-norm**:

$$\text{log loss with ridge penalty} = \sum_{j=1}^m \sigma_j^2 + \frac{C}{n} \sum_{i=1}^n -(y_i \log(p_i) + (1 - y_i) \log(1 - p_i))$$

Figure 4.15: Log-loss equation with ridge penalty

Note that if you look at the cost functions for logistic regression in the scikit-learn documentation, the specific form is different than what is used here, but the overall idea is similar. Additionally, after you become comfortable with the concepts of lasso and ridge penalties, you should be aware that there is an additional regularization method called **elastic net**, which is a combination of both the lasso and ridge.

Why Are There Two Different Formulations of Regularization?

It may be that one or the other will provide better results for you, so you may wish to test them both. There is another key difference in their results: the L1 penalty also performs feature selection, in addition to regularization. It does this by setting some coefficient values to exactly zero during the regularization process, effectively removing features from the model. L2 regularization makes the coefficient values smaller but does not completely eliminate them. Not all solver options in scikit-learn support both L1 and L2 regularization, so you will need to select an appropriate solver for the regularization technique you want to use.

Note

While the mathematical details of why this happens are beyond the scope of this book, for a more thorough explanation of this topic and further reading in general, we recommend the very readable (and free) resource, *An Introduction to Statistical Learning* by Gareth James, et al. In particular, see page 222 of the corrected 7th printing, which was available at the time of writing, for a helpful graphic on the difference between L1 and L2 regularization.

Intercepts and Regularization

We have not discussed intercepts very much, other than to note that we have been estimating them with our linear models, along with the coefficients that go with each feature. So, should you use an intercept? The answer is probably yes, until you've developed an advanced understanding of linear models and are certain that in a specific case you should not. However, such cases do exist, for example, in a linear regression where the features and the response variable have all been normalized to have a mean of zero.

Intercepts don't go with any particular feature. Therefore, it doesn't make much sense to regularize them, as they shouldn't contribute to overfitting. Notice that in the regularization penalty term for L1, the summation starts with $j = 1$, and similarly for L2: we have skipped σ_0 , which is the intercept term.

This is the ideal situation: not regularizing the intercept. However, because of the way that the different **solvers** are implemented in scikit-learn, the **liblinear** solver actually does this. There is, however, an **intercept_scaling** option that you can supply to the model class to counteract this effect. We have not illustrated this here as, although it is theoretically incorrect, regularizing the intercept often does not have much effect on the model's predictive quality in practice.

Scaling and Regularization

As noted in the previous exercise, it is best practice to **scale** the data so that all the features have roughly the same range of values before using regularization. This is because the coefficients are all going to be subject to the same penalty in the cost function. If the range of values for a particular feature, such as **LIMIT_BAL** in our dataset, is much larger than other features, such as **PAY_1**, it may, in fact, be desirable to have a larger value for the coefficient of **PAY_1** and a smaller value for that of **LIMIT_BAL** in order to put their effects on the same scale in the linear combination of features and coefficients that are used for model prediction. Normalizing all the features before using regularization avoids complications like this that arise simply from differences in scale.

In fact, scaling your data may also be necessary, depending on which solver you are using. The different variations on the gradient descent process available in scikit-learn may or may not be able to effectively work with unscaled data.

The Importance of Selecting the Right Solver

As we've come to learn, the different solvers available for logistic regression in scikit-learn have different behaviors regarding the following:

- Whether they support both L1 and L2 regularization
- How they treat the intercept during regularization
- How they deal with unscaled data

Note

There are other differences as well. A helpful table comparing these and other traits is available at https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression. You can use this table to decide which solver is appropriate for your problem.

To summarize this section, we have learned the mathematical foundations of lasso and ridge regularization. These methods work by shrinking the coefficient values toward 0, and in the case of the lasso, setting some coefficients to exactly 0 and thus performing feature selection. You can imagine that in our example of overfitting in Figure 4.12, if the complex, overfit model had some coefficients shrunk toward 0, it would look more like the ideal model, which has fewer coefficients.

Here is a plot of a regularized regression model, using the same high-degree polynomial features as the overfit model, but with a ridge penalty:

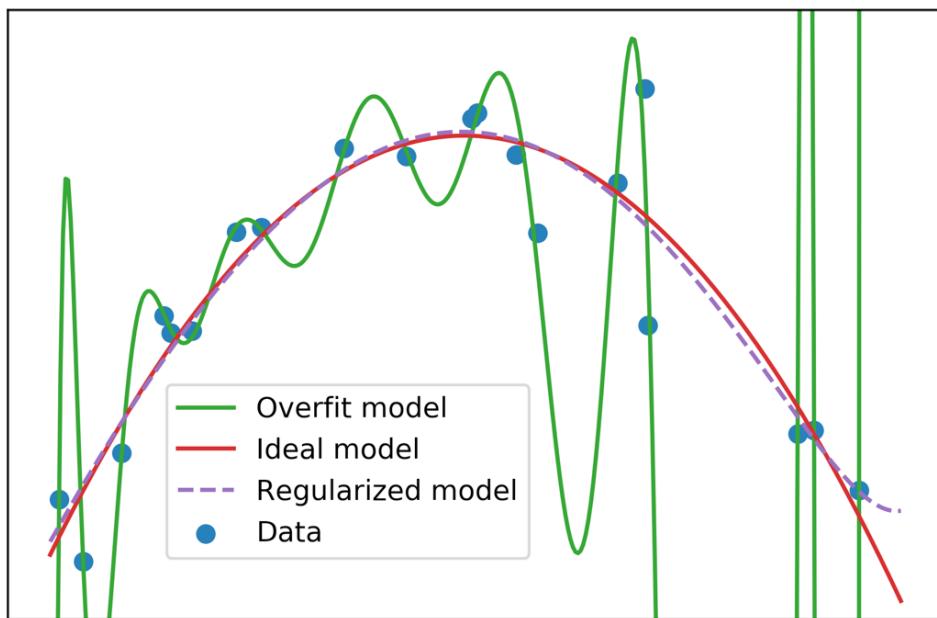


Figure 4.16: An overfit model and regularized model using the same features

The regularized model looks similar to the ideal model. Note, however, that the regularized model should not be recommended for extrapolation. Here, we can see that the regularized model starts to increase toward the right side of Figure 4.16. This increase should be viewed with suspicion, as there is nothing in the training data that makes it clear that this would be expected. This is an example of the general view that the *extrapolation of model predictions outside the range of training data is not recommended*. However, it is clear from Figure 4.16 that even if we didn't have knowledge of the model that was used to generate this synthetic data (as we typically don't have knowledge of the data-generating process in real-world predictive modeling work), we can still use regularization to reduce the effect of overfitting when a large number of candidate features are available.

Models and Feature Selection

L1 regularization is one way to use a model, such as logistic regression, to perform feature selection. Other methods include forward or backward **stepwise selection** with the pool of candidate features. Here is the high-level idea behind these methods: in the case of **forward selection**, features are added to the model one at a time, and the out-of-sample testing performance is observed along the way. At each iteration, the addition of all possible features from the candidate pool is considered, and the one resulting in the greatest increase in the out-of-sample performance is chosen. When adding additional features ceases to improve the model's performance, no more features need to be added from the candidates. In the case of **backward selection**, you first start with all the features in the model and determine which one you should remove: the one resulting in the smallest decrease in the out-of-sample testing performance. You can continue removing features in this way until the performance begins to decrease appreciably.

Cross Validation: Choosing the Regularization Parameter and Other Hyperparameters

By now, you should be interested in using regularization in order to decrease the overfitting we observed when we tried to model the synthetic data in Exercise 17, *Generating and modeling Synthetic Classification Data*. The question is, how do we choose the regularization parameter, C? C is an example of a model **hyperparameter**. Hyperparameters are different from the parameters that are estimated when a model is trained, such as the coefficients and the intercept of a logistic regression. Rather than being estimated by an automated procedure like the parameters are, hyperparameters are input directly by the user as keyword arguments, typically when instantiating the model class. So, how do we know what values to choose?

Hyperparameters are more difficult to estimate than parameters. This is because it is up to the data scientist to determine what the best value is, as opposed to letting an optimization algorithm find it. However, it is possible to programmatically choose hyperparameter values, which could arguably be viewed as a kind of optimization procedure in its own right. Practically speaking, in the case of the regularization parameter C, for example, this is most commonly done by fitting the model on one set of data with a particular value of C, determining model training performance, and then assessing the out-of-sample performance on another set of data.

We are already familiar with the concept of using model training and testing sets. However, there is a key difference here; for instance, what would happen if we were to use the testing set multiple times in order to see the effect of different values of C?

It may occur to you, that after the first time you use the unseen test set to assess the out-of-sample performance for a particular value of C, it is no longer an "unseen test set." While only the training data was used for estimating the model parameters (that is, the coefficients and the intercept), now the testing data is being used to estimate the hyperparameter C. Effectively, the testing data has now become additional training data, in the sense that it is being used to find a good value for the hyperparameter.

For this reason, it is common to divide the data into three parts: a training set, a testing set, and a **validation set**. The validation set serves multiple purposes:

Estimating Hyperparameters

The validation set can be repeatedly used to assess the out-of-sample performance with different hyperparameter values, to select hyperparameters.

A Comparison of Different Models

In addition to finding hyperparameter values for a model, the validation set can be used to determine the out-of-sample performance of different models; for example, if we wanted to compare logistic regression to random forest.

Note

Data Management Best Practices

As a data scientist, it's up to you to figure out how to divide up your data for different predictive modeling tasks. In the ideal case, you should reserve a portion of your data for the very end of the process, after you've already selected model hyperparameters and also selected the best model. This **unseen test set** is reserved for the last step, when it can be used to assess the endpoint of your model building efforts, to see how the final model generalizes to new unseen data. When reserving the testing set, it is good practice to make sure that the features and response have similar characteristics to the rest of the data. In other words, the class fraction should be the same, and the distribution of features should be similar. This way, the testing data should be representative of the data you build the model on.

While model validation is a good practice, it raises the question of whether the particular split we choose for the training, validation, and testing data has any effect on the outcomes that we are tracking. For example, perhaps the relationship between the features and the response variable is slightly different in the unseen test set that we have reserved, or in the validation set versus the training set. It is impossible to eliminate any such variability, but we can use the method of **cross-validation** to avoid placing too much faith in one particular split of the data.

Scikit-learn provides convenient functions to facilitate cross-validation analyses. These functions play a similar role to `train_test_split`, which we have already been using, although the default behavior is somewhat different. Let's get familiar with them now; first, import these two classes:

```
from sklearn.model_selection import StratifiedKFold  
from sklearn.model_selection import KFold
```

Similar to `train_test_split`, we need to specify what proportion of the dataset we would like to use for training versus testing. However, with cross-validation (specifically the **k folds cross-validation** that was implemented in the classes we just imported), rather than specifying a proportion directly we simply indicate how many folds we would like – that is, the "**k folds**." The idea here is that the data will be divided into **k** equal proportions. For example, if we specify four folds, then each fold will have 25% of the data. These folds will be the testing data in four separate instances of model training, while the remaining 75% from each fold will be used to train the model. In this procedure, each data point gets used as training data a total of $k - 1$ times, and as testing data only once.

When instantiating the class, we indicate the number of folds, whether or not to shuffle the data before splitting, and a random seed if we want repeatable results across different runs:

```
n_folds = 4  
k_folds = KFold(n_splits=n_folds, shuffle=False, random_state=1)
```

Here, we've instantiated an object with four folds and no shuffling. The way in which we use the object that is returned, which we've called **k_folds**, is by passing the features and response data that we wish to use for cross-validation, to the `.split` method of this object. This outputs an **iterator**, which means that we can loop through the output to get the different splits of training and testing data. If we took the training data from our synthetic classification problem, `X_syn_train` and `y_syn_train`, we could loop through the splits like this:

```
for train_index, test_index in k_folds.split(X_syn_train, y_syn_train):
```

The iterator has returned the row indices of `X_syn_train` and `y_syn_train`, which we can use to index the data. Inside this `for` loop, we can write code to use these indices to select data for repeatedly training and testing a model object with different subsets of the data. In this way, we can get a robust indication of the out-of-sample performance when using one particular hyperparameter value, and then repeat the whole process using another hyperparameter value. Consequently, the cross-validation loop may sit **nested** inside an outer loop over different hyperparameter values. We'll illustrate this in the following exercise.

First though, what do these splits look like? If we were to simply plot the indices from `train_index` and `test_index` as different colors, we would get something that looks like this:

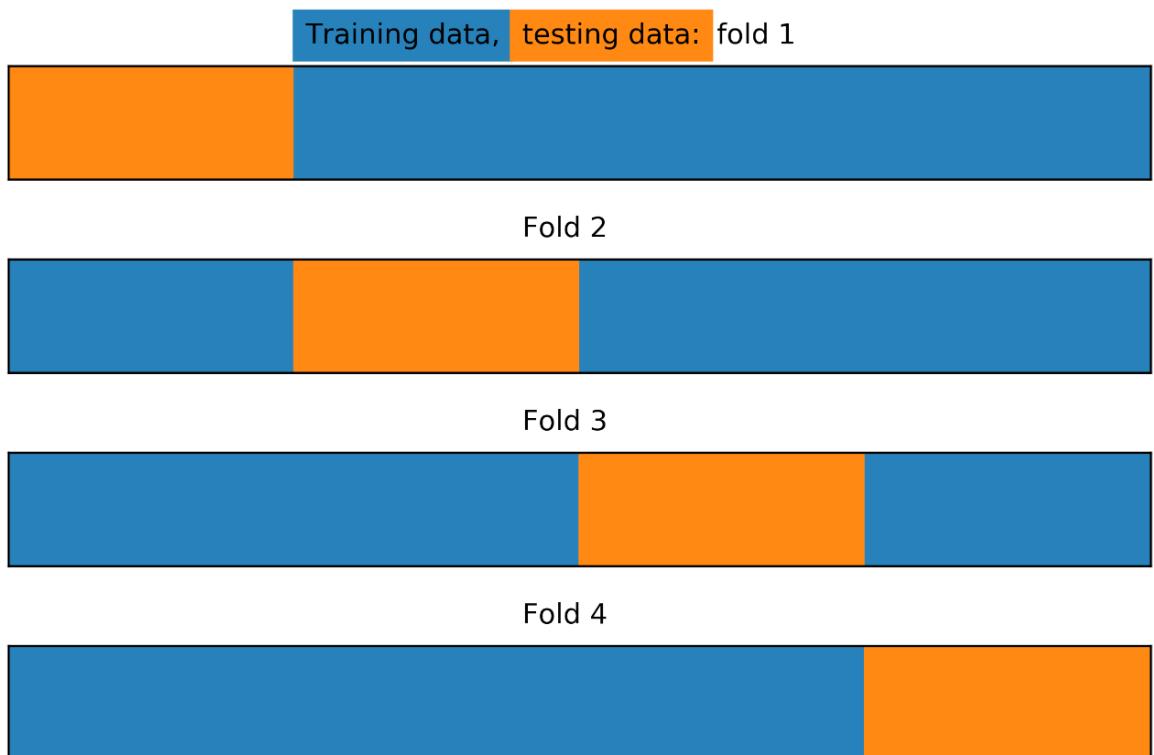


Figure 4.17: Train/test splits for k-folds with four folds and no shuffling

Here, we see that with the options we've indicated for the `KFold` class, the procedure has simply taken the first 25% of the data, according to the order of rows, as the first testing fold, then the next 25% of data for the second fold, and so on. But what if we wanted stratified folds? In other words, what if we wanted to ensure that the class fractions of the response variable were equal in every fold? While `train_test_split` allows for this option as a keyword argument, there is a separate `StratifiedKFold` class that implements this for cross-validation. While our synthetic data has a 50/50 class balance and this is unlikely to make a major difference in any cross-validation results, we can illustrate how the stratified splits will look as follows:

```
k_folds = StratifiedKFold(n_splits=n_folds, shuffle=False, random_state=1)
```

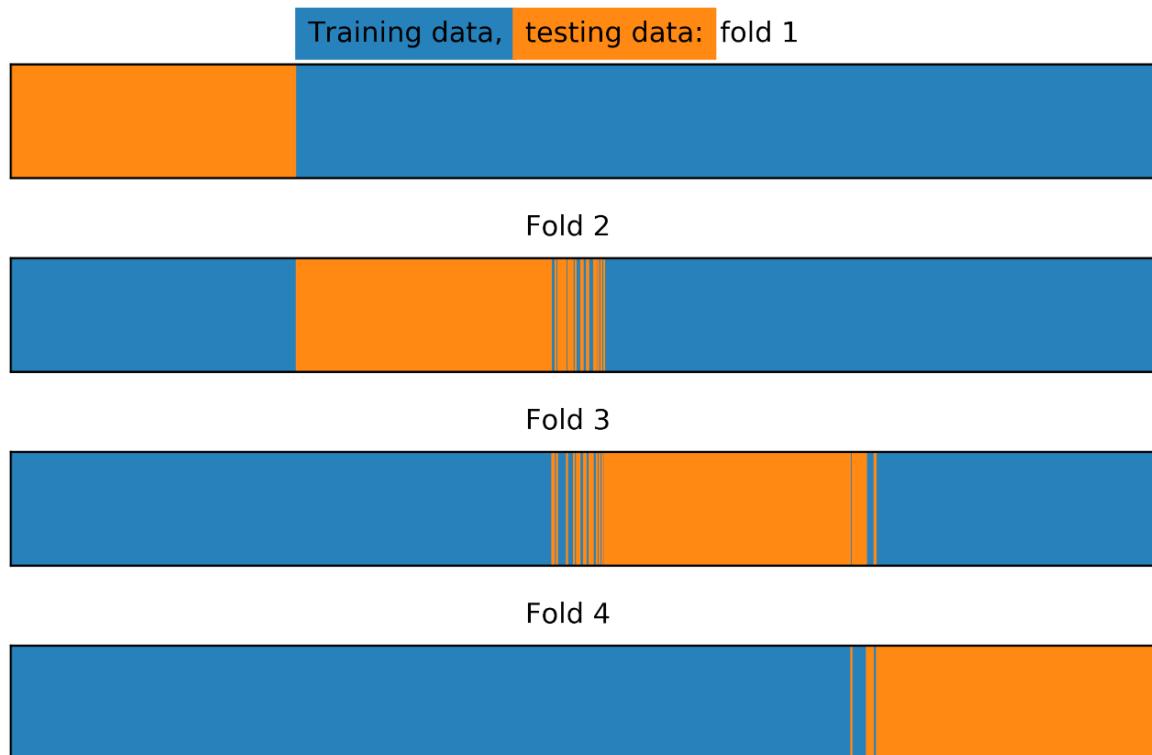


Figure 4.18: Train/test splits for stratified k-folds

In Figure 4.18 we can see that there has been some amount of "shuffling" between the different folds. The procedure has moved samples between folds as necessary, to ensure that the class fractions in each fold are equal.

Now, what if we want to shuffle the data to choose samples from throughout the range of indices for each testing fold? First, why might we want to do this? Well, with the synthetic that we've created for our problem, we can be certain that the data is in no particular order. However, in many real-world situations, the data we receive may be sorted in some way.

For instance, perhaps the rows of the data have been ordered by the date an account was created, or by some other logic. Therefore, it can be a good idea to shuffle the data before splitting. This way, any traits that might have been used for sorting can be expected to be consistent throughout the folds. Otherwise, the data in different folds may have different characteristics, possibly leading to different relationships between features and response. This can lead to a situation where model performance is uneven between the folds. In order to "mix up" the folds throughout all the row indices of a dataset, all we need to do is set the `shuffle` parameter to `True`:

```
k_folds = StratifiedKFold(n_splits=n_folds, shuffle=True, random_state=1)
```

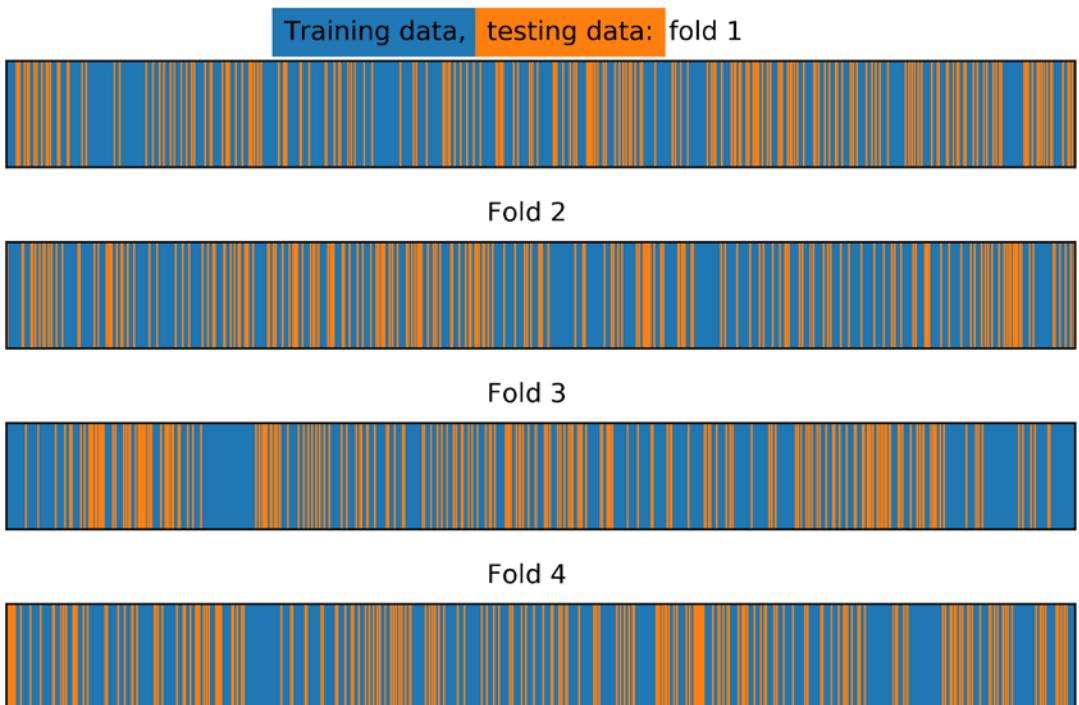


Figure 4.19: Train/test splits for stratified k-folds with shuffling

With shuffling, the testing folds are spread out randomly, and fairly evenly, across the indices of the input data.

K-folds cross-validation is a widely used method in data science. However, the choice of how many folds to use depends on the particular dataset at hand. Using a smaller number of folds means that the amount of training data in each fold will be relatively small. Therefore, this increases the chances that the model will be underfit, as models generally work better when trained on more data. It's a good idea to try a few different numbers of folds and see how the mean and the variability of the k-folds testing score changes. Common numbers of folds can range anywhere from 4 or 5 to 10.

In the event of a very small dataset, it may be necessary to use as much data as possible for training in the cross-validation folds. In these scenarios, you can use a method called **leave-one-out cross-validation (LOOCV)**. In LOOCV, the testing set for each fold consists of a single sample. In other words, there will be as many folds as there are samples in the training data. For each iteration, the model is trained on all but one sample, and a prediction is made for that sample. The accuracy, or other performance metric, can then be constructed using these predictions.

Other concerns that relate to creation of a test set, such as choosing an out-of-time test set for problems where observations from the past must be used to predict future events, also apply to cross-validation.

In exercise 17, we saw that fitting a logistic regression on our training data led to overfitting. Indeed, the testing score (ROC AUC = 0.81) was substantially lower than the training score (ROC AUC = 0.94). We had essentially used very little or no regularization by setting the regularization parameter C to a relatively large value (1,000). Now we will see what happens when we vary C through a wide range of values.

Exercise 18: Reducing Overfitting on the Synthetic Data Classification Problem

This exercise is a continuation of *Exercise 17, Generating and Modeling Synthetic Classification Data*. Here, we will use the cross-validation procedure in order to find a good value for the hyperparameter C. We will do this by using only the training data, reserving the testing data for after model building is complete. Be prepared, this is a long exercise – but it will illustrate a general procedure that you will be able to use with many different kinds of machine learning models, so it is worth the time spent here.

Perform the following steps to complete the exercise:

Note

The code and the resulting output for this exercise have been loaded in a Jupyter Notebook that can be found at <http://bit.ly/2ZAy2Pr>.

1. Vary the value of the regularization parameter, C, to have it range from $C = 1000$ to $C = 0.001$. You can use the following snippets to do this.

First, define exponents, which will be powers of 10, as follows:

```
C_val_exponents = np.linspace(3, -3, 13)
C_val_exponents
```

Here is the output of the preceding code:

```
array([ 3. ,  2.5,  2. ,  1.5,  1. ,  0.5,  0. , -0.5, -1. , -1.5, -2. ,
       -2.5, -3. ])
```

Now vary C by the powers of 10, as follows:

```
C_vals = np.float(10)**C_val_exponents
C_vals
```

Here is the output of the preceding code:

```
array([1.0000000e+03, 3.16227766e+02, 1.0000000e+02, 3.16227766e+01,
       1.0000000e+01, 3.16227766e+00, 1.0000000e+00, 3.16227766e-01,
       1.0000000e-01, 3.16227766e-02, 1.0000000e-02, 3.16227766e-03,
       1.0000000e-03])
```

It's generally a good idea to vary the regularization parameter by powers of 10, or by using a similar strategy, as training models can take substantial time, especially when using k-folds cross-validation. This gives you a good idea of how a wide range of C values impacts the bias-variance trade-off, without needing to train a very large number of models. In addition to the integer powers of 10, we also include points on the \log_{10} scale that appear half-way between.

2. Import the **roc_curve** class:

```
from sklearn.metrics import roc_curve
```

We'll continue to use the ROC AUC score for assessing, training, and testing performance. Now that we have several values of C to try and several folds (in this case four) for the cross-validation, we will want to store the training and testing scores for each fold and for each value of C.

3. Define a function that takes the `k_folds` cross-validation splitter, the array of C values (`C_vals`), the model object (`model`), and the features and response variable (`X` and `Y`, respectively) as inputs, with which to explore different amounts of regularization with k-folds cross-validation, using the following code:

```
def cross_val_C_search(k_folds, C_vals, model, X, Y):
```

Note

The function we started in this step will return the ROC AUCs and ROC curve data.

The return block will be written during a later step in the exercise. For now, you can simply write the preceding code as is, because we will be defining `k_folds`, `C_vals`, `model`, `X`, and `Y` as we progress in the exercise.

4. Create a NumPy array to hold `C_vals`, with dimensions `n_folds` by `len(C_vals)`:

```
n_folds = k_folds.n_splits
cv_train_roc_auc = np.empty((n_folds, len(C_vals)))
cv_test_roc_auc = np.empty((n_folds, len(C_vals)))
```

Next, we'll store the arrays of true and false positives rates and thresholds that go along with each of these testing ROC AUC scores, in a **list of lists**.

Note

This is a convenient way to store all this information, as a list in Python can contain any kind of data, including another list. Here, each item in the **list of lists** will be a tuple holding the arrays of TPR, FPR, and the thresholds for each of the folds, for each of the C values. This should be more obvious when we access these arrays later in order to examine them.

5. Create a list of empty lists using `[]` and `*len(C_vals)` as follows:

```
cv_test_roc = [[]]*len(C_vals)
```

Using `*len(C_vals)` implies that there should be a list of tuples of metrics (TPR, FPR, thresholds) for each value of C.

We have learned how to loop through the different folds for cross-validation in the preceding section. What we need to do now is write an outer loop, in which we will nest the cross-validation loop.

6. Create an outer loop for training and testing each of the k-folds for each value of C:

```
for c_val_counter in range(len(C_vals)):
    #Set the C value for the model object
    model.C = C_vals[c_val_counter]
    #Count folds for each value of C
    fold_counter = 0
```

We can reuse the same model object that we have already, and simply set a new value of C within each run of the loop. Inside the loop of C values, we run the cross-validation loop. We begin by yielding the training and testing data row indices for each split.

7. Obtain the training and testing indices for each fold:

```
for train_index, test_index in k_folds.split(X, Y):
```

8. Index the features and response variable to obtain the training and testing data for this fold using the following code:

```
X_cv_train, X_cv_test = X[train_index], X[test_index]
y_cv_train, y_cv_test = Y[train_index], Y[test_index]
```

The training data for the current fold is then used to train the model.

9. Fit the model on the training data, as follows:

```
model.fit(X_cv_train, y_cv_train)
```

This will effectively "reset" the model from whatever the previous coefficients and intercept were, to reflect the training on this new data.

The training and testing ROC AUC scores are then obtained, as well as the arrays of TPRs, FPRs, and thresholds that go along with the testing data.

10. Obtain the training ROC AUC score:

```
y_cv_train_predict_proba = model.predict_proba(X_cv_train)
cv_train_roc_auc[fold_counter, c_val_counter] = \
roc_auc_score(y_cv_train, y_cv_train_predict_proba[:,1])
```

11. Obtain the testing ROC AUC score:

```
y_cv_test_predict_proba = model.predict_proba(X_cv_test)
cv_test_roc_auc[fold_counter, c_val_counter] = \
roc_auc_score(y_cv_test, y_cv_test_predict_proba[:,1])
```

12. Obtain the testing ROC curves for each fold using the following code:

```
this_fold_roc = roc_curve(y_cv_test, y_cv_test_predict_proba[:,1])
cv_test_roc[c_val_counter].append(this_fold_roc)
```

We will use a fold counter to keep track of the folds that are incremented, and once outside the cross-validation loop, we print a status update to standard output. Whenever performing long computational procedures, it's a good idea to periodically print the status of the job, so that you can monitor its progress and confirm that things are still working correctly. This cross-validation procedure will likely take only a few seconds on your laptop, but for longer jobs this can be especially reassuring.

13. Increment the fold counter using the following code:

```
fold_counter += 1
```

14. Write the following code to indicate the progress of execution for each value of C:

```
print('Done with C = {}'.format(lr_syn.C))
```

15. Write the code to return the ROC AUCs and ROC curve data and finish the function:

```
return cv_train_roc_auc, cv_test_roc_auc, cv_test_roc
```

Note that we will continue to use the split into four folds that we illustrated previously, but you are encouraged to try this procedure with different numbers of folds to compare the effect.

We have covered a lot of material in the preceding steps. You may want to take a few moments to review this with your classmates in order to make sure that you understand each part. Running the function is comparatively simple. That is the beauty of a well-designed function – all the complicated parts get abstracted away, allowing you to concentrate on usage.

16. Run the function to search for the C values that we previously defined by using the model and data we were working with in the previous exercise and the following code:

```
cv_train_roc_auc, cv_test_roc_auc, cv_test_roc = \
cross_val_C_search(n_folds, C_vals, lr_syn, X_syn_train, y_syn_train)
```

When you run this code, you should see the following output populate below the code cell as the cross-validation is completed for each value of C.

```
Done with C = 1000.0
Done with C = 316.22776601683796
Done with C = 100.0
Done with C = 31.622776601683793
Done with C = 10.0
Done with C = 3.1622776601683795
Done with C = 1.0
Done with C = 0.31622776601683794
Done with C = 0.1
Done with C = 0.03162277660168379
Done with C = 0.01
Done with C = 0.0031622776601683794
Done with C = 0.001
```

So, what do the results of the cross-validation look like? There are a few ways to examine this. It is useful to look at the performance of each fold individually, so that you can see how variable the results are.

This tells you how different subsets of your data perform as test sets, leading to a general idea of the range of performance you can expect from the unseen test set. What we're interested in here is whether or not we are able to use regularization to alleviate the overfitting that we saw. We know that using $C = 1,000$ led to overfitting – we know this from comparing the training and testing scores. But what about the other C values that we've tried? A good way to visualize this will be to plot the training and testing scores on the y-axis and the values of C on the x-axis.

17. Loop over each of the folds to view their results individually by using the following code:

```
for this_fold in range(n_folds):
    plt.plot(C_val_exponents, cv_train_roc_auc[this_fold], '-o',
              color=cmap(this_fold), label='Training fold {}'.format(this_fold+1))
    plt.plot(C_val_exponents, cv_test_roc_auc[this_fold], '-x',
              color=cmap(this_fold), label='Testing fold {}'.format(this_fold+1))
    plt.ylabel('ROC AUC')
    plt.xlabel('log$_{10}$(C)')
    plt.legend(loc = [1.1, 0.2])
    plt.title('Cross validation scores for each fold')
```

You will obtain the following output:

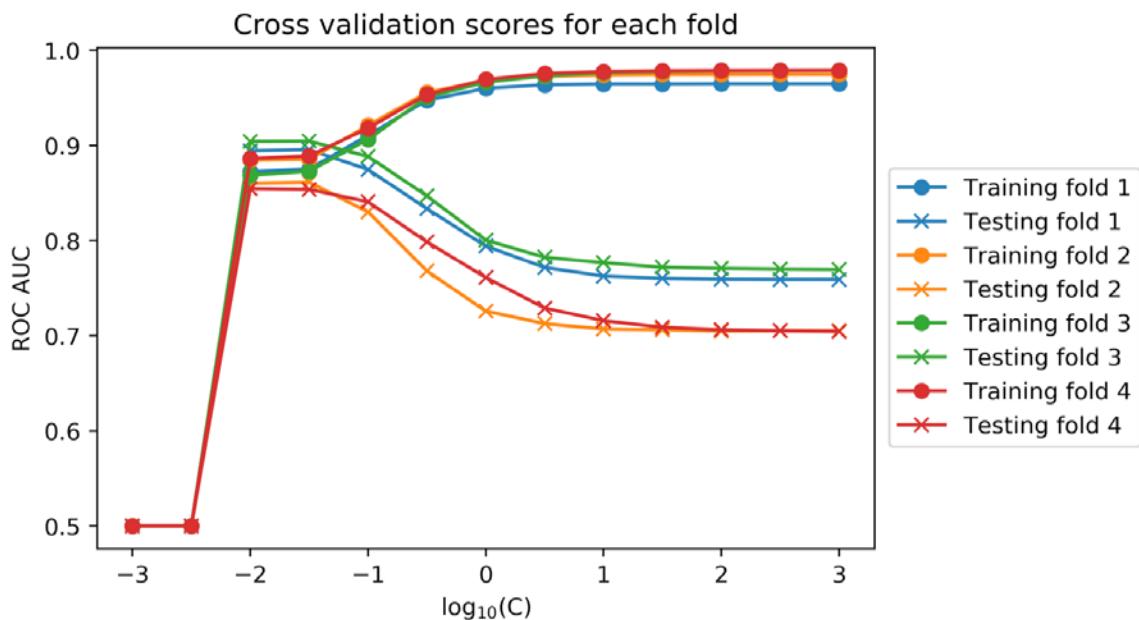


Figure 4.20: The training and testing scores for each fold and C-value

We can see that for each fold of the cross-validation, as C decreases, the training performance also decreases. However, at the same time the testing performance also increases. For some folds and values of C , the testing ROC AUC score actually exceeds that of the training data, while for others, these two metrics simply come closer together. In all cases, we can say that the C values of $10^{-1.5}$ and 10^{-2} appear to have a similar testing performance, which is substantially higher than the testing performance of $C = 10^3$. So, it appears that regularization has successfully addressed our overfitting problem.

But what about the lower values of C ? For values that are lower than $10^{-1.5}$, the ROC AUC metric suddenly drops to 0.5. As you know, this value means that the classification model is essentially useless, performing no better than a coin flip. You are encouraged to check on this later when exploring how regularization affects the coefficient values; however, this is what happens when so much L1 regularization is applied that all model coefficients shrink to 0. Obviously, such models are not useful to us.

Looking at the training and testing performance of each k-folds split is helpful in gaining insight into the variability of model performance that may be expected when the model is scored on new, unseen data. But in order to summarize the results of the k-folds procedure, a common approach is to average the performance metric over the folds, for each value of the hyperparameter being considered. We'll perform this in the next step.

18. Plot the mean of training and testing ROC AUC scores for each C value using the following code:

```
plt.plot(C_val_exponents, np.mean(cv_train_roc_auc, axis=0), '-o',
         label='Average training score')
plt.plot(C_val_exponents, np.mean(cv_test_roc_auc, axis=0), '-x',
         label='Average testing score')
plt.ylabel('ROC AUC')
plt.xlabel('log$_{10}$(C)')
plt.legend()
plt.title('Cross validation scores averaged over all folds')
```

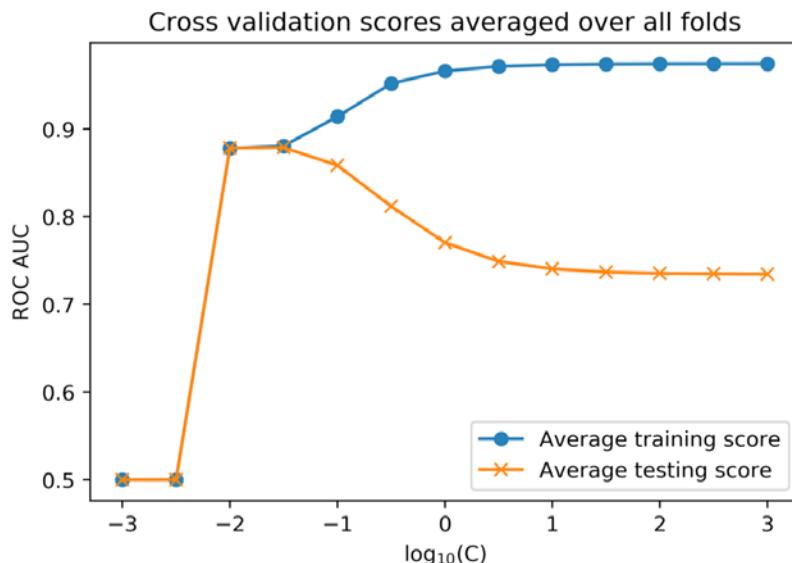


Figure 4.21: The average training and testing scores across cross-validation folds

From this plot, it's clear that $C = 10^{-1.5}$ and 10^{-2} are the best values of C. There is little or no overfitting here, as the average training and testing scores are nearly the same. You could search a finer grid of C values (that is $C = 10^{-1.1}, 10^{-1.2}$, and so on) in order to more precisely locate a C value. However, from our graph we can see that either $C = 10^{-1.5}$ or $C = 10^{-2}$ will likely be good solutions. We will move forward with $C = 10^{-1.5}$.

Examining the summary metric of ROC AUC is a good way to get a quick idea of how models will perform. However, for any real-world business application, you will often need to choose a specific threshold, which goes along with specific true and false positive rates. These will be needed to use the classifier to make the required "yes" or "no" decision, which in our case study is a prediction of whether or not an account will default. For this reason, it is useful to look at the ROC curves across the different folds of the cross-validation. To facilitate this, the preceding function has been designed to return the true and false positive rates, and thresholds, for each testing fold and value of C, in the `cv_test_roc` list of lists. First, we need to find the index of the outer list that corresponds to the C value that we've chosen, $10^{-1.5}$.

To accomplish this, we could simply look at our list of C values and count by hand, but it's safer to do this programmatically by finding the index of the non-zero element of a Boolean array as is shown in the next step.

19. Use a Boolean array to find the index where $C = 10^{-1.5}$ and convert to an integer data type with this code:

```
best_C_val_bool = C_val_exponents == -1.5  
best_C_val_bool.astype(int)
```

Here is the output of the preceding code:

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0])
```

20. Convert the integer version of the Boolean array into a single integer index using the `nonzero` function with this code:

```
best_C_val_ix = np.nonzero(best_C_val_bool.astype(int)) best_C_val_ix[0]  
[0]  
best_C_val_ix[0][0]
```

Here is the output of the preceding code:

9

We have now successfully located the C value that we wish to use.

21. Access the true and false positive rates in order to plot the ROC curves for each fold:

```
for this_fold in range(n_folds):  
    fpr = cv_test_roc[best_C_val_ix[0][0]][this_fold][0]  
    tpr = cv_test_roc[best_C_val_ix[0][0]][this_fold][1]  
    plt.plot(fpr, tpr, label='Fold {}'.format(this_fold+1))  
plt.xlabel('False positive rate')  
plt.ylabel('True positive rate')
```

```
plt.title('ROC curves for each fold at C = $10^{-1.5}$')
plt.legend()
```

You will obtain the following output:

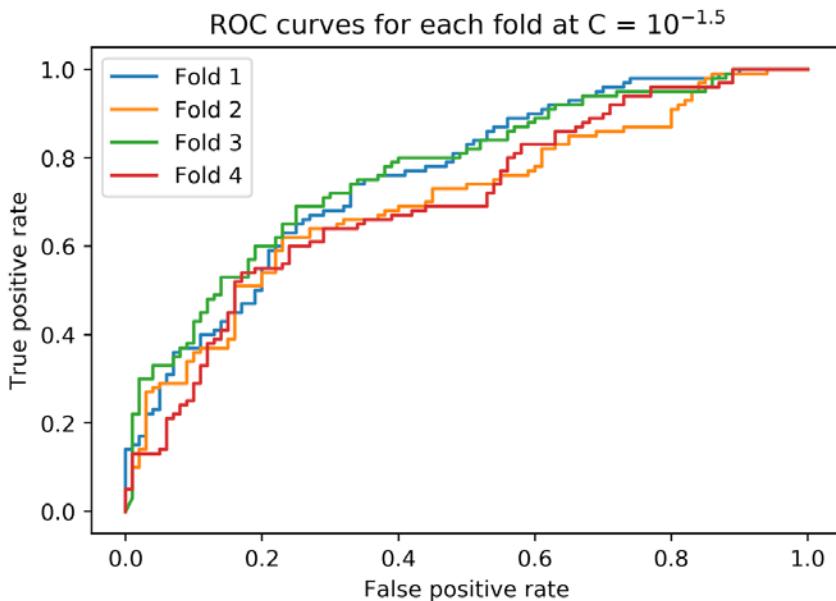


Figure 4.22: ROC curves for each fold

It appears that there is a fair amount of variability in the ROC curves. For example, if for some reason we want to limit the false positive rate to 40%, then from the plot it appears that we may be able to achieve a true positive rate of anywhere from approximately 65% to 80%. You can find the exact values by examining the arrays that we have plotted. This gives you an idea of how much variability in performance can be expected when deploying the model on new data. Generally, the more training data that is available, then the less variability there will be between the folds of cross-validation, so this could also be a sign that it will be a good idea to collect additional data, especially if the variability between training folds seems high. You also may wish to try different numbers of folds with this procedure, to see the effect on variability of results between folds.

While normally we would try other models on our synthetic data problem, such as a random forest or support vector machine, if we imagine that in cross-validation, logistic regression proved to be the best model, we would decide to make it our final choice. When the final model is selected, all the training data can be used to fit the model, using the hyperparameters chosen with cross-validation. It's best to use as much data as possible in model fitting, as models typically work better when trained on more data.

22. Train the logistic regression on all the training data from our synthetic problem and compare the training and testing scores, using the held-out test set as shown in the following steps.

Note

This is the last step in the model selection process. You should only use the unseen test set after your choice of model and hyperparameters are considered finished, otherwise it will not be "unseen."

23. Set the C value and train the model on all the training data with this code:

```
lr_syn.C = 10**(-1.5)  
lr_syn.fit(X_syn_train, y_syn_train)
```

Here is the output of the preceding code:

```
LogisticRegression(C=0.03162277660168379, class_weight=None, dual=False,  
fit_intercept=True, intercept_scaling=1, max_iter=100,  
multi_class='warn', n_jobs=None, penalty='l1', random_state=1,  
solver='liblinear', tol=0.0001, verbose=0, warm_start=False)
```

24. Obtain predicted probabilities and the ROC AUC score for the training data with this code:

```
y_syn_train_predict_proba = lr_syn.predict_proba(X_syn_train)  
roc_auc_score(y_syn_train, y_syn_train_predict_proba[:,1])
```

Here is the output of the preceding code

0.8802812499999999

25. Obtain predicted probabilities and the ROC AUC score for the testing data with this code:

```
y_syn_test_predict_proba = lr_syn.predict_proba(X_syn_test)  
roc_auc_score(y_syn_test, y_syn_test_predict_proba[:,1])
```

Here is the output of the preceding code

0.8847884788478848

Here, we can see that by using regularization, the model training and testing scores are similar, indicating that the overfitting problem has been solved. The training score is lower, since we have introduced bias into the model at the expense of variance. However, this is okay, since the testing score, which is the most important part, is higher. The out-of-sample testing score is what matters for predictive capability. You are encouraged to check that these training and testing scores are similar to those from the cross-validation procedure by printing the values from the arrays that we plotted previously; you should find that they are.

Note

In the real-world case, before delivering this model to your client for production use, you would likely train the model on all the data that you were given, including the unseen test set. This again follows the idea that the more data a model has seen, the better it is likely to perform in practice.

We know that L1 regularization works by decreasing the magnitude (that is, absolute value) of coefficients of the logistic regression. It can also set some coefficients to zero, therefore performing feature selection. In the next step, we will determine how many coefficients were set to zero.

26. Access the coefficients of the trained model and determine how many do not equal zero (`!= 0`) with this code:

```
sum((lr_syn.coef_ != 0)[0])
```

The output should be:

2

This code takes the sum of a Boolean array indicating the locations of non-zero coefficients, so it shows how many coefficients in the model did not get set to zero by L1 regularization. Only 2 of the 200 features were selected!

27. Examine the value of the intercept using this code:

```
lr_syn.intercept_
```

The output should be:

```
array([0.])
```

This shows that the intercept was regularized to 0.

In this exercise, we accomplished several goals. We used the k-folds cross-validation procedure to tune the regularization hyperparameter. We saw the power of regularization for reducing overfitting, and in the case of L1 regularization in logistic regression, selecting features.

Many machine learning algorithms offer some type of feature selection capability. Many also require the tuning of hyperparameters. The function here that loops over hyperparameters, and performs cross-validation, is a powerful concept that generalizes to other models. Scikit-learn offers functionality to make this process easier; in particular, the `sklearn.model_selection.GridSearchCV` procedure, which applies cross-validation to a grid search over hyperparameters. A **grid search** can be helpful when there are multiple hyperparameters to tune, by looking at all combinations of the ranges of different hyperparameters that you specify. A **randomized grid search** can speed up this process by randomly choosing a smaller number of combinations when an exhaustive grid search would take too long. Once you are comfortable with the concepts illustrated here, you are encouraged to streamline your workflow with convenient functions like these.

Options for Logistic Regression in Scikit-Learn

We have used and discussed most of the options that you may supply to scikit-learn when instantiating or tuning the hyperparameters of a `LogisticRegression` model class. Here, we list them all and give some general advice on their usage:

Parameter	Possible values	Notes and advice for choosing
<code>penalty</code>	string, 'l1' or 'l2'	L1 (lasso) or L2 (ridge) regularization of coefficients. L1 performs feature selection, while L2 does not. The best overall model performance should be assessed by trying both.
<code>dual</code>	bool, True, or False	This has to do with the optimization algorithm used to find coefficients. The documentation says "only implemented for l2 penalty with liblinear solver. Prefer dual = False when n_samples > n_features."
<code>tol</code>	float (decimal number)	Determines the size of the change in values for the optimization algorithm to stop. This is one way to control how long the optimization runs for, and how close to the ideal value the solution is.
<code>C</code>	float	The regularization parameter for L1 or L2 regularization. This needs to be determined using a validation set, or cross-validation.
<code>fit_intercept</code>	bool	Whether or not an intercept term should be estimated. Unless you are sure you don't need an intercept, it's probably best to have one.
<code>intercept_scaling</code>	float	Can be used to avoid regularizing the intercept, an undesirable practice, when using the liblinear solver.
<code>class_weight</code>	dictionary specifying weight for each class, string 'balanced', or None	Whether or not to weight different classes during the model training process. Otherwise all samples will be considered "equally important" when fitting the model. Can be useful for imbalanced data sets: try using 'balanced' in this case.
<code>random_state</code>	int	Seed for random number generator used by certain solver algorithms.

<code>solver</code>	<code>string('newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga')</code>	Select the type of optimization algorithm used to estimate the model parameters. See discussion in previous chapter or the documentation for the relative strengths and weaknesses of different solvers.
<code>max_iter</code>	<code>int</code>	The maximum number of iterations for the solution algorithm, which controls how close to the ideal parameters the solution is. If you get a warning that the solution algorithm did not converge, you can try increasing this.
<code>multi_class</code>	<code>string('ovr', 'multinomial', 'auto')</code>	Various strategies for multiclass classification, beyond the scope of this courseware.
<code>verbose</code>	<code>int</code>	Controls the nature of the output to the terminal, during the optimization procedure.
<code>warm_start</code>	<code>bool</code>	If re-using the same model object for multiple trainings, whether to use the previous solution as the starting point for the next optimization procedure.
<code>n_jobs</code>	<code>int or None</code>	Number of processors to use for parallel processing, in the case of 'ovr' multiclass classification.

Figure 4.23: A complete list of options for the logistic regression model in scikit-learn

If you are in doubt regarding which option to use for logistic regression, we recommend you consult the scikit-learn documentation for further guidance (https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression). Some options, such as the regularization parameter C, or the choice of L1 or L2 regularization, will need to be explored through the cross-validation process. Here, as with many choices to be made in data science, there is no universal approach that will apply to all datasets. The best way to see which options to use with a given dataset is to try several of them and see which gives the best out-of-sample performance. Cross-validation offers you a robust way to do this.

Scaling Data, Pipelines, and Interaction Features in Scikit-Learn

Scaling Data

Compared to the synthetic data we were just working with, the case study data is relatively large. If we want to use L1 regularization, then according to the official documentation (https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression), we ought to use the `saga` solver. However, this solver is not robust to unscaled datasets. So, we need to be sure to scale the data. This is also a good idea whenever doing regularization, so all the features are on the same scale and are equally penalized by the regularization process. A simple way to make sure that all the features have the same scale is to put them all through the transformation of subtracting the minimum, and dividing by the range from minimum to maximum. This transforms each feature so that it will have a minimum of 0 and a maximum of 1. To instantiate the `MinMaxScaler` scaler which does this, we can use the following code:

```
from sklearn.preprocessing import MinMaxScaler  
min_max_sc = MinMaxScaler()
```

Pipelines

Previously, we used a logistic regression model in the cross-validation loop. However, now that we're scaling data, what new considerations are there? The scaling is effectively "learned" from the minimum and maximum values of the training data. After this, a logistic regression model would be trained on data scaled by the extremes of the model training data. However, we won't know the minimum and maximum values of the new, unseen data. So, following the philosophy of making cross-validation an effective indicator of the out-of-sample performance, we need to use the minimum and maximum values of the training data in each cross-validation fold in order to scale the testing data in that fold, before making predictions on the testing data. Scikit-learn has streamlined this process for us in the form of a single tool: **Pipeline**. Our pipeline will consist of two steps: the **scaler** and the **logistic regression model**. These can both be fit on the training data, and then be used to make predictions on the testing data. These steps occur simultaneously, as if the pipeline were one entity. Here is how a **Pipeline** is instantiated:

```
from sklearn.pipeline import Pipeline
scale_lr_pipeline = Pipeline(steps=[('scaler', min_max_sc), ('model', lr)])
```

Interaction Features

Considering the case study data, what was the result of your discussion on whether a logistic regression model with all possible features would overfit or underfit? You can think about this from the perspective of rules of thumb such as the "rule of 10", and the number of features (17) versus samples (26,664) that we have. Alternatively, you can consider all the work we've done so far with this data. For instance, we've had a chance to visualize all the features and ensure they make sense. Since there are relatively few features, and we have relatively high confidence that they are high quality because of our data exploration work, we are in a different situation than with the synthetic data exercises in this chapter, where we had a large number of features about which we knew relatively little. So, it may be that overfitting will be less of an issue with our case study problem at this point, and the benefits of regularization will not be significant.

In fact, it may be that we will underfit the model using only the 17 features that came with the data. One strategy to deal with this is to engineer new features. Some simple feature engineering that we've discussed is to use interaction, or polynomial features. Polynomials may not make sense given the way that some of the data has been encoded; for example, $-1^2 = 1$, which may not be sensible for `PAY_1`. However, we may wish to try creating interaction features to capture the relationships between features. `PolynomialFeatures` can be used to create interaction features only, without polynomial features. Example code is as follows:

```
make_interactions = PolynomialFeatures(degree=2, interaction_only=True,  
include_bias=False)
```

Here, `degree` represents the degree of the polynomial features, `interaction_only` takes a Boolean value (setting it to `True` implies that only interaction features are created), and so does `include_bias`, which adds an intercept to the model (the default value is `False`, which is correct here as the logistic regression model will add an intercept).

Activity 4: Cross-Validation and Feature Engineering with the Case Study Data

In this activity, we'll apply the knowledge of cross-validation and regularization that we've learned in this chapter to the case study data. We'll perform basic feature engineering. In order to estimate parameters for the regularized logistic regression model for the case study data, which is larger in size than the synthetic data that we've worked with, we'll use the `saga` solver. In order to use this solver, and for the purpose of regularization, we'll need to `scale` our data as part of the modeling process, leading us to the use of `Pipelines` in scikit-learn. Once you have completed the activity, you should have obtained an improved cross-validation testing performance with the use of interaction features, as shown in the following figure:

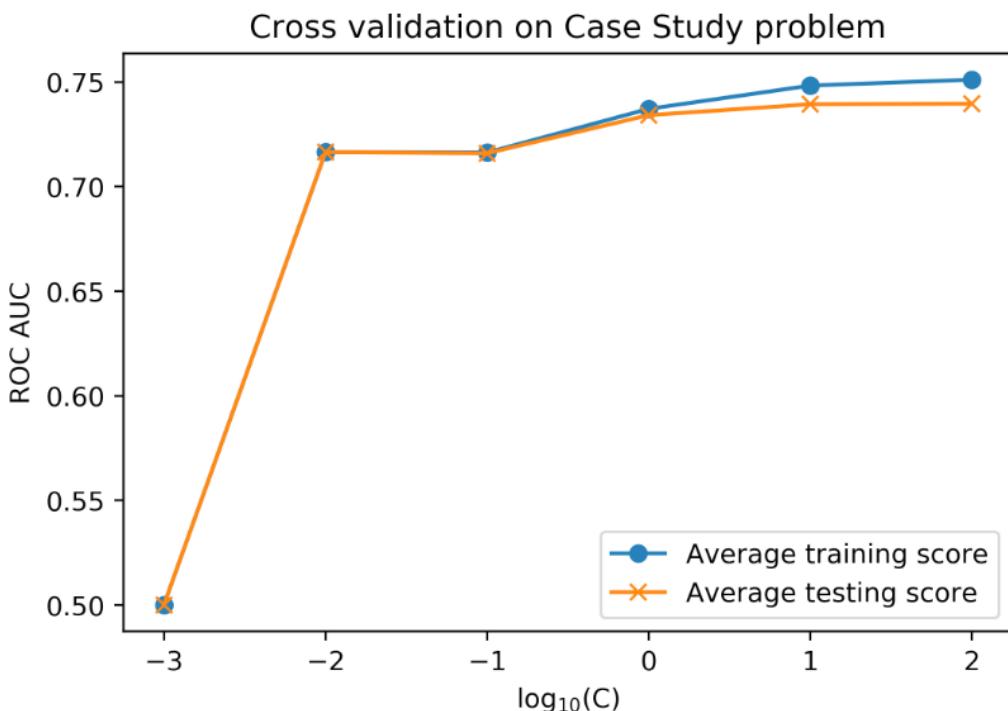


Figure 4.24: Improved model testing performance

Perform the following steps to complete the activity:

Note

The code and the resulting output for this activity have been loaded in a Jupyter Notebook and can be found at <http://bit.ly/2Z53aX4>.

1. Select the features from the DataFrame of the case study data.

You can use the list of feature names that we've already created in this chapter. But be sure not to include the response variable, which would be a very good (but entirely inappropriate) feature!

2. Make a train/test split using a random seed of 24.

We'll use this going forward and reserve this testing data as the unseen test set. This way, we can easily create separate notebooks with other modeling approaches, using the same training data.

3. Instantiate the **MinMaxScaler** to scale the data.
4. Instantiate a logistic regression model with the **saga** solver, L1 penalty, and set **max_iter** to 1,000 as we want the solver to have enough iterations to find a good solution.
5. Import the **Pipeline** class and create a **Pipeline** with the scaler and the logistic regression model, using the names '**scaler**' and '**model**' for the steps, respectively.
6. Use the **get_params** and **set_params** methods to see how to view the parameters from each stage of the pipeline and change them.
7. Create a smaller range of C values to test with cross-validation, as these models will take longer to train and test with more data than our previous exercises; we recommend $C = [10^2, 10, 1, 10^{-1}, 10^{-2}, 10^{-3}]$.
8. Make a new version of the **cross_val_C_search** function, called **cross_val_C_search_pipe**. Instead of the **model** argument, this function will take a **pipeline** argument. The changes inside the function will be to set the C value using **set_params(model__C = <value you want to test>)** on the pipeline, replacing the model with the pipeline for the **fit** and **predict_proba** methods, and accessing the C value using **pipeline.get_params()['model__C']** for the printed status update.

9. Run this function as in the previous exercise, but using the new range of C values, the pipeline you created, and the features and response variable from the training split of the case study data.

You may see warnings here, or in later steps, about the non-convergence of the solver; you could experiment with the `tol` or `max_iter` options to try and achieve convergence, although the results you obtain with `max_iter = 1000` are likely to be sufficient.

10. Plot the average training and testing ROC AUC across folds, for each C value.
11. Create interaction features for the case study data and confirm that the number of new features makes sense.
12. Repeat the cross-validation procedure and observe the model performance now.

Note that this will take substantially more time, due to the larger number of features, but it will probably take only a few minutes. So, does the average cross-validation testing performance improve with the interaction features? Is regularization useful?

Note

The solution for this activity can be found on page 339.

Summary

In this chapter, we introduced the final details of logistic regression and continued to use scikit-learn to fit logistic regression models. We gained more visibility of how the model fitting process works by learning about the concept of a cost function, which is minimized by the gradient descent procedure to estimate model parameters during model fitting.

We also learned of the need for regularization, by introducing the concepts of underfitting and overfitting. In order to reduce overfitting, we saw how to adjust the cost function to regularize the coefficients of a logistic regression model using an L1 or L2 penalty. We used cross-validation to select the amount of regularization, by tuning the regularization hyperparameter. To reduce underfitting, we gained experience with a simple feature engineering technique by creating interaction features for the case study data.

We are now familiar with some of the most important concepts in machine learning. We have, so far, only used a very basic classification model, that is, logistic regression. However, as you increase your toolbox of models that you know how to use, you will find that the concepts of overfitting, underfitting, the bias-variance trade-off, and hyperparameter tuning will come up again and again. These ideas, as well as convenient scikit-learn implementations of the cross-validation functions that we wrote in this chapter, will help us through our exploration of more advanced prediction methods.

In the next chapter, we will learn about decision trees, an entirely different type of predictive model than logistic regression, and the random forests that are based on them. However, we will use the same concepts of cross-validation and hyperparameter search to tune these models, that we learned here.

5

Decision Trees and Random Forests

Learning Objectives

By the end of this chapter, you will be able to:

- Train a decision tree model in scikit-learn
- Use Graphviz to visualize a trained decision tree model
- Formulate the cost functions used to split nodes in a decision tree
- Perform a hyperparameter grid search using cross-validation with scikit-learn functions
- Train a random forest model in scikit-learn
- Evaluate the most important features in a random forest model

This chapter introduces decision trees and random forests in scikit-learn in addition to describing the method to perform hyperparameter grid search.

Introduction

In the last two chapters, we have gained a thorough understanding of the workings of logistic regression. We have also gotten a lot of experience with using the scikit-learn package in Python to create logistic regression models.

In this chapter, we will introduce a powerful type of predictive model that takes a completely different approach from the logistic regression model: **decision trees**. The concept of using a tree process to make decisions is simple, and therefore, decision tree models are easy to interpret. However, a common criticism of decision trees is that they overfit the training data. In order to remedy this issue, researchers have developed **ensemble methods**, such as **random forests**, that combine many decision trees to work together and make better predictions than any individual tree could.

We will see that decision trees and random forests can improve the quality of our predictive modeling of the case study data beyond what we achieved so far with logistic regression.

Decision trees

Decision trees and the machine learning models that are based on them, in particular **random forests** and **gradient boosted trees**, are fundamentally different types of models than generalized linear models, such as logistic regression. GLMs are rooted in the theories of classical statistics, which have a long history. The mathematics behind linear regression were originally developed at the beginning of the 19th century, by Legendre and Gauss. Because of this, the normal distribution is also called the Gaussian.

In contrast, while the idea of using a tree process to make decisions is relatively simple, the popularity of decision trees as mathematical models has come about more recently. The mathematical procedures that we currently use for formulating decision trees in the context of predictive modeling were published in the 1980s. The reason for this more recent development is that the methods used to grow decision trees rely on computational power – that is, the ability to crunch a lot of numbers quickly. We take such capabilities for granted nowadays, but they weren't widely available until more recently in the history of mathematics.

So, what is meant by a decision tree? We can illustrate the basic concept using a practical example. Imagine that you are considering whether or not to venture outdoors on a certain day. The only information you will base your decision on involves the weather and, in particular, whether the sun is shining and how warm it is. If it is sunny, your tolerance for cool temperatures is increased, and you will go outside if the temperature is at least 10 °C.

However, if it's cloudy, you require somewhat warmer temperatures and will only go outside if the temperature is 15 °C or more. Your decision-making process could be represented by the following tree:

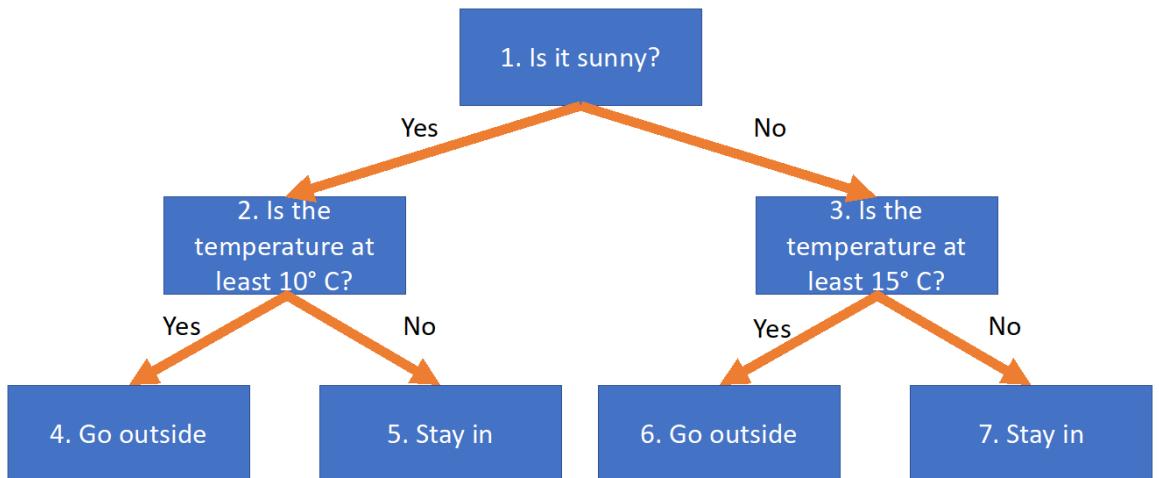


Figure 5.1: A decision tree for deciding whether to go outside given the weather

As you can see, decision trees have an intuitive structure and mimic the way that logical decisions might be made by humans. Therefore, they are a highly **interpretable** type of mathematical model, which can be a particularly desirable property depending on the audience. For example, the client for a data science project may be especially interested in a clear understanding of how a model works. Decision trees are a good way to deliver on this requirement, as long as their performance is sufficient.

The Terminology of Decision Trees and Connections to Machine Learning

Looking at the tree in Figure 5.1, we can begin to become familiar with some of the terminology of decision trees. Because there are two levels of decisions being made, based on cloud conditions at the first level, and temperature at the second level, we say that this decision tree has a **depth** of two. Here, both **nodes** at the second level are temperature-based decisions, but the kinds of decisions could be different within a level; for example, if we were to base our decision on whether or not it was raining, in the case that it was not sunny.

In the context of machine learning, the quantities that are used to make decisions at the nodes (in other words, to **split** the nodes) are the features. The features in the example in *Figure 5.1* are a binary categorical feature for whether it's sunny, and a continuous feature of temperature. While we have only illustrated each feature being used once in a given branch of the tree, the same feature could be used multiple times in a branch. For example, if we were to go outside on a sunny day of at least 10 °C, but not if it were more than 40 °C – that's too hot! In this case, node 4 of *Figure 5.1* would be split on the condition "Is the temperature at least 40 °C?", where "stay in" is the outcome if the answer is "yes", but "go outside" is the outcome if the answer is "no", meaning that the temperature is between 10 °C and 40 °C. Decision trees are, therefore, able to capture non-linear effects of the features, as opposed to a linear relationship which would assume that the hotter it was, the more likely we would be to go outside.

Consider the way that trees are typically represented, such as in *Figure 5.1*. The branches grow downward based on the binary decisions that can split the nodes into two more nodes. These binary decisions can be thought of as "if, then" rules. That is, if a certain criterion is met, do this, otherwise, do something else. The decision being made in our example tree is analogous to the concept of the response variable in machine learning. If we made a decision tree for the case study problem of credit default, the decisions would instead be predictions of the binary response values, which are "this account defaults" or "this account doesn't default". A tree that answers a binary yes/no type of question is a **classification tree**. However, decision trees are quite versatile and can also be used for multiclass classification and regression.

The terminal nodes at the bottom of the tree are called **leaves** or leaf nodes. In our example, the leaves are the final decisions of whether to go outside or stay in. There are four leaves on our tree, although you can imagine that if the tree only had a depth of one, where we made our decision based only on cloud conditions, there would be two leaves; and nodes 2 and 3 in *Figure 5.1* would be leaf nodes with "go outside" and "stay in" as the decisions, for example.

In our example, every node at every level before the final level is split. This is not strictly necessary as you may go outside on any sunny day, regardless of the temperature. In this case, node 2 will not be split, so this branch of the tree will end on the first level with a "yes" decision. Your decision on cloudy days, however, may involve temperature, meaning this branch can extend to a further level. But, in the case that every node before the final level is split, consider how quickly the number of leaves grows with the number of levels.

Think about what would happen if we grew the decision tree in *Figure 5.1* down through an additional level, perhaps with a wind speed feature, to factor in a wind chill for the four combinations of cloud conditions and temperature. Each of the four nodes that are now leaves, nodes numbered from four to seven in *Figure 5.1*, would be split into two more leaf nodes, based on wind speed in each case. Then, there would be $4 \times 2 = 8$ leaf nodes. In general, it should be clear that in a tree with n levels, where every node before the final level is split, there will be 2^n leaf nodes. This is important to bear in mind as the **maximum depth** is one of the hyperparameters that you can set for a decision tree classifier in scikit-learn. We'll now explore this in the following exercise.

Exercise 19: A Decision Tree in scikit-learn

In this exercise, we will use the case study data to grow a decision tree, where we specify the maximum depth. We'll also use some handy functionality to visualize the decision tree. In order to do this, you'll need to run `conda install graphviz` and `conda install python-graphviz` at the command line to install the Python interface for Graphviz. Perform the following steps to complete the exercise:

Note

For Exercises 19 to 21 and Activity 5, the code and the resulting output have been loaded in a Jupyter Notebook that can be found at <http://bit.ly/2GI7fjB>. You can scroll to the appropriate section within the Jupyter Notebook to locate the exercise or activity of choice.

1. Start a new notebook for this chapter. To begin with, load all the packages that we've been using, and an additional one, `graphviz`, so that we can visualize decision trees:

```
import numpy as np #numerical computation
import pandas as pd #data wrangling
import matplotlib.pyplot as plt #plotting package
#Next line helps with rendering plots
%matplotlib inline
import matplotlib as mpl #add'l plotting functionality
mpl.rcParams['figure.dpi'] = 400 #high res figures
import graphviz #to visualize decision trees
```

2. Load the cleaned case study data:

```
df = pd.read_csv('..../Data/Chapter_1_cleaned_data.csv')
```

Note

The location of the cleaned data may be different depending on where you saved it.

3. Get a list of column names of the **DataFrame**:

```
features_response = df.columns.tolist()
```

4. Make a list of columns to remove that aren't features or the response variable:

```
items_to_remove = ['ID', 'SEX', 'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5',
'PAY_6',
'EDUCATION_CAT', 'graduate school', 'high school',
'none',
'others', 'university']
```

5. Use a list comprehension to remove these column names from our list of features and the response variable:

```
features_response = [item for item in features_response if item not in
items_to_remove]
features_response
```

This should output the list of features and the response variable:

```
['LIMIT_BAL',
'EDUCATION',
'MARRIAGE',
'AGE',
'PAY_1',
'BILL_AMT1',
'BILL_AMT2',
'BILL_AMT3',
'BILL_AMT4',
'BILL_AMT5',
'BILL_AMT6',
'PAY_AMT1',
'PAY_AMT2',
'PAY_AMT3',
'PAY_AMT4',
'PAY_AMT5',
'PAY_AMT6',
'default payment next month']
```

Figure 5.2: A list of the feature and response variable names

Now the features are prepared for our usage. Next, we will make some imports from scikit-learn. We want to make a train/test split, which we are already familiar with. We also want to import the decision tree class.

6. Run this code to make imports from scikit-learn:

```
from sklearn.model_selection import train_test_split  
from sklearn import tree
```

The **tree** library is the library of decision tree-related classes in scikit-learn.

7. Split the data into training and testing, using the same random seed we have throughout the book for the train/test split:

```
X_train, X_test, y_train, y_test = \  
train_test_split(df[features_response[:-1]].values, df['default payment  
next month'].values,  
test_size=0.2, random_state=24)
```

Here, we use all but the last element of the list to get the names of the features, but not the response variable: **features_response[:-1]**. We use this to select columns from the **DataFrame**, and then retrieve their values using the **.values** method. We also do something similar for the response variable but directly specify the column name. In making the train/test split, we've used the same random seed as in previous work, and the same split size. This way, we can directly compare the work we will do in this chapter, with previous results. In particular, we have reserved the same "unseen test set" from the model development process so far.

Now we are ready to instantiate the decision tree class.

8. Instantiate the decision tree class by specifying the **max_depth** parameter to be 2:

```
dt = tree.DecisionTreeClassifier(max_depth=2)
```

We have used the **DecisionTreeClassifier** class because we have a classification problem. Since we specified **max_depth=2**, when we grow the decision tree using the case study data the tree will grow to a depth of at most 2. Let's now train this model.

9. Use this code to fit the decision tree model and grow the tree:

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

This should display the following output:

```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=2,
                      max_features=None, max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, presort=False, random_state=None,
                      splitter='best')
```

Figure 5.3: The output of fitting a decision tree classifier

In the output from model fitting, we can see all the options that are possible to set with a decision tree, most of which we have left at their default settings – we will discuss all of these shortly. For now, we have fit this decision tree model, so we can use the **graphviz** package to display a graphical representation of the tree.

10. Export the trained model in a format that can be read by the **graphviz** package using this code:

```
dot_data = tree.export_graphviz(dt, out_file=None, filled=True,
                                rounded=True, feature_names=features_
                                response[:-1],
                                proportion=True, class_names=['Not
                                defaulted', 'Defaulted'])
```

Here, we've specified a number of options to the **.export_graphviz** method. First, we need to say which trained model we'd like to graph, which we've got in the **dt** object. Next, we say we don't want an output file: **out_file=None**. Instead, we provide the **dot_data** variable to hold the output of this method. The rest of the options are used as follows:

filled=True: Each node will be filled with a color.

rounded=True: The nodes will appear with rounded edges as opposed to rectangles.

feature_names=features_response[:-1]: The names of the features from our list will be used as opposed to generic names such as **X[0]**.

proportion=True: The proportion of samples in each node will be displayed (we'll discuss this more later).

class_names=['Not defaulted', 'Defaulted']: The name of the predicted class will be displayed for each node.

What is the output of this method?

If you examine the contents of `dot_data`, you will see that it is a long text string. The `graphviz` package can interpret this text string to create a visualization.

11. Use the `.Source` method of the `graphviz` package to create an image from `dot_data` and display it:

```
graph = graphviz.Source(dot_data)
graph
```

The output should look like this:

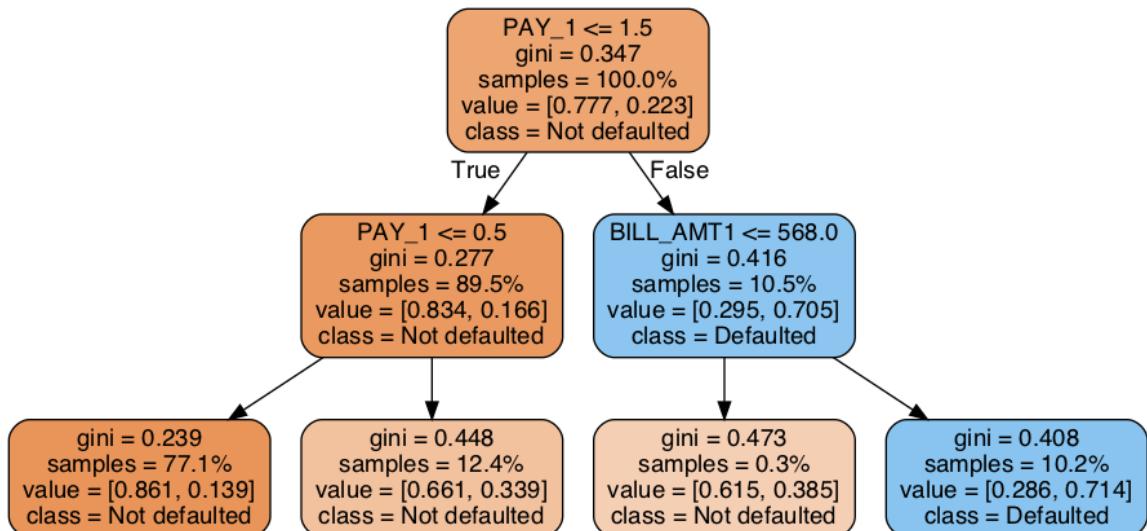


Figure 5.4: A decision tree plot from graphviz

The graphical representation of the decision tree in *Figure 5.4* should be rendered directly in your Jupyter notebook.

Note

Alternatively, you could save the output of `.export_graphviz` to disk by providing a file path to the `out_file` keyword argument. To turn this output file into an image file, for example, a `.png` file that you could use in a presentation, you could run this code at the command line, substituting in the filenames as appropriate: `$ dot -Tpng <exported_file_name> -o <image_file_name_you_want>.png`

For further details on the options to `.export_graphviz` you should consult the scikit-learn documentation (https://scikit-learn.org/stable/modules/generated/sklearn.tree.export_graphviz.html).

The visualization in *Figure 5.4* contains a lot of information about how the decision tree was trained, and how it can be used to make predictions. We will discuss the training process in more detail later, but suffice to say that training a decision tree works by starting with all the training samples in the initial node at the top of the tree, and then splitting these into two groups based on a **threshold** in one of the features. The cut point is represented by a Boolean condition in the top `PAY_1 <= 1.5` node.

All the samples where the value of the `PAY_1` feature is less than or equal to the cut point of 1.5, will be represented as `True` under the Boolean condition. As shown in *Figure 5.4*, these samples get sorted into the left side of the tree, following the arrow that says "True" next to it.

As you can see in the graph, each node that is split contains the splitting criteria on the first line of text. The next line relates to "gini", which we will discuss shortly.

On the next line after the gini line, there is information about the proportion of samples in each node. In the top node, we are starting with all the samples ("samples = 100.0%"). Following the first split, 89.5% of the samples get sorted into the node on the left, while the remaining 10.5% go into the node on the right. This information is shown directly in the visualization and reflects how the training data was used to create the tree. Let's confirm this by examining the training data.

12. To confirm the proportion of training samples where the **PAY_1** feature is less than or equal to 1.5, first identify the index of this feature in the list of **features_response[:-1]** feature names:

```
features_response[:-1].index('PAY_1')
```

This code should output the following:

4

Figure 5.5: Index of the **PAY_1** feature in the list of feature names

13. Now observe the shape of the training data:

```
X_train.shape
```

This should give you the following output:

(21331, 17)

Figure 5.6: The shape of the training data

To confirm the fraction of samples after the first split of the decision tree, we need to know the proportion of samples, where the **PAY_1** feature meets the Boolean condition, that was used to make this split. To do this, we can use the index of the **PAY_1** feature in the training data, corresponding to the index in the list of feature names, and the number of samples in the training data, which is the number of rows we observed from `.shape`.

14. Use this code to confirm the proportion of samples after the first split of the decision tree:

```
sum(X_train[:, 4] <= 1.5)/X_train.shape[0]
```

The output should be as follows:

0.8946134733486475

Figure 5.7: The proportion of training samples where **PAY_1** ≥ 1.5

By applying a logical condition to the column of the training data corresponding to the **PAY_1** feature, and then taking the sum of this, we calculated the number of samples meeting this condition. Then, by dividing by the total number of samples, we converted this to a proportion. We can see that the proportion we directly calculated from the training data is equal to the proportion displayed in the left node after the first split in *Figure 5.4*.

After the first split, the samples contained in each of the two nodes on the first level are split again. As further splits are made beyond the first split, smaller and smaller proportions of the training data will be assigned to any given node in the subsequent levels of a branch, as can be seen in *Figure 5.4*.

Now we want to interpret the remaining lines of text in the nodes in *Figure 5.4*. The lines starting with "value" give the class fractions of the response variable for the samples contained in each node. For example, in the top node, we see "value = [0.777, 0.223]". This is simply the class fraction for the overall training set, which you can confirm in the following step.

15. Calculate the class fraction in the training set with this code:

```
np.mean(y_train)
```

The output should be as follows:

0.223102526838873

Figure 5.8: The class fraction of positive samples in the training data

This is equal to the second member of the pair of numbers following "value" in the top node; the first number is just one minus this, in other words, the fraction of negative training samples. In each subsequent node, the class fraction of the samples that are contained in just that node are displayed. The class fractions are also how the nodes are colored: those with a higher proportion of the negative class than the positive class are orange, with darker orange signifying higher proportions, while those with a higher proportion of the positive class have a similar scheme using a blue color.

Finally, the line starting with "class" indicates how the decision tree will make predictions from a given node, if that node were a leaf node. Decision trees for classification make predictions by determining which leaf node a sample will be sorted in to, given the values of the features, and then predicting the class of the majority of the training samples in that leaf node. This strategy means that the class proportions in each node are the necessary information that is needed to make a prediction.

For example, if we've made no splits and we are forced to make a prediction knowing nothing but the class fractions for the overall training data, we will simply choose the majority class. Since most people don't default, the class on the top node is "Not defaulted." However, the class fractions in the nodes of deeper levels are different, leading to different predictions. We'll discuss the training process in the following section.

Importance of `max_depth`

Recall that the only hyperparameter we specified in this exercise was `max_depth`, that is, the maximum depth to which the decision tree can be grown during the model training process. It turns out that this is one of the most important hyperparameters. Without placing a limit on the depth, the tree will be grown until one of the other limitations, specified by other hyperparameters, takes effect. This can lead to very deep trees, with very many nodes. For example, consider how many leaf nodes there could be in a tree with a depth of 20. This would be 2^{20} leaf nodes, which is over 1 million! Do we even have 1 million training samples to sort into all these nodes? In this case, we do not. It would clearly be impossible to grow such a tree, with every node before the final level being split, using this training data. However, if we remove the `max_depth` limit and rerun the model training of this exercise, observe the effect:



Figure 5.9: A portion of the decision tree grown with no maximum depth

Here, we have shown a portion of the decision tree that is grown with the default options, which include `max_depth=None`, meaning no limitation on the depth of the tree. The entire tree is about twice as wide as the portion shown here. There are so many nodes that they only appear as very small orange or blue patches; the exact interpretation of each node is not important as we are just trying to illustrate how large trees can potentially be. It should be clear that without hyperparameters to govern the tree-growing process, extremely large and complex trees may result.

Training Decision Trees: Node Impurity

So far, we have treated the decision tree training process as a black box. At this point, you should have an understanding of how a decision tree makes predictions using features, and the class fractions of training samples in the leaf nodes.

But how are the splits decided during the training process?

Given that the method of prediction is to take the majority class of a leaf node, it makes intuitive sense that we'd like to find leaf nodes that are primarily from one class or the other. In the perfect case, the training data can be split so that every leaf node contains entirely positive or entirely negative samples. Then, we will have a high level of confidence that a new sample, once sorted into one of these nodes, will be either positive or negative. In practice, this rarely, if ever, happens. However, this illustrates the goal of training decision trees – that is, to make splits so that the next two nodes after the split have a higher **purity**, or, in other words, are closer to containing either only positive or only negative samples.

In practice, decision trees are actually trained using the inverse of purity, or **node impurity**. This is some measure of how far the node is from having 100% of the training samples belonging to one class and is analogous to the concept of a cost function, which signifies how far a given solution is from a theoretical perfect solution. The most intuitive concept of node impurity is the misclassification rate. Adopting a widely-used notation (for example, <https://scikit-learn.org/stable/modules/tree.html>) for the proportion of samples in each node belonging to a certain class, we can define p_{mk} as the proportion of samples belonging to the k^{th} class in the m^{th} node. In a binary classification problem, there are only two classes: $k = 0$ and $k = 1$. For a given node m , the **misclassification rate** is simply the proportion of the less common class in that node, since all these samples will be misclassified when the majority class in that node is taken as the prediction.

Let's visualize the misclassification rate as a way to start thinking about how decision trees are trained. Programmatically, we consider possible class fractions, p_{m0} , between 0.01 and 0.99 of the negative class, $k = 0$, in a node, m , using NumPy's **linspace** function:

```
pm0 = np.linspace(0.01, 0.99, 99)
```

Then, the fraction of the positive class for this node is simply the rest of the samples:

$$p_{m1} = 1 - p_{m0}$$

Figure 5.10: Equation to calculate the fraction of positive class for node m0

Now, the misclassification rate for this node will be whatever the smaller class fraction is, between **pm0** and **pm1**. We can find the smaller of the corresponding elements between two arrays with the same shape in NumPy by using the minimum function:

```
misclassification_rate = np.minimum(pm0, pm1)
```

What does the misclassification rate look like plotted against the possible class fractions of the negative class?

We can plot this using the following code:

```
mpl.rcParams['figure.dpi'] = 400
plt.plot(pm0, misclassification_rate, label='Misclassification rate')
plt.xlabel('$p_{m0}$')
plt.legend()
```

You should obtain this graph:

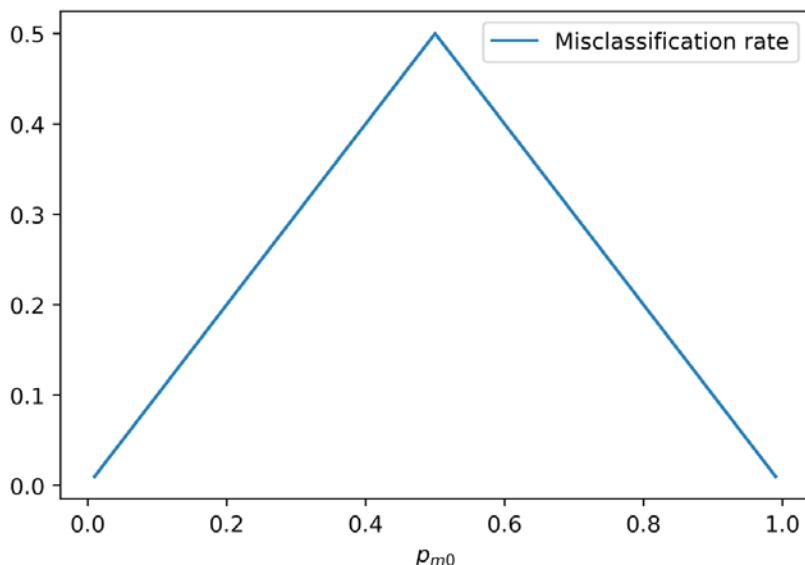


Figure 5.11: The misclassification rate for a node

Now, it's clear that the closer the class fraction of the negative class, p_{m0} , is to 0 or 1, the lower the misclassification rate will be. How is this information used when growing decision trees? Consider the process that might be followed.

Every time a node is split when growing a decision tree, two new nodes are created. Since the prediction from either of these new nodes is simply the majority class, an important goal will be to reduce the misclassification rate. Therefore, we will want to find a feature, from all the possible features, and a value of this feature at which to make a cut point, so that the misclassification rate in the two new nodes will be as low as possible when averaging over all the classes. This is very close to the actual process that is used to train decision trees.

Continuing for the moment with the idea of minimizing the misclassification rate, the decision tree training algorithm goes about node splitting by considering all the features. Although, the algorithm may possibly only consider a randomly-selected subset if you set the `max_features` hyperparameter to anything less than the total number of features. We'll discuss possible reasons for doing this later. In either case, the algorithm then considers each possible threshold for every candidate feature and chooses the one that results in the lowest impurity, calculated as the average impurity across the two possible new nodes, weighted by the number of samples in each node. The node splitting process is shown in Figure 5.12. This process is repeated until a stopping criterion of the tree, such as `max_depth`, is reached:

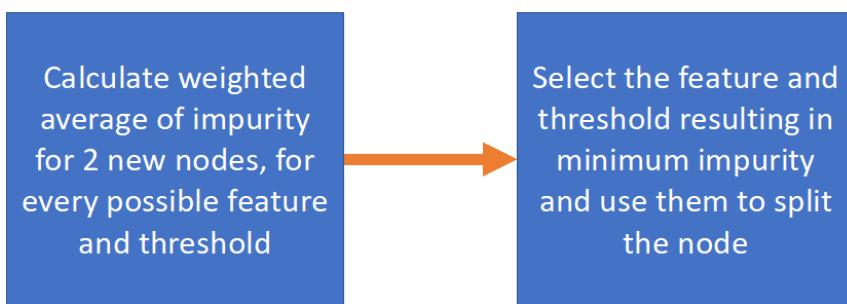


Figure 5.12: How to select a feature and threshold in order to split a node

While the misclassification rate is an intuitive measure of impurity, it happens that there are better measures that can be used to find splits during the model training process. The two options that are available in scikit-learn for the impurity calculation, which you can specify with the `criterion` keyword argument, are the **Gini impurity** and the **cross-entropy** options. Here, we will describe these mathematically and show how they compare with the misclassification rate.

The Gini impurity is calculated for a node m using the following formula:

$$\text{Gini} = \sum_k p_{mk} (1 - p_{mk})$$

Figure 5.13: Equation to calculate Gini impurity

Here, the summation is taken over all classes. In the case of a binary classification problem, there are only two classes, and we can write this programmatically as follows:

$$\text{gini} = (\text{pm0} * (1 - \text{pm0})) + (\text{pm1} * (1 - \text{pm1}))$$

Cross entropy is calculated using this formula:

$$\text{cross entropy} = - \sum_k p_{mk} \log(p_{mk})$$

Figure 5.14: Equation to calculate cross entropy

Using this code, we can calculate the cross entropy:

$$\text{cross_ent} = -1 * ((\text{pm0} * \text{np.log}(\text{pm0})) + (\text{pm1} * \text{np.log}(\text{pm1})))$$

In order to add the Gini impurity and cross entropy to our plot of misclassification rate and see how they compare, we just need to include the following lines of code, after we plot the misclassification rate:

```
plt.plot(pm0, gini, label='Gini impurity')
plt.plot(pm0, cross_ent, label='Cross entropy')
```

The final plot should appear as follows:

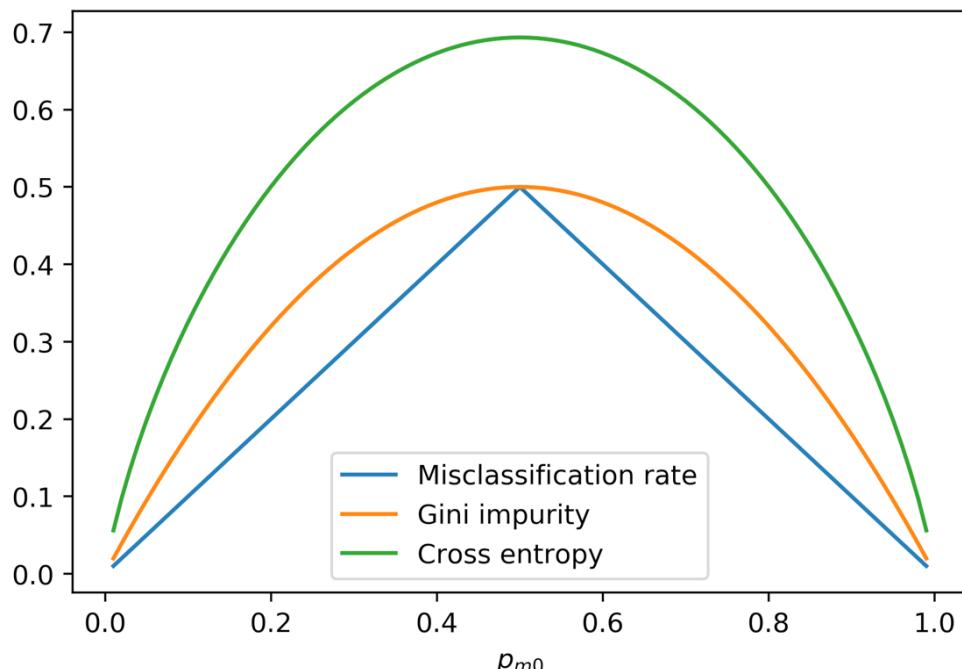


Figure 5.15: The misclassification rate, Gini impurity, and cross entropy

Like the misclassification rate, both the Gini impurity and the cross entropy are highest when the class fractions are equal at 0.5, and they decrease as the node becomes purer – in other words, when they contain a higher proportion of just one of the classes.

However, the Gini impurity is somewhat steeper than the misclassification rate in certain regions of the class fraction, which enables it to more effectively find the best split. Cross-entropy looks yet steeper. So, which one is better for your work? This is the kind of question that does not have a concrete answer across all datasets. You should consider both impurity metrics in a cross-validation search for hyperparameters in order to determine the appropriate one. Note that in scikit-learn, Gini impurity can be specified with the `criterion` argument using the '`gini`' string, while cross entropy is just referred to as '`entropy`'.

Features Used for the First splits: Connections to Univariate Feature Selection and Interactions

We can begin to get an impression of how important various features are to decision tree models, based on the small tree shown in Figure 5.4. Notice that `PAY_1` was the feature chosen for the first split. This means that it was the best feature in terms of decreasing node impurity on the node containing all of the training samples. Recalling our experience with univariate feature selection in *Chapter 3, Details of Logistic Regression and Feature Exploration*, `PAY_1` was the top-selected feature from the `F-test`. So, the appearance of this feature in the first split of the decision tree makes sense given our previous analysis.

In the second level of the tree, there is another split on `PAY_1`, as well as a split on `BILL_AMT_1`. `BILL_AMT_1` was not listed among the top features in univariate feature selection. However, it may be that there is an important interaction between `BILL_AMT_1` and `PAY_1`, which could not be picked up by univariate methods. In particular, from the splits chosen by the decision tree, it seems that those accounts with both a value of 2 or greater for `PAY_1`, and a `BILL_AMT_1` of greater than 568, are especially at risk of default. This combined effect of `PAY_1` and `BILL_AMT_1` is an interaction and may also be why we were able to improve logistic regression performance by including interaction terms in the activity of the previous chapter.

Training Decision Trees: A Greedy Algorithm

There is no guarantee that a decision tree trained by the process described previously will be the best possible decision tree for finding leaf nodes with the lowest impurity. This is because the algorithm used to train decision trees is what is called a greedy algorithm. In this context, this means that, at each opportunity to split a node, the algorithm is looking for the best possible split at that point in time, without any regard to the fact that the opportunities for later splits are being affected.

For example, consider the following hypothetical scenario: the best initial split for the training data of the case study involves **PAY_1**, as we've seen in Figure 5.4. But what if we instead split on **BILL_AMT_1**, and then make subsequent splits on **PAY_1** in the next level. Will this mean that the impurity of the leaf nodes will be lower? Even though the initial split on **BILL_AMT_1** is not the best one available at first, the end result will be better if it is done this way. The algorithm has no way of finding solutions like this, since it only considers the best possible split at each node. Just to be clear, this is only a hypothetical scenario and we have no reason to suspect this of the case study data.

The reason why we still use the algorithm that we previously described is because it takes substantially longer to consider all possible splits in a way that enables finding the truly optimal tree. Despite this shortcoming of the decision tree training process, there are methods that you can use to reduce the possible harmful effects of the greedy algorithm. Instead of searching for the best split at each node, the **splitter** keyword argument to the decision tree class can be specified as **random** in order to choose a random feature to make a split on. However, the default is **best**, which searches all features for the best split. Another option, which we've already discussed, is to limit the number of features using the **max_features** keyword, which will be searched at each splitting opportunity. Finally, you can also use ensembles of decision trees such as random forests, which we will describe shortly. Note that all these options, in addition to possibly avoiding the ill effects of the greedy algorithm, are also options to address the overfitting that decisions trees are often criticized for.

Training Decision Trees: Different Stopping Criteria

We have already reviewed using the **max_depth** parameter as a limit to how deep a tree will grow. However, there are several other options available in scikit-learn as well. These are mainly related to how many samples are present in a leaf node, or how much the impurity can be decreased by further splitting nodes. As we have already mentioned, you may be limited by the size of your dataset in terms of how deep you can grow a tree. And it may not make sense to grow trees deeper, especially if the splitting process is no longer finding nodes with higher purity.

We summarize all of the keyword arguments that you can supply to the **DecisionTreeClassifier** class in scikit-learn here:

Parameter	Possible values	Notes
criterion	string, 'gini' or 'entropy'	This is the formula used to calculate node impurity.
splitter	string, 'best' or 'random'	This determines whether to search among all candidate features when making a split, or to choose one at random.
max_depth	int or None	A stopping criterion; None means there is no limit to the maximum depth for growing the tree, although the tree may be stopped for reasons specified in other hyperparameters. An int integer means to stop growing the tree after that many levels.
min_samples_split	int or float	A stopping criterion. If an integer, then a node must have at least this many samples in order to be split. If a float value, it is the fraction of the total number of samples that must be in a node to split it.

Parameter	Possible values	Notes
min_samples_leaf	int or float	A stopping criterion. Similar to min_samples_split, but this refers to the number of samples that will be in the nodes after the split. It refers to a node at any depth.
min_weight_fraction_leaf	float	Similar to min_samples_leaf, but uses the weight fraction of samples instead of the raw fraction of samples. It is useful only if sample weighting has been specified with the class_weight parameter.
max_features	int, float, string: 'auto', 'sqrt', 'log2', or None	This is a strategy for how many features to consider when trying to split a node. If None, all the features are considered. If an int or float, then that number or fraction of features are considered. 'auto' and 'sqrt' both use the square root of the number of features, and 'log2' uses the base 2 logarithm. If the number here is less than the total number of features, a random selection of features is taken. However, more than this number of features may be considered if none of the random selection meets other criteria, such as min_impurity_decrease.

Parameter	Possible values	Notes
random_state	int or None	If an integer is supplied, this will seed the random number generator for repeatable results between runs of the same code.
max_leaf_nodes	int or None	A stopping criterion. It allows the maximum number of leaf nodes if an integer is supplied, or no limit for None.
min_impurity_decrease	float	A stopping criterion. If larger than 0, the required decrease in impurity to split a node. This means the tree only keeps growing if the nodes are getting purer.
min_impurity_split	float	A stopping criterion but being phased out. Use min_impurity_decrease instead.

Parameter	Possible values	Notes
class_weight	dict, list of dicts, 'balanced' or None	If the response variable has imbalanced classes, 'balanced' will weight samples similar to the logistic regression. Here, this will be used to calculate the weighted average of node impurity. A dictionary can be used to manually supply class weights, for example, if you have a reason to be more confident in a portion of the data. The list of dictionaries can be used for multi-output problems, which is when there are multiple response variables (which is beyond the scope of this course). None does no sample weighting.
presort	bool	This determines whether to sort the samples before growing the tree, which may speed up training for smaller datasets.

Figure 5.16: The complete list of options for the decision tree classifier in Scikit-Learn

Using Decision Trees: Advantages and Predicted Probabilities

While decision trees are simple in concept, there are several practical advantages that they can present:

No need to scale features

Consider the reasons why we needed to scale features for logistic regression. One reason is that, for some of the solution algorithms based on gradient descent, it is necessary that features be on the same scale. Another is that when we are using L1 or L2 regularization to penalize coefficients, all the features must be on the same scale so they are penalized equally. With decision trees, the node splitting algorithm considers each feature individually and, therefore, it doesn't matter whether the features are on the same scale or not.

Non-linear relationships and interactions

Because each successive split in a decision tree is performed on a subset of the samples resulting from previous split(s), decision trees can describe complex non-linear relationships of a single feature, as well as interactions between features. Consider our previous discussion of why `BILL_AMT_1` was chosen at the second level of the tree in Figure 5.4. Also, as a hypothetical example with synthetic data, consider the following dataset for classification:

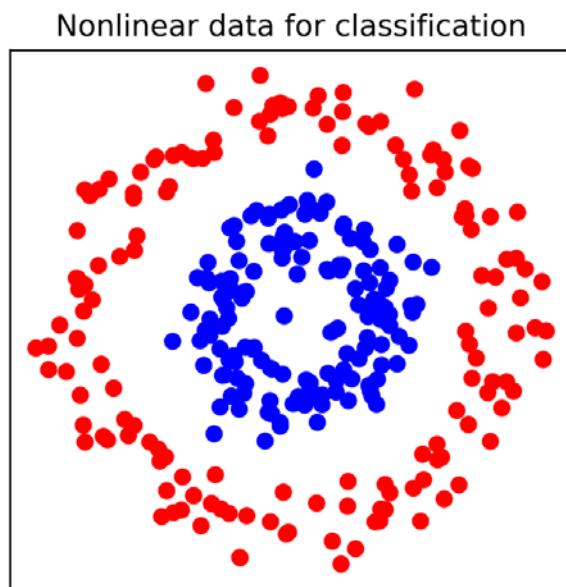


Figure 5.17: An example classification dataset, with the classes shown in red and blue.

We know from *Chapter 3, Details of Logistic Regression and Feature Exploration* that logistic regression has a linear decision boundary. So, how do you think logistic regression may cope with such a dataset as that shown in *Figure 5.17*? Where would you draw a line to separate the blue and red classes? It should be clear that without engineering additional features, a logistic regression is not likely to be a good classifier for these data. Now think about the set of "if, then" rules of a decision tree, which could be used with the features represented on the x and y axes of *Figure 5.17*. Do you think a decision tree will be effective with these data?

Here, we plot in the background the predicted probabilities of class membership using red and blue colors, for both of these models:

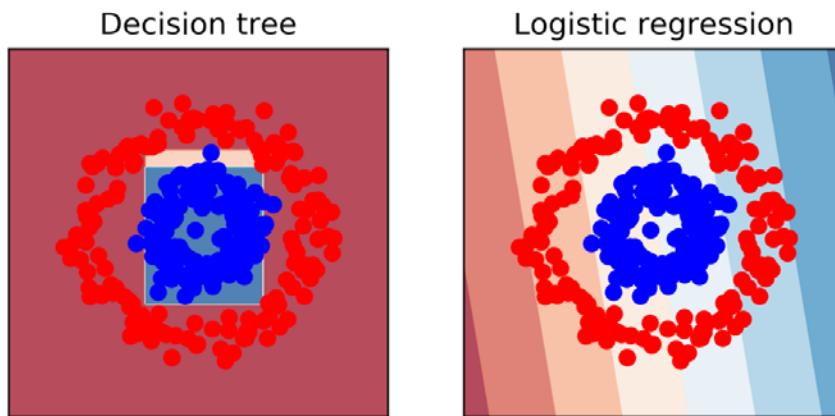


Figure 5.18: Decision tree and logistic regression predictions

In *Figure 5.18*, the predicted probabilities for both models are colored so that darker red corresponds to a higher predicted probability for the red class, while darker blue corresponds to a higher predicted probability for the blue class. We can see that the decision tree can isolate the blue class in the middle of the circle of red points. This is because, by using thresholds for the x and y coordinates in the node-splitting process, a decision tree can mathematically model the fact that the location of the blue and red classes depend on both the x and y coordinates together (interactions), and that the likelihood of either class is not a linearly increasing or decreasing function of x or y (non-linearities). Consequently, the decision tree approach is able to get most classifications right.

However, the logistic regression has a linear decision boundary, which will be the straight line between the lightest blue and red patches in the background. The logistic regression decision boundary goes right through the middle of the data and doesn't provide a useful classifier. This shows the power of decision trees "out of the box", without the need for engineering non-linear or interaction features.

Predicted probabilities

You may wonder how the predicted probabilities shown in Figure 5.18 were obtained from the decision tree model. We know that logistic regression produces probabilities as raw output. However, a decision tree makes predictions based on the majority of class of the leaf nodes. So, where will this probability come from? In fact, decisions trees do offer the `.predict_proba` method in scikit-learn. The probability is based on the proportion of the majority class in the leaf node. If the leaf node consisted 75% of the positive class, for example, the prediction for that node will be the positive class and the predicted probability will be 0.75. The predicted probabilities from decision trees are not considered to be as statistically rigorous as those from generalized linear models, but they are still commonly used to measure the performance of models by methods that depend on varying the threshold for classification, such as the ROC curve or the precision-recall curve.

Note

We are focusing here on decision trees for classification because of the nature of the case study. However, decision trees can also be used for regression, making them a versatile method. The tree-growing process is similar for regression as it is for classification, except that instead of seeking to reduce node impurity, a regression tree seeks to minimize other metrics such as the mean squared error (MSE) or mean absolute error (MAE) of the predictions, where the prediction for a node may be the average or median of the samples in the node, respectively.

A More Convenient Approach to Cross-Validation

In Chapter 4, *The Bias-Variance Trade-off*, we gained a deep understanding of cross-validation by writing our own function to do it, using the `KFold` class to generate the training and testing indices. This was helpful to get a thorough understanding of how the process works. However, scikit-learn offers a convenient class that can do more of the heavy lifting for us: `GridSearchCV`. `GridSearchCV` can take as input a model that we want to find optimal hyperparameters for, such as a decision tree or a logistic regression, and a "grid" of hyperparameters that we want to perform cross-validation over. For example, in a logistic regression, we may want to get the average cross-validation score over all the folds for different values of the regularization parameter `C`. With decision trees, we may want to explore different depths of trees. You can also search multiple parameters at once, for example, if we wanted to try different depths of trees and different numbers of `max_features` to consider at each node split.

GridSearchCV does what is called an exhaustive grid search over all the possible combinations of parameters that we supply. This means that if we supplied five different values for each of the two hyperparameters, the cross-validation procedure would be run $5 \times 5 = 25$ times. If you are searching many values of many hyperparameters, the amount of cross-validation runs can grow very quickly. In these cases, you may wish to use **RandomizedSearchCV**, which searches a randomly selected number of hyperparameter combinations from the universe of all possibilities in the grid you supply.

GridSearchCV can speed up your work by streamlining the cross-validation process. You should be familiar with the concepts of cross-validation from the previous chapter, so we proceed directly to listing all the options available for **GridSearchCV**. In the following exercise, we will get hands-on practice using **GridSearchCV** with the case study data, to search hyperparameters for a decision tree classifier. Here are the options for **GridSearchCV**:

Parameter	Possible values	Notes
estimator	estimator object	This is a model object that you have instantiated from a model class. The hyperparameters will be updated as GridSearchCV does its work.
param_grid	dict or list of dicts	The dictionary has parameter names as keys and lists of parameters as values. These are the hyperparameter values for which you want to search all possible combinations. To do multiple grid searches, supply a list of dictionaries.
scoring	string	This represents the model assessment metric that you want to use to measure training and testing performance across the folds, for example, 'roc_auc'.
n_jobs	int or None	This determines the number of processing jobs to run in parallel. It may speed up cross-validation to run parallel jobs, but it is a good idea to experiment in order to be sure.

Parameter	Possible values	Notes
pre_dispatch	int or string	This determines the number of jobs or formula for the number of jobs to dispatch. It is relevant for parallel processing using n_jobs.
iid	bool	This indicates whether the data is independent and identically distributed (i.i.d.). True to compute a weighted average score across folds, using the number of samples in each fold for weighting. False to compute an unweighted average. It is not relevant if the number of samples is the same in each fold.
cv	int, cross-validation generator or iterable	If supplying an integer, this is the number of folds to use for cross validation.
refit	bool or string	After doing the cross-validation, the “best” hyperparameters according to the metric specified in scoring can be used directly with the fitted GridSearchCV object to make predictions. If refit=True, the model will be refit to all of the data (not just one of the folds) using the best hyperparameters. Use the string argument if multiple metrics are specified.

Parameter	Possible values	Notes
verbose	int	This controls how much output you will see from the cross-validation procedure.
error_score	'raise' or numeric	This determines what to do if an error happens during model fitting.
return_train_score	bool	This determines whether or not to compute and return training scores on the folds. It is not required for selecting the best hyperparameters based on testing fold scores and for some datasets and models, this can take substantially more time. However, it does give insight into possible overfitting.

Figure 5.19: The options for `GridSearchCV`

Exercise 20: Finding Optimal Hyperparameters for a Decision Tree

In this exercise, we will use `GridSearchCV` to tune the hyperparameters for a decision tree model. You will learn about a convenient way of searching different hyperparameters with scikit-learn. Perform the following steps to complete the exercise:

Note

The code and the resulting output for this exercise have been loaded in a Jupyter Notebook and can be found at <http://bit.ly/2GI7fjB>.

1. Import the `GridSearchCV` class with this code:

```
from sklearn.model_selection import GridSearchCV
```

The next step is to define the hyperparameters that we want to search using cross-validation. We will find the best maximum depth of tree, using the `max_depth` parameter. Deeper trees have more node splits, which partition the training set into smaller and smaller subspaces using the features. While we don't know the best maximum depth ahead of time, it is helpful to consider some limiting cases when considering the range of parameters to use for the grid search.

We know that one is the minimum depth, consisting of a tree with just one split. As for the largest depth, you can consider how many samples you have in your training data, or more appropriately in this case, how many samples will be in the training fold for each split of the cross-validation. We will perform a 4-fold cross-validation like we did in the previous chapter. So, how many samples will be in each training fold and how does this relate to the depth of the tree?

2. Find the number of samples in the training data **using this code**:

```
X_train.shape
```

The output should be as follows:

(21331, 17)

Figure 5.20: The shape of the training data

With 21,331 training samples and 4-fold cross-validation, there will be three-fourth of the samples, or about 16,000 samples, in each training fold.

What does this mean for how deep we may wish to grow our tree?

A theoretical limitation is that we need at least one sample in each leaf. From our discussion about how the depth of the tree relates to the number of leaves, we know a tree that splits at every node before the last level, with n levels, has 2^n leaf nodes. Therefore, a tree with L leaf nodes has a depth of approximately $\log_2(L)$. In the limiting case, if we grow the tree deep enough so that every leaf node has one training sample for a given fold, the depth will be $\log_2(16,000) \approx 14$. So, 14 is the theoretical limit to the depth of a tree that we could grow in this case.

Practically speaking, you will probably not want to grow a tree this deep, as the rules used to generate the decision tree will be very specific to the training data and the model is likely to be overfit. However, this gives you an idea of the range of values we may wish to consider for the `max_depth` hyperparameter. We will explore a range of depths from 1 up to 12.

3. Define a dictionary with the key being the hyperparameter name and the value being the list of values of this hyperparameter that we want to search in cross-validation:

```
params = {'max_depth':[1, 2, 4, 6, 8, 10, 12]}
```

In this case, we are only searching one hyperparameter. However, you could define a dictionary with multiple key-value pairs to search for multiple hyperparameters simultaneously.

Now we want to instantiate the `GridSearchCV` class.

4. Instantiate the `GridSearchCV` class using these options:

```
cv = GridSearchCV(dt, param_grid=params, scoring='roc_auc', fit_
params=None,
n_jobs=None, iid=False, refit=True, cv=4, verbose=1,
pre_dispatch=None, error_score=np.nan, return_train_
score=True)
```

Notice that we can reuse the decision tree object, `dt`, that we already instantiated earlier in this chapter. When creating `dt`, we used the default arguments for all options but `max_depth`. However, this hyperparameter will be reset here using the `params` dictionary that we defined in each iteration of the cross-validation loop. The other notable options here are that we use the ROC AUC metric (`scoring='roc_auc'`), that we do a 4-fold cross-validation (`cv=4`), and that we calculate training scores (`return_train_score=True`) to assess the bias-variance trade-off.

Once the cross-validation object is defined, we can simply use the `.fit` method on it as we would with a model object. This encapsulates essentially all the functionality of the cross-validation loop we wrote in the previous chapter.

5. Perform a 4-fold cross-validation to search for the optimal maximum depth using this code:

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

The output should be as follows:

```
Fitting 4 folds for each of 7 candidates, totalling 28 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  28 out of  28 | elapsed:    2.8s finished

GridSearchCV(cv=4, error_score=nan,
            estimator=DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
                                              max_features=None, max_leaf_nodes=None,
                                              min_impurity_decrease=0.0, min_impurity_split=None,
                                              min_samples_leaf=1, min_samples_split=2,
                                              min_weight_fraction_leaf=0.0, presort=False, random_state=None,
                                              splitter='best'),
            fit_params=None, iid=False, n_jobs=None,
            param_grid={'max_depth': [1, 2, 4, 6, 8, 10, 12]},
            pre_dispatch=None, refit=True, return_train_score=True,
            scoring='roc_auc', verbose=1)
```

Figure 5.18: The cross-validation fitting output

All the options that we specified are printed as output. Additionally, there is some output information about how many cross-validation fits were performed. We had 4 folds and 7 hyperparameters, meaning $4 \times 7 = 28$ fits are performed. The amount of time this took is also displayed. You can control how much output you get from this procedure with the `verbose` keyword argument; larger numbers mean more output.

Now it's time to examine the results of the cross-validation procedure. Among the methods that are available on the fitted `GridSearchCV` object is `.cv_results_`. This is a dictionary containing the names of results as keys and the results themselves as values. For example, the `mean_test_score` key holds the average testing score across the folds for each of the seven hyperparameters. You could directly examine this output by running `cv.cv_results_` in a code cell. However, this is not easy to read. Dictionaries with this kind of structure are immediately usable for creation of a pandas DataFrame, which makes looking at the results a little easier.

6. Run the following code to create and examine a pandas `DataFrame` of cross-validation results:

```
cv_results_df = pd.DataFrame(cv.cv_results_)
cv_results_df
```

The output should be as follows:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_depth	params	split0_test_score
0	0.023161	0.002917	0.002712	0.000827	1	{'max_depth': 1}	0.639514
1	0.040478	0.005298	0.003616	0.001432	2	{'max_depth': 2}	0.695134
2	0.062975	0.001703	0.002196	0.000017	4	{'max_depth': 4}	0.732720
3	0.094926	0.005329	0.002393	0.000150	6	{'max_depth': 6}	0.743836
4	0.123850	0.008124	0.003107	0.000768	8	{'max_depth': 8}	0.727948
5	0.142454	0.001005	0.002620	0.000221	10	{'max_depth': 10}	0.709049
6	0.178841	0.016180	0.002716	0.000238	12	{'max_depth': 12}	0.675597

Figure 5.19: Cross-validation results

The DataFrame has one row for each combination of hyperparameters in the grid. Since we are only searching one hyperparameter here, there is one row for each of the seven values that we searched for. You can see a lot of output for each row, such as the mean and standard deviation of the time in seconds that each of the four folds took for both training (fitting) and testing (scoring). The hyperparameter values that were searched for are also shown. In figure 5.19 we can see the ROC AUC score for the testing data of the first fold (index 0). So, what are the rest of the columns in the results DataFrame?

- View the names of the remaining columns in the results DataFrame using this code:

```
cv_results_df.columns
```

The output should be as follows:

```
Index(['mean_fit_time', 'std_fit_time', 'mean_score_time', 'std_score_time',
       'param_max_depth', 'params', 'split0_test_score', 'split1_test_score',
       'split2_test_score', 'split3_test_score', 'mean_test_score',
       'std_test_score', 'rank_test_score', 'split0_train_score',
       'split1_train_score', 'split2_train_score', 'split3_train_score',
       'mean_train_score', 'std_train_score'],
      dtype='object')
```

Figure 5.20: Columns in the DataFrame of cross-validation results

The columns in the cross-validation results DataFrame include the testing scores for each fold, their average and standard deviation, and the same information for the training scores.

Generally speaking, the "best" combination of hyperparameters is that with the highest average testing score. This is an estimation of how well the model, fit using these hyperparameters, could perform when scored on new data. Let's make a plot showing how the average testing score varies with the `max_depth` hyperparameter. We will also show the average training scores on the same plot, to see how bias and variance change as we allow deeper and more complex trees to be grown during model fitting.

We include the standard deviations of the 4-fold training and testing scores as error bars, using the Matplotlib `errorbar` function. This gives you an indication of how variable the scores are across the folds.

8. Execute the following code to create an error bar plot of training and testing scores for each value of `max_depth` that was examined in cross-validation:

```
ax = plt.axes()
ax.errorbar(cv_results_df['param_max_depth'],
            cv_results_df['mean_train_score'],
            yerr=cv_results_df['std_train_score'],
            label='Mean $\pm$ 1 SD training scores')
ax.errorbar(cv_results_df['param_max_depth'],
            cv_results_df['mean_test_score'],
            yerr=cv_results_df['std_test_score'],
            label='Mean $\pm$ 1 SD testing scores')
ax.legend()
plt.xlabel('max_depth')
plt.ylabel('ROC AUC')
```

The plot should appear as follows:

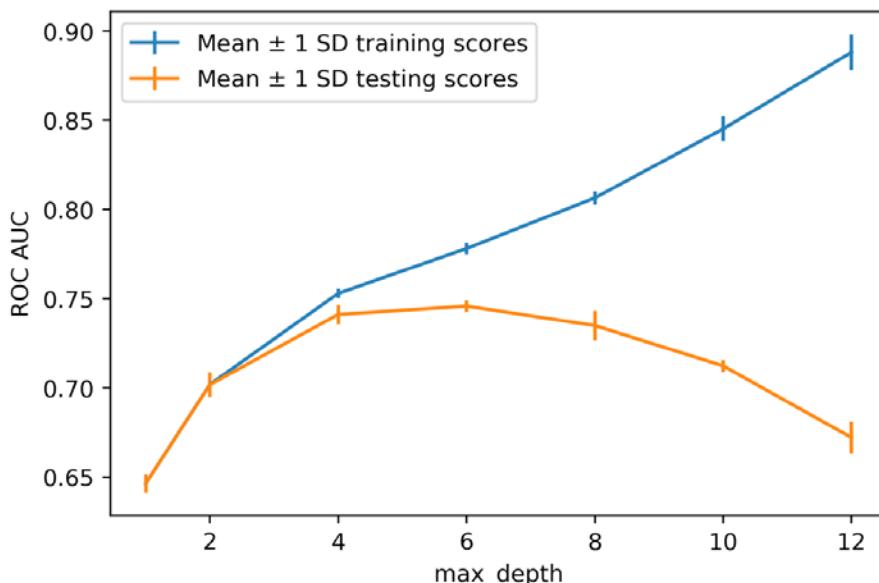


Figure 5.21: An error bar plot of training and testing scores across the four folds

The standard deviations of the training and testing scores are shown as vertical lines at each value of `max_depth` that was tried; the distance above and below the average score is 1 standard deviation. Whenever making error bar plots, it's best to ensure that the units of the error measurement are the same as the units of the y axis. In this case they are, since standard deviation has the same units as the underlying data, as opposed to variance, for example, which has squared units.

The error bars indicate how variable the scores are across folds. If there was a large amount of variation across the folds, it would indicate that the nature of the data across the folds was different in a way that affected the ability of our model to describe it. This could be concerning because it would indicate that we may not have enough data to train a model that would reliably perform on new data. However, in our case here, there is not much variability between the folds, so this is not an issue.

What about the general trends of the training and testing scores across the different values of `max_depth`? We can see that as we grow deeper and deeper trees, the model fits the training data better and better. As noted previously, if we grew trees deep enough so that each leaf node had just one training sample, we create a model that is very specific to the training data. In fact, it would fit the training data perfectly. We could say that such a model had extremely high **variance**.

But this performance on the training set does not necessarily translate over to the testing set. In Figure 5.21, it's apparent that increasing `max_depth` only increases testing scores up to a point, after which deeper trees in fact have lower testing performance. This is another example of how we can leverage the **bias-variance trade-off** to create a better predictive model – similar to how we use a regularized logistic regression. Shallower trees have more **bias**, since they are not fitting the training data as well. But this is fine because if we accept some bias, we will have better performance on the testing data, which is ultimately the metric we use to select model hyperparameters.

In this case, we would select `max_depth = 6`. You could also do a more thorough search, by trying every integer between 2 and 12, instead of going by 2's as we've done here. In general, it is a good idea to perform as thorough a search of parameter space as you can, up to the limits of the computational time that you have. In this case, it would lead to the same result.

Comparison between models

At this point, we've calculated a 4-fold cross-validation of several different machine learning models on the case study data. So, how are we doing? What's our best so far? In the last chapter, we got an average testing ROC AUC of 0.718 with logistic regression and 0.740 by engineering interaction features in a logistic regression. Here, with a decision tree, we can achieve 0.745. So, we are making gains in model performance. We will explore one more type of model to see if we can push performance even higher.

Random Forests: Ensembles of Decision Trees

As we saw in the previous exercise, decision trees are prone to overfitting. This is one of the principle criticisms of their usage, despite the fact that they are highly interpretable. We were able to limit this overfitting, to an extent, however, by limiting the maximum depth to which the tree could be grown.

It turns out that there are powerful and widely-used predictive models that use decision trees as the basis for more complex procedures. In particular, we will focus here on random forests of decision trees. Random forests are examples of what are called ensemble models, because they are formed by combining other models. By combining the predictions of many models, it is possible to improve upon the deficiencies of any given one of them.

Once you understand decision trees, the concept behind random forests is actually quite simple. That is because random forests are just ensembles of many decision trees; all the models in this kind of ensemble have the same mathematical form. So, how many decision tree models will be included in a random forest? This is one of the hyperparameters, `n_estimators`, that needs to be specified when building a random forest model. Generally speaking, the more trees, the better. As the number of trees increases, the variance of the overall ensemble will decrease. This should result in the random forest model having better generalization to new data, which will be reflected in increased testing scores. However, there will be a point of diminishing returns after which increasing the number of trees does not result in a substantial improvement in model performance.

So, how do random forests reduce the high variance (overfitting) issue that affects decision trees? The answer to this question lies in what is different about the different trees in the forest. There are two principle ways in which the trees are different, one of which we are already familiar with:

- The number of features considered at each split
- The samples used to grow different trees

The number of features considered at each split

We are already familiar with this option from the `DecisionTreeClassifier` class: `max_features`. In our previous usage of this class, we left `max_features` at its default value of `None`, which meant that all features were considered at each split. By using all the features to fit the training data, overfitting is possible. By limiting the number of features considered at each split, some of the decision trees in a random forest will potentially find better splits. This is because, although they are still greedily searching for the best split, they are doing it with a limited selection of features. This may make certain splits possible later in the tree that may not have been found if all features were being searched at each split.

There is a `max_features` option in the `RandomForestClassifier` class in scikit-learn just as there is for the `DecisionTreeClassifier` class and the options are similar. However, for the random forest, the default setting is 'auto', which means the algorithm will only search a random selection of the square root of the number of possible features at each split, for example a random selection of $\sqrt{9} = 3$ features from a total of 9 possible features. Because each tree in the forest will likely choose different random selections of features to split as the trees are being grown, the trees in the forest will not be the same.

The samples used to grow different trees

The other way that the trees in a random forest differ from each other is that they are usually grown with different training samples. To do this, a statistical procedure known as bootstrapping is used, which means to generate new synthetic datasets from the original data. The synthetic datasets are created by randomly selecting samples from the original dataset using replacement. Here, "replacement" means that if we select a certain sample, we will continue to consider it for selection, that is, it is "replaced" to the original dataset after we've sampled it. The number of samples in the synthetic datasets are the same as those in the original dataset, but some samples may be repeated because of replacement.

The procedure of using random sampling to create synthetic datasets, and training models on them separately, is called bagging, which is short for bootstrapped aggregation. Bagging can, in fact, be used with any machine learning model, not just decision trees, and scikit-learn offers functionality to do this for both classification (**BaggingClassifier**) and regression (**BaggingRegressor**) problems. In the case of random forest, bagging is turned on by default and the bootstrap option is set to **True**. But if you want all the trees in the forest to be grown using all of the training data, you can set this option to **False**.

Now you should have a good understanding of what a random forest is. As you can see, if you are already familiar with decision trees, understanding random forests does not involve much additional knowledge. A reflection of this fact is that the hyperparameters available for the **RandomForestClassifier** class in scikit-learn are mostly the same as those for the **DecisionTreeClassifier** class.

In addition to **n_estimators** and **bootstrap**, which we discussed previously, there are only two additional new options beyond what's available for decision trees:

- **oob_score, a bool**: This option controls whether or not to calculate an out of bag (OOB) score for each tree. This can be thought of as a testing score, where the samples that were not selected by the bagging procedure to grow a given tree are used to assess model performance of that tree. Here, use **True** to calculate the OOB score or **False** (the default) not to.
- **warm_start, a bool**: This is **False** by default – if you set this to **True**, then reusing the same random forest model object will cause additional trees to be added to the already-generated forest.

Other kinds of ensemble models

Random forest, as we now know, is an example of a bagging ensemble. Another kind of ensemble is a **boosting** ensemble. The general idea of boosting is to use successive new models of the same type and to train them on the errors of previous models. This way, successive models learn where earlier models didn't do well and correct these errors. Boosting has enjoyed successful application with decision trees and is available in scikit-learn and another popular Python package called **XGBoost**.

Stacking ensembles are a somewhat more advanced kind of ensemble, where the different models (estimators) within the ensemble do not need to be of the same type as they do in bagging and boosting. For example, you could build a stacking ensemble with a random forest and a logistic regression. The predictions of the different members of the ensemble are combined for a final prediction using yet another model (the **stacker**), which considers the predictions of the **stacked** models as features.

Random Forest: Predictions and Interpretability

Since a random forest is just a collection of decision trees, somehow the predictions of all those trees must be combined to create the prediction of the random forest.

After model training, classification trees will take an input sample and produce a predicted class, for example, whether or not a credit account in our case study problem will default. One intuitive approach to combining the predictions of these trees into the ultimate prediction of the forest is to take a majority vote. That is, whatever the most common prediction of all the trees becomes the prediction of the forest. This was, in fact, the approach taken in the publication first describing random forests (<https://scikit-learn.org/stable/modules/ensemble.html#forest>). However, scikit-learn uses a somewhat different approach: adding up the predicted probabilities for each class and then choosing the one with the highest probability. This captures more information from each tree than just the predicted class.

Interpretability of random forests

One of the main advantages of decision trees is that it is straightforward to see how any individual prediction is made. You can trace the decision path for any predicted sample through the series of "if then" rules used to make a prediction and know exactly how it came to have that prediction. By contrast, imagine that you have a random forest consisting of 1,000 trees. This would mean there are 1,000 sets of rules like this, which are much harder to communicate to human beings than 1 set of rules!

That being said, there are various methods that can be used to understand how random forests make predictions. There are advanced techniques for understanding how individual predictions are made, that are the subject of recent academic research and are beyond the scope of this book. However, a simple way to interpret the way a random forest works, that is available in scikit-learn, is to observe the feature importance. The feature importance of a random forest are a measure of how useful each of the features were when growing the trees in the forest. This usefulness is measured by using a combination of the fraction of training samples that were split using that feature, and the decrease in node impurity that resulted.

Because of the feature importance calculation, which can be used to rank features by how useful they are to the random forest model, random forests can also be used for feature selection.

Exercise 21: Fitting a Random Forest

In this exercise, we will extend our efforts with decision trees, by using the random forest model with cross-validation on the training data from the case study. We will observe the effect of increasing the number of trees in the forest and examine the feature importance that can be calculated using a random forest model. Perform the following steps to complete the exercise:

Note

The code and the resulting output for this exercise have been loaded in a Jupyter Notebook that can be found at <https://bit.ly/2UpGDW2>.

1. Import the random forest classifier model class as follows:

```
from sklearn.ensemble import RandomForestClassifier
```

2. Instantiate the class using these options:

```
rf = RandomForestClassifier(  
    n_estimators=10, criterion='gini', max_depth=3,  
    min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0,  
    max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0,  
    min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=None,  
    random_state=4, verbose=0, warm_start=False, class_weight=None)
```

For this exercise, we'll use mainly the default options. However, note that we will set `max_depth = 3`. Here, we are only going to explore the effect of using different numbers of trees, which we will illustrate with relatively shallow trees for the sake of shorter runtimes. To find the best model performance, we'd typically try more and deeper depths of trees.

We also set `random_state` for predictable results across runs.

3. Create a parameter grid for this exercise in order to search the numbers of trees ranging from 10 to 100:

```
rf_params_ex = {'n_estimators':list(range(10,110,10))}
```

We use Python's `range()` function to create an iterator for the integer values we want, and then convert it to a `list` using `list()`.

4. Instantiate a grid search cross-validation object for the random forest model using the parameter grid from the previous step. Otherwise, you can use the same options that were used for the cross-validation of the decision tree:

```
cv_rf_ex = GridSearchCV(rf, param_grid=rf_params_ex, scoring='roc_auc',
                        fit_params=None,
                        n_jobs=None, iid=False, refit=True, cv=4,
                        verbose=1,
                        pre_dispatch=None, error_score=np.nan, return_
                        train_score=True)
```

5. Fit the cross-validation object as follows:

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

The fitting procedure should output the following:

```
Fitting 4 folds for each of 10 candidates, totalling 40 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done  40 out of  40 | elapsed:   22.9s finished

GridSearchCV(cv=4, error_score=nan,
            estimator=RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                                             max_depth=3, max_features='auto', max_leaf_nodes=None,
                                             min_impurity_decrease=0.0, min_impurity_split=None,
                                             min_samples_leaf=1, min_samples_split=2,
                                             min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=None,
                                             oob_score=False, random_state=4, verbose=0, warm_start=False),
            fit_params=None, iid=False, n_jobs=None,
            param_grid={'n_estimators': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]},
            pre_dispatch=None, refit=True, return_train_score=True,
            scoring='roc_auc', verbose=1)
```

Figure 5.22: The output from the cross-validation of the random forest across different numbers of trees

You probably noticed that, although we are only cross-validating over 10 hyperparameter values, comparable to the 7 values that we examined for the decision tree in the previous exercise, this cross-validation took noticeably longer. Consider how many trees we are growing in this case. For the last hyperparameter, `n_estimators` = 100, we have grown a total of 400 trees across all the cross-validation splits.

So, how long has model fitting taken across the various numbers of trees that we just tried? What gains in cross-validation testing performance have we made by using more trees? These are good things to examine using plots. First, we'll pull the cross-validation results out in to a pandas DataFrame as we've done before.

6. Put the cross-validation results into a pandas **DataFrame**:

```
cv_rf_ex_results_df = pd.DataFrame(cv_rf_ex.cv_results_)
```

You can examine the whole **DataFrame** in the accompanying Jupyter notebook. Here, we move directly to creating plots of the quantities of interest. We'll make a line plot, with symbols, of the mean fit time across the folds for each hyperparameter, contained in the column `mean_fit_time`, as well as an error bar plot of testing scores, which we've already done for decision trees. Both plots will be against `max_depth` on the x axis.

7. Create two subplots, of the mean time and mean testing scores with standard deviation:

```
fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(6, 3))
axs[0].plot(cv_rf_ex_results_df['param_n_estimators'],
            cv_rf_ex_results_df['mean_fit_time'],
            '-o')
axs[0].set_xlabel('Number of trees')
axs[0].set_ylabel('Mean fit time (seconds)')
axs[1].errorbar(cv_rf_ex_results_df['param_n_estimators'],
                cv_rf_ex_results_df['mean_test_score'],
                yerr=cv_rf_ex_results_df['std_test_score'])
axs[1].set_xlabel('Number of trees')
axs[1].set_ylabel('Mean testing ROC AUC $\pm$ 1 SD ')
plt.tight_layout()
```

Here, we've used `plt.subplots` to create two axes at once, within a figure, in a one-row-by-two-column configuration. We then access the axes objects by indexing the array of `axs` axes returned from this operation, to create plots.

The output should be this plot:

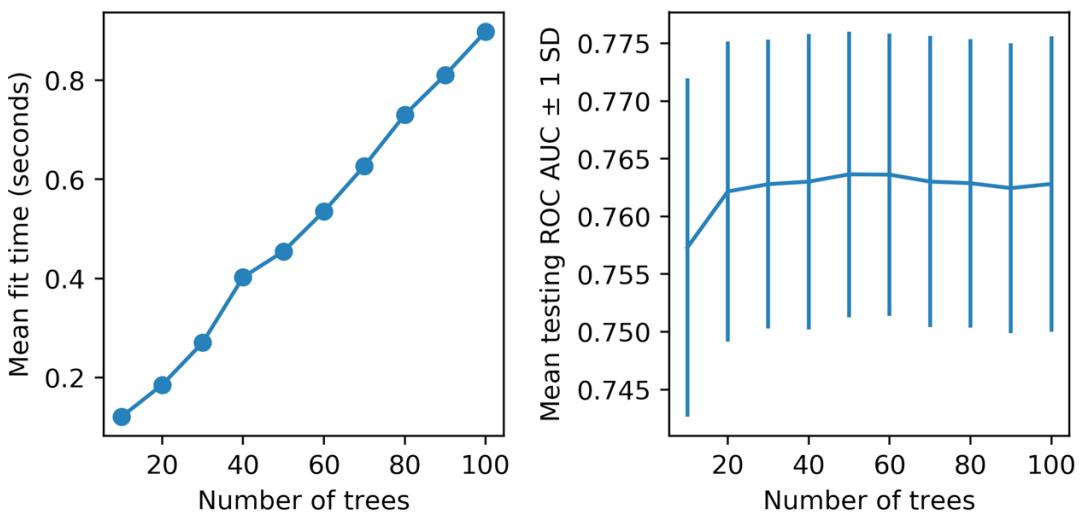


Figure 5.23: The mean fitting time and testing scores for different numbers of trees in the forest

Note

Your results may be different due to the differences in the platform or if you set a different random seed.

There are several things to note about these visualizations. First of all, we can see that by using a random forest, we have increased model performance on the cross-validation testing folds above that of any of our previous efforts. While we haven't made an attempt to tune the random forest hyperparameters to achieve the best model performance we can, this is a promising result and indicates that a random forest will be a valuable addition to our modeling efforts.

However, along with these higher model testing scores, notice that there is also more variability between the folds than what we saw with the decision tree; this variability is visible as larger standard deviations in model testing scores across the folds. While this indicates that there is a wider range in model performance that might be expected from using this model, you are encouraged to examine the model testing scores of the folds directly in the pandas DataFrame in the Jupyter notebook. You should see that even the lowest score from an individual fold is still higher than the average testing score from the decision tree, indicating that it will be better to use a random forest.

So, what about the other questions that we set out to explore with this visualization? We are interested in seeing how long it takes to fit random forest models with various numbers of trees, and what the gains in model performance are from using more trees. The subplot on the left of *Figure 5.23* shows that there is a linear increase in training time as more trees are added to the forest. This is probably to be expected; we are simply multiplying the amount of computation to be done in the training procedure by adding more trees.

But is this additional computational time worth it in terms of increased model performance? The subplot on the right of *Figure 5.23* shows that beyond about 20 trees, it's not clear that adding more trees reliably improves testing performance. While the model with 50 trees has the highest score, the fact that adding more trees actually decreases the testing score somewhat indicates that the gain in ROC AUC for 50 trees may just be due to randomness, as adding more trees theoretically should increase model performance. Based on this reasoning, if we were limited to `max_depth = 3`, we may choose a forest of 20 or perhaps 50 trees and proceed. However, we will explore the parameter space more fully in the activity at the end of this chapter.

Finally, note that we have not shown the training ROC AUC metrics here. If you were to plot these or look them up in the results DataFrame, you'd see that the training scores are higher than the testing scores, indicating that some amount of overfitting is happening. While this may be the case, it's still true that the testing scores for this random forest model are higher than those that we've observed for any other model. Based on this result, we would likely choose the random forest model.

As a few additional insights into what we can access using our fitted cross-validation object, let's take a look at the best hyperparameters and the feature importance.

8. Use this code to see the best hyperparameters from cross-validation:

```
cv_rf_ex.best_params_
```

This should output:

```
{ 'n_estimators': 50}
```

Figure 5.24: Best hyperparameters from cross-validation

Here, "best" just means the hyperparameters that resulted in the highest average model testing score.

9. Run this code to create a **DataFrame** of the feature names and importance, and then show it sorted by importance:

```
feat_imp_df = pd.DataFrame({
    'Feature name':features_response[:-1],
    'Importance':cv_rf_ex.best_estimator_.feature_importances_
})
feat_imp_df.sort_values('Importance', ascending=False)
```

The first few rows of output should be as follows:

	Feature name	Importance
4	PAY_1	0.609609
11	PAY_AMT1	0.094123
0	LIMIT_BAL	0.079265
13	PAY_AMT3	0.047067
12	PAY_AMT2	0.035393

Figure 5.25: Feature importance from a random forest

In this code, we've created a dictionary with feature names and importance. The feature importance came from the **best_estimator_** method of the fitted cross-validation object. This is a way to access the random forest model object, that was trained on all the training data, using the best hyperparameters we viewed in the previous step. **feature_importances_** is a method that can be used on fitted random forest models. After putting the feature names and importance in a dictionary, we create the DataFrame, and then show it sorted, descending by importance. Notice that the top 5 most important features from the random forest, are the same as the top 5 chosen by an ANOVA F-test in *Chapter 3, Details of Logistic Regression and Feature Exploration*, although they are in a somewhat different order. This is good confirmation between the different methods.

Checkerboard Graph

Before moving on to the Activity, we illustrate a visualization technique in Matplotlib. Plotting a two-dimensional grid with colored squares or other shapes on it, can be useful when you want to show three dimensions of data. Here, color illustrates the third dimension. For example, you may want to visualize model testing scores over a grid of two hyperparameters. This is, in fact, the use case in [Activity 5, Cross-Validation Grid Search with Random Forest](#).

The first step in the process is to create grids of x and y coordinates. The NumPy **meshgrid** function can be used to do this. This function takes one-dimensional arrays of x and y coordinates and creates the mesh grid with all the possible pairs from both. The points in the mesh grid will be the corners of the checkerboard plot. Here is how the code looks for a 4 x 4 grid of colored patches. Since we are specifying the corners, we need a 5 x 5 grid of points:

```
xx_example, yy_example = np.meshgrid(range(5), range(5))
print(xx_example)
print(yy_example)
```

The output is as follows:

```
[[0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]]
 [[0 0 0 0 0]
 [1 1 1 1 1]
 [2 2 2 2 2]
 [3 3 3 3 3]
 [4 4 4 4 4]]
```

Figure 5.26: The mesh grid output

The grid of data to plot on this mesh should also have a square shape. We take one-dimensional array of integers between 1 and 16, and reshape it to a two-dimensional, 4 x 4 grid:

```
z_example = np.arange(1,17).reshape(4,4)  
z_example
```

```
array([[ 1,  2,  3,  4],  
       [ 5,  6,  7,  8],  
       [ 9, 10, 11, 12],  
       [13, 14, 15, 16]])
```

Figure 5.27: Data for the checkerboard plot

We can plot the `z_example` data on the `xx_example`, `yy_example` mesh grid with the following code. Notice that we use `pcolormesh` to make the plot with the `jet` colormap, which gives a rainbow color scale. We add a `colorbar`, which needs to be passed the object `pcolor_ex` returned by `pcolormesh` as an argument, so the interpretation of the color scale is clear:

```
ax = plt.axes()  
  
pcolor_ex = ax.pcolormesh(xx_example, yy_example, z_example, cmap=plt.  
cm.jet)  
  
plt.colorbar(pcolor_ex, label='Color scale')  
ax.set_xlabel('X coordinate')  
ax.set_ylabel('Y coordinate')
```

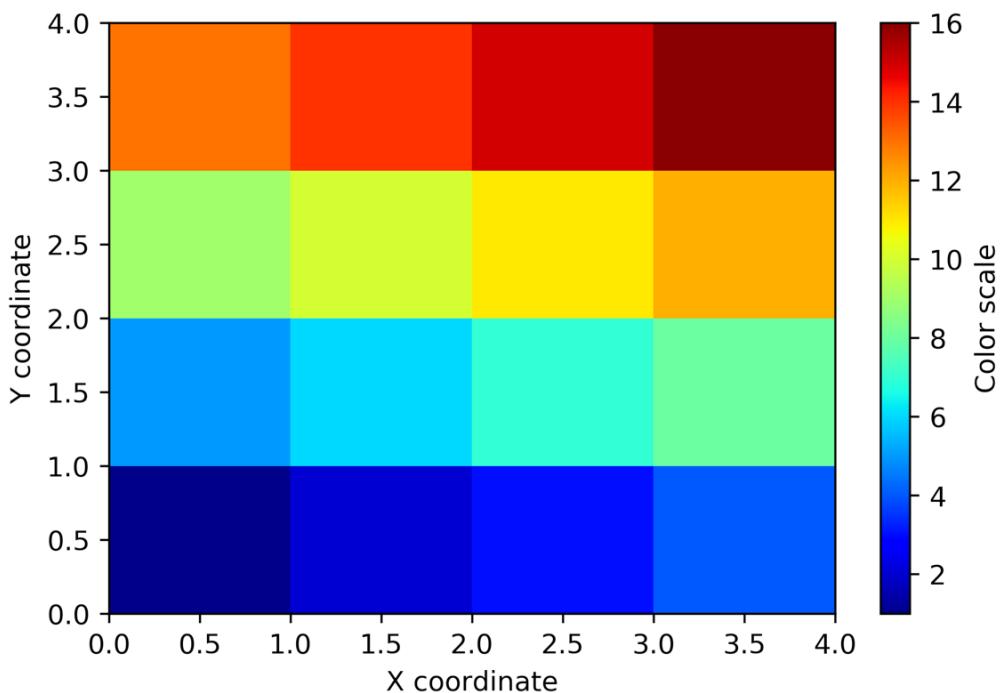


Figure 5.28: A `pcolor` mesh plot of consecutive integers

Activity 5: Cross-Validation Grid Search with Random Forest

In this activity, you will conduct a grid search over the number of trees in the forest (`n_estimators`) and the maximum depth of a tree (`max_depth`) for a random forest model on the case study data. You will then create a visualization showing the average testing score for the grid of hyperparameters that you searched over. Perform the following steps to complete the activity:

Note

The code and the resulting output for this activity have been loaded in a Jupyter Notebook that can be found at <http://bit.ly/2GI7fjB>.

1. Create a dictionary representing the grid for the `max_depth` and `n_estimators` hyperparameters that will be searched. Include depths of 3, 6, 9, and 12, and 10, 50, 100, and 200 trees. Leave the other hyperparameters at their defaults.
2. Instantiate a `GridSearchCV` object using the same options that we have previously in this chapter, but with the dictionary of hyperparameters created in step 1 here. Set `verbose=2` to see the output for each fit performed. You can reuse the same random forest model object `rf` that we have been using.
3. Fit the `GridSearchCV` object on the training data.
4. Put the results of the grid search in a pandas DataFrame.
5. Create a `pcolormesh` visualization of the mean testing score for each combination of hyperparameters.
6. Conclude which set of hyperparameters to use.

Note

The solution to this activity can be found on page 344.

Summary

In this chapter, we've learned how to use decision trees and the ensemble models called random forests that are made up of many decision trees. Using these simply conceived models, we were able to make better predictions than we could with logistic regression, judging by the cross-validation ROC AUC score. This is often the case for many real-world problems. Decision trees are robust to a lot of the potential issues that can prevent logistic regression models from good performance, such as non-linear relationships between features and the response variable, and the presence of complicated interactions among features.

Although a single decision tree is prone to overfitting, the random forest ensemble method has been shown to reduce this high-variance issue. Random forests are built by training many trees. The decreased variance of the ensemble of trees is achieved by increasing the bias of the individual trees in the forest, by only training them on a portion of the available training set (bootstrapped aggregation or bagging) and only considering a reduced number of features at each node split.

Now we have tried several different machine learning approaches to modeling the case study data. We found that some work better than others; for example, a random forest with tuned hyperparameters provides the highest average cross-validation ROC AUC score of 0.776.

Ultimately, however, the ROC AUC score is an abstract metric by which to judge a model. The client may not have much context for what the ROC AUC score means for them in a practical sense, but they definitely need to know whether the model can meet their business needs. For this reason, a crucial final step for machine learning in a business context is to try to determine the financial, or business, value of a predictive model. We will show you how to go about this, as well as address other issues having to do with delivering models that will be used to guide business decisions, in the next chapter.

6

Imputation of Missing Data, Financial Analysis, and Delivery to Client

Learning Objectives

By the end of this chapter, you will be able to:

- Compare the results of all models built for a case study
- Replace missing data using a range of imputation strategies
- Build a multiclass classification model
- Conduct financial analysis to find the optimal threshold for binary classification
- Derive financial insight from the model to help the client guide budgeting and operational strategy
- Deliver the model and make recommendations for usage

This chapter presents a comparison of the results from the various models built for a study, describes the financial insights derived from the final model, and outlines the final steps of the project needed to satisfy the client's requirements.

Introduction

In the previous chapter, we introduced decision trees and random forests and saw how they could be used to improve the quality of predictive modeling of the case study data.

In this chapter, we consider model building to be complete and address all the remaining issues that need attention before delivering the model to the client. The two key elements of this chapter are data imputation and financial analysis.

With data imputation, you will explore several strategies for making educated guesses of the missing values of features of the dataset. This should enable you to make predictions for all samples.

In the financial analysis, you will take the final yet crucial steps of understanding how a model can be used in the real world. Your client will likely appreciate the efforts you made in creating a more accurate model or one with higher ROC AUC. However, they will definitely appreciate understanding how much money the model can help them earn or save and will be happy to receive specific guidance on how to maximize the model's potential for this. Once we review imputation and financial analysis, we conclude with a few thoughts on how to deliver and monitor the model.

Review of Modeling Results

In order to develop a binary classification model to meet the business requirements of our client, we have now tried several modeling approaches to varying degrees of success. In the end, we will pick the one that worked the best, to perform additional analyses on and present to our client. However, it is also good to present the client with findings from the various options that were explored. This shows that a thorough job was done.

Here, we review the different models that we tried for the case study problem, the hyperparameters that we needed to tune, and results from cross-validation. We only include the work we did using all possible features, not the earlier models where we used only one or two features (e.g. **EDUCATION**) as a way to learn how to use model-fitting functions in scikit-learn.

Model	Location in courseware	Tuned hyperparameters	Average 4-fold cross-validated ROC AUC on training set
Logistic regression with L1 regularization	<i>Lesson 4, Activity 4: Cross Validation and Feature Engineering with the Case Study Data</i>	Regularization parameter C	0.718
Logistic regression with L1 regularization and engineered interaction features	<i>Lesson 4, Activity 4: Cross Validation and Feature Engineering with the Case Study Data</i>	Regularization parameter C	0.740
Decision tree	<i>Lesson 5, Exercise 20, Finding Optimal Hyperparameters for a Decision Tree</i>	Maximum depth	0.745
Random forest	<i>Lesson 5, Activity 5, Cross-Validation Grid Search with Random Forest</i>	Maximum depth and number of trees	0.776

Figure 6.1: Summary of modeling activities with case study data

From Figure 6.1, we can see that for this particular problem, our efforts in creating more **complex models**, either by engineering new features to add to a simple logistic regression, or by creating an ensemble of decision trees, yielded increased model performance.

When presenting results to the client, it would usually be better to only present the first and last columns of Figure 6.1. The client is likely not interested in all the details in the "Tuned hyperparameters" column, unless they are technically inclined. You should be prepared to interpret results for business partners at all levels of technical familiarity, including those with very little technical background.

A common situation would be a client who doesn't understand the derivation of the ROC AUC measure. In this case, you would need to explain that it's a metric that can vary between 0.5 and 1 and give intuitive explanations for these limits: 0.5 is no better than a coin flip and 1 is perfection, which is essentially unattainable. Our results are somewhere in between, getting closer to 0.8 with the best model we developed. While this number is not necessarily meaningful by itself, taken together with all the models we tried and the limitations on the ROC AUC measure, it shows that we've tried several methods and have achieved improved performance. In the end, for a business application like the case study, abstract model performance metrics like ROC AUC should be accompanied by a financial analysis if possible. We will explore this later in this chapter.

Depending on how much time you have for a project and your expertise in different modeling techniques, you would want to try as many modeling methods as possible. More advanced methods, such as gradient-boosted decision trees or neural networks for classification, may yield improved performance on this problem. We encourage you to continue your studies and learn how to use these models. In particular now that you know about decision trees and random forests, gradient-boosted trees would be a logical next step, since they are an ensemble model based on decision trees, similar to random forests. However, for the case study, let's assume that we've reached the end of the time we've budgeted for model-building. Now, we need to move on to the practical concerns associated with delivering a model to our client.

Therefore, our conclusion is that a random forest model, with the hyperparameters we determined in cross-validation (200 trees and a maximum depth of 9), will be the model we recommend.

As a final remark on model building for the case study, note that we did not address the class imbalance in the response variable. You are encouraged to try fitting models with the `class_weight='balanced'` option in scikit-learn to see the effect; you will find that it won't improve the model's performance substantially, as the class imbalance for the case study data is not particularly severe.

Dealing with Missing Data: Imputation Strategies

Recall that in *Chapter 1, Data Exploration and Cleaning*, we encountered a sizable proportion of samples in the dataset ($3,021/29,685 = 10.2\%$) where the value of the `PAY_1` feature was missing. This is a problem that needs to be dealt with, because many machine learning algorithms, including the implementations of logistic regression and random forest in scikit-learn, cannot accept input for model training or testing that includes missing values.

Our solution to this problem was to simply discard all the samples that had missing values for **PAY_1**. However, after discussing this issue with our client, we learned that the missing values of **PAY_1** were due to a reporting issue that they are working on correcting. In the near-term, if there is a method available that can enable the inclusion of the accounts with missing **PAY_1** information in the model prediction process, it would be preferable. So, we need to consider how we could make predictions for accounts that are missing this feature.

Solving the problem of missing data is a core skill for data scientists as this is a common issue that occurs in real-world, "dirty" datasets. Instead of ignoring the samples with missing data, another option is to use some method to fill in missing values of affected features. This filling in is known as **imputation**. The methods for working with missing data range from simple to more complex, as shown in *Figure 6.2*.

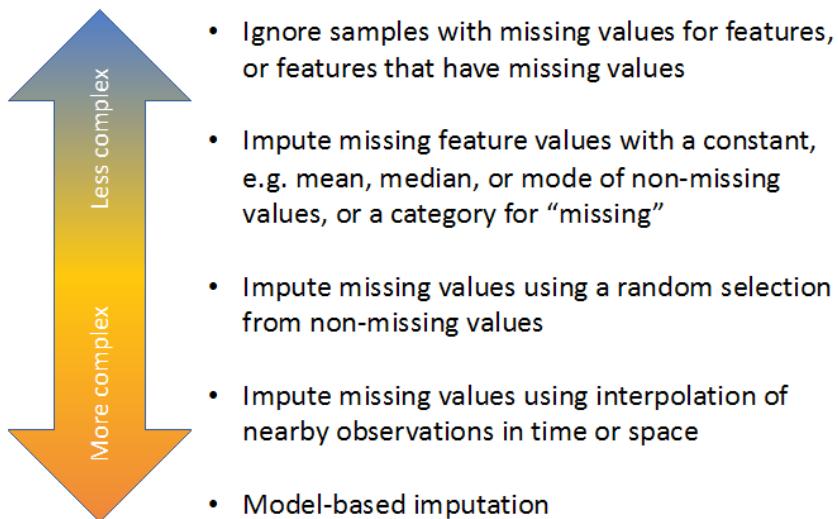


Figure 6.2: Strategies for imputing missing values of features

The simplest way to deal with samples that have missing values for certain features is to remove these samples from the dataset or to remove the features that have the missing values. While this approach allows you to immediately proceed, it has the undesirable effect of discarding potentially useful data from the dataset. For the case study data, you've learned that we will need to create predictions for samples even if the values of the features are missing, so we will have to include these data in some way.

As an alternative to throwing away data due to missing values of features, you can try to employ some kind of imputation strategy. Imputation involves using the known values of a certain feature to make a best educated guess as to the missing values. The simplest types of imputation involve using a summary statistic of the non-missing feature values, as the single **constant** value with which to replace all the missing values. This summary statistic may be the mean, median, or mode for continuous features. For categorial features, the mode is an option, as well as the median for ordinal categoricals.

An important additional case of imputation with a constant value for categorial variables is to *create a new level of the categorical variable, to indicate data is missing*. For example, if the **EDUCATION** feature had missing data, we could create a new level indicating that **EDUCATION** data was "missing," in addition to the existing levels of the **EDUCATION** feature. This may be the best approach when missing data for a feature indicates something about the response variable. In this case, the fact that values are missing represents important information that should not be replaced with other values for a feature. To make a decision on which method to use for imputation with a constant value, you can try cross-validation just like you would for the choice of model hyperparameters.

More sophisticated methods of imputation can be used to fill in missing values with non-constant values. These methods reflect an acknowledgment that the missing values may not all be equal. The simplest way to do this is to fill in missing values using a **random** draw from the set of non-missing values, with replacement. This way, the relative frequency of different values chosen to fill in missing data will be similar to the existing data for that feature.

However, the random imputation method doesn't use any other information from a given sample with missing data, when choosing the fill values. In case samples are located within time or space, such as a time series or geolocated data, then temporal or spatial **interpolation** methods may be used. These methods follow the general idea that a missing data point is probably located somewhere between the values of adjacent data points in time or space. There is a rich set of interpolation methods available in NumPy and Pandas that you may wish to explore if you are working with missing data for time series or geolocated data.

Lastly, perhaps the most sophisticated way to fill in missing data is to view the imputation problem as a predictive modeling "problem within a problem." In this method, the feature with missing values is considered the response variable, while the features with no missing values are considered the features of this sub-problem. You can take the samples with known values for all features, split them up in to training and testing sets similar to how you would with any other supervised learning problem, and develop a predictive model for the feature with missing values. Then, you can use this model to predict the unknown values of the feature. Because this method relies on predictive modeling, we'll refer to it as **model-based imputation**. We will use this method later on, with the case study data, to clarify how it works.

Note that for our case study dataset, the feature that has missing values is **PAY_1**, which turned out to be the most important feature as identified by both the univariate feature selection as well as the feature importance of the random forest. So, the quality of the imputation strategy we implement for this feature may be expected to have a more significant impact on the performance of the model than if we were imputing a less important feature. Therefore, we will explore the full range of imputation options available.

Preparing Samples with Missing Data

In order to test out different imputation strategies on the case study data and see how they affect the predictive capability of the modeling approach we will use, we need to replace samples with missing data back in to the dataset. In order to do this, we need to use the same data cleaning procedures we used in *Chapter 1, Data Exploration and Cleaning*; we shall perform them in the following exercise.

Exercise 22: Cleaning the Dataset

In this exercise, we will be cleaning our dataset to address the missing data entries. We will use the same approach as that in *Chapter 1, Data Exploration and Cleaning*. Perform the following steps to complete the exercise:

Note

For Exercises 22–25 and Activity 6, the code and the resulting output have been loaded in a Jupyter Notebook that can be found here: <http://bit.ly/2PtdRyj>. You can scroll to the appropriate section within the Jupyter Notebook to locate the exercise or activity of choice.

1. Load the original dataset from the beginning of our exploration, before any cleaning takes place:

```
df_orig = pd.read_excel('..../Data/default_of_credit_card_clients__courseware_version_1_21_19.xls')
```

Now, we need to repeat all the data cleaning steps we took, with the exception of removing the samples where the **PAY_1** feature had missing values. As a first step, we identify and drop any samples where the values of all the features are equal to 0. We had determined that this was an effective way to remove duplicate account IDs from the dataset.

2. Make a Boolean array indicating where entries of the **DataFrame** are equal to 0:

```
df_zero_mask = df_orig == 0
```

3. Collapse this 2-dimensional Boolean array to 1 dimension by indicating which rows have 0s in all columns, starting with the second column. This array tells which rows have a value of 0 for all the features and need to be removed:

```
feature_zero_mask = df_zero_mask.iloc[:, 1: ].all(axis=1)
```

4. Check that the number of rows with all 0 values for the features is the same as what we found in *Chapter 1, Data Exploration and Cleaning*:

```
sum(feature_zero_mask)
```

The output should be:

315

Figure 6.3: How many rows have a value of 0 for all features

This is the same as that in *Chapter 1, Data Exploration and Cleaning*.

5. Use this mask to select all the other rows, that is, those that don't have values of zero for all features, and check the shape of the resulting DataFrame:

```
df_clean = df_orig.loc[~feature_zero_mask, : ].copy()  
df_clean.shape
```

You should obtain the following output:

(29685, 25)

Figure 6.4: The shape of a cleaned DataFrame

In addition to getting rid of clearly invalid data (samples with all features equal to 0), the goal of this procedure was to eliminate duplicate account IDs.

6. Check that the number of unique account IDs is equal to the number of rows of the cleaned **DataFrame**:

```
df_clean['ID'].nunique()
```

The output should be as follows:

29685

Figure 6.5: The number of unique account IDs in cleaned data

This is the same as the number of rows of the DataFrame, indicating the data has been cleansed of duplicated account IDs.

7. Replace the undocumented values of **EDUCATION** and **MARRIAGE** features with the documented value for "unknown." We simply repeat the code from Chapter 1, *Data Exploration and Cleaning* that does this:

```
df_clean['EDUCATION'].replace(to_replace=[0, 5, 6], value=4, inplace=True)  
df_clean['MARRIAGE'].replace(to_replace=0, value=3, inplace=True)
```

The data is now in a cleaned state. The question at this point is how to include the samples with missing values for **PAY_1**, with the rest of the data.

We have reserved an unseen test set from all modeling activities so far. The samples in this dataset are only to be used after we have selected our final model and wish to see if the out-of-sample performance that we estimated with cross-validation is reflected in the new data. So, it would be best to continue to keep these samples isolated from the modeling work, including the selection of an imputation strategy. In order to do this, we will subset the samples from our cleaned dataset **df_clean** that have missing values for **PAY_1** and add them to the training and testing sets later, so that the proportion of training and testing data remains the same.

8. Recall what the values of **PAY_1** look like:

```
df_clean['PAY_1'].value_counts()
```

The output should be:

0	13087
-1	5047
1	3261
Not available	3021
-2	2476
2	2378
3	292
4	63
5	23
8	17
6	11
7	9

Figure 6.6: Values of **PAY_1**

Before we proceed to isolate samples where the value of **PAY_1** is missing, first note that the mode of non-missing values of **PAY_1**, in other words the most prevalent value, is 0. We will use this fact later when trying different imputation strategies.

The missing values in **PAY_1** are indicated by the string '**Not available**'.

9. Create a Boolean mask to identify the rows with this value:

```
missing_pay_1_mask = df_clean['PAY_1'] == 'Not available'
```

10. Confirm that the number of samples with missing data is 3,021 as we observed in *Chapter 1, Data Exploration and Cleaning*:

```
sum(missing_pay_1_mask)
```

The output should be as follows:

3021

Figure 6.7: The number of rows with missing values for **PAY_1**

The number of rows with missing data is as expected.

11. Make a copy of these rows in a new **DataFrame** which we can add to the data we've been using for modeling:

```
df_missing_pay_1 = df_clean.loc[missing_pay_1_mask, :].copy()
```

12. Load the cleaned data that we've been working with in Chapters 2 through 5. This data has had the 3,021 samples with a missing value of **PAY_1** already removed from it, so it is an entirely different set of samples than what we've got stored in **df_missing_pay_1**:

Note

The path to your cleaned data file may be different.

```
df = pd.read_csv('../Data/Chapter_1_cleaned_data.csv')
```

13. Make a list of the column names of this **DataFrame** and then remove the strings that are not part of the set of features and the response variable. Here are the column names:

```
df.columns
```

```
Index(['ID', 'LIMIT_BAL', 'SEX', 'EDUCATION', 'MARRIAGE', 'AGE', 'PAY_1',
       'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6', 'BILL_AMT1', 'BILL_AMT2',
       'BILL_AMT3', 'BILL_AMT4', 'BILL_AMT5', 'BILL_AMT6', 'PAY_AMT1',
       'PAY_AMT2', 'PAY_AMT3', 'PAY_AMT4', 'PAY_AMT5', 'PAY_AMT6',
       'default payment next month', 'EDUCATION_CAT', 'graduate school',
       'high school', 'none', 'others', 'university'],
      dtype='object')
```

Figure 6.8: Column names from the cleaned DataFrame

14. Assign the column names to a list:

```
features_response = df.columns.tolist()
```

15. Then, make a list of column names that are not the features or the response, so we can remove them:

```
items_to_remove = ['ID', 'SEX', 'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5',
                  'PAY_6',
                  'EDUCATION_CAT', 'graduate school', 'high school',
                  'none',
                  'others', 'university']
```

16. Use a list comprehension to remove the unwanted column names and display the result:

```
features_response = [item for item in features_response if item not in
                      items_to_remove]
features_response
```

The output from this should be:

```
['LIMIT_BAL',
 'EDUCATION',
 'MARRIAGE',
 'AGE',
 'PAY_1',
 'BILL_AMT1',
 'BILL_AMT2',
 'BILL_AMT3',
 'BILL_AMT4',
 'BILL_AMT5',
 'BILL_AMT6',
 'PAY_AMT1',
 'PAY_AMT2',
 'PAY_AMT3',
 'PAY_AMT4',
 'PAY_AMT5',
 'PAY_AMT6',
 'default payment next month']
```

Figure 6.9: Names of features and response

This list of column names will come in handy when we are selecting data from the DataFrames including missing and non-missing values for **PAY_1**, in order to combine them and test different imputation strategies, which will be done in the following exercise.

Exercise 23: Mode and Random Imputation of PAY_1

In this example, we will try some of the simpler imputation strategies available for **PAY_1** and see their effects on cross-validation performance. The first steps will be to append the samples with missing values for **PAY_1** to the end of the testing set we've been working with, that has non-missing **PAY_1**. We'll need to shuffle this when performing cross-validation so that the samples with missing **PAY_1** don't all wind up in the same fold, which would create a situation where data in one of the folds was different than the others.

Although **PAY_1** has numeric values, it's a sort of hybrid between a categorical and a numerical feature as we discussed previously. We will basically treat it as categorical for the purposes of imputation, since if we were to take an average of the non-missing values of **PAY_1**, for example, the result might not be an integer, which would not be interpretable given the definition of **PAY_1**. Therefore, the simplest imputation strategies we have available are mode and median. A slightly more complex one is a random selection from the samples with non-missing **PAY_1**. We'll explore these here. Perform the following steps to complete the exercise:

Note

The code and the resulting output for this exercise has been loaded in a Jupyter Notebook that can be found here: <http://bit.ly/2PtdRyj>.

1. Import the **train_test_split** class from **sklearn**, so we can work with the same held-out test set we have been:

```
from sklearn.model_selection import train_test_split
```

2. Create the 80/20 training/testing split using the same random seed we've been working with:

```
X_train, X_test, y_train, y_test = \
train_test_split(df[features_response[:-1]].values, df['default payment
next month'].values,
test_size=0.2, random_state=24)
```

3. Examine the shapes of the training and testing sets:

```
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

The output should be as follows:

```
(21331, 17)
(5333, 17)
(21331,)
(5333,)
```

Figure 6.10: Shapes of training and testing sets with non-missing PAY_1

4. Create the imputation values for **PAY_1**. First check that we know which index **PAY_1** is in the list of features and response names:

```
features_response[4]
```

The output should be:

```
'PAY_1'
```

Figure 6.11: Confirming the index of **PAY_1**

We'll need this index to access the non-missing values of **PAY_1** in the array of training features.

5. Find the values of **PAY_1** we may use for imputation. First check, what the median of **PAY_1** is using the following code:

```
np.median(X_train[:,4])
```

The output should be:

```
0.0
```

Figure 6.12: Median of non-missing values of **PAY_1**

We see that the median of the non-missing values of **PAY_1** is 0. Recall from our **value_counts** of **PAY_1** above, that the mode of non-missing **PAY_1** is also 0.

Since the median and mode are the same, we only have two imputation strategies to test in this exercise: the mode/median and a random selection of non-missing.

6. Create a list of values to test for imputation. We can use a single numerical value of 0 representing the median and mode, as well as use NumPy's **random.choice** function to take a random sample of non-missing **PAY_1**:

```
np.random.seed(seed=1)
fill_values = [0, np.random.choice(X_train[:,4], size=(3021,), replace=True)]
```

Notice here, for the random sample, that we've seeded the random number generator for consistent results across runs, indicated the array to take the random selection from (the non-missing values of **PAY_1** in the training data: **X_train[:,4]**), indicated the size of the random sample to be the same length as the column of missing **PAY_1** values in **df_missing_pay_1** (3,021), and that we would like to sample with replacement.

7. Create a list of names for the imputation strategies to help keep track of them:

```
fill_strategy = ['mode', 'random']
```

8. Examine the second element of the **fill_values** list, which is the array of random selections of **PAY_1**:

```
fill_values[-1]
```

The output should be:

```
array([ 0,  0,  0, ...,  2,  0, -2])
```

Figure 6.13: Randomly imputed values of **PAY_1**

The output looks as expected; these values are integers in the range [-2, 8] as we know **PAY_1** is. However, it would be better to view a summary graphic of all the imputed values, to compare them to the distribution of **PAY_1**. This would allow us to confirm the distributions are the same, as is intended by random imputation done in this way.

9. Use histograms to examine the distributions of the original non-missing **PAY_1** feature in the training set and the randomly selected imputed values:

```
fig, axs = plt.subplots(1, 2, figsize=(8, 3))
bin_edges = np.arange(-2, 9)
axs[0].hist(X_train[:, 4], bins=bin_edges, align='left')
axs[0].set_xticks(bin_edges)
axs[0].set_title('Non-missing values of PAY_1')
axs[1].hist(fill_values[-1], bins=bin_edges, align='left')
axs[1].set_xticks(bin_edges)
axs[1].set_title('Random selection for imputation')
plt.tight_layout()
```

This code should produce the following graph:

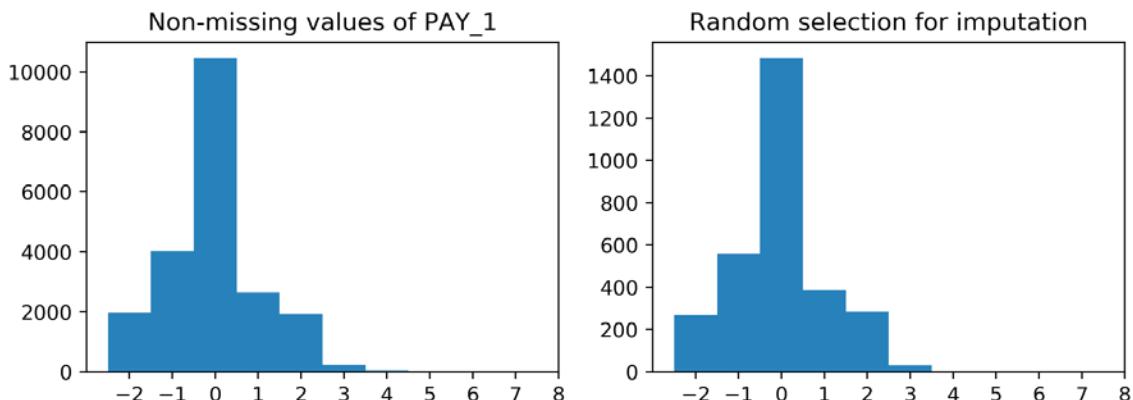


Figure 6.14: Distributions of PAY_1 and randomly imputed values

These two distributions look very similar. Only the scale of the y-axis indicates that there are fewer imputed values than in the original dataset. This shows that we have selected values for **PAY_1** that faithfully imitate the relative frequency of different values of this feature in the data. Now, we are ready to set up the cross-validation we can use to compare imputation methods.

We do not need to do a cross-validation search for hyperparameters here, as we did with the scikit-learn function **GridSearchCV**. At this point, we already know that random forest is the model we will use, and we know what hyperparameters we should use. We just want to perform cross-validation with one set of hyperparameters, in order to have several estimates of the out-of-sample testing score. For this, we can use a similar, but simpler class called **cross_validate**, in combination with the **KFold** class. These classes can help us carry out cross-validation like **GridSearchCV**, but do not involve a search over a grid of hyperparameters.

10. Import the **KFold** class with this code:

```
from sklearn.model_selection import KFold
```

-
11. Instantiate the **KFold** class as follows:

```
k_folds = KFold(n_splits=4, shuffle=True, random_state=1)
```

Here, we have specified to use four folds over the training set, as we have done previously when doing cross-validation. We also say that we would like to shuffle the data before splitting it into folds. This is important because we will append the samples with imputed values to the end of the training features and response arrays. However, when we perform cross-validation, we'd like these samples to be "mixed up" throughout the four folds so that each fold has some imputed values in it. We also set the random seed for repeatability.

12. Import the **cross_validate** class:

```
from sklearn.model_selection import cross_validate
```

13. Import the random forest classifier class:

```
from sklearn.ensemble import RandomForestClassifier
```

14. Instantiate the random forest class, using the hyperparameters we've determined are best for the case study data: 200 trees with maximum depth 9:

```
rf = RandomForestClassifier(\n    n_estimators=200, criterion='gini', max_depth=9,\n    min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0,\n    max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0,\n    min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=None,\n    random_state=4, verbose=1, warm_start=False, class_weight=None)
```

While we are working with slightly different data, due to the imputed values, we will assume that the hyperparameters we determined for the random forest are still the right ones to use. Now, we will embark on the cross-validation. This is a long code cell, so we will split it up over several steps.

15. Open a **for** loop. This loop will work over the different imputation strategies we have, of which there are two:

```
for counter in range(len(fill_values)):
```

16. Inside the for loop, the first step is to create a copy of the **DataFrame** we have of samples with missing values of **PAY_1** and then fill in these missing values with the imputation strategy under consideration:

```
#Copy the data frame with missing PAY_1 and assign imputed values
df_fill_pay_1_filled = df_missing_pay_1.copy()
df_fill_pay_1_filled['PAY_1'] = fill_values[counter]
```

17. Split the imputed data in to training and testing sets. We are only going to use the training portion here, for the cross-validation. This works the same way as when we split the cleaned data with missing values removed:

```
#Split imputed data in to training and testing, using the same
#80/20 split we have used for the data with non-missing PAY_1
X_fill_pay_1_train, X_fill_pay_1_test, y_fill_pay_1_train, y_fill_pay_1_
test = \
train_test_split(
    df_fill_pay_1_filled[features_response[:-1]].values,
    df_fill_pay_1_filled['default payment next month'].values,
    test_size=0.2, random_state=24)
```

18. Now, we want to combine the imputed data with the non-missing **PAY_1** data that we've been working with. We concatenate these together:

```
#Concatenate the imputed data with the array of non-missing data
X_train_all = np.concatenate((X_train, X_fill_pay_1_train), axis=0)
y_train_all = np.concatenate((y_train, y_fill_pay_1_train), axis=0)
```

19. Finally, we put everything together. We use this data with the **cross_validation** procedure, as well as with the **KFolds** splitter and a random forest model. Print the average testing score, with standard deviation, for each method.

```
#Use the KFolds splitter and the random forest model to get
#4-fold cross-validation scores for both imputation methods
imputation_compare_cv = cross_validate(rf, X_train_all, y_train_all,
scoring='roc_auc',
cv=k_folds, n_jobs=-1, verbose=1,
return_train_score=True, return_estimator=True,
error_score='raise-deprecating')

test_score = imputation_compare_cv['test_score']
print(fill_strategy[counter] + ' imputation: ' +
'mean testing score ' + str(np.mean(test_score)) +
', std ' + str(np.std(test_score)))
```

The output of the for loop, after it is all done, should look something like this:

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  2 out of  4 | elapsed:  11.7s remaining:  11.7s
[Parallel(n_jobs=-1)]: Done  4 out of  4 | elapsed:  11.8s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.

mode imputation: mean testing score 0.772866246168149, std 0.0031479941297533737
random imputation: mean testing score 0.7692540439833129, std 0.003660875187678248

[Parallel(n_jobs=-1)]: Done  2 out of  4 | elapsed:  19.9s remaining:  19.9s
[Parallel(n_jobs=-1)]: Done  4 out of  4 | elapsed:  19.9s finished
```

Figure 6.15: The output of the cross-validation loop that tests different imputation strategies

This is the step where the model fitting and testing of cross-validation is conducted, behind the scenes and abstracted away from our view by the **cross_validation** class. Notice that we supply a lot of information to this class: the model we are cross-validating (**rf**), the data **X_train_all** and **y_train_all**, and the **KFolds** splitter, among other options. One of these options is **n_jobs=-1**, indicating to run jobs in parallel to make things go more quickly.

The printed output here indicates that these two imputation strategies have very similar performances. However, the simplest imputation strategy, just filling in all missing values of **PAY_1** with the most common value of 0 from the non-missing samples, performs the best: the average testing score (ROC AUC) across the four folds is 0.773 for mode imputation, versus 0.769 using random imputation.

What about the average testing score here, where we've included the samples with missing data, versus the average testing score we observed when only using the data where all the values of the features were known? That was 0.776. So, we have experienced a little degradation in the estimate of out-of-sample model testing performance. This is to be expected, as we are now including data that has issues and uncertainties. However, this is still within 0.006 ROC AUC units of the best score we observed when working with the non-missing data, so it's not too much lower. We should be confident that we have found an acceptable way to make predictions for accounts that have missing values for **PAY_1**, as the client requested. Is it possible to increase the quality of prediction with imputed data more than this? We will explore if we can do this in the following section.

A Predictive Model for PAY_1

The most accurate, but also the most labor-intensive way to impute a feature with missing values is to create a predictive model for that feature. You can think of this as a natural extension of simpler methods, using progressively more information from the dataset when imputing the missing values:

- The simplest methods like mean, median, mode, and random selection just use information from non-missing values of the feature.
- Intermediately complex methods like interpolation use additional information about how "close" a given sample is to other samples, in the spatial or temporal context, and leverage this for more accurate imputation.
- Finally, model-based imputation additionally uses all features without missing values, to predict the missing values of the affected feature. Therefore, model-based imputation is similar to any other predictive model, where we consider the feature with missing values to be the response variable.

As we consider how to create a predictive model for **PAY_1**, think about how this model would be similar to, or different from, other models we've considered.

The different types of supervised machine learning models are regression and classification. As we discussed earlier, we would not consider **PAY_1** to be a numerical feature. Taking an average for imputation, which may lead to a decimal value, will not make sense according to the definition of this feature. We will face this same issue if we tried to model **PAY_1** on a continuous scale, as would be done using regression algorithms. Therefore, when considering what kind of supervised learning model to use for **PAY_1**, we know it should be a classification model.

However, the classification model for **PAY_1** will be different than the classification model for credit account default that we have built for the case study data. This is because, unlike the credit default problem, where the response variable can only take on two values (account defaults or account does not default), there are more than two values for **PAY_1**. These include all the levels of this variable from -2 through 8. Therefore, as opposed to a binary classification problem, the model for **PAY_1** will be a multiclass classification problem.

We will walk through the steps to set up and train this multiclass classification problem. Aside from a few key differences, you will see the process is very similar to creating a binary classification model.

First, create a copy of the cleaned **DataFrame** with non-missing values of **PAY_1**.

```
pay_1_df = df.copy()
```

This represents the data available for training and testing, since we know the value of the response variable, **PAY_1** in this case, for all of these samples.

Similar to our previous approach for selecting columns, we will make a list of column names that will be features.

```
features_for_imputation = pay_1_df.columns.tolist()
```

We subset this down to the list of features for the imputation model by removing the names of all metadata, as well as '**default payment next month**' which we don't need here and '**PAY_1**' which is the response variable for this problem:

```
items_to_remove = ['ID', 'SEX', 'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6',
                   'EDUCATION_CAT', 'graduate school', 'high school',
                   'none',
                   'others', 'university', 'default payment next month',
                   'PAY_1']
```

Remove these items using a list comprehension:

```
features_for_imputation = [item for item in features_for_imputation if item
                           not in items_to_remove]
features_for_imputation
```

The output should be only the features besides **PAY_1**:

```
[ 'LIMIT_BAL',
  'EDUCATION',
  'MARRIAGE',
  'AGE',
  'BILL_AMT1',
  'BILL_AMT2',
  'BILL_AMT3',
  'BILL_AMT4',
  'BILL_AMT5',
  'BILL_AMT6',
  'PAY_AMT1',
  'PAY_AMT2',
  'PAY_AMT3',
  'PAY_AMT4',
  'PAY_AMT5',
  'PAY_AMT6' ]
```

Figure 6.16: Features for the predictive model of PAY_1

Now that we've prepared the variable names for the imputation model, we'll go through the remaining steps to build this model as an exercise.

Exercise 24: Building a Multiclass Classification Model for Imputation

In this exercise, we will build a multi-class model for imputation. When building this model, we will assume that random forest is the best model to use, partly because of our experiences with this dataset, and also because scikit-learn's random forest algorithm readily supports multiclass classification. There are various ways to use logistic regression and other binary classification algorithms for multiclass classification. However, we won't go in to those details here.

Note

Two approaches which you may wish to learn more about, if you want more background in multiclass classification, are one versus all and one versus one (<https://scikit-learn.org/stable/modules/multiclass.html>). These approaches enable binary classification models, such as logistic regression, to be used for multiclass classification.

We will follow the usual steps of making a train/test split, doing a cross-validation parameter search using the training data, and confirming model performance on the test set. Most of this should be familiar by now, however here there is the added twist that we are modeling a multiclass problem. We'll choose accuracy as our model performance metric, since it's a little easier to use with multiclass problems. However, there are ways to extend threshold-based metrics like ROC AUC to multiclass settings.

Note

For more information on using ROC AUC for multiclass classification, refer to the following: https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html.

Perform the following steps to complete the exercise:

Note

The code and the resulting output for this exercise have been loaded in a Jupyter Notebook that can be found here: <http://bit.ly/2PtdRyj>.

1. Make a training and testing split of the imputation model data:

```
X_impute_train, X_impute_test, y_impute_train, y_impute_test = \
train_test_split(
    pay_1_df[features_for_imputation].values,
    pay_1_df['PAY_1'].values,
    test_size=0.2, random_state=24)
```

Here, **PAY_1** is the response variable.

2. Select a grid of hyperparameters to search using cross-validation:

```
rf_impute_params = {'max_depth':[3, 6, 9, 12],
                     'n_estimators':[10, 50, 100, 200]}
```

These are the same ones we used for the credit default model.

3. Import and instantiate the **GridSearchCV** class:

```
from sklearn.model_selection import GridSearchCV
cv_rf_impute = GridSearchCV(rf, param_grid=rf_impute_params,
                           scoring='accuracy',
                           fit_params=None, n_jobs=-1, iid=False, refit=True,
                           cv=4, verbose=2, error_score=np.nan, return_train_score=True)
```

Notice here that we are re-using the random forest model instance **rf** from the credit account default model. All the options should be at their defaults except **n_estimators** and **max_depth**, which will be varied here through the grid-search process.

When we instantiate the **GridSearchCV** class, we do so in a similar way to before, but here we use the accuracy method as noted earlier. We also allow for parallel processing with **n_jobs=-1**.

4. Run the grid search with this code:

```
cv_rf_impute.fit(X_impute_train, y_impute_train)
```

The output should be:

```
Fitting 4 folds for each of 16 candidates, totalling 64 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done 33 tasks      | elapsed:  21.8s
[Parallel(n_jobs=-1)]: Done 64 out of 64 | elapsed:  1.0min finished
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed:     5.1s finished

GridSearchCV(cv=4, error_score=nan,
            estimator=RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                max_depth=9, max_features='auto', max_leaf_nodes=None,
                min_impurity_decrease=0.0, min_impurity_split=None,
                min_samples_leaf=1, min_samples_split=2,
                min_weight_fraction_leaf=0.0, n_estimators=200, n_jobs=None,
                oob_score=False, random_state=4, verbose=1, warm_start=False),
            fit_params=None, iid=False, n_jobs=-1,
            param_grid={'max_depth': [3, 6, 9, 12], 'n_estimators': [10, 50, 100, 200]},
            pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
            scoring='accuracy', verbose=2)
```

Figure 6.17: The output of grid-search CV for the imputation model

Notice that we have not had to do anything special here to account for the fact that we are fitting a multiclass classification model. That's because scikit-learn's implementation of random forest automatically handles multiclass data without any preparation necessary.

5. Observe the hyperparameters from the best model of cross-validation:

```
cv_rf_impute.best_params_
```

The output should be

```
{'max_depth': 12, 'n_estimators': 100}
```

Figure 6.18: Optimal hyperparameters from cross-validation of the imputation model

6. See what the accuracy score of the best model is:

```
cv_rf_impute.best_score_
```

The output should be:

0.7337676389523727

Figure 6.19: Best average cross-validated accuracy score

How can we interpret this accuracy score? Consider the imputation method that uses the mode of non-missing values of **PAY_1**, which is 0, for all missing values. What would the accuracy of mode imputation be?

7. Examine the value counts of **PAY_1** in **pay_1_df** to see the relative frequency of different values:

```
pay_1_value_counts = pay_1_df['PAY_1'].value_counts().sort_index()  
pay_1_value_counts
```

The output should be:

-2	2476
-1	5047
0	13087
1	3261
2	2378
3	292
4	63
5	23
6	11
7	9
8	17

Figure 6.20: Value counts of **PAY_1**

8. Show the relative frequency of **PAY_1** values as fractions as follows:

```
pay_1_value_counts/pay_1_value_counts.sum()
```

The output should be:

-2	0.092859
-1	0.189281
0	0.490812
1	0.122300
2	0.089184
3	0.010951
4	0.002363
5	0.000863
6	0.000413
7	0.000338
8	0.000638

Figure 6.21: The fractions of values of **PAY_1**

We can see that the mode is, obviously, the most prevalent value in **PAY_1**. We can also see from the fractions in Figure 6.21 that 49% of the values are equal to the mode. So, using mode imputation, the imputation value would be expected to be correct about 49% of the time. However, using a model to impute the values of **PAY_1**, our results in step 6 indicate we can be correct 73% of the time. So, one benefit of using model-based imputation is that we can make more accurate fill values for the unknown values of a feature.

9. To check that the cross-validation accuracy generalizes to the test set, we need to make predictions on the test set:

```
y_impute_predict = cv_rf_impute.predict(X_impute_test)
```

10. Import the **accuracy_score** class and see the accuracy of testing predictions:

```
from sklearn import metrics  
metrics.accuracy_score(y_impute_test, y_impute_predict)
```

The output should be:

0.7387961747609225

Figure 6.22: The accuracy of imputation model on a held-out test set

The accuracy on the test set of 74% is comparable to, and a little higher than, the average cross-validation accuracy of 73%.

It would be good to check the distribution of predicted classes from this model, similar to how we did for the random imputation method. We know the model is likely to be about 73% accurate, but it is always good to visualize model predictions.

11. Use this code to plot actual values and predictions for the held-out test set:

```
fig, axs = plt.subplots(1, 2, figsize=(8,3))  
axs[0].hist(y_impute_test, bins=bin_edges, align='left')  
axs[0].set_xticks(bin_edges)  
axs[0].set_title('Non-missing values of PAY_1')  
axs[1].hist(y_impute_predict, bins=bin_edges, align='left')  
axs[1].set_xticks(bin_edges)  
axs[1].set_title('Model-based imputation')  
plt.tight_layout()
```

The resulting graphic should appear as follows:

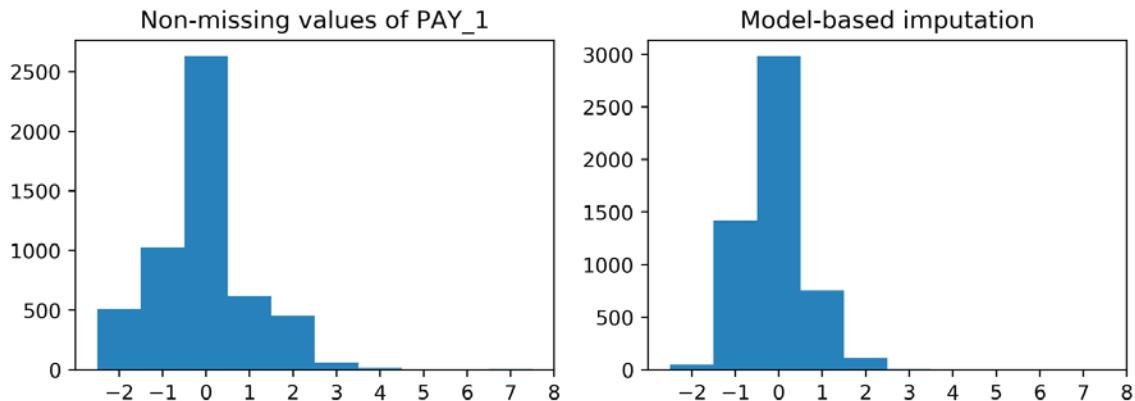


Figure 6.23: Actual and predicted values of PAY_1 for the held-out test set

Figure 6.23 shows that the imputation model is more likely to predict a value of 0 for PAY_1 than it should be. However, it does predict other values and consequently is more accurate than the mode imputation.

So, our imputation model has been tested and the selected hyperparameters are confirmed. After the held-out test set has been used to check model performance on unseen data, all the training and testing data should be put together, to train a model on as much data as possible. Increasing the amount of training data typically results in a better model.

12. Collect all the values with known PAY_1 to train the final version of the imputation model:

```
X_impute_all = pay_1_df[features_for_imputation].values
y_impute_all = pay_1_df['PAY_1'].values
```

13. Define a random forest model with the optimal hyperparameters for imputation:

```
rf_impute = RandomForestClassifier(n_estimators=100, max_depth=12)
```

14. Fit the imputation model on all available data:

```
rf_impute.fit(X_impute_all, y_impute_all)
```

The output should be:

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                      max_depth=12, max_features='auto', max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=None,
                      oob_score=False, random_state=None, verbose=0,
                      warm_start=False)
```

Figure 6.24: Output from training the imputation model

This model is now ready to be used for imputation.

Using the Imputation Model and Comparing it to Other Methods

We created the model for model-based imputation, so we may now use model-imputed values for **PAY_1** in cross-validation with the credit account default model and see how the performance is, in comparison to the simpler imputation methods we already tried. To start, make a copy of the **DataFrame** with missing **PAY_1**:

```
df_fill_pay_1_model = df_missing_pay_1.copy()
```

Examine the values of **PAY_1**, to confirm they represent missing data:

```
df_fill_pay_1_model['PAY_1'].head()
```

The output should be:

```
17      Not available
28      Not available
29      Not available
54      Not available
60      Not available
Name: PAY_1, dtype: object
```

Figure 6.25: Missing data for PAY_1

Now, replace these missing values with imputation model predictions:

```
df_fill_pay_1_model['PAY_1'] = rf_impute.predict(df_fill_pay_1_model[features_for_imputation].values)
```

Examine the values of **PAY_1** now, to confirm they've been imputed:

```
df_fill_pay_1_model['PAY_1'].head()
```

The output should now be:

```
17      0
28     -1
29      0
54      0
60      0
Name: PAY_1, dtype: int64
```

Figure 6.26: Model-imputed values of PAY_1

This shows the values were imputed as desired. We can see from the output in figure 6.26 that not all the values are the same, as we'd expect. We can make a quick examination of the predictions using the **value_counts** method:

```
df_fill_pay_1_model['PAY_1'].value_counts().sort_index()
```

The output should be:

```
-2        30
-1      763
0      1715
1       438
2        64
3         7
4         2
6         1
8         1
```

Figure 6.27: Predictions of the imputation model

From Figure 6.27, we can see that there is a range of prediction values of **PAY_1** from the imputation model, roughly in the relative frequencies that we expect. Now, we need to take an 80% sample of these to combine with the model training data for the case study, to examine how this imputation method affects model performance, as we did for mode and random imputation.

Split the model-imputed data as follows:

```
X_fill_pay_1_train, X_fill_pay_1_test, y_fill_pay_1_train, y_fill_pay_1_test = \
train_test_split(
    df_fill_pay_1_model[features_response[:-1]].values,
    df_fill_pay_1_model['default payment next month'].values,
    test_size=0.2, random_state=24)
```

Combine this with the data with known PAY_1 as before:

```
X_train_all = np.concatenate((X_train, X_fill_pay_1_train), axis=0)
y_train_all = np.concatenate((y_train, y_fill_pay_1_train), axis=0)
```

Confirm that the `rf` object still holds the model with optimal hyperparameters for the case study problem:

rf

The output should be:

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                      max_depth=9, max_features='auto', max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, n_estimators=200, n_jobs=None,
                      oob_score=False, random_state=4, verbose=1, warm_start=False)
```

Figure 6.28: Confirming the model for the case study

The `rf` object holds a random forest classifier with 200 trees and maximum depth of 9, as expected. Now, use this model with the combined imputed and non-missing training data in cross-validation as we did with the other imputation methods, to test model-based imputation:

```
imputation_compare_cv = cross_validate(rf, X_train_all, y_train_all,
scoring='roc_auc',
cv=k_folds, n_jobs=-1, verbose=1,
return_train_score=True, return_estimator=True,
error_score='raise-deprecating')
```

When this process completes, you can examine the average model testing score across the four folds as follows:

```
np.mean(imputation_compare_cv['test_score'])
```

The output should be:

```
0.7726757126815554
```

Figure 6.29: Average cross-validated testing score using model-imputed PAY_1

How does the result of model-based imputation compare with the simpler mode and random methods? An average cross-validated ROC AUC here of 0.7727 is very close to, but slightly worse than, our previous best result of 0.7729 from mode-based imputation.

For all intents and purposes, mode-based and model-based imputations have essentially the same performance. Note that the standard deviation for cross-validation score with mode-based imputation calculated above is 0.003, so the average model-based imputation score is within one standard deviation of the best method. Which should we use?

From a simplicity standpoint, the mode-based imputation is the clear favorite. The code to implement this is much more concise than the model-based imputation we've just completed. The only reason to go with model-based imputation would be if the client knows we could also make predictions for PAY_1 and would like to have them. We consult with our client about this and learn that they are only interested in the predictions of credit default. So, we move forward with the simpler imputation method.

First, reassign the filled values using imputation with the mode of 0:

```
df_fill_pay_1_model['PAY_1'] = np.zeros_like(df_fill_pay_1_model['PAY_1']).values
```

The **zeros_like** function of Numpy creates an array of zeros in the same shape as the input array. Check that this worked:

```
df_fill_pay_1_model['PAY_1'].unique()
```

The output should be:

```
array([0])
```

Figure 6.30: Confirm mode imputation

There should only be 0 values present from the mode-based imputation, and we can see in *Figure 6.30* that this has worked, since there is only 1 unique value for **PAY_1** and it is 0. Now, we repeat the splitting of the imputed data:

```
X_fill_pay_1_train, X_fill_pay_1_test, y_fill_pay_1_train, y_fill_pay_1_test = \
train_test_split(
    df_fill_pay_1_model[features_response[:-1]].values,
    df_fill_pay_1_model['default payment next month'].values,
    test_size=0.2, random_state=24)
```

and combined with the non-missing data, this time including the unseen test set which we'll use shortly:

```
X_train_all = np.concatenate((X_train, X_fill_pay_1_train), axis=0)
X_test_all = np.concatenate((X_test, X_fill_pay_1_test), axis=0)
y_train_all = np.concatenate((y_train, y_fill_pay_1_train), axis=0)
y_test_all = np.concatenate((y_test, y_fill_pay_1_test), axis=0)
```

Just to confirm that the data has been restored to the state it was in when we tested mode-based imputation earlier, we run the cross-validation to confirm that we get the same score:

```
imputation_compare_cv = cross_validate(rf, X_train_all, y_train_all,
scoring='roc_auc',
cv=k_folds, n_jobs=-1, verbose=1,
return_train_score=True, return_estimator=True,
error_score='raise-deprecating')
```

After running the cross-validation, compute the average score:

```
np.mean(imputation_compare_cv['test_score'])
```

The result should be:

0.772866246168149

Figure 6.31: Confirming cross-validation score with mode imputation

We have now completed the selection of imputation strategy.

Confirming Model Performance on the Unseen Test Set

We are nearly done building the model for the case study data. The final step in the model building process is to examine model performance on the test set, which we have reserved from using for any model-building activities up till now.

You may notice that we actually did use the samples of the test set already, in a few places. When carrying out initial data exploration and cleaning, we had not separated them out yet. This is typical, as one would like to make sure that the testing data are cleaned in the same way the training data have been. Another place we used the testing samples is just now, when we made predictions for **PAY_1** and compared them with actual values. However, we viewed this as a separate modeling effort.

The main motivation behind having an "unseen" test set is to have an independent data source on which we can test the model we selected along with the hyperparameters we selected for it. We used all the training data for these purposes. While we can say we've estimated out-of-sample performance using cross-validation, the test set gives us a way to truly see how the model performs on data that was never used to fit the model. Ideally, the results will be similar to the cross-validation estimates. If not, one should explore why: are the testing data different in some way than the training data? Visualizations of the features and response would be helpful in such an inquiry.

Now that we have assembled all the non-imputed and imputed data for training and testing, we can train the final model like this:

```
rf.fit(X_train_all, y_train_all)
```

The output should be:

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                      max_depth=9, max_features='auto', max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, n_estimators=200, n_jobs=None,
                      oob_score=False, random_state=4, verbose=1, warm_start=False)
```

Figure 6.32: Training the final model

Finally, we make predictions for credit default on the testing set. We need to predict probabilities so we can calculate ROC AUC:

```
y_test_all_predict_proba = rf.predict_proba(X_test_all)
```

Import the `roc_auc_score` function and calculate the testing score:

```
from sklearn.metrics import roc_auc_score  
roc_auc_score(y_test_all, y_test_all_predict_proba[:,1])
```

The output should be:

0.7696243835824927

Figure 6.33: Testing score of the final model with imputed and non-imputed data

What has the testing score revealed? The appropriate comparison to make here is with the cross-validated testing score we computed using the mode-imputed `PAY_1`, which was 0.773. We can see that the testing score here, of 0.770, is roughly within 1 standard deviation of the cross-validation estimate. We conclude that the model is robust and is ready for delivery to the client.

At this point, model building activities have completed. A final step before delivering a trained scikit-learn model would be to fit it on all of the available data, including the unseen test set. This would be done by concatenating the training and testing data features (`X_train_all`, `X_test_all`) and labels (`y_train_all`, `y_test_all`), and using them to fit a model.

Our main concerns now are helping the client use the model to meet their business goals, providing guidance on how the model's performance can be monitored as time goes on, and presenting all our results and recommendations to the client.

Financial Analysis

The model performance metrics we have calculated so far were based on abstract measures that could be applied to analyze any classification model: how accurate a model is or how skillful a model is at identifying true positives relative to false positives (ROC AUC), or the correctness of positive predictions (precision). These metrics are important for understanding the basic workings of a model and are widely used within the machine learning community, so it's important to thoroughly understand them. However, for the application of a model to business processes, clients will not always be able to use such model performance metrics to establish an understanding of exactly how they will use a model to guide business decisions, or how much value a model can be expected to create. To go the extra mile and make the connection of the mathematical world of predicted probabilities and thresholds, to the business world of costs and benefits, a financial analysis of some kind is usually required.

In order to help the client with this analysis, the data scientist needs to understand what kinds of decisions and actions might be taken, based on predictions made by the model. This should be the topic of a conversation with the client, preferably early on in the project lifecycle. We have left it until the end of the book so that we could establish a baseline understanding of what predictive modeling is and how it works. However, *learning the business context around model usage at the beginning of a project allows you to set goals for model performance in terms of the creation of value that you can work toward during model development.* Translating model performance metrics into financial terms is the topic of this section.

For a binary classification model like that of the case study, here are a few questions that the data scientist needs to know the answers to, in order to figure out how to apply the model for the client:

- What kinds of decisions does the client want to use the model to help them make?
- How can the predicted probabilities of a binary classification model be used to help make these decisions?
- Are they yes/no decisions? Then choosing a single threshold of predicted probability will be sufficient.
- Are there more than two levels of activity that will be decided on, based on model results? Then choosing 2 or more thresholds, for example to sort predictions in to low, medium, and high risk, may be the solution. In this case, predicted probabilities below 0.5 may be considered low risk, those between 0.5 and 0.75 medium risk, and those above 0.75 high risk.
- What are the costs of taking all the different courses of action that are available, based on model guidance?
- What are the potential benefits to be gained from successful actions taken as a result of model guidance?

Financial Conversation with the Client

We ask the case study client about the points outlined above and learn the following: for credit accounts that are at a high risk of default, the client is designing a new program to provide individualized counseling for the account holder, to encourage them to pay their bill on time or provide alternative payment options if that will not be possible. Credit counseling is performed by trained customer service representatives who work in a call center. The cost per attempted counseling session is NT\$7,500 and the expected success rate of a session is 70%, meaning that on average 70% of the recipients of phone calls offering counseling will pay their bill on time, or make alternative arrangements that are acceptable to the creditor. The potential benefits of successful counseling are that the amount of an account's monthly bill will be realized as savings, if it was going to default but instead didn't, as a result of the counseling. Currently, the monthly bills for accounts that default are reported as losses.

After having the above conversation with the client, we have the materials we need to make a financial analysis. The client would like us to help them decide which members to contact and offer credit counseling to. If we can help them narrow down the list of people who will be contacted for counseling, we can help save them money by avoiding unnecessary and expensive contacts. The clients' limited resources for counseling will be more appropriately spent on accounts that are at higher risk of default. This should create greater savings due to prevented defaults. Additionally, the client lets us know that our analysis can help them request a budget for the counseling program, if we can give them an idea of how many counseling sessions it would be worthwhile to offer.

As we proceed to the financial analysis, we see that the decision that the model will help the client make, on an account by account basis, is a yes/no decision: whether to offer counseling to the holder of a given account. Therefore, our analysis should focus on finding an appropriate threshold of predicted probability, by which we may divide our accounts into two groups: higher risk accounts that will receive counseling and lower risk ones that won't.

Exercise 25: Characterizing Costs and Savings

The connection between model output and business decisions the client will make, comes down to selecting a threshold for model-predicted probability. Therefore, in this exercise, we will characterize the expected costs of the counseling program, in terms of costs of offering individual counseling sessions, as well as the expected savings, in terms of prevented defaults, at a range of thresholds. There will be a different cost and savings at each threshold, because each threshold is expected to result in a different number of positive predictions, as well as a different number of true positives within these. The first step is to create an array of potential thresholds. We will use 0 through 1, going by an increment of 0.01. Perform the following steps to complete the exercise.

Note

The code and resulting output for this exercise have been loaded in a Jupyter Notebook that can be found here: <http://bit.ly/2PtdRyj>.

1. Create a range of thresholds to calculate expected costs and benefits of counseling with this code:

```
thresholds = np.linspace(0, 1, 101)
```

This creates 101 linearly spaced points between 0 and 1, inclusive.

Now, we need to know the potential savings of a prevented default. To calculate this precisely, we would need to know the next month's monthly bill. However, the client has informed us that this will not be available at the time they need to create the list of account holders to be contacted. Therefore, in order to estimate the potential savings, we will use the average amount of the most recent monthly bill across all accounts, to represent the potential savings of a prevented default for each account.

In reality, some accounts will have larger monthly bill amounts and others will have smaller ones. If there were an important relationship between the bill amount and the probability of default, using an average bill amount to calculate savings for all accounts may not produce accurate results. In this case, a predictive modeling exercise could be attempted to estimate the next month's bill for each account, and thus make a more accurate guess as to the potential savings per account. However, bill amounts were not among the top five features for the random forest model, so the association with probability of default is not as strong as with other features. Consequently, this simplifying assumption will probably not affect the results too much.

We will use the testing data to create this analysis, as this provides a simulation of how the model will be used after we deliver it to the client: on new accounts that weren't used for model training.

2. Confirm the index of the testing data features array that corresponds to the most recent month's bill:

```
df[features_response[:-1]].columns[5]
```

The output should be:

'BILL_AMT1'

Figure 6.34: Feature name of most recent months' bill

This is the most recent months' bill.

3. Capture the average bill amount as the potential savings per default and observe it:

```
savings_per_default = np.mean(X_test_all[:, 5])  
savings_per_default
```

The output should be:

51601.7433479286

Figure 6.35: Average of most recent months' bill across accounts

The average of the most recent months' bill across all accounts is NT\$51,602. So, using the assumption that this is the opportunity for savings of a prevented default for each account, the net savings after a cost of NT\$7,500 for credit counseling will be NT\$44,102. This indicates a potential for net savings in the credit counseling program.

The issue is that not all accounts will default. For an account that wouldn't default, a counseling session represents a wasted NT\$7,500. Our analysis needs to balance the costs of counseling with the risk of default.

4. Store the cost of counseling in a variable to use for analysis:

```
cost_per_counseling = 7500
```

We also know from the client that the counseling program isn't 100% effective. We should take this in to account in our analysis.

5. Store the effectiveness rate the client gave us for use in analysis:

```
effectiveness = 0.70
```

Now, we will calculate costs and savings for each of the thresholds. We'll step through each calculation and explain it, but for now, we need to create empty arrays to hold the results for each threshold.

6. Create empty arrays to store analysis results. We'll explain what each one will hold in the following steps:

```
n_pos_pred = np.empty_like(thresholds)
cost_of_all_counselings = np.empty_like(thresholds)
n_true_pos = np.empty_like(thresholds)
savings_of_all_counselings = np.empty_like(thresholds)
savings_based_on_balances = np.empty_like(thresholds)
```

These create empty arrays with the same number of elements as there are thresholds in our analysis. We will loop through each threshold value to fill these arrays.

7. Make a counter variable and open a **for** loop to go through thresholds:

```
counter = 0
for threshold in thresholds:
```

For each threshold, there will be a different number of positive predictions, according to how many predicted probabilities are above that threshold. These correspond to accounts that are predicted to default. Each account that is predicted to default, will receive a counseling phone call, which has a cost associated with it. So, this is the first part of the cost calculation.

8. Determine which accounts get positive predictions, at this threshold:

```
pos_pred = y_test_all_predict_proba[:,1]>threshold
```

pos_pred is a Boolean array. The sum of **pos_pred** indicates the number of predicted defaults, at this threshold.

9. Calculate the number of positive predictions, at this threshold:

```
n_pos_pred[counter] = sum(pos_pred)
```

10. Calculate the cost of all counseling, at this threshold:

```
cost_of_all_counselings[counter] = n_pos_pred[counter] * cost_per_
counseling
```

Now that we have characterized the possible costs of the counseling program, at each threshold, we need to see what the potential savings are. Savings are obtained when counseling is offered to the right account holders: those who would otherwise default. In terms of the classification problem, these are positive predictions, where the actual value of the response variable is also positive. In other words, true positives.

11. Determine which accounts are true positives, based on the array of positive predictions and the response variable:

```
true_pos = pos_pred & y_test_all.astype(bool)
```

12. Calculate the number of true positives as the sum of the true positive array:

```
n_true_pos[counter] = sum(true_pos)
```

The savings we can get from successfully counseling account holders who would otherwise default depends on the savings per prevented default, as well as the effectiveness rate of counseling. We won't be able to prevent every default.

13. Calculate the anticipated savings at this threshold using the number of true positives, the savings per prevented default, and the effectiveness rate of counseling:

```
savings_of_all_counselings[counter] = n_true_pos[counter] * savings_per_
default * effectiveness
```

14. Increment the counter:

```
counter += 1
```

Steps 7 through 14 should be run as a for loop in one cell in the Jupyter notebook. Afterwards, the net savings for each threshold can be calculated as the savings minus the cost.

15. Calculate net savings for all the thresholds by subtracting the savings and cost arrays:

```
net_savings = savings_of_all_counselings - cost_of_all_counselings
```

Now, we're in a position to visualize how much money we might help our client save by providing counseling to the appropriate account holders. Let's visualize this.

16. Plot the net savings against the thresholds as follows:

```
mpl.rcParams['figure.dpi'] = 400
plt.plot(thresholds, net_savings)
plt.xlabel('Threshold')
plt.ylabel('Net savings (NT$)')
plt.xticks(np.linspace(0,1,11))
plt.grid(True)
```

The resulting plot should look like this:

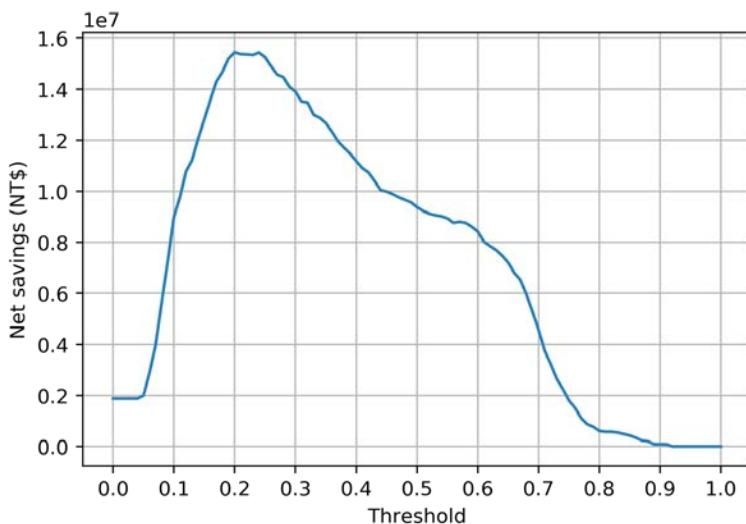


Figure 6.36: Plot of net savings versus thresholds

The plot indicates that the choice of threshold is important. While it will be possible to create net savings at many different values of the threshold, it looks like the highest net savings will be generated by setting the threshold somewhere in the range of about 0.2 to 0.25.

Let's confirm the optimal threshold for creating the greatest savings and see how much the savings are.

17. Find the index of the largest element of the net savings array using NumPy's `argmax`:

```
max_savings_ix = np.argmax(net_savings)
```

18. Display the threshold that results in the greatest net savings:

```
thresholds[max_savings_ix]
```

The output should be as follows:

0 . 2

Figure 6.37: Threshold for greatest net savings

19. Display the greatest possible net savings:

```
net_savings[max_savings_ix]
```

The output should be as follows:

15446325.35991916

Figure 6.38: Greatest net savings

We see that the greatest net savings occurs at a threshold of 0.2 of predicted probability of default. The amount of net savings realized at this threshold is over NT\$15 million, for the testing dataset of accounts. These savings would need to be scaled by the number of accounts served by the client, to estimate the total possible savings, assuming the data we are working with is representative of all these accounts.

Note, however, that the savings is about the same up to a threshold of about 0.25, as seen in *Figure 6.36*.

As the threshold increases, we are "raising the bar" for how risky a client must be, in order for us to contact them and offer counseling. Increasing the threshold from 0.2 to 0.25 means we would be only contacting riskier clients whose probability is > 0.25 . This means contacting fewer clients, reducing the up-front cost of the program. *Figure 6.36* indicates that we may be still able to create the same amount of net savings, by contacting fewer people. While the net effect is the same, the initial expenditure on counseling will be smaller. This may be desirable to the client. We explore this concept further in the following activity.

Activity 6: Deriving Financial Insights

The raw materials of the financial analysis are completed. However, we can generate some additional insights from these results, to provide the client with more context around how the predictive model we built can generate value for them. In particular, we have looked at results for the testing set we reserved from model building. The client may have more accounts than those they supplied to us, as representative of their business. We should report to them results that could be easily scaled to however big their business is, in terms of number of accounts.

We can also help them understand how much this program will cost; while the net savings are an important number to consider, the client will have to fund the counseling program before any of these savings will be obtained. Finally, we will link the financial analysis back to standard machine learning model performance metrics.

Once you complete the activity, you should be able to communicate the initial cost of the counseling program to the client, as well as obtain plots of precision and recall such as this:

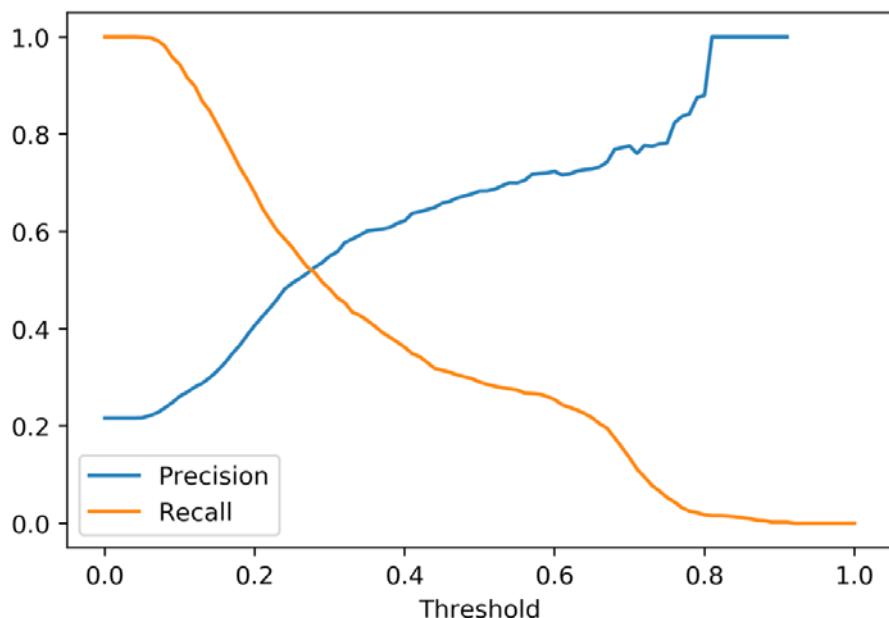


Figure 6.39: Expected precision-recall curve

This curve will be useful in interpreting the value created by the model at different thresholds.

Perform the following steps to complete the activity:

Note

The code and the resulting output for this exercise have been loaded in a Jupyter Notebook that can be found here: <http://bit.ly/2PtdRyj>. Using the testing set, calculate the cost of all defaults if there were no counseling program.

1. Using the testing set, calculate the cost of all defaults if there were no counseling program.
2. Calculate by what percent can the cost of defaults be decreased by the counseling program.
3. Calculate the net savings per account at the optimal threshold.
4. Plot the net savings per account against the cost of counseling per account for each threshold.
5. Plot the fraction of accounts predicted as positive (this is called the "flag rate") at each threshold.
6. Plot a precision-recall curve for the testing data.
7. Plot precision and recall separately on the y-axis against threshold on the x-axis.

Note

The solution to this activity can be found on page 349.

Final Thoughts on Delivering the Predictive Model to the Client

We have now completed modeling activities and also created a financial analysis to indicate to the client how they can use the model. While we have created the essential intellectual contributions that are the data scientists' responsibility, it is necessary to agree with the client on the form in which all these contributions will be delivered.

A key contribution is the predictive capability embodied in the trained model. Assuming the client has the capability to work with the trained model object we created in scikit-learn, this model could be saved to disk and sent to the client. Then, the client would be in a position to use it within their workflow. Alternatively, it may be necessary to express the model as a mathematical equation (i.e. logistic regression) or a set of if-then statements (i.e. decision tree or random forest) that the client could use to implement the predictive capability in SQL. While expressing random forests in SQL code is cumbersome due to the possibility of having many trees with many levels, there are software packages which will create this representation for you.

Before using the model to make predictions, the client would need to ensure that the *data were prepared in the same way they were for the model building we have done*. For example, the removal of samples with values of 0 for all the features, the cleaning of the **EDUCATION** and **MARRIAGE** features, and the imputation of **PAY_1**, would all have to be done in the same way we demonstrated earlier in this chapter. Alternatively, there are other possible ways to deliver model predictions, such as an arrangement where the client delivers data to the data scientist and receives the predictions back.

Another important consideration for the discussion of deliverables is: *what format should the predictions be delivered in?* A typical delivery format for predictions from a binary classification model, like that we've created for the case study, is to rank accounts on their predicted probability of default. The predicted probability should be supplied along with the account ID and whatever other columns the client would like. This way, when the call center is working their way through the list of account holders to offer counseling to, they can contact those at highest risk for default first and proceed to lower priority account holders as time and resources allow. The client has been informed of which threshold to use for predicted probabilities, to result in the highest net savings. This threshold would represent the stopping point on the list of account holders to contact, if it is ranked on predicted probability of default.

Finally, depending on how long the client has engaged the data scientist for, it is always beneficial to monitor the performance of the model over time, as it is being used. Does predictive capability remain the same or degrade over time? When assessing this, it's important to keep in mind that if account holders are receiving counseling, their probability of default would be expected to be lower than the predicted probability indicates, due to the intended effects of the new counseling program. For this reason, and also to test the effectiveness of the counseling program, it is good practice to reserve a randomly chosen portion of account holders who will not receive any counseling, regardless of credit default risk. This group will be known as the **control group** and should be small compared to the rest of the population who receives counseling, but large enough to draw statistically significant inferences from.

While it's beyond the scope of this book to go in to details about how to design and use a control group, suffice to say here that model predictive capability could be assessed on the control group since they have received no counseling, similar to the population of accounts the model was trained on. Another benefit of a control group is that the rate of default, and financial loss due to defaults, can be compared to those accounts that received the model-guided counseling program. If the program is working as intended, the accounts receiving the counseling program will have a lower rate of default and a smaller financial loss due to default. The control group can provide evidence that the program is, in fact, working.

A relatively simple way to monitor a model implementation is to see if the nature of model predictions is changing over time, as compared to the population used for model training. This can be done by plotting the predicted probabilities, using a histogram:

```
plt.hist(y_test_all_predict_proba[:,1], bins=30)  
plt.xlabel('Predicted probability of default')  
plt.ylabel('Number of accounts')
```

The output graphic should be as follows:

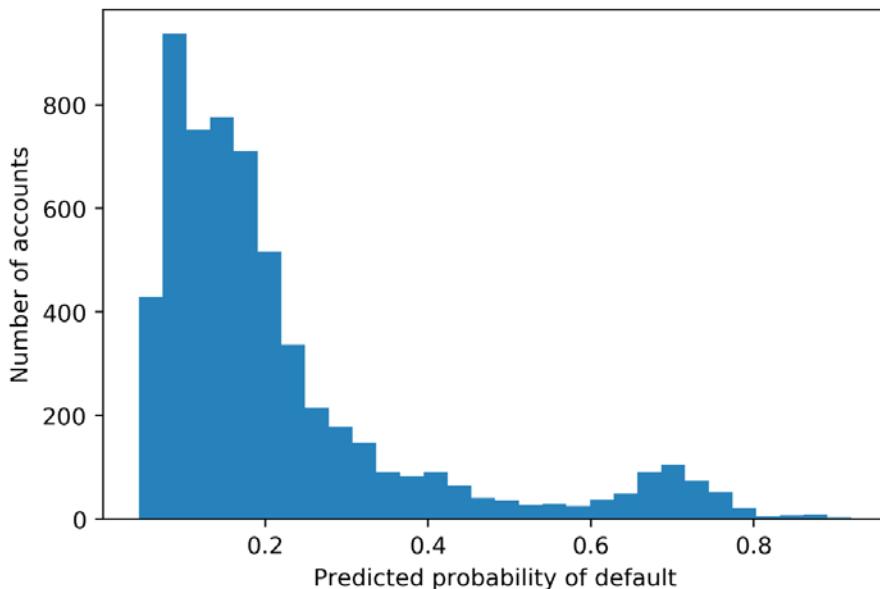


Figure 6.40: Histogram of predicted probabilities of default

If the shape of the histogram of predicted probabilities changes substantially, it may be a sign that the relationship between the features and response in the population of accounts has changed and the model may need to be re-trained or rebuilt. This may also become evident if the number of accounts predicted to default, according to a chosen threshold, changes in a noticeable way.

Summary

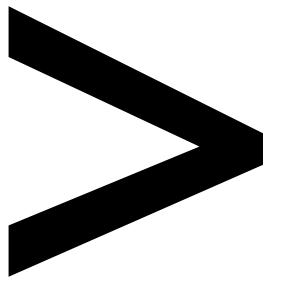
In this chapter, you learned the core data science skill of imputation. Imputation allows the replacement of missing values for features with educated guesses as to what the real values are. Imputation is necessary to provide predictions for all samples, including those with missing feature values. This is because many machine learning algorithms, including those we've worked with in this book, cannot take input which includes missing values. Imputation methods range from simple ones, such as imputing with the mean, median, or mode of non-missing values, to more complex methods, including creating a predictive model for the feature that has missing values.

After ensuring that the model we deliver can provide predictions for all samples, via imputation, we conducted a financial analysis. While we left this to the end of the book, an understanding of the costs and savings going along with the decisions to be guided by the model should be understood from the beginning of a typical project. These allow the data scientist to work toward a tangible goal in terms of increased profit or savings. A key step in this process, for binary classification models, is to choose a threshold of predicted probability at which to declare a positive sample, so that the profits or savings due to model-guided decision making are maximized.

Finally, we reviewed a few points related to delivering and monitoring the model, including the idea of establishing a control group to monitor model performance and test the effectiveness of any programs guided by model output. The structure of control groups and model monitoring strategies will be different from project to project, so you will need to determine the appropriate course of action in each new case.

Congratulations, you have now completed the project and are ready to deliver your findings to the client! Along with trained models saved to disk, or other data products or services you may provide to the client, you will probably also want to create a presentation, typically a slide show, detailing your progress. Contents of such presentations usually include a problem statement, results of data exploration and cleaning, a comparison of different models you built, any other necessary steps such as imputation, and the financial analysis which shows how valuable your work is. As you craft presentations of your work, it's usually better to tell your story with pictures as opposed to a lot of text. We've demonstrated many visualization techniques throughout the book that you can use to do this, and you should continue to explore ways to depict data and modeling results.

Always be sure to ask the client which specific things they may want to see in a presentation and be sure to answer all their questions. When a client sees that you can create value for them in an understandable way, you have succeeded.



Appendix

About

This section is included to assist the students to perform the activities in the book. It includes detailed steps that are to be performed by the students to achieve the objectives of the activities.

Chapter 1: Data Exploration and Cleaning

Activity 1: Exploring Remaining Financial Features in the Dataset

1. Create lists of feature names for the remaining financial features.

These fall into two groups, so we will make lists of feature names as before, to facilitate analyzing them together. You can do this with the following code:

```
bill_feats = ['BILL_AMT1', 'BILL_AMT2', 'BILL_AMT3', 'BILL_AMT4', 'BILL_AMT5', 'BILL_AMT6']
pay_amt_feats = ['PAY_AMT1', 'PAY_AMT2', 'PAY_AMT3', 'PAY_AMT4', 'PAY_AMT5', 'PAY_AMT6']
```

2. Use `.describe()` to examine statistical summaries of the bill amount features. Reflect on what you see. Does it make sense?

Use the following code to view the summary:

```
df[bill_feats].describe()
```

The output should appear as follows:

```
df[bill_feats].describe()
```

	BILL_AMT1	BILL_AMT2	BILL_AMT3	BILL_AMT4	BILL_AMT5	BILL_AMT6
count	26664.000000	26664.000000	26664.000000	26664.000000	26664.000000	26664.000000
mean	51405.730723	49300.001500	47026.340047	43338.894539	40338.136701	38889.872337
std	73633.687106	70934.549534	68705.359524	64275.250740	60705.944083	59432.541657
min	-165580.000000	-69777.000000	-157264.000000	-170000.000000	-81334.000000	-339603.000000
25%	3580.000000	2999.750000	2627.250000	2341.750000	1745.000000	1256.000000
50%	22361.000000	21150.000000	20079.500000	19037.000000	18066.000000	17005.000000
75%	67849.750000	64395.500000	60360.000000	54727.500000	50290.500000	49253.750000
max	746814.000000	671563.000000	855086.000000	706864.000000	823540.000000	699944.000000

Figure 6.41: Statistical description of bill amounts for the past 6 months

We see that the average monthly bill is roughly 40,000 to 50,000 NT dollars. The reader is encouraged to examine the conversion rate to their local currency. For example, 1 US dollar \approx 30 NT dollars. Do the conversion and ask yourself, is this a reasonable monthly payment? We should also confirm this with the client, but it seems reasonable.

We also notice there are some negative bill amounts. This seems reasonable because of possible overpayment of the previous months' bill, perhaps in anticipation of a purchase that would show up on the current months' bill. A scenario like this would leave that account with a negative balance, in the sense of a credit to the account holder.

3. Visualize the bill amount features using a 2 by 3 grid of histogram plots using the following code:

```
df[bill_feats].hist(bins=20, layout=(2,3))
```

The graph should look like this:

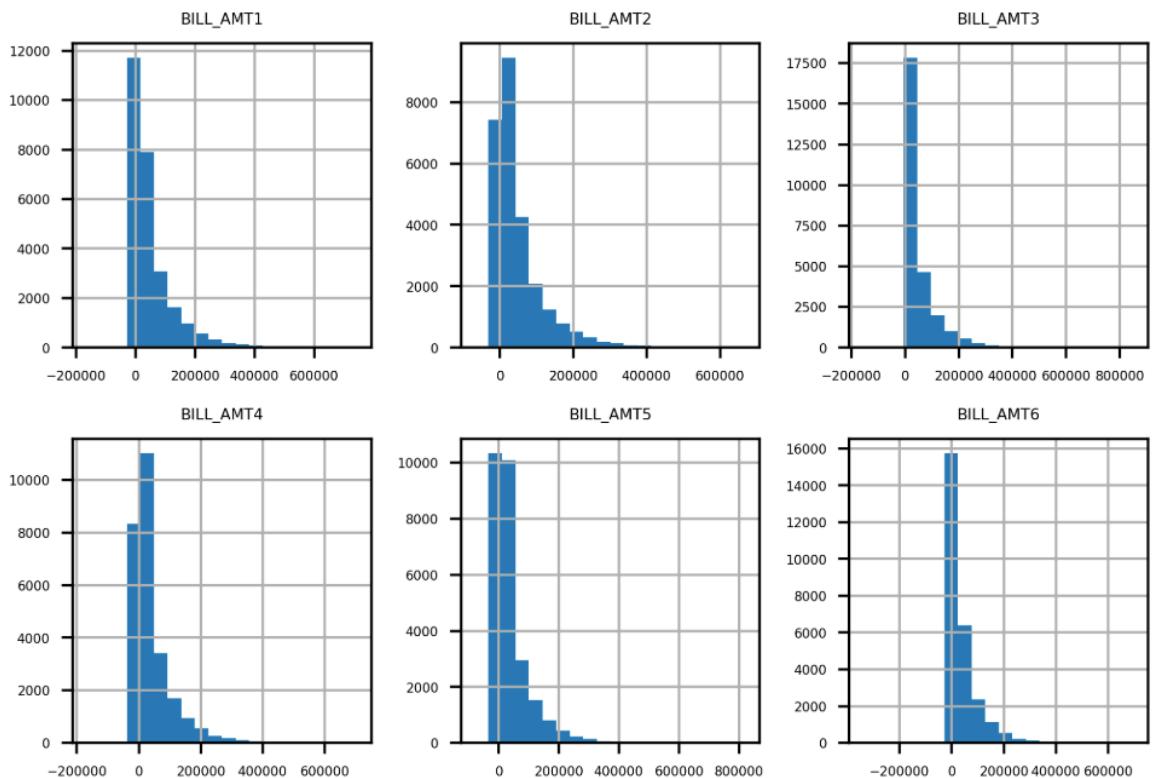


Figure 6.42: Histograms of bill amounts

The histogram plots in Figure 6.42 make sense in several respects. Most accounts have relatively small bills. There is a steady decrease in the number of accounts as the amount of the bill increases. It also appears that the distribution of payments is roughly similar month-to-month, so we don't notice any data inconsistency issues as we did with the payment status features. This feature appears to pass our data quality inspection. Now, we move on to the final set of features.

4. Use the `.describe()` method to obtain a summary of the payment amount features using the following code:

```
df[pay_amt_feats].describe()
```

The output should appear thus:

```
df[pay_amt_feats].describe()
```

	PAY_AMT1	PAY_AMT2	PAY_AMT3	PAY_AMT4	PAY_AMT5	PAY_AMT6
count	26664.000000	2.666400e+04	26664.000000	26664.000000	26664.000000	26664.000000
mean	5704.085771	5.881110e+03	5259.514964	4887.048717	4843.729973	5257.843047
std	16699.398632	2.121431e+04	17265.439561	15956.349371	15311.721795	17635.468185
min	0.000000	0.000000e+00	0.000000	0.000000	0.000000	0.000000
25%	1000.000000	8.020000e+02	390.000000	294.750000	242.750000	111.000000
50%	2114.500000	2.007000e+03	1822.000000	1500.000000	1500.000000	1500.000000
75%	5027.000000	5.000000e+03	4556.250000	4050.500000	4082.750000	4015.000000
max	873552.000000	1.227082e+06	889043.000000	621000.000000	426529.000000	528666.000000

Figure 6.43: Statistical description of bill payment amounts for the past 6 months

The average payment amounts are about an order of magnitude (power of 10) lower than the average bill amounts we summarized earlier in the Activity. This means that the "average case" is an account that is not paying off its entire balance from month to month. This makes sense in light of our exploration of the `PAY_1` feature, for which the most prevalent value was 0 (account made at least the minimum payment but did not pay off the whole balance). There are no negative payments, which also seems right.

5. Plot a histogram of the bill payment features similar to the bill amount features, but also apply some rotation to the `x-axis` labels with the `xrot` keyword argument so that they don't overlap. Use the `xrot=<angle>` keyword argument to rotate `x-axis` labels by a given angle in degrees using the following code:

```
df[pay_amt_feats].hist(layout=(2,3), xrot=30)
```

In our case, we found that 30 degrees of rotation worked well. The plot should look like this:

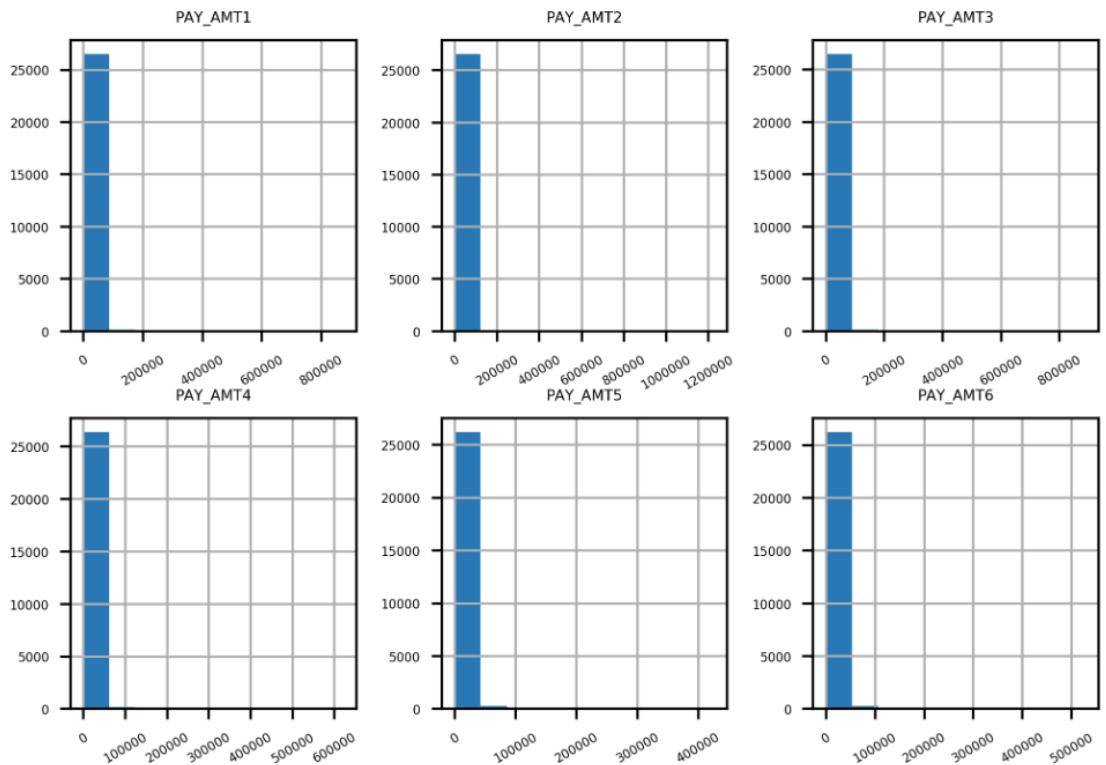


Figure 6.44: Histograms of raw payment amount data

A quick glance at this figure indicates that this is not a very informative graphic; there is only one bin in most of the histograms that is of any noticeable height. This is not an effective way to visualize this data. It appears that the monthly payment amounts are mainly in a bin that includes 0. How many are in fact 0?

6. Use a Boolean mask to see how many of the payment amount data are exactly equal to 0 using the following code: Do this with the following code:

```
pay_zero_mask = df[pay_amt_feats] == 0
pay_zero_mask.sum()
```

The output should look like this:

```
pay_zero_mask = df[pay_amt_feats] == 0

pay_zero_mask.sum()

PAY_AMT1      4656
PAY_AMT2      4833
PAY_AMT3      5293
PAY_AMT4      5697
PAY_AMT5      5981
PAY_AMT6      6373
dtype: int64
```

Figure 6.45: Counts of bill payments equal to 0

Does this data make sense given the histogram in the previous step?

The first line here creates a new DataFrame called `pay_zero_mask`, which is a DataFrame of `True` and `False` values according to whether the payment amount is equal to 0. The second line takes the column sums of this DataFrame, interpreting `True` as 1 and `False` as 0, so the column sums indicate how many accounts have a value of 0 for each feature.

We see that a substantial portion, roughly around 20-25% of accounts, have a bill payment equal to 0 in any given month. However, most bill payments are above 0. So, why can't we see them in the histogram? This is due to the `range` of values for bill payments relative to the values of the majority of the bill payments.

In the statistical summary, we can see that the maximum bill payment in a month is typically 2 orders of magnitude (100 times) larger than the average bill payment. It seems likely there are only a small number of these very large bill payments. But, because of the way the histogram is created, using equal sized bins, nearly all the data is lumped into the smallest bin, and the larger bins are nearly invisible because they have so few accounts. We need a strategy to effectively visualize this data.

7. Ignoring the payments of 0 using the mask you created in the previous step, use pandas' `.apply()` and NumPy's `np.log10()` method to plot histograms of logarithmic transformations of the non-zero payments. You can use `.apply()` to apply any function, including `log10`, to all the elements of a DataFrame. Use the following code to complete the preceding step:

```
df[pay_amt_feats][~pay_zero_mask].apply(np.log10).hist(layout=(2,3))
```

This is a relatively advanced use of pandas, so don't worry if you couldn't figure it out by yourself. However, it's good to start to get an impression of how you can do a lot in pandas with relatively little code.

The output should be as follows:

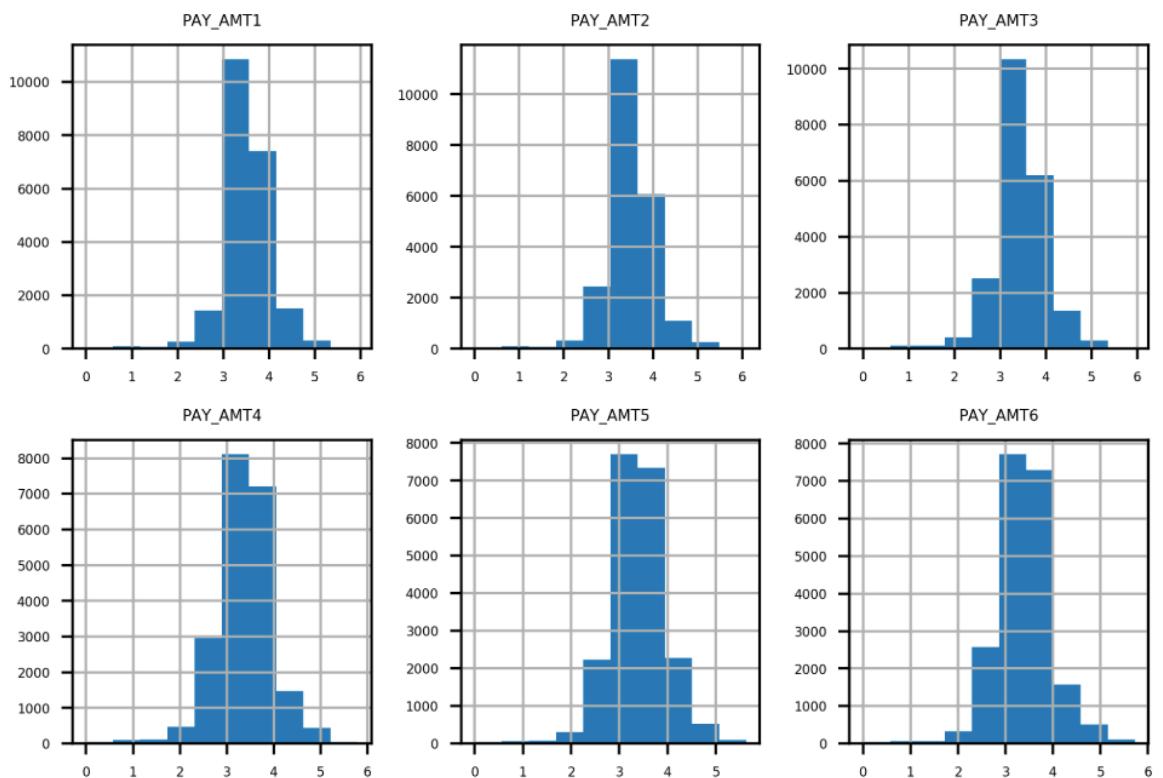


Figure 6.46: Base-10 logs of non-zero bill payment amounts

While we could have tried to create variable width bins for better visualization of the payment amounts, a more convenient approach that is often used to visualize, and sometimes even model, data that has a few values on a much different scale than most of the values, is a logarithmic transformation, or **log transform**. We used a base-10 log transform. Roughly speaking, this transform tells us the number of zeros in a value. In other words, a million-dollar balance would have a log transform of at least 6 but less than 7, because $10^6 = 1,000,000$ (and conversely $\log_{10}(1,000,000) = 6$) while $10^7 = 10,000,000$.

To apply this transformation to our data, first, we needed to mask out the zero payments, because `log10(0)` is undefined. We did this with the Python logical not operator `~` and the zero mask we created already. Then we used the pandas `.apply()` method, which applies any function we like to the data we have selected. In this case, we wished to apply a base-10 logarithm, calculated by `np.log10`. Finally, we made histograms of these values.

The result is a more effective data visualization: the values are spread in a more informative way across the histogram bins. We can see that the most commonly occurring bill payments are in the range of thousands ($\log_{10}(1,000) = 3$), which matches what we observed for the mean bill payment in the statistical summary. There are some pretty small bill payments, and also a few pretty large ones. Overall, the distribution of bill payments appears pretty consistent from month to month, so we don't see any potential issues with these data.

Chapter 2: Introduction to Scikit-Learn and Model Evaluation

Activity 2: Performing Logistic Regression with a New Feature and Creating a Precision-Recall Curve

1. Use scikit-learn's `train_test_split` to make a new set of training and testing data. This time, instead of `EDUCATION`, use `LIMIT_BAL`: the account's credit limit.

Execute the following code to do this:

```
X_train_2, X_test_2, y_train_2, y_test_2 = train_test_split(
    df['LIMIT_BAL'].values.reshape(-1,1), df['default payment next month'].values,
    test_size=0.2, random_state=24)
```

Notice here we create new training and testing splits, with new variable names.

2. Train a logistic regression model using the training data from your split.

The following code does this:

```
example_lr.fit(X_train_2, y_train_2)
```

We reuse the same model object, `example_lr`. We can **re-train** this object to learn the relationship between this new feature and the response. We could even try a different train/test split, if we wanted to, without creating a new model object. The existing model object has been updated **in-place**.

3. Create the array of predicted probabilities for the testing data:

Here is the code for this step:

```
y_test_2_pred_proba = example_lr.predict_proba(X_test_2)
```

4. Calculate the ROC AUC using the predicted probabilities and the true labels of the testing data. Compare this to the ROC AUC from using the `EDUCATION` feature:

Run this code for this step:

```
metrics.roc_auc_score(y_test_2, y_test_2_pred_proba[:,1])
```

The output is as follows:

```
metrics.roc_auc_score(y_test_2, y_test_2_pred_proba[:,1])
```

```
0.6201990844642832
```

Figure 6.47: Calculating the ROC AUC

Notice we index the predicted probabilities array in order to get the predicted probability of the positive class from the second column. How does this compare to the ROC AUC from the **EDUCATION** logistic regression? The AUC is higher. This may be because now we are using a feature that has something to do with an account's financial status (credit limit), to predict something else related to the account's financial status (whether or not it will default), instead of using something less directly related to finances.

5. Plot the ROC curve.

Here is the code to do this; it's similar to the code we used in the previous exercise:

```
fpr_2, tpr_2, thresholds_2 = metrics.roc_curve(y_test_2, y_test_2_pred_proba[:,1])
plt.plot(fpr_2, tpr_2, '*-')
plt.plot([0, 1], [0, 1], 'r--')
plt.legend(['Logistic regression', 'Random chance'])
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.title('ROC curve for logistic regression with LIMIT_BAL feature')
```

The plot should appear as follows:

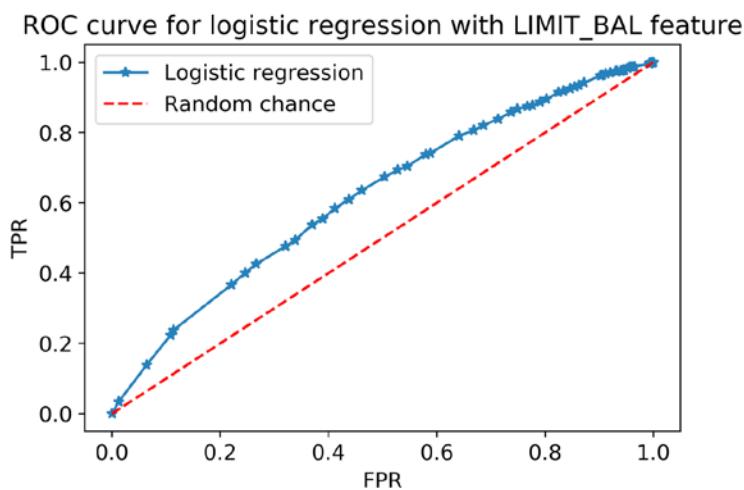


Figure 6.48: ROC curve for the LIMIT_BAL logistic regression

This looks a little closer to an ROC curve that we'd like to see: it's a bit further from the random chance line than the model using only **EDUCATION**. Also notice that the variation in pairs of true and false positive rates is a little smoother over the range of thresholds, reflective of the larger number of distinct values of the **LIMIT_BAL** feature.

6. Calculate the data for the precision-recall curve on the testing data using scikit-learn functionality.

Precision is often considered in tandem with recall. You are already familiar with recall. This is just another word for the true positive rate. We can use `precision_recall_curve` in `sklearn.metrics` to automatically vary the threshold and calculate pairs of precision and recall values at each one. Here is the code to retrieve these values, which is similar to `roc_curve`:

```
precision, recall, thresh_3 = \
    metrics.precision_recall_curve(y_test_2, y_test_2_pred_proba[:,1])
```

7. Plot the precision-recall curve using matplotlib: we can do this with the following code.

Note that we put recall on the x-axis, precision on the y-axis, and set the axes limits to the range [0, 1]:

```
plt.plot(recall, precision, '-x')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Another logistic regression with just one feature: LIMIT_BAL')
plt.xlim([0, 1])
plt.ylim([0, 1])
```

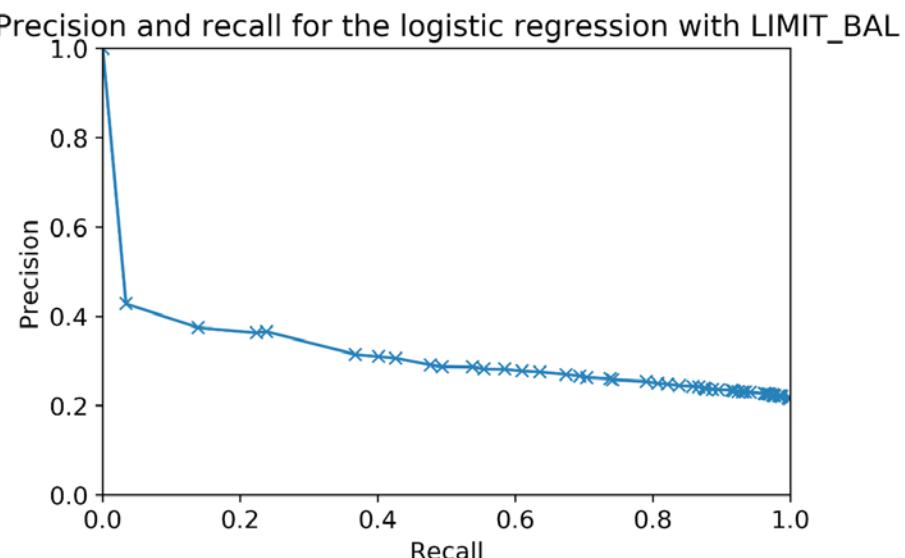


Figure 6.49: Plot of the precision-recall curve

8. Use scikit-learn to calculate the area under the precision-recall curve.

Here is the code for this:

```
metrics.auc(recall, precision)
```

You will obtain the following output:

```
metrics.auc(recall, precision)
```

```
0.31566964427378624
```

Figure 6.50: Area under the precision-recall curve

We saw that the precision-recall curve shows that precision is generally fairly low for this model; for nearly all of the range of thresholds, the precision, or portion of positive classifications that are correct, is less than half. We can calculate the area under the precision-recall curve as a way to compare this classifier with other models or feature sets we may consider.

Scikit-learn offers functionality for calculating an area under the curve for any set of x-y data, using the trapezoid rule, which you may recall from calculus: `metrics.auc`. We used this functionality to get the area under the precision-recall curve.

9. Now recalculate the ROC AUC, except this time do it for the training data. How is this different, conceptually and quantitatively, from your earlier calculation?

First, we need to calculate predicted probabilities using the training data, as opposed to the testing data. Then we can calculate the ROC AUC using the training data labels. Here is the code:

```
y_train_2_pred_proba = example_lr.predict_proba(X_train_2)  
metrics.roc_auc_score(y_train_2, y_train_2_pred_proba[:,1])
```

You should obtain the following output:

```
y_train_2_pred_proba = example_lr.predict_proba(X_train_2)
```

```
metrics.roc_auc_score(y_train_2, y_train_2_pred_proba[:,1])
```

```
0.6182918113358344
```

Figure 6.51: Training data ROC AUC

Quantitatively, we can see that this AUC is not all that different from the testing data ROC AUC we calculated earlier. Both are about 0.62. Conceptually, what is the difference? When we calculate this metric on the training data, we are measuring the model's skill in predicting the same data that "taught" the model how to make predictions. We are seeing *how well the model fits the data*. When we compare this to a testing data metric, we are comparing **training and testing scores**. If there was much of a difference in these scores, which usually would come in the form of a higher training score than testing score, it would indicate that while the model fits the data well, the trained model does not generalize well to new, unseen data.

In this case, the training and testing scores are similar, meaning the model does about as well on out-of-sample data as it does on the same data used in model training. We will learn more about the insights we can gain by comparing training and testing scores in *Chapter 4, The Bias-Variance Trade-off*.

Chapter 3: Details of Logistic Regression and Feature Exploration

Activity 3: Fitting a Logistic Regression Model and Directly Using the Coefficients

The first few steps are similar to things we've done in previous activities:

1. Create a train/test split (80/20) with **PAY_1** and **LIMIT_BAL** as features:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    df[['PAY_1', 'LIMIT_BAL']].values, df['default payment next month'].values,
    test_size=0.2, random_state=24)
```

2. Import **LogisticRegression**, with the default options, but set the solver to '**liblinear**'.

```
from sklearn.linear_model import LogisticRegression
lr_model = LogisticRegression(solver='liblinear')
```

3. Train on the training data and obtain predicted classes, as well as class probabilities, using the testing data:

```
lr_model.fit(X_train, y_train)
y_pred = lr_model.predict(X_test)
y_pred_proba = lr_model.predict_proba(X_test)
```

4. Pull out the coefficients and intercept from the trained model and manually calculate predicted probabilities. You'll need to add a column of 1s to your features, to multiply by the intercept.

First, let's create the array of features, with a column of 1s added, using horizontal stacking:

```
ones_and_features = np.hstack([np.ones((X_test.shape[0],1)), X_test])
```

Now we need the intercept and coefficients, which we reshape and concatenate from scikit-learn output:

```
intercept_and_coefs = np.concatenate([lr_model.intercept_.reshape(1,1),
    lr_model.coef_], axis=1)
```

To repeatedly multiply the intercept and coefficients by the all the rows of `ones_and_features`, and take the sum of each row (that is, find the linear combination), you could write this all out using multiplication and addition. However, it's much faster to use the dot product:

```
X_lin_comb = np.dot(intercept_and_coefs, np.transpose(ones_and_features))
```

Now `X_lin_comb` has the argument we need to pass to the sigmoid function we defined, in order to calculate predicted probabilities:

```
y_pred_proba_manual = sigmoid(X_lin_comb)
```

- Using a threshold of `0.5`, manually calculate predicted classes. Compare this to the class predictions output by scikit-learn.

The manually predicted probabilities, `y_pred_proba_manual`, should be the same as `y_pred_proba`; we'll check that momentarily. First, manually predict the classes with the threshold:

```
y_pred_manual = y_pred_proba_manual >= 0.5
```

This array will have a different shape than `y_pred`, but it should contain the same values. We can check whether all the elements of two arrays are equal like this:

```
np.array_equal(y_pred.reshape(1,-1), y_pred_manual)
```

```
True
```

Figure 6.52: Equality of NumPy arrays

- Calculate ROC AUC using both scikit-learn's predicted probabilities, and your manually predicted probabilities, and compare.

First, import the following:

```
from sklearn.metrics import roc_auc_score
```

Then, calculate this metric on both versions, taking care to access the correct column, or reshape as necessary:

```
roc_auc_score(y_test, y_pred_proba_manual.reshape(y_pred_proba_manual.shape[1],))
```

```
0.627207450280691
```

```
roc_auc_score(y_test, y_pred_proba[:,1])
```

```
0.627207450280691
```

Figure 6.53: Calculating the ROC AUC's from predicted probabilities

The AUCs are, in fact, the same. What have we done here? We've confirmed that all we really need from this fitted scikit-learn model, are three numbers: the intercept and the two coefficients. Once we have these, we could create model predictions using a few lines of code, with mathematical functions, that are equivalent to the predictions directly made from scikit-learn.

This is good to confirm your understanding, but otherwise, why would you ever want to do this? We'll talk about **model deployment** in the final chapter. However, depending on your circumstances, you may be in a situation where you don't have access to Python in the environment where new features will need to be input to the model for prediction. For example, you may need to make predictions entirely in SQL. While this is a limitation in general, with logistic regression you can use mathematical functions that are available in SQL to re-create the logistic regression prediction, only needing to copy and paste the intercept and coefficients somewhere in your SQL code. The dot product may not be available, but you can use multiplication and addition to accomplish the same purpose.

Now, what about the results themselves? What we've seen here is that we can slightly boost model performance above our previous efforts: using just `LIMIT_BAL` as a feature in the previous chapter's Activity, the ROC AUC was a bit less at 0.62, instead of 0.63 here. In the next chapter, we'll learn advanced techniques with logistic regression that we can use to boost performance higher than this.

Chapter 4: The Bias-Variance Trade-off

Activity 4: Cross-Validation and Feature Engineering with the Case Study Data

1. Select out the features from the DataFrame of the case study data.

You can use the list of feature names that we've already created in this chapter. But be sure not to include the response variable, which would be a very good (but entirely inappropriate) feature:

```
features = features_response[:-1]
X = df[features].values
```

2. Make a train/test split using a random seed of 24:

```
X_train, X_test, y_train, y_test = train_test_split(X, df['default payment
next month'].values,
test_size=0.2, random_state=24)
```

We'll use this going forward and reserve this testing data as the unseen test set. This way, we can easily create separate notebooks with other modeling approaches, using the same training data.

3. Instantiate the **MinMaxScaler** to scale the data, as shown in the following code:

```
from sklearn.preprocessing import MinMaxScaler
min_max_sc = MinMaxScaler()
```

4. Instantiate a logistic regression model with the **saga** solver, L1 penalty, and set **max_iter** to 1,000 as we'd like to allow the solver enough iterations to find a good solution:

```
lr = LogisticRegression(solver='saga', penalty='l1', max_iter=1000)
```

5. Import the **Pipeline** class and create a **Pipeline** with the scaler and the logistic regression model, using the names '**scaler**' and '**model**' for the steps, respectively:

```
from sklearn.pipeline import Pipeline
scale_lr_pipeline = Pipeline(steps=[('scaler', min_max_sc), ('model', lr)])
```

6. Use the **get_params** and **set_params** methods to see how to view the parameters from each stage of the pipeline and change them:

```
scale_lr_pipeline.get_params()
scale_lr_pipeline.get_params()['model__C']
scale_lr_pipeline.set_params(model__C = 2)
```

7. Create a smaller range of C values to test with cross-validation, as these models will take longer to train and test with more data than our previous exercises; we recommend $C = [10^2, 10, 1, 10^{-1}, 10^{-2}, 10^{-3}]$:

```
C_val_exponents = np.linspace(2, -3, 6)
C_vals = np.float(10)**C_val_exponents
```

8. Make a new version of the `cross_val_C_search` function, called `cross_val_C_search_pipe`. Instead of the `model` argument, this function will take a `pipeline` argument. The changes inside the function will be to set the C value using `set_params(-model__C = <value you want to test>)` on the pipeline, replacing `model` with the pipeline for the `fit` and `predict_proba` methods, and accessing the C value using `pipeline.get_params()['model__C']` for the printed status update.

The changes are as follows:

```
def cross_val_C_search_pipe(k_folds, C_vals, pipeline, X, Y):
    ##...
    pipeline.set_params(model__C = C_vals[c_val_counter])
    ##...
    pipeline.fit(X_cv_train, y_cv_train)
    ##...
    y_cv_train_predict_proba = pipeline.predict_proba(X_cv_train)
    ##...
    y_cv_test_predict_proba = pipeline.predict_proba(X_cv_test)
    ##...
    print('Done with C = {}'.format(pipeline.get_params()['model__C']))
```

Note

For the complete code, refer to <http://bit.ly/2ZAy2Pr>.

9. Run this function as in the previous exercise, but using the new range of C values, the pipeline you created, and the features and response variable from the training split of the case study data. You may see warnings here, or in later steps, about the non-convergence of the solver; you could experiment with the `tol` or `max_iter` options to try and achieve convergence, although the results you obtain with `max_iter = 1000` are likely to be sufficient. Here is the code to do this:

```
cv_train_roc_auc, cv_test_roc_auc, cv_test_roc = \
cross_val_C_search_pipe(k_folds, C_vals, scale_lr_pipeline, X_train, y_train)
```

You will obtain the following output:

```
Done with C = 100.0
Done with C = 10.0
Done with C = 1.0
Done with C = 0.1
Done with C = 0.01
Done with C = 0.001
```

10. Plot the average training and testing ROC AUC across folds, for each C value, using the following code:

```
plt.plot(C_val_exponents, np.mean(cv_train_roc_auc, axis=0), '-o',
         label='Average training score')
plt.plot(C_val_exponents, np.mean(cv_test_roc_auc, axis=0), '-x',
         label='Average testing score')
plt.ylabel('ROC AUC')
plt.xlabel('log$_{10}$(C)')
plt.legend()
plt.title('Cross validation on Case Study problem')
np.mean(cv_test_roc_auc, axis=0)
```

You will obtain the following output:

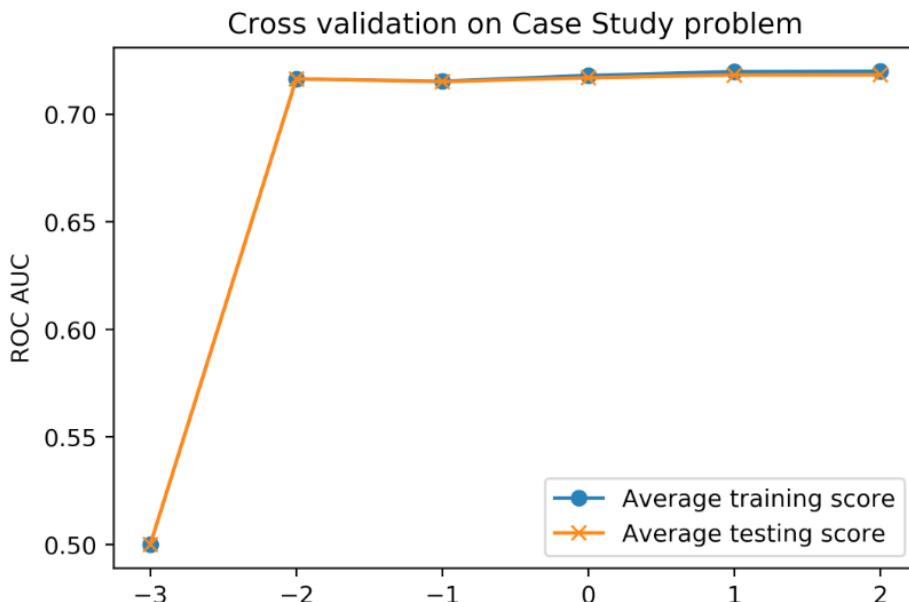


Figure 6.54: Cross-validation testing performance

You should notice that regularization does not impart much benefit here, as may be expected. While we are able to increase model performance over our previous efforts by using all the features available, it appears there is no overfitting going on. Instead, the training and testing scores are about the same. Instead of overfitting, it's possible that we may be underfitting. Let's try engineering some interaction features to see if they can improve performance.

11. Create interaction features for the case study data and confirm that the number of new features makes sense using the following code:

```
from sklearn.preprocessing import PolynomialFeatures
make_interactions = PolynomialFeatures(degree=2, interaction_only=True,
include_bias=False)
X_interact = make_interactions.fit_transform(X)
X_train, X_test, y_train, y_test = train_test_split(
X_interact, df['default payment next month'].values,
test_size=0.2, random_state=24)
print(X_train.shape)
print(X_test.shape)
```

You will obtain the following output:

```
(21331, 153)
(5333, 153)
```

From this you should see the new number of features is 153, which is $17 + "17 \text{ choose } 2" = 17 + 136 = 153$. The " $17 \text{ choose } 2$ " part comes from choosing all possible combinations of 2 features to interact from the possible 17.

12. Repeat the cross-validation procedure and observe the model performance now; that is, repeat Steps 9 and 10. Note that this will take substantially more time, due to the larger number of features, but it will probably take only a few minutes.

You will obtain the following output:

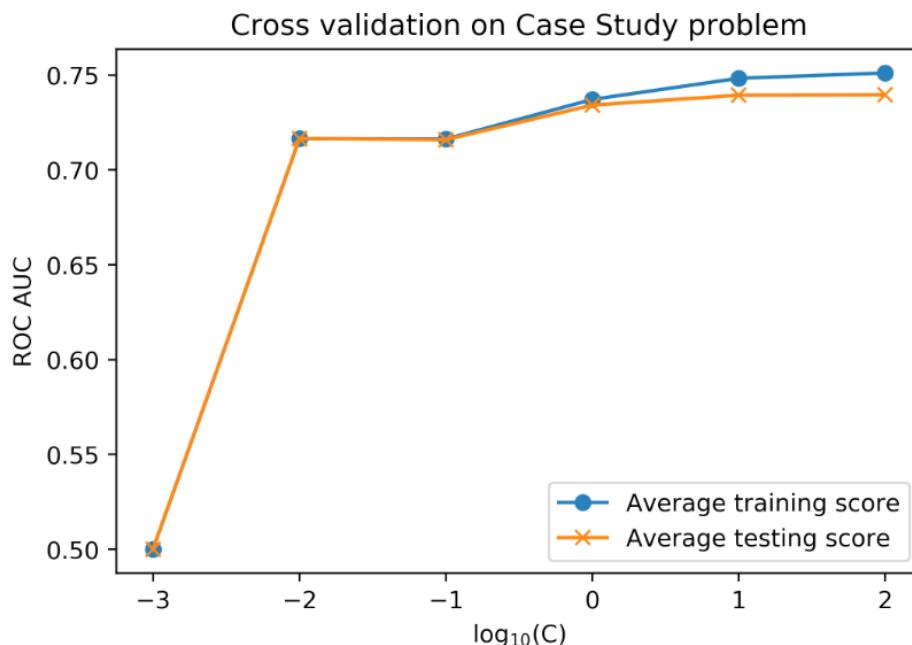


Figure 6.55: Improved cross-validation testing performance from adding interaction features

So, does the average cross-validation testing performance improve with the interaction features? Is regularization useful?

Engineering the interaction features increases the best model testing score to about ROC AUC = 0.74 on average across the folds, from about 0.72 without including interactions. These scores happen at $C = 100$, that is, with negligible regularization. On the plot of training versus testing scores for the model with interactions, you can see that the training score is a bit higher than the testing score, so it could be said that some amount of overfitting is going on. However, we cannot increase the testing score through regularization here, so this may not be a problematic instance of overfitting. In most cases, whatever strategy yields the highest testing score is the best strategy.

We will reserve the step of fitting on all the training data for later, when we've tried other models in cross-validation to find the best model.

Chapter 5: Decision Trees and Random Forests

Activity 5: Cross-Validation Grid Search with Random Forest

1. Create a dictionary representing the grid for the `max_depth` and `n_estimators` hyperparameters that will be searched in. Include depths of 3, 6, 9, and 12, and 10, 50, 100, and 200 trees. Leave the other hyperparameters at their defaults. Create the dictionary using this code:

```
rf_params = {'max_depth':[3, 6, 9, 12],  
             'n_estimators':[10, 50, 100, 200]}
```

Note

There are many other possible hyperparameters to search for. In particular, the scikit-learn documentation for random forest indicates the following:

"The main parameters to adjust when using these methods is `n_estimators` and `max_features`" and that "Empirical good default values are ... `max_features=sqrt(n_features)` for classification tasks."

Source: <https://scikit-learn.org/stable/modules/ensemble.html#parameters>

For the purposes of this book, we will use `max_features='auto'` (which is equal to `sqrt(n_features)`) and limit our exploration to `max_depth` and `n_estimators` for the sake of a shorter runtime. In a real-world situation, you should explore other hyperparameters according to how much computational time you can afford. Remember that in order to search in especially large parameter spaces, you can use `RandomizedSearchCV` to avoid exhaustively calculating metrics for every combination of hyperparameters in the grid that you specify.

2. Instantiate a `GridSearchCV` object using the same options that we have previously in this chapter, but with the dictionary of hyperparameters created in step 1 here. Set `verbose=2` to see the output for each fit performed. You can reuse the same random forest model object `rf` that we have been using. Instantiate the class using this code:

```
cv_rf = GridSearchCV(rf, param_grid=rf_params, scoring='roc_auc', fit_
params=None,
                    n_jobs=None, iid=False, refit=True, cv=4, verbose=2,
                    pre_dispatch=None, error_score=np.nan, return_train_
score=True)
```

3. Fit the `GridSearchCV` object on the training data. Perform the grid search using this code:

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

Because we chose the `verbose=2` option, you will see a relatively large amount of output in the notebook. There will be output for each combination of hyperparameters and for each fold, as it is fit and tested. Here are the first few lines of output:

```
Fitting 4 folds for each of 16 candidates, totalling 64 fits
[CV] max_depth=3, n_estimators=10 .....
[CV] ..... max_depth=3, n_estimators=10, total= 0.1s
[CV] max_depth=3, n_estimators=10 .....

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 1 out of 1 | elapsed: 0.1s remaining: 0.0s

[CV] ..... max_depth=3, n_estimators=10, total= 0.1s
[CV] max_depth=3, n_estimators=10 .....
[CV] ..... max_depth=3, n_estimators=10, total= 0.1s
[CV] max_depth=3, n_estimators=10 .....
[CV] ..... max_depth=3, n_estimators=10, total= 0.1s
[CV] max_depth=3, n_estimators=50 .....
[CV] ..... max_depth=3, n_estimators=50, total= 0.5s
```

Figure 6.56: The verbose output from cross-validation

While it's not necessary to see all this output for shorter cross-validation procedures, for longer ones it can be reassuring to see that the cross-validation is working and to give you an idea of how long the fits are taking for various combinations of hyperparameters. If things are taking too long, you may want to interrupt the kernel by pushing the stop button (square) at the top of the notebook and choose hyperparameters that will take less time to run or use a more limited set of hyperparameters.

When this is all done, you should see the following output:

```
[Parallel(n_jobs=1)]: Done 64 out of 64 | elapsed: 2.0min finished
GridSearchCV(cv=4, error_score=nan,
    estimator=RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
        max_depth=5, max_features='auto', max_leaf_nodes=None,
        min_impurity_decrease=0.0, min_impurity_split=None,
        min_samples_leaf=1, min_samples_split=2,
        min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=None,
        oob_score=False, random_state=1, verbose=0, warm_start=False),
    fit_params=None, iid=False, n_jobs=None,
    param_grid={'max_depth': [3, 6, 9, 12], 'n_estimators': [10, 50, 100, 200]},
    pre_dispatch=None, refit=True, return_train_score=True,
    scoring='roc_auc', verbose=2)
```

Figure 6.57: the cross-validation output upon completion

This cross-validation job took about two minutes to run. As your jobs grow, you may wish to explore parallel processing with the `n_jobs` parameter to see whether it's possible to speed up the search. Using `n_jobs=-1` and omitting the `pre_dispatch` option so the default is used, we were able to achieve a run-time of 52 seconds, compared to a 2-minute runtime with serial processing as shown here. However, with parallel processing, you won't be able to see the output of each individual model fitting operation as shown in *Figure 5.30*.

4. Put the results of the grid search in a pandas `DataFrame`. Use this code to put the results in a `DataFrame`:

```
cv_rf_results_df = pd.DataFrame(cv_rf.cv_results_)
```

5. Create a `pcolormesh` visualization of the mean testing score for each combination of hyperparameters. Here is the code to create a mesh graph of cross-validation results. It's similar to the example graph that we previously created, but with annotation that is specific to the cross-validation we performed here:

```
xx_rf, yy_rf = np.meshgrid(range(5), range(5))
cm_rf = plt.cm.jet
ax_rf = plt.axes()
pcolor_graph = ax_rf.pcolormesh(xx_rf, yy_rf, cv_rf_results_df['mean_test_score'].values.reshape((4,4)), cmap=cm_rf)
plt.colorbar(pcolor_graph, label='Average testing ROC AUC')
ax_rf.set_aspect('equal')
ax_rf.set_xticks([0.5, 1.5, 2.5, 3.5])
```

```

ax_rf.set_yticks([0.5, 1.5, 2.5, 3.5])
ax_rf.set_xticklabels([str(tick_label) for tick_label in rf_params['n_estimators']])
ax_rf.set_yticklabels([str(tick_label) for tick_label in rf_params['max_depth']])
ax_rf.set_xlabel('Number of trees')
ax_rf.set_ylabel('Maximum depth')

```

The main changes from our previous example are that instead of plotting the integers from 1 to 16, we're plotting the mean testing scores that we've retrieved and reshaped with `cv_rf_results_df['mean_test_score'].values.reshape((4,4))`. The other new things here are that we are using list comprehensions to create lists of strings for tick labels, based on the numerical values of hyperparameters in the grid. We access them from the dictionary that we defined, and then convert them individually to the `str` (string) data type within the list comprehension, for example: `ax_rf.set_xticklabels([str(tick_label) for tick_label in rf_params['n_estimators']])`. We have already set the tick locations to the places that we want the ticks using `set_xticks`. The graph should appear as follows:

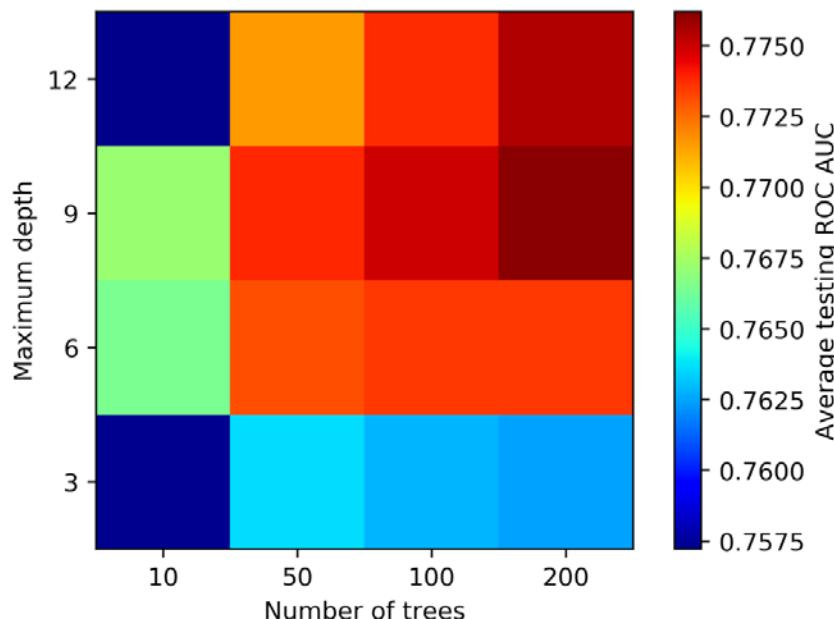


Figure 6.58: Results of cross-validation of a random forest over a grid with two hyperparameters

6. Conclude which set of hyperparameters to use.

What can we conclude from our grid search? There certainly seems to be an advantage to using trees with a depth of more than three. Of the parameter combinations that we tried, `max_depth=9` with 200 trees yields the best average testing score, which you can look up in the `DataFrame` and find is ROC AUC = 0.776. This is the best model we've found so far.

In a real-world scenario, we'd likely do a more thorough search. Some good next steps would be to try a larger number of trees and not spend any more time with `n_estimators < 200`, since we know that we need at least 200 trees to get the best performance. You may search a more granular space of `max_depth` instead of jumping by 3's as we've done here and try a couple of other hyperparameters such as `max_features`. However, for the book, we'll assume that we've done a more thorough search like this and concluded that `max_depth=9` and `n_estimators=200` is the optimal set of hyperparameters.

Chapter 6: Imputation of Missing Data, Financial Analysis, and Delivery to Client

Activity 6: Deriving Financial Insights

1. Using the testing set, calculate the cost of all defaults if there were no counseling program.

Use this code for the calculation:

```
cost_of_defaults = sum(y_test_all) * savings_per_default
cost_of_defaults
```

The output should be:

66308240.202088244

Figure 6.59: Cost of all defaults assuming no counseling

2. Calculate by what percent the cost of defaults can be decreased by the counseling program.

The potential decrease in cost of default is the greatest possible net savings of the counseling program, divided by the cost of all defaults in the absence of a program:

```
net_savings[max_savings_ix]/cost_of_defaults
```

The output should be:

0.2329472975431598

Figure 6.60: Fractional decrease in cost of defaults that could result from a counseling program

Results indicate that we can decrease the cost of defaults by 23% using a counseling program, guided by predictive modeling.

3. Calculate the net savings per account at the optimal threshold.

Use this code for the calculation:

```
net_savings[max_savings_ix]/len(y_test_all)
```

The output should be:

2601.2673223171373

Figure 6.61: Net savings per account possible with the counseling program

Results like these help the client scale the potential amount of savings they could create with the counseling program, to as many accounts as they serve.

4. Plot the net savings per account against the cost of counseling per account for each threshold.

Create the plot with this code:

```
plt.plot(cost_of_all_counselings/len(y_test_all), net_savings/len(y_test_all))
plt.xlabel('Upfront investment: cost of counselings per account (NT$)')
plt.ylabel('Net savings per account (NT$)')
```

The resulting plot should appear like this:

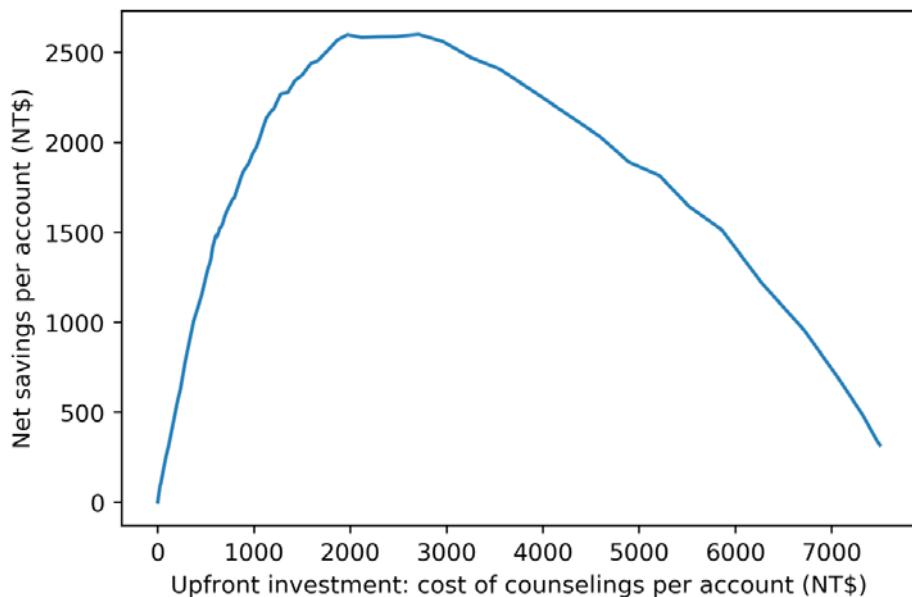


Figure 6.62: The initial cost of the counseling program needed to achieve a given amount of savings

This indicates how much money the client needs to budget to the counseling program in a given month, to achieve a given amount of savings. It looks like the greatest benefit can be created by budgeting up to about NT\$2000 per account. After this, net savings are relatively flat, and then decline. The client may not actually be able to budget this much to the program. However, this graphic gives them evidence to argue for a larger budget if they need to.

This result corresponds to our graphic from the previous exercise. Although we've shown the optimal threshold is 0.2, it may be fine for the client to use a higher threshold up to about 0.25, thus making fewer positive predictions, offering counseling to fewer account holders, and having a smaller upfront program cost. *Figure 6.62* shows how this plays out in terms of cost and net savings per account.

5. Plot the fraction of accounts predicted as positive (this is called the "flag rate") at each threshold.

Use this code to plot the flag rate against the threshold:

```
plt.plot(thresholds, n_pos_pred/len(y_test_all))
plt.ylabel('Flag rate')
plt.xlabel('Threshold')
```

The plot should appear as follows:

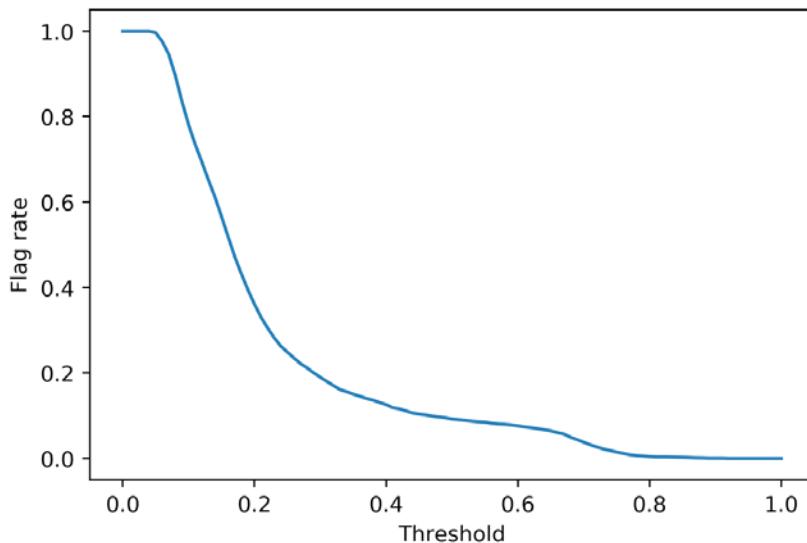


Figure 6.63: Flag rate against threshold for the credit counseling program

This plot shows the fraction of people who will be predicted to default, and therefore recommended outreach at each threshold. It appears that at the optimal threshold of 0.2, only about 30% of accounts will be flagged for counseling. This shows how using a model to prioritize accounts for counseling can help focus on the right accounts and reduce wasted resources. Higher thresholds, which may result in nearly-optimal savings up to a threshold of about 0.25 as shown in *Figure 6.36*, (*Exercise 25, Characterizing Costs and Savings*), result in lower flag rates.

6. Plot a precision-recall curve for the testing data using the following code:

```
plt.plot(n_true_pos/sum(y_test_all), np.divide(n_true_pos, n_pos_pred))
plt.xlabel('Recall')
plt.ylabel('Precision')
```

The plot should look like this:

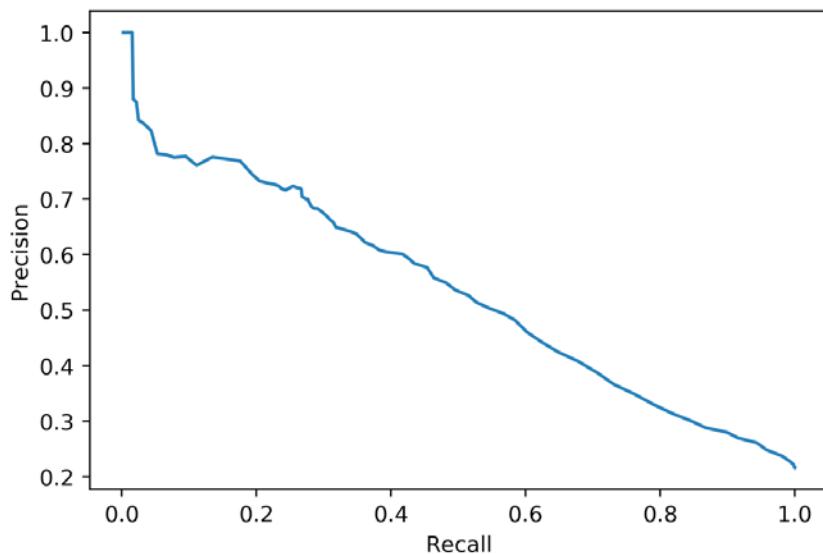


Figure 6.64: Precision-recall curve

Figure 6.64 shows that in order to start getting a true positive rate (that is, recall) much above 0, we need to accept a precision of about 0.75 or lower. So, it appears there is room for improvement in our model. While this would not necessarily be communicated to the client, it shows that by using more advanced modeling techniques, or a richer set of features, model performance could be improved.

Precision and recall have a direct link to the cost and savings of the program: the more precise our predictions are, the less money we are wasting on counseling due to incorrect model predictions. And, the higher the recall, the more savings we can create by successfully identifying accounts that would default. Compare the code in this step to the code used to calculate cost and savings in the previous exercise to see this. This links the financial analysis to machine learning metrics we have examined earlier in the case study.

To see the connection of precision and recall with the threshold used to define positive and negative predictions, it can be instructive to plot them separately.

7. Plot precision and recall separately on the y-axis against threshold on the x-axis.

Use this code to produce the plot:

```
plt.plot(thresholds, np.divide(n_true_pos, n_pos_pred), label='Precision')
plt.plot(thresholds, n_true_pos/sum(y_test_all), label='Recall')
plt.xlabel('Threshold')
plt.legend()
```

The plot should appear as follows:

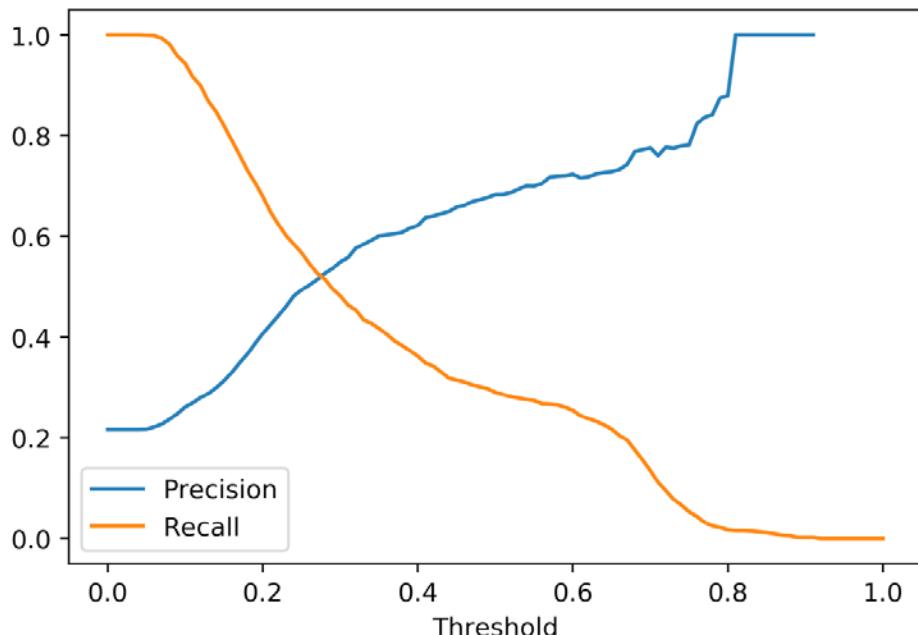
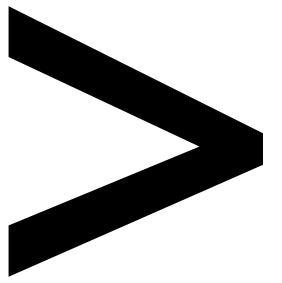


Figure 6.65: Precision and recall plotted separately against threshold

This plot sheds some light on why the optimal threshold turned out to be 0.2. While the optimal threshold also depends on the financial analysis of costs and savings, we can see here that the steepest part of the initial increase in precision, which represents the correctness of positive predictions and is therefore a measure of how cost-effective the model-guided counseling can be, happens up to a threshold of about 0.2.



Index

About

All major keywords used in this book are captured alphabetically in this section. Each one is accompanied by the page number of where they appear.

A

accuracy: 85-88, 94, 100, 106, 200, 296-300
analyses: 2, 126, 196, 276
anomalies: 69
argmax: 315
arithmetic: 132, 135
astype: 37, 208, 314
automated: 194
average: 44-45, 67, 86, 116, 119, 122, 147, 167-168, 207, 219, 238, 248, 254, 256-258, 265-266, 270, 272, 287, 292-294, 298, 300, 305-306, 310-312
axhline: 128

B

barstacked: 99
baseline: 126, 309
binary: 45, 65-66, 68, 81-82, 85-89, 93-94, 103, 106, 117-118, 124, 146, 148, 153-155, 179, 181, 184, 226, 236, 239, 275-276, 294, 296, 309, 319, 321
binning: 56, 134
boolean: 24-27, 29-31, 36, 59, 62, 124, 208, 211, 217, 232-233, 282, 284, 313
bootstrap: 260, 262, 291
boundaries: 162
budgeting: 275

C

calculus: 102, 170, 173
categorical: 127, 280
clause: 113

code-style: 15
colorbar: 115, 269
colormap: 269
counts: 4, 23-24, 27-28, 34-35, 37, 41-43, 52-54, 57-58, 60, 106, 127, 284, 288, 299, 303
c-value: 206

D

database: 60
dataframe: 13-14, 16-17, 21-22, 29-32, 34, 37, 43, 47, 49-51, 54, 61-62, 73-75, 111, 114, 120, 122, 124, 128-129, 147, 149-150, 218, 228-229, 254-256, 264-267, 271, 282-283, 285, 292, 294, 302
dataframes: 4, 13, 26, 50, 286
dataset: 2, 9, 11, 16-21, 23, 27, 32, 35, 41, 44, 47, 51, 53-54, 60-61, 63, 66-67, 126-128, 179, 184-186, 191, 196, 199-200, 215, 241, 246-247, 260, 276, 278-279, 281-283, 290, 294, 296, 316
datatype: 1
delete: 30-31
deviation: 41, 76-77, 116-117, 185, 255-257, 264, 292, 305, 308
dollar: 19, 21-22, 41, 139
dollars: 41
domain: 106
downsample: 118
dubious: 27

duplicate: 27-32, 68, 282-283

E

ecosystem: 55, 69
edgecolors: 158, 160
educated: 276, 280, 321
entropy: 167, 239-240
errorbar: 256, 264
exponent: 137

F

facecolors: 158, 160
feature: 2, 31, 35, 41-47, 51, 53-55, 59-60, 62-63, 68, 72-79, 84, 90, 98, 103, 105-106, 109, 112, 116, 119-120, 122-127, 129, 134, 144, 146-147, 149-150, 152-153, 159, 162-163, 165-166, 177, 179-181, 184, 186, 189-192, 194, 211-212, 215-219, 226-228, 230, 232-233, 238, 240-241, 246, 262, 266-267, 278-282, 287, 289-290, 294, 300, 312, 321
features: 4, 9-10, 18-19, 21, 29, 31-34, 37-41, 43-46, 51-54, 57-63, 66, 68-69, 72-76, 79, 82, 84-87, 94, 103, 106, 109-127, 129, 133-134, 142, 144, 146-147, 150-153, 155-156, 162-163, 165, 177-179, 181-186, 189-196, 199, 202-203, 211-212, 215-219, 223, 226, 228-230,

232-234, 236, 238,
240-241, 246-248, 252,
258-259, 261-262, 267,
271, 276-277, 279-283,
285-288, 291-295, 297,
301-302, 304, 306-308,
311-312, 319-321

female:21
fourth:18

G

gareth:190
gaussian: 76-77, 224
gradient: 165-166, 168-176,
191, 219, 224, 246
graphical: 19, 38, 54,
125, 128, 230, 232
graphics: 12, 20, 27,
38, 61, 126
graphs: 2, 63, 133
graphviz: 223, 227,
230-232
greedy:241
groupby: 45, 68, 110,
127, 144, 147, 149
grouped: 24, 56, 128
grouping: 51, 55, 68, 147
groups: 38, 51, 124,
127-128, 147-148,
232, 310, 321
guardrails:189

H

hypercube:185
hyperplane:162
hypotheses: 125-126, 182
hypothesis:133

I

if-then:319
increment: 124, 131,
204, 311, 314
indexing: 2-4, 29, 31-32,
98, 119, 123, 264
indices: 2-4, 13, 27, 84,
123, 157-158, 160, 197,
199-200, 203, 248
in-line:12
integer: 4, 13, 31, 34-35,
53, 59, 119, 125, 201,
208, 258, 263, 287
integers: 3, 25-26, 34,
76, 269-270, 289
integral:131
integrals:102
isolate: 54, 99, 247, 284

K

kfolds: 292-293
k-folds: 197-203, 207, 212

L

legendre:224
liblinear: 71, 156, 162,
187-188, 191, 210
logarithms:137
logical: 25, 27, 29, 31-32,
35, 86, 90, 98, 123,
129, 225, 233, 278
logistic: 45-46, 70-73,
79-80, 85, 90, 94,
100-101, 103, 105-106,
109-110, 116, 118, 134,
136, 142-147, 150-153,
156, 159, 161-163,
165-169, 175-177,
181-182, 187-190, 192,

194-195, 200, 209-212,
214-220, 224, 240,
246-248, 258, 261, 267,
271, 277-278, 296, 319
log-loss: 165, 167-168,
189-190
log-odds:189

M

matlab:2
matplotlib: 38-39, 54-56,
67, 75, 77, 97, 105, 109,
111, 126, 128, 131-133,
163, 227, 256, 268
matrices: 4, 137
matrix: 30, 88-90,
92-94, 106, 114-115,
118, 125-126, 155
meshgrid:268
message:12
metadata: 9, 19,
32-33, 37, 295
microsoft:16

N

navigate: 12, 14
nbviewer:143
nunique: 23, 32, 283

O

one-hot: 1, 46,
49-50, 112, 181
operators:103
ordinal: 44-48, 51, 280
output: 6-8, 14, 16, 20-29,
31-37, 39, 41-43, 46-50,
52-53, 56-57, 59, 61,
66, 71, 73-76, 78-79,
83-84, 90-91, 93,

P

pcolor:269
pcolormesh: 269-271
pearson: 114, 116-118,
163, 177
penalize:246
penalized: 215, 246
penalties: 188, 190
penalty: 71, 187-192,
210, 218-219
percentile: 40, 122
polynomial: 134, 144,
177, 183, 192, 216-217
predicting: 10, 104,
142, 189, 234
prediction: 10, 66, 75,
81, 87-88, 90, 94, 104,
106, 116, 144, 152, 157,
168, 189, 191, 200, 208,
220, 234-236, 238, 248,
261, 279, 293, 303

predictive: 2, 9-10, 18,
32-33, 53, 57, 60, 66,
69, 75, 80-82, 104, 110,
116-117, 119, 125, 133-134,
163, 184, 191, 193, 195,
211, 220, 224, 258, 272,
276, 281, 294-295,
309, 311, 317-321
predictors: 125, 179
predicts: 87, 100, 151
p-value: 118-121
p-values: 70, 118, 120,
122, 124-125
pydata: 16, 54
pyplot: 39, 55, 67,
111, 132, 227
python: 1-9, 13-14, 25,
35, 55, 62, 66, 69-70,
75, 90, 92, 109-113,
134-138, 141, 143, 202,
224, 227, 261, 263
pythonic:112
pytorch:70

Q

quadratic:183
quartiles:41

A

randint:25
random: 25-26, 46,
70-71, 75-77, 79, 83-85,
101-103, 125, 134, 153,
162, 182, 185, 187-189,
195-196, 198-199,
209-210, 218, 220,
223-224, 229, 241,
258-263, 265-268,
270-272, 276, 278,

280-281, 286-294,
296-298, 300-301,
303-306, 311, 319
randomized:212
randomly: 68, 84,
200, 212, 249, 260,
289-290, 319
randomness:266
ranges:212
rcparams: 39, 58, 67,
77, 97, 111, 132, 178,
227, 237, 315

S

seaborn: 111, 114-115, 118
sklearn: 70, 78, 83, 120,
122, 156, 184-185, 187,
189, 196, 201, 212,
215-216, 229, 232, 252,
262, 287, 290-291,
297, 300, 308
slight:132
software: 63, 117, 319
solution: 62, 105, 117,
163, 168, 170, 175-176,
218, 236, 246, 271,
279, 309, 318
solutions: 207, 241
solver: 71-72, 156, 162,
168-169, 187-188,
190-192, 210,
215, 217-219
solvers: 187, 191-192
spanning:82
spatial: 179, 280, 294
splits: 196-199, 202,
234-236, 238, 240-241,
252, 259, 264, 291
splitter: 202, 241, 292-293

splitting: 82, 84, 196,
199, 232, 238, 241,
246, 291, 306
sql-like:13
square: 23, 27, 102,
144, 259, 269
squared: 248, 257
squares: 190, 268
squaring:166
statistic: 68, 118, 120, 280
statistics: 18, 39-41, 52,
70, 116, 125, 224
stepwise:194
summation: 167, 191, 239
supervised: 11, 72, 281, 294

Y

y-axes:117
y-axis: 44, 100, 128, 132,
140, 152, 205, 290, 318

T

tabular: 9, 40
temporal: 179, 280, 294
tensorflow:70

V

vector: 76, 209
vectorized: 137-138

X

x-axis: 62, 100, 130,
152, 205, 318
xgboost: 70, 261
xlabel: 45, 56, 97, 99, 101,
130, 132, 150, 154, 158,
160, 169, 172, 174-175,
205, 207-208, 237, 256,
264, 269, 315, 320
xticks: 289, 300, 315
x-values:141