

Artificial Intelligence with Python **Cookbook**

Proven recipes for applying AI algorithms and deep learning techniques using TensorFlow 2.x and PyTorch 1.6

Ben Auffarth



Artificial Intelligence with Python Cookbook

Proven recipes for applying AI algorithms and deep learning techniques using TensorFlow 2.x and PyTorch 1.6

Ben Auffarth

Packt

BIRMINGHAM - MUMBAI

Artificial Intelligence with Python Cookbook

Copyright © 2020 Packt Publishing

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

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

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

Commissioning Editor: Sunith Shetty

Acquisition Editor: Siddharth Mandal

Content Development Editor: Nathanya Dias

Senior Editor: David Sugarman

Technical Editor: Sonam Pandey

Copy Editor: Safis Editing

Project Coordinator: Aishwarya Mohan

Proofreader: Safis Editing

Indexer: Priyanka Dhadke

Production Designer: Jyoti Chauhan

First published: October 2020

Production reference: 1291020

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78913-396-7

www.packt.com



Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and videos from over 4,000 industry professionals
- Improve your learning with skill plans tailored especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and, as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Ben Auffarth is a full-stack data scientist with more than 15 years of work experience. With a background and Ph.D. in computational and cognitive neuroscience, he has designed and conducted wet lab experiments on cell cultures, analyzed experiments with terabytes of data, run brain models on IBM supercomputers with up to 64k cores, built production systems processing hundreds of thousands of transactions per day, and trained neural networks on millions of text documents. He resides in West London with his family, where you might find him in a playground with his young son. He co-founded and is the former president of Data Science Speakers, London.

I am deeply grateful to the editors at Packt, who provided practical help and competent advice, and to everyone who has been close to me and supported me, especially my partner Diane. This book is dedicated to Diane and my son, Nicholas.

About the reviewers

va barbosa is a software engineer in the Qiskit Community at IBM, focused on building open source tools and creating educational content for developers, researchers, students, and educators in the field of quantum computing. Previously, va was a developer advocate at the Center for Open-Source Data and AI Technologies where he assisted developers discover and make use of data science and machine learning technologies. He is fueled by his passion to help others and guided by his enthusiasm for open source technology.

Eyal Wirsansky is an Artificial Intelligence consultant and mentor, a senior software engineer, and a technology community leader. Eyal leads the Jacksonville (FL) Java User Group, hosts the Artificial Intelligence for Enterprise virtual user group, writes a developer-oriented Artificial Intelligence blog, and often speaks about topics related to Artificial Intelligence, Machine Learning and Genetic Algorithms. Eyal is the author of the book *Hands-On Genetic Algorithms with Python*.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface	1
Chapter 1: Getting Started with Artificial Intelligence in Python	8
Technical requirements	8
Setting up a Jupyter environment	9
Getting ready	9
How to do it...	10
Installing libraries with Google Colab	10
Self-hosting a Jupyter Notebook environment	11
How it works...	14
There's more...	14
See also	15
Getting proficient in Python for AI	16
Getting ready	17
How to do it...	18
Obtaining the history of Jupyter commands and outputs	18
Execution history	19
Outputs	19
Auto-reloading packages	19
Debugging	20
Timing code execution	21
Displaying progress bars	22
Compiling your code	23
Speeding up pandas DataFrames	25
Parallelizing your code	26
See also	28
Classifying in scikit-learn, Keras, and PyTorch	29
Getting ready	29
How to do it...	30
Visualizing data in seaborn	31
Modeling in scikit-learn	33
Modeling in Keras	35
Modeling in PyTorch	42
How it works...	45
Neural network training	45
The SELU activation function	49
Softmax activation	49
Cross-entropy	49
See also	50
Modeling with Keras	51
Getting ready	52
How to do it...	53

Data loading and preprocessing	54
Model training	62
How it works...	66
Maximal information coefficient	66
Data generators	67
Permutation importance	68
See also	68
Chapter 2: Advanced Topics in Supervised Machine Learning	70
Technical requirements	71
Transforming data in scikit-learn	71
Getting ready	72
How to do it...	73
Encoding ranges numerically	73
Deriving higher-order features	76
Combining transformations	78
How it works...	80
There's more...	81
See also	82
Predicting house prices in PyTorch	82
Getting ready	82
How to do it...	85
How it works...	92
There's more...	93
See also	95
Live decisioning customer values	96
Getting ready	96
How to do it...	97
How it works...	100
Active learning	101
Hoeffding Tree	101
Class weighting	101
See also	102
Battling algorithmic bias	102
Getting ready	103
How to do it...	105
How it works...	113
There's more...	114
See also	117
Forecasting CO₂ time series	118
Getting ready	118
How to do it...	119
Analyzing time series using ARIMA and SARIMA	121
How it works...	123
There's more...	126
See also	127

Chapter 3: Patterns, Outliers, and Recommendations	129
Clustering market segments	130
Getting ready	130
How to do it...	131
How it works...	136
There's more...	138
See also	140
Discovering anomalies	141
Getting ready	141
How to do it...	142
How it works...	149
k-nearest neighbors	149
Isolation forest	149
Autoencoder	149
See also	150
Representing for similarity search	150
Getting ready	151
How to do it...	152
Baseline – string comparison functions	153
Bag-of-characters approach	154
Siamese neural network approach	155
How it works...	159
Recommending products	160
Getting ready	160
How to do it...	162
How it works...	164
Precision at k	165
Matrix factorization	165
The lightfm model	166
See also	166
Spotting fraudster communities	167
Getting ready	167
How to do it...	168
Creating an adjacency matrix	168
Community detection algorithms	169
Evaluating the communities	170
How it works...	172
Graph community algorithms	172
Louvain algorithm	172
Girvan–Newman algorithm	173
Information entropy	173
There's more...	173
See also	175
Chapter 4: Probabilistic Modeling	176
Technical requirements	177
Predicting stock prices with confidence	177

Getting ready	177
How to do it...	178
How it works...	182
Featurization	182
Platt scaling	183
Isotonic regression	184
Naive Bayes	184
See also	185
Estimating customer lifetime value	186
Getting ready	186
How to do it...	186
How it works...	188
The BG/NBD model	189
The Gamma-Gamma model	189
See also	189
Diagnosing a disease	190
Getting ready	190
How to do it...	191
How it works...	196
Aleatoric uncertainty	196
Negative log-likelihood	197
Bernoulli distribution	197
Metrics	197
See also	198
Stopping credit defaults	198
Getting ready	198
How to do it...	199
How it works...	203
Epistemic uncertainty	203
See also	204
Chapter 5: Heuristic Search Techniques and Logical Inference	205
Making decisions based on knowledge	205
Getting ready	206
How to do it...	206
Logical reasoning	207
Knowledge embedding	207
How it works...	211
Logical reasoning	212
Logic provers	213
Knowledge embedding	213
Graph embedding with Walklets	213
See also	214
Solving the n-queens problem	216
Getting ready	216
How to do it...	217
Genetic algorithm	217

Particle swarm optimization	221
SAT solver	225
How it works...	227
Genetic algorithm	227
Particle swarm optimization	229
SAT solver	230
See also	232
Finding the shortest bus route	232
Getting ready	232
How to do it...	233
Simulated annealing	233
Ant colony optimization	235
How it works...	236
Simulated annealing	237
Ant colony optimization	237
See also	238
Simulating the spread of a disease	239
Getting ready	239
How to do it...	240
How it works...	247
There's more...	249
See also	250
Writing a chess engine with Monte Carlo tree search	250
Getting ready	250
How to do it...	251
Tree search	251
Implementing a node	253
Playing chess	254
How it works...	256
There's more...	257
See also	258
Chapter 6: Deep Reinforcement Learning	259
Technical requirements	260
Optimizing a website	260
How to do it...	260
How it works...	266
See also	268
Controlling a cartpole	268
Getting ready	269
How to do it...	269
How it works...	275
There's more...	276
Watching our agents in the environment	276
Using the RLlib library	277
See also	277
Playing blackjack	278

Getting ready	278
How to do it...	279
How it works...	285
See also	287
Chapter 7: Advanced Image Applications	288
Technical requirements	288
Recognizing clothing items	288
Getting ready	290
How to do it...	290
Difference of Gaussians	291
Multilayer perceptron	292
LeNet5	294
MobileNet transfer learning	296
How it works...	297
Difference of Gaussian	298
LeNet5	298
MobileNet transfer learning	299
See also	300
Generating images	301
Getting ready	301
How to do it...	301
How it works...	308
See also	309
Encoding images and style	309
Getting ready	310
How to do it...	310
How it works...	319
See also	321
Chapter 8: Working with Moving Images	322
Technical requirements	323
Localizing objects	323
Getting ready	323
How to do it...	324
How it works...	327
There's more...	328
See also	329
Faking videos	330
Getting ready	330
How to do it...	331
How it works...	333
See also	336
Deep fakes	336
Detection of deep fakes	337
Chapter 9: Deep Learning in Audio and Speech	338

Technical requirements	339
Recognizing voice commands	339
Getting ready	339
How to do it...	340
How it works...	346
See also	346
Synthesizing speech from text	347
Getting ready	347
How to do it...	348
How it works...	350
Deep Convolutional Networks with Guided Attention	350
WaveGAN	351
There's more...	352
See also	353
Generating melodies	354
Getting ready	354
How to do it...	355
How it works...	359
See also	360
Chapter 10: Natural Language Processing	361
Technical requirements	362
Classifying newsgroups	362
Getting ready	362
How to do it...	363
Bag-of-words	364
Word embeddings	364
Custom word embeddings	366
How it works...	367
The CBOW algorithm	368
TFIDF	369
There's more...	369
See also	371
Chatting to users	372
Getting ready	373
How to do it...	373
How it works...	377
ELIZA	377
Eywa	378
See also	379
Translating a text from English to German	380
Getting ready	380
How to do it...	381
How it works...	392
There's more...	394
See also	394

Writing a popular novel	395
Getting ready	396
How to do it...	397
How it works...	400
See also	401
Chapter 11: Artificial Intelligence in Production	402
Technical requirements	402
Visualizing model results	403
Getting ready	403
How to do it...	403
Streamlit hello-world	403
Creating our data app	404
How it works...	411
See also	412
Serving a model for live decisioning	413
Getting ready	413
How to do it...	414
How it works...	418
Monitoring	419
See also	420
Securing a model against attack	421
Getting ready	422
How to do it...	422
How it works...	429
Differential privacy	430
Private aggregation of teacher ensembles	431
See also	432
Other Books You May Enjoy	434
Index	437

Preface

Artificial Intelligence (AI) is the field concerned with automating tasks in a way that exhibits some form of intelligence to human spectators. This apparent intelligence could be similar to human intelligence, or simply some insightful action a machine or program surprises us with. Since our understanding of the world improves along with our tools, our expectations of what would surprise us or strike us as intelligent are continuously being raised. Rodney Brooks, a well-known researcher in the field of AI, expressed this effect (often referred to as the **AI effect**):

Every time we figure out a piece of it, it stops being magical; we say, "Oh, that's just a computation." We used to joke that AI means "almost implemented."

(Cited from Kahn, Jennifer (March 2002). *It's Alive*, in *Wired*, 10 (30): <https://www.wired.com/2002/03/everywhere/>)

AI has made huge strides, especially over the last few years with the arrival of powerful hardware, such as **Graphics Processing Units (GPUs)** and now **Tensor Processing Units (TPUs)**, that can facilitate more powerful models, such as deep learning models with hundreds of thousands, millions, or even billions of parameters. These models perform better and better on benchmarks, often reaching human or even super-human levels. Excitingly for anyone involved in the field, some of these models, trained for many thousands of hours that would be worth hundreds of thousands of dollars if run on **Amazon Web Services (AWS)**, are available for download to play with and extend.

This giant leap in performance is especially remarkable in image processing, audio processing, and increasingly in natural language processing. Nowhere has this been as evident and as showcased in media as it has in games. While the 1997 chess match between Kasparov and Deep Blue is still in the mind of many people, it can be argued that the success of the machine against the human chess champion was mostly due to the brute-force searching and analyzing of 200 million positions per second on a powerful supercomputer. Since then, however, a combination of algorithmic and computational capacities has given machines proficiency and mastery in even more complex games.

The following table illustrates the progress in AI:

Game	Champion year	Legal states (\log_{10})
Othello (reversi)	1997	28
Draughts (checkers)	1994	21
Chess	1997	46
Scrabble	2006	
Shogi	2017	71
Go	2016	172
2p no-limit hold 'em	2017	
Starcraft	-	270+

Please refer to the Wikipedia article *Progress in Artificial Intelligence* for more information. You can see, for a series of games of varying complexity (as per the third column, showing legal states in powers of 10), when AI reached the level of top human players. More generally, you can find out more about state-of-the-art performances in different disciplines on a dedicated website: <https://paperswithcode.com/sota>.

It is therefore more timely than ever to look at and learn to use the state-of-the-art methods in AI, and this is what this book is about. You'll find carefully chosen recipes that will help you refresh your knowledge and bring you up to date with cutting edge algorithms.

If you are looking to build AI solutions for work or even for your hobby projects, you will find this cookbook useful. With the help of easy-to-follow recipes, this book will take you through the AI algorithms required to build smart models for problem solving. By the end of this book, you'll be able to identify an AI approach for solving applied problems, implement and test algorithms, and deal with model versioning, reports, and monitoring.

Who this book is for

This AI machine learning book is for Python developers, data scientists, machine learning engineers, and deep learning practitioners who want to learn how to build artificial intelligence solutions with easy-to-follow recipes. You'll also find this book useful if you're looking for state-of-the-art solutions to perform different machine learning tasks in various use cases. Basic working knowledge of the Python programming language and machine learning concepts will help you to work with code effectively in this book.

What this book covers

Chapter 1, *Getting Started with Artificial Intelligence in Python*, describes a basic setup with Python for data crunching and AI. We'll perform data loading in pandas, plotting, and writing first models in scikit-learn and Keras. Since data preparation is such a time-consuming activity, we will present state-of-the-art techniques to facilitate this activity.

Chapter 2, *Advanced Topics in Supervised Machine Learning*, explains how to deal with common issues in supervised machine learning problems, such as class imbalance, time series, and dealing with algorithmic bias.

Chapter 3, *Patterns, Outliers, and Recommendations*, goes through an example involving clustering in real-world situations, and how to detect anomalies and outliers in data using sklearn and Keras. Then we will cover how to build a nearest neighbor search for fuzzy string matching, collaborative filtering by building a latent space, and fraud detection in a graph network.

Chapter 4, *Probabilistic Modeling*, explains how to build probabilistic models for predicting stock prices, and how we estimate customer lifetimes, diagnose a disease, and quantify credit risk under conditions of uncertainty.

Chapter 5, *Heuristic Search Techniques and Logical Inference*, introduces a broad class of problem solving tools, starting with ontologies and knowledge-based reasoning, through to optimization in the context of satisfiability, and combinatorial optimization with methods such as Particle Swarm Optimization, a genetic algorithm. We will simulate the spread of a pandemic in a multi-agent system, implement a Monte-Carlo tree search for a chess engine, we'll write a basic logic solver, and we'll embed knowledge through a graph algorithm.

Chapter 6, *Deep Reinforcement Learning*, applies multi-armed bandits to website optimization, and implements the REINFORCE algorithm for control tasks and a deep Q network for a simple game.

Chapter 7, *Advanced Image Applications*, takes you on a journey from more basic to state-of-the-art approaches in image recognition. We'll then learn how to create image samples using generative adversarial networks, and then perform style transfer using an adversarial autoencoder.

Chapter 8, *Working with Moving Images*, starts with image detection on a video feed and then creates videos using a deep fake model.

Chapter 9, *Deep Learning in Audio and Speech*, classifies different voice commands, before going through a text-to-speech architecture, and concludes with a recipe for modeling and generating sequences of music with a recurrent neural network.

Chapter 10, *Natural Language Processing*, explains how to classify sentiment, create a chatbot, and translate a text using sequence-to-sequence models. Finally, we'll attempt to write a popular novel using state-of-the-art text generation models.

Chapter 11, *Artificial Intelligence in Production*, covers monitoring and model versioning, visualizations as dashboards, and explains how to secure a model against malicious hacking attacks that could leak user data.

To get the most out of this book

Chapter 1, *Getting Started with Artificial Intelligence in Python*, goes through the requirements and a suitable setup, along with the installation of an environment in detail, and you'll also learn a lot of practical tricks to become more productive. Generally, you'll need to set up Python with libraries such as scikit-learn, PyTorch, and TensorFlow. We recommend a GPU or online services such as Google Colab. All code samples have been tested on Linux and macOS. Where necessary, however, we comment in relation to Windows operating systems.

Some of the software and libraries most prominently covered in this book are listed in the following table:

Software/hardware covered in the book	OS requirements
Python 3.6 or later	Windows, macOS X, and Linux (any)
TensorFlow 2.0 or later	Windows, macOS X, and Linux (any)
PyTorch 1.6 or later	Windows, macOS X, and Linux (any)
Pandas 1.0 or later	Windows, macOS X, and Linux (any)
Scikit-learn 0.22.0 or later	Windows, macOS X, and Linux (any)

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

Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/Artificial-Intelligence-with-Python-Cookbook>. In case there's an update to the code, it will be updated on the existing GitHub repository.

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

Download the color images

You can download the example code files for this book from GitHub at https://static.packt-cdn.com/downloads/9781789133967_ColorImages.pdf. In case there's an update to the code, it will be updated on the existing GitHub repository.

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

Conventions used

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

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Here is the simplified code for `RangeTransformer`."

A block of code is set as follows:

```
import openml
dataset = openml.datasets.get_dataset(40536)
X, y, categorical_indicator, _ = dataset.get_data(
    dataset_format='DataFrame',
    target=dataset.default_target_attribute
)
```

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

```
import operator
operator.sub(1, 2) == 1 - 2
# True
```

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

```
$ mkdir css
$ cd css
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Sections

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

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

Getting ready

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

How to do it...

This section contains the steps required to follow the recipe.

How it works...

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

There's more...

This section consists of additional information about the recipe in order to further your knowledge of it.

See also

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

Get in touch

Feedback from our readers is always welcome.

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

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

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

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

Reviews

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

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

1

Getting Started with Artificial Intelligence in Python

In this chapter, we'll start by setting up a Jupyter environment to run our experiments and algorithms in, we'll get into different nifty Python and Jupyter hacks for **artificial intelligence** (AI), we'll do a toy example in scikit-learn, Keras, and PyTorch, and then a slightly more elaborate example in Keras to round things off. This chapter is largely introductory, and a lot of what see in this chapter will be built on in subsequent chapters as we get into more advanced applications.

In this chapter, we'll cover the following recipes:

- Setting up a Jupyter environment
- Getting proficient in Python for AI
- Classifying in scikit-learn, Keras, and PyTorch
- Modeling with Keras

Technical requirements

You really should have a GPU available in order to run some of the recipes in this book, or you would better off using Google Colab. There are some extra steps required to make sure you have the correct NVIDIA graphics drivers installed, along with some additional libraries. Google provides up-to-date instructions on the TensorFlow website at <https://www.tensorflow.org/install/gpu>. Similarly, PyTorch versions have minimum requirements for NVIDIA driver versions (which you'd have to check manually for each PyTorch version). Let's see how to use dockerized environments to help set this up.

You can find the recipes in this chapter in the GitHub repository of this book at <https://github.com/PacktPublishing/Artificial-Intelligence-with-Python-Cookbook>.

Setting up a Jupyter environment

As you are aware, since you've acquired this book, Python is the dominant programming language in AI. It has the richest ecosystem of all programming languages, including many implementations of state-of-the-art algorithms that make using them often a matter of simply importing and setting a few selected parameters. It should go without saying that we will go beyond the basic usage in many cases and we will talk about a lot of the underlying ideas and technologies as we go through the recipes.

We can't emphasize enough the importance of being able to quickly prototype ideas and see how well they work as part of a solution. This is often the main part of AI or data science work. A **read-eval-print loop (REPL)** is essential for quick iteration when turning an idea into a prototype, and you want functionality such as edit history, graphing, and more. This explains why Jupyter Notebook (where **Jupyter** is short for **Julia, Python, R**) is so central to working in AI.



Please note that, although we'll be focusing on Jupyter Notebook, or Google Colab, which runs Jupyter notebooks in the cloud, there are a few functionally similar alternatives around such as JupyterLab or even PyCharm running with a remote interpreter. Jupyter Notebook is still, however, the most popular (and probably the best supported) choice.

In this recipe, we will make sure we have a working Python environment with the software libraries that we need throughout this book. We'll be dealing with installing relevant Python libraries for working with AI, and we'll set up a Jupyter Notebook server.

Getting ready

Firstly, ensure you have Python installed, as well as a method of installing libraries. There are different ways of using and installing libraries, depending on the following two scenarios:

- You use one of the services that host interactive notebooks, such as Google Colab.
- You install Python libraries on your own machine(s).



In Python, a **module** is a Python file that contains functions, variables, or classes. A **package** is a collection of modules within the same path. A **library** is a collection of related functionality, often in the form of different packages or modules. Informally, it's quite common to refer to a Python library as a package, and we'll sometimes do this here as well.

How to do it...

Let's set up our Python environment(s)!

As we've mentioned, we'll be looking at two scenarios:

- Working with Google Colab
- Setting up a computer ourselves to host a Jupyter Notebook instance

In the first case, we won't need to set up anything on our server as we'll only be installing a few additional libraries. In the second case, we'll be installing an environment with the Anaconda distribution, and we'll be looking at setup options for Jupyter.

In both cases, we'll have an interactive Python notebook available through which we'll be running most of our experiments.

Installing libraries with Google Colab

Google Colab is a modified version of Jupyter Notebook that runs on Google hardware and provides access to runtimes with hardware acceleration such as TPUs and GPUs.

The downside of using Colab is that there is a maximum timeout of 12 hours; that is, jobs that run longer than 12 hours will stop. If you want to get around that, you can do either of the following:

- Run Colab with local kernels. This means you use the Colab interface but the models compute on your own computer (<https://research.google.com/Colaboratory/local-runtimes.html>).
- Install Jupyter Notebook yourself and don't use Google Colab.

For Google Colab, just go to <https://colab.research.google.com/>, and sign in with your Google credentials. In the following section, we'll deal with hosting notebooks on your own machine(s).

In Google Colab, you can save and re-load your models to and from the remote disk on Google servers. From there you can either download the models to your own computer or synchronize with Google Drive. The Colab GUI provides many useful code snippets for these use cases. Here's how to download files from Colab:



```
from joblib import dump
dump(
    my_model,
    'my_model_auc0.84.joblib'
)
files.download('my_model_auc0.84.joblib')
```

Self-hosting a Jupyter Notebook environment

There are different ways to maintain your Python libraries (see <https://packaging.python.org/tutorials/installing-packages/> for more details). For installations of Jupyter Notebook and all libraries, we recommend the Anaconda Python distribution, which works with the `conda` environment manager.

Anaconda is a Python distribution that comes with its own package installer and environment manager, called `conda`. This makes it easier to keep your libraries up to date and it handles system dependency management as well as Python dependency management. We'll mention a few alternatives to Anaconda/conda later; for now, we will quickly go through instructions for a local install. In the online material, you'll find instructions that will show how to serve similar installations to other people across a team, for example, in a company using a dockerized setup, which helps manage the setup of a machine or a set of machines across a network with a Python environment for AI.



If you have your computer already set up, and you are familiar with `conda` and `pip`, please feel free to skip this section.

For the Anaconda installation, we will need to download an installer and then choose a few settings:

1. Go to the Anaconda distribution page at <https://www.anaconda.com/products/individual> and download the appropriate installer for Python 3.7 for your system, such as **64-Bit (x86) Installer (506 MB)**.



Anaconda supports Linux, macOS, and Windows installers.

For macOS and Windows, you also have the choice of a graphical installer. This is all well explained in the Anaconda documentation; however, we'll just quickly go through the terminal installation.

2. Execute the downloaded shell script from your terminal:

```
bash Anaconda3-2019.10-Linux-x86_64.sh
```

You need to read and confirm the license agreement. You can do this by pressing the spacebar until you see the question asking you to agree. You need to press *Y* and then *Enter*.

You can go with the suggested download location or choose a directory that's shared between users on your computer. Once you've done that, you can get yourself a cup of tasty coffee or stay to watch the installation of Python and lots of Python libraries.

At the end, you can decide if you want to run the `conda init` routine. This will set up the PATH variables on your terminal, so when you type `python`, `pip`, `conda`, or `jupyter`, the `conda` versions will take precedence before any other installed version on your computer.

Note that on Unix/Linux based systems, including macOS, you can always check the location of the Python binary you are using as follows:

```
> which Python
```

On Windows, you can use the `where.exe` command.

If you see something like the following, then you know you are using the right Python runtime:

```
/home/ben/anaconda3/bin/Python
```

If you don't see the correct path, you might have to run the following:

```
source ~/.bashrc
```

This will set up your environment variables, including `PATH`. On Windows, you'd have to check your `PATH` variable.

It's also possible to set up and switch between different environments on the same machine. Anaconda comes with Jupyter/iPython by default, so you can start your Jupyter notebook from the terminal as follows:

```
> jupyter notebook
```

You should see the Jupyter Notebook server starting up. As a part of this information, a URL for login is printed to the screen.



If you run this from a server that you access over the network, make sure you use a screen multiplexer such as GNU screen or tmux to make sure your Jupyter Notebook client doesn't stop once your terminal gets disconnected.

We'll use many libraries in this book such as pandas, NumPy, scikit-learn, TensorFlow, Keras, PyTorch, Dash, Matplotlib, and others, so we'll be installing lots as we go through the recipes. This will often look like the following:

```
pip install <LIBRARY_NAME>
```

Or, sometimes, like this:

```
conda install <LIBRARY_NAME>
```

If we use conda's pip, or conda directly, this means the libraries will all be managed by Anaconda's Python installation.

3. You can install the aforementioned libraries like this:

```
pip install scikit-learn pandas numpy tensorflow-gpu torch
```

Please note that for the tensorflow-gpu library, you need to have a GPU available and ready to use. If not, change this to tensorflow (that is, without --gpu).

This should use the pip binary that comes with Anaconda and run it to install the preceding libraries. Please note that Keras is part of the TensorFlow library.

Alternatively, you can run the conda package installer as follows:

```
conda install scikit-learn pandas numpy tensorflow-gpu pytorch
```

Well done! You've successfully set up your computer for working with the many exciting recipes to come.

How it works...

Conda is an environment and package manager. Like many other libraries that we will use throughout this book, and like the Python language itself, conda is open source, so we can always find out exactly what an algorithm does and easily modify it. Conda is also cross-platform and not only supports Python but also R and other languages.

Package management can present many vexing challenges and, if you've been around for some time, you will probably remember spending many hours on issues such as conflicting dependencies or re-compiling packages and fixing paths – and you might be lucky if it's only that.

Conda goes beyond the earlier pip package manager (see <https://pip.pypa.io/en/stable/>) in that it checks dependencies of all packages installed within the environment and tries to come up with a way to resolve all the requirements. It also not only installs packages, but also allows us to set up environments that have separate installations of Python and binaries from different software repositories such as Bioconda (<https://bioconda.github.io/>), which specializes in bioinformatics, or the Anaconda repositories (<https://anaconda.org/anaconda/repo>).

There are hundreds of dedicated channels that you can use with conda. These are sub-repositories that can contain hundreds or thousands of different packages. Some of them are maintained by companies that develop specific libraries or software.

For example, you can install the pytorch package from the PyTorch channel as follows:

```
conda install -c pytorch pytorch
```



It's tempting to enable many channels in order to get the bleeding edge technology for everything. There's one catch, however, with this. If you enable many channels, or channels that are very big, conda's dependency resolution can become very slow. So be careful with using many additional channels, especially if they contain a lot of libraries.

There's more...

There are a number of Jupyter options you should probably be familiar with. These are in the file at `$HOME/.jupyter/jupyter_notebook_config.py`. If you don't have the file yet, you can create it using this command:

```
> jupyter notebook --generate-config
```

Here is an example configuration for `/home/ben/.jupyter/jupyter_notebook_config.py`:

```
import random, string
from notebook.auth import passwd

c = get_config()
c.NotebookApp.ip = '*'

password = ''.join(random.SystemRandom().choice(string.ascii_letters +
string.digits + string.punctuation) for _ in range(8))
print('The password is {}'.format(password))
c.NotebookApp.password = passwd(password)
c.NotebookApp.open_browser = False
c.NotebookApp.port = 8888
```

If you install your Python environment on a server that you want to access from your laptop (I have my local compute server in the attic), you'd first want make sure you can access the compute server remotely from another computer such as a laptop (`c.NotebookApp.ip = '*'.`).

Then we create a random password and configure it. We disable the option to have the browser open when we run Jupyter Notebook, and we then set the default port to 8888.

So Jupyter Notebook will be available when you open `localhost:8888` in a browser on the same computer. If

you are in a team as part of a larger organization, you'd be mostly working on remote number-crunching machines, and as a convenience, you – or your sysadmins – can set up a hosted Jupyter Notebook environment. This has several advantages:

- You can use the resources of a powerful server while simply accessing it through your browser.
- You can manage your packages in a contained environment on that server, while not affecting the server itself.
- You'll find yourself interacting with Jupyter Notebook's familiar REPL, which allows you to quickly test ideas and prototype projects.

If you are a single person, you don't need this; however, if you work in a team, you can put each person into a contained environment using either Docker or JupyterHub. Online, you'll find setup instructions for setting up a Jupyter environment with Docker.

See also

You can read up more on conda, Docker, JupyterHub, and other related tools on their respective documentation sites, as follows:

- The conda documentation: <https://docs.conda.io/en/latest/>
- The Docker documentation: <https://docs.docker.com/>
- JupyterHub: <https://jupyterhub.readthedocs.io/en/stable/>
- Jupyter: <https://jupyter.org/>
- JupyterLab: <https://jupyterlab.readthedocs.io/en/stable/>
- PyCharm: <https://www.jetbrains.com/pycharm/>
- Colab: <https://Colab.research.google.com>
- Pipenv: <https://pipenv-fork.readthedocs.io/en/latest/>
- Pip: <https://pip.pypa.io/en/stable/>

Getting proficient in Python for AI

In this set of quick-fire recipes, we'll look at ways to become more productive in Jupyter and to write more efficient Python code. If you are not familiar with Jupyter Notebook, please read a tutorial and then come back here. You can find a well-written tutorial at <https://realPython.com/jupyter-notebook-introduction/>. In the following recipe, we'll assume you have some familiarity with Jupyter Notebook.

Let's look at some simple but very handy tricks to make working in notebooks more comfortable and efficient. These are applicable whether you are relying on a local or hosted Python environment.

In this recipe, we'll look at a lot of different things that can help you become more productive when you are working in your notebook and writing Python code for AI solutions. Some of the built-in or externally available magic commands or extensions can also come in handy (see <https://ipython.readthedocs.io/en/stable/interactive/magics.html> for more details).

It's important to be aware of some of the Python efficiency hacks when it comes to machine learning, especially when working with some of the bigger datasets or more complex algorithms. Sometimes, your jobs can take very long to run, but often there are ways around it. For example, one, often relatively easy, way of finishing a job faster is to use parallelism.

The following short recipes cover the following:

- Obtaining the Jupyter command history
- Auto-reloading packages
- Debugging
- Timing code execution
- Compiling your code
- Speeding up pandas DataFrames
- Displaying progress bars
- Parallelizing your code

Getting ready

If you are using your own installation, whether directly on your system or inside a Docker environment, make sure that it's running. Then put the address of your Colab or Jupyter Notebook instance into your browser and press *Enter*.

We will be using the following libraries:

- `tqdm` for progress bars
- `swifter` for quicker pandas processing
- `ray` and `joblib` for multiprocessing
- `numba` for **just-in-time (JIT)** compilation
- `jax` (later on in this section) for array processing with autograd
- `cython` for compiling Cython extensions in the notebook

We can install them with `pip` as before:

```
pip install swifter tqdm ray joblib jaxlib seaborn numba cython
```

With that done, let's get to some efficiency hacks that make working in Jupyter faster and more convenient.

How to do it...

The sub-recipes here are short and sweet, and all provide ways to be more productive in Jupyter and Python.

If not indicated otherwise, all of the code needs to be run in a notebook, or, more precisely, in a notebook cell.

Let's get to these little recipes!

Obtaining the history of Jupyter commands and outputs

There are lots of different ways to obtain the code in Jupyter cells programmatically. Apart from these inputs, you can also look at the generated outputs. We'll get to both, and we can use global variables for this purpose.

Execution history

In order to get the execution history of your cells, the `_ih` list holds the code of executed cells. In order to get the complete execution history and write it to a file, you can do the following:

```
with open('command_history.py', 'w') as file:  
    for cell_input in _ih[:-1]:  
        file.write(cell_input + '\n')
```

If up to this point, we only ran a single cell consisting of `print('hello, world!')`, that's exactly what we should see in our newly created file, `command_history.py`:

```
!cat command_history.py  
print('hello, world!')
```

On Windows, to print the content of a file, you can use the `type` command.

Instead of `_ih`, we can use a shorthand for the content of the last three cells. `_i` gives you the code of the cell that just executed, `_ii` is used for the code of the cell executed before that, and `_iii` for the one before that.

Outputs

In order to get recent outputs, you can use `_` (single underscore), `__` (double underscore), and `___` (triple underscore), respectively, for the most recent, second, and third most recent outputs.

Auto-reloading packages

`autoreload` is a built-in extension that reloads the module when you make changes to a module on disk. It will automatically reload the module once you've saved it.

Instead of manually reloading your package or restarting the notebook, with `autoreload`, the only thing you have to do is to load and enable the extension, and it will do its magic.

We first load the extension as follows:

```
%load_ext autoreload
```

And then we enable it as follows:

```
%autoreload 2
```

This can save a lot of time when you are developing (and testing) a library or module.

Debugging

If you cannot spot an error and the traceback of the error is not enough to find the problem, debugging can speed up the error-searching process a lot. Let's have a quick look at the debug magic:

1. Put the following code into a cell:

```
def normalize(x, norm=10.0):  
    return x / norm  
  
normalize(5, 1)
```

You should see 5.0 as the cell output.

However, there's an error in the function, and I am sure the attentive reader will already have spotted it. Let's debug!

2. Put this into a new cell:

```
%debug  
normalize(5, 0)
```

3. Execute the cell by pressing *Ctrl + Enter* or *Alt + Enter*. You will get a debug prompt:

```
> <iPython-input-11-a940a356f993>(2)normalize()  
1 def normalize(x, norm=10): ---->  
2     return x / norm  
3  
4 normalize(5, 1)  
ipdb> a  
x = 5  
norm = 0  
ipdb> q  
-----  
----- ZeroDivisionError  
recent call last)  
<iPython-input-13-8ade44ebcb0c> in <module>()  
1 get_iPython().magic('debug') ---->  
Traceback (most
```

```
2 normalize(5, 0)

<iPython-input-11-a940a356f993> in normalize(a, norm)
    1 def normalize(x, norm=10): ---->
    2     return x / norm
    3
    4 normalize(5, 1)
ZeroDivisionError: division by zero
```

We've used the argument command to print out the arguments of the executed function, and then we quit the debugger with the quit command. You can find more commands on **The Python Debugger (pdb)** documentation page at <https://docs.python.org/3/library/pdb.html>.

Let's look at a few more useful magic commands.

Timing code execution

Once your code does what it's supposed to, you often get into squeezing every bit of performance out of your models or algorithms. For this, you'll check execution times and create benchmarks using them. Let's see how to time executions.

There is a built-in magic command for timing cell execution – %timeit. The %timeit functionality is part of the Python standard library (<https://docs.python.org/3/library/timeit.html>). It runs a command 10,000 times (by default) in a period of 5 times inside a loop (by default) and shows an average execution time as a result:

```
%%timeit -n 10 -r 1
import time
time.sleep(1)
```

We see the following output:

```
1 s ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
```

The iPython-autotime library (<https://github.com/cpccloud/iPython-autotime>) is an external extension that provides you the timings for all the cells that execute, rather than having to use %%timeit every time:

1. Install autotime as follows:

```
pip install iPython-autotime
```

Please note that this syntax works for Colab, but not in standard Jupyter Notebook. What always works to install libraries is using the pip or conda magic commands, %pip and %conda, respectively. Also, you can execute any shell command from the notebook if you start your line with an exclamation mark, like this:

```
!pip install iPython-autotime
```

2. Now let's use it, as follows:

```
%load_ext autotime
```

3. Test how long a simple list comprehension takes with the following command:

```
sum([i for i in range(10)])
```

We'll see this output: time: 5.62 ms.

Hopefully, you can see how this can come in handy for comparing different implementations. Especially in situations where you have a lot of data, or complex processing, this can be very useful.

Displaying progress bars

Even if your code is optimized, it's good to know if it's going to finish in minutes, hours, or days. tqdm provides progress bars with time estimates. If you aren't sure how long your job will run, it's just one letter away – in many cases, it's just a matter of changing range for trange:

```
from tqdm.notebook import trange
from tqdm.notebook import tqdm
tqdm.pandas()
```

The tqdm pandas integration (optional) means that you can see progress bars for pandas apply operations. Just swap apply for progress_apply.

For Python loops just wrap your loop with a tqdm function and voila, there'll be a progress bar and time estimates for your loop completion!

```
global_sum = 0.0
for i in trange(1000000):
    global_sum += 1.0
```

Tqdm provides different ways to do this, and they all require minimal code changes - sometimes as little as one letter, as you can see in the previous example. The more general syntax is wrapping your loop iterator with `tqdm` like this:

```
for _ in tqdm(range(10)):  
    print()
```

You should see a progress bar like in this screenshot:

```
for _ in tqdm(range(10)):  
    print()  
  
100%|██████████| 10/10 [00:00<00:00, 2871.04it/s]
```

So, next time you are just about to set off long-running loop, and you are not just sure how long it will take, just remember this sub-recipe, and use `tqdm`.

Compiling your code

Python is an interpreted language, which is a great advantage for experimenting, but it can be detrimental to speed. There are different ways to compile your Python code, or to use compiled code from Python.

Let's first look at Cython. Cython is an optimizing static compiler for Python, and the programming language compiled by the Cython compiler. The main idea is to write code in a language very similar to Python, and generate C code. This C code can then be compiled as a binary Python extension. SciPy (and NumPy), scikit-learn, and many other libraries have significant parts written in Cython for speed up. You can find out more about Cython on its website at <https://cython.org/>:

1. You can use the Cython extension for building cython functions in your notebook:

```
%load_ext Cython
```

2. After loading the extension, annotate your cell as follows:

```
%%cython  
def multiply(float x, float y):  
    return x * y
```

3. We can call this function just like any Python function – with the added benefit that it's already compiled:

```
multiply(10, 5) # 50
```

This is perhaps not the most useful example of compiling code. For such a small function, the overhead of compilation is too big. You would probably want to compile something that's a bit more complex.

Numba is a JIT compiler for Python (<https://numba.pydata.org/>). You can often get a speed-up similar to C or Cython using `numba` and writing idiomatic Python code like the following:

```
from numba import jit
@jit
def add_numbers(N):
    a = 0
    for i in range(N):
        a += i

add_numbers(10)
```

With `autotime` activated, you should see something like this:

```
time: 2.19 s
```

So again, the overhead of the compilation is too big to make a meaningful impact. Of course, we'd only see the benefit if it's offset against the compilation. However, if we use this function again, we should see a speedup. Try it out yourself! Once the code is already compiled, the time significantly improves:

```
add_numbers(10)
```

You should see something like this:

```
time: 867 µs
```

There are other libraries that provide JIT compilation including TensorFlow, PyTorch, and JAX, that can help you get similar benefits.

The following example comes directly from the JAX documentation, at <https://jax.readthedocs.io/en/latest/index.html>:

```
import jax.numpy as np
from jax import jit
def slow_f(x):
```

```
    return x * x + x * 2.0

x = np.ones((5000, 5000))
fast_f = jit(slow_f)
fast_f(x)
```

So there are different ways to get speed benefits from using JIT or ahead-of-time compilation. We'll see some other ways of speeding up your code in the following sections.

Speeding up pandas DataFrames

One of the most important libraries throughout this book will be `pandas`, a library for tabular data that's useful for **Extract, Transform, Load (ETL)** jobs. Pandas is a wonderful library, however; once you get to more demanding tasks, you'll hit some of its limitations. Pandas is the go-to library for loading and transforming data. One problem with data processing is that it can be slow, even if you vectorize the function or if you use `df.apply()`.

You can move further by parallelizing `apply`. Some libraries, such as `swifter`, can help you by choosing backends for computations for you, or you can make the choice yourself:

- You can use Dask DataFrames instead of `pandas` if you want to run on multiple cores of the same or several machines over a network.
- You can use CuPy or cuDF if you want to run computations on the GPU instead of the CPU. These have stable integrations with Dask, so you can run both on multiple cores and multiple GPUs, and you can still rely on a `pandas`-like syntax (see <https://docs.dask.org/en/latest/gpu.html>).

As we've mentioned, `swifter` can choose a backend for you with no change of syntax. Here is a quick setup for using `pandas` with `swifter`:

```
import pandas as pd
import swifter

df = pd.read_csv('some_big_dataset.csv')
df['datacol'] = df['datacol'].swifter.apply(some_long_running_function)
```

Generally, `apply()` is much faster than looping over DataFrames.

You can further improve the speed of execution by using the underlying NumPy arrays directly and accessing NumPy functions, for example, using `df.values.apply()`. NumPy vectorization can be a breeze, really. See the following example of applying a NumPy vectorization on a pandas DataFrame column:

```
square = lambda t: t ** 2
vfunc = np.vectorize(square)
df['squared'] = vfunc(df[col].values)
```

These are just two ways, but if you look at the next sub-recipe, you should be able to write a parallel map function as yet another alternative.

Parallelizing your code

One way to get something done more quickly is to do multiple things at once. There are different ways to implement your routines or algorithms with parallelism. Python has a lot of libraries that support this functionality. Let's see a few examples with multiprocessing, Ray, joblib, and how to make use of scikit-learn's parallelism.

The multiprocessing library comes as part of Python's standard library. Let's look at it first. We don't provide a dataset of millions of points here – the point is to show a usage pattern – however, please imagine a large dataset. Here's a code snippet of using our pseudo-dataset:

```
# run on multiple cores
import multiprocessing

dataset = [
    {
        'data': 'large arrays and pandas DataFrames',
        'filename': 'path/to/files/image_1.png'
    }, # ... 100,000 datapoints
]

def get_filename(datapoint):
    return datapoint['filename'].split('/')[-1]

pool = multiprocessing.Pool(64)
result = pool.map(get_filename, dataset)
```

Using Ray, you can parallelize over multiple machines in addition to multiple cores, leaving your code virtually unchanged. Ray efficiently handles data through shared memory (and zero-copy serialization) and uses a distributed task scheduler with fault tolerance:

```
# run on multiple machines and their cores
import ray
ray.init(ignore_reinit_error=True)

@ray.remote
def get_filename(datapoint):
    return datapoint['filename'].split('/')[-1]

result = []
for datapoint in dataset:
    result.append(get_filename.remote(datapoint))
```

Scikit-learn, the machine learning library we installed earlier, internally uses joblib for parallelization. The following is an example of this:

```
from joblib import Parallel, delayed

def complex_function(x):
    '''this is an example for a function that potentially could take very
    long.'''
    return sqrt(x)

Parallel(n_jobs=2)(delayed(complex_function)(i ** 2) for i in range(10))
```

This would give you [0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]. We took this example from the joblib examples about parallel for loops, available at <https://joblib.readthedocs.io/en/latest/parallel.html>.

When using scikit-learn, watch out for functions that have an n_jobs parameter. This parameter is directly handed over to joblib.Parallel ([https://github.com/joblib/joblib/parallel.py](https://github.com/joblib/joblib/blob/master/joblib/parallel.py)). None (the default setting) means sequential execution, in other words, no parallelism. So if you want to execute code in parallel, make sure to set this n_jobs parameter, for example, to -1 in order to make full use of all your CPUs.

PyTorch and Keras both support multi-GPU and multi-CPU execution. Multi-core parallelization is done by default. Multi-machine execution in Keras is getting easier from release to release with TensorFlow as the default backend.

See also

While notebooks are convenient, they are often messy, not conducive to good coding habits, and they cannot be versioned cleanly. Fastai has developed an extension for literate code development in notebooks called nbdev (<https://github.com/fastai/nbdev>), which provides tools for exporting and documenting code.

There are a lot more useful extensions that you can find in different places:

- The extension index: <https://github.com/iPython/iPython/wiki/Extensions-Index>
- Jupyter contrib extensions: <https://jupyter-contrib-nbextensions.readthedocs.io/en/latest/nbextensions.html>
- The awesome-jupyter list: <https://github.com/markusschanta/awesome-jupyter>

We would also like to highlight the following extensions:

- SQL Magic, which performs SQL queries: <https://github.com/catherinedevlin/iPython-sql>
- Watermark, which extracts version information for used packages: <https://github.com/rasbt/watermark>
- Pyheatmagic, for profiling with heat maps: <https://github.com/csurfer/pyheatmagic>
- Nose testing, for testing using nose: https://github.com/taavi/iPython_nose
- Pytest magic, for testing using pytest: <https://github.com/cjdrake/iPython-magic>
- Dot and others, used for drawing diagrams using graphviz: <https://github.com/cjdrake/iPython-magic>
- Scalene, a CPU and memory profiler: <https://github.com/emeryberger/scalene>

Some other libraries used or mentioned in this recipe include the following:

- Swifter: <https://github.com/jmcarpenter2/swifter>
- Autoreload: <https://ipython.org/ipython-doc/3/config/extensions/autoreload.html>
- pdb: <https://docs.python.org/3/library/pdb.html>
- tqdm: <https://github.com/tqdm/tqdm>
- JAX: <https://jax.readthedocs.io/>
- Seaborn: <https://seaborn.pydata.org/>

- Numba: <https://numba.pydata.org/numba-doc/latest/index.html>
- Dask: <https://ml.dask.org/>
- CuPy: <https://cupy.chainer.org>
- cuDF: <https://github.com/rapidsai/cudf>
- Ray: <http://ray.readthedocs.io/en/latest/rllib.html>
- joblib: <https://joblib.readthedocs.io/en/latest/>

Classifying in scikit-learn, Keras, and PyTorch

In this section, we'll be looking at data exploration, and modeling in three of the most important libraries. Therefore, we'll break things down into the following sub-recipes:

- Visualizing data in Seaborn
- Modeling in scikit-learn
- Modeling in Keras
- Modeling in PyTorch

Throughout these recipes and several subsequent ones, we'll focus on covering first the basics of the three most important libraries for AI in Python: scikit-learn, Keras, and PyTorch. Through this, we will introduce basic and intermediate techniques in supervised machine learning with deep neural networks and other algorithms. This recipe will cover the basics of these three main libraries in machine learning and deep learning.

We'll go through a simple classification task using scikit-learn, Keras, and PyTorch in turn. We'll run both of the deep learning frameworks in offline mode.

These recipes are for introducing the basics of the three libraries. However, even if you've already worked with all of them, you might still find something of interest.

Getting ready

The Iris Flower dataset is one of the oldest machine learning datasets still in use. It was published by Ronald Fisher in 1936 to illustrate linear discriminant analysis. The problem is to classify one of three iris flower species based on measurements of sepal and petal width and length.

Although this is a very simple problem, the basic workflow is as follows:

1. Load the dataset.
2. Visualize the data.
3. Preprocess and transform the data.
4. Choose a model to use.
5. Check the model performance.
6. Interpret and understand the model (this stage is often optional).

This is a standard process template that we will have to apply to most of the problems shown throughout this book. Typically, with industrial-scale problems, *Steps 1* and *2* can take much longer (sometimes estimated to take about 95 percent of the time) than for one of the already preprocessed datasets that you will get for a Kaggle competition or at the UCI machine learning repository. We will go into the complexities of each of these steps in later recipes and chapters.

We'll assume you've installed the three libraries earlier on and that you have your Jupyter Notebook or Colab instance running. Additionally, we will use the seaborn and scikit-plot libraries for visualization, so we'll install them as well:

```
!pip install seaborn scikit-plot
```

The convenience of using a dataset so well known is that we can easily load it from many packages, for example, like this:

```
import seaborn as sns
iris = sns.load_dataset('iris')
```

Let's jump right in, starting with data visualization.

How to do it...

Let's first have a look at the dataset.

Visualizing data in seaborn

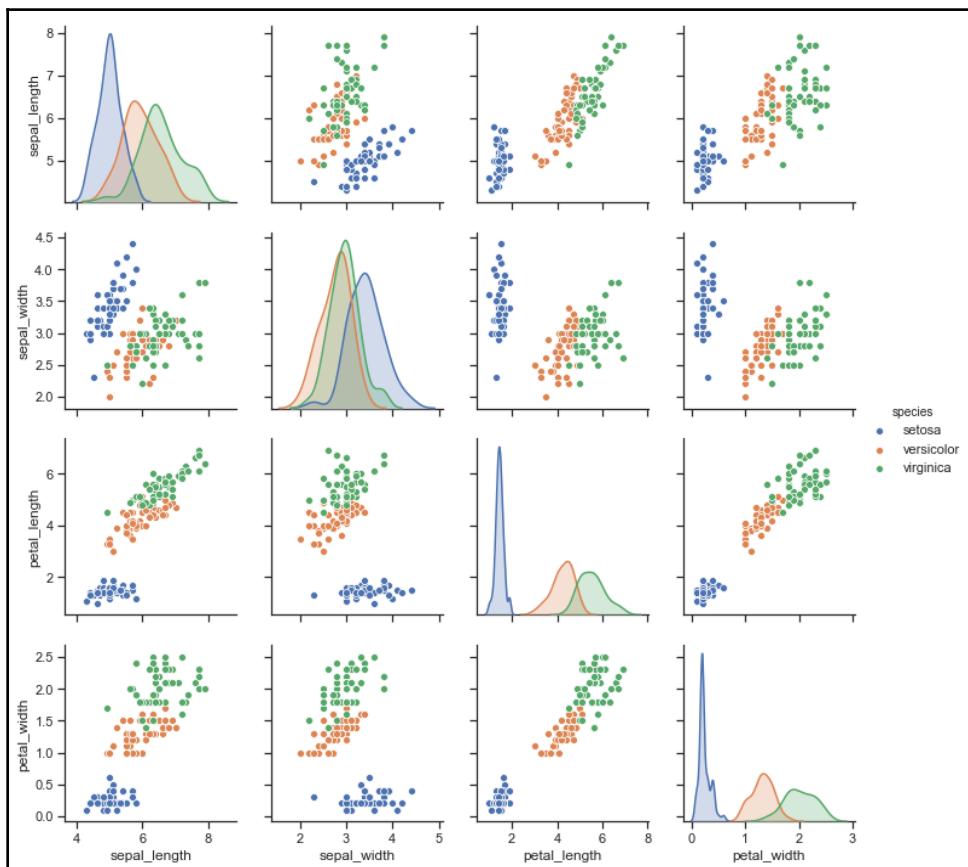
In this recipe, we'll go through the basic steps of data exploration. This is often important to understand the complexity of the problem and any underlying issues with the data:

1. Plot a pair-plot:

```
%matplotlib inline
# this^ is not necessary on Colab
import seaborn as sns
sns.set(style="ticks", color_codes=True)

g = sns.pairplot(iris, hue='species')
```

Here it comes (rendered in seaborn's pleasant spacing and coloring):



A pair-plot in seaborn visualizes pair-wise relationships in a dataset. Each subplot shows one variable plotted against another in a scatterplot. The subplots on the diagonal show the distribution of the variables. The colors correspond to the three classes.

From this plot, especially if you look along the diagonal, we can see that the virginica and versicolor species are not (linearly) separable. This is something we are going to struggle with, and that we'll have to overcome.

2. Let's have a quick look at the dataset:

```
iris.head()
```

We only see **setosa**, since the flower species are ordered and listed one after another:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

3. Separate the features and target in preparation for training as follows:

```
classes = {'setosa': 0, 'versicolor': 1, 'virginica': 2}
X = iris[['sepal_length', 'sepal_width', 'petal_length',
          'petal_width']].values
y = iris['species'].apply(lambda x: classes[x]).values
```

The last line converted the three strings corresponding to the three classes into numbers – this is called an ordinal coding. A multiclass machine learning algorithm can deal with this. For neural networks, we'll use another encoding, as you'll see later.

After these basic steps, we are ready to start developing predictive models. These are models that predict the flower class from the features. We'll see this in turn for each of the three most important machine learning libraries in Python. Let's start with scikit-learn.

Modeling in scikit-learn

In this recipe, we'll create a classifier in scikit-learn, and check its performance.

Scikit-learn (also known as sklearn) is a Python machine learning framework developed since 2007. It is also one of the most comprehensive frameworks available, and it is interoperable with the pandas, NumPy, SciPy, and Matplotlib libraries. Much of scikit-learn has been optimized for speed and efficiency in Cython, C, and C++.

Please note that not all scikit-learn classifiers can do multiclass problems. All classifiers can do binary classification, but not all can do more than two classes. The random forest model can, fortunately. The **random forest** model (sometimes referred to as **random decision forest**) is an algorithm that can be applied to classification and regression tasks, and is an ensemble of decision trees. The main idea is that we can increase precision by creating decision trees on bootstrapped samples of the dataset, and average over these trees.

Some of the following lines of code should appear to you as boilerplate, and we'll use them over and over:

1. Separate training and validation.

As a matter of good practice, we should always test the performance of our models on a sample of our data that wasn't used in training (referred to as a **hold-out set** or **validation set**). We can do this as follows:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.33, random_state=0
)
```

2. Define a model.

Here we define our model **hyperparameters**, and create the model instance with these hyperparameters. This goes as follows in our case:



Hyperparameters are parameters that are not part of the learning process, but control the learning. In the case of neural networks, this includes the learning rate, model architecture, and activation functions.

```
params = dict(
    max_depth=20,
    random_state=0,
    n_estimators=100,
```

```
)  
clf = RandomForestClassifier(**params)
```

3. Train the model.

Here, we pass the training dataset to our model. During training, the parameters of the model are being fit so that we obtain better results (where *better* is defined by a function, called the **cost function** or **loss function**).

For training we use the `fit` method, which is available for all sklearn-compatible models:

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

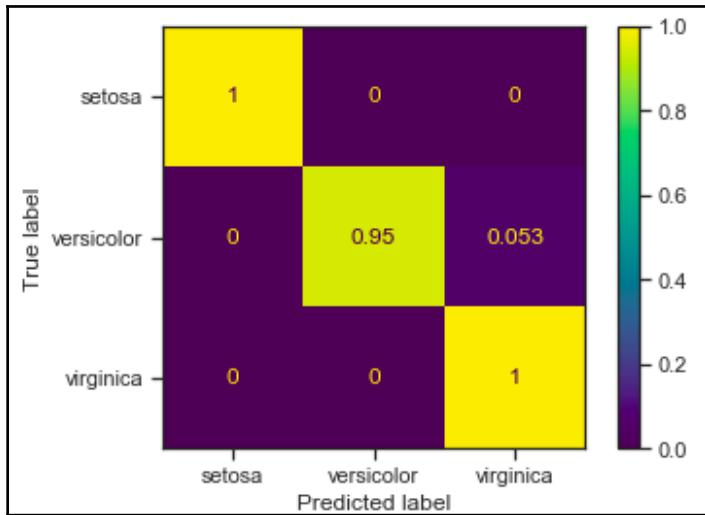
4. Check the performance of the model.

While there's a measure internal to the model (the cost function), we might want to look at additional measures. In the context of modeling, these are referred to as metrics. In scikit-learn, we have a lot of metrics at our fingertips. For classification, we would usually look at the confusion matrix, and often we'd want to plot it:

```
from sklearn.metrics import plot_confusion_matrix  
  
plot_confusion_matrix(  
    clf, X_test, y_test,  
    display_labels=['setosa', 'versicolor', 'virginica'],  
    normalize='true'  
)
```

The confusion matrix is relatively intuitive, especially when the presentation is as clear as with sklearn's `plot_confusion_matrix()`. Basically, we see how well our class predictions line up with the actual classes. We can see the predictions against actual labels, grouped by class, so that each entry corresponds to how many times class A was predicted given the actual class B. In this case, we've normalized the matrix, so that each row (actual labels) sums to one.

Here is the confusion matrix:



Since this is a normalized matrix, the numbers on the diagonal are also called the **hit rate** or **true positive rate**. We can see that setosa was predicted as setosa 100% (1) of the time. By contrast, versicolor was predicted as versicolor 95% of the time (0.95), while 5% of the time (0.053) it was predicted as virginica.

The performance is very good in terms of hit rate, however; as expected, we are having a small problem distinguishing between versicolor and virginica.

Let's move on to Keras.

Modeling in Keras

In this recipe, we'll be predicting the flower species in Keras.

Keras is a high-level interface for (deep) neural network models that can use TensorFlow as a backend, but also **Microsoft Cognitive Toolkit (CNTK)**, Theano, or PlaidML. Keras is an interface for developing AI models, rather than a standalone framework itself. Keras has been integrated as part of TensorFlow, so we import Keras from TensorFlow. Both TensorFlow and Keras are open source and developed by Google.

Since Keras is tightly integrated with TensorFlow, Keras models can be saved as TensorFlow models and then deployed in Google's deployment system, TensorFlow Serving (see <https://www.tensorflow.org/tfx/guide/serving>), or used from any of the programming languages such as, C++ or Java. Let's get into it:

1. Run the following code. If you are familiar with Keras, you'll recognize it as boilerplate:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import tensorflow as tf

def create_iris_model():
    """
    Create the iris classification model
    """
    iris_model = Sequential()
    iris_model.add(Dense(10, activation='selu', input_dim=4))
    iris_model.add(Dense(3, activation='softmax'))
    iris_model.compile(
        optimizer='rmsprop',
        loss='categorical_crossentropy',
        metrics=['accuracy'])
    iris_model.summary()
    return iris_model

iris_model = create_iris_model()
```

This yields the following model construction:

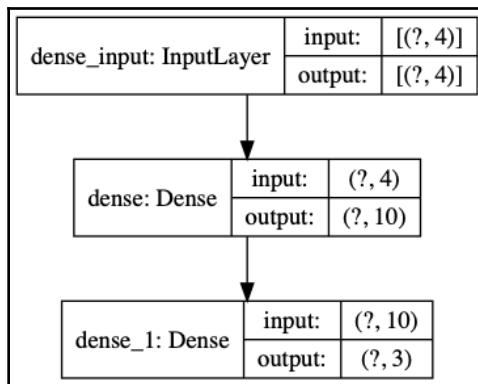
Layer (type)	Output Shape	Param #
dense (Dense)	(None, 10)	50
dense_1 (Dense)	(None, 3)	33
<hr/>		
Total params: 83		
Trainable params: 83		
Non-trainable params: 0		

We can visualize this model in different ways. We can use the built-in Keras functionality as follows:

```
dot = tf.keras.utils.model_to_dot(
    iris_model,
    show_shapes=True,
```

```
    show_layer_names=True,  
    rankdir="TB",  
    expand_nested=True,  
    dpi=96,  
    subgraph=False,  
)  
dot.write_png('iris_model_keras.png')
```

This writes a visualization of the network to a file called `iris_model_keras.png`. The image produced looks like this:



This shows that we have 4 input neurons, 10 hidden neurons, and 3 output neurons, fully connected in a feed-forward fashion. This means that all neurons in the input feed input to all neurons in the hidden layer, which in turn feed to all neurons in the output layer.

We are using the sequential model construction (as opposed to the graph). The sequential model type is more straightforward to build than the graph type. The layers are constructed the same way; however, for the sequential model, you have to define the input dimensionality, `input_dim`.

We use two dense layers, the intermediate layer with SELU activation function, and the final layer with the softmax activation function. We'll explain both of these in the *How it works...* section. As for the **SELU activation function**, suffice it to say for now that it provides a necessary nonlinearity so that the neural network can deal with more variables that are not linearly separable, as in our case. In practice, it is rare to use a linear (**identity function**) activation in the hidden layers.

Each unit (or neuron) in the final layer corresponds to one of the three classes. The **softmax function** normalizes the output layer so that its neural activations add up to 1. We train with categorical cross-entropy as our loss function. Cross-entropy is typically used for classification problems with neural networks. The binary cross-entropy loss is for two classes, and categorical cross-entropy is for two or more classes (cross-entropy will be explained in more detail in the *How it works...* section).

2. Next, one-hot encode the features.

This means we have three columns that each stand for one of the species, and one of them will be set to 1 for the corresponding class:

```
y_categorical = tf.keras.utils.to_categorical(y, 3)
```

Our `y_categorical` therefore has the shape (150, 3). This means that to indicate class 0 as the label, instead of having a 0 (this would be sometimes called **label encoding** or **integer encoding**), we have a vector of [1.0, 0.0, 0.0]. This is called **one-hot encoding**. The sum of each row is equal to 1.

3. Normalize the features.

For neural networks, our features should be normalized in a way that the activation functions can deal with the whole range of inputs – often this normalization is to the standard distribution, which has a mean of 0.0 and standard deviation of 1.0:

```
X = (X - X.mean(axis=0)) / X.std(axis=0)  
X.mean(axis=0)
```

The output of this cell is this:

```
array([-4.73695157e-16, -7.81597009e-16, -4.26325641e-16,  
-4.73695157e-16])
```

We see that the mean values for each column are very close to zero. We can also see the standard deviations with the following command:

```
X.std(axis=0)
```

The output is as follows:

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

The standard deviation is exactly 1, as expected.

4. Display our training progress in TensorBoard.

TensorBoard is a visualization tool for neural network learning, such as tracking and visualizing metrics, model graphs, feature histograms, projecting embeddings, and much more:

```
%load_ext tensorboard
import os

logs_base_dir = "./logs"
os.makedirs(logs_base_dir, exist_ok=True)
%tensorboard --logdir {logs_base_dir}
```

At this point, a TensorBoard widget should pop up in your notebook. We just have to make sure it gets the information it needs:

5. Plug the TensorBoard details into the Keras training function as a callback so TensorBoard gets the training information:

```
import datetime
logdir = os.path.join(
    logs_base_dir,
    datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
)
tensorboard_callback = tf.keras.callbacks.TensorBoard(
    logdir, histogram_freq=1
)
X_train, X_test, y_train, y_test = train_test_split(
    X, y_categorical, test_size=0.33, random_state=0
)
iris_model.fit(
    x=X_train,
    y=y_train,
    epochs=150,
    callbacks=[tensorboard_callback]
)
```

This runs our training. An epoch is an entire pass of the dataset through the neural network. We use 150 here, which is a bit arbitrary. We could have used a stopping criterion to stop training automatically when validation and training errors start to diverge, or in other words, when overfitting occurs.

In order to use `plot_confusion_matrix()` as before, for comparison, we'd have to wrap the model in a class that implements the `predict()` method, and has a list of `classes_` and an attribute of `_estimator_type` that is equal to the classifier. We will show that in the online material.

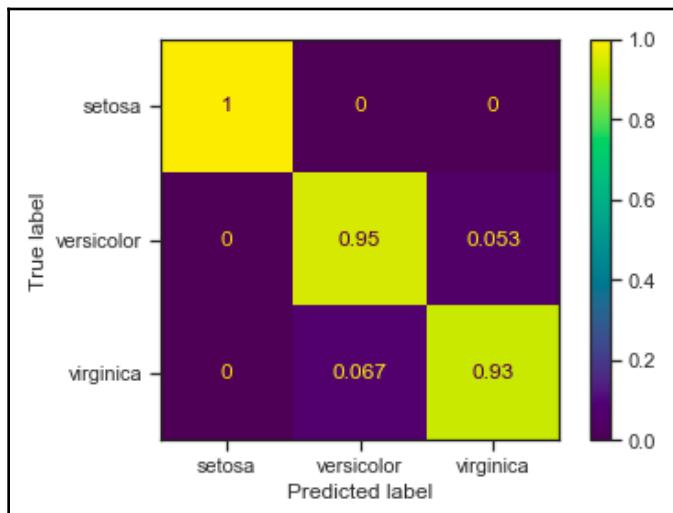
6. Plot the confusion matrix.

Here, it's easier to use a `scikitplot` function:

```
import scikitplot as skplt

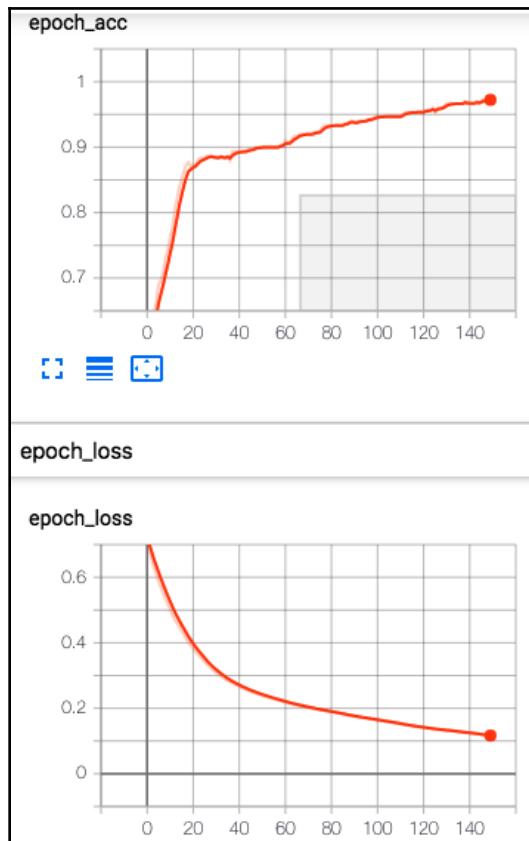
y_pred = iris_model.predict(X_test).argmax(axis=1)
skplt.metrics.plot_confusion_matrix(
    y_test.argmax(axis=1),
    y_pred,
    normalize=True
)
```

Again, as before, we normalize the matrix, so we get fractions. The output should look similar to the following:

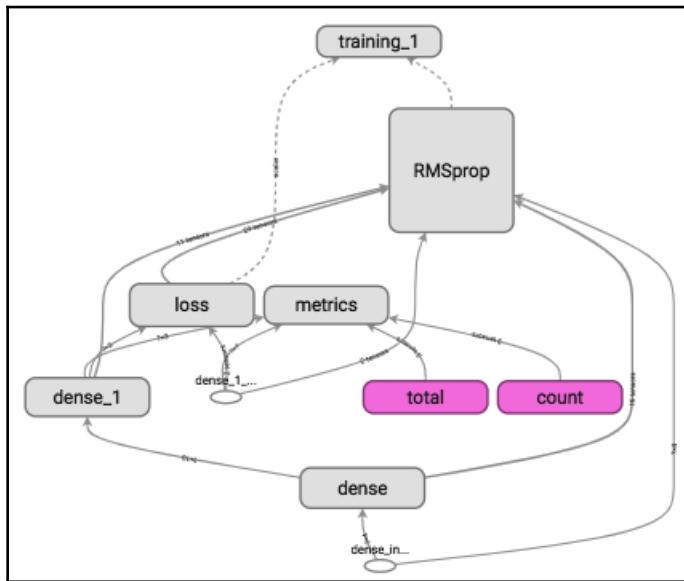


This is a bit worse than our previous attempt in scikit-learn, but with some tweaking we can get to a comparable level, or maybe even better performance. Examples of tweaking would be changing any of the model's hyperparameters such as the number of neurons in the hidden layer, any changes to the network architecture (adding a new layer), or changing the activation function of the hidden layer.

7. Check the charts from TensorBoard: the training progress and the model graph.
Here they are:



These plots show the accuracy and loss, respectively, over the entire training. We also get another visualization of the network in TensorBoard:



This shows all the network layers, the loss and metrics, the optimizer (RMSprop), and the training routine, and how they are related. As for the network architecture, we can see four dense layers (the presented input and targets are not considered proper parts of the network, and are therefore colored in white). The network consists of a dense hidden layer (being fed by the input), and an dense output layer (being fed by the hidden layer). The loss function is calculated between the output layer activation and the targets. The optimizer works with all layers based on the loss. You can find a tutorial on TensorBoard at https://www.tensorflow.org/tensorboard/get_started. The TensorBoard documentation explains more about configuration and options.

So the classification accuracy is improving and the loss is decreasing over the course of the training epochs. The final graph shows the network and training architecture, including the two dense layers, the loss and metrics, and the optimizer.

Modeling in PyTorch

In this recipe, we will describe a network equivalent to the previous one shown in Keras, train it, and plot the performance.

PyTorch is a deep learning framework that is based on the Torch library primarily developed by Facebook. For some time, Facebook was developing another deep learning framework, called Caffe2; however, it was merged into PyTorch in March 2018. Some of the strengths of PyTorch are in image and language processing applications. Apart from Python, Torch provides a C++ interface, both for learning and model deployment:

1. Let's define the model architecture first. This looks very similar to Keras:

```
import torch
from torch import nn

iris_model = nn.Sequential(
    torch.nn.Linear(4, 10), # equivalent to Dense in keras
    torch.nn.SELU(),
    torch.nn.Linear(10, 3),
    torch.nn.Softmax(dim=1)
)
print(iris_model)
```

This is the same architecture that we defined before in Keras: this is a feed-forward, two-layer neural network with a SELU activation on the hidden layer, and 10 and 3 neurons in the 2 layers.



If you prefer an output similar to the `summary()` function in Keras, you can use the `torchsummary` package (<https://github.com/sksq96/pytorch-summary>).

2. We need to convert our NumPy arrays to Torch tensors:

```
from torch.autograd import Variable

X_train = Variable(
    torch.Tensor(X_train).float()
)
Y_train = Variable(torch.Tensor(
    y_train.argmax(axis=1)).long()
)
X_test = Variable(
    torch.Tensor(X_test).float()
)
Y_test = Variable(
    torch.Tensor(y_test.argmax(axis=1)).long()
)
```

`y_train` is the one-hot encoded target matrix that we created earlier. We are converting it back to integer encoding since the PyTorch cross-entropy loss expects this.

3. Now we can train, as follows:

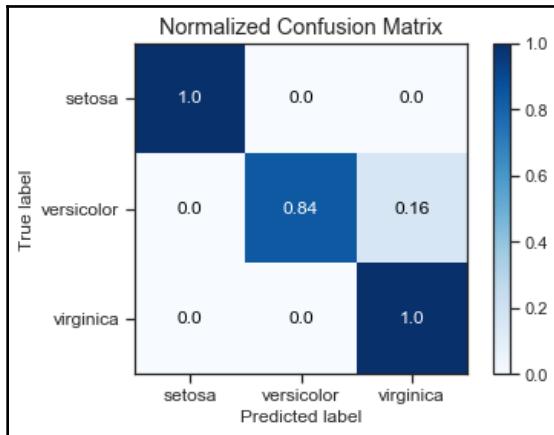
```
criterion = torch.nn.CrossEntropyLoss()    # cross entropy loss
optimizer = torch.optim.RMSprop(
    iris_model.parameters(), lr=0.01
)
for epoch in range(1000):
    optimizer.zero_grad()
    out = iris_model(X_train)
    loss = criterion(out, y_train)
    loss.backward()
    optimizer.step()
    if epoch % 10 == 0:
        print('number of epoch', epoch, 'loss', loss)
```

4. And then we'll use `scikitplot` to visualize our results, similar to before:

```
import scikitplot as skplt

y_pred = iris_model(X_test).detach().numpy()
skplt.metrics.plot_confusion_matrix(
    y_test,
    y_pred.argmax(axis=1),
    normalize=True
)
labels = ['setosa', 'versicolor', 'virginica']
ax.set_xticklabels(labels)
ax.set_yticklabels(labels)
```

This is the plot we get:



Your plot might differ. Neural network learning is not deterministic, so you could get better or worse numbers, or just different ones.

We can get better performance if we let this run longer. This is left as an exercise for you.

How it works...

We'll first look at the intuitions behind neural network training, then we'll look a bit more at some of the technical details that we will use in the PyTorch and Keras recipes.

Neural network training

The basic idea in machine learning is that we try to minimize an error by changing the parameters of a model. This adaption of the parameter is called learning. In supervised learning, the error is defined by a loss function calculated between the prediction of the model and the target. This error is calculated at every step and the model parameters are adjusted accordingly.

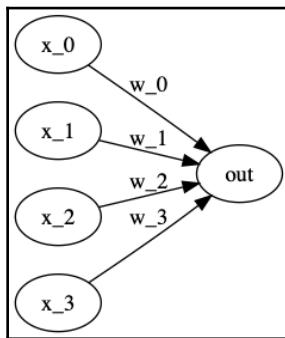
Neural networks are composable function approximators consisting of tunable affine transformations (f) with an activation function (σ):

$$F(x) = \sigma_m \circ f_m \circ \sigma_{m-1} \circ f_{m-1} \cdots \circ x$$

In the simplest terms, in a feed-forward neural network of one layer with linear activations, the model predictions are given by the sum of the product of the coefficients with the input in all of its dimensions:

$$F(x) = \sum_{i=0}^n x_i w_i = w \times x, \text{ with } n \text{ neurons and a single output.}$$

This is called a **perceptron**, and it is a linear binary classifier. A simple illustration with four inputs is shown in the following diagram:



The predictor for a one-dimensional input breaks down to the slope-intercept form of a line in two dimensions, $\hat{y} = mx + b$. Here, m is the slope and b the y-intercept. For higher-dimensional inputs, we can write (changing notation and vectorizing) $\hat{y} = W^T x + b$ with bias term b and weights W . This is still a line, just in a space of the same dimensionality as the input. Please note that \hat{y} denotes our model prediction for y , and for the examples where we know y , we can calculate the difference between the two as our prediction error.

We can also use the same very simple linear algebra to define the binary classifier by thresholding as follows:

$$\hat{y} = \begin{cases} 1 & \text{if } W^T x > 0, \\ 0 & \text{otherwise.} \end{cases}$$

This is still very simple linear algebra. This linear model with just one layer, called a perceptron, has difficulty predicting any more complex relationships. This lead to deep concern about the limitations of neural networks following an influential paper by Minsky and Papert in 1969. However, since the 1990s, neural networks have been experiencing a resurgence in the shape of **support vector machines (SVMs)** and the **multilayer perceptron (MLP)**. The MLP is a feed-forward neural network with at least one layer between the input and output (**hidden layer**). Since a multilayer perceptron with many layers of linear activations can be reduced to just one layer, non-trivially, we'll be referring to neural networks with hidden layers and nonlinear activation functions. These types of models can approximate arbitrary functions and perform nonlinear classification (according to the Universal Approximation Theorem). The activation function on any layer can be any differentiable nonlinearity; traditionally, the sigmoid, $\hat{y} = \tanh(W^T x)$, has been used a lot for this purpose.

For illustration, let's write this down with jax:

```
import jax.numpy as np
from jax import grad, jit
import numpy.random as npr

def predict(params, inputs):
    for W, b in params:
        outputs = np.dot(inputs, W) + b
        inputs = np.tanh(outputs)
    return outputs

def construct_network(layer_sizes=[10, 5, 1]):
    '''Please make sure your final layer corresponds to
    the target dimensionality.
    '''
    def init_layer(n_in, n_out):
        W = npr.randn(n_in, n_out)
        b = npr.randn(n_out, )
        return W, b
    return list(
        map(init_layer, layer_sizes[:-1], layer_sizes[1:])
    )

params = construct_network()
```

If you look at this code, you'll see that we could have equally written this up with operations in NumPy, TensorFlow, or PyTorch. You'll also note that the `construct_network()` function takes a `layer_sizes` argument. This is one of the hyperparameters of the network, something to decide on before learning. We can choose just an output of [1] to get the perceptron, or [10, 1] to get a two-layer perceptron. So this shows how to get a network as a set of parameters and how to get a prediction from this network. We still haven't discussed how we learn the parameters, and this brings us to errors.

There's an adage that says, "*all models are wrong, but some are useful.*" We can measure the error of our model, and this can help us to calculate the magnitude and direction of changes that we can make to our parameters in order to reduce the error.

Given a (differentiable) loss function (also called the cost function), \mathcal{E} , such as the **mean squared error (MSE)**, we can calculate our error. In the case of the MSE, the loss function is as follows:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2.$$

Then in order to get the change to our weights, we'll use the derivative of the loss over the points in training:

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathcal{E}(n)}{\partial v_j(n)} y_i(n)$$

This means we are applying a gradient descent, which means that over time, our error will be reduced proportionally to the gradient (scaled by learning rate η). Let's continue with our code:

```
def mse(preds, targets):
    return np.sum((preds - targets)**2)

def propagate_and_error(loss_fun):
    def error(params, inputs, targets):
        preds = predict(params, inputs)
        return loss_fun(preds, targets)
    return error

error_grads = jit(grad(propagate_and_error(mse)))
```

Both PyTorch and JAX have `autograd` functionality, which means that we can automatically get derivatives (gradients) of a wide range of functions.

We'll encounter a lot of different activation and loss functions throughout this book. In this chapter, we used the SELU activation function.

The SELU activation function

The **scaled exponential linear unit (SELU)** activation function was published quite recently by Klambauer et al in 2017 (<http://papers.nips.cc/paper/6698-self-normalizing-neural-networks.pdf>):

$$\text{SELU}(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha(\epsilon^x - 1) & \text{else} \end{cases}$$

The SELU function is linear for positive values of x , a scaled exponential for negative values, and 0 when x is 0. λ is a value greater than 1. You can find the details in the original paper. The SELU function has been shown to have better convergence properties than other functions. You can find a comparison of activation functions in Padamonti (2018) at <https://arxiv.org/pdf/1804.02763.pdf>.

Softmax activation

As our activation function for the output layer in the neural networks, we use a softmax function. This works as a normalization to sum 1.0 of the neural activations of the output layer. The output can be therefore interpreted as the class probabilities. The softmax activation function is defined as follows:

$$P(y = j \mid \mathbf{x}) = \frac{e^{a_j}}{\sum_{k=1}^K e^{a_k}}, \text{ with output activation vector } a \in \mathbb{R}^K.$$

Cross-entropy

In the multiclass training with neural networks, it's common to train for cross-entropy. The binary cross-entropy for multiclass cases looks like the following:

$$-\sum_{c=1}^M y_{o,c} \log(p_{o,c})$$

Here, M is the number of classes (setosa, versicolor, and virginica), y is 0 or 1 if the class label c is correct, and p is the predicted probability that the observation o is of class c . You can read up more on different loss functions and metrics on the ml-cheatsheet site, at https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html.

See also

You can find out more details on the website of each of the libraries used in this recipe:

- Seaborn: <https://seaborn.pydata.org/>
- Scikit-plot: <https://scikit-plot.readthedocs.io/>
- Scikit-learn: <https://github.com/scikit-learn/scikit-learn>
- Keras: <https://github.com/keras-team/keras>
- TensorFlow: <http://tensorflow.org/>
- TensorBoard: <https://www.tensorflow.org/tensorboard>
- PyTorch: <https://pytorch.org/>

TensorboardX is a TensorBoard interface for other deep learning frameworks apart from TensorFlow (PyTorch, Chainer, MXNet, and others), available at <https://github.com/lanpa/tensorboardX>.

It should probably be noted that scikit-plot is not maintained anymore. For the plotting of machine learning metrics and charts, mlxtend is a good option, at <http://rasbt.github.io/mlxtend/>.

Some other libraries we used here and that we will encounter throughout this book include the following:

- Matplotlib: <https://matplotlib.org/>
- NumPy: <https://docs.scipy.org/doc/numpy>
- SciPy: <https://docs.scipy.org/doc/scipy/reference>
- pandas: <https://pandas.pydata.org/pandas-docs/stable>

In the following recipe, we'll get to grips with a more realistic example in Keras.

Modeling with Keras

In this recipe, we will load a dataset and then we will conduct **exploratory data analysis (EDA)**, such as visualizing the distributions.

We will do typical preprocessing tasks such as encoding categorical variables, and normalizing and

rescaling for neural network training. We will then create a simple neural network model in Keras, train the model plotting using a generator, and plot the training and validation performance. We will look at a still quite simple dataset: the `dult` dataset from the UCI machine learning repository. With this dataset (also known as the Census Income dataset), the goal is to predict from census data whether someone earns more than US\$50,000 per year.

Since we have a few categorical variables, we'll also deal with the encoding of categorical variables.

Since this is still an introductory recipe, we'll go through this problem with a lot of detail for illustration. We'll have the following parts:

- Data loading and preprocessing:
 1. Loading the datasets
 2. Inspecting the data
 3. Categorical encoding
 4. Plotting variables and distributions
 5. Plotting correlations
 6. Label encoding
 7. Normalizing and scaling
 8. Saving the preprocessed data
- Model training:
 1. Creating the model
 2. Writing a data generator
 3. Training the model
 4. Plotting the performance
 5. Extracting performance metrics
 6. Calculating feature importances

Getting ready

We'll need a few libraries for this recipe in addition to the libraries we installed earlier:

- `category_encoders` for the encoding of categorical variables
- `minepy` for information-based correlation measures
- `eli5` for the inspection of black-box models

We've used Seaborn before for visualization.

We can install these libraries as follows:

```
!pip install category_encoders minepy eli5 seaborn
```



As a note to you, the reader: if you use `pip` and `conda` together, there is a danger that some of the libraries might become incompatible, creating a broken environment. We'd recommend using `conda` when a version of `conda` is available, although it is usually faster to use `pip`.

This dataset is already split into training and test. Let's download the dataset from UCI as follows:

```
!wget  
http://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data  
!wget  
http://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.test
```

`wget` doesn't ship with macOS by default; we suggest installing `wget` using `brew` (<https://formulae.brew.sh/formula/wget>). On Windows, you can visit the two preceding URLs and download both via the **File** menu. Make sure you remember the directory where you save the files, so you can find them later. There are a few alternatives, however:

- You can use the download script we provide in Chapter 2, *Advanced Topics in Supervised Machine Learning*, in the *Predicting house prices in PyTorch* recipe.
- You can install the `wget` library and run `import wget; wget.download(URL, filepath)`.

We have the following information from the UCI dataset description page:

- age: continuous.
- workclass: Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay, Never-worked.
- fnlwgt: continuous.
- education: Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc, 9th, 7th-th, 12th, Masters, 1st-4th, 10th, Doctorate, 5th-6th, Preschool.
- education-num: continuous.
- marital-status: Married-civ-spouse, Divorced, Never-married, Separated, Widowed, Married-spouse-absent, Married-AF-spouse.
- occupation: Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Prof-specialty, Handlers-cleaners, Machine-op-inspct, Adm-clerical, Farming-fishing, Transport-moving, Priv-house-serv, Protective-serv, Armed-Forces.
- relationship: Wife, Own-child, Husband, Not-in-family, Other-relative, Unmarried.
- race: White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black.
- sex: Female, Male.
- capital-gain: continuous.
- capital-loss: continuous.
- hours-per-week: continuous.
- native-country: United-States, and so on.



fnlwgt actually stands for the final weight; in other words, the total number of people constituting the entry.

Please keep in mind that this dataset is a well-known dataset that has been used many times in scientific publications and in machine learning tutorials. We are using it here to go over some basics in Keras without having to focus on the dataset.

How to do it...

As we've mentioned before, we'll first load the dataset, do some EDA, then create a model in Keras, train it, and look at the performance.

We've split this recipe up into data loading and preprocessing, and secondly, model training.

Data loading and preprocessing

We will start by loading the training and test sets:

- 1. Loading the dataset:** In order to load the dataset, we'll use pandas again. We use pandas' `read_csv()` command as before:

```
import pandas as pd
cols = [
    'age', 'workclass', 'fnlwgt',
    'education', 'education-num',
    'marital-status', 'occupation',
    'relationship', 'race', 'sex',
    'capital-gain', 'capital-loss',
    'hours-per-week', 'native-country', '50k'
]
train = pd.read_csv(
    'adult.data',
    names=cols
)
test = pd.read_csv(
    'adult.test',
    names=cols
)
```

Now let's look at the data!

- 2. Inspecting the data:** The beginning of the DataFrame we can see with the `head()` method:

```
train.head()
```

This yields the following output:

	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	50k
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	40	United-States	<=50K
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13	United-States	<=50K
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40	United-States	<=50K
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	40	United-States	<=50K
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	40	Cuba	<=50K

Next, we'll look at the test data:

```
test.head()
```

This looks as follows:

	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	50k
0	[1x3 Cross validator	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	25	Private	226802.0	11th	7.0	Never-married	Machine-op-inspect	Own-child	Black	Male	0.0	0.0	40.0	United-States	<=50K.
2	38	Private	89814.0	HS-grad	9.0	Married-civ-spouse	Farming-fishing	Husband	White	Male	0.0	0.0	50.0	United-States	<=50K.
3	28	Local-gov	336951.0	Assoc-acdm	12.0	Married-civ-spouse	Protective-serv	Husband	White	Male	0.0	0.0	40.0	United-States	>50K.
4	44	Private	160323.0	Some-college	10.0	Married-civ-spouse	Machine-op-inspect	Husband	Black	Male	7688.0	0.0	40.0	United-States	>50K.

The first row has 14 nulls and 1 unusable column out of 15 columns. We will discard this row:

```
test.drop(0, axis=0, inplace=True)
```

And it's gone.

3. **Categorical encoding:** Let's start with category encoding. For EDA, it's good to use ordinal encoding. This means that for a categorical feature, we map each value to a distinct number:

```
import category_encoders as ce

X = train.drop('50k', axis=1)
encoder = ce.OrdinalEncoder(cols=list(
    X.select_dtypes(include='object').columns) [:])
encoder.fit(X, train['50k'])
X_cleaned = encoder.transform(X)

X_cleaned.head()
```

We are separating X , the features, and y , the targets, here. The features don't contain the labels; that's the purpose of the `drop()` method – we could have equally used `del train['50k']`.

Here is the result:

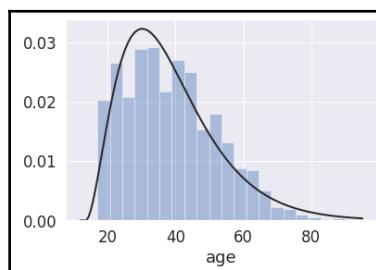
	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country
0	39	1	77516	1	13	1	1	1	1	1	2174	0	40	1
1	50	2	83311	1	13	2	2	2	1	1	0	0	13	1
2	38	3	215646	2	9	3	3	1	1	1	0	0	40	1
3	53	3	234721	3	7	2	3	2	2	1	0	0	40	1
4	28	3	338409	1	13	2	4	3	2	2	0	0	40	2

When starting with a new task, it's best to do EDA. Let's plot some of these variables.

- To plot variables and distributions, use the following code block:

```
from scipy import stats
import seaborn as sns
sns.set(color_codes=True)
sns.set_context(
    'notebook', font_scale=1.5,
    rc={"lines.linewidth": 2.0}
)
sns.distplot(train['age'], bins=20, kde=False, fit=stats.gamma)
```

We'll get the following plot:



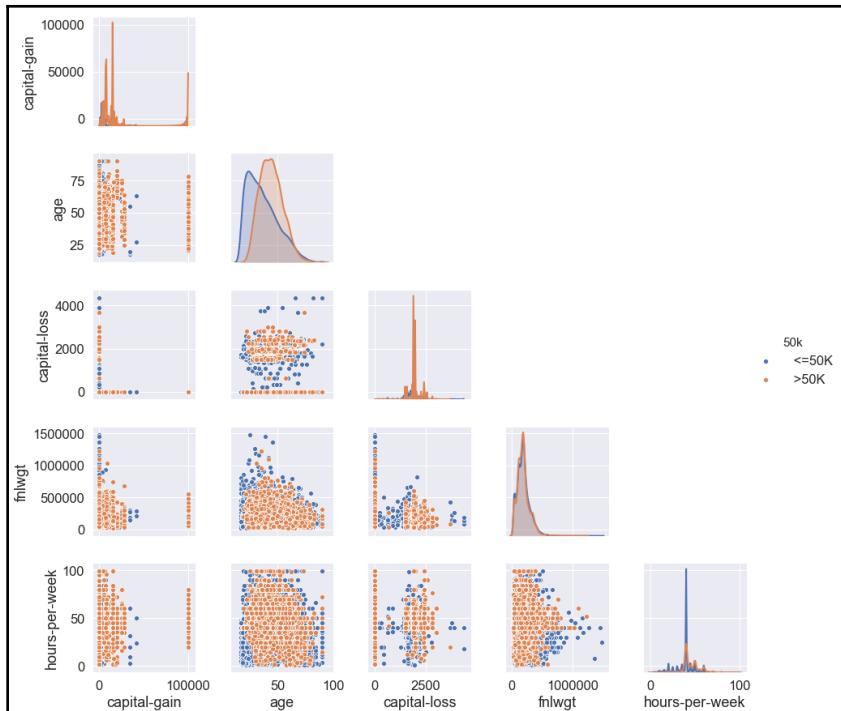
Next, we'll look at a pair-plot again. We'll plot all numerical variables against each other:

```
import numpy as np

num_cols = list(
```

```
set(
    train.select_dtypes(
        include='number'
    ).columns
) - set(['education-num'])
) + ['50k']
g = sns.pairplot(
    train[num_cols],
    hue='50k',
    height=2.5,
    aspect=1,
)
for i, j in zip(*np.triu_indices_from(g.axes, 1)):
    g.axes[i, j].set_visible(False)
```

As discussed in the previous recipe, the diagonal in the pair-plot shows us histograms of single variables – that is, the distribution of the variable – with the hue defined by the classes. Here we have orange versus blue (see the legend on the right of the following plot). The following subplots on the diagonal show scatter plots between the two variables:

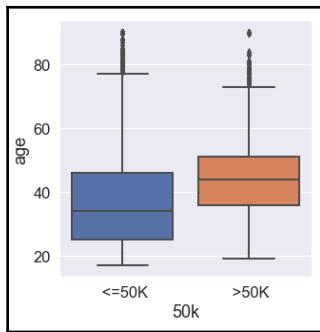


If we look at the age variable on the diagonal (second row), we see that the two classes have a different distribution, although they are still overlapping. Therefore, age seems to be discriminative with respect to our target class.

We can see that in a categorical plot as well:

```
sns.catplot(x='50k', y='age', kind='box', data=train)
```

Here's the resulting plot:



After this, let's move on to a correlation plot.

5. **Plotting correlations:** In order to get an idea of the redundancy between variables, we'll plot a correlation matrix based on the **Maximal Information Coefficient (MIC)**, a correlation metric based on information entropy. We'll explain the MIC at the end of this recipe.

Since the MIC can take a while to compute, we'll take the parallelization pattern we introduced earlier. Please note the creation of the thread pool and the `map` operation:

```
import numpy as np
import os
from sklearn.metrics.cluster import adjusted_mutual_info_score
from minepy import MINE
import multiprocessing

def calc_mic(args):
    (a, b, i1, i2) = args
    mine = MINE(alpha=0.6, c=15, est='mic_approx')
    mine.compute_score(a, b)
    return (mine.mic(), i1, i2)
```

```
pool = multiprocessing.Pool(os.cpu_count())

corrs = np.zeros((len(X_cleaned.columns), len(X_cleaned.columns)))
queue = []
for i1, col1 in enumerate(X_cleaned.columns):
    if i1 == 1:
        continue
    for i2, col2 in enumerate(X_cleaned.columns):
        if i1 < i2:
            continue
        queue.append((X_cleaned[col1], X_cleaned[col2], i1, i2))

results = pool.map(calc_mic, queue)

for (mic, i1, i2) in results:
    corrs[i1, i2] = mic

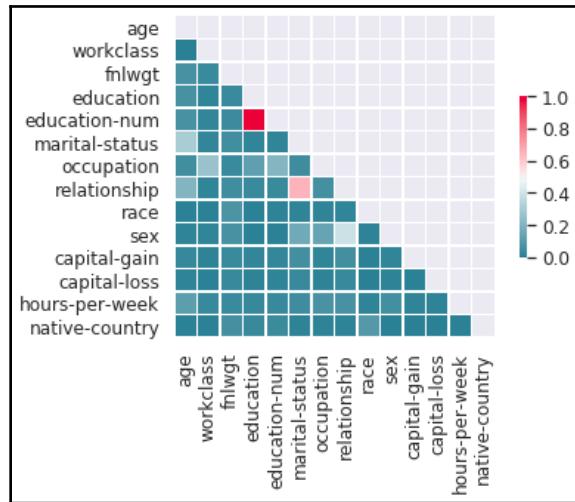
corrs = pd.DataFrame(
    corrs,
    columns=list(X_cleaned.columns),
    index=list(X_cleaned.columns)
)
```

This can still take a while, but should be much faster than doing the computations in sequence.

Let's visualize the correlation matrix as a heatmap: since the matrix is symmetric, here, we'll only show the lower triangle and apply some nice styling:

```
mask = np.zeros_like(corrs, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True
cmap = sns.diverging_palette(
    h_neg=220, h_pos=10, n=50, as_cmap=True
)
sns.set_context(
    'notebook', font_scale=1.1,
    rc={'lines.linewidth': 2.0}
)
sns.heatmap(
    corrs,
    square=True,
    mask=mask,
    cmap=cmap, vmax=1.0, center=0.5,
    linewidths=.5,
    cbar_kws={"shrink": .5}
)
```

This looks as follows:



We can see in the correlation matrix heatmap that most pair correlations are pretty low (most correlations are below 0.4), meaning that most features are relatively uncorrelated; however, there is one pair of variables that stands out, those of `education-num` and `education`:

```
corrs.loc['education-num', 'education']
```

The output is 0.9995095286140694.

This is about as close to a perfect correlation as it can get. These two variables do in fact refer to the same information.

Let's see the variance in `education-num` for each value in `education`:

```
train.groupby(by='education')['education-num'].std()
```

We only see zeros. There's no variance. In other words, each value in `education` corresponds to exactly one value in `education-num`. The variables are exactly the same! We should be able to remove one of them, for example with `del train['education']`, or just ignore one of them during training.

The UCI description page mentions missing variables. Let's look for missing variables now:

```
train.isnull().any()
```

We only see `False` for each variable, so we cannot see any missing values here.

In neural network training, for categorical variables, we have the choice of either using embeddings (we'll get to these in Chapter 10, *Natural Language Processing*) or feeding them as one-hot encodings; this means that each factor, each possible value, is encoded in a binary variable that indicates whether it is given or not. Let's try one-hot encodings for simplicity.

So, first, let's re-encode the variables:

```
encoder = ce.OneHotEncoder(  
    cols=list(X.select_dtypes(include='object').columns) [:]  
)  
encoder.fit(X, train['50k'])  
X_cleaned = encoder.transform(X)  
x_cleaned_cols = X_cleaned.columns  
x_cleaned_cols
```

Our `x_cleaned_cols` looks as follows:

```
Index(['age', 'workclass_1', 'workclass_2', 'workclass_3', 'workclass_4',  
       'workclass_5', 'workclass_6', 'workclass_7', 'workclass_8',  
       'workclass_9',  
       ...  
       'native-country_33', 'native-country_34', 'native-country_35',  
       'native-country_36', 'native-country_37', 'native-country_38',  
       'native-country_39', 'native-country_40', 'native-country_41',  
       'native-country_42'],  
      dtype='object', length=108)
```

After this, it's time to encode our labels.

6. **Label encoding:** We are going to encode target values in two columns as 1 if present and 0 if not present. It is good to remember that the Python truth values correspond to 0 and 1, respectively, for false and true. Since we have a binary classification task (that is, we only have two classes), we can use 0 and 1 in a single output. If we have more than two classes, we'd have to use categorical encoding for the output, which typically means we use as many output neurons as we do classes. Often, we have to try different solutions in order to see what works best.

In the following code block, we just made a choice and stuck with it:

```
y = np.zeros((len(X_cleaned), 2))  
y[:, 0] = train['50k'].apply(lambda x: x == ' <=50K')  
y[:, 1] = train['50k'].apply(lambda x: x == ' >50K')
```

7. **Normalizing and scaling:** We have to convert all values to z-values. This is when we subtract the mean and divide by the standard deviation, in order to get a normal distribution with a mean of 0.0 and a standard deviation of 1.0. It's not necessary to have normal distributions for neural network input. However, it's important that numerical values are scaled to the sensitive part of the neural network activation functions. Converting to z-scores is a standard way to do this:

```
from sklearn.preprocessing import StandardScaler

standard_scaler = StandardScaler()
X_cleaned = standard_scaler.fit_transform(X_cleaned)
X_test =
    standard_scaler.transform(encoder.transform(test[cols[:-1]]))
```

8. **Saving our preprocessing:** For good practice, we save our datasets and the transformers so we have an audit trail. This can be useful for bigger projects:

```
import joblib
joblib.dump(
    [encoder, standard_scaler, X_cleaned, X_test],
    'adult_encoder.joblib'
)
```

We are ready to train now.

Model training

We'll create the model, train it, plot performance, and then calculate the feature importance.

1. To create the model, we use the Sequential model type again. Here's our network architecture:

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(20, activation='selu', input_dim=108))
model.add(Dense(2, activation='softmax'))
model.compile(
    optimizer='rmsprop',
    loss='categorical_hinge',
    metrics=['accuracy']
)
model.summary()
```

Here's the Keras model summary:

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 20)	2180
dense_1 (Dense)	(None, 2)	42
<hr/>		
Total params: 2,222		
Trainable params: 2,222		
Non-trainable params: 0		

- Now, let's write a data generator. To make this a bit more interesting, we will use a generator this time to feed in our data in batches. This means that we stream in our data instead of putting all of our training data into the `fit()` function at once. This can be useful for very big datasets.

We'll use the `fit_generator()` function as follows:

```
def adult_feed(X_cleaned, y, batch_size=10, shuffle=True):
    def init_batches():
        return (
            np.zeros((batch_size, X_cleaned.shape[1])),
            np.zeros((batch_size, y.shape[1]))
        )
    batch_x, batch_y = init_batches()
    batch_counter = 0
    while True: # this is for every epoch
        indexes = np.arange(X_cleaned.shape[0])
        if shuffle == True:
            np.random.shuffle(indexes)
        for index in indexes:
            batch_x[batch_counter, :] = X_cleaned[index, :]
            batch_y[batch_counter, :] = y[index, :]
            batch_counter += 1
            if batch_counter >= batch_size:
                yield (batch_x, batch_y)
                batch_counter = 0
            batch_x, batch_y = init_batches()
```

If we had not done our preprocessing already, we could put it into this function.

- Now that we have our data generator, we can train our model as follows:

```
history = model.fit_generator(
    adult_feed(X_cleaned, y, 10),
    steps_per_epoch=len(X_cleaned) // 10,
```

```
    epochs=50  
)
```

This should be relatively quick since this is a small dataset; however, if you find that this takes too long, you can always reduce the dataset size or the number of epochs.

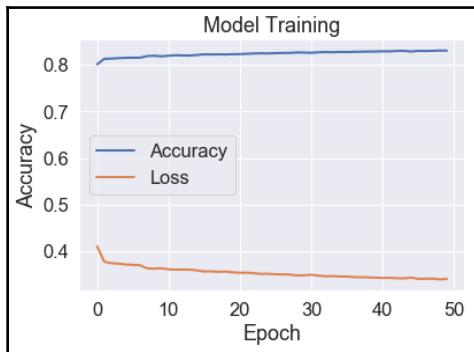
We have the output from the training, such as loss and metrics, in our `history` variable.

4. This time we will plot the training progress over epochs from the Keras training history instead of using TensorBoard. We didn't do validation, so we will only plot the training loss and training accuracy:

```
import matplotlib.pyplot as plt  
  
plt.plot(history.history['accuracy'])  
plt.plot(history.history['loss'])  
plt.title('Model Training')  
plt.ylabel('Accuracy')  
plt.xlabel('Epoch')  
plt.legend(['Accuracy', 'Loss'], loc='center left')
```

Please note that in some versions of Keras, accuracy is stored as `accuracy` rather than `acc` in the history.

Here's the resulting graph:



Over the training epochs, the accuracy is increasing while the loss is decreasing, so that's good.

5. Since we've already one-hot encoded and scaled our test data, we can directly predict and calculate our performance. We will calculate the **AUC (area-under-the-curve)** score using sklearn's built-in functions. The AUC score comes from the receiver operating characteristics, which is a visualization of the false positive rate (also called the false alarm rate) on the *x* axis, against the true positive rate (also called the hit rate) on the *y* axis. The integral under this curve, the AUC score, is a popular measure of classification performance and is useful for understanding the trade-off between a high hit rate and any false alarms:

```
from sklearn.metrics import roc_auc_score

predictions = model.predict(X_test)
# Please note that the targets have slightly different names in the
# test set than in the training dataset. We'll need to take care of
# this here:
target_lookup = {'<=50K.': 0, '>50K.': 1}
y_test = test['50k'].apply(
    lambda x: target_lookup[x]
).values
roc_auc_score(y_test, predictions.argmax(axis=1))
```

We get 0.7579310072282265 as the AUC score. An AUC score of 76% can be a good or bad score depending on the difficulty of the task. It's not bad for this dataset, but we could probably improve the performance by tweaking the model more. However, for now, we'll leave it as it is here.

6. Finally, we are going to check the feature importances. For this, we are going to use the eli5 library for black-box permutation importance. Black-box permutation importance encompasses a range of techniques that are model-agnostic, and, roughly speaking, permute features in order to establish their importance. You can read more about permutation importance in the *How it works...* section.

For this to work, we need a scoring function, as follows:

```
from eli5.permutation_importance import get_score_importances

def score(data, y=None, weight=None):
    return model.predict(data).argmax(axis=1)

base_score, score_decreases = get_score_importances(score, X_test,
                                                    y_test)
feature_importances = np.mean(score_decreases, axis=0).mean(axis=1)
```

Now we can print the feature importances in sorted order:

```
import operator

feature_importances_annotated = {col: imp for col, imp in
zip(x_cleaned_cols, feature_importances)}
sorted_feature_importances_annotated =
sorted(feature_importances_annotated.items(),
key=operator.itemgetter(1), reverse=True)

for i, (k, v) in enumerate(sorted_feature_importances_annotated):
    print('{i}: {k}: {v}'.format(i=i, k=k, v=v))
    if i > 9:
        break
```

We obtain something like the following list:

```
0: relationship_2: 0.02743074749708249
1: marital-status_2: 0.02671826054910632
2: age: 0.019335421657146367
3: education-num: 0.012689638228610035
4: education_1: 0.008377863767581843
5: hours-per-week: 0.007284564830170138
6: sex_1: 0.0072722805724464105
7: occupation_5: 0.006903752840734599
8: relationship_4: 0.006780910263497328
9: relationship_3: 0.006584362139917695
10: sex_2: 0.006436951047232972
```

Your final list might differ from the list here. The neural network training is not deterministic, although we could have tried to fix the random generator seed. Here, as we've expected, age is a significant factor; however, some categories in relationship status and marital status come up before age.

How it works...

We went through a typical process in machine learning: we loaded a dataset, plotted and explored it, and did preprocessing with the encoding of categorical variables and normalization. We then created and trained a neural network model in Keras, and plotted the training and validation performance. Let's talk about what we did in more detail.

Maximal information coefficient

There are many ways to calculate and plot correlation matrices, and we'll see some more possibilities in the recipes to come. Here we've calculated correlations based on the **maximal information coefficient (MIC)**. The MIC comes from the framework of *maximal information-based nonparametric exploration*. This was published in *Science Magazine* in 2011, where it was hailed as the correlation metric of the 21st century (the article can be found at <https://science.sciencemag.org/content/334/6062/1518.full>).

Applied to two variables, X and Y , it heuristically searches for bins in both variables, so that the mutual information between X and Y given the bins is maximal. The coefficient ranges between 0 (no correlation) and 1 (perfect correlation). It has an advantage with respect to the Pearson correlation coefficient, firstly in that it finds correlations that are non-linear, and secondly that it works with categorical variables.

Data generators

If you are familiar with Python generators, you won't need an explanation for what this is, but maybe a few clarifying words are in order. Using a generator gives the possibility of loading data **on-demand** or **on-line**, rather than at once. This means that you can work with datasets much larger than your available memory.

Some important terminology for generators in neural networks and Keras is as follows

- *Iterations* (`steps_per_epoch`) are the number of batches needed to complete one epoch.
- The *batch size* is the number of training examples in a single batch.

There are different ways to implement generators with Keras, such as the following:

- Using any Python generator
- Implementing `tensorflow.keras.utils.Sequence`

For the first option, we can use any generator really, but this uses a function with `yield`. This means we're providing the `steps_per_epoch` parameter for the Keras `fit_generator()` function.

As for the second, we write a class that inherits from `tensorflow.keras.utils.Sequence`, which implements the following methods:

- `len()`, in order for the `fit_generator()` function to know how much more data is to come. This corresponds to `steps_per_epoch` and is `len(data)/batch_size`.
- `__getitem__()`, for the `fit_generator` to ask for the next batch.
- `on_epoch_end()` to do some shuffling or other things at the end of an epoch – this is optional.

For simplicity, we've taken the former approach.

We'll see later that batch data loading using generators is often a part of online learning, that is, the type of learning where we incrementally train a model on more and more data as it comes in.

Permutation importance

The `e115` library can calculate permutation importance, which measures the increase in the prediction error when features are not present. It's also called the **mean decrease accuracy (MDA)**. Instead of re-training the model in a leave-one-feature-out fashion, the feature can be replaced by random noise. This noise is drawn from the same distribution as the feature so as to avoid distortions. Practically, the easiest way to do this is to randomly shuffle the feature values between rows. You can find more details about permutation importance in Breiman's *Random Forests* (2001), at <https://www.stat.berkeley.edu/~breiman/randomforest2001.pdf>.

See also

We'll cover a lot more about Keras, the underlying TensorFlow library, online learning, and generators in the recipes to come. I'd recommend you get familiar with layer types, data loaders and preprocessors, losses, metrics, and training options. All this is transferable to other frameworks such as PyTorch, where the **application programming interface (API)** differs; however, the essential principles are the same.

Here are links to the documentation for TensorFlow/Keras:

- Layer types: https://www.tensorflow.org/api_docs/Python/tf/keras/layers
- Data loading: <https://www.tensorflow.org/guide/data>
- Losses: https://www.tensorflow.org/api_docs/Python/tf/keras/losses
- Metrics: https://www.tensorflow.org/api_docs/Python/tf/keras/metrics
- Training: https://www.tensorflow.org/guide/keras/train_and_evaluate

Both the Keras/TensorFlow combination and PyTorch provide a lot of interesting functionality that's beyond the scope of this recipe – or even this book. To name just a few, PyTorch has automatic differentiation functionality (in the form of autograd, with more info at <https://pytorch.org/docs/stable/autograd.html>), and TensorFlow has an estimator API, which is an abstraction similar to Keras (for more detail on this, see <https://www.tensorflow.org/guide/estimator>).

For information on eli5, please visit its website at <https://eli5.readthedocs.io/>.

For more datasets, the following three websites are your friends:

- UCI machine learning datasets: <http://archive.ics.uci.edu/ml/datasets>
- Kaggle datasets: <https://www.kaggle.com/datasets/>
- Google Dataset Search: <https://datasetsearch.research.google.com/>

2

Advanced Topics in Supervised Machine Learning

Following the tasters with scikit-learn, Keras, and PyTorch in the previous chapter, in this chapter, we will move on to more end-to-end examples. These examples are more advanced in the sense that they include more complex transformations and model types.

We'll be predicting partner choices with sklearn, where we'll implement a lot of custom transformer steps and more complicated machine learning pipelines. We'll then predict house prices in PyTorch and visualize feature and neuron importance. After that, we will perform active learning to decide customer values together with online learning in sklearn. In the well-known case of repeat offender prediction, we'll build a model without racial bias. Last, but not least, we'll forecast time series of CO₂ levels.



Online learning in this context (as opposed to internet-based learning) refers to a model update strategy that incorporates training data that comes in sequentially. This can be useful in cases where the dataset is very big (often the case with images, videos, and texts) or where it's important to keep the model up to date given the changing nature of the data.

In many of these recipes, we've shortened the description to the most salient details in order to highlight particular concepts. For the full details, please refer to the notebooks on GitHub.

In this chapter, we'll be covering the following recipes:

- Transforming data in scikit-learn
- Predicting house prices in PyTorch
- Live decisioning customer values
- Battling algorithmic bias
- Forecasting CO₂ time series

Technical requirements

The code and notebooks for this chapter are available on GitHub at <https://github.com/PackPublishing/Artificial-Intelligence-with-Python-Cookbook/tree/master/chapter02>.

Transforming data in scikit-learn

In this recipe, we will be building more complex pipelines using mixed-type columnar data. We'll use a speed dating dataset that was published in 2006 by Fisman *et al.*: <https://doi.org/10.1162/qjec.2006.121.2.673>

Perhaps this recipe will be informative in more ways than one, and we'll learn something useful about the mechanics of human mating choices.

The dataset description on the OpenML website reads as follows:

*This data was gathered from participants in experimental speed dating events from 2002-2004. During the events, the attendees would have a four-minute **first date** with every other participant of the opposite sex. At the end of their 4 minutes, participants were asked whether they would like to see their date again. They were also asked to rate their date on six attributes: attractiveness, sincerity, intelligence, fun, ambition, and shared interests. The dataset also includes questionnaire data gathered from participants at different points in the process. These fields include demographics, dating habits, self-perception across key attributes, beliefs in terms of what others find valuable in a mate, and lifestyle information.*

The problem is to predict mate choices from what we know about participants and their matches. This dataset presents some challenges that can serve an illustrative purpose:

- It contains 123 different features, of different types:
 - Categorical
 - Numerical
 - Range features

It also contains the following:

- Some missing values
- Target imbalance

On the way to solving this problem of predicting mate choices, we will build custom encoders in scikit-learn and a pipeline comprising all features and their preprocessing steps.

The primary focus in this recipe will be on pipelines and transformers. In particular, we will build a custom transformer for working with range features and another one for numerical features.

Getting ready

We'll need the following libraries for this recipe. They are as follows:

- OpenML to download the dataset
- openml_speed_dating_pipeline_steps to use our custom transformer
- imbalanced-learn to work with imbalanced classes
- shap to show us the importance of features

In order to install them, we can use pip again:

```
pip install -q openml openml_speed_dating_pipeline_steps==0.5.5
imbalanced_learn category_encoders shap
```



OpenML is an organization that intends to make data science and machine learning reproducible and therefore more conducive to research. The OpenML website not only hosts datasets, but also allows the uploading of machine learning results to public leaderboards under the condition that the implementation relies solely on open source. These results and how they were obtained can be inspected in complete detail by anyone who's interested.

In order to retrieve the data, we will use the OpenML Python API. The `get_dataset()` method will download the dataset; with `get_data()`, we can get pandas DataFrames for features and target, and we'll conveniently get the information on categorical and numerical feature types:

```
import openml
dataset = openml.datasets.get_dataset(40536)
X, y, categorical_indicator, _ = dataset.get_data(
    dataset_format='DataFrame',
    target=dataset.default_target_attribute
)
categorical_features = list(X.columns[categorical_indicator])
numeric_features = list(
```

```
X.columns[[not(i) for i in categorical_indicator]]  
)
```



In the original version of the dataset, as presented in the paper, there was a lot more work to do. However, the version of the dataset on OpenML already has missing values represented as `numpy.nan`, which lets us skip this conversion. You can see this preprocessor on GitHub if you are interested: <https://github.com/benman1/OpenML-Speed-Dating>

Alternatively, you can use a download link from the OpenML dataset web page at https://www.openml.org/data/get_csv/13153954/speeddating.arff.

With the dataset loaded, and the libraries installed, we are ready to start cracking.

How to do it...

Pipelines are a way of describing how machine learning algorithms, including preprocessing steps, can follow one another in a sequence of transformations on top of the raw dataset before applying a final predictor. We will see examples of these concepts in this recipe and throughout this book.

A few things stand out pretty quickly looking at this dataset. We have a lot of categorical features. So, for modeling, we will need to encode them numerically, as in the *Modeling and predicting in Keras* recipe in Chapter 1, *Getting Started with Artificial Intelligence in Python*.

Encoding ranges numerically

Some of these are actually encoded ranges. This means these are ordinal, in other words, they are categories that are ordered; for example, the `d_interests_correlate` feature contains strings like these:

```
[[0-0.33], [0.33-1], [-1-0]]
```

If we were to treat these ranges as categorical variables, we'd lose the information about the order, and we would lose information about how different two values are. However, if we convert them to numbers, we will keep this information and we would be able to apply other numerical transformations on top.

We are going to implement a transformer to plug into an sklearn pipeline in order to convert these range features to numerical features. The basic idea of the conversion is to extract the upper and lower bounds of these ranges as follows:

```
def encode_ranges(range_str):
    splits = range_str[1:-1].split('-')
    range_max = splits[-1]
    range_min = '-'.join(splits[:-1])
    return range_min, range_max

examples = X['d_interests_correlate'].unique()
[encode_ranges(r) for r in examples]
```

We'll see this for our example:

```
[('0', '0.33'), ('0.33', '1'), ('-1', '0')]
```

In order to get numerical features, we can then take the mean between the two bounds. As we've mentioned before, on OpenML, not only are results shown, but also the models are transparent. Therefore, if we want to submit our model, we can only use published modules. We created a module and published it in the pypi Python package repository, where you can find the package with the complete code: <https://pypi.org/project/openml-speed-dating-pipeline-steps/>.

Here is the simplified code for RangeTransformer:

```
from sklearn.base import BaseEstimator, TransformerMixin
import category_encoders.utils as util

class RangeTransformer(BaseEstimator, TransformerMixin):
    def __init__(self, range_features=None, suffix='_range/mean', n_jobs=-1):
        assert isinstance(range_features, list) or range_features is None
        self.range_features = range_features
        self.suffix = suffix
        self.n_jobs = n_jobs

    def fit(self, X, y=None):
        return self

    def transform(self, X, y=None):
        X = util.convert_input(X)
        if self.range_features is None:
            self.range_features = list(X.columns)

        range_data = pd.DataFrame(index=X.index)
        for col in self.range_features:
            range_data[str(col) + self.suffix] = pd.to_numeric(
```

```
        self._vectorize(X[col])
    )
self.feature_names = list(range_data.columns)
return range_data

def _vectorize(self, s):
    return Parallel(n_jobs=self.n_jobs)(
        delayed(self._encode_range)(x) for x in s
    )

@staticmethod
@lru_cache(maxsize=32)
def _encode_range(range_str):
    splits = range_str[1:-1].split('-')
    range_max = float(splits[-1])
    range_min = float('-'.join(splits[:-1]))
    return sum([range_min, range_max]) / 2.0

def get_feature_names(self):
    return self.feature_names
```

This is a shortened snippet of the custom transformer for ranges. Please see the full implementation on GitHub at <https://github.com/benman1/OpenML-Speed-Dating>.

Please pay attention to how the `fit()` and `transform()` methods are used. We don't need to do anything in the `fit()` method, because we always apply the same static rule. The `transfer()` method applies this rule. We've seen the examples previously. What we do in the `transfer()` method is to iterate over columns. This transformer also shows the use of the parallelization pattern typical to scikit-learn. Additionally, since these ranges repeat a lot, and there aren't so many, we'll use a cache so that, instead of doing costly string transformations, the range value can be retrieved from memory once the range has been processed once.

An important thing about custom transformers in scikit-learn is that they should inherit from `BaseEstimator` and `TransformerMixin`, and implement the `fit()` and `transform()` methods. Later on, we will require `get_feature_names()` so we can find out the names of the features generated.

Deriving higher-order features

Let's implement another transformer. As you may have noticed, we have different types of features that seem to refer to the same personal attributes:

- Personal preferences
- Self-assessment
- Assessment of the other person

It seems clear that differences between any of these features could be significant, such as the importance of sincerity versus how sincere someone assesses a potential partner. Therefore, our next transformer is going to calculate the differences between numerical features. This is supposed to help highlight these differences.

These features are derived from other features, and combine information from two (or potentially more features). Let's see what the `NumericDifferenceTransformer` feature looks like:

```
import operator

class NumericDifferenceTransformer(BaseEstimator, TransformerMixin):
    def __init__(self, features=None,
                 suffix='_numdist', op=operator.sub, n_jobs=-1):
        assert isinstance(
            features, list
        ) or features is None
        self.features = features
        self.suffix = suffix
        self.op = op
        self.n_jobs = n_jobs

    def fit(self, X, y=None):
        X = util.convert_input(X)
        if self.features is None:
            self.features = list(
                X.select_dtypes(include='number').columns
            )
        return self

    def _col_name(self, col1, col2):
        return str(col1) + '_' + str(col2) + self.suffix

    def _feature_pairs(self):
        feature_pairs = []
        for i, col1 in enumerate(self.features[:-1]):
```

```

        for col2 in self.features[i+1:]:
            feature_pairs.append((col1, col2))
        return feature_pairs

    def transform(self, X, y=None):
        X = util.convert_input(X)

        feature_pairs = self._feature_pairs()
        columns = Parallel(n_jobs=self.n_jobs) (
            delayed(self._col_name)(col1, col2)
            for col1, col2 in feature_pairs
        )
        data_cols = Parallel(n_jobs=self.n_jobs) (
            delayed(self.op)(X[col1], X[col2])
            for col1, col2 in feature_pairs
        )
        data = pd.concat(data_cols, axis=1)
        data.rename(
            columns={i: col for i, col in enumerate(columns)},
            inplace=True, copy=False
        )
        data.index = X.index
        return data

    def get_feature_names(self):
        return self.feature_names

```

This is a custom transformer that calculates differences between numerical features. Please refer to the full implementation in the repository of the OpenML-Speed-Dating library at <https://github.com/benman1/OpenML-Speed-Dating>.

This transformer has a very similar structure to `RangeTransformer`. Please note the parallelization between columns. One of the arguments to the `__init__()` method is the function that is used to calculate the difference. This is `operator.sub()` by default. The `operator` library is part of the Python standard library and implements basic operators as functions. The `sub()` function does what it sounds like:

```

import operator
operator.sub(1, 2) == 1 - 2
# True

```

This gives us a prefix or functional syntax for standard operators. Since we can pass functions as arguments, this gives us the flexibility to specify different operators between columns.

The `fit()` method this time just collects the names of numerical columns, and we'll use these names in the `transform()` method.

Combining transformations

We will put these transformers together with `ColumnTransformer` and the pipeline. However, we'll need to make the association between columns and their transformations. We'll define different groups of columns:

```
range_cols = [
    col for col in X.select_dtypes(include='category')
    if X[col].apply(lambda x: x.startswith('[')
    if isinstance(x, str) else False).any()
]
cat_columns = list(
    set(X.select_dtypes(include='category').columns) - set(range_cols)
)
num_columns = list(
    X.select_dtypes(include='number').columns
)
```

Now we have columns that are ranges, columns that are categorical, and columns that are numerical, and we can assign pipeline steps to them.

In our case, we put this together as follows, first in a preprocessor:

```
from imblearn.ensemble import BalancedRandomForestClassifier
from sklearn.feature_selection import SelectKBest, f_classif
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import FunctionTransformer
import category_encoders as ce
import openml_speed_dating_pipeline_steps as pipeline_steps

preprocessor = ColumnTransformer(
    transformers=[
        ('ranges', Pipeline(steps=[
            ('impute',
            pipeline_steps.SimpleImputerWithFeatureNames(strategy='constant',
            fill_value=-1)),
            ('encode', pipeline_steps.RangeTransformer())
        ]), range_cols),
        ('cat', Pipeline(steps=[
            ('impute',
            pipeline_steps.SimpleImputerWithFeatureNames(strategy='constant',
            fill_value='-1')),
```

```
('encode', ce.OneHotEncoder(
    cols=None, # all features that it given by ColumnTransformer
    handle_unknown='ignore',
    use_cat_names=True
)
),
],
[]),
cat_columns,
('num', pipeline_steps.SimpleImputerWithFeatureNames(strategy='median'),
num_columns),
],
remainder='drop', n_jobs=-1
)
```

And then we'll put the preprocessing in a pipeline, together with the estimator:

```
def create_model(n_estimators=100):
    return Pipeline(
        steps=[('preprocessor', preprocessor),
               ('numeric_differences',
                pipeline_steps.NumericDifferenceTransformer()),
               ('feature_selection', SelectKBest(f_classif, k=20)),
               ('rf', BalancedRandomForestClassifier(
                   n_estimators=n_estimators,
                   )
               )
        ]
    )
```

Here is the performance in the test set:

```
from sklearn.metrics import roc_auc_score, confusion_matrix
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.33,
    random_state=42,
    stratify=y
)
clf = create_model(50)
clf.fit(X_train, y_train)
y_predicted = clf.predict(X_test)
auc = roc_auc_score(y_test, y_predicted)
print('auc: {:.3f}'.format(auc))
```

We get the following performance as an output:

```
auc: 0.779
```

This is a very good performance, as you can see comparing it to the leaderboard on OpenML.

How it works...

It is time to explain basic scikit-learn terminology relevant to this recipe. Neither of these concepts corresponds to existing machine learning algorithms, but to composable modules:

- Transformer (in scikit-learn): A class that is derived from `sklearn.base.TransformerMixin`; it has `fit()` and `transform()` methods. These involve preprocessing steps or feature selection.
- Predictor: A class that is derived from either `sklearn.base.ClassifierMixin` or `sklearn.base.RegressorMixin`; it has `fit()` and `predict()` methods. These are machine learning estimators, in other words, classifiers or regressors.
- Pipeline: An interface that wraps all steps together and gives you a single interface for all steps of the transformation and the resulting estimator. A pipeline again has `fit()` and `predict()` methods.

There are a few things to point out regarding our approach. As we said before, we have missing values, so we have to impute (meaning replace) missing values with other values. In this case, we replace missing values with -1. In the case of categorical variables, this will be a new category, while in the case of numerical variables, it will become a special value that the classifier will have to handle.

`ColumnTransformer` came with version 0.20 of scikit-learn and was a long-awaited feature. Since then, `ColumnTransformer` can often be seen like this, for example:

```
from sklearn.compose import ColumnTransformer, make_column_transformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

feature_preprocessing = make_column_transformer(
    (StandardScaler(), ['column1', 'column2']),
    (OneHotEncoder(), ['column3', 'column4', 'column5'])
)
```

`feature_preprocessing` can then be used as usual with the `fit()`, `transform()`, and `fit_transform()` methods:

```
processed_features = feature_preprocessing.fit_transform(X)
```

Here, `X` means our features.

Alternatively, we can put `ColumnTransformer` as a step into a pipeline, for example, like this:

```
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression

make_pipeline(
    feature_preprocessing,
    LogisticRegression()
)
```

Our classifier is a modified form of the random forest classifier. A random forest is a collection of decision trees, each trained on random subsets of the training data. The balanced random forest classifier (Chen *et al.*: <https://statistics.berkeley.edu/sites/default/files/tech-reports/666.pdf>) makes sure that each random subset is balanced between the two classes.

Since `NumericDifferenceTransformer` can provide lots of features, we will incorporate an extra step of model-based feature selection.

There's more...

You can see the complete example with the speed dating dataset, a few more custom transformers, and an extended imputation class in the GitHub repository of the `openml_speed_dating_pipeline_steps` library and notebook, on GitHub at <https://github.com/PacktPublishing/Artificial-Intelligence-with-Python-Cookbook/blob/master/chapter02/Transforming%20Data%20in%20Scikit-Learn.ipynb>.

Both `RangeTransformer` and `NumericDifferenceTransformer` could also have been implemented using `FunctionTransformer` in scikit-learn.

`ColumnTransformer` is especially handy for pandas DataFrames or NumPy arrays since it allows the specification of different operations for different subsets of the features. However, another option is `FeatureUnion`, which allows concatenation of the results from different transformations. For yet another way to chain our operations together, please have a look at `PandasPicker` in our repository.

See also

In this recipe, we used ANOVA f-values for univariate feature selection, which is relatively simple, yet effective. Univariate feature selection methods are usually simple filters or statistical tests that measure the relevance of a feature with regard to the target. There are, however, many different methods for feature selection, and scikit-learn implements a lot of them: https://scikit-learn.org/stable/modules/feature_selection.html.

Predicting house prices in PyTorch

In this recipe, the aim of the problem is to predict house prices in Ames, Iowa, given 81 features describing the house, area, land, infrastructure, utilities, and much more. The Ames dataset has a nice combination of categorical and continuous features, a good size, and, perhaps most importantly, it doesn't suffer from problems of potential redlining or data entry like other, similar datasets, such as Boston Housing. We'll concentrate on the main aspects of PyTorch modeling here. We'll do online learning, analogous to Keras, in the *Modeling and predicting in Keras* recipe in Chapter 1, *Getting Started with Artificial Intelligence in Python*. If you want to see more details on some of the steps, please look at our notebook on GitHub.

As a little extra, we will also demonstrate neuron importance for the models developed in PyTorch. You can try out different network architectures in PyTorch or model types. The focus in this recipe is on the methodology, not an exhaustive search for the best solution.

Getting ready

In order to prepare for the recipe, we need to do a few things. We'll download the data as in the previous recipe, *Transforming data in scikit-learn*, and perform some preprocessing by following these steps:

```
from sklearn.datasets import fetch_openml  
data = fetch_openml(data_id=42165, as_frame=True)
```

You can see the full dataset description at OpenML: <https://www.openml.org/d/42165>.

Let's look at the features:

```
import pandas as pd  
data_ames = pd.DataFrame(data.data, columns=data.feature_names)  
data_ames['SalePrice'] = data.target  
data_ames.info()
```

Here is the DataFrame information:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1460 entries, 0 to 1459
Data columns (total 81 columns):
Id           1460 non-null float64
MSSubClass   1460 non-null float64
MSZoning     1460 non-null object
LotFrontage  1201 non-null float64
LotArea       1460 non-null float64
Street        1460 non-null object
Alley         91 non-null object
LotShape      1460 non-null object
LandContour   1460 non-null object
Utilities     1460 non-null object
LotConfig     1460 non-null object
LandSlope     1460 non-null object
Neighborhood  1460 non-null object
Condition1    1460 non-null object
Condition2    1460 non-null object
BldgType      1460 non-null object
HouseStyle    1460 non-null object
OverallQual   1460 non-null float64
OverallCond   1460 non-null float64
YearBuilt     1460 non-null float64
YearRemodAdd  1460 non-null float64
RoofStyle     1460 non-null object
RoofMatl     1460 non-null object
Exterior1st   1460 non-null object
Exterior2nd   1460 non-null object
MasVnrType   1452 non-null object
MasVnrArea   1452 non-null float64
ExterQual     1460 non-null object
ExterCond     1460 non-null object
Foundation    1460 non-null object
BsmtQual     1423 non-null object
BsmtCond     1423 non-null object
BsmtExposure  1422 non-null object
BsmtFinType1  1423 non-null object
BsmtFinSF1   1460 non-null float64
BsmtFinType2  1422 non-null object
BsmtFinSF2   1460 non-null float64
BsmtUnfSF    1460 non-null float64
TotalBsmtSF  1460 non-null float64
Heating       1460 non-null object
HeatingQC     1460 non-null object
CentralAir    1460 non-null object
Electrical    1459 non-null object
```

```
1stFlrSF           1460 non-null float64
2ndFlrSF          1460 non-null float64
LowQualFinSF      1460 non-null float64
GrLivArea          1460 non-null float64
BsmtFullBath       1460 non-null float64
BsmtHalfBath       1460 non-null float64
FullBath           1460 non-null float64
HalfBath            1460 non-null float64
BedroomAbvGr        1460 non-null float64
KitchenAbvGr       1460 non-null float64
KitchenQual         1460 non-null object
TotRmsAbvGrd       1460 non-null float64
Functional          1460 non-null object
Fireplaces          1460 non-null float64
FireplaceQu        770 non-null object
GarageType          1379 non-null object
GarageYrBlt         1379 non-null float64
GarageFinish        1379 non-null object
GarageCars           1460 non-null float64
GarageArea          1460 non-null float64
GarageQual          1379 non-null object
GarageCond          1379 non-null object
PavedDrive          1460 non-null object
WoodDeckSF          1460 non-null float64
OpenPorchSF         1460 non-null float64
EnclosedPorch       1460 non-null float64
3SsnPorch           1460 non-null float64
ScreenPorch          1460 non-null float64
PoolArea            1460 non-null float64
PoolQC              7 non-null object
Fence                281 non-null object
MiscFeature          54 non-null object
MiscVal              1460 non-null float64
MoSold               1460 non-null float64
YrSold               1460 non-null float64
SaleType              1460 non-null object
SaleCondition         1460 non-null object
SalePrice             1460 non-null float64
dtypes: float64(38), object(43)
memory usage: 924.0+ KB
```

PyTorch and seaborn are installed by default in Colab. We will assume, even if you are working with your self-hosted install by now, that you'll have the libraries installed.

We'll use one more library, however, `captum`, which allows the inspection of PyTorch models for feature and neuron importance:

```
!pip install captum
```

There is one more thing. We'll assume you have a GPU available. If you don't have a GPU in your computer, we'd recommend you try this recipe on Colab. In Colab, you'll have to choose a runtime type with GPU.

After all these preparations, let's see how we can predict house prices.

How to do it...

The Ames Housing dataset is a small- to mid-sized dataset (1,460 rows) with 81 features, both categorical and numerical. There are no missing values.

In the Keras recipe previously, we've seen how to scale the variables. Scaling is important here because all variables have different scales. Categorical variables need to be converted to numerical types in order to feed them into our model. We have the choice of one-hot encoding, where we create dummy variables for each categorical factor, or ordinal encoding, where we number all factors and replace the strings with these numbers. We could feed the dummy variables in like any other float variable, while ordinal encoding would require the use of embeddings, linear neural network projections that re-order the categories in a multi-dimensional space.

We take the embedding route here:

```
import numpy as np
from category_encoders.ordinal import OrdinalEncoder
from sklearn.preprocessing import StandardScaler

num_cols = list(data_ames.select_dtypes(include='float'))
cat_cols = list(data_ames.select_dtypes(include='object'))

ordinal_encoder = OrdinalEncoder().fit(
    data_ames[cat_cols]
)
standard_scaler = StandardScaler().fit(
    data_ames[num_cols]
)

X = pd.DataFrame(
    data=np.column_stack([
        ordinal_encoder.transform(data_ames[cat_cols]),
        standard_scaler.transform(data_ames[num_cols])
    ]),
    columns=cat_cols + num_cols
)
```

We go through the data analysis, such as correlation and distribution plots, in a lot more detail in the notebook on GitHub.

Now we can split the data into training and test sets, as we did in previous recipes. Here, we add a stratification of the numerical variable. This makes sure that different sections (five of them) are included at equal measure in both training and test sets:

```
np.random.seed(12)
from sklearn.model_selection import train_test_split

bins = 5
sale_price_bins = pd.qcut(
    X['SalePrice'], q=bins, labels=list(range(bins)))
)
X_train, X_test, y_train, y_test = train_test_split(
    X.drop(columns='SalePrice'),
    X['SalePrice'],
    random_state=12,
    stratify=sale_price_bins
)
```

Before going ahead, let's look at the importance of the features using a model-independent technique.

Before we run anything, however, let's make sure we are running on the GPU:

```
device = torch.device('cuda')
torch.backends.cudnn.benchmark = True
```

Let's build our PyTorch model, similar to the *Classifying in scikit-learn, Keras, and PyTorch* recipe in Chapter 1, *Getting Started with Artificial Intelligence in Python*.

We'll implement a neural network regression with batch inputs using PyTorch. This will involve the following steps:

1. Converting data to torch tensors
2. Defining the model architecture
3. Defining the loss criterion and optimizer
4. Creating a data loader for batches
5. Running the training

Without further preamble, let's get to it:

1. Begin by converting the data to torch tensors:

```
from torch.autograd import Variable

num_features = list(
    set(num_cols) - set(['SalePrice', 'Id']))
)
X_train_num_pt = Variable(
    torch.cuda.FloatTensor(
        X_train[num_features].values
    )
)
X_train_cat_pt = Variable(
    torch.cuda.LongTensor(
        X_train[cat_cols].values
    )
)
y_train_pt = Variable(
    torch.cuda.FloatTensor(y_train.values)
).view(-1, 1)
X_test_num_pt = Variable(
    torch.cuda.FloatTensor(
        X_test[num_features].values
    )
)
X_test_cat_pt = Variable(
    torch.cuda.LongTensor(
        X_test[cat_cols].values
    ).long()
)
y_test_pt = Variable(
    torch.cuda.FloatTensor(y_test.values)
).view(-1, 1)
```

This makes sure we load our numerical and categorical data into separate variables, similar to NumPy. If you mix data types in a single variable (array/matrix), they'll become objects. We want to get our numerical variables as floats, and the categorical variables as long (or int) indexing our categories. We also separate the training and test sets.

Clearly, an ID variable should not be important in a model. In the worst case, it could introduce a target leak if there's any correlation of the ID with the target. We've removed it from further processing.

2. Define the model architecture:

```
class RegressionModel(torch.nn.Module):
    def __init__(self, X, num_cols, cat_cols,
device=torch.device('cuda'), embed_dim=2, hidden_layer_dim=2,
p=0.5):
        super(RegressionModel, self).__init__()
        self.num_cols = num_cols
        self.cat_cols = cat_cols
        self.embed_dim = embed_dim
        self.hidden_layer_dim = hidden_layer_dim
        self.embeddings = [
            torch.nn.Embedding(
                num_embeddings=len(X[col].unique()),
                embedding_dim=embed_dim
            ).to(device)
            for col in cat_cols
        ]
        hidden_dim = len(num_cols) + len(cat_cols) * embed_dim,
        # hidden layer
        self.hidden = torch.nn.Linear(torch.IntTensor(hidden_dim),
hidden_layer_dim).to(device)
        self.dropout_layer = torch.nn.Dropout(p=p).to(device)
        self.hidden_act = torch.nn.ReLU().to(device)
        # output layer
        self.output = torch.nn.Linear(hidden_layer_dim,
1).to(device)
    def forward(self, num_inputs, cat_inputs):
        '''Forward method with two input variables -
        numeric and categorical.
        '''
        cat_x = [
            torch.squeeze(embed(cat_inputs[:, i] - 1))
            for i, embed in enumerate(self.embeddings)
        ]
        x = torch.cat(cat_x + [num_inputs], dim=1)
        x = self.hidden(x)
        x = self.dropout_layer(x)
        x = self.hidden_act(x)
        y_pred = self.output(x)
        return y_pred

house_model = RegressionModel(
    data_ames, num_features, cat_cols
)
```

Our activation function on the two linear layers (dense layers, in Keras terminology) is the **rectified linear unit activation (ReLU)** function. Please note that we couldn't have encapsulated the same architecture (easily) as a sequential model because of the different operations occurring on categorical and numerical types.

3. Next, define the loss criterion and optimizer. We take the **mean square error (MSE)** as the loss and stochastic gradient descent as our optimization algorithm:

```
criterion = torch.nn.MSELoss().to(device)
optimizer = torch.optim.SGD(house_model.parameters(), lr=0.001)
```

4. Now, create a data loader to input a batch of data at a time:

```
data_batch = torch.utils.data.TensorDataset(
    X_train_num_pt, X_train_cat_pt, y_train_pt
)
dataloader = torch.utils.data.DataLoader(
    data_batch, batch_size=10, shuffle=True
)
```

We set a batch size of 10. Now we can do our training.

5. Run the training!

Since this seems so much more verbose than what we saw in Keras in the *Classifying in scikit-learn, Keras, and PyTorch* recipe in Chapter 1, *Getting Started with Artificial Intelligence in Python*, we commented this code quite heavily. Basically, we have to loop over epochs, and within each epoch an inference is performed, an error is calculated, and the optimizer applies the adjustments according to the error.

This is the loop over epochs without the inner loop for training:

```
from tqdm.notebook import trange

train_losses, test_losses = [], []
n_epochs = 30
for epoch in trange(n_epochs):
    train_loss, test_loss = 0, 0
    # training code will go here:
    # <...>

    # print the errors in training and test:
    if epoch % 10 == 0 :
        print(
```

```
    'Epoch: {} / {} \t'.format(epoch, 1000),
    'Training Loss: {:.3f} \t'.format(
        train_loss / len(dataloader)
    ),
    'Test Loss: {:.3f}'.format(
        test_loss / len(dataloader)
    )
)
```

The training is performed inside this loop over all the batches of the training data. This looks as follows:

```
for (x_train_num_batch,
      x_train_cat_batch,
      y_train_batch) in dataloader:
    # predict y by passing x to the model
    (x_train_num_batch,
     x_train_cat_batch, y_train_batch) = (
        x_train_num_batch.to(device),
        x_train_cat_batch.to(device),
        y_train_batch.to(device)
    )
    pred_ytrain = house_model.forward(
        x_train_num_batch, x_train_cat_batch
    )
    # calculate and print loss:
    loss = torch.sqrt(
        criterion(pred_ytrain, y_train_batch)
    )

    # zero gradients, perform a backward pass,
    # and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    train_loss += loss.item()
    with torch.no_grad():
        house_model.eval()
        pred_ytest = house_model.forward(
            X_test_num_pt, X_test_cat_pt
        )
        test_loss += torch.sqrt(
            criterion(pred_ytest, y_test_pt)
        )

    train_losses.append(train_loss / len(dataloader))
    test_losses.append(test_loss / len(dataloader))
```

This is the output we get. TQDM provides us with a helpful progress bar. At every tenth epoch, we print an update to show training and validation performance:

```
100% [██████████] 30/30 [01:02<00:00, 2.08s/it]
Epoch: 0/1000    Training Loss: 0.977    Test Loss: 0.993
Epoch: 10/1000   Training Loss: 0.395    Test Loss: 0.526
Epoch: 20/1000   Training Loss: 0.319    Test Loss: 0.550
```

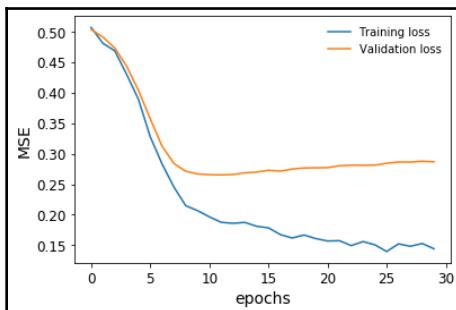
Please note that we take the square root of `nn.MSELoss` because `nn.MSELoss` in PyTorch is defined as follows:

```
((input-target)**2).mean()
```

Let's plot how our model performs for training and validation datasets during training:

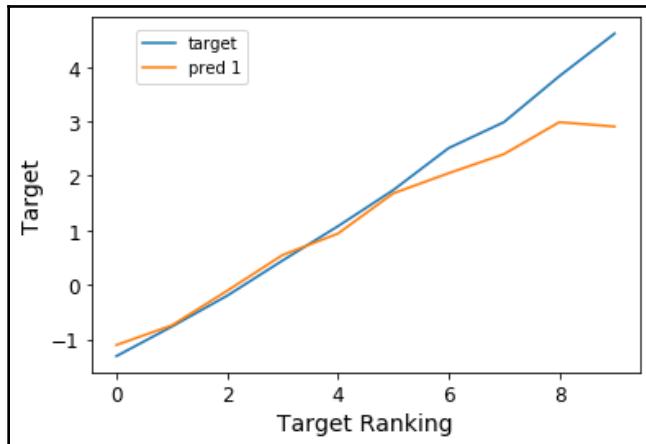
```
plt.plot(
    np.array(train_losses).reshape((n_epochs, -1)).mean(axis=1),
    label='Training loss'
)
plt.plot(
    np.array(test_losses).reshape((n_epochs, -1)).mean(axis=1),
    label='Validation loss'
)
plt.legend(frameon=False)
plt.xlabel('epochs')
plt.ylabel('MSE')
```

The following diagram shows the resulting plot:



We stopped our training just in time before our validation loss stopped decreasing.

We can also rank and bin our target variable and plot the predictions against it in order to see how the model is performing across the whole spectrum of house prices. This is to avoid the situation in regression, especially with MSE as the loss, that you only predict well for a mid-range of values, close to the mean, but don't do well for anything else. You can find the code for this in the notebook on GitHub. This is called a lift chart (here with 10 bins):



We can see that the model, in fact, predicts very closely across the whole range of house prices. In fact, we get a Spearman rank correlation of about 93% with very high significance, which confirms that this model performs with high accuracy.

How it works...

The deep learning neural network frameworks use different optimization algorithms. Popular among them are **Stochastic Gradient Descent (SGD)**, **Root Mean Square Propagation (RMSProp)**, and **Adaptive Moment Estimation (ADAM)**.

We defined stochastic gradient descent as our optimization algorithm. Alternatively, we could have defined other optimizers:

```
opt_SGD = torch.optim.SGD(net_SGD.parameters(), lr=LR)
opt_Momentum = torch.optim.SGD(net_Momentum.parameters(), lr=LR,
momentum=0.6)
opt_RMSprop = torch.optim.RMSprop(net_RMSprop.parameters(), lr=LR,
alpha=0.1)
opt_Adam = torch.optim.Adam(net_Adam.parameters(), lr=LR, betas=(0.8,
0.98))
```

SGD works the same as gradient descent except that it works on a single example at a time. The interesting part is that the convergence is similar to the gradient descent and is easier on the computer memory.

RMSProp works by adapting the learning rates of the algorithm according to the gradient signs. The simplest of the variants checks the last two gradient signs and then adapts the learning rate by increasing it by a fraction if they are the same, or decreases it by a fraction if they are different.

ADAM is one of the most popular optimizers. It's an adaptive learning algorithm that changes the learning rate according to the first and second moments of the gradients.

Captum is a tool that can help us understand the ins and outs of the neural network model learned on the datasets. It can assist in learning the following:

- Feature importance
- Layer importance
- Neuron importance

This is very important in learning interpretable neural networks. Here, integrated gradients have been applied to understand feature importance. Later, neuron importance is also demonstrated by using the layer conductance method.

There's more...

Given that we have our neural network defined and trained, let's find the important features and neurons using the captum library:

```
from captum.attr import (
    IntegratedGradients,
    LayerConductance,
    NeuronConductance
)
house_model.cpu()
for embedding in house_model.embeddings:
    embedding.cpu()

house_model.cpu()
ing_house = IntegratedGradients(forward_func=house_model.forward, )
#X_test_cat_pt.requires_grad_()
X_test_num_pt.requires_grad_()
attr, delta = ing_house.attribute(
    X_test_num_pt.cpu(),
    target=None,
```

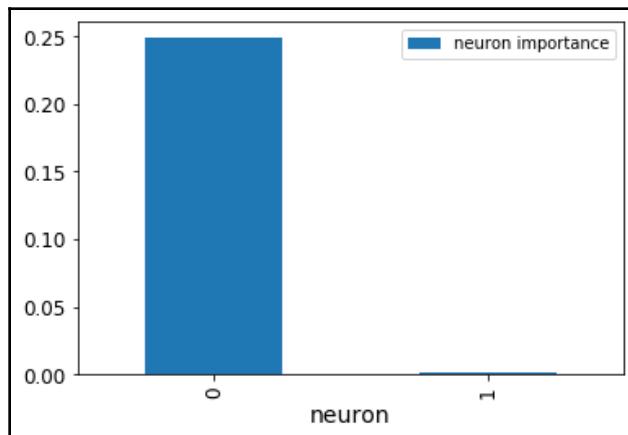
```
    return_convergence_delta=True,  
    additional_forward_args=X_test_cat_pt.cpu()  
)  
attr = attr.detach().numpy()
```

Now, we have a NumPy array of feature importances.

Layer and neuron importance can also be obtained using this tool. Let's look at the neuron importances of our first layer. We can pass on `house_model.act1`, which is the ReLU activation function on top of the first linear layer:

```
cond_layer1 = LayerConductance(house_model, house_model.act1)  
cond_vals = cond_layer1.attribute(X_test, target=None)  
cond_vals = cond_vals.detach().numpy()  
df_neuron = pd.DataFrame(data = np.mean(cond_vals, axis=0),  
columns=['Neuron Importance'])  
df_neuron['Neuron'] = range(10)
```

This is how it looks:



The diagram shows the neuron importances. Apparently, one neuron is just not important.

We can also see the most important variables by sorting the NumPy array we've obtained earlier:

```
df_feat = pd.DataFrame(np.mean(attr, axis=0), columns=['feature  
importance'])  
df_feat['features'] = num_features  
df_feat.sort_values(  
    by='feature importance', ascending=False  
) .head(10)
```

So here's a list of the 10 most important variables:

	feature importance	features
0	0.014723	OverallQual
9	0.014181	GrLivArea
12	0.013901	TotalBsmtSF
13	0.012599	1stFlrSF
31	0.007386	TotRmsAbvGrd
35	0.006422	GarageArea
8	0.006201	YearBuilt
11	0.005817	GarageCars
24	0.005403	FullBath
1	0.005221	OpenPorchSF

Often, feature importances can help us to both understand the model and prune our model to become less complex (and hopefully less overfitted).

See also

The PyTorch documentation includes everything you need to know about layer types, data loading, losses, metrics, and training: <https://pytorch.org/docs/stable/nn.html>

A detailed discussion about optimization algorithms can be found in the following article: <https://imaddabbura.github.io/post/gradient-descent-algorithm/>. Geoffrey Hinton and others explain mini-batch gradient descent in a presentation slide deck: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf. Finally, you can find all the details on ADAM in the article that introduced it: <https://arxiv.org/abs/1412.6980>.

Captum provides a lot of functionality as regards the interpretability and model inspection of PyTorch models. It's worth having a look at its comprehensive documentation at <https://captum.ai/>. Details can be found in the original paper at <https://arxiv.org/pdf/1703.01365.pdf>.

Live decisioning customer values

Let's assume we have the following scenario: we have a list of customers to call in order to sell them our product. Each phone call costs money in call center personal salaries, so we want to reduce these costs as much as possible. We have certain information about each customer that could help us determine whether they are likely to buy. After every call, we can update our model. The main goal is to call only the most promising customers and to improve our insights into which customers are more likely to pay for our product.

In this recipe, we will approach this with active learning, a strategy where we actively decide what to explore (and learn) next. Our model will help decide whom to call. Because we will update our model after each query (phone call), we will use online learning models.

Getting ready

We'll prepare for our recipe by downloading our dataset and installing a few libraries.

Again, we will get the data from OpenML:

```
!pip install -q openml

import openml
dataset = openml.datasets.get_dataset(1461)
X, y, categorical_indicator, _ = dataset.get_data(
    dataset_format='DataFrame',
    target=dataset.default_target_attribute
)
categorical_features = X.columns[categorical_indicator]
numeric_features = X.columns[
    [not(i) for i in categorical_indicator]
]
```

This dataset is called `bank-marketing`, and you can see a description on OpenML at <https://www.openml.org/d/1461>.

For each row, describing a single person, we have different features, numerical and categorical, that tell us about demographics and customer history.

To model the likelihood of customers signing up for our product, we will use the `scikit-multiflow` package that specializes in online models. We will also use the `category_encoders` package again:

```
!pip install scikit-multiflow category_encoders
```

With these two libraries in place, we can start the recipe.

How to do it...

We need to implement an exploration strategy and a model that is being continuously updated. We are using the online version of the random forest, the Hoeffding Tree, as our model. We are estimating the uncertainties at every step, and based on that we will return a candidate to call next.

As always, we will need to define a few preprocessing steps:

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import FunctionTransformer
import category_encoders as ce

ordinal_encoder = ce.OrdinalEncoder(
    cols=None, # all features that it encounters
    handle_missing='return_nan',
    handle_unknown='ignore'
).fit(X)

preprocessor = ColumnTransformer(
    transformers=[
        ('cat', ordinal_encoder, categorical_features),
        ('num', FunctionTransformer(validate=False), numeric_features)
    ]
)
preprocessor = preprocessor.fit(X)
```

Then we come to our active learning approach itself. This is inspired by modAL.models.ActiveLearner:

```
import numpy as np
from skmultiflow.trees.hoeffding_tree import HoeffdingTreeClassifier
from sklearn.metrics import roc_auc_score
import random

class ActivePipeline:
    def __init__(self, model, preprocessor, class_weights):
        self.model = model
        self.preprocessor = preprocessor
        self.class_weights = class_weights

    @staticmethod
    def values(X):
        if isinstance(X, (np.ndarray, np.int64)):
            return X
```

```
    else:
        return X.values

def preprocess(self, X):
    X_ = pd.DataFrame(
        data=self.values(X),
        columns=[
            'V1', 'V2', 'V3', 'V4',
            'V5', 'V6', 'V7', 'V8',
            'V9', 'V10', 'V11', 'V12',
            'V13', 'V14', 'V15', 'V16'
        ])
    return self.preprocessor.transform(X_)

def fit(self, X, ys):
    weights = [self.class_weights[y] for y in ys]
    self.model.fit(self.preprocess(X), self.values(ys))

def update(self, X, ys):
    if isinstance(ys, (int, float)):
        weight = self.class_weights[y]
    else:
        weight = [self.class_weights[y] for y in ys]

    self.model.partial_fit(
        self.preprocess(X),
        self.values(ys),
        weight
    )

def predict(self, X):
    return self.model.predict(
        self.preprocess(X)
    )

def predict_proba(self, X):
    return self.model.predict_proba(
        self.preprocess(X)
    )

@staticmethod
def entropy(preds):
    return -np.sum(
        np.log((preds + 1e-15) * preds)
        / np.log(np.prod(preds.size))
    )

def max_margin_uncertainty(self, X, method: str='entropy',
```

```

exploitation: float=0.9, favor_class: int=1, k: int=1
):
    '''similar to modAL.uncertainty.margin_uncertainty
    '''
    probs = self.predict_proba(X)
    if method=='margin':
        uncertainties = np.abs(probs[:,2] - probs[:, 1]) / 2.0
    elif method=='entropy':
        uncertainties = np.apply_along_axis(self.entropy, 1, probs[:, (1,2)])
    else: raise(ValueError('method not implemented!'))

    if favor_class is None:
        weights = uncertainties
    else: weights = (1.0 - exploitation) * uncertainties + exploitation *
        probs[:, favor_class]

    if self.sampling:
        ind = random.choices(
            range(len(uncertainties)), weights, k=k
        )
    else:
        ind = np.argsort(weights, axis=0)[::-1][:k]
    return ind, np.mean(uncertainties[ind])

def score(self, X, y, scale=True):
    probs = self.predict_proba(X, probability=2)
    if scale:
        probs = np.clip(probs - np.mean(probs) + 0.5, 0, 1)
    return roc_auc_score(y, probs)

```

Again, we create a scikit-learn-compatible class. It basically holds a machine learning model and a data preprocessor. We implement `fit()` and `predict()`, but also `score()` to get a model performance. We also implement an `update()` method that calls `partial_fit()` of the machine learning model. Calling `partial_fit()` instead of `fit()` considerably speeds up the computations, because we don't have to start from scratch every time we get new data.

Here's how to create the active learning pipeline:

```

active_pipeline = ActivePipeline(
    HoeffdingTreeClassifier(),
    preprocessor,
    class_weights.to_dict()
)
active_pipeline.model.classes = [0, 1, 2]

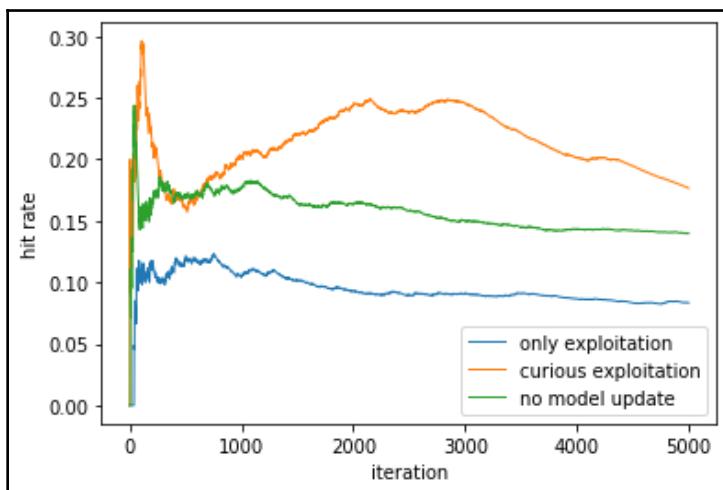
```

We can run different simulations on our dataset with this setup. For example, we can compare a lot of experimentation (0.5 exploitation) against only exploitation (1.0), or no learning at all after the first batch. We basically go through a loop:

- Via `active_pipeline.max_margin_uncertainty()`, we present data to the active pipeline, and get a number of data points that integrate uncertainty and target predictions according to our preferences of the integration method.
- Once we get the actual results for these data points, we can update our model: `active_pipeline.update()`.

You can see an example of this in the notebook on GitHub.

We can see that curious wins out after the first few examples. Exploitation is actually the least successful scheme. By not updating the model, performance deteriorates over time:



This is an ideal scenario for active learning or reinforcement learning, because, not unlike in reinforcement learning, uncertainty can be an additional criterion, apart from positive expectation, from a customer. Over time, this entropy reduction-seeking behavior reduces as the model's understanding of customers improves.

How it works...

It's worth delving a bit more into a few of the concepts and strategies employed in this recipe.

Active learning

Active learning means that we can actively query for more information; in other words, exploration is part of our strategy. This can be useful in scenarios where we have to actively decide what to learn, and where what we learn influences not only how much our model learns and how well, but also how much return on an investment we can get.

Hoeffding Tree

The Hoeffding Tree (also known as the *Very Fast Decision Tree*, VFDT for short) was introduced in 2001 by Geoff Hulten and others (*Mining time-changing data streams*). It is an incrementally growing decision tree for streamed data. Tree nodes are expanded based on the Hoeffding bound (or additive Chernoff bound). It was theoretically shown that, given sufficient training data, a model learned by the Hoeffding tree converges very closely to the one built by a non-incremental learner.

The Hoeffding bound is defined as follows:

$$\text{if } G(A_i, D) > G(A_j, D) + \epsilon(|S|, \delta), \forall j \neq i,$$

For dataset D and attribute A_i , and gain function $G()$, then we conclude that A_i is the best attribute with probability $1 - \delta$.

It's important to note that the Hoeffding Tree doesn't deal with data distributions that change over time.

Class weighting

Since we are dealing with an imbalanced dataset, let's use class weights. This basically means that we are upsampling the minority (signing up) class and downsampling the majority class (not signing up).

The formula for the class weights is as follows:

$$\frac{\text{n_samples}}{\text{n_classes} \times \text{np.bincount}(y)}$$

Similarly, in Python, we can write the following:

```
class_weights = len(X) / (y.astype(int).value_counts() * 2)
```

We can then use these class weights for sampling.

We'll close the recipe with a few more pointers.

See also

Only a few models in scikit-learn allow incremental or online learning. Refer to the list at <https://scikit-learn.org/stable/modules/computing.html>.

A few linear models include the `partial_fit()` method. The scikit-multiflow library specializes in incremental and online/streaming models: <https://scikit-multiflow.github.io/>

You can find more resources and ideas regarding active learning from a recent review that concentrates on biomedical image processing (Samuel Budd and others, *A Survey on Active Learning and Human-in-the-Loop Deep Learning for Medical Image Analysis*, 2019; <https://arxiv.org/abs/1910.02923>).

Our approach is inspired by the modalAI Python active learning package, which you can find at <https://modal-python.readthedocs.io/>. We recommend you check it out if you are interested in active learning approaches. A few more Python packages are available, as follows:

- Alipy: Active Learning in Python: <http://parnec.nuua.edu.cn/huangsj/alipy/>
- Active Learning: A Google repo about active learning: <https://github.com/google/active-learning>

One of the main decisions in active learning is the trade-off between exploration and exploitation. You can find out more about this in a paper called *Exploration versus exploitation in active learning: a Bayesian approach*: http://www.vincentlemaire-labs.fr/publis/ijcnn_2_2010_camera_ready.pdf

Battling algorithmic bias

Correctional Offender Management Profiling for Alternative Sanctions (COMPAS) is a commercial algorithm that assigns risk scores to criminal defendants based on their case records. This risk score corresponds to the likelihood of reoffending (recidivism) and of committing violent crimes, and this score is used in court to help determine sentences. The ProPublica organization obtained scores and data in 1 county in Florida, containing data on about 7,000 people. After waiting for 2 years to see who reoffended, they audited the COMPAS model in 2016 and found very concerning issues with the model. Among the ProPublica findings was discrimination according to gender, race, and ethnicity, particularly in the case of over-predicting recidivism for ethnic minorities.

Discrimination presents a major problem for AI systems, and illustrates the importance of auditing your model and the data you feed into your model. Models built on human decisions will amplify human biases if this bias is ignored. Not just from a legal perspective, but also ethically, we want to build models that don't disadvantage certain groups. This poses an interesting challenge for model building.



Generally, we would think that justice should be blind to gender or race. This means that court decisions should not take these sensitive variables like race or gender into account. However, even if we omit them from our model training, these sensitive variables might be correlated to some of the other variables, and therefore they can still affect decisions, to the detriment of protected groups such as minorities or women.

In this section, we are going to work with the COMPAS modeling dataset as provided by ProPublica. We are going to check for racial bias, and then create a model to remove it. You can find the original analysis by ProPublica at <https://github.com/propublica/compas-analysis>.

Getting ready

Before we can start, we'll first download the data, mention issues in preprocessing, and install the libraries.

Let's get the data:

```
!wget
https://raw.githubusercontent.com/propublica/compas-analysis/master/compas-
scores-two-years.csv
import pandas as pd
date_cols = [
    'compas_screening_date', 'c_offense_date',
    'c_arrest_date', 'r_offense_date',
    'vr_offense_date', 'screening_date',
    'v_screening_date', 'c_jail_in',
    'c_jail_out', 'dob', 'in_custody',
    'out_custody'
]
data = pd.read_csv(
    'compas-scores-two-years.csv',
    parse_dates=date_cols
)
```

Each row represents the risk of violence and the risk of recidivism scores for an inmate. The final column, `two_year_recid`, indicates our target.

ProPublica compiled their dataset from different sources, which they matched up according to the names of offenders:

- Criminal records from the Broward County Clerk's Office website
- Public incarceration records from the Florida Department of Corrections website
- COMPAS scores, which they obtained through a public record information request

We can highlight a few issues in the dataset:

1. The column `race` is a protected category. It should not be used as a feature for model training, but as a control.
2. There are full names in the dataset, which will not be useful, or might even give away the ethnicity of the inmates.
3. There are case numbers in the dataset. These will likely not be useful for training a model, although they might have some target leakage in the sense that increasing case numbers might give an indication of the time, and there could be a drift effect in the targets over time.
4. There are missing values. We will need to carry out imputation.
5. There are date stamps. These will probably not be useful and might even come with associated problems (see point 3). However, we can convert these features into UNIX epochs, which indicates the number of seconds that have elapsed since 1970, and then calculate time periods between date stamps, for example, by repurposing `NumericDifferenceTransformer` that we saw in an earlier recipe. We can then use these periods as model features rather than the date stamps.
6. We have several categorical variables.
7. The charge description (`c_charge_desc`) might need to be cleaned up.

Mathias Barenstein has pointed out (<https://arxiv.org/abs/1906.04711>) a data processing error in ProPublica's cutoff that resulted in keeping 40% more recidivists than they should have. We'll apply his correction to the two-year cutoff:

```
import datetime
indexes = data.compas_screening_date <= pd.Timestamp(datetime.date(2014, 4,
1))
assert indexes.sum() == 6216
data = data[indexes]
```

We will use a few libraries in this recipe, which can be installed as follows:

```
!pip install category-encoders
```

category-encoders is a library that provides functionality for categorical encoding beyond what scikit-learn provides.

How to do it...

Let's get some basic terminology out of the way first. We need to come up with metrics for fairness. But what does fairness (or, if we look at unfairness, bias) mean?

Fairness can refer to two very different concepts:

- [equal opportunity]: There should be no difference in the relative ratios of predicted by the model versus actually true; or
- [equal outcome]: There should be no difference between model outcomes at all.

The first is also called **equal odds**, while the latter refers to **equal false positive rates**. While equal opportunity means that each group should be given the same chance regardless of their group, the equal outcome strategy implies that the underperforming group should be given more lenience or chances relative to the other group(s).

We'll go with the idea of false positive rates, which intuitively appeals, and which is enshrined in law in many jurisdictions in the case of equal employment opportunities. We'll provide a few resources about these terms in the *See also* section.

Therefore, the logic for the impact calculation is based on values in the confusion matrix, most importantly, false positives, which we've just mentioned. These cases are predicted positive even though they are actually negative; in our case, people predicted as reoffenders, who are not reoffenders. Let's write a function for this:

```
def confusion_metrics(actual, scores, threshold):  
    y_predicted = scores.apply(  
        lambda x: x >= threshold  
    ).values  
    y_true = actual.values  
    TP = (  
        (y_true==y_predicted) &  
        (y_predicted==1)  
    ).astype(int)  
    FP = (  
        (y_true!=y_predicted) &  
        (y_predicted==1)
```

```

).astype(int)
TN = (
    (y_true==y_predicted) &
    (y_predicted==0)
).astype(int)
FN = (
    (y_true!=y_predicted) &
    (y_predicted==0)
).astype(int)
return TP, FP, TN, FN

```

We can now use this function in order to summarize the impact on particular groups with this code:

```

def calculate_impacts(data, sensitive_column='race', recid_col='is_recid',
score_col='decile_score.1', threshold=5.0):
    if sensitive_column == 'race':
        norm_group = 'Caucasian'
    elif sensitive_column == 'sex':
        norm_group = 'Male'
    else:
        raise ValueError('sensitive column not implemented')
    TP, FP, TN, FN = confusion_metrics(
        actual=data[recid_col],
        scores=data[score_col],
        threshold=threshold
    )
    impact = pd.DataFrame(
        data=np.column_stack([
            FP, TN, FN, TN,
            data[sensitive_column].values,
            data[recid_col].values,
            data[score_col].values / 10.0
        ]),
        columns=['FP', 'TP', 'FN', 'TN', 'sensitive', 'reoffend', 'score']
    ).groupby(by='sensitive').agg({
        'reoffend': 'sum', 'score': 'sum',
        'sensitive': 'count',
        'FP': 'sum', 'TP': 'sum', 'FN': 'sum', 'TN': 'sum'
    }).rename(
        columns={'sensitive': 'N'}
    )

    impact['FPR'] = impact['FP'] / (impact['FP'] + impact['TN'])
    impact['FNR'] = impact['FN'] / (impact['FN'] + impact['TP'])
    impact['reoffend'] = impact['reoffend'] / impact['N']
    impact['score'] = impact['score'] / impact['N']
    impact['DFP'] = impact['FPR'] / impact.loc[norm_group, 'FPR']

```

```
impact['DFN'] = impact['FNR'] / impact.loc[norm_group, 'FNR']
return impact.drop(columns=['FP', 'TP', 'FN', 'TN'])
```

This first calculates the confusion matrix with true positives and false negatives, and then encodes the **adverse impact ratio (AIR)**, known in statistics also as the **Relative Risk Ratio (RRR)**. Given any performance metric, we can write the following:

$$\text{AIR}_{\text{metric}} = \text{metric}_{\text{protected_group}} / \text{metric}_{\text{norm_group}}$$

This expresses an expectation that the metric for the protected group (African-Americans) should be the same as the metric for the norm group (Caucasians). In this case, we'll get 1.0. If the metric of the protected group is more than 20 percentage points different to the norm group (that is, lower than 0.8 or higher than 1.2), we'll flag it as a significant discrimination.



Norm group: a **norm group**, also known as a **standardization sample** or **norming group**, is a sample of the dataset that represents the population to which the statistic is intended to be compared. In the context of bias, its legal definition is the group with the highest success, but in some contexts, the entire dataset or the most frequent group are taken as the baseline instead. Pragmatically, we take the white group, since they are the biggest group, and the group for which the model works best.

In the preceding function, we calculate the false positive rates by sensitive group. We can then check whether the false positive rates for African-Americans versus Caucasians are disproportionate, or rather whether the false positive rates for African-Americans are much higher. This would mean that African-Americans get flagged much more often as repeat offenders than they should be. We find that this is indeed the case:

	reoffend	score	N	FPR	FNR	DFP	DFN
sensitive							
African-American	0.471169	0.521822	3139	0.439157	0.317949	1.895936	1.301351
Asian	0.250000	0.260714	28	0.047619	0.090909	0.205582	0.372087
Caucasian	0.329737	0.354268	2132	0.231631	0.244322	1.000000	1.000000
Hispanic	0.303730	0.337655	563	0.206633	0.230198	0.892079	0.942191
Native American	0.500000	0.571429	14	0.285714	0.000000	1.233492	0.000000
Other	0.311765	0.282941	340	0.141026	0.263736	0.608839	1.079461

A short explanation about this table follows:

- reoffend: frequencies of reoffending
- score: average score for the group
- N: the total number of people in the group
- FPR: false positive rates
- FNR: false negative rates
- DFP: disproportionate false positive
- DFN: disproportionate false negative

The last FPR and FNR columns together can give an idea about the general quality of the model. If both are high, the model just doesn't perform well for the particular group. The last two columns express the adverse impact ratio of FPR and FNR ratios, respectively, which is what we'll mostly focus on. We need to reduce the racial bias in the model by reducing the FPR of African-Americans to a tolerable level.

Let's do some preprocessing and then we'll build the model:

```
from sklearn.feature_extraction.text import CountVectorizer
from category_encoders.one_hot import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

charge_desc = data['c_charge_desc'].apply(lambda x: x if isinstance(x, str)
else '')
count_vectorizer = CountVectorizer(
    max_df=0.85, stop_words='english',
    max_features=100, decode_error='ignore'
)
charge_desc_features = count_vectorizer.fit_transform(charge_desc)

one_hot_encoder = OneHotEncoder()
charge_degree_features = one_hot_encoder.fit_transform(
    data['c_charge_degree']
)

data['race_black'] = data['race'].apply(lambda x: x == 'African-
American').astype(int)
stratification = data['race_black'] + (data['is_recid']).astype(int) * 2
```

CountVectorizer counts a vocabulary of words, indicating how many times each word is used. This is called a bag-of-words representation, and we apply it to the charge description column. We exclude English stop words, which are very common words such as prepositions (such as **on** or **at**) and personal pronouns (for example **I** or **me**); we also limit the vocabulary to 100 words and words that don't appear in more than 85% of the fields.

We apply dummy encoding (one-hot encoding) to the charge degree.

Why do we use two different transformations? Basically, the description is a textual description of why someone was charged with a crime. Every field is different. If we used one-hot encoding, every field would get their own dummy variable, and we wouldn't be able to see any commonalities between fields.

In the end, we create a new variable for stratification in order to make sure that we have similar proportions in the training and test datasets for both recidivism (our target variable) and whether someone is African-American. This will help us to calculate metrics to check for discrimination:

```
y = data['is_recid']
X = pd.DataFrame(
    data=np.column_stack(
        [data[['juv_fel_count', 'juv_misd_count',
               'juv_other_count', 'priors_count', 'days_b_screening_arrest']],
         charge_degree_features,
         charge_desc_features.todense()
    ],
    columns=['juv_fel_count', 'juv_misd_count', 'juv_other_count',
             'priors_count', 'days_b_screening_arrest'] \
        + one_hot_encoder.get_feature_names() \
        + count_vectorizer.get_feature_names(),
    index=data.index
)
X['jailed_days'] = (data['c_jail_out'] - data['c_jail_in']).apply(lambda x:
abs(x.days))
X['waiting_jail_days'] = (data['c_jail_in'] -
data['c_offense_date']).apply(lambda x: abs(x.days))
X['waiting_arrest_days'] = (data['c_arrest_date'] -
data['c_offense_date']).apply(lambda x: abs(x.days))
X.fillna(0, inplace=True)

columns = list(X.columns)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.33,
    random_state=42,
```

```
    stratify=stratification  
    ) # we stratify by black and the target
```

We do some data engineering, deriving variables to record how many days someone has spent in jail, has waited for a trial, or has waited for an arrest.

We'll build a neural network model using jax similar to the one we've encountered in the *Classifying in scikit-learn, Keras, and PyTorch* recipe in Chapter 1, *Getting Started with Artificial Intelligence in Python*. This time, we'll do a fully fledged implementation:

```
import jax.numpy as jnp  
from jax import grad, jit, vmap, ops, lax  
import numpy.random as npr  
import numpy as np  
import random  
from tqdm import trange  
from sklearn.base import ClassifierMixin  
from sklearn.preprocessing import StandardScaler  
  
class JAXLearner(ClassifierMixin):  
    def __init__(self, layer_sizes=[10, 5, 1], epochs=20, batch_size=500,  
     lr=1e-2):  
        self.params = self.construct_network(layer_sizes)  
        self.perex_grads = jit(grad(self.error))  
        self.epochs = epochs  
        self.batch_size = batch_size  
        self.lr = lr  
  
    @staticmethod  
    def construct_network(layer_sizes=[10, 5, 1]):  
        '''Please make sure your final layer corresponds to targets in  
        dimensions.  
        '''  
        def init_layer(n_in, n_out):  
            W = npr.randn(n_in, n_out)  
            b = npr.randn(n_out, )  
            return W, b  
        return list(map(init_layer, layer_sizes[:-1], layer_sizes[1:]))  
  
    @staticmethod  
    def sigmoid(X): # or tanh  
        return 1/(1+jnp.exp(-X))  
  
    def _predict(self, inputs):  
        for W, b in self.params:  
            outputs = jnp.dot(inputs, W) + b  
            inputs = self.sigmoid(outputs)
```

```

        return outputs

    def predict(self, inputs):
        inputs = self.standard_scaler.transform(inputs)
        return onp.asarray(self._predict(inputs))

    @staticmethod
    def mse(preds, targets, other=None):
        return jnp.sqrt(jnp.sum((preds - targets)**2))

    @staticmethod
    def penalized_mse(preds, targets, sensitive):
        err = jnp.sum((preds - targets)**2)
        err_s = jnp.sum((preds * sensitive - targets * sensitive)**2)
        penalty = jnp.clip(err_s / err, 1.0, 2.0)
        return err * penalty

    def error(self, params, inputs, targets, sensitive):
        preds = self._predict(inputs)
        return self.penalized_mse(preds, targets, sensitive)

    def fit(self, X, y, sensitive):
        self.standard_scaler = StandardScaler()
        X = self.standard_scaler.fit_transform(X)
        N = X.shape[0]
        indexes = list(range(N))
        steps_per_epoch = N // self.batch_size

        for epoch in trange(self.epochs, desc='training'):
            random.shuffle(indexes)
            index_offset = 0
            for step in trange(steps_per_epoch, desc='iteration'):
                grads = self.perex_grads(
                    self.params,
                    X[indexes[index_offset:index_offset+self.batch_size], :],
                    y[indexes[index_offset:index_offset+self.batch_size]],

                    sensitive[indexes[index_offset:index_offset+self.batch_size]]
                )
                # print(grads)
                self.params = [(W - self.lr * dW, b - self.lr * db)
                               for (W, b), (dW, db) in zip(self.params, grads)]
                index_offset += self.batch_size

```

This is a scikit-learn wrapper of a JAX neural network. For scikit-learn compatibility, we inherit from `ClassifierMixin` and implement `fit()` and `predict()`. The most important part here is the penalized MSE method, which, in addition to model predictions and targets, takes into account a sensitive variable.

Let's train it and check the performance. Please note that we feed in `x`, `y`, and `sensitive_train`, which we define as the indicator variable for African-American for the training dataset:

```
sensitive_train = X_train.join(
    data, rsuffix='_right'
)[['race_black']]
jax_learner = JAXLearner([X.values.shape[1], 100, 1])
jax_learner.fit(
    X_train.values,
    y_train.values,
    sensitive_train.values
)
```

We visualize the statistics as follows:

```
X_predicted = pd.DataFrame(
    data=jax_learner.predict(
        X_test.values
    ) * 10,
    columns=['score'],
    index=X_test.index
).join(
    data[['sex', 'race', 'is_recid']],
    rsuffix='_right'
)
calculate_impacts(X_predicted, score_col='score')
```

This is the table we get:

	reoffend	score	N	FPR	FNR	DFP	DFN
sensitive							
African-American	0.471042	2.923125	1036	0.832117	0.563981	0.982103	1.478884
Asian	0.222222	3.543877	9	1.000000	NaN	1.180247	NaN
Caucasian	0.335188	2.969506	719	0.847280	0.381356	1.000000	1.000000
Hispanic	0.293478	3.098000	184	0.861538	0.419355	1.016828	1.099642
Native American	0.500000	3.567670	2	1.000000	NaN	1.180247	NaN
Other	0.294118	2.724061	102	0.847222	0.450000	0.999931	1.180000

We can see that the disproportionate false positive rate for African-Americans is very close to (and even lower than) 1.0, which is what we wanted. The test set is small and doesn't contain enough samples for Asians and native Americans, so we can't calculate meaningful statistics for those groups. We could, however, extend our approach to encompass these two groups as well if we wanted to ensure that these two groups had equal false positive rates.

How it works...

The keys for this to work are custom objective functions or loss functions. This is far from straightforward in scikit-learn, although we will show an implementation in the following section.

Generally, there are different possibilities for implementing your own cost or loss functions.

- LightGBM, Catboost, and XGBoost each provide an interface with many loss functions and the ability to define custom loss functions.
- PyTorch and Keras (TensorFlow) provide an interface.
- You can implement your model from scratch (this is what we've done in the main recipe).

Scikit-learn generally does not provide a public API for defining your own loss functions. For many algorithms, there is only a single choice, and sometimes there are a couple of alternatives. The rationale in the case of split criteria with trees is that loss functions have to be performant, and only Cython implementations will guarantee this. This is only available in a non-public API, which means it is more difficult to use.

Finally, when there's no (straightforward) way to implement a custom loss, you can wrap your algorithms in a general optimization scheme such as genetic algorithms.

In neural networks, as long as you provide a differentiable loss function, you can plug in anything you want.

Basically, we were able to encode the adverse impact as a penalty term with the **Mean Squared Error (MSE)** function. It is based on the MSE that we've mentioned before, but has a penalty term for adverse impact. Let's look again at the loss function:

```
@staticmethod
def penalized_mse(preds, targets, sensitive):
    err = jnp.sum((preds - targets)**2)
    err_s = jnp.sum((preds * sensitive - targets * sensitive)**2)
    penalty = jnp.clip(err_s / err, 1.0, 2.0)
    return err * penalty
```

The first thing to note is that instead of two variables, we pass three variables. `sensitive` is the variable relevant to the adverse impact, indicating if we have a person from a protected group.

The calculation works as follows:

1. We calculate the MSE overall, `err`, from model predictions and targets.
2. We calculate the MSE for the protected group, `err_s`.
3. We take the ratio of the MSE for the protected group over the MSE overall (AIR) and limit it to between 1.0 and 2.0. We don't want values lower than 1, because we are only interested in the AIR if it's negatively affecting the protected group.
4. We then multiply AIR by the overall MSE.

As for 2, the MSE can simply be calculated by multiplying the predictions and targets, each by `sensitive`. That would cancel out all points, where `sensitive` is equal to 0.

As for 4, it might seem that this would cancel out the overall error, but we see that it actually seems to work. We probably could have added the two terms as well to give both errors a similar importance.

We use the autograd functionality in Jax to differentiate this.

There's more...

In the following, we'll use the non-public scikit-learn API to implement a custom split criterion for decision trees. We'll use this to train a random forest model with the COMPAS dataset:



This extends the implementation of the Hellinger criterion by Evgeni Dubov (<https://github.com/EvgeniDubov/hellinger-distance-criterion>).

```
%%cython

from sklearn.tree._criterion cimport ClassificationCriterion
from sklearn.tree._criterion cimport SIZE_t

import numpy as np
cdef double INFINITY = np.inf

from libc.math cimport sqrt, pow
from libc.math cimport abs
```

```

cdef class PenalizedHellingerDistanceCriterion(ClassificationCriterion):
    cdef double proxy_impurity_improvement(self) nogil:
        cdef double impurity_left
        cdef double impurity_right
        self.children_impurity(&impurity_left, &impurity_right)
        return impurity_right + impurity_left

    cdef double impurity_improvement(self, double impurity) nogil:
        cdef double impurity_left
        cdef double impurity_right

        self.children_impurity(&impurity_left, &impurity_right)
        return impurity_right + impurity_left

    cdef double node_impurity(self) nogil:
        cdef SIZE_t* n_classes = self.n_classes
        cdef double* sum_total = self.sum_total
        cdef double hellinger = 0.0
        cdef double sq_count
        cdef double count_k
        cdef SIZE_t k
        cdef SIZE_t c

        for k in range(self.n_outputs):
            for c in range(n_classes[k]):
                hellinger += 1.0

        return hellinger / self.n_outputs

    cdef void children_impurity(self, double* impurity_left,
                                double* impurity_right) nogil:
        cdef SIZE_t* n_classes = self.n_classes
        cdef double* sum_left = self.sum_left
        cdef double* sum_right = self.sum_right
        cdef double hellinger_left = 0.0
        cdef double hellinger_right = 0.0
        cdef double count_k1 = 0.0
        cdef double count_k2 = 0.0

        cdef SIZE_t k
        cdef SIZE_t c

        # stop splitting in case reached pure node with 0 samples of second
        class
        if sum_left[1] + sum_right[1] == 0:
            impurity_left[0] = -INFINITY
            impurity_right[0] = -INFINITY
            return
        for k in range(self.n_outputs):

```

```

        if(sum_left[0] + sum_right[0] > 0):
            count_k1 = sqrt(sum_left[0] / (sum_left[0] + sum_right[0]))
        if(sum_left[1] + sum_right[1] > 0):
            count_k2 = sqrt(sum_left[1] / (sum_left[1] + sum_right[1]))

        hellinger_left += pow((count_k1 - count_k2), 2)
        if(sum_left[0] + sum_right[0] > 0):
            count_k1 = sqrt(sum_right[0] / (sum_left[0] +
sum_right[0]))
        if(sum_left[1] + sum_right[1] > 0):
            count_k2 = sqrt(sum_right[1] / (sum_left[1] +
sum_right[1]))

        if k==0:
            hellinger_right += pow((count_k1 - count_k2), 2)
        else:
            hellinger_right -= pow((count_k1 - count_k2), 2)
    impurity_left[0] = hellinger_left / self.n_outputs
    impurity_right[0] = hellinger_right / self.n_outputs

```

Let's use this for training and test it:

```

ensemble = [
    DecisionTreeClassifier(
        criterion=PenalizedHellingerDistanceCriterion(
            2, np.array([2, 2], dtype='int64')
        ),
        max_depth=100
    ) for i in range(100)
]
for model in ensemble:
    model.fit(
        X_train,
        X_train.join(
            data,
            rsuffix='_right'
        )[['is_recid', 'race_black']]
    )
Y_pred = np.array(
    [model.predict(X_test) for model in
     ensemble]
)
predictions2 = Y_pred.mean(axis=0)

```

This gives us an AUC of 0.62:

	reoffend	score	N	FPR	FNR	DFP	DFN
sensitive							
African-American	0.471042	0.575290	1036	0.456204	0.322727	1.290330	1.263508
Asian	0.222222	0.333333	9	0.142857	0.000000	0.404057	0.000000
Caucasian	0.335188	0.422809	719	0.353556	0.255422	1.000000	1.000000
Hispanic	0.293478	0.364130	184	0.307692	0.230769	0.870278	0.903483
Native American	0.500000	0.500000	2	0.000000	0.000000	0.000000	0.000000
Other	0.294118	0.274510	102	0.208333	0.229730	0.589250	0.899414

We can see that, although we came a long way, we didn't completely remove all bias. 30% (DFP for African-Americans) would still be considered unacceptable. We could try different refinements or sampling strategies to improve the result. Unfortunately, we wouldn't be able to use this model in practice.

As an example, a way to address this is to do model selection within the random forest. Since each of the trees would have their own way of classifying people, we could calculate the adverse impact statistics for each of the individual trees or combinations of trees. We could remove trees until we are left with a subset of trees that satisfy our adverse impact conditions. This is beyond the scope of this chapter.

See also

You can read up more on algorithmic fairness in different places. There's a wide variety of literature available on fairness:

- A Science Magazine article about the COMPAS model (Julia Dressel and Hany Farid, 2018, *The accuracy, fairness, and limits of predicting recidivism*): <https://advances.sciencemag.org/content/4/1/eaao5580>
- A comparative study of fairness-enhancing interventions in machine learning (Sorelle Friedler and others, 2018): <https://arxiv.org/pdf/1802.04422.pdf>
- A Survey on Bias and Fairness in Machine Learning (Mehrabi and others, 2019): <https://arxiv.org/pdf/1908.09635.pdf>
- The effect of explaining fairness (Jonathan Dodge and others, 2019): <https://arxiv.org/pdf/1901.07694.pdf>

Different Python libraries are available for tackling bias (or, inversely, algorithmic fairness):

- fairlearn: <https://github.com/fairlearn/fairlearn>
- AIF360: <https://github.com/IBM/AIF360>
- FairML: <https://github.com/adebayoj/fairml>
- BlackBoxAuditing: <https://github.com/algofairness/BlackBoxAuditing>
- Balanced Committee Election: <https://github.com/huanglx12/Balanced-Committee-Election>

Finally, Scikit-Lego includes functionality for fairness: <https://scikit-lego.readthedocs.io/en/latest/fairness.html>

While you can find many datasets on recidivism by performing a Google dataset search (<https://toolbox.google.com/datasetsearch>), there are many more applications and corresponding datasets where fairness is important, such as credit scoring, face recognition, recruitment, or predictive policing, to name just a few.

There are different places to find out more about custom losses. The article *Custom loss versus custom scoring* (<https://kiwidamien.github.io/custom-loss-vs-custom-scoring.html>) affords a good overview. For implementations of custom loss functions in gradient boosting, towardsdatascience (<https://towardsdatascience.com/custom-loss-functions-for-gradient-boosting-f79c1b40466d>) is a good place to start.

Forecasting CO₂ time series

In this recipe, we will test out some well-known models (ARIMA, SARIMA) and signal decomposition by forecasting using Facebook's Prophet library on the time series data, in order to check their performance at forecasting our time series of CO₂ values.

Getting ready

In order to prepare for this recipe, we'll install libraries and download a dataset.

We'll use the statsmodels package and prophet:

```
pip install statsmodels fbprophet
```

We will analyze the CO₂ concentration data in this recipe. You can see the data loading in the notebook on GitHub accompanying this recipe, or in the scikit-learn **Gaussian process regression (GPR)** example regarding Mauna Loa CO₂ data: https://scikit-learn.org/stable/auto_examples/gaussian_process/plot_gpr_co2.html#sphx-glr-auto-examples-gaussian-process-plot-gpr-co2-py

This dataset is one of the earliest recordings available on atmospheric recordings of CO₂. As it will be later observed, this data follows a sinusoidal pattern, with the CO₂ concentration rising in winter and falling in the summer owing to the decreasing quantity of plants and vegetation in the winter season:

```
x,y = load_mauna_loa_atmospheric_co2()
```

The dataset contains the average CO₂ concentration measured at Mauna Loa Observatory in Hawaii from 1958 to 2001. We will model the CO₂ concentration with respect to that.

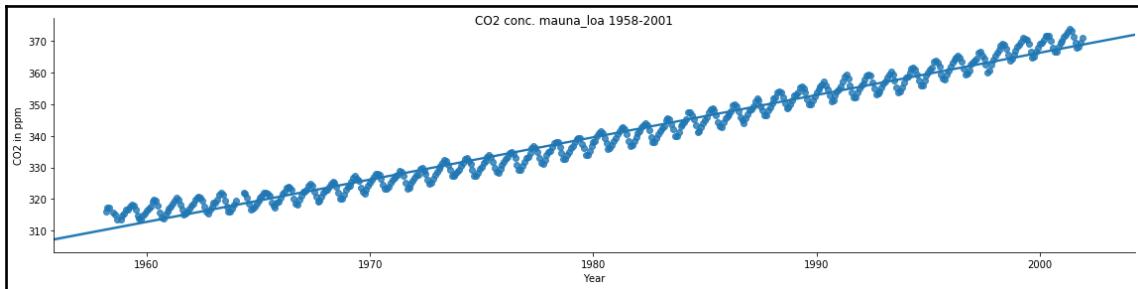
How to do it...

Now we'll get to forecasting our time series of CO₂ data. We'll first explore the dataset, and then we'll apply the ARIMA and SARIMA techniques.

1. Let's have a look at the time series:

```
df_CO2 = pd.DataFrame(data = X, columns = ['Year'])
df_CO2['CO2 in ppm'] = y
lm = sns.lmplot(x='Year', y='CO2 in ppm', data=df_CO2, height=4,
                 aspect=4)
fig = lm.fig
fig.suptitle('CO2 conc. mauna_loa 1958-2001', fontsize=12)
```

Here's the graph:



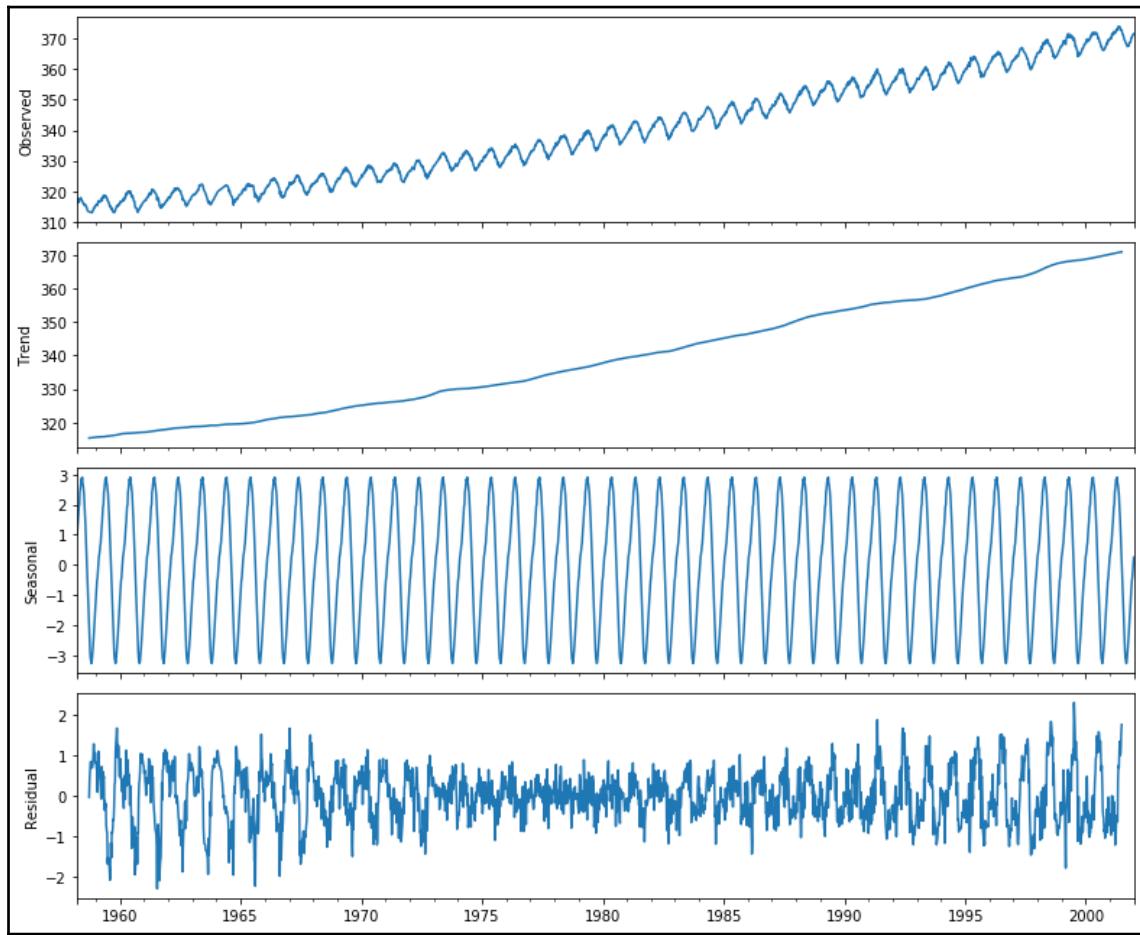
The script here shows the time series seasonal decomposition of the CO₂ data, showing a clear seasonal variation in the CO₂ concentration, which can be traced back to the biology:

```
import statsmodels.api as smd
d = sm.datasets.co2.load_pandas()
co2 = d.data
co2.head()
y = co2['co2']
y = y.fillna(
    y.interpolate())
) # Fill missing values by interpolation
```

Now that we have preprocessed data for decomposition, let's go ahead with it:

```
from pylab import rcParams
rcParams['figure.figsize'] = 11, 9
result = sm.tsa.seasonal_decompose(y, model='additive')
pd.plotting.register_matplotlib_converters()
result.plot()
plt.show()
```

Here, we see the decomposition: the observed time series, its trend, seasonal components, and what remains unexplained, the residual element:



Now, let's analyze the time series.

Analyzing time series using ARIMA and SARIMA

We will fit ARIMA and SARIMA models to the dataset.

We'll define our two models and apply them to each point in the test dataset. Here, we iteratively fit the model on all the points and predict the next point, as a one-step-ahead.

1. First, we split the data:

```
# taking a 90/10 split between training and testing:  
future = int(len(y) * 0.9)  
print('number of train samples: %d test samples %d' (future,  
len(y)-future)  
)  
train, test = y[:future], y[future:]
```

This leaves us with 468 samples for training and 53 for testing.

2. Next, we define the models:

```
from statsmodels.tsa.arima_model import ARIMA  
from statsmodels.tsa.statespace.sarimax import SARIMAX  
  
def get_arima_model(history, order=(5, 1, 0)):  
    return ARIMA(history, order=order)  
  
def get_sarima_model(  
    history,  
    order=(5, 1, 1),  
    seasonal_order=(0, 1, 1, 4)  
):  
    return SARIMAX(  
        history,  
        order=order,  
        enforce_stationarity=True,  
        enforce_invertibility=False,  
        seasonal_order=seasonal_order  
)
```

3. Then we train the models:

```
from sklearn.metrics import mean_squared_error  
  
def apply_model(train, test, model_fun=get_arima_model):  
    '''we just roll with the model and apply it to successive  
    time steps  
    '''  
    history = list(train)  
    predictions = []  
    for t in test:  
        model = model_fun(history).fit(disp=0)
```

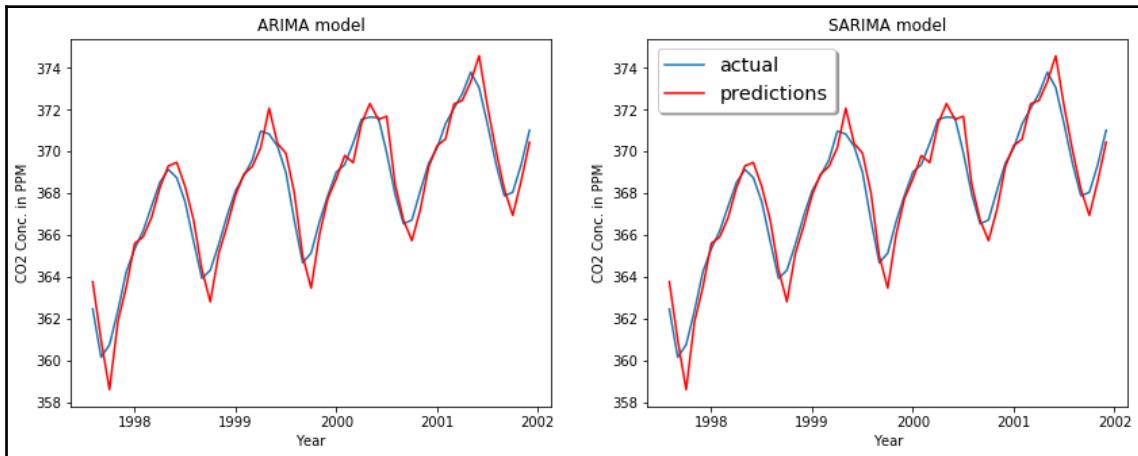
```

        output = model.forecast()
        predictions.append(output[0])
        history.append(t)
        error = mean_squared_error(test, predictions)
        print('Test MSE: %.3f' % error)
        #print(model.summary().tables[1])
        return predictions, error

    predictions_arima, error_arima = apply_model(train, test)
    predictions_sarima, error_sarima = apply_model(
        train, test, get_sarima_model
    )

```

We get an MSE in the test of 0.554 and 0.405 for ARIMA and SARIMA models, respectively. Let's see how the models fit graphically:



We could do a parameter exploration using the **Akaike information criterion (AIC)**, which expresses the quality of the model relative to the number of parameters in the model. The model object returned by the fit function in statsmodels includes the AIC, so we could do a grid search over a range of parameters, and then select the model that minimizes the AIC.

How it works...

Time series data is a collection of observations $x(t)$, where each data point is recorded at time t . In most cases, time is a discrete variable, that is, $0, \dots, N$. We are looking at forecasting, which is the task of predicting future values based on the previous observations in the time series.

In order to explain the models that we've used, ARIMA and SARIMA, we'll have to go step by step, and explain each in turn:

- **Autoregression (AR)**
- **Moving Average (MA)**
- **Autoregressive Moving Average (ARMA)**
- **Autoregressive Integrated Moving Average (ARIMA)** and
- **Seasonal Autoregressive Integrated Moving Average (SARIMA)**

ARIMA and SARIMA are based on the ARMA model, which is an **autoregressive moving average** model. Let's briefly go through some of the basics.

ARMA is a linear model, defined in two parts. First, the autoregressive linear model:

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

Here, $\varphi_1, \dots, \varphi_p$ are parameters and c is a constant, ε_t is white noise, and p is the order of the model (or the window size of the linear model). The second part of ARMA is the moving average, and this is again a linear regression, but of non-observable, lagged error terms, defined as follows:

$$X_t = \mu + \varepsilon_t + \theta_1 \varepsilon_{t-1} + \dots + \theta_q \varepsilon_{t-q} = \mu + \varepsilon_t + \sum_{i=1}^q \theta_i \varepsilon_{t-i},$$

Here, q is the order of the moving average, $\theta_1, \dots, \theta_q$ are the parameters, and μ the expectation or the mean of the time series X . The ARMA(p, q) model is then the composite of both of these models, AR(p) and MA(q):

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

The fitting procedure is a bit involved, particularly because of the MA part. You can read up on the Box-Jenkins method on Wikipedia if you are interested: https://en.wikipedia.org/wiki/Box-Jenkins_method

There are a few limitations to note, however. The time series has to be the following:

- Stationary: Basically, mean and covariance across observations have to be constant over time.
- Nonperiodic: Although bigger values for p and q could be used to model seasonality, it is not part of the model.
- Linear: Each value for X_t can be modeled as a linear combination of previous values and error terms.

There are different extensions of ARMA to address the first two limitations, and that's where ARIMA and SARIMA come in.

ARIMA (p, d, q) stands for **autoregressive integrated moving average**. It comes with three parameters:

- **p**: The number of autoregressive terms (autoregression)
- **d**: The number of non-seasonal differences needed for stationarity (integration)
- **q**: The number of lagged forecast errors (moving average)

The *integration* refers to differencing. In order to stabilize the mean, we can take the difference between consecutive observations. This can also remove a trend or eliminate seasonality. It can be written as follows:

$$x'_t = x_t - x_{t-1}.$$

This can be repeated several times, and this is what the parameter d describes that ARIMA comes with. Please note that ARIMA can handle drifts and non-stationary time series. However, it is still unable to handle seasonality.

SARIMA stands for seasonal ARIMA, and is an extension of ARIMA in that it also takes into account the seasonality of the data.

$SARIMA(p, d, q)(P, D, Q)m$ contains the non-seasonal parameters of ARIMA and additional seasonal parameters. Uppercase letters P, D, and Q annotate the seasonal moving average and autoregressive components, where m is the number of periods in each season. Often this is the number of periods in a year; for example $m=4$ would stand for a quarterly seasonal effect, meaning that D stands for seasonal differencing between observations X_t and X_{t-m} , and P and Q stand for linear models with backshifts of m .

In Python, the statsmodels library provides a method to perform the decomposition of the signal based on the seasonality of the data.

There's more...

Prophet is a library provided by Facebook for forecasting time series data. It works on an additive model and fits non-linear models. The library works best when the data has strong seasonal effects and has enough historic trends available.

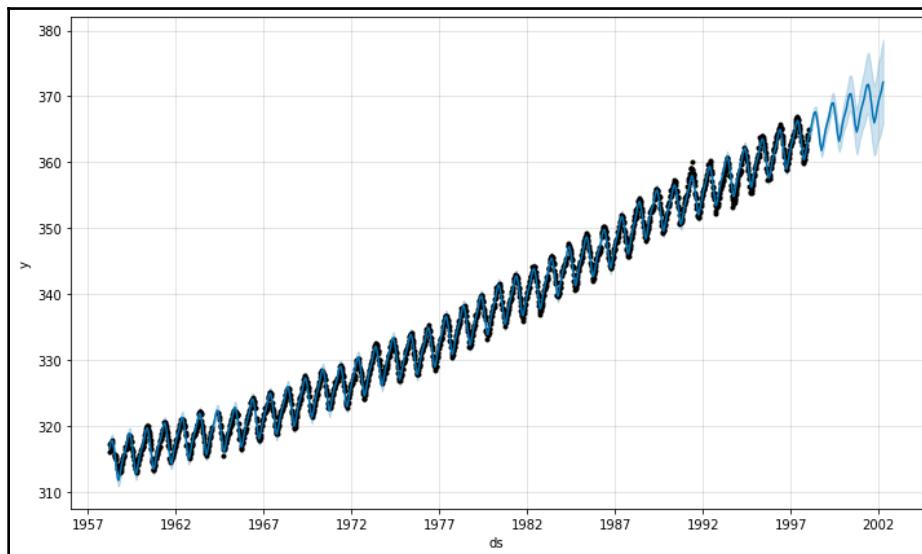
Let's see how we can use it:

```
from fbprophet import Prophet

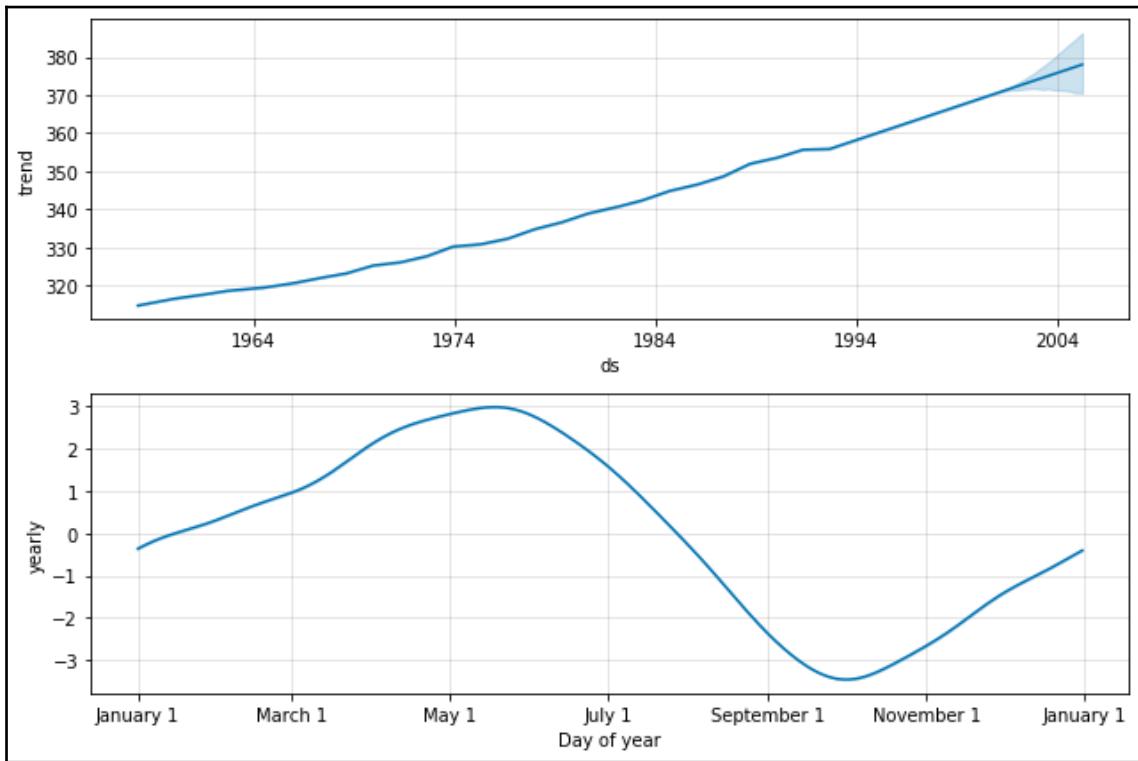
train_df = df_CO2_fb['1958':'1997']
test_df = df_CO2_fb['1998':'2001']
train_df = train_df.reset_index()
test_df = test_df.reset_index()
Co2_model = Prophet(interval_width=0.95)

Co2_model.fit(train_df)
train_forecast = Co2_model.predict(train_df)
test_forecast = Co2_model.predict(test_df)
fut = Co2_model.make_future_dataframe(periods=12, freq='M')
forecast_df = Co2_model.predict(fut)
Co2_model.plot(forecast_df)
```

Here are our model forecasts:



We get a similar decomposition as before with the ARIMA/SARIMA models, namely, the trend and the seasonal components:



The yearly variation nicely shows the rise and fall of the CO₂ concentration according to the seasons. The trend clearly goes up over time, which could be worrisome if you think about global warming.

See also

We've used the following libraries in this recipe:

- Statsmodels: <http://statsmodels.sourceforge.net/stable/>
- Prophet: <https://facebook.github.io/prophet/>

There are many more interesting libraries relating to time series, including the following:

- Time series modeling using state space models in statsmodels:
<https://www.statsmodels.org/dev/statespace.html>
- GluonTS: Probabilistic Time Series Models in MXNet (Python): <https://gluon-ts.mxnet.io/>
- SkTime: A unified interface for time series modeling: <https://github.com/alanturing-institute/sktime>

3

Patterns, Outliers, and Recommendations

In order to gain knowledge from data, it's important to understand the structure behind a dataset. The way we represent a dataset can make it more intuitive to work with in a certain way, and consequently, easier to draw insights from it. The law of the instrument states that when holding a hammer, everything seems like a nail (based on Andrew Maslow's *The Psychology of Science*, 1966) and is about the tendency to adapt jobs to the available tools. There is no silver bullet, however, as all methods have their drawbacks given the problem at hand. It's therefore important to know the basic methods in the arsenal of available tools in order to recognize the situations where we should use a hammer as supposed to a screwdriver.

In this chapter, we'll look at different ways of representing data, be it visualizing customer groups for marketing purposes and finding unusual patterns, or projecting data to emphasize differences, recommending products to customers based on their own previous choices, along with those of other customers, and identifying fraudsters by their similarities.

Specifically, we'll present the following recipes:

- Clustering market segments
- Discovering anomalies
- Representing for similarity search
- Recommending products
- Spotting fraudster communities

Clustering market segments

In this recipe, we'll apply clustering methods in order to find groups of customers for marketing purposes. We'll look at the German Credit Risk dataset, and we'll try to identify different segments of customers; ideally, we'd want to find the groups that are most profitable and different at the same time, so we can target them with advertising.

Getting ready

For this recipe, we'll be using a dataset of credit risk, usually referred to in full as the German Credit Risk dataset. Each row describes a person who took a loan, gives us a few attributes about the person, and tells us whether the person paid the loan back (that is, whether the credit was a good or bad risk).

We'll need to download and load up the German credit data as follows:

```
import pandas as pd
!wget
http://archive.ics.uci.edu/ml/machine-learning-databases/statlog/german/german.data
names = ['existingchecking', 'duration', 'credithistory',
         'purpose', 'creditamount', 'savings',
         'employmentsince', 'installmentrate',
         'statussex', 'otherdebtors', 'residencesince',
         'property', 'age', 'otherinstallmentplans',
         'housing', 'existingcredits', 'job',
         'peopleliable', 'telephone', 'foreignworker',
         'classification']

customers = pd.read_csv('german.data', names=names, delimiter=' ')
```

For visualizations, we'll use the dython library. The dython library works directly on categorical and numeric variables, and makes adjustments for numeric-categorical or categorical-categorical comparisons. Please see the documentation for details, at <http://shakedzy.xyz/dython/>. Let's install the library as follows:

```
!pip install dython
```

We can now play with the German credit dataset, visualize it with dython, and see how the people represented inside can be clustered together in different groups.

How to do it...

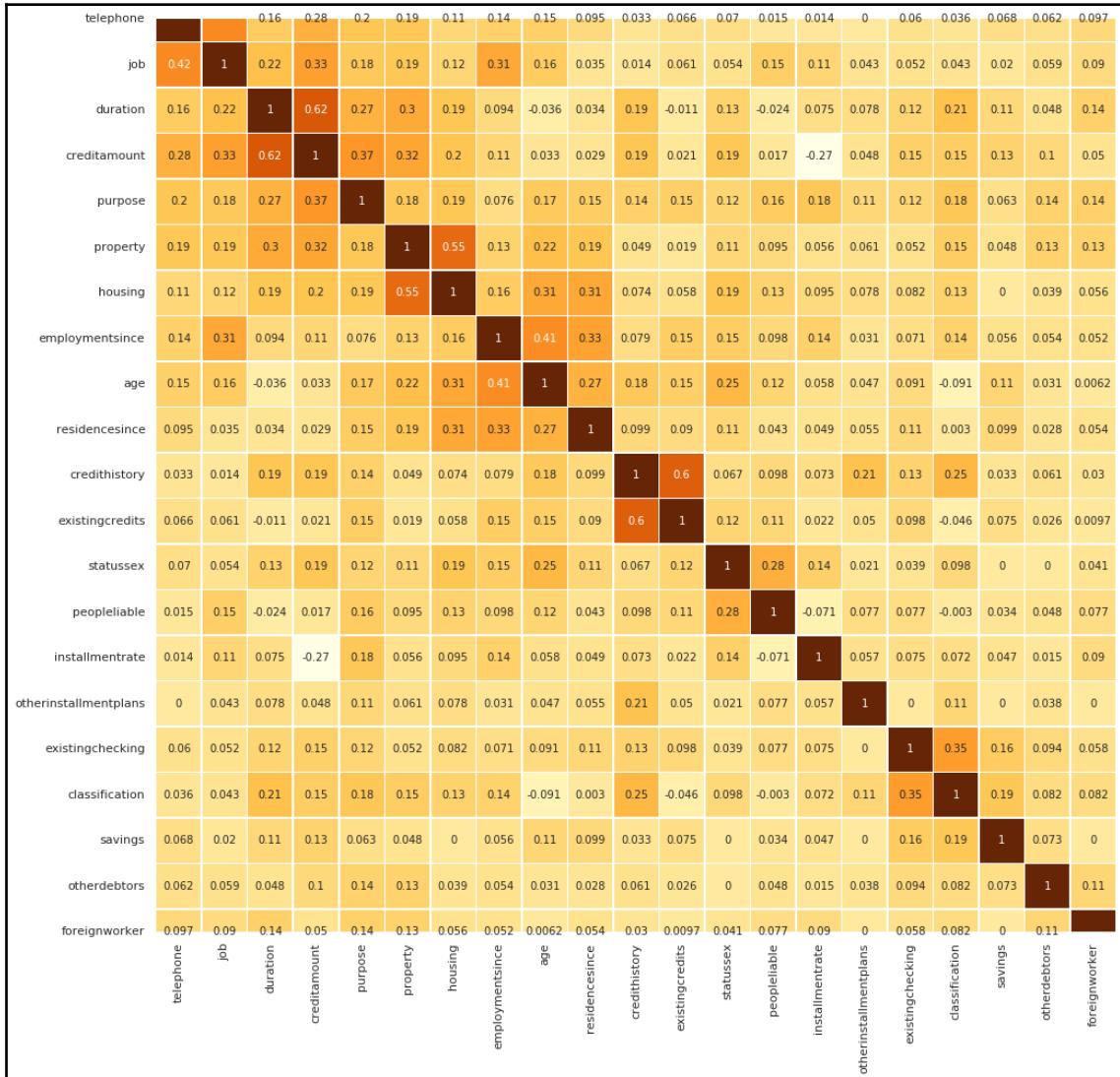
We'll first visualize the dataset, do some preprocessing, and apply a clustering algorithm. We'll try to make sense out of the clusters, and – with the new insights – cluster again.

We'll start by visualizing the features:

1. **Visualizing correlations:** In this recipe, we'll use the dython library. We can calculate the correlations with dython's associations function, which calls categorical, numerical (Pearson correlation), and mixed categorical-numerical correlation functions depending on the variable types:

```
from dython.nominal import associations  
  
associations(customers, clustering=True, figsize=(16, 16),  
cmap='YlOrBr');
```

This call not only calculates correlations, but also cleans up the correlation matrix by clustering variables together that are correlated. The data is visualized as shown in the following screenshot:



We can't really see clear cluster demarcations; however, there seem to be a few groups if you look along the diagonal.

Also, a few variables such as **telephone** and **job** stand out a bit from the rest. In the notebook on GitHub, we've tried dimensionality reduction methods to see if this would help our clustering. However, dimensionality reduction didn't work that well, while clustering directly worked better: <https://github.com/PacktPublishing/Artificial-Intelligence-with-Python-Cookbook/blob/master/chapter03/Clustering%20market%20segments.ipynb>.

As the first step for clustering, we'll convert some variables into dummy variables; this means we will do a one-hot-encoding of the categorical variables.

2. **Preprocessing variables:** We one-hot encode (also called *dummy-transform*) the categorical variables as follows:

```
catvars = ['existingchecking', 'credithistory', 'purpose',
'savings', 'employmentsince',
'statussex', 'otherdebtors', 'property', 'otherinstallmentplans',
'housing', 'job',
'telephone', 'foreignworker']
numvars = ['creditamount', 'duration', 'installmentrate',
'residencesince', 'age',
'existingcredits', 'peopleliable', 'classification']

dummyvars = pd.get_dummies(customers[catvars])
transactions = pd.concat([customers[numvars], dummyvars], axis=1)
```

Unfortunately, when we visualize this dataset to highlight customer differences, the result is not appealing. You can see the notebook online for some attempts at this.

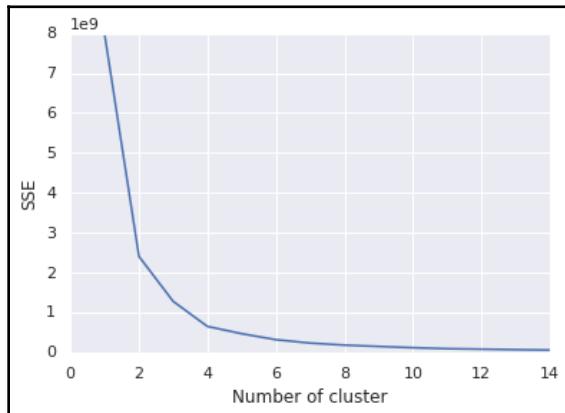
3. **First attempt at clustering:** A typical method for clustering is `kmeans`. Let's try it out:

```
from sklearn.cluster import KMeans
from matplotlib import pyplot as plt

sse = {}
for k in range(1, 15):
    kmeans = KMeans(n_clusters=k).fit(transactions)
    sse[k] = kmeans.inertia_
plt.figure()
plt.plot(list(sse.keys()), list(sse.values()))
plt.xlabel("Number of cluster")
plt.ylabel("SSE")
plt.show()
```

The inertia is the sum of distances to the closest cluster center over all the data points. A visual criterion for choosing the best number of clusters (the hyperparameter k in the k -means clustering algorithm) is called the **elbow criterion**.

Let's visualize the inertia over a number of clusters:



The basic idea of the elbow criterion is to choose the number of clusters where the error or inertia flattens off. According to the elbow criterion, we could be taking 4 clusters. Let's get the clusters again:

```
kmeans = KMeans(n_clusters=4).fit(transactions)
y = kmeans.labels_
```

4. Summarizing clusters: Now we can summarize the clusters:

```
clusters = transactions.join(
    pd.DataFrame(data=y, columns=['cluster'])
).groupby(by='cluster').agg(
    age_mean=pd.NamedAgg(column='age', aggfunc='mean'),
    age_std=pd.NamedAgg(column='age', aggfunc='std'),
    creditamount=pd.NamedAgg(column='creditamount',
    aggfunc='mean'),
    duration=pd.NamedAgg(column='duration', aggfunc='mean'),
    count=pd.NamedAgg(column='age', aggfunc='count'),
    class_mean=pd.NamedAgg(column='classification',
    aggfunc='mean'),
    class_std=pd.NamedAgg(column='classification', aggfunc='std'),
).sort_values(by='class_mean')
clusters
```

Here's the summary table for the clusters. We've included marketing characteristics, such as age, and others that give us an idea about how much money the customers make us. We are showing standard deviations over some of these in order to understand how consistent the groups are:

cluster	age_mean	age_std	creditamount	duration	count	class_mean	class_std
0	35.616943	11.683110	1469.364641	15.060773	543	1.272560	0.445687
2	34.673684	10.752440	3583.589474	23.505263	285	1.280702	0.450132
3	36.800000	11.230081	7127.523077	33.346154	130	1.361538	0.482305
1	36.666667	11.802577	12511.714286	40.261905	42	1.595238	0.496796

We see in this little excerpt that differences are largely due to differences in credit amount. This brings us back to where we started out, namely that we largely get out of the clustering what we put in. There's no trivial way of resolving this problem, but we can select the variables we want to focus on in our clusters.

5. **New attempt at clustering:** We can go back to the drawing board, simplify our aims, and start again with the question of what we actually want to find: groups of customers that fulfill two characteristics:
 - The clusters should distinguish customers by who makes us money: this leads us to variables such as credit amount, duration of their loan, and whether they paid it back.
 - The clusters should highlight different characteristics with respect to marketing, such as age, gender, or some other characteristic.

With this in mind, we'll make a new attempt:

```
from scipy.spatial.distance import pdist, squareform
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import AgglomerativeClustering

distances = squareform(pdist(
    StandardScaler().fit_transform(
        transactions[['classification', 'creditamount',
        'duration']])))
clustering = AgglomerativeClustering(
    n_clusters=5, affinity='precomputed', linkage='average'
).fit(distances)
y = clustering.labels_
```

We can now produce the overview table again in order to view the cluster stats:

```
clusters = transactions.join(
    pd.DataFrame(data=y, columns=['cluster'])
).groupby(by='cluster').agg(
    age_mean=pd.NamedAgg(column='age', aggfunc='mean'),
    age_std=pd.NamedAgg(column='age', aggfunc='std'),
    creditamount=pd.NamedAgg(column='creditamount',
    aggfunc='mean'),
    duration=pd.NamedAgg(column='duration', aggfunc='mean'),
    count=pd.NamedAgg(column='age', aggfunc='count'),
    class_mean=pd.NamedAgg(column='classification',
    aggfunc='mean'),
    class_std=pd.NamedAgg(column='classification', aggfunc='std'),
).sort_values(by='class_mean')
clusters
```

And here comes the new summary:

	age_mean	age_std	creditamount	duration	count	class_mean	class_std
cluster							
4	35.920000	7.348025	8289.740000	42.640000	50	1.000000	0.000000
2	35.548165	11.489624	2473.405963	17.779817	872	1.259174	0.438433
0	38.200000	16.457352	15271.600000	51.300000	10	1.600000	0.516398
1	34.530303	10.989813	7845.363636	41.545455	66	2.000000	0.000000
3	45.500000	31.819805	14725.500000	6.000000	2	2.000000	0.000000

I would argue this is more useful than the previous clustering, because it clearly shows us which customers can make us money, and highlights other differences between them that are relevant to marketing.

How it works...

Clustering is a very common visualization technique in business intelligence. In marketing, you'll target people differently, say teens, versus pensioners, and some groups are more valuable than others. Often, as a first step, dimensionality is reduced using a dimensionality reduction method or by feature selection, then groups are separated by applying a clustering algorithm. For example, you could first apply **Principal Component Analysis (PCA)** to reduce dimensionality (the number of features), and then k -means for finding groups of data points.

Since visualizations are difficult to judge objectively, in the previous section, what we did was to take a step back and look at the actual purpose, the business goal, that we want to achieve. We took the following steps to achieve this goal:

- We concentrated on the variables that are important for our goal.
- We created a function that would help us determine the quality of the clusters.

From this premise, we then tried different methods and evaluated them against our business goal.

If you've paid attention when looking at the recipe, you might have noticed that we don't standardize our output (z-scores). In standardization with the z-score, a raw score x is converted into a standard score by subtracting the mean and dividing by the standard deviation, so every standardized variable has a mean of 0 and a standard deviation of 1:

$$z = \frac{x - \mu}{\sigma}$$

We don't apply standardization because variables that have been dummy-transformed would have higher importance proportional to their number of factors. To put it simply, z-scores mean that every variable would have the same importance. One-hot encoding gives us a separate variable for each value that it can take. If we calculate and use z-scores after dummy-transforming, a variable that was converted to many new (dummy) variables, because it has many values, would be less important than another variable that has fewer values and consequently fewer dummy columns. This situation is something we want to avoid, so we don't apply z-scores.

The important thing to take away, however, is that we have to focus on differences that we can understand and describe. Otherwise, we might end up with clusters that are of limited use.

In the next section, we'll go more into detail with the k -means algorithm.

There's more...

PCA was proposed in 1901 (by Karl Pearson, in *On Lines and Planes of Closest Fit to Systems of Points in Space*) and *k*-means in 1967 (by James MacQueen, in *Some Methods for Classification and Analysis of Multivariate Observations*). While both methods had their place when data and computing resources were hard to come by, today many alternatives exist that can work with more complex relationships between data points and features. On a personal note, as the authors of this book, we often find it frustrating to see methods that rely on normality or a very limited kind of relationship between variables, such as classic methods like PCA or *K*-means, especially when there are so many better methods.

Both PCA and *k*-means have serious shortcomings that affect their usefulness in practice. Since PCA operates over the correlation matrix, it can only find linear correlations between data points. This means that if variables were related, but not linearly (as you would see in a scatter plot), then PCA would fail. Furthermore, PCA is based on mean and variance, which are parameters for Gaussian distribution. *K*-means, being a centroid-based clustering algorithm, can only find spherical groups in Euclidean space – that is, it fails to uncover any more complicated structures. More information on this can be found at <https://developers.google.com/machine-learning/clustering/algorithm/advantages-disadvantages>.

Other robust, nonlinear methods are available, for example, affinity propagation, fuzzy *c*-means, agglomerative clustering, and others. However, it's important to remember that, although these methods separate data points into groups, the following statements are also true:

- This is done according to a heuristic.
- It's based on the differences apparent in the dataset and based on the distance metric applied.
- The purpose of clustering is to visualize and simplify the output for human observers.

Let's look at the *k*-means algorithm in more detail. It's actually really simple and can be written down from scratch in numpy or jax. This implementation is based on the one in NapkinML (<https://github.com/eriklindernoren/NapkinML>):

```
import jax.numpy as jnp
import numpy as np
from jax import jit, vmap
from sklearn.base import ClassifierMixin
import jax
import random
from scipy.stats import hmean
```

```
class KMeans(ClassifierMixin):
    def __init__(self, k, n_iter=100):
        self.k = k
        self.n_iter = n_iter
        self.euclidean = jit(vmap(
            lambda x, y: jnp.linalg.norm(
                x - y, ord=2, axis=-1, keepdims=False
            ), in_axes=(0, None), out_axes=0
        ))
    def adjust_centers(self, X):
        jnp.row_stack([X[self.clusters == c].mean(axis=0)
                      for c in self.clusters
                     ])
    def initialize_centers(self):
        '''roughly the kmeans++ initialization
        '''
        key = jax.random.PRNGKey(0)
        # jax doesn't have uniform_multivariate
        self.centers = jax.random.multivariate_normal(
            key, jnp.mean(X, axis=0), jnp.cov(X, rowvar=False), shape=(1, )
        )
        for c in range(1, self.k):
            weights = self.euclidean(X, self.centers)
            if c>1:
                weights = hmean(weights ,axis=-1)
                print(weights.shape)

            new_center = jnp.array(
                random.choices(X, weights=weights, k=1) [0],
                ndmin=2
            )
            self.centers = jnp.row_stack(
                (self.centers, new_center)
            )
        print(self.centers.shape)

    def fit(self, X, y=None):
        self.initialize_centers()
        for iter in range(self.n_iter):
            dists = self.euclidean(X, self.centers)
            self.clusters = jnp.argmin(dists, axis=-1)
            self.adjust_centers(X)
        return self.clusters
```

The main logic – as should be expected – is in the `fit()` method. It comes in three steps that are iterated as follows:

1. Calculate the distances between each point and the centers of the clusters.
2. Each point gets assigned to the cluster of its closest cluster center.
3. The cluster centers are recalculated as the arithmetic mean.

It's surprising that such a simple idea can result in something that looks meaningful to human observers. Here's an example of it being used. Let's try it out with the Iris dataset that we already know from the *Classifying in scikit-learn, Keras, and PyTorch* recipe in Chapter 1, *Getting Started with Artificial Intelligence in Python*:

```
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
X, y = load_iris(return_X_y=True)

kmeans = KMeans(k=3)
kmeans.fit(X)
```

We end up with clusters that we can visualize or inspect, similar to before.

See also

In order to get an overview of different clustering methods, please refer to a survey or review paper. Saxena et al. cover most of the important terminology in their article, *Review of Clustering Techniques and Developments* (2017).

We would recommend looking at the following methods relevant to clustering and dimensionality reduction (we link to implementations):

- **Affinity propagation** (<https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AffinityPropagation.html>): A clustering method that finds a number of clusters as part of its heuristics
- **Fuzzy c-means** (https://pythonhosted.org/scikit-fuzzy/auto_examples/plot_cmeans.html): A fuzzy version of the k-means algorithm
- **Local linear embedding** (<https://scikit-learn.org/stable/modules/generated/sklearn.manifold.LocallyLinearEmbedding.html>): A lower-dimensional embedding, where local similarities are maintained

- T-distributed stochastic neighbor embedding (<https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html>): A nonlinear dimensionality reduction method suited for visualization in two- or three-dimensional space.
- Mixture-modeling (https://mixem.readthedocs.io/en/latest/examples/old_faithful.html)

The idea of taking advantage of a pre-trained random forest in order to provide a custom-tailored kernel is discussed in *The Random Forest Kernel and other kernels for big data from random partitions* (2014) by Alex Davies and Zoubin Ghahramani, available at <https://arxiv.org/abs/1402.4293>.

Discovering anomalies

An anomaly is anything that deviates from the expected or normal outcomes. Detecting anomalies can be important in **Industrial Process Monitoring (IPM)**, where data-driven fault detection and diagnosis can help achieve higher levels of safety, efficiency, and quality.

In this recipe, we'll look at methods for outlier detection. We'll go through an example of outlier detection in a time series with **Python Outlier Detection (pyOD)**, a toolbox for outlier detection that implements many state-of-the-art methods and visualizations. PyOD's documentation can be found at <https://pyod.readthedocs.io/en/latest/>.

We'll apply an autoencoder for a similarity-based approach, and then an online learning approach suitable for finding events in streams of data.

Getting ready

This recipe will focus on finding outliers. We'll demonstrate how to do this with the pyOD library including an autoencoder approach. We'll also outline the upsides and downsides to the different methods.

The streams of data are time series of **key performance indicators (KPIs)** of website performance. This dataset is provided in the DONUT outlier detector repository, available at <https://github.com/haowen-xu/donut>.

Let's download and load it as follows:

```
import pandas as pd

!wget
https://raw.githubusercontent.com/haowen-xu/donut/master/sample_data/cpu4.csv
cpu_data = pd.read_csv('cpu4.csv')
```

We'll use methods from pyOD, so let's install that as well:

```
!pip install pyOD
```

Please note that some pyOD methods have dependencies such as TensorFlow and Keras, so you might have to make sure that these are also installed. If you get a message reading No module named Keras you can install Keras separately as follows:

```
!pip install keras
```

Please note that it's usually better to use the Keras version that ships with TensorFlow.

Let's have a look at our dataset, and then apply different outlier detection methods.

How to do it...

We'll cover different steps and methods in this section. They are as follows:

1. Visualizing
2. Benchmarking
3. Running an isolation forest
4. Running an autoencoder

Let's start by exploring and visualizing our dataset:

1. We can visualize our dataset over time as a time series:

Let's have a look at our dataset with the following command:

```
cpu_data.head()
```

This is what it looks like:

	timestamp	value	label
0	1469376000	0.847300	0
1	1469376300	-0.036137	0
2	1469376600	0.074292	0
3	1469376900	0.074292	0
4	1469377200	-0.036137	0

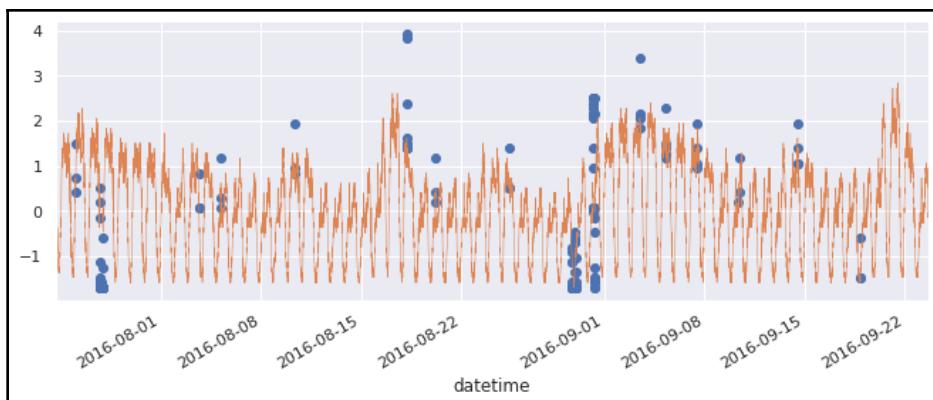
This time series of KPIs is geared toward monitoring the operation and maintenance of web services. They come with a label that indicates an abnormality – in other words, an outlier – if a problem has occurred with the service:

```
from datetime import datetime
import seaborn as sns

cpu_data['datetime'] = cpu_data.timestamp.astype(int).apply(
    datetime.fromtimestamp
)
# Use seaborn style defaults and set the default figure size
sns.set(rc={'figure.figsize':(11, 4)})

time_data = cpu_data.set_index('datetime')
time_data.loc[time_data['label'] == 1.0,
    'value'].plot(linewidth=0.5, marker='o', linestyle='')
time_data.loc[time_data['label'] == 0.0,
    'value'].plot(linewidth=0.5)
```

This is the resulting plot, where the dots represent outliers:



Alternatively, we can see where outliers are located in the spectrum of the KPIs, and how clearly they are distinguishable from normal data, with the following code:

```
import numpy as np
from matplotlib import pyplot as plt

markers = ['r--', 'b-^']

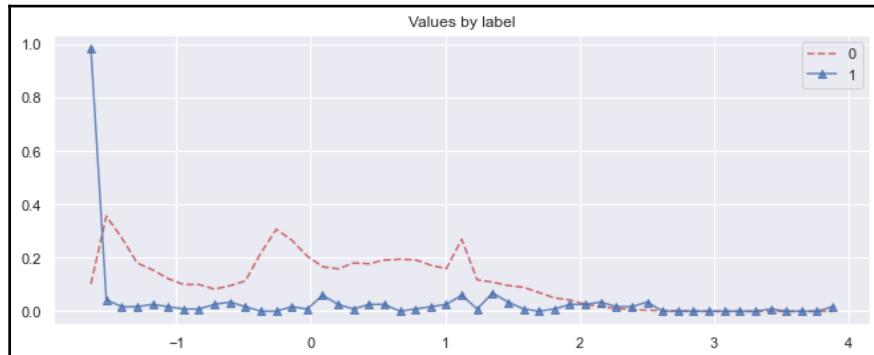
def hist2d(X, by_col, n_bins=10, title=None):
    bins = np.linspace(X.min(), X.max(), n_bins)
    vals = np.unique(by_col)
    for marker, val in zip(markers, vals):
        n, edges = np.histogram(X[by_col==val], bins=bins)
        n = n / np.linalg.norm(n)
        bin_centers = 0.5 * (edges[1:] + edges[:-1])
        plt.plot(bin_centers, n, marker, alpha=0.8, label=val)

    plt.legend(loc='upper right')
    if title is not None:
        plt.title(title)
    plt.show()

hist2d(cpu_data.value, cpu_data.label, n_bins=50, title='Values by
label')
```

With the preceding code, we plot two histograms against each other using line plots. Alternatively, we could be using `hist()` with opacity.

The following plot is the outlier distribution density, where the values of the time series are on the x axis, and the two lines show what's recognized as normal and what's recognized as an outlier, respectively – 0 indicates normal data points, and 1 indicates outliers:



We'll be using the same visualization for all subsequent methods so we can compare them graphically.

Outliers (shown with the dotted line) are hardly distinguishable from normal data points (the squares), so we won't be expecting perfect performance.

2. We'll now implement benchmarking.

Before we go on and test methods for outlier detection, let's set down a process for comparing them, so we'll have a benchmark of the relative performances of the tested methods.

We split our data into training and test sets as usual:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    cpu_data[['value']].values, cpu_data.label.values
)
```

Now let's write a testing function that we can use with different outlier detection methods:

```
from pyod.utils.data import evaluate_print
from pyod.models.knn import KNN

def test_outlier_detector(X_train, y_train,
                         X_test, y_test, only_neg=True,
                         basemethod=KNN()):
    clf = basemethod
    if only_neg:
        clf.fit(X_train[y_train==0.0],
        np.zeros(shape=((y_train==0.0).sum(), 1)))
    else:
        clf.fit(X_train, y_train) # most algorithms ignore y

    y_train_pred = clf.predict(X_train) # labels_
    y_train_scores = clf.decision_scores_

    y_test_pred = clf.predict(X_test)
    y_test_scores = clf.decision_function(X_test)

    print("\nOn Test Data:")
    evaluate_print(type(clf).__name__, y_test, y_test_scores)
    hist2d(X_test, y_test_pred, title='Predicted values by label')
```

This function tests an outlier detection method on the dataset. It trains a model, gets performance metrics from the model, and plots a visualization.

It takes these parameters:

- `X_train`: Training features
- `y_train`: Training labels
- `X_test`: Test features
- `y_test`: Test labels
- `only_neg`: Whether to use only normal points (that is, not outliers) for training
- `basemethod`: The model to test

We can choose to only train on normal points (all points excluding outliers) in order to learn the distribution or general characteristics of these points, and the outlier detection method can then decide whether the new points do or don't fit these characteristics.

Now that this is done, let's test two methods for outlier detection: the isolation forest and an autoencoder.

The isolation forest comes first.

3. Here we'll look at the isolation forest.

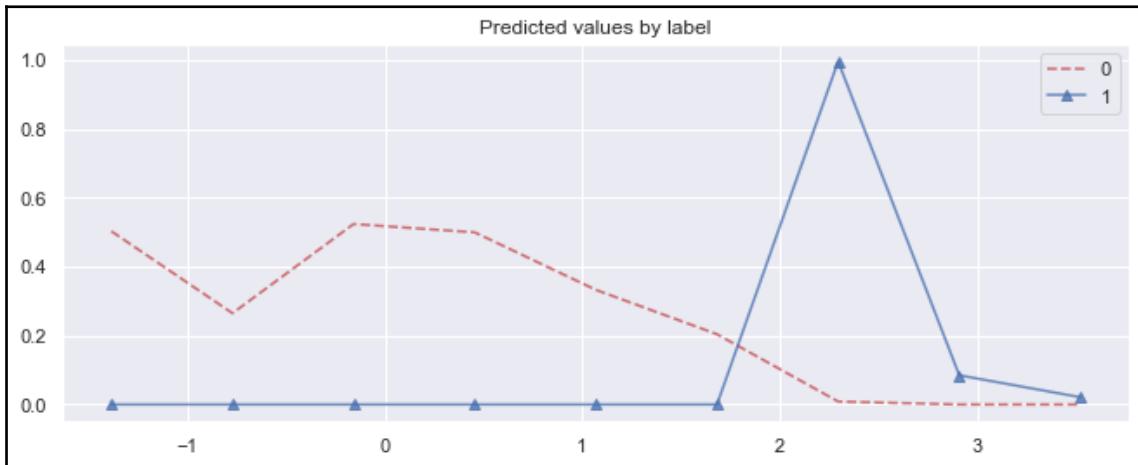
We run the benchmarking method and hand over an isolation forest detection method:

```
from pyod.models.iforest import IForest

test_outlier_detector(
    X_train, y_train, X_test, y_test,
    only_neg=True, basemethod=IForest(contamination=0.01),
)
#On Test Data:
#IForest ROC:0.867, precision @ rank n:0.1
```

The **Receiver Operating Characteristic (ROC)** performance of the isolation forest predictions against the test data is about 0.86, so it does reasonably well.

We can see from the following graph, however, that there are no **1s** (predicted outliers) in the lower range of the KPI spectrum. The model misses out on outliers in the lower range:



It only recognizes points as outliers that have higher values (≥ 1.5).

4. Next, let's try running an autoencoder:

```
from pyod.models.auto_encoder import AutoEncoder

test_outlier_detector(
    X_train, y_train, X_test, y_test,
    only_neg=False,
    basemethod=AutoEncoder(hidden_neurons=[1], epochs=10)
)
```

We can see the Keras network structure and the output from the test function:

Layer (type)	Output Shape	Param #
<hr/>		
dense_39 (Dense)	(None, 1)	2
dropout_30 (Dropout)	(None, 1)	0
dense_40 (Dense)	(None, 1)	2
dropout_31 (Dropout)	(None, 1)	0
dense_41 (Dense)	(None, 1)	2

```

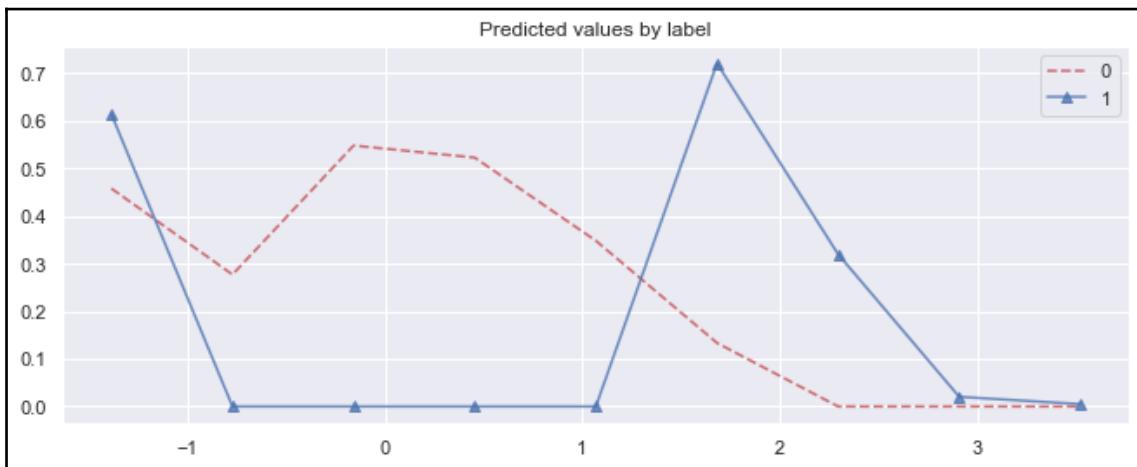
dropout_32 (Dropout)           (None, 1)      0
dense_42 (Dense)              (None, 1)      2
=====
Total params: 8
Trainable params: 8
Non-trainable params: 0

...
On Test Data:
AutoEncoder ROC:0.8174, precision @ rank n:0.1

```

The performance of the autoencoder is very similar to the isolation forest; however, the autoencoder finds outliers both in the lower and upper ranges of the KPI spectrum.

Furthermore, we don't get an appreciable difference when providing either only normal data or both normal data and outliers. We can see how the autoencoder works in the following graph:



This doesn't look too bad, actually – values in the mid-range are classified as normal, while values on the outside of the spectrum are classified as outliers.

Please remember that these methods are unsupervised; of course, we could get better results if we used a supervised method. As a practical consideration, if we use supervised methods with our own datasets, this would require us to do additional work by annotating anomalies, which we don't have to do with unsupervised methods.

How it works...

Outliers are extreme values that deviate from other observations on the data. Outlier detection is important in many domains, including network security, finance, traffic, social media, machine learning, the monitoring of machine model performance, and surveillance. A host of algorithms have been proposed for outlier detection in these domains. The most prominent algorithms include **k-Nearest Neighbors (kNN)**, **Local Outlier Factors (LOF)**, and the isolation forest, and more recently, autoencoders, **Long Short-Term Memory (LSTM)**, and **Generative Adversarial Networks (GANs)**. We'll discover some of these methods in later recipes. In this recipe, we've used kNN, an autoencoder, and the isolation forest algorithm. Let's talk about these three methods briefly.

k-nearest neighbors

The KNN classifier is a non-parametric classifier introduced by Thomas Cover and Peter Hart (in *Nearest neighbor pattern classification*, 1967). The main idea is that a new point is likely to belong to the same class as its neighbors. The hyperparameter k is the number of neighbors to compare. There are weighted versions based on the relative distance of a new point from its neighbors.

Isolation forest

The idea of the isolation forest is relatively simple: create random decision trees (this means each leaf uses a randomly chosen feature and a randomly chosen split value) until only one point is left. The length of the path across the trees to get to a terminal node indicates whether a point is an outlier.



You can find out more details about the isolation forest in its original publication by Liu et al., *Isolation Forest*. ICDM 2008: 413–422: <https://cs.nju.edu.cn/zhouzh/zhouzh.files/publication/icdm08b.pdf>.

Autoencoder

An autoencoder is a neural network architecture, where we learn a representation based on a set of data. This is usually done by a smaller hidden layer (**bottleneck**) from which the original is to be restored. In that way, it's similar to many other dimensionality reduction methods.

An autoencoder consists of two parts: the encoder and the decoder. What we are really trying to learn is the transformation of the encoder, which gives us a code or the representation of the data that we look for.

More formally, we can define the encoder as the function $\phi : \mathcal{X} \rightarrow \mathcal{F}$, and the decoder as the function $\psi : \mathcal{F} \rightarrow \mathcal{X}$. We try to find ϕ and ψ so that the reconstruction error is minimized:

$$\phi, \psi = \arg \min_{\phi, \psi} \|X - (\psi \circ \phi)X\|^2$$

The autoencoder represents data in an intermediate network layer, and the closer they can be reconstructed based on the intermediate representation, the less of an outlier they are.

See also

Many implementations of outlier detection are publicly available for Python:

- As part of Numenta's Cortical Learning Algorithm: <http://nupic.docs.numenta.org/stable/guides/anomaly-detection.html#temporalanomaly-model>
- Banpei, for singular spectrum transformation: <https://github.com/tsurubee/banpei>
- Deep anomaly detection methods for time series: <https://github.com/KDD-OpenSource/DeepADoTS>
- Telemanom – LSTMs for multivariate time-series data: <https://github.com/khundman/telemanom>
- DONUT – a variational auto-encoder for seasonal KPIs: <https://github.com/haowen-xu/donut>

A fantastic resource for material about outlier detection is the PyOD author's dedicated repository, available at <https://github.com/yzhao062/anomaly-detection-resources>.

Representing for similarity search

In this recipe, we want to find a way to decide whether two strings are similar given a representation of those two strings. We'll try to improve the way strings are represented in order to make more meaningful comparisons between strings. But first, we'll get a baseline using more traditional string comparison algorithms.

We'll do the following: given a dataset of paired string matches, we'll try out different functions for measuring string similarity, then a bag-of-characters representation, and finally a **Siamese neural network** (also called a **twin neural network**) dimensionality reduction of the string representation. We'll set up a twin network approach for learning a latent similarity space of strings based on character n-gram frequencies.



A **Siamese neural network**, also sometimes called **twin neural network**, is named as such using the analogy of conjoined twins. It is a way to train a projection or a metric space. Two models are trained at the same time, and the output of the two models is compared. The training takes the comparison output rather than the models' outputs.

Getting ready

As always, we need to download or load a dataset and install the necessary dependencies.

We'll use a dataset of paired strings, where they are either matched or not based on whether they are similar:

```
!wget  
https://raw.githubusercontent.com/ofrendo/WebDataIntegration/7db877abadd2be  
94d5373f5f47c8cccd1d179bea6/data/goldstandard/forbes_freebase_goldstandard_t  
rain.csv
```

We can then read it in as follows:

```
import pandas as pd  
  
data = pd.read_csv(  
    'forbes_freebase_goldstandard_train.csv',  
    names=['string1', 'string2', 'matched'])
```

The dataset includes pairs of strings that either correspond to each other or don't correspond. It starts like this:

	string1	string2	matched
0	General Electric	General Electric	True
1	Wells Fargo	Wells Fargo	True
2	Bank of China	Industrial and Commercial Bank of China (Asia)	True
3	PetroChina	PetroChina	True
4	Apple	Apple Inc.	True

There's also a test dataset available from the same GitHub repo:

```
!wget  
https://raw.githubusercontent.com/ofrendo/WebDataIntegration/7db877abadd2be  
94d5373f5f47c8cccd1d179bea6/data/goldstandard/forbes_freebase_goldstandard_t  
est.csv
```

We can read it into a pandas DataFrame the same way as before:

```
test = pd.read_csv(  
    'forbes_freebase_goldstandard_test.csv',  
    names=['string1', 'string2', 'matched'])
```

Finally, we'll use a few libraries in this recipe that we can install like this:

```
!pip install python-Levenshtein annoy
```

The **Levenshtein distance** (also sometimes referred to as **edit distance**) measures the number of insertions, deletions, and substitutions that are necessary to transform one string into another string. It performs a search in order to come up with the shortest way to do this transformation. The library used here is a very fast implementation of this algorithm. You can find more about the `python-Levenshtein` library at <https://github.com/ztane/python-Levenshtein>.

The `annoy` library provides a highly optimized implementation of the nearest-neighbor search. Given a set of points and a distance, we can index all the points using a tree representation, and then for any point, we can traverse the tree to find similar data points. You can find out more about `annoy` at <https://github.com/spotify/annoy>.

Let's get to it.

How to do it...

As mentioned before, we'll first calculate the baseline using standard string comparison functions, then we'll use a bag-of-characters approach, and then we'll learn a projection using a Siamese neural network approach. You can find the corresponding notebook on the book's GitHub repo, at <https://github.com/PacktPublishing/Artificial-Intelligence-with-Python-Cookbook/blob/master/chapter03/Representing%20for%20similarity%20search.ipynb>.

Baseline – string comparison functions

Let's first implement a few standard string comparison functions.

We first need to make sure we clean up our strings:

```
def clean_string(string):
    return ''.join(map(lambda x: x.lower() if str.isalnum(x) else ' ', string)).strip()
```

We'll use this cleaning function in each of the string comparison functions that we'll see in the following code. We will use this function to remove special characters before any string distance calculation.

Now we can implement simple string comparison functions. Let's do the Levenshtein distance first:

```
import Levenshtein

def levenshtein_distance(s1_, s2_):
    s1, s2 = clean_string(s1_), clean_string(s2_)
    len_s1, len_s2 = len(s1), len(s2)
    return Levenshtein.distance(
        s1, s2
    ) / max([len_s1, len_s2])
```

Now let's do the Jaro-Winkler distance, which is the minimum number of single-character transpositions:

```
def jaro_winkler_distance(s1_, s2_):
    s1, s2 = clean_string(s1_), clean_string(s2_)
    return 1 - Levenshtein.jaro_winkler(s1, s2)
```

We'll also use the longest common substring between the compared pair. We can use SequenceMatcher for this, which is part of the Python standard library:

```
from difflib import SequenceMatcher

def common_substring_distance(s1_, s2_):
    s1, s2 = clean_string(s1_), clean_string(s2_)
    len_s1, len_s2 = len(s1), len(s2)
    match = SequenceMatcher(
        None, s1, s2
    ).find_longest_match(0, len_s1, 0, len_s2)
    len_s1, len_s2 = len(s1), len(s2)
    norm = max([len_s1, len_s2])
    return 1 - min([1, match.size / norm])
```

Now we can run over all string pairs and calculate the string distances based on each of the three methods. For each of the three algorithms, we can calculate the **area under the curve (AUC)** score to see how well it does at separating matching strings from non-matching strings:

```
import numpy as np
from sklearn.metrics import roc_auc_score

dists = np.zeros(shape=(len(data), 3))
for algo_i, algo in enumerate([
    levenstein_distance, jaro_winkler_distance, common_substring_distance]):
    for i, string_pair in data.iterrows():
        dists[i, algo_i] = algo(string_pair['string1'],
                               string_pair['string2'])
    print('AUC for {}: {}'.format(
        algo.__name__,
        roc_auc_score(data['matched'].astype(float), 1 - dists[:, algo_i])))
#AUC for levenstein_distance: 0.9508904955034385
#AUC for jaro_winkler_distance: 0.9470992770234525
#AUC for common_substring_distance: 0.9560042320578381
```

The AUC scores for all algorithms are around 95%, which seems good. All three distances perform quite well already. Let's try to beat this.

Bag-of-characters approach

We'll now implement a bag-of-characters method for string similarity.

A bag of characters means that we will create a histogram of characters, or in other words, we will count the occurrences of the characters in each word:

```
from sklearn.feature_extraction.text import CountVectorizer

# We clean the strings as before and we take ngrams.
ngram_featurizer = CountVectorizer(
    min_df=1,
    analyzer='char',
    ngram_range=(1, 1), # this is the range of ngrams that are to be
    extracted!
    preprocessor=clean_string
).fit(
    np.concatenate([
        data['string1'], data['string2']],
    axis=0
```

```
)  
)
```

We've set the range of `ngrams` to just 1, which means we want only single characters. This parameter can be interesting, however, if you want to include longer-range dependencies between characters, rather than just the character frequencies.

Let's see what performance we can get out of this:

```
string1cv = ngram_featurizer.transform(data['string1'])  
string2cv = ngram_featurizer.transform(data['string2'])  
  
def norm(string1cv):  
    return string1cv / string1cv.sum(axis=1)  
  
similarities = 1 - np.sum(np.abs(norm(string1cv) - norm(string2cv)),  
axis=1) / 2  
roc_auc_score(data['matched'].astype(float), similarities)  
#0.9298183741844471
```

As you can see in the AUC score of about 93%, this approach doesn't yet perform quite as well overall, although the performance is not completely bad. So let's try to tweak this.

Siamese neural network approach

Now we'll implement a Siamese network to learn a projection that represents the similarities (or differences) between strings.

The Siamese network approach may seem a little daunting if you are not familiar with it. We'll discuss it further in the *How it works...* section.

Let's start with the string featurization function:

```
from tensorflow.keras.models import Sequential, Model  
from tensorflow.keras.layers import Dense, Lambda, Input  
import tensorflow as tf  
from tensorflow.keras import backend as K  
  
def create_string_featurization_model(  
    feature_dimensionality, output_dim=50):  
    preprocessing_model = Sequential()  
    preprocessing_model.add(  
        Dense(output_dim, activation='linear',  
        input_dim=feature_dimensionality)  
)
```

```
preprocessing_model.summary()
return preprocessing_model
```

The `create_string_featurization_model` function returns a model for string featurization. The featurization model is a non-linear projection of the bag-of-characters output.

The function has the following parameters:

- `feature_dimensionality`: The number of features coming from the vectorizer (that is, the dimensionality of the bag-of-characters output)
- `output_dim`: The dimensions of the embedding/projection that we are trying to create

Next, we need to create the **conjoined twins** of the two models. For this, we need a comparison function. We take the normalized Euclidean distance. This is the Euclidean distance between the two L_2 -normalized projected vectors.

The L_2 norm of a vector x is defined as follows:

$$\sqrt{\sum_i x_i^2}$$

The L_2 normalization is dividing the vector x by its norm.

We can define the distance function as follows:

```
def euclidean_distance(vects):
    x, y = vects
    x = K.l2_normalize(x, axis=-1)
    y = K.l2_normalize(y, axis=-1)
    sum_square = K.sum(
        K.square(x - y),
        axis=1,
        keepdims=True
    )
    return K.sqrt(K.maximum(
        sum_square,
        K.epsilon()
    ))
```

Now the Siamese network can use the function by wrapping it as a Lambda layer. Let's define how to conjoin the twins or, in other words, how we can wrap it into a bigger model so we can train with pairs of strings and the label (that is, similar and dissimilar):

```
def create_siamese_model(preprocessing_models, #initial_bias =
                         input_shapes=(10,)):
    if not isinstance(preprocessing_models, (list, tuple)):
        raise ValueError('preprocessing models needs to be a list or tuple
of models')
    print('{} models to be trained against each
other'.format(len(preprocessing_models)))
    if not isinstance(input_shapes, list):
        input_shapes = [input_shapes] * len(preprocessing_models)
    inputs = []
    intermediate_layers = []
    for preprocessing_model, input_shape in zip(preprocessing_models,
                                                input_shapes):
        inputs.append(Input(shape=input_shape))
        intermediate_layers.append(preprocessing_model(inputs[-1]))

    layer_diffs = []
    for i in range(len(intermediate_layers)-1):
        layer_diffs.append(
            Lambda(euclidean_distance)([intermediate_layers[i],
                                         intermediate_layers[i+1]]))
    )
    siamese_model = Model(inputs=inputs, outputs=layer_diffs)
    siamese_model.summary()
    return siamese_model
```

This is a verbose way of saying: take the two networks, calculate the normalized Euclidean distance, and take the distance as the output.

Let's create the twin network and train:

```
def compile_model(model):
    model.compile(
        optimizer='rmsprop',
        loss='mse',
    )

feature_dims = len(ngram_featurizer.get_feature_names())
string_featurization_model =
create_string_featurization_model(feature_dims, output_dim=10)

siamese_model = create_siamese_model(
    preprocessing_models=[string_featurization_model,
    string_featurization_model],
```

```
    input_shapes=[(feature_dims,), (feature_dims,)],
)
compile_model(siamese_model)
siamese_model.fit(
    [string1cv, string2cv],
    1 - data['matched'].astype(float),
    epochs=1000
)
```

This creates a model with an output of 10 dimensions; given 41 dimensions from the n-gram featurizer, this means we have a total of 420 parameters ($41 * 10 + 10$).

As we've mentioned before, the output of our combined network is the Euclidean distance between the two outputs. This means we have to invert our target (matched) column in order to change the meaning from similar to distant, so that 1 corresponds to different and 0 to the same. We can do this easily by subtracting from 1.

We can now get the performance of this new projection:

```
from scipy.spatial.distance import euclidean

string_rep1 = string_featurization_model.predict(
    ngram_featurizer.transform(data['string1']))
)
string_rep2 = string_featurization_model.predict(
    ngram_featurizer.transform(data['string2']))
)
dists = np.zeros(shape=(len(data), 1))
for i, (v1, v2) in enumerate(zip(string_rep1, string_rep2)):
    dists[i] = euclidean(v1, v2)
roc_auc_score(data['matched'].astype(float), 1 - dists)
0.9802944806912361
```

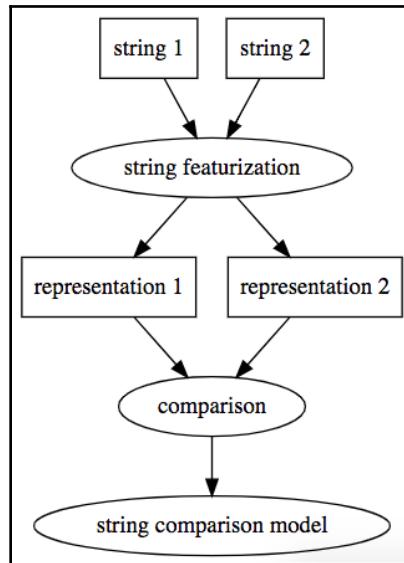
We've managed to beat the other methods. And that's before we've even tried to tune any hyperparameters. Our projection clearly works in highlighting differences between strings that are important for string similarity comparisons.

How it works...

The scikit-learn CountVectorizer counts the occurrences of features in strings. A common use case is to count words in sentences – this representation is called a **bag of words**, and in that case, the features would be words. In our case, we are interested in character-based features, so we just count how many times an **a** occurred, a **b** occurred, and so on. We could make this representation cleverer by representing tuples of successive characters such as **ab** or **ba**; however, that's beyond our scope here.

A Siamese network training is the situation where two (or more) neural networks are trained against each other by comparing the output of the networks given a pair (or tuple) of inputs and the knowledge of the difference between these inputs. Often the Siamese network consists of the same network (that is, the same weights). The comparison function between the two network outputs can be metrics such as the Euclidean distance or the cosine similarity. Since we know whether the two inputs are similar, and even how similar they are, we can train against this knowledge as the target.

The following diagram illustrates the information flow and the different building blocks that we'll be using:



Given the two strings that we want to compare, we'll use the same model to create features from each one, resulting in two representations. We can then compare these representations, and we hope that the comparison correlates with the outcome, so that if our comparison shows a big difference, the strings will be dissimilar, and if the comparison shows little difference, then the strings will be similar.

We can actually directly train this complete model, given a string comparison model and a dataset consisting of a pair of strings and a target. This training will tune the string featurization model so the representation will be more useful.

Recommending products

In this recipe, we'll be building a recommendation system. A recommender is an information-filtering system that predicts rankings or similarities by bringing content and social connections together.

We'll download a dataset of book ratings that have been collected from the Goodreads website, where users rank and review books that they've read. We'll build different recommender models, and we'll suggest new books based on known ratings.

Getting ready

To prepare for our recipe, we'll download the dataset and install the required dependencies.

Let's get the dataset and install the two libraries we'll use here – `spotlight` and `lightfm` are recommender libraries:

```
!pip install git+https://github.com/maciejkula/spotlight.git lightfm
```

Then we need to get the dataset of book ratings:

```
from spotlight.datasets.goodbooks import get_goodbooks_dataset
from spotlight.cross_validation import random_train_test_split

import numpy as np

interactions = get_goodbooks_dataset()
train, test = random_train_test_split(
    interactions, random_state=np.random.RandomState(42)
)
```

The dataset comes in the shape of an interaction object. According to spotlight's documentation, an interaction object can be defined as follows:

[It] contains (at a minimum) a pair of user-item interactions, but can also be enriched with ratings, timestamps, and interaction weights.

For implicit feedback scenarios, user IDs and item IDs should only be provided for user-item pairs where an interaction was observed. All pairs that are not provided are treated as missing observations, and often interpreted as (implicit) negative signals.

For explicit feedback scenarios, user IDs, item IDs, and ratings should be provided for all user-item-rating triplets that were observed in the dataset.

We have the following training and test datasets:

```
<Interactions dataset (53425 users x 10001 items x 4781183 interactions)>
<Interactions dataset (53425 users x 10001 items x 1195296 interactions)>
```

In order to know which books the item numbers refer to, we'll download the following CSV file:

```
!wget
https://raw.githubusercontent.com/zygmuntz/goodbooks-10k/master/books.csv
```

Next, we'll implement a function to get the book titles by id. This will be useful for showing our recommendations later:

```
import pandas as pd
books = pd.read_csv('books.csv', index_col=0)

def get_book_titles(book_ids):
    '''Get book titles by book ids
    '''
    if isinstance(book_ids, int):
        book_ids = [book_ids]
    titles = []
    for book_id in book_ids:
        titles.append(books.loc[book_id, 'title'])
    return titles

book_labels = get_book_titles(list(train.item_ids))
```

Now we can use this function as follows:

```
get_book_titles(1)
['The Hunger Games (The Hunger Games, #1)']
```

Now that we've got the dataset and the libraries installed, we can start our recipe.

How to do it...

We'll first use a matrix factorization model, then a deep learning model. You can find more examples in the Jupyter notebook available at https://github.com/PacktPublishing/Artificial-Intelligence-with-Python-Cookbook/blob/master/chapter03/Recommending_products.ipynb.

We have to set a lot of parameters, including the number of latent dimensions and the number of epochs:

```
import torch
from spotlight.factorization.explicit import ExplicitFactorizationModel
from spotlight.evaluation import (
    rmse_score,
    precision_recall_score
)

model = ExplicitFactorizationModel(
    loss='regression',
    embedding_dim=128,
    n_iter=10,
    batch_size=1024,
    l2=1e-9,
    learning_rate=1e-3,
    use_cuda=torch.cuda.is_available()
)
model.fit(train, verbose=True)
train_rmse = rmse_score(model, train)
test_rmse = rmse_score(model, test)
print('Train RMSE {:.3f}, test RMSE {:.3f}'.format(
    train_rmse, test_rmse
))
```

The result is shown in the following screenshot:

```
Epoch 0: loss 2.770107564591749
Epoch 1: loss 0.7509063941125441
Epoch 2: loss 0.6736548199643415
Epoch 3: loss 0.5723841978727623
Epoch 4: loss 0.4470017605662601
Epoch 5: loss 0.32827346986698136
Epoch 6: loss 0.23731406738849983
Epoch 7: loss 0.174902199253408
Epoch 8: loss 0.13326672287996882
Epoch 9: loss 0.10586522202579828
Train RMSE 0.264, test RMSE 0.964
```

We get the following recommendations:

```
User 3
Known positives:
    The Atlantis Complex (Artemis Fowl, #7)
    Sentinel (Covenant, #5)
    The Devil Wears Prada (The Devil Wears Prada, #1)
Recommended:
    Wool (Wool, #1)
    The Magicians' Guild (Black Magician Trilogy, #1)
    Where the Heart Is
User 9999
Known positives:
    City of Glass (The Mortal Instruments, #3)
    The Magicians' Guild (Black Magician Trilogy, #1)
    Bridge to Terabithia
Recommended:
    Gap Creek
    Great Expectations
    A Tale for the Time Being
User 15000
Known positives:
    Enchanters' End Game (The Belgariad, #5)
    The Life and Times of the Thunderbolt Kid
    Beautiful Creatures (Caster Chronicles, #1)
Recommended:
    The Name of the Wind (The Kingkiller Chronicle, #1)
    A Game of Thrones (A Song of Ice and Fire, #1)
    A Prayer for Owen Meany
```

Now we'll use the lightfm recommendation algorithm:

```
from lightfm import LightFM
from lightfm.evaluation import precision_at_k

# Instantiate and train the model
model = LightFM(loss='warp')
model.fit(train.tocoo(), epochs=30, num_threads=2)
```

```
test_precision = precision_at_k(model, test.tocoo(), k=5)
print(
    'mean test precision at 5: {:.3f}'.format(
        test_precision.mean()
))
mean test precision at 5: 0.114
```

We can also look at the recommendations, as follows:

```
User 3
Known positives:
The Atlantis Complex (Artemis Fowl, #7)
Sentinel (Covenant, #5)
The Devil Wears Prada (The Devil Wears Prada, #1)
Recommended:
The Life and Times of the Thunderbolt Kid
The Atlantis Complex (Artemis Fowl, #7)
Beautiful Creatures (Caster Chronicles, #1)
User 9999
Known positives:
City of Glass (The Mortal Instruments, #3)
The Magicians' Guild (Black Magician Trilogy, #1)
Bridge to Terabithia
Recommended:
Enchanters' End Game (The Belgariad, #5)
City of Glass (The Mortal Instruments, #3)
Frankenstein
User 15000
Known positives:
Enchanters' End Game (The Belgariad, #5)
The Life and Times of the Thunderbolt Kid
Beautiful Creatures (Caster Chronicles, #1)
Recommended:
Where the Heart Is
Mockingjay (The Hunger Games, #3)
A Map of the World
```

Both recommenders have their applications. On the basis of the precision at k ($k=5$) for both recommenders, we can conclude that the second recommender, lightfm, performs better.

How it works...

Recommenders recommend products to users.

They can produce recommendations based on different principles, such as the following:

- They can predict based on the assumption that customers who have shown similar tastes in previous purchases will buy similar items in the future (**collaborative filtering**).
- Predictions based on the idea that customers will have an interest in items similar to the ones they've bought in the past (**content-based filtering**).
- Predictions based on a combination of collaborative filtering, content-based filtering, or other approaches (a **hybrid recommender**).

Hybrid models can combine approaches in different ways, such as making content-based and collaborative-based predictions separately and then adding up the scores, or by unifying the approaches into a single model.

Both models we've tried are based on the idea that we can separate the influences of users and items. We'll explain each model in turn, and how they combine approaches, but first let's explain the metric we are using: precision at k .

Precision at k

The metric we are extracting here is **precision at k** . For example, precision at 10 calculates the number of relevant results among the top k documents, with typically $k=5$ or $k=10$.

Precision at k doesn't take into account the ordering within the top k results, nor does it include how many of the really good results that we absolutely should have captured are actually returned: that would be recall. That said, precision at k is a sensible metric, and it's intuitive.

Matrix factorization

The explicit model in spotlight is based on the matrix factorization technique presented by Yehuda Koren and others (in *Matrix Factorization Techniques for Recommender Systems*, 2009). The basic idea is that a user-item (interaction) matrix can be decomposed into two components, H and W , representing user latent factors and item latent factors respectively, so that recommendations given an item i and a user u can be calculated as follows:

$$\tilde{r}_{ui} = \sum_{f=0}^{nfactors} H_{u,f} W_{f,i}$$



Matrix decomposition or **matrix factorization** is the factorization of a matrix into a product of matrices. Many different such decompositions exist, serving a variety of purposes.

A relatively simple decomposition is the **singular value decomposition (SVD)** however, modern recommenders use other decompositions. Both the `spotlight` matrix factorization and the `lightfm` model use linear integrations.

The lightfm model

The `lightfm` model was introduced in a paper by Kula (*Metadata Embeddings for User and Item Cold-start Recommendations*, 2015). More specifically, we use the WARP loss, which is explained in the paper *WSABIE: Scaling Up To Large Vocabulary Image Annotation* by Jason Weston et al., from 2011.

In the `lightfm` algorithm, the following function is used for predictions:

$$\tilde{r}_{ui} = f(H_{u,f}W_{f,i}b_i b_u)$$

In the preceding function, we have bias terms for users and items and f is the sigmoid function.

The model training maximizes the likelihood of the data conditional on the parameters expressed as follows:

$$\prod_{(u,i) \in S_+} \tilde{r}_{ui} \prod_{(u,i) \in S_-} (1 - \tilde{r}_{ui})$$

There are different ways to measure how well recommenders are performing and, as always, which one we choose to use depends on the goal we are trying to achieve.

See also

Again, there are a lot of libraries around that make it easy to get up and running. First of all, I'd like to highlight these two, which we've already used in this recipe:

- `lightfm`: <https://github.com/lyst/lightfm>
- `Spotlight`: <https://maciejkula.github.io/spotlight>

But some others are very promising, too:

- Polara, which includes an algorithm called HybridSVD that seems to be very strong: <https://github.com/evfro/polara>
- DeepRec, which provides deep learning models for recommendations (based on TensorFlow): <https://github.com/cheungdaven/DeepRec>

You can find a demonstration of library functionality for item ranking with a dataset at the following repo: https://github.com/cheungdaven/DeepRec/blob/master/test/test_item_ranking.py.

Microsoft has been writing about recommender best practices: <https://github.com/Microsoft/Recommenders>.

Last, but not least, you might find the following reading list about recommender systems useful: <https://github.com/DeepGraphLearning/RecommenderSystems/blob/master/readingList.md>.

Spotting fraudster communities

In this recipe, we'll try to detect fraud communities using methods from network analysis. This is a use case that often seems to come up in graph analyses and intuitively appeals because, when carrying out fraud detection, we are interested in relationships between people, such as whether they live close together, are connected over social media, or have the same job.

Getting ready

In order to get everything in place for the recipe, we'll install the required libraries and we'll download a dataset.

We will use the following libraries:

- networkx - is a graph analysis library: <https://networkx.github.io/documentation>.
- annoy - is a very efficient nearest-neighbors implementation: <https://github.com/sparkify/annoy>.
- tqdm - to provide us with progress bars: <https://github.com/tqdm/tqdm>.

Furthermore, we'll use SciPy, but this comes with the Anaconda distribution:

```
!pip install networkx annoy tqdm python-louvain
```

We'll use the following dataset of fraudulent credit card transactions: <https://www.kaggle.com/mlg-ulb/creditcardfraud>.



The Credit Card Fraud dataset contains transactions made by credit cards in September 2013 by European cardholders. This dataset presents transactions that occurred over two days, with 492 fraudulent transactions out of 284,807 transactions. The dataset is highly unbalanced: the positive class (fraud) accounts for 0.172% of all transactions.

Let's import the dataset, and then split it into training and test sets:

```
import pandas as pd
from sklearn.datasets import fetch_openml
import random

X, y = fetch_openml(data_id=1597, return_X_y=True)

samples = random.choices(
    list(range(X.shape[0])), k=int(X.shape[0] * 0.33)
)
X_train = X[(samples), :]
```

We are ready! Let's do the recipe!

How to do it...

We'll first create an adjacency matrix, then we can apply the community detection methods to it, and lastly, we'll evaluate the quality of the generated communities. The whole process has the added difficulty associated with a large dataset, which means we can only apply certain algorithms.

Creating an adjacency matrix

First, we need to calculate the distances of all points. This is a real problem with a large dataset such as this. You can find several approaches online.

We use the `annoy` library from Spotify for this purpose, which is very fast and memory-efficient:

```
from annoy import AnnoyIndex
t = AnnoyIndex(X_train.shape[1], 'euclidean') # Length of item vector that
will be indexed
for i, v in enumerate(X_train):
    t.add_item(i, v)

t.build(10) # 10 trees
```

We can then initialize our adjacency matrix with the distances as given by our index:

```
from tqdm import trange
from scipy.sparse import lil_matrix

MAX_NEIGHBORS = 10000 # Careful: this parameter determines the run-time of
the loop!
THRESHOLD = 6.0

def get_neighbors(i):
    neighbors, distances = t.get_nns_by_item(i, MAX_NEIGHBORS,
include_distances=True)
    return [n for n, d in zip(neighbors, distances) if d < THRESHOLD]

n_rows = X_train.shape[0]
A = lil_matrix((n_rows, n_rows), dtype=np.bool_)
for i in trange(n_rows):
    neighborhood = get_neighbors(i)
    for n in neighborhood:
        A[i, n] = 1
        A[n, i] = 1
```

We can now apply some community detection algorithms.

Community detection algorithms

The size of our matrix leaves us with limited choice. We'll apply the two following algorithms:

- **Strongly Connected Components (SCC)**
- The Louvain algorithm

We can apply the SCC algorithm directly onto the adjacency matrix as follows:

```
from scipy.sparse.csgraph import connected_components

n_components, labels = connected_components(
    A,
    directed=False,
    return_labels=True
)
```

For the second algorithm, we first need to convert the adjacency matrix to a graph; this means that we treat each point in the matrix as an edge between nodes. In order to save space, we use a simplified graph class for this:

```
import networkx as nx

class ThinGraph(nx.Graph):
    all_edge_dict = {'weight': 1}

    def single_edge_dict(self):
        return self.all_edge_dict
    edge_attr_dict_factory = single_edge_dict

G = ThinGraph(A)
```

Then we can apply the Louvain algorithm as follows:

```
import community # this is the python-louvain package

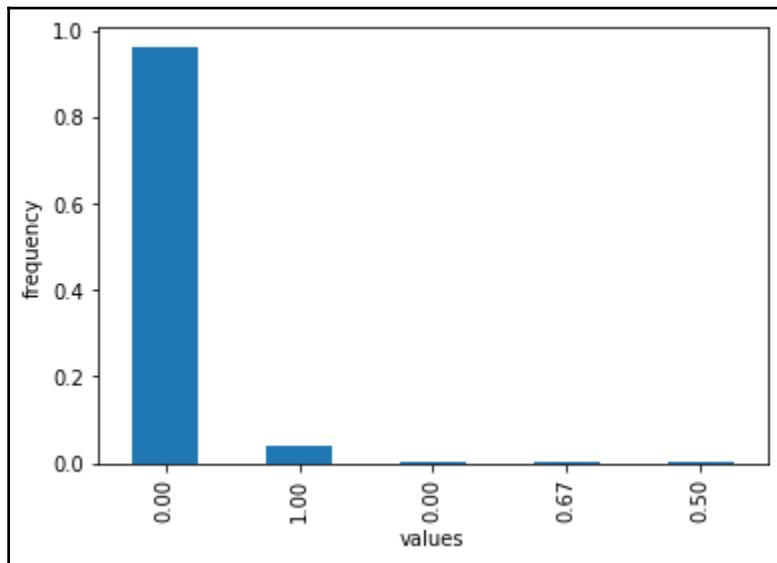
partition = community.best_partition(G)
```

Now we have two different partitions of our dataset. Let's find out if they are worth anything!

Evaluating the communities

In the ideal case, we'd expect that some communities have only fraudsters in them, while others (most) have none at all. This purity is what we would be looking for in a perfect community. However, since we also possibly want some suggestions of who else might be a fraudster, we would anticipate some points to be labeled as fraudsters in a majority-nonfraudster group and vice versa.

We can start by looking at the histograms of fraud frequency per community. The Louvain fraudster distribution looks like this:



This shows that communities have a very high frequency of people who are not fraudsters, and very few other values. But can we quantify how good this is?

We can describe the fraudster distribution by calculating the **class entropy** in each cluster. We'll explain entropy in the *How it works...* section.

We can then create appropriately chosen random experiments to see if any other community assignments would have resulted in a better class entropy. If we randomly shuffle the fraudsters and then calculate the entropy across communities, we get an entropy distribution. This will give us a **p-value**, the **statistical significance**, for the entropy of the Louvain communities.



The **p-value** is the probability that we get a distribution like this (or better) purely by chance.

You can find the implementation for the sampling in the notebook on GitHub.

We get a very low significance, meaning that it is highly unlikely to have gotten anything like this by chance, which leads us to conclude that we have found meaningful clusters in terms of identifying fraudsters.

How it works...

The hardest part of network analysis with a big dataset is constructing the adjacency matrix. You can find different approaches in the notebook available online. The two problems are runtime and memory. Both can increase exponentially with the number of data points.

Our dataset contains 284,807 points. This means a full connectivity matrix between all points would take a few hundred gigabytes (at 4 bytes per point), $284807^2 \times 4/10^9$.

We are using a sparse matrix where most adjacencies are 0s if they don't exceed the given threshold. We represent each connection between the points as a Boolean (1 bit) and we take a sample of 33%, 93,986 points, rather than the full dataset.

Graph community algorithms

Let's go through two graph community algorithms to get an idea of how they work.

Louvain algorithm

We've used the Louvain algorithm in this recipe. The algorithm was published in 2008 by Blondel *et al.* (<https://arxiv.org/abs/0803.0476>). Since its time complexity is $O(n^2)$, the Louvain algorithm can and has been used with big datasets, including data from Twitter containing 2.4 million nodes and 38 million links.

The main idea of the Louvain algorithm is to proceed in an agglomerative manner by successively merging communities together so as to increase their connectedness. The connectedness is measured by edge modularity, Q , which is the density of edges within a community connected to other vertices of the same community versus the vertices of other communities. Any switch of community for a vertex i has an associated ΔQ . After the initial assignment of each of the vertices to their own communities, the heuristic operates in two steps: a greedy assignment of vertices to communities, and a coarse graining.

- For all vertices i , assign them to the community so that ΔQ is the highest it can be. This step can be repeated a few times until no improvement in modularity occurs.
- All communities are treated as vertices. This means that edges are also grouped together so that all edges that are part of the vertices that were grouped together are now edges of the newly created vertex.

These two steps are iterated until no further improvement in modularity occurs.

Girvan–Newman algorithm

As an example of another algorithm, let's look at the Girvan–Newman algorithm.

The Girvan–Newman algorithm (by Girvan and Newman, 2002, with the paper available at <https://www.pnas.org/content/99/12/7821>) is based on the concept of the shortest path between nodes. The **edge betweenness** of an edge is the number of shortest paths between nodes that run along the edge.

The algorithm works as follows:

1. Calculate the edge betweenness of all edges.
2. Remove the edge of the highest edge betweenness.
3. Recalculate the edge betweenness.
4. Iterate steps 2 and 3 until no edges remain.

The result is a dendrogram that shows the arrangement of clusters by the steps of the algorithm.

The whole algorithm has a time complexity of $O(m^2n)$, with edges m and vertices n .

Information entropy

Given a discrete random variable X with possible values (or outcomes) x_1, x_2, \dots, x_n that occur with probability $P(x_i)$, the entropy of X is formally defined as follows:

$$H(X) = - \sum_{i=1}^n P(x_i) \log P(x_i)$$

This is generally taken as the level of surprise, uncertainty, or chaos in a random variable.

If a variable is not discrete, we can apply binning (for example, via a histogram) or use non-discrete versions of the formula.

There's more...

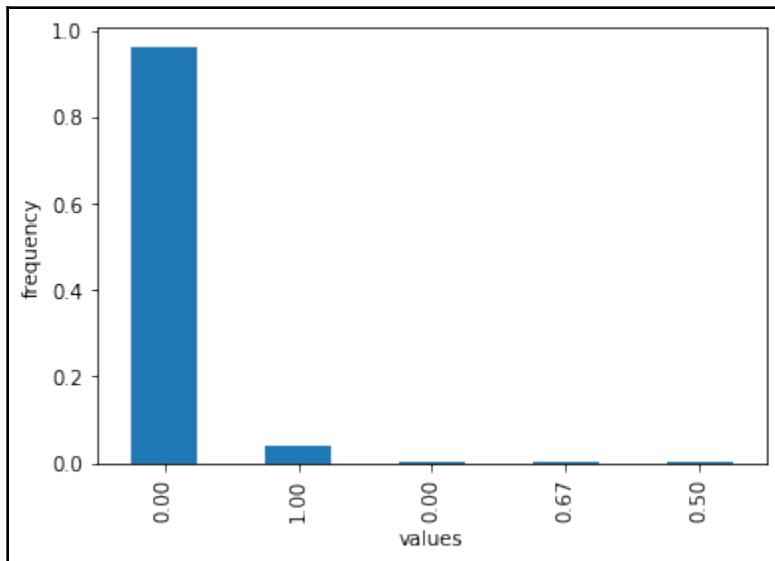
We could have applied other algorithms, such as SCC, published by David Pearce in 2005 (in *An Improved Algorithm for Finding the Strongly Connected Components of a Directed Graph*).

We can try this method out as well:

```
from scipy.sparse.csgraph import connected_components

n_components, labels = connected_components(
    A,
    directed=False,
    return_labels=True
)
```

The SCC community fraudster distribution looks like this:



And again we get a p-value that shows very high statistical significance. This means that this is unlikely to have occurred by pure chance, and indicates that our method is indeed a good classifier for fraud.

We could have also applied more traditional clustering algorithms. For example, the affinity propagation algorithm takes an adjacency matrix, as follows:

```
from sklearn.cluster import AffinityPropagation

ap = AffinityPropagation(
    affinity='precomputed'
).fit(A)
```

There are a host of other methods that we could apply. For some of them, we'd have to convert the adjacency matrix to a distance matrix.

See also

You can find reading materials about graph classification and graph algorithms on GitHub, collected by Benedek Rozemberczki, at <https://github.com/benedekrozemberczki/awesome-graph-classification>.

If you are interested in graph convolution networks or graph attention networks, there's also a useful list for you at <https://github.com/Jiakui/awesome-gcn>.

There are some very nice graph libraries around for Python with many implementations for community detection or graph analysis:

- **Cdlib:** <https://cdlib.readthedocs.io/en/latest/>
- **Karateclub:** <https://karateclub.readthedocs.io/en/latest/>
- **Networkx:** <https://networkx.github.io/>
- **Label propagation:** https://github.com/yamaguchiyuto/label_propagation

Most Python libraries work with small- to medium-sized adjacency matrices (perhaps up to around 1,000 edges). Libraries suited for bigger data sizes include the following:

- **Snap.py:** <https://snap.stanford.edu/snappy/index.html>
- **Python-louvain:** <https://github.com/taynaud/python-louvain>
- **Graph-tool:** <https://graph-tool.skewed.de/>

Cdlib also contains the BigClam algorithm, which works with big graphs.

Some graph databases such as neo4j, which comes with a Python interface, implement community detection algorithms: <https://neo4j.com/docs/graph-algorithms/current/>.

4

Probabilistic Modeling

This chapter is about uncertainty and probabilistic approaches. State-of-the-art machine learning systems have two significant shortcomings.

First of all, they can be overconfident (or sometimes underconfident) in their prediction. In practice, given noisy data, even if we observe the best practice of cross-validating with unseen datasets, this confidence might not be warranted. Especially in regulated or sensitive environments, such as in financial services, healthcare, security, and intelligence, we need to be very careful about our predictions and how accurate they are.

Secondly, the more complex a machine learning system is, the more data we need to fit our model, and the more severe the risk of overfitting.

Probabilistic models are models that produce probabilistic inferences using stochastic sampling techniques. By parametrizing distributions and inherent uncertainties, we can overcome these problems and obtain accuracies that would otherwise take more data without these assumptions.

In this chapter, we'll build a stock-price prediction model with different plug-in methods for confidence estimation. We'll then cover estimating customer lifetime, a common problem in businesses that serve customers. We'll also look at diagnosing a disease, and we'll quantify credit risk, taking into account different types of uncertainty.

This chapter covers the following recipes:

- Predicting stock prices with confidence
- Estimating customer lifetime value
- Diagnosing a disease
- Stopping credit defaults

Technical requirements

In this chapter, we mainly use the following:

- scikit-learn, as before
- Keras, as before
- Lifetimes (<https://lifetimes.readthedocs.io/>), a library for customer lifetime value
- tensorflow-probability (**tfp**; <https://www.tensorflow.org/probability>)

You'll find the code for this chapter on GitHub at <https://github.com/PacktPublishing/AI-with-Python-Cookbook/tree/master/chapter04>.

Predicting stock prices with confidence

The efficient market hypothesis postulates that at any given time, stock prices integrate all information about a stock, and therefore, the market cannot be consistently outperformed with superior strategy or, more generally, better information. However, it can be argued that current practice in investment banking, where machine learning and statistics are built into algorithmic trading systems, contradicts this. But these algorithms can fail, as seen in the 2010 flash crash or when systemic risks are underestimated, as discussed by Roger Lowenstein in his book *When Genius Failed: The Rise and Fall of Long-Term Capital Management*.

In this recipe, we'll build a simple stock prediction pipeline in scikit-learn, and we'll produce probability estimates using different methods. We'll then evaluate our different approaches.

Getting ready

We'll retrieve historical stock prices using the `yfinance` library.

Here's how we install it:

```
pip install yfinance
```

`yfinance` will help us to download historical stock prices.

How to do it...

In a practical setting, we'd want to answer the following question: given the level of prices, are they going to rise or to fall, and how much?

In order to make progress toward this goal, we'll proceed with the following steps:

1. Download stock prices.
2. Create a featurization function.
3. Write an evaluation function.
4. Train models to predict stocks and compare performance.

Particularly, we'll compare the following methods for generating confidence values:

- Platt scaling
- Naive Bayes
- Isotonic regression

We'll discuss these methods and their background in the *How it works...* section.

Let's give it a go!

1. **Download stock prices:** We'll download Microsoft's prices:

```
import yfinance as yf  
  
msft = yf.Ticker('MSFT')  
hist = msft.history(period='max')
```

Now we have our stock prices available as the pandas DataFrame `hist`.

2. **Create a featurization function:** So, let's start with a function that will give us a dataset for training and prediction given a window size and a shift; basically, how many descriptors we want for each price and how far we look into the future:

```
from typing import Tuple  
import numpy as np  
import pandas as pd  
import scipy  
  
  
def generate_data(  
    data: pd.DataFrame, window_size: int, shift: int  
) -> Tuple[np.array, np.array]:
```

```
y = data.shift(shift + window_size)
observation_window = []
for i in range(window_size):
    observation_window.append(
        data.shift(i)
    )
X = pd.concat(observation_window, axis=1)
y = (y - X.values[:, -1]) / X.values[:, -1]
X = X.pct_change(axis=1).values[:, 1:]
inds = (~np.isnan(X)).any(axis=1) & (~np.isnan(y))
X, y = X[inds], y[inds]
return X, y
```

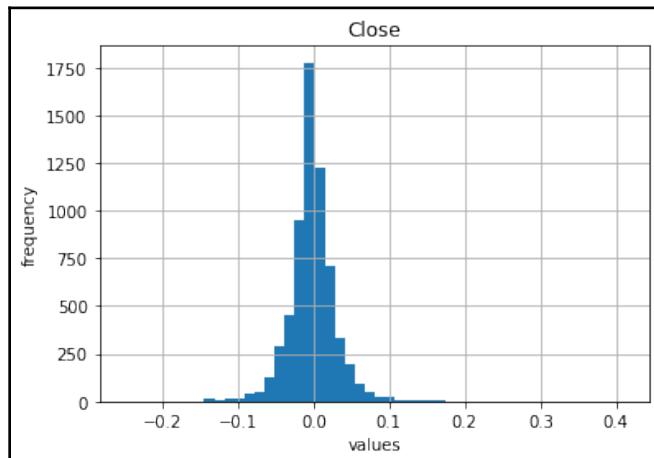
We'll then use our new function, `generate_data()`, to generate our training and testing datasets:

```
from sklearn.model_selection import train_test_split

X, y = generate_data(hist.Close, shift=1, window_size=30)
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

This is a common pattern, of course, and we've seen this a few times by now in recipes leading up to this: we generate our dataset, and then split it into training and validation sets, where the training set is used (as the name suggests) for training, and the validation set is used for checking how well our algorithm works (in particular, whether we've overfitted).

Our datasets are approximately normally distributed. Here's what our target looks like in training:



We can see that there's a skew to the left, in the sense that more values are below zero (about 49%) than above (about 43%). This means that in training, prices go down rather than up.

We are not done with our dataset yet, however; we need to do one more transformation. Our scenario is that we want to apply this model to help us decide whether to buy a stock on the chance that prices are going up. We are going to separate three different classes:

- Prices go up by x .
- Prices stay the same.
- Prices go down by x .

In the following code block, we apply this cutoff by x given the `threshold` parameter:

```
def threshold_vector(x, threshold=0.02):
    def threshold_scalar(f):
        if f > threshold:
            return 1
        elif f < -threshold:
            return -1
        return 0
    return np.vectorize(threshold_scalar)(x)

y_train_classes, y_test_classes = threshold_vector(y_train),
threshold_vector(y_test)
```

After this, we have the thresholded y values for training and testing (validation).

3. **Write an evaluation function:** This is to measure our performance at predicting stock prices with a given model. For the evaluation, we need a helper function to convert from integer encoding into one-hot encoding.

In the evaluation, we calculate and print the **Area Under the Curve (AUC)** as the performance measure. We create a function, `measure_perf()`, which measures performance and prints out relevant metrics, given a model such as this:

```
from sklearn import metrics

def to_one_hot(a):
    """Convert from integer encoding to one-hot"""
    b = np.zeros((
        a.size, 3
    ))
    b[np.arange(a.size), a+1] = 1
```

```
    return b

def measure_perf(model, y_test_classes):
    y_pred = model.predict(X_test)
    auc = metrics.roc_auc_score(
        to_one_hot(y_test_classes),
        to_one_hot(y_pred),
        multi_class='ovo'
    )
    print('AUC: {:.3f}'.format(auc))
```

We can use our new method now to evaluate the performance after training our models.

4. **Train models to predict stocks and compare performance:** We'll now compare the following methods to generate probabilistic outcomes from our three models, the first two of which we can implement quickly:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.calibration import CalibratedClassifierCV

rf = RandomForestClassifier(
    n_estimators=500, n_jobs=-1
).fit(X_train, y_train_classes)
platt = CalibratedClassifierCV(
    rf, method='sigmoid'
).fit(X_train, y_train_classes)
isotonic = CalibratedClassifierCV(
    rf, method='isotonic'
).fit(X_train, y_train_classes)
print('Platt:')
measure_perf(platt, y_test_classes)
print('Isotonic:')
measure_perf(isotonic, y_test_classes)
#Platt:
#AUC: 0.504
#Isotonic:
#AUC: 0.505
```

For Naive Bayes, we try different variants: categorical Naive Bayes and complement Naive Bayes:

```
from sklearn.ensemble import StackingClassifier
from sklearn.naive_bayes import ComplementNB, CategoricalNB

def create_classifier(final_estimator):
    estimators = [
        ('rf', RandomForestClassifier(
```

```
        n_estimators=100,
        n_jobs=-1
    ))
]
return StackingClassifier(
    estimators=estimators,
    final_estimator=final_estimator,
    stack_method='predict_proba'
).fit(X_train, y_train_classes)

measure_perf(create_classifier(CategoricalNB()), y_test_classes)
measure_perf(create_classifier(ComplementNB()), y_test_classes)
#CategoricalNB:
#AUC: 0.500
#ComplementNB:
#AUC: 0.591
```

We find that neither Platt scaling (logistic regression) nor isotonic regression can deal well with our dataset. Naive Bayes regression doesn't get much better than 50%, which is nothing that we'd want to bet our money on, even if it's slightly better than random choice. However, the complement Naive Bayes classifier performs much better, at 59% AUC.

How it works...

We've seen that we can create a predictor for stock prices. We've broken this down into creating data, and validating and training a model. In the end, we found a method that would give us hope that we could actually use it in practice.

Let's go through our data generation first, and then over our different methods.

Featurization

This is central to any work in artificial intelligence. Before doing any work or the first we look at our dataset, we should ask ourselves what we choose as the unit of our observational units, and how are we going to describe our points in a way that's meaningful and can be captured by an algorithm. This is something that becomes automatic with experience.

In our `generate_data()` function, we extract a dataset for training and testing from stock price history data. We are focused on predicting individual prices, so our observational unit is a single stock price. For each price, we need to extract features, other prices running up to it. We extract prices across a time period that can help us predict future values. To be more precise, we don't use prices directly; we have to normalize them first, so it's better to refer to them as price levels rather than prices.

Using our method, we parametrize our data for predictions with different time horizons and a number of points. The price level is extracted over a window, a period of days (features). Finally, a price level, some days later, is to be predicted (targets). The time period and the shift are our two additional parameters: `window_size` and `shift`. This function returns x , the history of stock prices with their window, and y , the stock prices in the future to be predicted.

There are more concerns that we have to address. We've already seen a few methods for data treatment in time series, in the *Forecasting CO₂ time series* recipe in Chapter 2, *Advanced Topics in Supervised Machine Learning*. In particular, stationarity and normalization are concerns that are shared in this recipe as well (you might want to flip back and have a look at the explanation there).

Features are normalized to a mean of 0 and then differenced (each value in a window to the previous values) as a percentage change. The differencing step is done to introduce a measure of stationarity. Particularly, the target is expressed as the percentage change with respect to the last value in the window, the features.

We'll look next at Platt scaling, which is one of the simplest ways of scaling model predictions to get probabilistic outcomes.

Platt scaling

Platt scaling (John Platt, 1999, *Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods*) is the first method of scaling model outcomes that we've used. Simply stated, it's applying logistic regression on top of our classifier predictions. The logistic regression can be expressed as follows (equation 1):

$$P(y = 1|x) = \frac{1}{1 + \exp(Af(x) + B)}$$

Here, A and B are learned by a maximum likelihood method.

We are searching for A and B , as follows:

$$\operatorname{argmin}_{A,B} - \sum_i y_i \log(p_i) + (1 - y_i) \log(1 - p_i),$$

Here p refers to the preceding equation 1.

As a gradient descent, we can iteratively apply the following two steps:

1. Calculate the gradient as the differential of the likelihood function.
2. Update the parameters according to the gradient scaled by the learning rate.

In the next subsection, we'll look at an alternative method of probabilistic calibration using isotonic regression.

Isotonic regression

Isotonic regression (Zadrozny and Elkan, 2001, *Learning and Making Decisions When Costs and Probabilities are Both Unknown*) is regression using an isotonic function – that is, a function that is monotonically increasing or non-decreasing, as a function approximation while minimizing the mean squared error.

We can express this as follows:

$$y_i = m(f(x_i)) + \epsilon_i,$$

Here m is our isotonic function, x and y are features and target, and f is our classifier.

Next, we'll look at one of the simplest probabilistic models, Naive Bayes.

Naive Bayes

The Naive Bayes classifier is based on the Bayes theorem.

The Bayes theorem is about the conditional probability of an event A occurring given B :

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

$P(A)$ is the probability of observing A (marginal probability of A). Given the formulation, $P(B)$, in the denominator, can't be 0. The reasoning behind this is worth reading up on.

A Naive Bayes classifier is about the probability of classes given features. We can plug class k and feature x into the Bayes theorem, which looks like this:

$$p(C_k \mid \mathbf{x}) = \frac{p(C_k) p(\mathbf{x} \mid C_k)}{p(\mathbf{x})}$$

It is called naive because it assumes that features are independent of each other, so the nominator can be simplified as follows:

$$p(C_k) \prod_{i=1}^n p(x_i \mid C_k)$$

In the next section, we'll look at additional material.

See also

Here are some resources that you can go through:

- For Platt scaling, refer to *Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods* by John Platt, (1999).
- For isotonic regression, as in our application for probability estimates in classification, please refer to *Transforming classifier scores into accurate multi-class probability estimates* by Zadrozny, B. and Elkan, C., (2002).
- For a comparison between the two, refer to *Predicting Good Probabilities with Supervised Learning* by A. Niculescu-Mizil & R. Caruana, ICML, (2005). Refer to Rennie, J. D. and others, *Tackling the Poor Assumptions of Naive Bayes Text Classifiers* (2003), on the complement Naive Bayes algorithm.
- The scikit-learn documentation gives an overview of confidence calibration (https://scikit-learn.org/stable/auto_examples/calibration/plot_calibration_curve.html#sphx-glr-auto-examples-calibration-plot-calibration-curve-py).
- For an approach applied to deep learning models, see the ICLR 2018 paper by Lee and others, *Training Confidence-Calibrated Classifiers for Detecting Out-of-Distribution Samples* (<https://arxiv.org/abs/1711.09325>). Their code is available on GitHub at https://github.com/alinlab/Confident_classifier.

You can find more examples of probabilistic analyses of time series with different frameworks at the following links:

- **bayesloop**: Analyzing stock-market fluctuations – <http://docs.bayesloop.com/en/stable/examples/stockmarketfluctuations.html>
- **Tensorflow Probability**: Different methods – https://www.tensorflow.org/probability/examples/Structural_Time_Series_Modeling_Case_Studies_Atmospheric_CO2_and_Electricity_Demand
- **Pyro**: Gaussian process time-series modeling – <https://pyro.ai/examples/timeseries.html>

Estimating customer lifetime value

In this recipe, we will learn how to compute lifetime values and the value a customer provides to this company. This is important for the marketing budget – for example, in lead acquisition or ads spent based on customer segments. We'll do this by modeling separately changes in customer purchase patterns over time and purchase values.

Getting ready

We'll need the `lifetimes` package for this recipe. Let's install it as shown in the following code:

```
pip install lifetimes
```

Now we can get started.

How to do it...

Datasets used for customer lifetime values can be either transactional or summarized by the customer.

The summary data should include the following statistics:

- **T**: The transaction period; the elapsed time since the first purchase by the customer
- **Frequency**: The number of purchases by a customer within the observation period

- **Monetary value:** The average value of purchases
- **Recency:** The age of the customer at the time of the last purchase

Let's start with the first step!

1. We'll first fit the **BetaGeo (BD)/Negative Binomial Distribution (NBD)** model to a summary dataset of customer transactions:

```
from lifetimes.datasets import
load_cdnow_summary_data_with_monetary_value
from lifetimes import BetaGeoFitter

bgf = BetaGeoFitter(penalizer_coef=0.0)
bgf.fit(
    data['frequency'],
    data['recency'],
    data['T']
)
```

2. The Gamma-Gamma model for purchase values can't handle customers who don't have repeat purchases, so we'll exclude those before we fit it:

```
from lifetimes import GammaGammaFitter

data_repeat = data[data.frequency>0]
ggf = GammaGammaFitter(penalizer_coef=0.0)
ggf.fit(
    data_repeat.frequency,
    data_repeat.monetary_value
)
```

3. We can then combine the predictions of the model that predicts the number of future transactions (bgf) and the model that predicts average purchase values (ggf) using another of the Lifetimes library's methods. It includes a parameter for discounting future values. We'll include a discount that corresponds to an annualized 12.7%. We'll print five customers' lifetime values:

```
print(ggf.customer_lifetime_value(
    bgf,
    data['frequency'],
    data['recency'],
    data['T'],
    data['monetary_value'],
    time=12,
    discount_rate=0.01
).head(5))
```

The output shows us the customer lifetime values:

```
customer_id
1        140.096218
2        18.943466
3        38.180575
4        38.180575
5        38.180575
```

Now we know who our best customers are, and therefore where to invest our time and resources!

Let's go over some of the methods in this recipe.

How it works...

In this recipe, we've estimated lifetime values of customers based on their purchase patterns.

Each customer has a value to the company. This is important for the marketing budget – for example, in lead acquisition or ads spent based on customer segments. The actual customer lifetime value is known after a customer has left the company; however, we can instead build two different probabilistic forecasting models for each customer:

- Modeling the likelihood of buying more products
- Modeling the average value (revenue) of purchases

We model purchase frequency – or, more precisely, changes in customer purchase patterns over time – using the BG/NBD model, and purchase values using the Gamma-Gamma model. Both of these models exploit non-linear associations between variables.

Finally, we can combine the predictions to obtain lifetime values according to the following formula:

$$\text{CLV} = \text{margin} \times \text{transactions} \times \frac{\text{revenue}}{\text{transaction}}$$

Let's go over the two submodels that we've used here.

The BG/NBD model

This takes into account the purchasing frequency of customers and the dropout probability of customers.

It comes with the following assumptions:

- Purchases for each customer follow a Poisson distribution with a lambda parameter.
- Customer churn probability p after each transaction follows a beta distribution.
- Transaction rates and the dropout probabilities are independent across customers.

The lambda and p parameters can be estimated according to maximum likelihood estimation.

The Gamma-Gamma model

This model is used to estimate the mean transaction value over the customer's lifetime, $E(M)$, for which we have an imperfect estimate, as follows:

$$m_x = \sum_{i=1}^x \frac{z_i}{x},$$

Here, x is the (unknown) total number of purchases over a customer's lifetime, and z is the value of each purchase.

We assume z to be sampled from gamma distributions, and therefore, the model fit involves finding the shape and scale parameters at the individual level.

See also

This recipe was relatively short because of the excellent work that's been put into the Lifetimes library, which makes a lot of the needed functionality plug-and-play. An extended explanation of this analysis can be found in the Lifetimes documentation (<https://lifetimes.readthedocs.io/en/latest/Quickstart.html>).

The Lifetimes library comes with a range of models (called **fitters**), which you might want to look into. You can find more details about the two methods in this recipe in Fader and others, *Counting your Customers the Easy Way: An Alternative to the Pareto/NBD Model*, 2005, and Batislam and others, *Empirical validation and comparison of models for customer base analysis*, 2007. You can find the details of the Gamma-Gamma model in Fader and Hardi's report, *Gamma-Gamma Model of Monetary Value* (2013).

The Google Cloud Platform GitHub repo shows a model comparison for estimation of customer lifetime values (<https://github.com/GoogleCloudPlatform/tensorflow-lifetime-value>) that includes Lifetimes, a TensorFlow neural network, and AutoML. You can find a very similar dataset of online retail in the UCI machine learning archive (<http://archive.ics.uci.edu/ml/datasets/Online+Retail>).

Lifelines is a library for survival regression by the same author as Lifetimes, Cameron Davidson-Pilon (<https://lifelines.readthedocs.io/en/latest/Survival%20Regression.html>).

Diagnosing a disease

For probabilistic modeling, experimental libraries abound. Running probabilistic networks can be much slower than algorithmic (non-algorithmic) approaches, which until not long ago rendered them impractical for anything but very small datasets. In fact, most of the tutorials and examples relate to toy datasets.

However, this has changed in recent years due to faster hardware and variational inference. With TensorFlow Probability, it is often straightforward to define architectures, losses, and layers, even with probabilistic sampling with full GPU support, and state-of-the-art implementations that support fast training.

In this recipe, we'll implement an application in healthcare – we'll diagnose a disease.

Getting ready

We already have scikit-learn and TensorFlow installed from previous chapters.

For this recipe, we'll need tensorflow-probability as well:

```
pip install tensorflow-probability
```

Now that tensorflow-probability is installed, we'll use it extensively in the next section.

How to do it...

We'll break this down into several steps:

1. Downloading and preparing the data
2. Creating a neural network
3. Model training
4. Validation

We'll start with getting the dataset into Python:

1. **Downloading and preparing the data:** We'll download a dataset of symptoms and heart disease diagnoses collected at the Hungarian Institute of Cardiology, Budapest, by the group of Andras Janosi (<https://www.openml.org/d/1565/>), then preprocess it, construct a neural network in Keras, and probabilistically diagnose based on symptoms.

We will download it from OpenML as we have before. You can see a complete description there. The target originally encodes different statuses, where 0 is healthy, and the others indicate a disease. We'll therefore separate between healthy and not-healthy, and treat this as a binary classification problem. We apply a standard scaler so that we can feed z-scores to the neural network. All of this should be familiar from several earlier recipes in Chapter 1, *Getting Started with Artificial Intelligence in Python*, Chapter 2, *Advanced Topics in Supervised Machine Learning*, Chapter 3, *Patterns, Outliers, and Recommendations*, and Chapter 4, *Probabilistic Modeling*:

```
from sklearn.datasets import fetch_openml
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

X, y = fetch_openml(data_id=1565, return_X_y=True, as_frame=True)
target = (y.astype(int) > 1).astype(float)
scaler = StandardScaler()
X_t = scaler.fit_transform(X)
X_train, X_test, y_train, y_test = train_test_split(
    X_t, target, test_size=0.33, random_state=42
)
```

Now, we have preprocessed and split our dataset into training and test.

2. **Creating a neural network:** The network construction itself is straightforward, and looks very much like any of the other Keras networks that we've seen. The difference is a `DistributionLambda` layer at the end, which we will explain in the next section:

```
import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions
from tensorflow import keras

negloglik = lambda y, p_y: -p_y.log_prob(y)

model = keras.Sequential([
    keras.layers.Dense(12, activation='relu', name='hidden'),
    keras.layers.Dense(1, name='output'),
    tfp.layers.DistributionLambda(
        lambda t: tfd.Bernoulli(logits=t)
    ),
])
model.compile(optimizer=tf.optimizers.Adagrad(learning_rate=0.05),
              loss=negloglik)
```



It is important to notice that instead of finishing off with a final layer, `Dense(2, activation = 'softmax'`, as we would do in binary classification tasks, we'll reduce the outputs to the number of parameters our probability distribution needs, which is just one in the case of the Bernoulli distribution, which takes a single parameter, which is the expected average of the binary outcome.

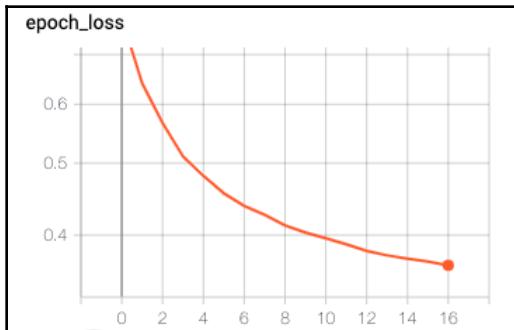
We are using a relatively small model of only 181 parameters. We'll explain the loss function in the *How it works...* section.

3. **Model training:** Now, we can train our model. We'll plot our training loss in tensorboard and we'll enable early stopping:

```
%load_ext tensorboard
callbacks = [
    keras.callbacks.EarlyStopping(patience=10, monitor='loss'),
    keras.callbacks.TensorBoard(log_dir='./logs'),
]
history = model.fit(
    Xt_train,
    y_train.values,
    epochs=2000,
    verbose=False,
```

```
    callbacks=callbacks  
)
```

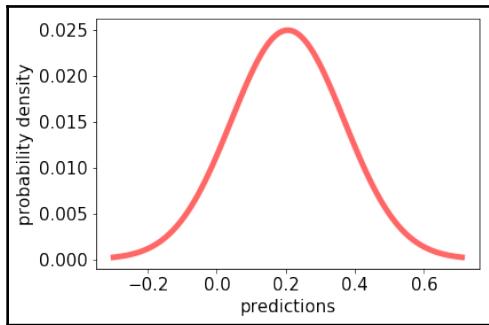
This will run for 2,000 epochs, and it might take a while to complete. From TensorBoard, we can see the training loss over epochs:



4. **Validating the model:** We can now sample from the model. Each network prediction gives us a mean and variance. We can have a look at a single prediction. We've arbitrarily chosen prediction number 10:

```
from scipy.stats import norm  
import matplotlib.pyplot as plt  
  
y_pred = model(Xt_test)  
a = y_pred.mean().numpy()[10]  
b = y_pred.variance().numpy()[10]  
fig, ax = plt.subplots(1, 1)  
x = np.linspace(  
    norm.ppf(0.001, a, b),  
    norm.ppf(0.999, a, b),  
    100  
)  
pdf = norm.pdf(x, a, b)  
ax.plot(  
    x,  
    pdf / np.sum(pdf),  
    'r-', lw=5, alpha=0.6,  
    label='norm pdf'  
)  
plt.ylabel('probability density')  
plt.xlabel('predictions')
```

This prediction looks as follows:



So, each prediction is a sample from a Bernoulli process. We can convert each of these predictions into class probabilities with the cumulative distribution function:

```
def to_classprobs(y_pred):
    N = y_pred.mean().numpy().shape[0]
    class_probs = np.zeros(
        shape=(N, 2)
    )
    for i, (a, b) in enumerate(
        zip(
            y_pred.mean().numpy(),
            y_pred.variance().numpy()
        )
    ):
        conf = norm.cdf(0.5, a, b)
        class_probs[i, 0] = conf
        class_probs[i, 1] = 1 - conf
    return class_probs

class_probs = to_classprobs(y_pred)
```

Now, we can calculate the area under the curve and other metrics against the test targets:

```
import sklearn

def to_one_hot(a):
    """convert from integer encoding to one-hot"""
    b = np.zeros((a.size, 2))
    b[np.arange(a.size), np.int(a.astype(int))] = 1
    return b
```

```
    sklearn.metrics.roc_auc_score(  
        to_one_hot(y_test),  
        class_probs  
)  
    print('{:.3f}'.format(sklearn.metrics.roc_auc_score(to_one_hot(y_te  
st), class_probs)))  
0.859
```

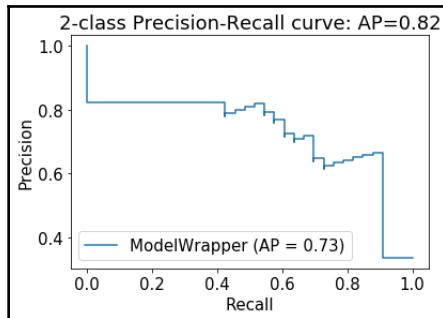
85% AUC sounds good. But we are working in healthcare, so we need to check recall (also called sensitivity) and precision; in other words, do we detect all of the ill patients, and if we diagnose someone, are they actually ill? If we miss someone, they could die of an untreated condition. If we find everyone as ill, it would put a strain on resources.

In the following code segment, we'll look in more detail at the results that we get:

```
from sklearn.metrics import plot_precision_recall_curve  
import matplotlib.pyplot as plt  
from sklearn.metrics import average_precision_score  
  
class ModelWrapper(sklearn.base.ClassifierMixin):  
    _estimator_type = 'classifier'  
    classes_ = [0, 1]  
    def predict_proba(self, X):  
        pred = model(X)  
        return to_classprobs(pred)  
model_wrapper = ModelWrapper()  
average_precision = average_precision_score(  
    to_one_hot(y_test),  
    class_probs  
)  
fig = plot_precision_recall_curve(  
    model_wrapper, Xt_test, y_test  
)  
fig.ax_.set_title(  
    '2-class Precision-Recall curve: '  
    'AP={0:0.2f}'.format(average_precision)  
)
```

This visualizes our results in order to give us a better understanding of the trade-off between precision and recall.

We get the following graph:



This curve visualizes the trade-off inherent in our model, between recall and precision. Given different cutoffs on our confidence (or class probability), we can make a call about whether someone is ill or not. If we want to find everyone ($\text{recall}=100\%$), precision drops down to below 40%. On the other hand, if we want to be always right ($\text{precision}=100\%$) when we diagnose someone as ill, then we'd miss everyone ($\text{recall}=0\%$).

It's now a question of the cost of, respectively, missing people or diagnosing too many, to make a decision on a cutoff for saying someone is ill. Given the importance of treating people, perhaps there's a sweet spot around 90% recall and around 65% precision.

How it works...

We've trained a neural network for probabilistic predictions diagnosing a disease. Let's take this apart a bit, and go through what we've used here.

Aleatoric uncertainty

TensorFlow Probability comes with layer types for modeling different types of uncertainty. Aleatoric uncertainty refers to the stochastic variability of our outcomes given the same input – in other words, we can learn the spread in our data.

We can implement this in Keras and TensorFlow Probability by parameterizing a distribution describing predictions, rather than predicting the input directly. Basically, `DistributionLambda` draws from the distribution (in our case, Bernoulli).

Negative log-likelihood

We use negative log-likelihood as our loss. This loss is often used in maximum likelihood estimation.

What we defined was this:

```
negloglik = lambda y, p_y: -p_y.log_prob(y)
```

The loss function takes two values: y , the target, and a probability distribution that provides the `log_prob()` method. This method returns the log of the probability density at y . Since high values are good, we want to invert the function with the negative.

Bernoulli distribution

The Bernoulli distribution (sometimes called coin-flip distribution) is a discrete distribution of an event with two outcomes occurring with probabilities p and $q = 1 - p$. There's a single parameter to it, which is p . We could have chosen other modeling options, such as the categorical distribution on top of a softmax activation layer.

Metrics

Finally, we touched on recall and precision.

They are defined as follows:

$$\text{Precision} = \frac{tp}{tp + fp}$$

$$\text{Recall} = \frac{tp}{tp + fn}$$

We've seen **True Positives (tp)**, **False Positives (fp)**, and **False Negatives (fn)** before. As a reminder, true positives refer to correct predictions, false positives refer to values incorrectly predicted as positive, and false negatives refer to those incorrectly predicted as negatives.

See also

In this recipe, you've seen how to use a probabilistic model for a health application. There are many other datasets and many different ways of doing probabilistic inference. Please see TensorFlow Probability as one of the frameworks in probabilistic modeling that has the most traction (<https://www.tensorflow.org/probability>). It comes with a wide range of tutorials.

Stopping credit defaults

For a company that extends credit to its customers, in order to be profitable, the most important criterion for approving applications is whether they can pay back their debts. This is determined by a process called credit scoring that is based on the financial history and socio-economic information of the customer. Traditionally, for credit scoring, scorecards have been used, although in recent years, these simple models have given way to more sophisticated machine learning models. Scorecards are basically checklists of different items of information, each associated with points that are all added up in the end and compared to a pass mark.

We'll use a relatively small dataset of credit card applications; however, it can still give us some insights into how to do credit scoring with neural network models. We'll implement a model that includes a distribution of weights as well as a distribution over outputs. This is called epistemic, aleatoric uncertainty, and will give us even better information about how trustworthy predictions are.

Getting ready

We'll be using `tensorflow-probability`. Just in case you skipped the previous recipe, *Diagnosing a disease*, here's how to install it:

```
pip install tensorflow-probability
```

Now, we should be ready with Keras and `tensorflow-probability`.

How to do it...

Let's get the dataset and preprocess it, then we create the model, train the model, and validate it:

1. **Download and prepare the dataset:** The dataset that we'll use for this recipe was published in 2009 (I-Cheng Yeh and Che-hui Lien, *The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients*), and originally hosted on the UCI machine learning repository at <https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients>.

We'll download the data with `openml` using scikit-learn's utility function:

```
import numpy as np
from sklearn.datasets import fetch_openml

openml_frame = fetch_openml(data_id=42477, as_frame=True)
data = openml_frame['data']
```

This gives us features about customer demographics and their application.

We'll use a very standard process of preprocessing that we've seen many times before in this book and that we'll largely breeze through. We could have examined the features more, or done some more work on transformations and feature engineering, but this is beside the point of this recipe.

All the features register as numeric, so we only apply standard scaling. Here's the preprocessing, and then we'll separate it into training and test datasets:

```
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

scaler = StandardScaler()
X = scaler.fit_transform(
    data
)
target_dict = {val: num for num, val in
enumerate(list(openml_frame['target'].unique()))}
y = openml_frame['target'].apply(lambda x:
target_dict[x]).astype('float').values
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.33, random_state=42
)
```

Now that that's done, let's create the model.

2. **Create a model:** First, we need the priors and posteriors. This is done by directly following the online TensorFlow Probability tutorial (http://www.github.com/tensorflow/probability/blob/master/tensorflow_probability/examples/jupyter_notebooks/Probabilistic_Layers_Regression.ipynb), and is appropriate for normal distributions:

```

import tensorflow as tf
import tensorflow_probability as tfp
import matplotlib.pyplot as plt
tfd = tfp.distributions
%matplotlib inline

negloglik = lambda y, rv_y: -rv_y.log_prob(y)
def prior_trainable(kernel_size, bias_size=0, dtype=None):
    n = kernel_size + bias_size
    return tf.keras.Sequential([
        tfp.layers.VariableLayer(n, dtype=dtype),
        tfp.layers.DistributionLambda(lambda t: tfd.Independent(
            tfd.Normal(loc=t, scale=1),
            reinterpreted_batch_ndims=1)),
    ])
def posterior_mean_field(kernel_size, bias_size=0, dtype=None):
    n = kernel_size + bias_size
    c = np.log(np.expm1(1.))
    return tf.keras.Sequential([
        tfp.layers.VariableLayer(2 * n, dtype=dtype),
        tfp.layers.DistributionLambda(lambda t: tfd.Independent(
            tfd.Normal(
                loc=t[..., :n],
                scale=1e-5 + tf.nn.softplus(c + t[..., n:])),
            reinterpreted_batch_ndims=1)),
    ])

```

Please note DenseVariational.

Now to the main model, where we'll use the priors and posteriors. You'll recognize DistributionLambda. We've replaced Binomial from the previous recipe, *Diagnosing a disease*, with Normal, which will give us an estimate of the variance of predictions:

```

model = tf.keras.Sequential([
    tfp.layers.DenseVariational(2, posterior_mean_field,
    prior_trainable, kl_weight=1/X.shape[0]),
    tfp.layers.DistributionLambda(
        lambda t: tfd.Normal(
            loc=t[..., :1],

```

```

        scale=1e-3 + tf.math.softplus(0.01 * t[...,1:])
    )
),
])
model.compile(
    optimizer=tf.optimizers.Adam(learning_rate=0.01),
    loss=negloglik
)
callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss',
patience=5)
model.fit(
    X_train,
    y_train,
    validation_data=(X_test, y_test),
    epochs=1000,
    verbose=False,
    callbacks=[callback]
)

```

After fitting, we apply the model to our test dataset and obtain predictions for it.

3. Validation:

Let's check how good our model is:

```

from sklearn.metrics import roc_auc_score
preds = model(X_test)
roc_auc_score(y_test, preds.mean().numpy())

```

We get around 70% AUC. Since this summary figure often doesn't tell a full story, let's look at the confusion matrix as well:

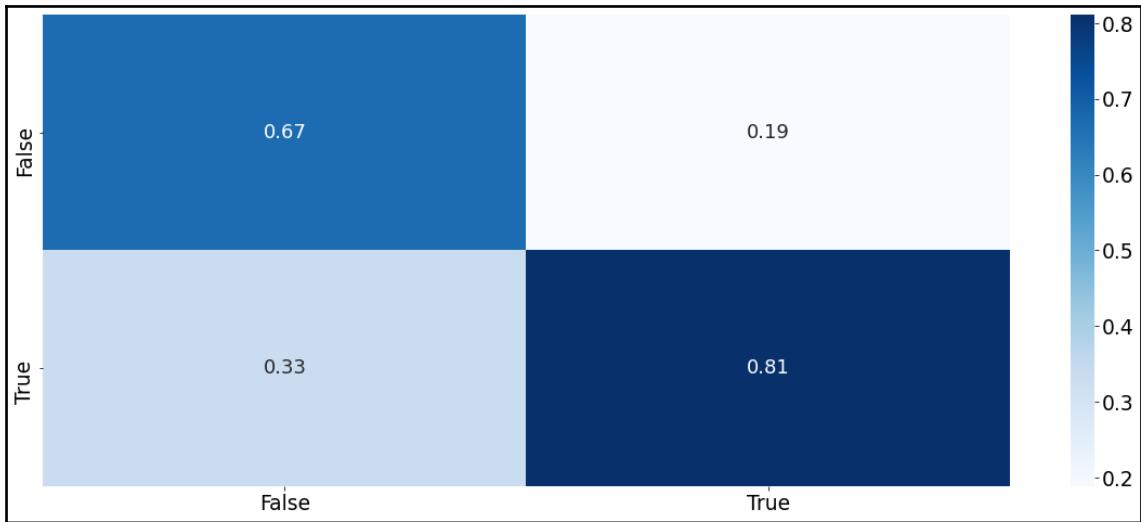
```

from sklearn.metrics import confusion_matrix
import pandas as pd
import seaborn as sns

cm = confusion_matrix(y_test, preds.mean().numpy() >= 0.5)
cm = pd.DataFrame(
    data=cm / cm.sum(axis=0),
    columns=['False', 'True'],
    index=['False', 'True']
)
sns.heatmap(
    cm,
    fmt='.2f',
    cmap='Blues',
    annot=True,
    annot_kws={'fontsize': 18}
)

```

This code gives us the following confusion matrix:



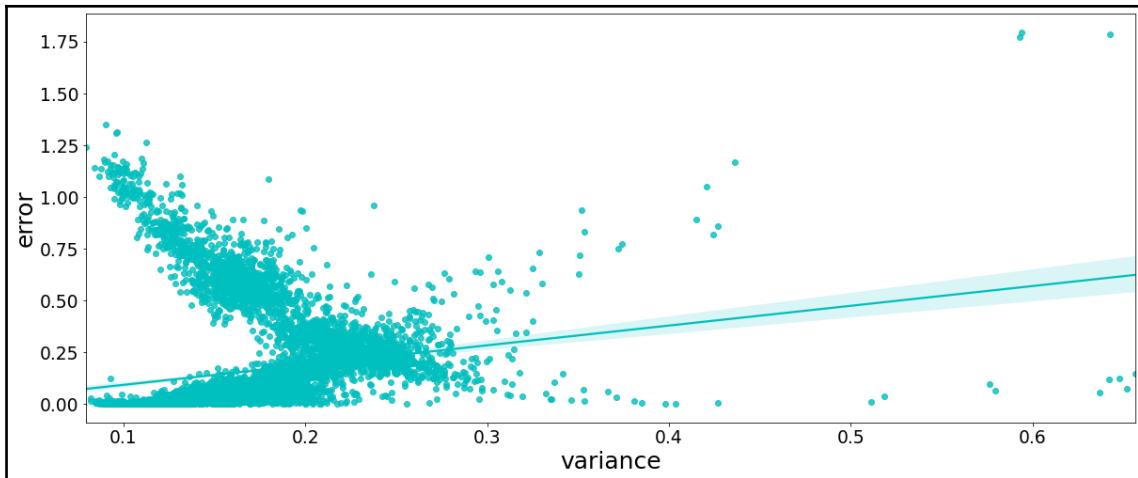
This confusion matrix tabulates predictions of default against the actual defaults. In the diagonal of the points, false-false and true-true are the correct predictions (true positives and true negatives). We can see that the number of correct predictions are higher than the false predictions, so that's comforting.

However, the most interesting point is that variances of predictions correlate with errors!

```
import scipy
scipy.stats.spearmanr(np.abs(y_test -
preds.mean().numpy().squeeze()),
preds.variance().numpy().squeeze())
```

We get a rank correlation of about 60% between absolute errors and the variance of predictions.

This gives us a confidence estimate that can be very useful in practice. We find that the variance is higher for test points where the error is high and that it is lower where we expect the error to be lower. We can look at absolute errors and variance in a scatter plot:



This concludes our recipe. This recipe is as an exercise for you to try better preprocessing, tweaking the model more, or switching the distributions.

How it works...

Models for credit scoring often use logistic regression models, which we've encountered in the *Predicting stock prices with confidence* recipe in this chapter. Alternatively, boosted models or interpretable decision trees are also in use. Given the ability to do online learning and to represent residual uncertainties, tensorflow-probability offers itself as another practical alternative.

In this recipe, we've created a probabilistic credit default prediction model that works with epistemic uncertainty. It's time to explain what that means.

Epistemic uncertainty

This is the uncertainty related to incomplete information – in other words, the uncertainty inherent in the model. We come across epistemic uncertainty all the time with noisy real-world datasets.

In TensorFlow Probability, this can be modeled as weight uncertainty. In this recipe, we've used a Bayesian neural network, where weights are a probability distribution rather than scalar estimates. This weight uncertainty translates to uncertainty in predictions, which is what we want to see.

As our final network layer, we included stochasticity from a normal distribution to model aleatoric uncertainty. Instead, we could have assumed that this variability was known.

See also

There are other routes to explore as well, such as libraries or additional material, which we will list here.

You can find similar problems online, such as the following:

- Credit scoring by predicting financial distress: <https://www.kaggle.com/c/GiveMeSomeCredit/data>.
- Lending Club provides a huge dataset of loan applications: <https://www.lendingclub.com/info/download-data.action>.
- Claim severity prediction for insurance: <https://www.kaggle.com/c/allstate-claims-severity/data>.

We couldn't use these here because of copyright restrictions.

As for libraries, we recommend you have a look at these:

- **risk-slim** is a Python library for customizable risk scores: <https://github.com/ustunb/risk-slim>.
- **scorecardpy** is a library for scorecard development: <https://github.com/ShichenXie/scorecardpy>.

As for tutorials, the Open Risk Manual offers open resources for credit scoring in Python: https://www.openriskmanual.org/wiki/Credit_Scoring_with_Python.

NumPyro provides a tutorial about Bayesian regression for divorce rates: http://pyro.ai/numpyro/bayesian_regression.html#Regression-Model-to-Predict-Divorce-Rate.

5

Heuristic Search Techniques and Logical Inference

In this chapter, we will introduce a broad range of problem-solving tools. We will start by looking at ontologies and knowledge-based reasoning before moving on to optimization in the context of **Boolean satisfiability (SAT)** and combinatorial optimization, where we'll simulate the result of individual behavior and coordination in society. Finally, we'll implement Monte Carlo tree search to find the best moves in chess.

We'll be dealing with various techniques in this chapter, including logic solvers, graph embeddings, **genetic algorithms (GA)**, **particle swarm optimization (PSO)**, SAT solvers, **simulated annealing (SA)**, ant colony optimization, multi-agent systems, and Monte Carlo tree search.

In this chapter, we will cover the following recipes:

- Making decisions based on knowledge
- Solving the n-queens problem
- Finding the shortest bus route
- Simulating the spread of a disease
- Writing a chess engine with Monte Carlo tree search

Let's get started!

Making decisions based on knowledge

When a lot of background knowledge is available about a topic, why not use it when making decisions? This is called a knowledge-based system. Inference engines in expert systems and unification, as done in logic solvers, are examples of this.

Another way to retrieve knowledge when making decisions is based on representing knowledge in a graph. Every node in the graph represents a concept, while every edge represents a relationship. Both can be embedded and represented as numerical features that express their location with respect to the other elements of the graph.

In this recipe, we'll go through two examples for each of these possibilities.



From Aristotle to Linnaeus to today's mathematicians and physicists, people have tried to put order into the world by categorizing objects into a systematic order, called taxonomy. Mathematically, taxonomies are expressed as graphs, which represent information as tuples (s, o) , in that subject s which is connected to object o ; or triplets (s, p, o) , in that s is related to (a predicate of) p to o . A frequently used type of taxonomy is the ISA taxonomy, where relationships are of the type $\text{is-}a$. For example, a car is a vehicle, and a plane is also a vehicle.

Getting ready

In this recipe, we'll use a logic solver interfaced from the `nltk` (**natural language toolkit**) library from Python, and then use the graph libraries known as `networkx` and `karateclub`.

The `pip` command you'll need to use to download these libraries is as follows:

```
pip install nltk karateclub networkx
```

For the second part of this recipe, we'll also need to download the zoo dataset from Kaggle, which is available at <https://www.kaggle.com/uciml/zoo-animal-classification>.

How to do it...

As we explained in the introduction to this recipe, we'll look at two different problems from two different approaches.

We'll start with logical reasoning using a logic solver.

Logical reasoning

In this part of this recipe, we'll look at a simple example of logical reasoning using libraries bundled with the `nltk` library. There are many other ways to approach logical inference, some of which we'll look at in the *See also...* section at the end of this recipe.

We'll use a very simple toy problem that you could find in any *101 – Introduction to Logic* book, though a more complex approach to such problems could be taken.

Our problem is well-known: if all men are mortal, and Socrates is a man, is Socrates mortal?

We can express this very naturally in `nltk`, as shown here:

```
from nltk import *
from nltk.sem import Expression

p1 = Expression.fromstring('man(socrates)')
p2 = Expression.fromstring('all x.(man(x) -> mortal(x))')
c = Expression.fromstring('mortal(socrates)')
ResolutionProver().prove(c, [p1, p2], verbose=True)
```

The preceding code gives us the following output:

```
[1] {-mortal(socrates)}      A
[2] {man(socrates)}         A
[3] {-man(z2), mortal(z2)} A
[4] {-man(socrates)}        (1, 3)
[5] {mortal(socrates)}      (2, 3)
[6] {}                      (1, 5)
True
```

The reasoning provided by the solver can also be read naturally, so we won't explain this here. We'll learn how this works internally in the *How it works...* section.

Next, we'll look at knowledge embedding.

Knowledge embedding

In this part of this recipe, we'll try to make use of how information is interrelated by embedding it into a multidimensional space that can serve as part of featurization.

Here, we'll load the data, preprocess it, embed it, and then test our embedding by classifying species, given their new features. Let's get started:

1. **Dataset loading and preprocessing:** First, we'll load the zoo dataset into pandas, as we've done many times already. Then, we'll make sure that the binary columns are represented as `bool` instead of `int`:

```
import pandas as pd
zoo = pd.read_csv('zoo.csv')
binary_cols = zoo.columns[zoo.nunique() == 2]
for col in binary_cols:
    zoo[col] = zoo[col].astype(bool)
labels = [
    'Mammal', 'Bird', 'Reptile',
    'Fish', 'Amphibian', 'Bug',
    'Invertebrate'
]
training_size = int(len(zoo) * 0.8)
```

The zoo dataset contains 101 animals, each with features describing whether it, for example, has hair or produces milk. Here, the target class is the biological class of the animal.

2. **Graph embedding:** The `get_triplet()` function returns triplets for binary and integer elements in the format (s, p, o) . Note that we create triplets from the full dataset, rather than just the training dataset. However, to avoid target leakage, we don't create triplets from the target for data points outside the training set:

```
all_labels = { i+1: c for i, c in enumerate(labels) }
cols = list(zoo.columns)

triplets = []
def get_triplet(row, col):
    if col == 'class_type':
        return (
            all_labels[row[col]],
            'is_a',
            row['animal_name'],
        )
    # int properties:
    if col in ['legs']:
        #if row[col] > 0:
        return (
            row['animal_name'],
            'has' + col,
            str(row[col]) + '_legs'
        )
```

```
#else:  
#    return ()  
# binary properties:  
if row[col]:  
    return (  
        row['animal_name'],  
        'has',  
        str(col)  
    )  
else:  
    return ()  
  
for i, row in zoo.iterrows():  
    for col in cols:  
        if col == 'animal_name':  
            continue  
        if col == 'class_type' and i > training_size:  
            continue  
        triplet = get_triplet(row, col)  
        if triplet:  
            triplets.append(triplet)
```

The preceding code will create our triplets. Let's take a look at some of them to get an idea of what they look like. The following are the first 20 entries we get; we used `triplets[:20]` to obtain them:

```
[('aardvark', 'has', 'hair'),  
 ('aardvark', 'has', 'milk'),  
 ('aardvark', 'has', 'predator'),  
 ('aardvark', 'has', 'toothed'),  
 ('aardvark', 'has', 'backbone'),  
 ('aardvark', 'has', 'breathes'),  
 ('aardvark', 'haslegs', '4_legs'),  
 ('aardvark', 'has', 'catsize'),  
 ('Mammal', 'is_a', 'aardvark'),  
 ('antelope', 'has', 'hair'),  
 ('antelope', 'has', 'milk'),  
 ('antelope', 'has', 'toothed'),  
 ('antelope', 'has', 'backbone'),  
 ('antelope', 'has', 'breathes'),  
 ('antelope', 'haslegs', '4_legs'),  
 ('antelope', 'has', 'tail'),  
 ('antelope', 'has', 'catsize'),  
 ('Mammal', 'is_a', 'antelope'),  
 ('bass', 'has', 'eggs'),  
 ('bass', 'has', 'aquatic')]
```

The preceding code block shows a few examples of the resultant triplets. In total, we got 842 triplets out of 101 rows.

Now, we can load this dataset into a graph using the `networkx` API:

```
import networkx as nx

class Vocabulary:
    label2id = {}
    id2label = {}
    def lookup(self, word):
        """get word id; if not present, insert"""
        if word in self.label2id:
            return self.label2id[word]
        ind = len(self.label2id)
        self.label2id[word] = ind
        return ind
    def inverse_lookup(self, index):
        if len(self.id2label) == 0:
            self.id2label = {
                ind: label
                for label, ind in self.label2id.items()
            }
        return self.id2label.get(index, None)
vocab = Vocabulary()
nx_graph = nx.Graph()
for (a, p, b) in triplets:
    id1, id2 = vocab.lookup(a), vocab.lookup(b)
    nx_graph.add_edge(id1, id2)
```

The `Vocabulary` class is a wrapper for the `label2id` and `id2label` dictionaries. We need this because some graph embedding algorithms don't accept string names for nodes or relationships. Here, we converted the concept labels into IDs before storing them in the graph.

Now, we can embed the graph numerically with different algorithms. We'll use `Walklets` here:

```
from karateclub.node_embedding.neighbourhood import Walklets

model_w = Walklets(dimensions=5)
model_w.fit(nx_graph)
embedding = model_w.get_embedding()
```

The preceding code shows that every concept in the graph will be represented by a 5-dimensional vector.

Now, we can test whether these features are useful for predicting the target (the animal):

```
trainamals = [
    vocab.label2id[animal]
    for animal in zoo.animal_name.values[:training_size]
]
testimals = [
    vocab.label2id[animal]
    for animal in zoo.animal_name.values[training_size:]
]
clf = SVC(random_state=42)
clf.fit(embedding[trainamals, :], zoo.class_type[:training_size])

test_labels = zoo.class_type[training_size:]
test_embeddings = embedding[testimals, :]
print(end='Support Vector Machine: Accuracy: ')
print('{:.3f}'.format(
    accuracy_score(test_labels, clf.predict(test_embeddings))
))
print(confusion_matrix(test_labels, clf.predict(test_embeddings)))
```

The output looks as follows:

```
Support Vector Machine: Accuracy = 0.809
[[5 0 0 0 0 0]
 [0 4 0 0 0 0]
 [2 0 0 1 0 0 0]
 [0 0 0 3 0 0 0]
 [1 0 0 0 0 0 0]
 [0 0 0 0 2 0]
 [0 0 0 0 0 3]]
```

This looks quite good, though the technique only becomes really interesting if we have a knowledge base that goes beyond our training set. It is hard to show graph embedding without loading millions of triplets or huge graphs. We'll mention a few large knowledge bases in the following section.

How it works...

In this section, we'll look at the basic concepts behind this recipe, as well as its corresponding methods. First, we'll cover logical reasoning and logic provers, before looking at knowledge embedding and graph embedding with Walklets.

Logical reasoning

Logical reasoning is a term that bridges logical inference techniques such as deduction, induction, and abduction. **Abductive reasoning**, which is often used in expert systems, is the process of examining the available observations and deriving possible conclusions (the **best explanation**) from them.



An expert system is a reasoning system that emulates the decision-making abilities of human experts. Expert systems are designed to solve complex problems by reasoning through bodies of knowledge, represented mainly as if-then-else rules (this is called a **knowledge base**).

Inductive reasoning is about determining a conclusion while following the initial premises and a rule. In **deductive reasoning**, we infer a rule from observations.

To apply logical reasoning, a linguistic statement has to be encoded as a well-formed logical formula so that we can apply logical calculus. A well-formed formula can contain the following entities:

- Predicate symbols such as P
- The equality symbol, $=$
- Negation, \neg
- Binary connectives such as \rightarrow
- Quantifiers such as \forall (for all) and \exists (there exists).

For example, the reasoning that *Socrates is a man. Man is mortal. Therefore, Socrates is mortal*, can be expressed as a logical statement using propositional logic, as follows:

$$\frac{\begin{array}{l} 1. \quad Ps \\ 2. \quad \forall x[Px \rightarrow Qx] \end{array}}{\therefore Qx}$$

Next, we'll look at logic provers.

Logic provers

Automated theorem proving is a wide field that includes work based on logical theorems and mathematical formulas. We've already looked at the problem of proving a first-order logic equation that consists of logical equations. A search algorithm is combined with logical equations so that the satisfiability of a propositional formula can be decided on (see the *Solving the n-queens problem* recipe in this chapter), as well as the validity of a sentence, given a set of axioms. The *Resolution Theorem Prover* in `nltk` provides other functionality, such as unification, subsumption, and **question answering (QA)**: <http://www.nltk.org/howto/resolution.html>.

In the next subsection, we'll look at knowledge embedding.

Knowledge embedding

A **knowledge embedding (KE)** refers to the distributed representations of concepts derived from their relationships. These are often represented in a **knowledge graph (KG)**.

A well-known example of a KG is **WordNet** (G.A. Miller and others, *WordNet: An online lexical database*; 1990), which provides synonyms, hypernyms, and other expansions of words, similar to a thesaurus, and is available with different GUIs and an appropriate command line for all major operating systems. WordNet is available in more than 200 languages, and each word (synset) is related to other words over directed semantic relationships, such as hypernyms or hyponyms, meronyms or holonyms, and others.

KGs can be used in **natural language processing (NLP)** applications to support decision-making, where they can be used effectively as lookup engines or for reasoning.

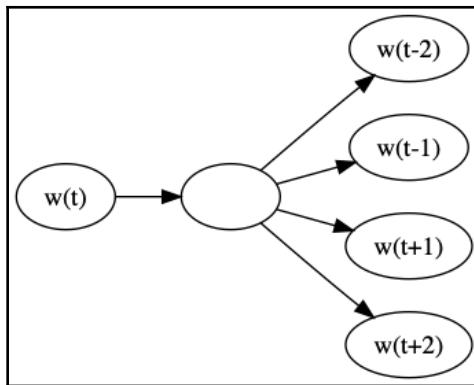
Knowledge embeddings are low-dimensional representations of concept relationships and can be extracted using embedding or more generic dimensionality reduction methods. In the next subsection, we'll look at the Walklet embedding method.

Graph embedding with Walklets

The Walklet algorithm basically applies the Word2Vec skipgram algorithm to vertices in a graph, so instead of embeddings of words (the original application of Word2Vec), we'll get embeddings of concepts based on their connections. The Walklet algorithm subsamples short random walks on the vertices of a graph, as paths, which are then passed to the shallow neural network (see the following diagram) for skipgram training.

The skipgram algorithm (Mikolov and others 2013; <https://arxiv.org/abs/1301.3781>) predicts the context of words (that is, vertices) based on the word itself. Each word is featurized as a continuous bag of words vector (in practice, each word gets indexed in a dictionary that we use), and we predict the indexes of the surrounding words (concepts) based on hidden layer projection. The dimensionality of this hidden layer's projection and the window size of the context are the main parameters of the algorithm. After training, we use the hidden layer as the embedding.

The following diagram illustrates the skipgram network architecture, which contains an input layer, a hidden layer, and an output layer of word predictions:



$w(t)$ refers to the current word (or concept), while $w(t-2)$, $w(t-1)$, $w(t+1)$, and $w(t+2)$ refer to two words before and after, respectively. We predict the word context based on the current word. As we've already mentioned, the size of the context (window size) is a hyperparameter of the skipgram algorithms.

A related algorithm is the **continuous bag-of-words algorithm (CBOW)**, where the architecture is inverted - we predict a single word based on the context. Both are based on the hypothesis that words that co-occur have related meaning or that they have distributional similarity, which implies that they are similar in terms of their meaning. This is called a **distributional hypothesis** (Harris, 1954, *Distributional structure*).

The Walklet algorithm performs well on large graphs and – since it's a neural network – can be trained online. You can find out more about Walklets in the 2017 paper by Brian Perozzi and others, *Don't Walk, Skip! Online Learning of Multi-scale Network Embeddings* (<https://arxiv.org/abs/1605.02115>).

See also

The following are libraries that can be used for logical inference in Python:

- SymPy: <https://docs.sympy.org/latest/modules/logic.html>
- Kanren logic programming: <https://github.com/logpy/logpy>
- PyDatalog: <https://sites.google.com/site/pydatalog/>

We've been following the inference guide in `nltk` for this recipe. You can find more tools at the official `nltk` website: <http://www.nltk.org/howto/inference.html>.

Some other libraries for graph embedding are as follows:

- KarateClub: <https://karateclub.readthedocs.io/en/latest/index.html>
- pykg2vec: <https://github.com/Sujit-O/pykg2vec>
- PyTorch BigGraph (by Facebook Research): <https://github.com/facebookresearch/PyTorch-BigGraph>
- GraphVite: <https://graphvite.io/>
- AmpliGraph (by Accenture): <https://docs.ampligraph.org/>
- pyRDF2Vec: <https://github.com/IBCNServices/pyRDF2Vec>

KarateClub, which is maintained by Benedek Rozemberczki, a PhD student at the University of Edinburgh, contains many implementations of unsupervised graph embedding algorithms.

Some graph libraries provide link prediction as well. This means that, for a given set of nodes, you can infer whether a relationship to other nodes exists. A review of link prediction can be found in *Knowledge Graph Embedding for Link Prediction: A Comparative Analysis*, by Andrea Rossi and others. (2020; <https://arxiv.org/abs/2002.00819>).

Some resources for reasoning about the real world and/or with common sense are as follows:

- ActionCores: <http://www.actioncores.org/apidoc.html#pracinference>
- KagNet: <https://github.com/INK-USC/KagNet>
- Allen AI Commonsense Knowledge Graphs: <https://mosaic.allenai.org/projects/commonsense-knowledge-graphs>
- Commonsense Reasoning Problem Page at NYU CS: http://commonsensereasoning.org/problem_page.html

Learning on graphs: Open Graph Benchmark: Datasets for Machine Learning on Graphs, Hu and others, 2020 (<https://arxiv.org/pdf/2005.00687.pdf>) is another reference for graph embedding with machine learning.

There are several large real-world knowledge databases available, such as the following:

- Wikidata: <https://www.wikidata.org/>
- Conceptnet5: <https://github.com/commonsense/conceptnet5>
- The Open Multilingual Wordnet: <http://compling.hss.ntu.edu.sg/omw/>
- Yago: <https://github.com/yago-naga/yago3>

Solving the n-queens problem

In mathematical logic, satisfiability is about whether a formula can be valid under some interpretation (parameters). We say that a formula is unsatisfiable if it can't be true under any interpretation. A **Boolean satisfiability problem**, or **SAT**, is all about whether a Boolean formula is valid (satisfiable) under any of the values of its parameters. Since many problems can be reduced to SAT problems, and solvers and optimizations for it exist, it is an important class of problems.



SAT problems have been proven to be NP-complete. NP-completeness (short for **nondeterministic polynomial time**) means that a solution to a problem can be verified in polynomial time. Note that this doesn't mean that a solution can be found quickly, only that a solution can be verified quickly. NP-complete problems are often approached with search heuristics and algorithms.

In this recipe, we'll address a SAT problem in various ways. We'll take a relatively simple and well-studied case known as the n-queens problem, where we try to place queens on a chessboard of n by n squares so that any column, row, and diagonal can only take, at most, one queen.

First, we'll apply a GA, then PSO, and then a specialized SAT solver.

Getting ready

We'll be using the `dd` solver for one of the approaches in this recipe. To install it, we also need the `omega` library. We can get both by using the `pip` command, as follows:

```
pip install dd omega
```

We'll use the dd SAT solver libraries later, but first, we'll look at some other algorithmic approaches.

How to do it...

We'll start with the GA.

Genetic algorithm

First, we'll define how a chromosome is represented and how it can mutate. Then, we'll define a feedback loop for testing these chromosomes and changing them. We'll explain the algorithm itself in the *How it works...* section, toward the end of this recipe. Let's get started:

1. **Representing a solution** (a chromosome): An object-oriented style lends itself to defining chromosomes. Let's look at our implementation. First, we need to know what a chromosome is and what it does:

```
import random
from typing import Optional, List, Tuple

class Chromosome:
    def __init__(self, configuration: Optional[List]=None, nq: Optional[int]=None):
        if configuration is None:
            self.nq = nq
            self.max_fitness = np.sum(np.arange(nq))
            self.configuration = [
                random.randint(1, nq) for _ in range(nq)
            ]
        else:
            self.configuration = configuration
            self.nq = len(configuration)
            self.max_fitness = np.sum(np.arange(self.nq))
    def fitness(self):
        return cost_function(self.configuration) / self.max_fitness

    def mutate(self):
        ind = random.randint(0, self.nq-1)
        val = random.randint(1, self.nq)
        self.configuration[ind] = val
```

The preceding code creates our basic data structure, which contains a candidate solution that can replicate and mutate. This code refers to a cost function.

We need a cost function so that we know how to fit our genes:

```
def cost_function(props):
    res = 0
    for i1, q1 in enumerate(props[:-1]):
        for i2, q2 in enumerate(props[i1+1:], i1+1):
            if (q1 != q2) and (abs(i1 - i2) != abs(q1 - q2)):
                res += 1
    return res
```

We can select genes based on this cost function (see the `fitness()` method).

2. Writing the main algorithm: The GA for the n-queens problem is as follows (we've omitted the visualization here):

```
class GeneticQueen:
    def __init__(self, nq, population_size=20, mutation_prob=0.5):
        self.nq = nq
        self.population_size = population_size
        self.mutation_prob = mutation_prob
        self.population = [Chromosome(nq=nq) for _ in
                           range(population_size)]
        self.solution = None
        self.best_fitness = None

    def iterate(self):
        new_population = []
        best_fitness = -1
        for i in range(len(self.population)):
            p1, p2 = self.get_parents()
            child = Chromosome(self.cross_over(p1, p2))
            if random.random() < self.mutation_prob:
                child.mutate()
            new_population.append(child)
            fit = child.fitness()
            if fit > best_fitness:
                best_fitness = fit
            if fit == 1:
                self.solution = child
                break
        self.best_fitness = best_fitness
        self.population = new_population
    def cross_over(self, p1, p2):
        return [
            yi
            if random.random() > 0
            else xi
            for xi, yi in zip(
```

```

                p1.configuration,
                p2.configuration
            )
        ]
    def get_parents(self) -> Tuple[Chromosome, Chromosome]:
        weights = [chrom.fitness() for chrom in self.population]
        return tuple(
            random.choices(
                self.population,
                weights=weights,
                k=2
            )
        )

```

This class contains the population of chromosomes and can have methods applied to it (population control, if you like), such as what parents to use (`get_parents()`) and mating them (`cross_over()`). Take note of the `iterate()` method, which is where the main logic is implemented. We'll comment on the main decisions we've made here in the *How it works...* section.

3. **Running the algorithm:** We execute our algorithm by simply instantiating a `GeneticQueen` and calling `iterate()`. We can also put in a few extra lines to get regular updates and to collect fitness data over time. We then run the algorithm, as follows:

```

def ga_solver(nq):
    fitness_trace = []
    gq = GeneticQueen(nq=nq)
    generation = 0
    while not gq.solution:
        gq.iterate()
        if (generation % 100) == 0:
            print('Generation {}'.format(generation))
            print('Maximum Fitness: {:.3f}'.format(gq.best_fitness))
            fitness_trace.append(gq.best_fitness)
        generation += 1

    gq.visualize_solution()
    return fitness_trace

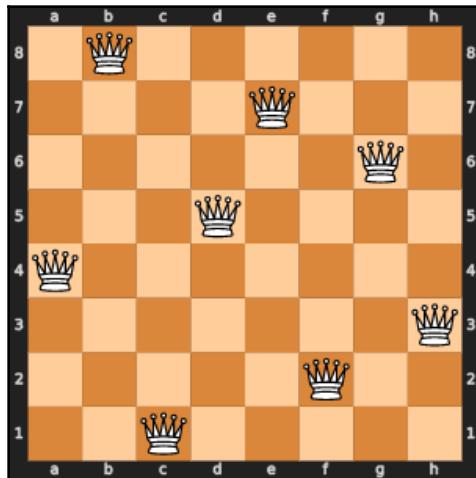
```

Finally, we can visualize the solution.

If we run the preceding code, we'll get a single run that looks like this (yours may look different):

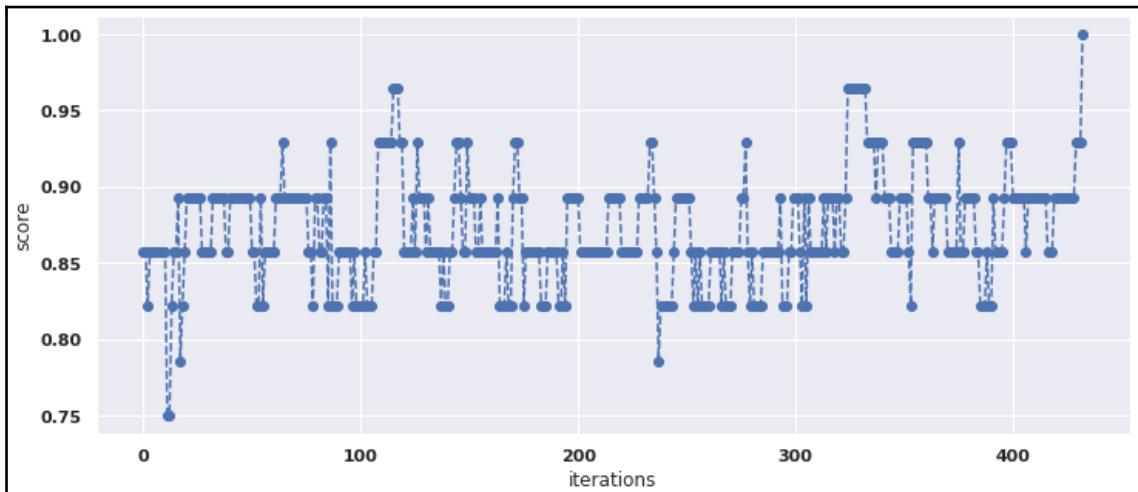
```
Generation 0
Maximum Fitness: 0.857
Generation 100
Maximum Fitness: 0.821
Generation 200
Maximum Fitness: 0.892
Generation 300
Maximum Fitness: 0.892
Generation 400
Maximum Fitness: 0.892
```

The preceding code gives us the following output:



This took close to 8 seconds to complete.

The following plot shows the fitness of the best chromosome at each iteration of the algorithm:



Here, we can see that the fitness of the algorithm doesn't always improve; it can also go down. We could have chosen to keep the best chromosome here. In that case, we wouldn't have seen any decline (but the potential downside is that we could have ended up in a local minimum).

Now, let's move on to PSO!

Particle swarm optimization

In this part of this recipe, we'll be implementing a PSO algorithm for the n-queens problem from scratch. Let's get started:

- 1. Representing a solution:** Similar to the GA, we need to define what a solution looks like. In PSO, this means we define a particle:

```
class Particle:
    best_fitness: int = 0
    def __init__(self, N=None, props=None,
                 velocities=None):
        if props is None:
            self.current_particle = np.random.randint(0, N-1, N)
            self.best_state = np.random.randint(0, N-1, N)
```

```

        self.velocities = np.random.uniform(-(N-1), N-1, N)
    else:
        self.current_particle = props
        self.best_state = props
        self.velocities = velocities
        self.best_fitness = cost_function(self.best_state)
def set_new_best(self, props: List[int], new_fitness: int):
    self.best_state = props
    self.best_fitness = new_fitness
def __repr__(self):
    return f'{self.__class__.__name__}(\n' +\
           f'\tcurrent_particle={self.current_particle}\n' +\
           f'\tbest_state={self.best_state}\n' +\
           f'\tvelocities={self.velocities}\n' +\
           f'\tbest_fitness={self.best_fitness}\n' +\
           ')'

```

This is the main data structure that we'll be working with. It contains a candidate solution. Applying PSO will involve changing a bunch of these particles. We'll explain how `Particle` works in more detail in the *How it works...* section.

We are going to use the same cost function that we defined for the GA. This cost function tells us how well our particles fit the given problem – in other words, how good a property vector is.

We'll wrap our initialization and the main algorithm into a class:

```

class ParticleSwarm:
    def __init__(self, N: int, n_particles: int,
                 omega: float, phip: float, phig: float
                 ):
        self.particles = [Particle(N=N) for i in range(n_particles)]
        self.omega = omega
        self.phip = phip
        self.phig = phig

    def get_best_particle(self):
        best_particle = 0
        best_score = -1
        score = -1
        for i, particle in enumerate(self.particles):
            score = cost_function(particle.current_particle)
            if score > best_score:
                best_score = score
                best_ind = i
        return self.particles[best_ind].current_particle, best_score

```

```

def iterate(self):
    for particle in self.particles:
        rg = np.random.rand((N))
        rp = np.random.rand((N))
        delta_p = particle.best_state - particle.current_particle
        delta_g = best_particle - particle.current_particle
        update = (rp * self.phip * delta_p +
                  \ rg * self.phig * delta_g) # local vs global
        particle.velocities = self.omega * particle.velocities +
        update
        particle.current_particle = (np.abs(
            particle.current_particle + particle.velocities
        ) % N ).astype(int) # update the particle best
        current_fitness = cost_function(particle.current_particle)
        if current_fitness > particle.best_fitness:
            particle.set_new_best(
                particle.current_particle, current_fitness
            )
        particle_candidate, score_candidate =
        get_best_particle(particles)
        if best_score_cand > best_score:
            best_particle = particle_candidate
            best_score = score_candidate
    return best_particle, best_score

```

The `get_best_particle()` method returns the best configuration and the best score. Take note of the `iterate()` method, which updates our particles and returns the best particle, along with its score. Details regarding this update are provided in the *How it works...* section. The optimization process itself is done using a few formulas that are relatively simple.

We'll also want to display our solutions. The code for showing the board positions is as follows:

```

import chess
import chess.svg
from IPython.display import display

def show_board(queens):
    fen = '/'.join([queen_to_str(q) for q in queens])
    display(chess.svg.board(board=chess.Board(fen), size=300))

```

The following is the main algorithm for PSO:

```
def particle_swarm_optimization(
    N: int, omega: float, phip: float, phig: float,
    n_particles: int, visualize=False, max_iteration=999999
) -> List[int]:
    def print_best():
        print(f'iteration {iteration} - best particle: {best_particle},'
              f'score: {best_score}')
    solved_cost = np.sum(np.arange(N))
    pso = ParticleSwarm(N, n_particles, omega, phip, phig)
    iteration = 0
    best_particle, best_score = get_best_particle(particles)
    scores = [best_score]
    if visualize:
        print('iteration:', iteration)
        show_board(best_particle)
    while best_score < solved_cost and iteration < max_iteration:
        if (iteration % 500) == 0 or iteration == 0:
            print_best()
            best_particle, best_score = pso.iterate()
        if iteration > 0 and visualize:
            print('iteration:', iteration)
            show_board(best_particle)
        scores.append(best_score)
        iteration += 1
    print_best()
    return best_particle, scores
```

Similar to what we did in the case of the GA, we track how well our solutions do over the iterations (via our cost function). The main function returns the following:

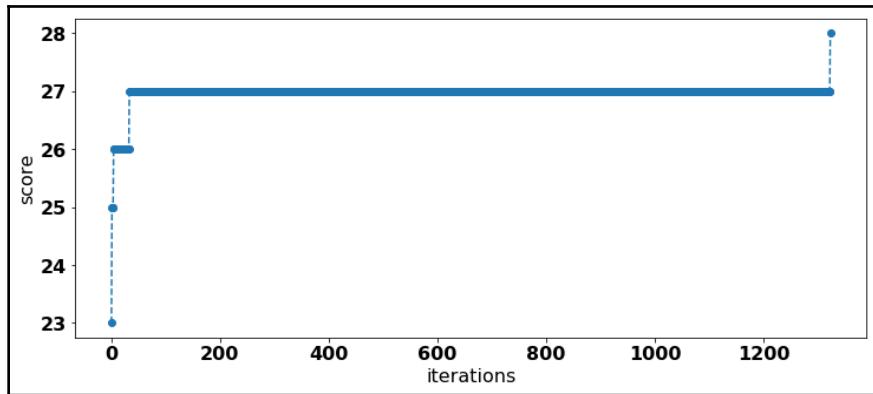
- `best_particle`: The best solution
- `scores`: The best scores over our iterations

As we mentioned previously, we'll explain how all of this works in the *How it works...* section.

You can view the output of the algorithm being run with $n = 8$ at <https://github.com/PacktPublishing/Artificial-Intelligence-with-Python-Cookbook/blob/master/chapter05/solving-n-queens.md>.

We are using the chess library for visualization here.

In the following plot, you can see the quality of the solutions over our iterations:



Since all the particles maintain their own records of the best solution, the score can never decline. At iteration 1,323, we reached a solution and the algorithm stopped.

SAT solver

This is heavily based on the example that can be found in the `dd` library, copyright of California Institute of Technology, at <https://github.com/tulip-control/dd/blob/0f6d16483cc13078edebac9e89d1d4b99d22991e/examples/queens.py>.

In a modern SAT solver in Python, we can define our constraints as simple functions.

Basically, there's one formula that incorporates all the constraints. Once all the constraints have been satisfied (or the conjunction of all the constraints), the solution is found:

```
def queens_formula(n):
    present = at_least_one_queen_per_row(n)
    rows = at_most_one_queen_per_line(True, n)
    cols = at_most_one_queen_per_line(False, n)
    slash = at_most_one_queen_per_diagonal(True, n)
    backslash = at_most_one_queen_per_diagonal(False, n)
    s = conj([present, rows, cols, slash, backslash])
    return s
```

Here's the constraint for at_least_one_queen_per_row:

```
def at_least_one_queen_per_row(n):
    c = list()
    for i in range(n):
        xijs = [var_str(i, j) for j in range(n)]
        s = disj(xijs)
        c.append(s)
    return conj(c)
```

Here, we take disjunctions over the queens on each row.

The main run looks like this:

```
def benchmark(n):
    t0 = time.time()
    u, bdd = solve_queens(n)
    t1 = time.time()
    dt = t1 - t0
    for i, d in enumerate(bdd.pick_iter(u)):
        if len(d) > 0:
            visualize_solution(d)
            break
    n_solutions = bdd.count(u)

    s = (
        '-----\n'
        'queens: {n}\n'
        'time: {dt} (sec)\n'
        'node: {u}\n'
        'total nodes: {k}\n'
        'number solutions: {n_solutions}\n'
        '-----\n'
    ).format(
        n=n, dt=dt, u=u, k=len(bdd),
        n_solutions=n_solutions,
    )
    print(s)
    return dt
```

When we run this, we should see an example solution. We should also get some statistics regarding how many solutions were found and how long it took to find them.

The following is our example solution for the eight queens problem:

0	0	0	0	0	0	0	0	1
0	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0	0
0	1	2	3	4	5	6	7	

The textual output looks like this:

```
queens: 8
time: 4.775595426559448 (sec)
node: -250797
total nodes: 250797
number solutions: 92
```

This solver not only got all the solutions (we only visualized one of them) but was also about twice as fast as the GA!

How it works...

In this section, we'll explain the different approaches we employed in this recipe, starting with the GA.

Genetic algorithm

The GA is quite simple at heart: we maintain a set of candidate solutions (called chromosomes) and we have two operations we can use to change them:

- cross-over: Two chromosomes have children (which means they mix)
- mutation: A chromosome changes randomly

A chromosome has a candidate solution that's stored in `configuration`. When initializing a chromosome, we have to give it either the number of queens or the initial configuration. We covered what chromosomes actually are previously in this chapter. If a configuration is not given, then we need to create one with a list comprehension, such as `[random.randint(1, nq) for _ in range(nq)]`.

A chromosome can calculate its own fitness; here, we used the same cost function that we used previously, but this time, we scaled it to be between 0 and 1, where 1 means we found a solution and anything in-between shows how close we are to getting a solution. A chromosome can also mutate itself; that is, it can randomly change one of its values.

At each iteration of the algorithm, we create a new generation of chromosomes by using these two operations. The algorithm itself is straightforward:

1. First, we initialize our first generation of chromosomes with different values representing the different parameters of the solution.
2. Then, we calculate the fitness of our chromosomes. This can be done by interacting with the environment or it could be intrinsic to the solution, as in our combinatorial problem of the nine queens problem.
3. Next, we create a new generation of chromosomes, as follows:
 - Choose parents while taking their fitness into account
 - Mutate a few chromosomes according to a certain probability
4. Finally, we repeat from *Step 2* until the fitness is high enough or we have iterated a lot.

We've expressed the last step very loosely here. Basically, we can decide when the fitness is high enough and how many times we want to iterate. These are our stopping criteria.

This is very legible in our implementation of `GeneticQueen.iterate()`, so let's have another look for visualization purposes (only slightly simplified):

```
def iterate(self):  
    new_population = []  
    for i in range(len(self.population)):  
        p1, p2 = self.get_parents()  
        child = Chromosome(self.cross_over(p1, p2))  
        if random.random() < self.mutation_prob:  
            child.mutate()  
        new_population.append(child)
```

A major decision we have to make regarding the GA is whether we maintain the best solution or whether all the chromosomes (even the best) must die (potentially after giving birth to offspring). Here, each iteration creates a completely new generation.

We randomly choose parents by weighting them by their fitness, where the fittest are more likely to be chosen. The `cross-over` function in our implementation randomly decides between two of the parents for each parameter.

The main hyperparameters and major decisions that must be made for the GA are as follows:

- Population size (how many chromosomes do we have?)
- Mutation rate (how much do chromosomes change when they mutate?)
- How many (and which) chromosomes create offspring? Typically, these are the ones with the highest fitness.
- What are our stopping criteria? Typically, there's a threshold value for the algorithm's fitness and a set number of iterations.

As we can see, the GA is quite flexible and very intuitive. In the next section, we'll look at PSO.

Particle swarm optimization

We started our implementation with the `Particle` data structure. To initialize a particle, we pass the number of queens (`N`) or the vectors for our velocities and parameters.

Basically, a particle has a configuration, or a set of parameters – a vector, in this case – that fits a problem to a certain degree (`current_particle`), and a velocity (similar to a learning rate). Each property vector from a particle represents the queens' positions.

PSO then applies changes to particles in a particular way. PSO is a combination of local search and global search; that is, at each particle, we try to direct our search toward the best particle globally and the best the particle has been in the past. A particle maintains a record of its best instantiation; that is, the vector of its best parameters and the corresponding score. We also maintain the corresponding velocities of the parameters. These velocities can slow down, increase, or change direction according to the formula being used.

PSO takes a few parameters, as follows (most of these were named in our implementation; here, we're omitting the ones that are specific to our nine queens problem):

- `omega`: The decay parameter
- `phip`: Controls the contribution of the local search
- `phig`: Controls the contribution of the global search
- `n_particles`: The number of particles
- `max_iterations`: Used for early stopping without a solution

In our PSO problem, there were two deltas, `delta_p` and `delta_g`, where *p* and *g* stand for particle and global, respectively. This is because one of them is calculated with respect to the particle's historic best and the other is calculated with respect to the particle's global best.

The update is calculated according to the following code:

```
delta_p = particle.best_state - particle.current_particle
delta_g = best_particle - particle.current_particle
update = (rp * phip * delta_p +\
          rg * phig * delta_g) # local vs global
```

Here, `rp` and `rg` are random numbers and `phip` and `phig` are the local and global factors, respectively. They refer to either a unique particle or all the particles, as shown in the `delta_p` and `delta_g` variables.

There's also another parameter, `omega`, that regulates the decay of the current velocities. At each iteration, the new velocities are calculated according to the following formula:

```
particle.velocities = omega * particle.velocities + update
```

In turn, the particle parameters are incremented according to their velocities.

Note that the algorithm is sensitive to what's chosen for `phip`, `phig`, and `omega`.

Our cost function (or goodness function) calculates the score for a particle according to a given configuration of queens. This configuration is represented as a list of indexes in the range $]0, N-1[$. For each pair of queens, the function checks whether they overlap either in the diagonal, vertical, or horizontal sense. Each non-conflicting check awards a point, so the maximal number of points is $\sum_{i=1}^{N-1} i$. This is 28 for the 8 queens problem.

SAT solver

There are lots of different ways **specialized satisfiability (SAT)** solvers work. A survey by Weiwei Gong and Xu Zhou (2017) provides a broad overview of the different approaches: <https://aip.scitation.org/doi/abs/10.1063/1.4981999>.

The dd solver, which we used in our recipe, works using **binary decision diagrams (BDD)**, which were introduced by Randal Bryant (*Graph-based algorithms for Boolean function manipulation*, 1986). Binary decision diagrams (sometimes called **branching programs**) are constraints represented as Boolean functions as opposed to other encodings, such as negation normal.

In a BDD, an algorithm or a set of constraints is expressed as a Boolean function over a Boolean domain of dimension, n , that evaluates to true or false:

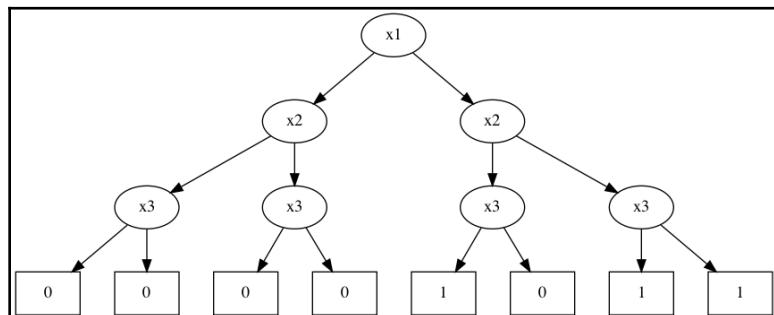
$$f : \{0, 1\}^n \rightarrow \{0, 1\}$$

This means that we can represent problems as binary trees or, equivalently, as truth tables.

To illustrate this, let's look at an example. We can enumerate all the states over our binary variables (x_1 , x_2 , and x_3) and then come up with a final state that's the result of f . The following truth table summarizes the states of our variables, as well as our function evaluation:

x_1	x_2	x_3	f
False	False	False	False
False	False	True	False
False	True	False	False
False	True	True	False
True	False	False	True
True	False	True	False
True	True	False	True
True	True	True	True

This corresponds to the following binary tree:



Binary trees and truth tables have highly optimized library implementations, which means they can run very fast. This explains how we got our results so quickly.

See also

There are lots of other SAT solvers in Python, some of which are as follows:

- Microsoft's PDP solver: <https://github.com/microsoft/PDP-Solver>
- Z3, by Microsoft Research: <https://github.com/Z3Prover/z3>
- Python bindings to picosat, developed by Continuum: <https://github.com/ContinuumIO/pycosat>

A discussion of the SAT solver, when applied to Sudoku, can be found here: <https://codingnest.com/modern-sat-solvers-fast-neat-underused-part-1-of-n/>.

An example of Z3 for the Knights and Knaves problem can be found here: <https://jamiecollinson.com/blog/solving-knights-and-knaves-with-z3/>.

Finding the shortest bus route

Finding the shortest bus route implies finding a path that connects points (bus stops) on a map. This is an instance of the traveling salesman problem. In this recipe, we'll approach the problem of finding the shortest bus route with different algorithms, including simulated annealing and ant colony optimization.

Getting ready

Apart from standard dependencies such as `scipy` and `numpy`, which we always rely on, we'll be using the `scikit-opt` library, which implements many different algorithms for swarm intelligence.



Swarm intelligence is the collective behavior of decentralized, self-organized systems that leads to the emergence of apparent intelligence in the eyes of an observer. This concept is used in work based on artificial intelligence. Natural systems, such as ant colonies, bird flocking, hawks hunting, animal herding, and bacterial growth, display a certain level of intelligence at the global level, even though ants, birds, and hawks typically exhibit relatively simple behavior. Swarm algorithms, which are inspired by biology, include the genetic algorithm, particle swarm optimization, simulated annealing, and ant colony optimization.

We can install `scikit-opt` with `pip` as follows:

```
pip install scikit-opt
```

Now, we are ready to tackle the traveling salesman problem.

How to do it...

As we mentioned previously, we'll approach our shortest bus route problem in two different ways.

First, we need to create a list of coordinates (longitude, latitude) for bus stops. The difficulty of the problem depends on the number of stops (N). Here, we've set N to 15:

```
import numpy as np
N = 15
stops = np.random.randint(0, 100, (N, 2))
```

We can also precalculate a distance matrix between stops, as follows:

```
from scipy import spatial

distance_matrix = spatial.distance.cdist(stops, stops, metric='euclidean')
```

We can feed this distance matrix into the two algorithms to save time.

We'll start with simulated annealing.

Simulated annealing

In this subsection, we'll write our algorithm for finding the shortest bus route. This is based on Luke Mile's Python implementation of simulated annealing, when applied to the traveling salesman problem: <https://gist.github.com/qpwo/a46274751cc5db2ab1d936980072a134>. Let's get started:

1. The implementation itself is short and succinct:

```
def find_tour(stops, distance_matrix, iterations=10**5):
    def calc_distance(i, j):
        """sum of distance to and from i and j in tour
        """
        return sum(
            distance_matrix[tour[k], tour[k+1]]
            for k in [j - 1, j, i - 1, i]
```

```

        )
n = len(stops)
tour = np.random.permutation(n)
lengths = []
for temperature in np.logspace(4, 0, num=iterations):
    i = np.random.randint(n - 1) # city 1
    j = np.random.randint(i + 1, n) # city 2
    old_length = calc_distance(i, j)
    # swap i and j:
    tour[[i, j]] = tour[[j, i]]
    new_length = calc_distance(i, j)
    if np.exp((old_length - new_length) / temperature) <
np.random.random(): # bad swap
        tour[[i, j]] = tour[[j, i]] # undo swap
        lengths.append(old_length)
    else:
        lengths.append(new_length)
return tour, lengths

```

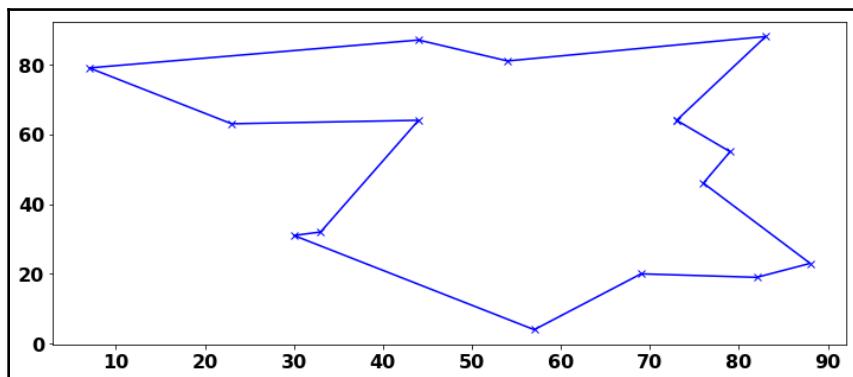
2. Next, we need to call the algorithm, as follows:

```

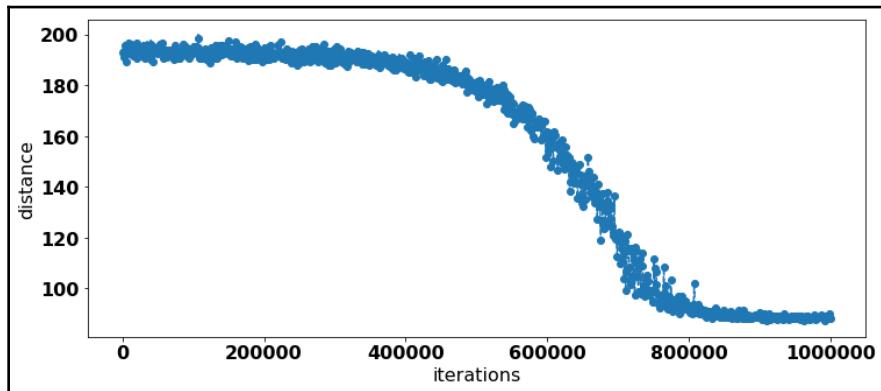
from scipy.spatial.distance import euclidean
tour, lengths = find_tour(
    stops, distance_matrix, iterations=1000000
)

```

This is the final solution – the path looks as follows:



We can also plot the internal distance measure of the algorithm. Please note how this internal cost function goes down all the time until about 800,000 iterations:



Now, let's try out ant colony optimization.

Ant colony optimization

Here, we're loading the implementation from a library. We'll explain the details of this in the *How it works...* section:

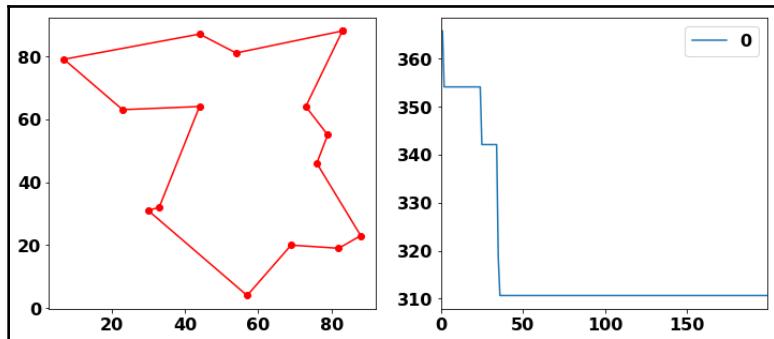
```
from sko.ACA import ACA_TSP

def cal_total_distance(tour):
    return sum([
        distance_matrix[tour[i % N], tour[(i + 1) % N]]
        for i in range(N)
    ])

aca = ACA_TSP(
    func=cal_total_distance,
    n_dim=N,
    size_pop=N,
    max_iter=200,
    distance_matrix=distance_matrix
)
best_x, best_y = aca.run()
```

We are using the distance calculations based on the point distances (`distance_matrix`) we retrieved previously.

Again, we can plot the best path and the path distance over iterations, as follows:



Once again, we can see the final path, which is the result of our optimization (the subplot on the left), as well as the distance as it goes down over iterations of the algorithm (the subplot on the right).

How it works...

The shortest bus route problem is an example of the **traveling salesman problem (TSP)**, which, in turn, is a well-known example of combinatorial optimization.



Combinatorial optimization refers to using combinatorial techniques to solve discrete optimization problems. In other words, it is the act of finding a solution among a combination of objects. Discrete, in this case, means that there are a finite number of options. The intelligence part of combinatorial optimization goes into either reducing the search space or accelerating the search. The traveling salesman problem, the minimum spanning tree problem, the marriage problem, and the knapsack problem are all applications of combinatorial optimization.

The TSP can be stated as follows: given a list of towns to visit, which is the shortest path that traverses all of them and leads back to the point of origin? The TSP has applications in domains such as planning, logistics, and microchip design.

Now, let's take a look at simulated annealing and ant colony optimization in more detail.

Simulated annealing

Simulated annealing is a probabilistic optimization technique. The name comes from metallurgy, where heating and cooling is used to reduce the defects in materials. Briefly, at each iteration, a state transition (a change) can occur. If the change is successful, the system will lower its temperature. This can be repeated until the state is good enough or until we reach a certain number of iterations.

In this recipe, we randomly initialized our city tour and then iterated for simulated annealing. The main idea of SA is that the rate of changes depends on a certain temperature. In our implementation, we decreased the temperature logically from 4 to 0. In each iteration, we tried swapping (we could have tried other operations) two random bus stops, indexes i and j in our path (tour), where $i < j$, and then we calculated the sum of distances to i from $i-1$ to i , from i to $i+1$, from $j-1$ to j , and from j to $j+1$ (see `calc_distance`). We also needed a distance measure for `calc_distance`. We chose the Euclidean distance here, but we could have chosen others.

The temperature gets factored in when we need to decide whether to accept the swap. We calculate the exponential of the difference in path length before and after the swap:

$$\exp((d_{\text{before}} - d_{\text{after}})/\text{tmp})$$

Then, we draw a random number. We accept the change if this random number is lower than our expression; otherwise, we undo it.

Ant colony optimization

As the name suggests, **ant colony optimization** is inspired by ant colonies. Let's use pheromones, which are secreted by ants as they follow a path, as an analogy: here, the agents have candidate solutions that are more attractive the closer they get to the solution.

In general, ant number k moves from state x to state y with the following probability:

$$p_{xy}^k = \frac{(\tau_{xy}^\alpha)(\eta_{xy}^\beta)}{\sum_{z \in \text{allowed}_x} (\tau_{xz}^\alpha)(\eta_{xz}^\beta)}$$

Tau is the pheromone trail that's deposited between x and y . The *eta* parameter controls the influence of the pheromone, where *eta* to the power of *beta* is the state transition (for example, one over the cost of the transition). Pheromone trails are updated according to how good the overall solution that included the state transition was.

The `scikit-opt` function does the heavy lifting here. We only have to pass a few parameters, such as the distance function, the number of points, the number of ants in the population, the number of iterations, and the distance matrix, before calling `run()`.

See also

You can also solve this problem as a mixed-integer problem. The Python-MIP library solves mixed-integer problems, and you can find an example for the TSP at <https://python-mip.readthedocs.io/en/latest/examples.html>.

The TSP can be solved with a Hopfield Network as well, as explained in this tutorial: https://www.tutorialspoint.com/artificial_neural_network/artificial_neural_network_optimization_using_hopfield.htm. A cuckoo search approach is discussed here: <https://github.com/Ashwin-Surana/cuckoo-search>.

`scikit-opt` is a powerful library for heuristic algorithms. It includes the following algorithms:

- Differential evolution
- Genetic algorithm
- Particle swarm optimization
- Simulated annealing
- Ant colony algorithm
- Immune algorithm
- Artificial fish swarm algorithm

The `scikit-opt` documentation contains more examples of solving the TSP: https://scikit-opt.github.io/scikit-opt/#/en/README?id=_22-genetic-algorithm-for-tsptravelling-salesman-problem. Another library similar to `scikit-opt` is `pyswarms`, available at <https://pyswarms.readthedocs.io/en/latest/index.html>.

As we mentioned in the introduction to this recipe, transport logistics has its own application in the TSP, even in its purest form. A dataset of 30,000 public buses, minibuses, and vans in Mexico is available at <https://thelivinglib.org/mapaton-cdmx/>.

Simulating the spread of a disease

Pandemics such as smallpox, tuberculosis, and the Black Death have affected the human population significantly throughout history. As of 2020, Covid-19 is spreading through populations all over the world, and the politics and economics of getting the virus under control with little casualties have been widely discussed.

Regarding Covid-19, to libertarians, Sweden was, for some time, the poster child for how you didn't need a lockdown, although secondary factors such as having a high proportion of single-person households and a cultural tendency to social distance weren't taken into account. Recently, fatalities in Sweden have been on the rise, and its per capita rate is one of the highest recorded (<https://www.worldometers.info/coronavirus/>).

In the UK, the initial response was to rely on herd immunity, and the lockdown was declared only weeks after other countries had already imposed it. The **National Health Service (NHS)** were using makeshift beds and renting beds in commercial hospitals because they didn't have the capacity to cope.

A **multi-agent system (MAS)** is a computer simulation consisting of participants known as agents. The individual agents can respond heuristically or based on reinforcement learning. Conjunctively, the system behavior of these agents responding to each other and to the environment can be applied to study topics, including the following:



- Cooperation and coordination
- Distributed constraint optimization
- Communication and negotiation
- Distributed problem solving, especially distributed constraint optimization

In this recipe, a relatively simple, multi-agent simulation will show you how different responses can cause a difference in the number of fatalities, and the spread, of a pandemic.

Getting ready

We'll be using the `mesa` multi-agent modeling library to implement our multi-agent simulation.

The pip command for this is as follows:

```
pip install mesa
```

Now, we are ready to go!

How to do it...

This simulation is based on work by Maple Rain Research Co., Ltd. For this recipe, we've made a few changes regarding introducing factors such as hospital beds and lockdown policies, and we've also changed how infections and active cases are accounted for. You can find the complete code at <https://github.com/benman1/covid19-sim-mesa>.



Disclaimer: This recipe's intent is not to provide medical advice, nor are we qualified medical practitioners or specialists.

First, we are going to define our agents through the Person class:

```
class Person(Agent):
    def __init__(self, unique_id, model):
        super().__init__(unique_id, model)
        self.alive = True
        self.infected = False
        self.hospitalized = False
        self.immune = False
        self.in_quarantine = False # self-quarantine
        self.time_infected = 0
```

This defines an agent as a person with a health and quarantine status.

We still need a few methods to change how other properties can change. We won't go through all of them, just the ones that should suffice for you to gain an understanding of how everything comes together. The core thing we need to understand is what agents do while they're infected. Basically, while infected, we need to understand whether the agents infect others, die from the infection, or recover:

```
def while_infected(self):
    self.time_infected += 1
    if self.hospitalized:
        # stay in bed, do nothing; maybe die
        if self.random.random() < (
            self.model.critical_rate *
            self.model.hospital_factor
```

```

) :
    # die
    self.alive = False
    self.hospitalized = False
    self.infected = False
    return
    self.hospitalized -= 1
    return
if self.random.random() < (
    self.model.quarantine_rate /
    self.model.recovery_period
):
    self.set_quarantine()
if not self.in_quarantine:
    self.infect_others() # infect others in same cell
if self.time_infected < self.model.recovery_period:
    if self.random.random() < self.model.critical_rate:
        if self.model.hospital_takeup:
            self.hospitalized = self.model.hospital_period
            self.set_quarantine()
        else:
            self.alive = False # person died from infection
            self.infected = False
    else: # person has passed the recovery period so no longer
infected
        self.infected = False
        self.quarantine = False
        if self.random.random() < self.model.immunity_chance:
            self.immune = True

```

Here, we can see quite a few variables that are defined at the model level, such as `self.model.critical_rate`, `self.model.hospital_factor`, and `self.model.recovery_period`. We'll look at these model variables in more detail later.

Now, we need a way for our agents to record their position, which is what in `mesa` is called a `MultiGrid`:

```

def move_to_next(self):
    possible_steps = self.model.grid.get_neighborhood(
        self.pos,
        moore=True,
        include_center=False
    )
    new_position = self.random.choice(possible_steps)
    self.model.grid.move_agent(self, new_position)

```

This is relatively straightforward. If agents move, they move within their neighborhood; that is, the next adjacent cell.

The entry method, which is called at every cycle (iteration), is the `step()` method:

```
def step(self):
    if self.alive:
        self.move()
```

Agents move at every step if they are alive. Here's what happens when they move:

```
def move(self):
    if self.in_quarantine or self.model.lockdown:
        pass
    else:
        self.move_to_next()
    if self.infected:
        self.while_infected()
```

This concludes the main logic of our agents; that is, `Person`. Now, let's look at how everything comes together at the model level. This can be found in the `Simulation` class inside `model.py`.

Let's see how the agents are created:

```
def create_agents(self):
    for i in range(self.num_agents):
        a = Person(i, self)
        if self.random.random() < self.start_infected:
            a.set_infected()
        self.schedule.add(a)
        x = self.random.randrange(self.grid.width)
        y = self.random.randrange(self.grid.height)
        self.grid.place_agent(a, (x, y))
```

The preceding code creates as many agents as we need. Some of them will be infected according to the `start_infected` parameter. We also add the agents to a map of cells organized in a grid.

We also need to define a few data collectors, as follows:

```
def set_reporters(self):
    self.datacollector = DataCollector(
        model_reporters={
            'Active Cases': active_cases,
            'Deaths': total_deaths,
            'Immune': total_immune,
```

```

'Hospitalized': total_hospitalized,
'Lockdown': get_lockdown,
})

```

The variables in this dictionary of lists are appended in every cycle so that we can plot them or evaluate them statistically. As an example, let's see how the `active_cases` function is defined:

```

def active_cases(model):
    return sum([
        1
        for agent in model.schedule.agents
        if agent.infected
    ])

```

When called, the function iterates over the agents in the model and counts the ones whose status is `infected`.

Again, just like for `Person`, the main logic of `Simulation` is in the `step()` method, which advances the model by one cycle:

```

def step(self):
    self.datacollector.collect(self)
    self.hospital_takeup = self.datacollector.model_vars[
        'Hospitalized'
    ][-1] < self.free_beds
    self.schedule.step()
    if self.lockdown:
        self.lockdown -= 1
    else:
        if self.lockdown_policy(
            self.datacollector.model_vars['Active Cases'],
            self.datacollector.model_vars['Deaths'],
            self.num_agents
        ):
            self.lockdown = self.lockdown_period
    self.current_cycle += 1

```

Let's see how different lockdown policies affect deaths and the spread of the disease over time.

We'll use the same set of variables that we used previously in these simulations. We've set them so that they roughly correspond to the UK according to a factor of 1/1,000:

```
scale_factor = 0.001
area = 242495 # km2 uk
side = int(math.sqrt(area)) # 492

sim_params = {
    'grid_x': side,
    'grid_y': side,
    'density': 259 * scale_factor, # population density uk,
    'initial_infected': 0.05,
    'infect_rate': 0.1,
    'recovery_period': 14 * 12,
    'critical_rate': 0.05,
    'hospital_capacity_rate': .02,
    'active_ratio': 8 / 24.0,
    'immunity_chance': 1.0,
    'quarantine_rate': 0.6,
    'lockdown_policy': lockdown_policy,
    'cycles': 200 * 12,
    'hospital_period': 21 * 12,
}
```

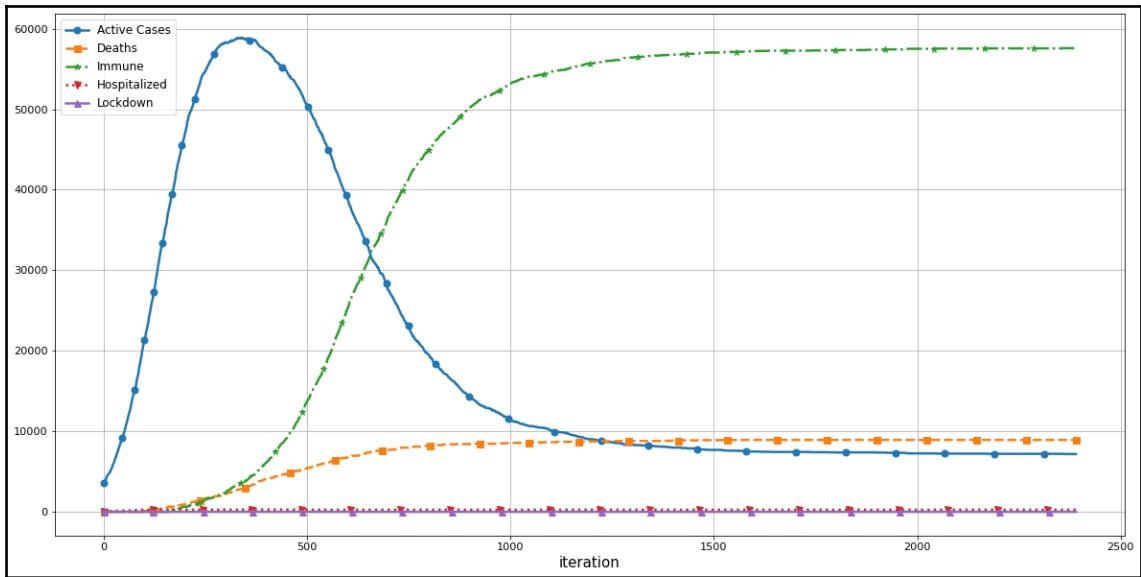
We'll explain the motivation for the grid in the *How it works...* section.

Lockdown is declared by the `lockdown_policy` method, which is passed to the constructor of `Simulation`.

First, let's look at the data when no lockdown was introduced. We can create this policy if our `policy` function always returns `False`:

```
def lockdown_policy(infected, deaths, population_size):
    return 0
```

The resulting graph shows our five collected variables over time:

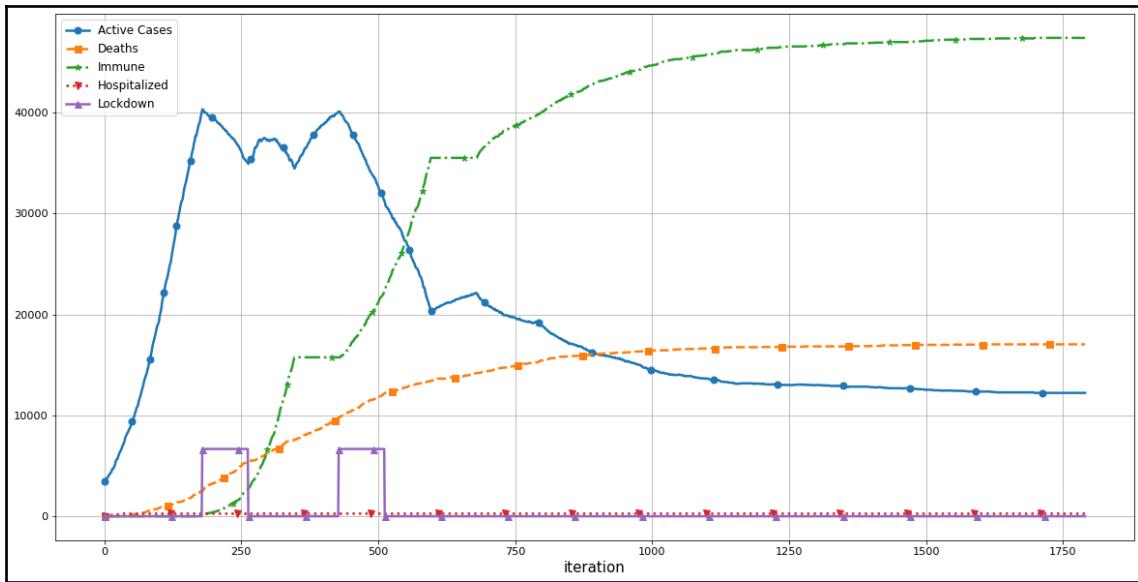


Overall, we have 8,774 deaths.

Here, we can see several waves of infections as lockdown is lifted early according to this policy:

```
def lockdown_policy(infected, deaths, population_size):
    if (
        (max(infected[-5 * 10:]) / population_size) > 0.6
        and
        (len(deaths) > 2 and deaths[-1] > deaths[-2])
    ):
        return 7 * 12
    return 0
```

When we run this simulation, we get a completely different result, as shown here:

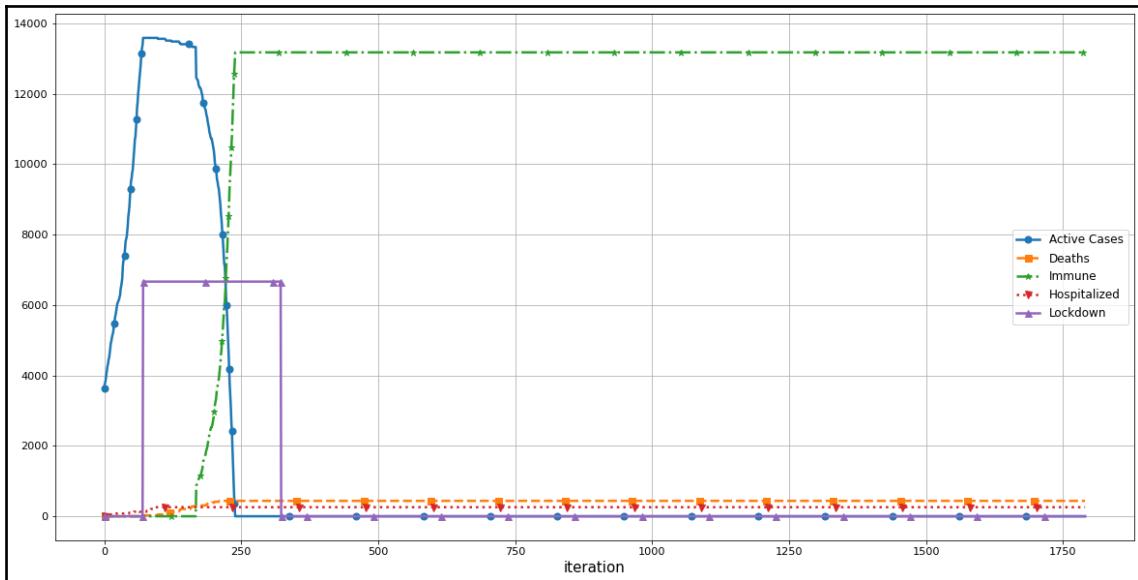


The rectangular shape around 250 iterations shows when lockdown was declared (ignore the scale or shape). Overall, we can see that this resulted in 20,663 deaths. This extremely high death rate –much higher than the `critical_rate` parameter – has been set to 5% due to reinfection before immunity.

Let's compare this to a very cautious policy of declaring lockdown every time the death rate rises or if the rate of infected in the population rises above 20% within (roughly) 3 weeks:

```
def lockdown_policy(infected, deaths, population_size):
    if infected[-1] / population_size > 0.2:
        return 21 * 12
    return 0
```

With a single lockdown, we get the following graph, which shows about 600 deaths overall:



You can change these parameters or play with the logic to create more sophisticated and/or realistic simulations.

More details around the original work can be found online (<https://teck78.blogspot.com/2020/04/using-mesa-framework-to-simulate-spread.html>).

How it works...

The simulation is quite simple: it's composed of agents and proceeds in iterations (called cycles). Each agent represents a part of the population.

Here, a certain population is infected with the disease. At each cycle (which corresponds to 1 hour), infected people can go to the hospital (if there's capacity), die, or make progress toward recovery. They can also go into quarantine. While alive, not recovered, and not in quarantine, they can infect other people in spatial proximity to them. When recovering, agents can become immune.

At each cycle, agents can move around. They move to a new position if they're not in quarantine or the national lockdown has been declared; otherwise, they stay in place. If a person is infected, they can die, go to the hospital, recover, infect others, or go into quarantine.

A national lockdown can be declared according to different policies, depending on death and infection rates. This is the main focus of our simulation: how does the introduction of a national lockdown affect fatalities?

We need to consider different variables for this. One is population density. We can introduce population density by putting our agents on a map or a grid, where the grid size is defined by `grid_x` and `grid_y`. The `infect_rate` parameter must be tweaked according to grid size and population density.

We have a lot more parameters to take into account here, such as the following:

- `initial_infected` is the rate at which the population is initially infected.
- `recovery_period` declares the number of cycles (roughly in hours) it takes to recover after being infected, with 0 for never.
- `critical_rate` is the rate over the whole recovery period in which an ill person can get critically ill, which means they'd go to the hospital, if possible, or die.
- `hospital_capacity_rate` is the number of hospital beds per person over the whole population. We found this through online searches (<https://www.hsj.co.uk/acute-care/nhs-hospitals-have-four-times-more-empty-beds-than-normal/7027392.article>, <https://www.kingsfund.org.uk/publications/nhs-hospital-bed-numbers>).
- There's also `active_ratio`, which defines how active a person is; `quarantine_rate`, which determines how likely someone will go into self-quarantine (if not the hospital); and `immunity_chance`, which is relevant after recovery.
- The simulation is going to run for a set amount of `cycles`, and our lockdown policy is declared in the `lockdown_policy` function.

In the `step()` method of `Simulation`, we performed data collection. Then, we checked whether the hospitals can take on any more patients according to the `free_beds` variable. Then, we ran the agents with `self.schedule.step()`. If we were in lockdown, we counted down. Lockdown is set from `False` to the `lockdown_period` variable (by taking liberty with Python's duck typing) once lockdown is declared.

The `lockdown_policy()` function determines how long the national lockdown should be, given how many agents are infected and deaths over time (lists). Here, 0 means we don't declare lockdown.

There's more...

Since the simulations can take a long time to run, it can be very slow to try out parameters. Instead of having to do a full run, and only then see if we get the desired effect or not, we can use the live plotting functionality of matplotlib.

In order to get faster feedback, let's live plot the simulation loop, as follows:

```
%matplotlib inline
from collections import defaultdict
from matplotlib import pyplot as plt
from IPython.display import clear_output

def live_plot(data_dict, figsize=(7,5), title=''):
    clear_output(wait=True)
    plt.figure(figsize=figsize)
    for label,data in data_dict.items():
        plt.plot(data, label=label)
    plt.title(title)
    plt.grid(True)
    plt.xlabel('iteration')
    plt.legend(loc='best')
    plt.show()

model = Simulation(sim_params)
cycles_to_run = sim_params.get('cycles')
print(sim_params)
for current_cycle in range(cycles_to_run):
    model.step()
    if (current_cycle % 10) == 0:
        live_plot(model.datacollector.model_vars)

print('Total deaths: {}'.format(
    model.datacollector.model_vars['Deaths'][-1]
))
```

This will continuously (every 10 cycles) update our plot of the simulation parameters. Instead of having to wait for a full simulation, we can abort it if it doesn't work out.

See also

You can find out more about mesa's multi-agent-based modeling in Python at <https://mesa.readthedocs.io/en/master/>). Some other multi-agent libraries are as follows:

- MAgent specializes in 2D environments with a very large number of agents that learn through reinforcement learning: <https://github.com/PettingZoo-Team/MAgent>.
- osBrain and PADE are general-purpose multi-agent system libraries. They can be found at <https://osbrain.readthedocs.io/en/stable/> and <https://pade.readthedocs.io/en/latest/>, respectively.
- SimPy is a discrete event simulator that can be used for a broader range of simulations: <https://simpy.readthedocs.io/en/latest/>.

Other simulators have also been released, most prominently the CovidSim microsimulation model (<https://github.com/mrc-ide/covid-sim>), which was developed by the MRC Centre for Global Infectious Disease Analysis, hosted at Imperial College, London.

Writing a chess engine with Monte Carlo tree search

Chess is a two-player board game that's been popular as a game of wits since the 15th century. A computer beat the first human player in the 1950s (a complete novice) before one beat the human world champion in 1997. They have since moved on to having superhuman intelligence. One of the main difficulties of writing a chess engine – a computer program that plays chess – is searching through the many variations and combinations and picking the best one.

In this recipe, we'll use Monte Carlo tree search to create a basic chess engine.

Getting ready

We'll use the `python-chess` library for visualization, to get valid moves, and to know if a state is terminal. We can install it with the `pip` command, as follows:

```
pip install python-chess
```

We'll be using this library for visualization, to generate valid moves at each position, and to check if we've reached a final position.

How to do it...

This recipe is based on a minimal implementation of Monte Carlo tree search by Luke Miles at <https://gist.github.com/qpwo/c538c6f73727e254fdc7fab81024f6e1>.

First, we'll look at the code we'll be using to define our tree search class, and then look at how the search works. After that, we'll learn how this can be adapted to chess.

Tree search

A tree search is a search that employs a search tree as a data structure. In general, in a search tree, nodes (or leaves) represent a concept or a situation, and these are then connected over edges (branches). The tree search traverses the tree to come up with the best solution.

Let's start by implementing the tree search class:

```
import random

class MCTS:
    def __init__(self, exploration_weight=1):
        self.Q = defaultdict(int)
        self.N = defaultdict(int)
        self.children = dict()
        self.exploration_weight = exploration_weight
```

We'll look at these variables in more detail in the *How it works...* section. We'll be adding more methods to this class shortly.

The different steps in our tree search are performed in our `do_rollout` method:

```
def do_rollout(self, node):
    path = self._select(node)
    leaf = path[-1]
    self._expand(leaf)
    reward = self._simulate(leaf)
    self._backpropagate(path, reward)
```

Each `rollout()` call adds one layer to our tree.

Let's get through the four main steps in turn:

1. The `select` step finds a leaf node from which no simulation has been initiated yet:

```
def _select(self, node):
    path = []
    while True:
        path.append(node)
        if node not in self.children or not
            self.children[node]:
            return path
        unexplored = self.children[node] - self.children.keys()
        if unexplored:
            n = unexplored.pop()
            path.append(n)
            return path
    node = self._select(random.choice(self.children[node]))
```

This is defined recursively, so if we don't find an unexplored node, we explore a child of the current node.

2. The expansion step adds the children nodes – the nodes that can be reached via valid moves, given a board position:

```
def _expand(self, node):
    if node in self.children:
        return
    self.children[node] = node.find_children()
```

This function updates the `children` dictionary with the descendants (or children) of the node. These nodes are any valid board positions that can be reached from the node in a single move.

3. The simulation step runs a series of moves until the game is ended:

```
def _simulate(self, node):
    invert_reward = True
    while True:
        if node.is_terminal():
            reward = node.reward()
            return 1 - reward if invert_reward else reward
        node = node.find_random_child()
        invert_reward = not invert_reward
```

This function plays out the simulation until the end of the game.

4. The backpropagation step associates a reward with each step of the path:

```
def _backpropagate(self, path, reward):
    for node in reversed(path):
        self.N[node] += 1
        self.Q[node] += reward
        reward = 1 - reward
```

Finally, we need a way to choose the best move, which can be as simple as going through the Q and N dictionaries and choosing the descendent with the maximum utility (reward):

```
def choose(self, node):
    if node not in self.children:
        return node.find_random_child()

    def score(n):
        if self.N[n] == 0:
            return float('-inf')
        return self.Q[n] / self.N[n]

    return max(self.children[node], key=score)
```

We set the score of any unseen node to $-\infty$, in order to avoid choosing an unseen move.

Implementing a node

Now, let's learn how to use a node for our chess implementation.

Since this is based on the `python-chess` library, it is relatively easy to implement:

```
import hashlib
import copy

class ChessGame:
    def find_children(self):
        if self.is_terminal():
            return set()
        return {
            self.make_move(m) for m in self.board.legal_moves
        }

    def find_random_child(self):
        if self.is_terminal():
            return None
```

```

moves = list(self.board.legal_moves)
m = choice(moves)
return self.make_move(m)

def player_win(self, turn):
    if self.board.result() == '1-0' and turn:
        return True
    if self.board.result() == '0-1' and not turn:
        return True
    return False
def reward(self):
    if self.board.result() == '1/2-1/2':
        return 0.5
    if self.player_win(not self.board.turn):
        return 0.0

def make_move(self, move):
    child = self.board.copy()
    child.push(move)
    return ChessGame(child)
def is_terminal(self):
    return self.board.is_game_over()

```

We've omitted a few methods here, but don't worry – we will be covering them in the *How it works...* section.

Now that everything has been prepared, we can finally play chess.

Playing chess

Let's play some chess!

The following is just a simple loop with a graphical prompt stating the board position:

```

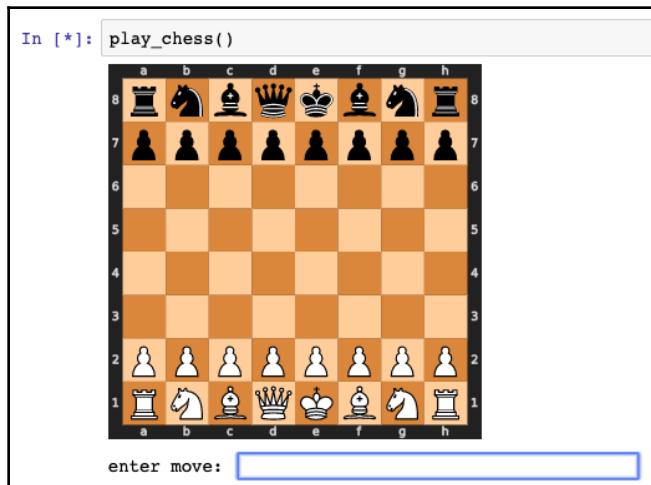
from IPython.display import display
import chess
import chess.svg

def play_chess():
    tree = MCTS()
    game = ChessGame(chess.Board())
    display(chess.svg.board(board=game.board, size=300))
    while True:
        move_str = input('enter move: ')
        move = chess.Move.from_uci(move_str)
        if move not in list(game.board.legal_moves):
            raise RuntimeError('Invalid move')

```

```
game = game.make_move(move)
display(chess.svg.board(board=game.board, size=300))
if game.is_terminal():
    break
for _ in range(50):
    tree.do_rollout(game)
game = tree.choose(game)
print(game)
if game.is_terminal():
    break
```

You should then be asked to enter a move to go to a certain position on the chessboard. After each move, a board will appear, showing the current position of the chess pieces. This can be seen in the following screenshot:



Note that moves have to be entered in UCI notation. If you enter the move in a format that's square to square – for example, a2a4 – it should always work.

The playing strength being used here isn't very high, but it should still be easy to see a few improvements that you can make while play around with it. Note that this implementation is not parallelized.

How it works...

In **Monte Carlo tree search (MCTS)**, we apply the Monte Carlo method – which is basically random sampling – in order to obtain an idea of the strength of the moves that are made by the player. For each move, we play random moves until the game finishes. If we do this often enough, we'll get a good estimate.

The tree search maintains different variables:

- Q is the total reward of each node.
- N is the total visit count for each node.
- `children` holds the children of each node – the nodes that can be reached from a board position.
- A node is a board state in our case.

These dictionaries are important because we average the utility of a node (a board state) over rewards, and we sample nodes based on how often (or rather, how little) they have been visited.

Each iteration of the search consists of four steps:

1. Selection
2. Expansion
3. Simulation
4. Backpropagation

The selection step, at its most basic, looks for a node (such as a board position) that hasn't been explored yet.

The expansion step updates the `children` dictionary with the children of the selected node.

The simulation step is straightforward: we play out a chain of random moves until a terminal position is reached and return the reward. Since this is a two-player, zero-sum board game, we have to invert the reward when it's the opponent's turn.

The backpropagation step follows the path in a backward direction to associate the reward with all the nodes in the explored path. The `_backpropagate()` method backtracks all the nodes along a series of moves (a path), attributes them with a reward, and updates the number of visits.

As for implementing a node, nodes must be hashable and comparable since we'll store them in the dictionaries we mentioned previously. So, here, we need to implement the `__hash__` and `__eq__` methods. We omitted them previously since we didn't need them to understand the algorithm itself, so we've added them here for completeness:

```
def __hash__(self):
    return int(
        hashlib.md5(
            self.board.fen().encode('utf-8')
        ).hexdigest()[:8],
        16
    )
def __eq__(self, other):
    return self.__hash__() == other.__hash__()

def __repr__(self):
    return '\n' + str(self.board)
```

The `__repr__()` method can be quite useful when you are debugging.

For the main functionality of the `ChessGame` class, we also need the following methods:

- `find_children()`: Finds all possible successors from the current node
- `find_random_child()`: Finds a random successor from the current node
- `is_terminal()`: Determines whether a node is terminal
- `reward()`: Provides a reward for the current node

Please take a look at the implementation of `ChessGame` again to see this in action.

There's more...

One major extension of MCTS is **Upper Confidence Trees (UCTs)**, which are used to balance exploration and exploitation. The first Go programs to reach dan level on a 9x9 board used MCTS with UCT.

To implement the UCT extension, we have to go back to our MCTS class and make a couple of changes:

```
def _uct_select(self, node):
    log_N_vertex = math.log(self.N[node])

    def uct(n):
        return self.Q[n] / self.N[n] + self.exploration_weight *
    math.sqrt(
        log_N_vertex / self.N[n]
    )

    return max(self.children[node], key=uct)
```

The `uct()` function applies the **Upper-Confidence-Bound (UCB)** formula, which provides a move with a score. The score of node n is the sum of the number of simulations won among all simulations, starting with node n , plus a confidence term:

$$\text{score}(n) = \text{wins}_n / \text{nsims}_n + c \times \sqrt{\log(2 + \text{nsims}) / \text{nsims}_n}$$

Here, c is a constant.

Next, we need to replace the last line of code so that it uses `_uct_select()` instead of `_select()` for recursion. Here, we'll replace the last line of `_select()` so that it states the following:

```
node = self._uct_select(node)
```

Making this change should increase the agent's playing strength further.

See also

To find out more about UCTs, take a look at the following article on MoGO regarding the first computer Go program to reach dan level on a 9x9 board: https://hal.inria.fr/file/index/docid/369786/filename/TCIAIG-2008-0010_Accepted_.pdf. It also provides a description of MCTS in pseudocode.

The easyAI library contains a lot of different search algorithms: <https://zulko.github.io/easyAI/index.html>.

6

Deep Reinforcement Learning

Reinforcement learning is about developing goal-driven agents to automate problem-solving by optimizing their actions within an environment. This involves predicting and classifying the available data and training agents to execute tasks successfully. Generally, an agent is an entity that has the capacity to interact with an environment, and the learning is done by applying feedback in terms of cumulative rewards from the environment to inform future actions.

Three different types of reinforcement learning can be distinguished:

- Value-based—a value function provides an estimate of how good the current state of the environment is.
- Policy-based—where a function determines an action based on a state.
- Model-based—a model of the environment including state transitions, rewards, and action planning.

In this chapter, we'll start with a relatively basic use case of reinforcement learning for website optimization with multi-armed bandits, where we'll look at an agent and an environment, and how they interact. Then we'll move on to a simple demonstration of control, where it gets a bit more complex, and we'll get to see an agent environment and a policy-based method, REINFORCE. Finally, we'll learn how to play blackjack, where we'll use a **deep Q network (DQN)**, a value-based algorithm that was used in the wave-making AI that could play Atari games created by DeepMind in 2015.

In this chapter, we will cover the following recipes:

- Optimizing a website
- Controlling a cartpole
- Playing blackjack

Technical requirements

The full notebooks are available online on GitHub: <https://github.com/PacktPublishing/Artificial-Intelligence-with-Python-Cookbook/tree/master/chapter06>.

Optimizing a website

In this recipe, we'll deal with website optimization. Often, it is necessary to try changes (or better, a single change) on a website to see the effect they will have. In a typical scenario of what's called an **A/B test**, two versions of the website will be compared systematically. An *A/B* test is conducted by showing versions *A* and *B* of a web page to a pre-determined number of users. Later, statistical significance or a confidence interval is calculated in order to quantify the differences in click-through rates, with the goal of deciding which of the two web page variants to keep.

Here, we'll look at website optimization from a reinforcement point of view, where for each view (or user loading the page), we choose the best version given the available data at the time when they load the website. After each piece of feedback (click or no click), we update the statistics. In comparison to A/B testing, this procedure can yield a more reliable outcome, and additionally we show the best web page variant more often over time. Please note that we are not limited to two variants, but we can compare many variants.

This example use case of website optimization will help us to introduce the notions of agent and environment, and show us the trade-off between exploration and exploitation. We'll explain these concepts in the *How it works...* section.

How to do it...

In order to implement our recipe, we'll need two components:

- Our agent decides which web page to present to the user.
- The environment is a test bed that will give our agent feedback (click or no click).

Since we are only using standard Python, we don't need to install anything, and we can delve right into implementing our recipe:

1. We'll implement our environment first. We are considering this as a multi-armed bandit problem, which we'll be explaining in the *How it works...* section. Consequently, we'll call our environment Bandit:

```
import random
import numpy as np

class Bandit:
    def __init__(self, K=2, probs=None):
        self.K = K
        if probs is None:
            self.probs = [
                random.random() for _ in range(self.K)
            ]
        else:
            assert len(probs) == K
            self.probs = probs

        self.probs = list(np.array(probs) / np.sum(probs))
        self.best_probs = max(self.probs)

    def play(self, i):
        if random.random() < self.probs[i]:
            return 1
        else:
            return 0
```

This bandit is initialized with the number of available choices, K . This will set up a probability for each of these choices to get a click. In practice, the environment would be real user feedback; here, we simulate user behavior instead. The `play()` method plays the i th machine, and returns a reward of 1 or 0.

2. Now we need to interact with this environment. This is where our agent comes in. The agent has to make decisions, and we'll give it a strategy to make decisions. We'll include metrics collection as well. An abstract agent looks like this:

```
class Agent:
    def __init__(self, env):
        self.env = env
        self.listeners = {}
        self.metrics = {}
        self.reset()
```

```

def reset(self):
    for k in self.metrics:
        self.metrics[k] = []

def add_listener(self, name, fun):
    self.listeners[name] = fun
    self.metrics[name] = []
def run_metrics(self, i):
    for key, fun in self.listeners.items():
        fun(self, i, key)

def run_one_step(self):
    raise NotImplementedError
def run(self, n_steps):
    raise NotImplementedError

```

Any agent will need an environment to interact with. It needs to make a single decision (`run_one_step(self)`), and, in order to see how good its decision-making is, we'll need to run a simulation (`run(self, n_steps)`).

Agents will contain a lookup list of metric functions and also inherit a metric collection functionality. We can run the metrics collection through the `run_metrics(self, i)` function.

The strategy that we use here is called UCB1. We'll explain this strategy in the *How to do it...* section:

```

class UCB1(Agent):
    def __init__(self, env, alpha=2.):
        self.alpha = alpha
        super(UCB1, self).__init__(env)
    def run_exploration(self):
        for i in range(self.env.K):
            self.estimates[i] = self.env.play(i)
            self.counts[i] += 1
            self.history.append(i)
            self.run_metrics(i)
            self.t += 1
    def update_estimate(self, i, r):
        self.estimates[i] += (r - self.estimates[i]) /
        (self.counts[i] + 1)

    def reset(self):
        self.history = []
        self.t = 0
        self.counts = [0] * self.env.K
        self.estimates = [None] * self.env.K

```

```
super(UCB1, self).reset()
def run(self, n_steps):
    assert self.env is not None
    self.reset()
    if self.estimates[0] is None:
        self.run_exploration()
    for _ in range(n_steps):
        i = self.run_one_step()
        self.counts[i] += 1
        self.history.append(i)
        self.run_metrics(i)

def upper_bound(self, i):
    return np.sqrt(
        self.alpha * np.log(self.t) / (1 + self.counts[i])
    )
def run_one_step(self):
    i = max(
        range(self.env.K),
        key=lambda i: self.estimates[i] + self.upper_bound(i)
    )
    r = self.env.play(i)
    self.update_estimate(i, r)
    self.t += 1
    return i
```

Our UCB1 agent needs an environment (a bandit) to interact with, and a single parameter alpha, which weighs the importance of exploring actions (versus exploiting the best known action). The agent maintains its history of choices over time, and a record of estimates for each possible choice.

What we should look at is the `run_one_step(self)` method, which makes a single choice by picking the best optimistic choice. The `run(self, n_step)` method runs a series of choices and gets back the feedback from the environment.

Let's track two metrics: regret, which is the sum of expected losses occurred because of suboptimal choices, and—as a measure of convergence of the agent's estimates against the actual configuration of the environment—the Spearman rank correlation (`stats.spearmanr()`).

The **Spearman rank correlation** is equal to the Pearson correlation (often briefly called just the correlation or product-moment correlation) of the ranked variables.

The Pearson correlation between two variables, X and Y , can be expressed as follows:

$$\rho_{X,Y} = \frac{\text{cov}_{X,Y}}{\sigma_X \sigma_Y},$$

where $\text{cov}_{X,Y}$ is the covariance of X and Y , and σ_X is the standard deviation of X .



The Spearman correlation, instead of operating on the raw scores, is calculated on the ranked scores. A rank transformation means that a variable is sorted by value, and each entry is assigned its order. Given a rank r_{X_i} of point i in X , the Spearman rank correlation is calculated like this:

$$\rho_{r_X, r_Y} = \frac{\text{cov}(r_X, r_Y)}{\sigma_{r_X} \sigma_{r_Y}}.$$

This assesses how well the relationship between two variables can be described as a monotonic, but not necessarily linear (as in the case of Pearson correlation) function. Like the Pearson correlation, the Spearman correlation ranges between -1 for perfectly negatively correlated and 1 for perfectly correlated. 0 means there's no correlation.

Our tracking functions are `update_regret()` and `update_rank_corr()`:

```
from scipy import stats

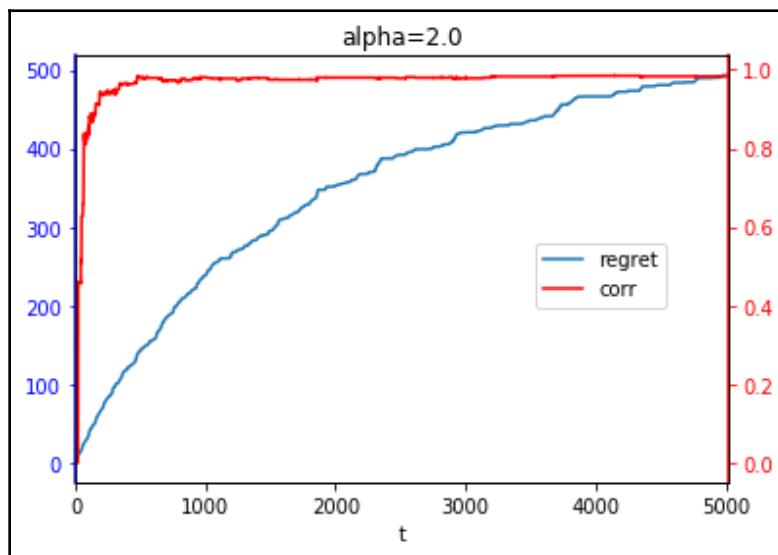
def update_regret(agent, i, key):
    regret = agent.env.best_probs - agent.env.probs[i]
    if agent.metrics[key]:
        agent.metrics[key].append(
            agent.metrics[key][-1] + regret
        )
    else:
        agent.metrics[key] = [regret]

def update_rank_corr(agent, i, key):
    if agent.t < agent.env.K:
        agent.metrics[key].append(0.0)
    else:
        agent.metrics[key].append(
            stats.spearmanr(agent.env.probs, agent.estimates)[0]
        )
```

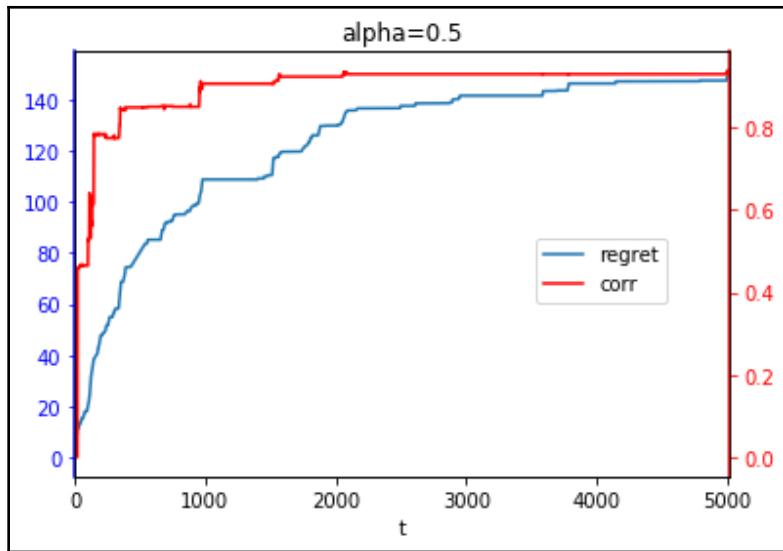
We can now track these metrics in order to compare the influence of the alpha parameter (more or less exploration). We can then observe convergence and cumulative regret over time:

```
random.seed(42.0)
bandit = Bandit(20)
agent = UCB1(bandit, alpha=2.0)
agent.add_listener('regret', update_regret)
agent.add_listener('corr', update_rank_corr)
agent.run(5000)
```

So we have 20 different choices of web pages, and we collect `regret` and `corr` as defined, and we run for 5000 iterations. If we plot this, we can get an idea of how well this agent performed:



For the second run, we'll change alpha to 0.5, so we'll do less exploration:



We can see that the cumulative regret with $\alpha=0.5$, less exploration, is much lower than with $\alpha=2.0$; however, the overall correlation of the estimates to the environment parameters is lower.

So, with less exploration our agent models the true parameters of the environment less well. This comes from the fact that with less exploration the ordering of the lower ranked features has not converged. Even though they are ranked as suboptimal, they haven't been chosen often enough to determine whether they are worst or second worst, for example. This is what we see with less exploration, and this could be fine since we might only care about knowing which choice is best.

How it works...

In this recipe, we dealt with the problem of website optimization. We simulated user choices to different versions of web pages while live-updating statistics about how good each variant is, and how often it should be shown. Furthermore, we compared the upsides and downsides of an explorative scenario and a more exploitative scenario.

We framed user responses to web pages as a multi-armed bandit problem. A **multi-armed bandit (MABP)** is a slot machine where gamblers insert a coin and pull one of several levers, each of which is associated with a different reward distribution that is unknown to the gambler. More generally, a MABP; also called the **K-armed bandit problem**) is the problem of allocating resources between competing choices in a situation, where the results of each choice are only partially known, but may become better known over time. When observations of the world are considered when making a decision, this is known as the **contextual bandit**.

We've used the **Upper Confidence Bound version 1 (UCB1)** algorithm (Auer et al., *Finite-time analysis of the multi-armed bandit problem*, 2002), which is easy to implement.

It works as follows:

- Play each action once in order to get initial estimates for the mean rewards (exploration phase).
- For each round t update $Q(a)$ and $N(a)$, and play the action a' according to this formula:

$$a' \leftarrow \operatorname{argmax}_{a \in A} Q(a) + \sqrt{\frac{\alpha \ln t}{N(a)}},$$

where $Q(a)$ is the lookup table for the mean reward and $N(a)$ is the number of times action a has been played. $\alpha > 0$ is a parameter.

UCB algorithms follow the so-called optimism in the face of uncertainty principle by choosing the arm with the highest UCB on its confidence interval rather than the one with the highest estimated reward. It uses a naive mean estimator for the action reward.

The second term in the preceding equation quantifies the uncertainty. The lower the uncertainty, the more we rely on $Q(a)$. The uncertainty decreases linearly with the number of times an action has been played and increases logarithmically with the number of rounds.

Multi-armed bandits are useful in many domains, including online advertising, clinical trials, network routing, or switching between two or more versions of a machine learning model in production.

There are many variants of the bandit algorithm that address more complex scenarios, for example, costs for switching between choices, or choices with finite lifespans such as the secretary problem. The basic setting of the secretary problem is that you want to hire a secretary from a finite pool of applicants. Each applicant is interviewed in turn in random order, and a definite decision (to hire or not) is to be made immediately after the interview. The secretary problem is also called the marriage problem.

See also

The Ax library implements many bandit algorithms in Python: <https://ax.dev/>.

Facebook's PlanOut is a library for massive online field experiments: <https://facebook.github.io/planout/index.html>.

As reading material we'd recommend these:

- Russo and others, 2007, *A Tutorial on Thompson Sampling* (<https://arxiv.org/pdf/1707.02038.pdf>)
- Szepesvari and Lattimore, *Bandit Algorithms*, 2020 (online version available: <https://tor-lattimore.com/downloads/book/book.pdf>)

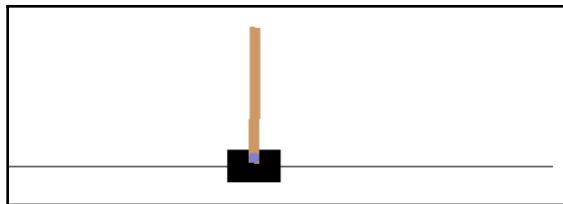
Controlling a cartpole

The cartpole is a control task available in OpenAI Gym, and has been studied for many years. Although it is relatively simple compared to others, it contains all that we need in order to implement a reinforcement learning algorithm, and everything that we develop here can be applied to other, more complex learning tasks. It can also serve as an example of robotic manipulation in a simulated environment. The advantage of taking one of the less demanding tasks is that training and turnaround is quicker.



OpenAI Gym is an open source library that can help to develop reinforcement algorithms by standardizing a broad range of environments for agents to interact with. OpenAI Gym comes with hundreds of environments and integrations ranging from robotic control, and walking in 3D to computer games and self-driving cars: <https://gym.openai.com/>

The cartpole task is depicted in the following screenshot of the OpenAI Gym environment and consists of moving a cart to the left or right in order to balance a pole in an upright position:



In this recipe, we'll implement the REINFORCE policy gradient method in PyTorch to solve the cartpole task. Let's get to it.

Getting ready

There are many libraries that provide collections of test problems and environments. One of the libraries with the most integrations is OpenAI Gym, which we'll utilize in this recipe:

```
pip install gym
```

We can now use OpenAI Gym in our recipe.

How to do it...

OpenAI Gym saves us work—we don't have to define the environment ourselves and come up with reward signals, encode the environment, or state which actions are allowed.

We'll first load the environment, define a deep learning policy for action selection, define an agent that uses this policy to select actions to execute, and finally we'll test how the agent performs in our task:

1. First, we'll load the environment. Every move that the pole doesn't fall over, we get a reward. We have two available moves, left or right, and an observation space that includes a representation of the cart position and velocity and the pole angle and velocity, as in the following table:

Num	Observation	Min	Max
0	Cart Position	-2.4	2.4
1	Cart Velocity	-Inf	-Inf
2	Pole Angle	~ -41.8°	~ 41.8°
3	Pole Velocity At Tip	-Inf	-Inf

You can find out more about this environment here: <https://gym.openai.com/envs/CartPole-v1/>.

We can load the environment and print these parameters as follows:

```
import gym

env = gym.make('CartPole-v1')
print('observation space: {}'.format(
    env.observation_space
))
print('actions: {}'.format(
    env.action_space.n
))
#observation space: Box(4,)
#actions: 2
```

So, we confirm we have four inputs and two actions that our agent has to deal with. Our agent will be defined similarly from the previous recipe, *Optimizing a website*, only this time, we will define our neural network outside of the agent.

The agent will create a policy network and use it to take decisions until an end state is reached; then it will feed the cumulative rewards into the network to learn. Let's start with the policy network.

2. Let's create a policy network. We'll take a fully connected feed-forward neural network that predicts moves based on the observation space. This is based partly on a PyTorch implementation available at https://github.com/pytorch/examples/blob/master/reinforcement_learning/reinforce.py:

```
import torch as T
import torch.nn as nn
import torch.nn.functional as F
import numpy as np

class PolicyNetwork(nn.Module):
    def __init__(self, lr, n_inputs, n_hidden, n_actions):
        super(PolicyNetwork, self).__init__()
        self.lr = lr
        self.fc1 = nn.Linear(n_inputs, n_hidden)
        self.fc2 = nn.Linear(n_hidden, n_actions)
        self.optimizer = optim.Adam(self.parameters(), lr=self.lr)
        self.device = T.device('cuda:0')
```

```

        if T.cuda.is_available()
            else 'cpu:0'
        )
        self.to(self.device)
    def forward(self, observation):
        x = T.Tensor(observation.reshape(-1).astype('float32'),
                     ).to(self.device)
        x = F.relu(self.fc1(x))
        x = F.softmax(self.fc2(x), dim=0)
        return x

```

This is a neural network module that learns a policy, or, in other words, a mapping from observations to actions. This sets up a two-layer neural network with one hidden layer and one output layer, where each neuron in the output layer corresponds to one possible action. We set the following parameters:

- lr: Learning rate
- n_inputs: Number of inputs
- n_hidden: Number of hidden neurons
- n_actions: Input dimensionality

3. We are ready now to define our agent:

```

class Agent:
    eps = np.finfo(
        np.float32
    ).eps.item()

    def __init__(self, env, lr, params, gamma=0.99):
        self.env = env
        self.gamma = gamma
        self.actions = []
        self.rewards = []
        self.policy = PolicyNetwork(
            lr=lr,
            **params
        )

    def choose_action(self, observation):
        output = self.policy.forward(observation)
        action_probs = T.distributions.Categorical(
            output
        )
        action = action_probs.sample()
        log_probs = action_probs.log_prob(action)
        action = action.item()

```

```
    self.actions.append(log_probs)
    return action, log_probs
```

Agents evaluate policies to take actions and get rewards. `gamma` is the discount factor.

With the `choose_action(self, observation)` method, our agent chooses an action given an observation. The action is sampled according to the categorical distribution from our network.

We've omitted the `run()` method, which looks as follows:

```
def run(self):
    state = self.env.reset()
    probs = []
    rewards = []
    done = False
    observation = self.env.reset()
    t = 0
    while not done:
        action, prob =
            self.choose_action(observation.reshape(-1))
        probs.append(prob)
        observation, reward, done, _ = self.env.step(action)
        rewards.append(reward)
        t += 1

    policy_loss = []
    returns = []
    R = 0
    for r in rewards[::-1]:
        R = r + self.gamma * R
        returns.insert(0, R)
    returns = T.tensor(returns)
    returns = (returns - returns.mean()) / (returns.std() +
    self.eps)

    for log_prob, R in zip(probs, returns):
        policy_loss.append(-log_prob * R)

    if(len(policy_loss)) > 0:
        self.policy.optimizer.zero_grad()
        policy_loss = T.stack(policy_loss, 0).sum()
        policy_loss.backward()
        self.policy.optimizer.step()
    return t
```

The `run(self)` method similar to the previous recipe, *Optimizing a website*, runs a full simulation in the environment until the end is reached. This is until the pole is about to fall over or 500 steps are reached (which is the default value of `env._max_episode_steps`).

4. Next, we'll test our agent. We'll start running our agent in the environment by simulating interactions with the environment. In order to get a cleaner curve for our learning rate, we'll set `env._max_episode_steps` to 10000. This means the simulation stops after 10,000 steps. If we'd left it at 500, the default value, the algorithm would plateau at a certain performance or its performance would stagnate once about 500 steps are reached. Instead, we are trying to optimize a bit more:

```
env._max_episode_steps = 10000
input_dims = env.observation_space.low.reshape(-1).shape[0]
n_actions = env.action_space.n

agent = Agent(
    env=env,
    lr=0.01,
    params=dict(
        n_inputs=input_dims,
        n_hidden=10,
        n_actions=n_actions
    ),
    gamma=0.99,
)
update_interval = 100
scores = []
score = 0
n_episodes = 25000
stop_criterion = 1000
for i in range(n_episodes):
    mean_score = np.mean(scores[-update_interval:])
    if (i>0) and (i % update_interval) == 0:
        print('Iteration {}, average score: {:.3f}'.format(
            i, mean_score
        ))
    T.save(agent.policy.state_dict(), filename)

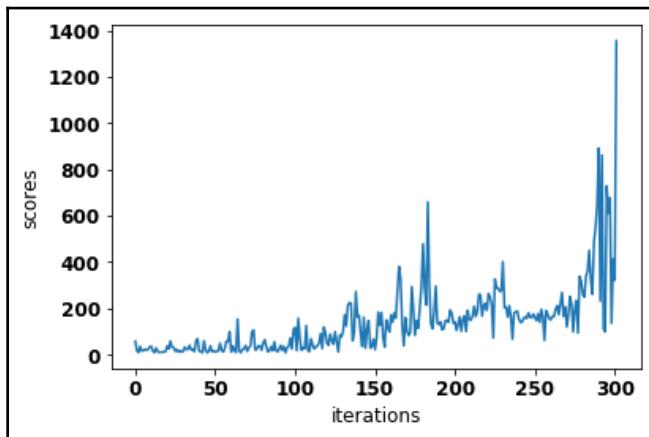
    score = agent.run()
    scores.append(score)
    if score >= stop_criterion:
        print('Stopping. Iteration {}, average score: {:.3f}'.format(
            i, mean_score
        ))
```

```
)  
break
```

We should see the following output:

```
Iteration 100, average score: 31.060  
Iteration 200, average score: 132.340  
Iteration 300, average score: 236.550  
Stopping. Iteration 301, average score: 238.350
```

While the simulations are going on we are seeing updates every 100 iterations with average scores since the last update. We stop once a score of 1,000 is reached. This is our score over time:



We can see that our policy is continuously improving—the network is learning successfully to manipulate the cartpole. Please note that your results can vary. The network can learn more quickly or more slowly.

In the next section, we'll get into how this algorithm actually works.

How it works...

In this recipe, we've looked at a policy-based algorithm in a cartpole control scenario. Let's look at some of this in more detail.

Policy gradient methods find a policy with a given gradient ascent that maximizes cumulative rewards with respect to the policy parameters. We've implemented a model-free policy-based method, the REINFORCE algorithm (R. Williams, *Simple statistical gradient-following algorithms for connectionist reinforcement learning*, 1992).

At the heart of a policy-based method we have a policy function. The policy function is defined over an environment s and an action a , and returns the probability of an action given an environment. In the case of N discrete choices, we can use the softmax function:

$$\pi(a_t = a_i, s_{t-1}) = \frac{e^{p(a_i, s_{t-1})}}{\sum_{j=0}^N e^{p(a_j, s_{t-1})}}$$

This is what we've done in our policy network, and this helps us to make our action choice.

The value function $Q(a_t, s_t) \in \mathbb{R}$ (sometimes denoted by v) returns the reward for any action in a given environment. The update to the policy is defined as follows:

$$\nabla \theta_t = \alpha \nabla_\theta \log \pi(a_t, s_t) Q(a, s)$$

where α is the learning rate.

After the initialization of the parameters, the REINFORCE algorithm proceeds by applying this update function each time an action is executed.

You should be able to run our implementation on any Gym environment with few to no changes. We've deliberately put in a few things (for example, reshaping observations to a vector) to make it easier to reuse it; however, you should make sure that your network architecture corresponds to the nature of your observations. For example, you might want to use a 1D convolutional network or a recurrent neural network for time series (such as in stock trading or sounds) or 2D convolutions if your observations are images.

There's more...

There are a few more things that we can play around with. For one, we'd like to see the agent interacting with the pole, and secondly, instead of implementing an agent from scratch, we can use a library.

Watching our agents in the environment

We can play many hundreds of games or try different control tasks. If we want to actually watch our agent interact with the environment in a Jupyter notebook, we can do it:

```
from IPython import display
import matplotlib.pyplot as plt
%matplotlib inline

observation = env.reset()
img = plt.imshow(env.render(mode='rgb_array'))
for _ in range(100):
    img.set_data(env.render(mode='rgb_array'))
    display.display(plt.gcf())
    display.clear_output(wait=True)
    action, prob = agent.choose_action(observation)
    observation, _, done, _ = agent.env.step(action)
    if done:
        break
```

We should see our agent interacting with the environment now.

If you are on a remote connection (such as running on Google Colab), you might have to do some extra work:

```
!sudo apt-get install -y xvfb ffmpeg
!pip install 'gym==0.10.11'
!pip install 'imageio==2.4.0'
!pip install PILLOW
!pip install 'pyglet==1.3.2'
!pip install pyvirtualdisplay
display = pyvirtualdisplay.Display(
    visible=0, size=(1400, 900)
).start()
```

In the next section, we'll use a reinforcement algorithm that's implemented in a library, RLLib.

Using the RLlib library

Instead of implementing algorithms from scratch, we can make use of implementations in Python libraries and packages. For example, we can train the PPO algorithm (Schulman et al., *Proximal Policy Optimization Algorithms*, 2017), which comes with the `RLlib` package. `RLlib` is part of the `Ray` library that we encountered in the *Getting started with Artificial Intelligence in Python* recipe in Chapter 1, *Getting Started with Artificial Intelligence in Python*. PPO is a policy gradient method that introduces a surrogate objective function that can be optimized with gradient descent:

```
import ray
from ray import tune
from ray.rllib.agents.ppo import PPOTrainer

ray.init()
trainer = PPOTrainer

analysis = tune.run(
    trainer,
    stop={'episode_reward_mean': 100},
    config={'env': 'CartPole-v0'},
    checkpoint_freq=1,
)
```

This will run the training. Your agents will be stored in a local directory, so you can load them up later. `RLlib` lets you use PyTorch and TensorFlow with the '`torch`' : `True` option.

See also

Some reinforcement libraries come with lots of implementations of deep reinforcement learning algorithms:

- OpenAI's Baselines (needs TensorFlow version < 2): <https://github.com/openai/baselines>
- RLlib (part of Ray) is a library for scalable (and distributed) reinforcement learning: <https://docs.ray.io/en/master/rllib-algorithms.html>.
- Machin is a reinforcement learning library based on PyTorch: <https://github.com/iiffiX/machin>.
- TF-Agents provides many tools and algorithms (and it works with TensorFlow version 2.0 and later): <https://github.com/tensorflow/agents>.

Please note that the installation of these libraries can take a while and might take up gigabytes of your hard disk.

Finally, OpenAI provides a repository with many educational resources related to reinforcement learning: <https://spinningup.openai.com/>.

Playing blackjack

One of the benchmarks in reinforcement learning is gaming. Many different environments related to gaming have been designed by researchers or aficionados. A few of the milestones in gaming have been mentioned in Chapter 1, *Getting Started with Artificial Intelligence in Python*. The highlights for many would certainly be beating the human champions in both chess and Go—chess champion Garry Kasparov in 1997 and Go champion Lee Sedol in 2016—and reaching super-human performance in Atari games in 2015.

In this recipe, we get started with one of the simplest game environments: blackjack. Blackjack has an interesting property that it has in common with the real world: indeterminism.

Blackjack is a card game where, in its simplest form, you play against a card dealer. You have a deck of cards in front of you, and you can hit, which means you get one more card, or stick, where the dealer gets to draw cards. In order to win, you want to get as close as possible to a card score of 21, but not surpass 21.

In this recipe, we'll implement a model in Keras of the value of different actions given a configuration of the environment, a value function. The variant we'll implement is called the DQN, which was used in the 2015 Atari milestone achievement. Let's get to it.

Getting ready

We need to install a dependency if you haven't installed it yet.

We'll be using OpenAI Gym, and we need to have it installed:

```
pip install gym
```

We'll be using the Gym environment for blackjack.

How to do it...

We need an agent that maintains a model of what effects its actions have. These actions are played back from its memory for the purpose of learning. We will start with a memory that records past experiences for learning:

1. Let's implement this memory. This memory is essentially a FIFO queue. In Python, you could use a deque; however, we found the implementation of the replay memory in the PyTorch examples very elegant, so this is based on Adam Paszke's PyTorch design:

```
#this is based on
https://pytorch.org/tutorials/intermediate/reinforcement\_q\_learning.html
from collections import namedtuple

Transition = namedtuple(
    'Transition',
    ('state', 'action', 'next_state', 'reward')
)

class ReplayMemory:
    def __init__(self, capacity=2000):
        self.capacity = capacity
        self.memory = []
        self.position = 0

    def push(self, *args):
        if len(self.memory) < self.capacity:
            self.memory.append(None)
        self.memory[self.position] = Transition(*args)
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        batch = random.sample(self.memory, batch_size)
        batch = Transition(
            *(np.array(el).reshape(batch_size, -1) for
el in zip(*batch))
        )
        return batch

    def __len__(self):
        return len(self.memory)
```

We only really need two methods:

- We need to push new memories, overwriting old ones in the process if our capacity is reached.
- We need to sample memories for learning.

The latter point is worth stressing: instead of using all the memories for learning, we only take a part of them.

In the `sample()` method, we made a few alterations to get our data in the right shape.

2. Let's look at our agent:

```
import random
import numpy as np
import numpy.matlib
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras import optimizers
from tensorflow.keras import initializers

class DQNAgent():
    def __init__(self, env, epsilon=1.0, lr=0.5, batch_size=128):
        self.env = env
        self.action_size = self.env.action_space.n
        self.state_size = env.observation_space
        self.memory = ReplayMemory()
        self.epsilon = epsilon
        self.lr = lr
        self.batch_size = batch_size
        self.model = self._build_model()
    def encode(self, state, action=None):
        if action is None:
            action = np.reshape(
                list(range(self.action_size)),
                (self.action_size, 1)
            )
        return np.hstack([
            np.matlib.repmat(state, self.action_size, 1),
            action
        ])
    return np.hstack([state, action])

    def play(self, state):
        state = np.reshape(state, (1, 3)).astype(float)
        if np.random.rand() <= self.epsilon:
```

```
        action = np.random.randint(0, self.action_size)
    else:
        action_value =
    self.model.predict(self.encode(state)).squeeze()
        action = np.argmax(action_value)
    next_state1, reward, done, _ = self.env.step(action)
    next_state = np.reshape(next_state1, (1, 3)).astype(float)
    if done:
        self.memory.push(state, action, next_state, reward)
    return next_state1, reward, done

def learn(self):
    if len(self.memory) < self.batch_size:
        return
    batch = self.memory.sample(
        self.batch_size
    )
    result = self.model.fit(
        self.encode(batch.state, batch.action),
        batch.reward,
        epochs=1,
        verbose=0
    )
```

Please note the action choice at the beginning of the `play()` method. We throw a die to determine if we want to choose an action randomly or if we want to follow our model's judgment. This is called an **epsilon-greedy action selection**, which leads to more exploration and better adaptation to the environment.

The agent comes with a few hyperparameters to configure:

- `lr`: The learning rate of the network.
- `batch_size`: The batch size for the sampling from memory and network training.
- `epsilon`: This factor, between 0 and 1, controls how much randomness we want in our responses. 1 means random exploration, 0 means no exploration at all (exploitation).

We found that these three parameters can significantly change the trajectory of our learning.

We've omitted a method from the listing, which defines the neural network model:

```
def _build_model(self):
    model = tf.keras.Sequential([
        layers.Dense(
            100,
            input_shape=(4,),
            kernel_initializer=initializers.RandomNormal(stddev=5.0),
            bias_initializer=initializers.Ones(),
            activation='relu',
            name='state'
        ),
        layers.Dense(
            2,
            activation='relu'
        ),
        layers.Dense(1, name='action', activation='tanh')
    ])
    model.summary()
    model.compile(
        loss='hinge',
        optimizer=optimizers.RMSprop(lr=self.lr)
    )
    return model
```

This is a three-layer neural network with two hidden layers, one with 100 neurons and the other with 2 neurons, that come with ReLU activations, and an output layer with 1 neuron.

3. Let's load the environment and initialize our agent. We initialize our agent and the environment as follows:

```
import gym
env = gym.make('Blackjack-v0')

agent = DQNAgent(
    env=env, epsilon=0.01, lr=0.1, batch_size=100
)
```

This loads the **Blackjack** OpenAI Gym environment and our **DQNAgent** as implemented in *Step 2* of this recipe.

The `epsilon` parameter defines the random behavior of the agent. We don't want to set this too low. The learning rate is a value we are choosing after experimenting. Since we are playing a stochastic card game, if we set it too high, the algorithm will unlearn very quickly. The batch size and memory size parameters are important for how much we train at every step and how much we remember about the history of rewards, respectively.

We can see the structure of this network (as shown by Keras's `summary()` method):

Layer (type)	Output Shape	Param #
<hr/>		
state (Dense)	(None, 100)	500
dense_4 (Dense)	(None, 2)	202
action (Dense)	(None, 1)	3
<hr/>		
Total params:	705	
Trainable params:	705	
Non-trainable params:	0	

For the simulation, one of our key questions is the value of the `epsilon` parameter. If we set it too low, our agent won't learn anything; if we set it too high, we'd lose money because the agent makes random moves.

- Now let's play blackjack. We chose to steadily decrease `epsilon` in a linear fashion, and then we exploit for a number of rounds. When `epsilon` reaches 0, we stop learning:

```
num_rounds = 5000
exploit_runs = num_rounds // 5
best_100 = -1.0

payouts = []
epsilons = np.hstack([
    np.linspace(0.5, 0.01, num=num_rounds - exploit_runs),
    np.zeros(exploit_runs)
])
```

This is the code that actually starts playing blackjack:

```
from tqdm.notebook import trange

for sample in trange(num_rounds):
```

```
epsilon = epsilons[sample]
agent.epsilon = epsilon
total_payout = 0
state = agent.env.reset()
for _ in range(10):
    state, payout, done = agent.play(state)
    total_payout += payout
    if done:
        break
if epsilon > 0:
    agent.learn()

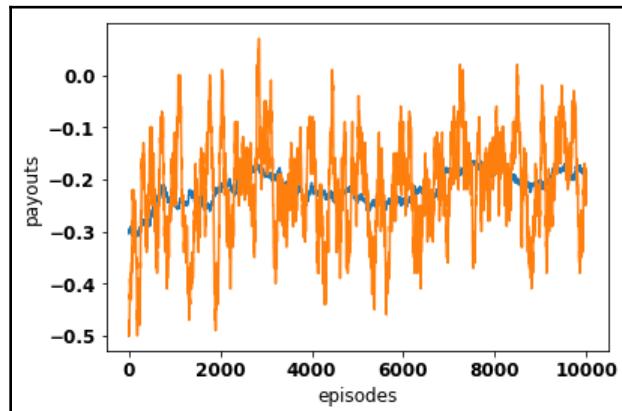
mean_100 = np.mean(payouts[-100:])
if mean_100 > best_100:
    best_100 = mean_100

payouts.append(total_payout)
if (sample % 100) == 0 and sample >= 100:
    print('average payout: {:.3f}'.format(
        mean_100
    ))
    print(agent.losses[-1])

print('best 100 average: {:.3f}'.format(best_100))
```

You can see we are collecting statistics of the network training loss to monitor during simulations, and we collect the the maximum payout over any 100 consecutive plays.

In OpenAI Gym, the rewards, or if we want to stay within the terminology of blackjack, the payouts, can be either **-1** (we lost), **0** (nothing), or **1** (we win). The payouts over time with our learned strategy look as follows:



Because of the huge variability, rather than showing the raw data, we've plotted this with moving averages of 100 and 1,000, which results in two lines, one that is highly variable and another that is smooth, as you can see in the graph.

We do see an increase in the payouts over time; however, we are still below 0, which means we lose money on average. This happens even if we stop learning in the exploitation phase.

Our blackjack environment does not have a reward threshold at which it's considered solved; however, a write-up lists 100 best episodes with an average of 1.0, which is what we reach as well: https://gym.openai.com/evaluations/eval_21dT2zxJTbKa1TJg9NB8eg/.

How it works...

In this recipe, we've seen a more advanced algorithm in reinforcement learning, more specifically, a value-based algorithm. In value-based reinforcement learning, algorithms build an estimator of the value function, which, in turn, lets us choose a policy.

The agent deserves a few more comments. If you've read the previous recipe, *Controlling a cartpole*, you might think there's really not that much going on—there's a network, a `play()` method to decide between actions, and a `learn()` method. The code is relatively small. A basic threshold strategy (do my cards sum to 17?) is already quite successful, but hopefully what we show in this recipe can still be instructive and helpful for more complex use cases. As opposed to the policy network we've seen before, this time, rather than suggesting the best action directly, the network takes the combination of environment and action as an input and outputs the expected reward. Our model is a feed-forward model of two layers, where the hidden layer with two neurons get summed in the final layer composed of a single neuron. The agent plays in an epsilon-greedy fashion—it makes a random move with probability `epsilon`; otherwise it makes the best move according to its knowledge. The `play` function suggests the action that has the highest utility by comparing expected outcomes over all available actions.

The Q-value function, $Q^\pi(s, a) : S \times A \rightarrow \mathbb{R}$, is defined as follows:

$$Q^\pi(s, a) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+1|s_t=s, a_t=a, \pi}\right]$$

where $r_t, s_t \in S, a_t \in A$ are the reward, state, and action at time t . γ is the discount factor; the policy π selects the actions.

In the simplest case, Q can be a lookup table with an entry for every state-action pair.

The optimal Q-value function is defined as follows:

$$Q^*(s, a) = \max_{\pi \in \Pi} Q^\pi(s, a)$$

And, consequently, the best policy can be determined according to this formula:

$$\pi^*(s) = \operatorname{argmax}_{a \in A} Q^*(s, a).$$

In **neural fitted Q-learning (NFQ)** (Riedmiller, *Neural Fitted Q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method*, 2005), a neural network runs a forward pass given a state with the outputs corresponding to available actions. The neural Q-value function can be updated with gradient descent according to a squared error:

$$J = (Q(s, a; \theta) - Y_k^Q)^2, \text{ where}$$

$$Y_k^Q = r + \gamma \max_{a' \in A} Q(s', a'; \theta_k),$$

$$J = (Q(s, a; \theta) - Y_k^Q)^2, \text{ where}$$

$$Y_k^Q = r + \gamma \max_{a' \in A} Q(s', a'; \theta_k),$$

where θ_k refers to the parameters at iteration k , and a', s' refer to action and state at the next time step.

The DQN (Mnih et al., *Playing Atari with Deep Reinforcement Learning*, 2015) builds on NFQ, but introduces a few changes. These include updating parameters only in mini-batches every few iterations, based on random samples from a replay memory. Since, in the original paper, the algorithm learned from pixel values on the screen, the first layers of the network are convolutional (we'll introduce these in Chapter 7, *Advanced Image Applications*).

See also

Here is the website for Sutton and Barto's seminal book *Reinforcement Learning: An Introduction*: <http://incompleteideas.net/book/the-book-2nd.html>.

They've described a simple agent for blackjack in there. If you are looking for other card games, you can have a look at neuron-poker, an OpenAI poker environment; they've implemented DQN and other algorithms: https://github.com/dickreuter/neuron_poker.

For more details about the DQNs and how to use it, we recommend reading Mnih et al.'s article, *Playing Atari with Deep Reinforcement Learning*: <https://arxiv.org/abs/1312.5602>.

Finally, the DQN and its successors, the Double DQN and Dueling DQNs form the basis for AlphaGo, which has been published as *Mastering the game of Go without human knowledge* (Silver and others, 2017) in Nature: <https://www.nature.com/articles/nature24270>.

7

Advanced Image Applications

The applications of artificial intelligence in computer vision include robotics, self-driving cars, facial recognition, recognizing diseases in biomedical images, and quality control in manufacturing, among many others.

In this chapter, we'll start with image recognition (or image classification), where we'll look into basic models and more advanced models. We'll then create images using **Generative Adversarial Networks (GANs)**.

In this chapter, we will cover the following recipes:

- Recognizing clothing items
- Generating images
- Encoding images and style

Technical requirements

We'll use many standard libraries, such as NumPy, Keras, and PyTorch, but we'll also see a few more libraries that we'll mention at the beginning of each recipe as they become relevant.

You can find the notebooks for this chapter's recipes on GitHub at <https://github.com/PacktPublishing/Artificial-Intelligence-with-Python-Cookbook/tree/master/chapter07>.

Recognizing clothing items

A popular example of image classification is the MNIST dataset, which contains digits from 0 to 9 in different styles. Here, we'll use a drop-in replacement, called Fashion-MNIST, consisting of different pieces of clothing.

Fashion-MNIST is a dataset of Zalando's article images consisting of a training set of 60,000 examples and a test set of 10,000 examples: <https://github.com/zalandoresearch/fashion-mnist>.

Here are a few examples from the dataset:



In this recipe, we'll recognize clothing items with different models – we'll start with generic image features (**Difference of Gaussians**, or **DoG**) and a support vector machine; then we'll move on to a feedforward **Multilayer Perceptron** (**MLP**); then we'll use a **Convolutional Neural Network** (**ConvNet**); and finally, we'll look at transfer learning with MobileNet.

Getting ready

Before we can start, we have to install a library. In this recipe, we'll use `scikit-image`, a library for image transformations, so we'll quickly set this up:

```
pip install scikit-image
```

We are now ready to move on to the recipe!

How to do it...

We'll first load and prepare the dataset, then we'll learn models for classifying clothing items from the Fashion-MNIST dataset using different approaches. Let's start by loading the Fashion-MNIST fashion dataset.

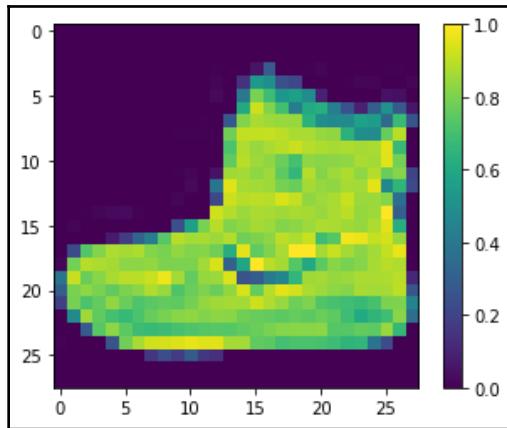
We can get the dataset directly via a `keras` utility function:

```
from tensorflow import keras
from matplotlib import pyplot as plt

(train_images, train_labels), (test_images, test_labels) =
keras.datasets.fashion_mnist.load_data()
train_images = train_images / 255.0
test_images = test_images / 255.0
plt.imshow(train_images[0])
plt.colorbar()
plt.grid(False)
```

We are also normalizing the images within the range of 0 and 1 by dividing by the maximal pixel intensity (255.0), and we visualize the first image.

We should see the following image of a sneaker, the first image in the training set:



As we mentioned in the introduction of this recipe, we'll apply different approaches in the upcoming sections:

- DoG features
- MLP
- LeNet
- Transfer learning with MobileNet

Let's start with DoG.

Difference of Gaussians

Before the breakthroughs of deep learning in image recognition, images were featurized using filters from the Difference of Laplacians or Gaussians. These are implemented in `scikit-image`, so we can take an off-the-shelf implementation.

Let's write a function that extracts image features using a Gaussian pyramid:

```
import skimage.transform
import numpy as np

def get_pyramid_features(img):
    return np.hstack([
        layer.reshape(-1)
        for layer in skimage.transform.pyramids.pyramid_gaussian(img)
    ])
```

The `get_pyramid_features()` function applies the Gaussian pyramid and returns these features flattened as a vector. We'll explain what a Gaussian pyramid is in the *How it works...* section.

We are nearly ready to start learning. We only need to iterate over all the images and extract our Gaussian pyramid features. Let's create another function that does that:

```
from sklearn.svm import LinearSVC

def featurize(x_train, y_train):
    data = []
    labels = []
    for img, label in zip(x_train, y_train):
        data.append(get_pyramid_features(img))
        labels.append(label)

    data = np.array(data)
    labels = np.array(labels)
    return data, labels
```

For training the model, we apply the `featurize()` function on our training dataset. We'll use a linear support vector machine as our model. Then, we'll apply this model to the features extracted from our test dataset - please note that this might take a while to run:

```
x_train, y_train = featurize(train_images, train_labels)
clf = LinearSVC(C=1, loss='hinge').fit(x_train, y_train)

x_val, y_val = featurize(test_images, test_labels)
print('accuracy: {:.3f}'.format(clf.score(x_val, y_val)))
```

We get 84% accuracy over the validation dataset from a linear support vector machine using these features. With some more tuning of the filters, we could have achieved higher performance, but that is beyond the scope of this recipe. Before the publication of AlexNet in 2012, this method was one of the state-of-the-art methods for image classification.

Another way to train a model is to flatten the images and feed the normalized pixel values directly into a classifier, such as an MLP. That is what we'll try now.

Multilayer perceptron

A relatively simple way to classify images is with an MLP. In this case of a two-layer MLP with 10 neurons, you can think of the hidden layer as a feature extraction layer of 10 feature detectors.

We have seen examples of MLPs already a few times in this book, so we'll skip over the details here; perhaps of interest is that we flatten images from 28x28 to a vector of 784. As for the rest, suffice it to say that we train for categorical cross-entropy and we'll monitor accuracy.

You'll see this in the following code block:

```
import tensorflow as tf
from tensorflow.keras.losses import SparseCategoricalCrossentropy

def compile_model(model):
    model.summary()
    model.compile(
        optimizer='adam',
        loss=SparseCategoricalCrossentropy(
            from_logits=True
        ),
        metrics=['accuracy']
    )

def create_mlp():
    model = tf.keras.Sequential([
        tf.keras.layers.Flatten(input_shape=(28, 28)),
        tf.keras.layers.Dense(128, activation='relu'),
        tf.keras.layers.Dense(10)
    ])
    compile_model(model)
    return model
```

This model has 101,770 trainable parameters between the two layers and their connections.

We'll use the following function to wrap our training set. It should be fairly self-explanatory:

```
def train_model(model, train_images, test_images):
    model.fit(
        train_images,
        train_labels,
        epochs=50,
        verbose=1,
        validation_data=(test_images, test_labels)
    )
    loss, accuracy = model.evaluate(test_images, test_labels, verbose=0)
    print('loss:', loss)
    print('accuracy:', accuracy)
```

After 50 epochs, our accuracy in the validation set is 0.886.

The next model is the classic ConvNet proposed for MNIST, employing convolutions, pooling, and fully connected layers.

LeNet5

LeNet5 is a feedforward neural network with convolutional layers and max pooling, as well as fully connected feedforward layers for features leading to the read-out. Let's see how it performs:

```
from tensorflow.keras.layers import (
    Conv2D, MaxPooling2D, Flatten, Dense
)

def create_lenet():
    model = tf.keras.Sequential([
        Conv2D(
            filters=6,
            kernel_size=(5, 5),
            padding='valid',
            input_shape=(28, 28, 1),
            activation='tanh'
        ),
        MaxPooling2D(pool_size=(2, 2)),
        Conv2D(
            filters=16,
            kernel_size=(5, 5),
            padding='valid',
            activation='tanh'
        ),
        MaxPooling2D(pool_size=(2, 2)),
        Flatten(),
        Dense(120, activation='tanh'),
        Dense(84, activation='tanh'),
        Dense(10, activation='softmax')
    ])
    compile_model(model)
    return model
```

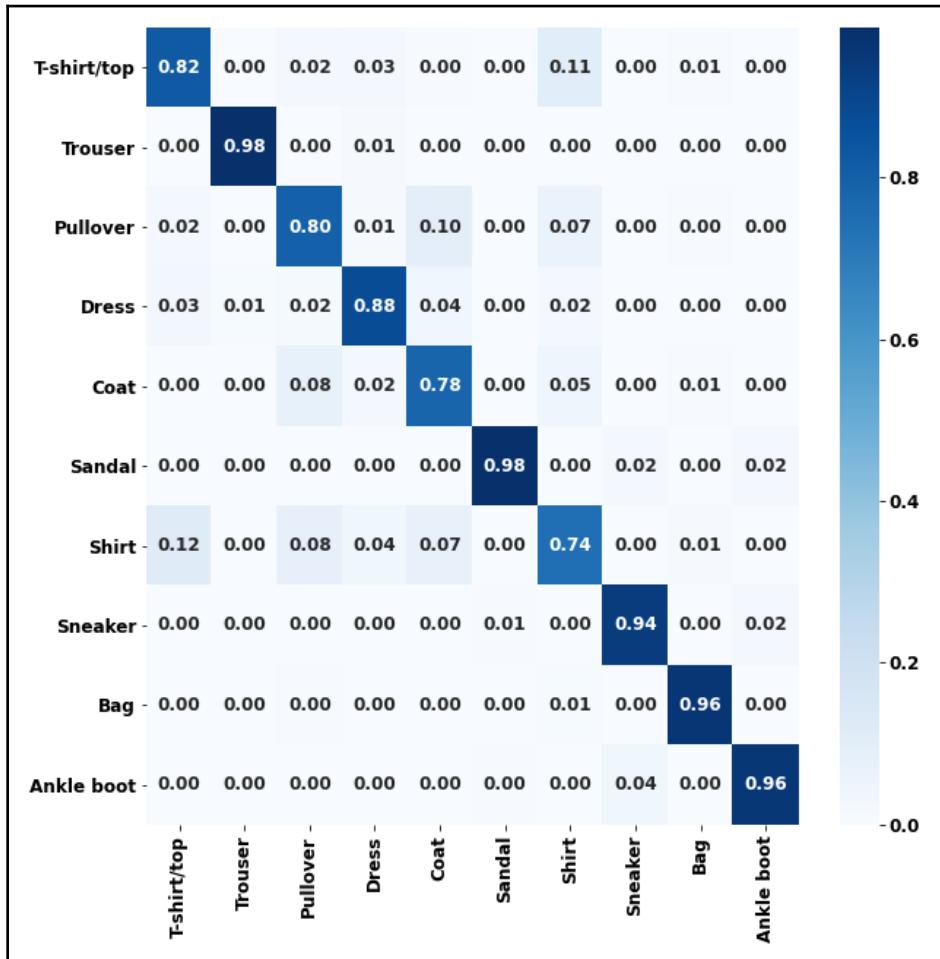
The `create_lenet()` function builds our model. We only have to call it and run our `train_model()` function with it in order to fit it to the training dataset and see our test performance:

```
train_model(
    create_lenet(),
```

```
train_images.reshape(train_images.shape + (1,)),
test_images.reshape(test_images.shape + (1,)),
)
```

After 50 episodes, our validation accuracy is at 0.891.

We can also have a look at the confusion matrix to see how well we distinguish particular pieces of clothing from others:



Let's move on to our last attempt at classifying the clothing items.

MobileNet transfer learning

The MobileNetV2 model was trained on ImageNet, a database of 14 million images that have been hand-annotated into categories of the WordNet taxonomy.

MobileNet can be downloaded with weights for transfer learning. This means that we leave most or all of MobileNet's weights fixed. In most cases, we would only add a new output projection in order to discriminate a new set of classes on top of the MobileNet representation:

```
base_model = tf.keras.applications.MobileNetV2(
    input_shape=(224, 224, 3),
    include_top=False,
    weights='imagenet'
)
```

MobileNet comprises 2,257,984 parameters. Downloading MobileNet, we have the convenient option of leaving out the output layer (`include_top=False`), which saves us work.

For our transfer model, we have to append a pooling layer, and then we can append an output layer just as in the previous two neural networks:

```
def create_transfer_model():
    base_model = tf.keras.applications.MobileNetV2(
        input_shape=(224, 224, 3),
        include_top=False,
        weights='imagenet'
    )
    base_model.trainable = False
    model = tf.keras.Sequential([
        base_model,
        tf.keras.layers.GlobalAveragePooling2D(),
        tf.keras.layers.Dense(10)
    ])
    compile_model(model)
    return model
```

Please note that we freeze or fix the weights in the MobileNet model, and only learn the two layers that we add on top.

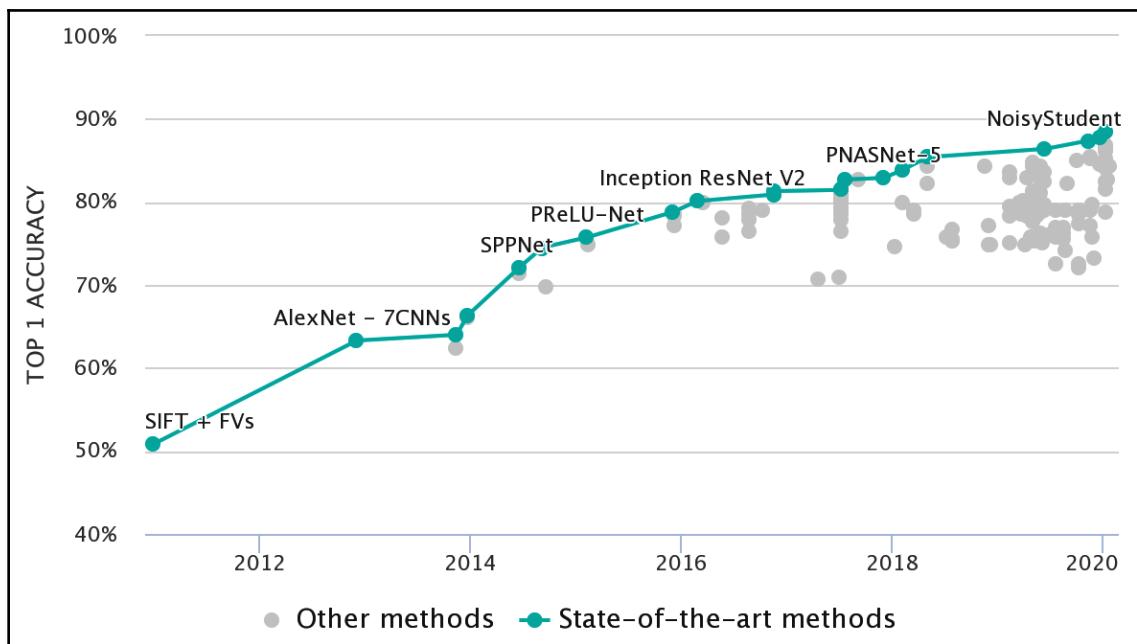
You might have noticed a detail when we downloaded MobileNet: we specified dimensions of 224x224x3. MobileNet comes with different input shapes, and 224x224x3 is one of the smallest. This means that we have to rescale our images and convert them to RGB (by concatenating the grayscales). You can find the details for that in the online notebook on GitHub.

The validation accuracy for MobileNet transfer learning is very similar to LeNet and our MLP: 0.893.

How it works...

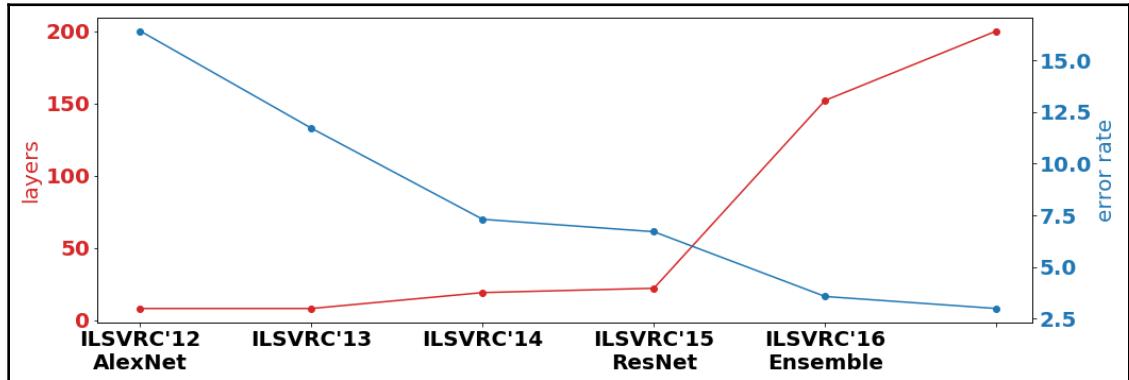
Image classification consists of assigning a label to an image, and this was where the deep learning revolution started.

The following graph, taken from the preceding URL, illustrates the performance increase for the ImageNet benchmark for image classification over time:



TOP 1 ACCURACY (also more simply called accuracy) on the y axis is a metric that measures the proportion of correct predictions over all predictions, or in other words, the ratio of how often an object was correctly identified. The **State-of-the-art** line on the graph has been continuously improving over time (the x axis), until now, reaching an 87.4% accuracy rate with the **NoisyStudent** method (see here for details: <https://paperswithcode.com/paper/self-training-with-noisy-student-improves>).

In the following graph, you can see a timeline of deep learning in image recognition, where you can see the increasing complexity (in terms of the number of layers) and the decreasing error rate in the **ImageNet Large-Scale Visual Recognition Challenge (ILSVRC)**:



You can find more details about the challenge at <http://www.image-net.org/challenges/LSVRC/>.

Difference of Gaussian

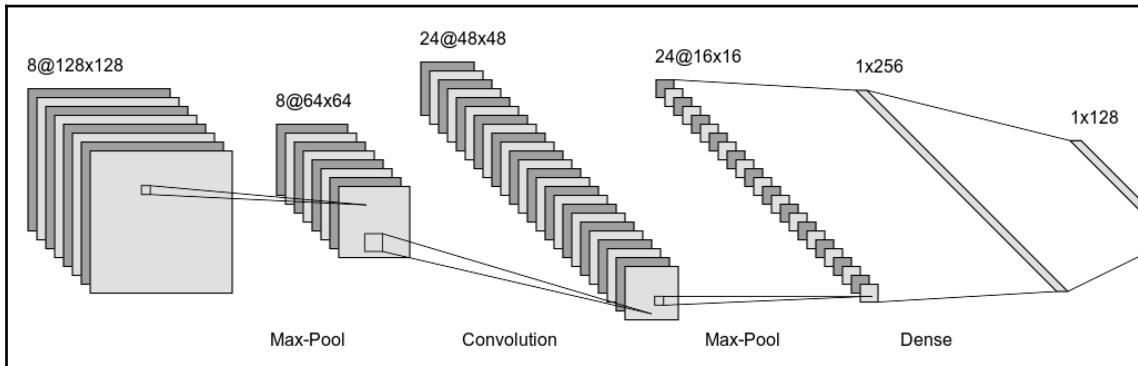
The Gaussian pyramid is a series of recursively downsampled versions of the original image, where the downscaling is performed using a Gaussian filter. You can find detailed information on Scholarpedia at http://www.scholarpedia.org/article/Scale_Invariant_Feature_Transform.

We used utility functions from `skimage` to extract the features, then we applied a linear support vector machine on top as the classifier. We could have tried other classifiers, such as random forest or gradient boosting, instead in order to improve the performance.

LeNet5

A CNN or ConvNet is a neural network with at least one convolutional layer. LeNet, a classic example of a ConvNet, was proposed by Yann LeCun and others in its original form in 1989 (*backpropagation applied to handwritten zip code recognition*) and in the revised form (called LeNet5) in 1998 (*gradient-based learning applied to document recognition*).

You can see the LeNet architecture in the following diagram (created using the NN-SVG tool at <http://alexlenail.me/NN-SVG>):



Convolutions are a very important transformation in image recognition and are one of the most important building blocks of very deep neural networks in image recognition. Convolutions consist of feedforward connections, called filters or kernels, which are applied to rectangular patches of the image (the previous layer). Each resulting map is then the sliding window of the kernel over the whole image. These convolutional maps are usually followed by subsampling by pooling layers (in the case of LeNet, the maximum of each kernel is extracted).

MobileNet transfer learning

MobileNets (Howard and others, *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*; 2017; <https://arxiv.org/abs/1704.04861>) are a class of models developed by Google for efficiency with mobile and embedded applications, keeping in mind explicitly trading off latency versus accuracy. MobileNet consists largely of stacked convolutional layers of different shapes. All convolutional layers are followed by batch normalization and ReLU activation. The final convolution is followed by an average pooling layer, which removes the spatial dimension, and a final dense read-out layer with a softmax function.

Loading the model is a matter of a single command in Keras.

The `tf.keras.applications` package exposes architectures and weights for many models, such as DenseNet, EfficientNet Inception-ResNet V2, Inception V3, MobileNetV1, MobileNetV2, NASNet-A, ResNet, ResNet v2, VGG16, VGG19, and Xception V1. In our case, we have a pre-trained model, which means it has architecture and weights with the `tf.keras.applications.MobileNetV2()` function.

We can retrain (fine-tune) the model to improve performance for our application, or we could use the model as is and put additional layers on top in order to classify new classes.

What we do to load the model is this:

```
base_model = tf.keras.applications.MobileNetV2(  
    input_shape=(224, 224, 3),  
    include_top=False,  
    weights='imagenet'  
)
```

As mentioned before, we have the option of specifying different input shapes from a selection of choices. `include_top` specifies whether we want to include the classification layer. If we wanted to use the original model output for the ImageNet dataset on which it has been trained, we would set this to `True`. Since we have different classes in our dataset, we want to set this to `False`.

If we wanted to fine-tune the model (with or without the top), we would leave the base model (MobileNetV2) trainable. Obviously, the training could take much longer that way since many more layers would need to be trained. That's why we've frozen all of MobileNetV2's layers during training, setting its `trainable` attribute to `False`.

See also

You can find a review of ConvNet, from LeNet over AlexNet to more recent architectures, in *A Survey of the Recent Architectures of Deep Convolutional Neural Networks* by Khan and others (2020), available from arXiv: <https://arxiv.org/pdf/1901.06032.pdf>.

A more recent architecture is EfficientNet (Mingxing Tan and Quoc V. Le, 2019) which achieves state-of-the-art performance on ImageNet, while close to a magnitude smaller and about five times faster than the best ConvNets: <https://arxiv.org/abs/1905.11946>.

Generating images

Adversarial learning with GANs, introduced by Ian Goodfellow and others in 2014, is a framework for fitting the distributions of a dataset by pairing two networks against each other in a way that one model generates examples and the others discriminate, whether they are real or not. This can help us to extend our dataset with new training examples. Semi-supervised training with GANs can help achieve higher performance in supervised tasks while using only small amounts of labeled training examples.

The focus of this recipe is implementing a **Deep Convolutional Generative Adversarial Network (DCGAN)** and a discriminator on the MNIST dataset, one of the best-known datasets, consisting of 60,000 digits between 0 and 9. We'll explain the terminology and background in the *How it works...* section.

Getting ready

We don't need any special libraries for this recipe. We'll use TensorFlow with Keras, NumPy, and Matplotlib, all of which we've seen earlier. For saving images, we'll use the Pillow library, which you can install or upgrade as follows:

```
pip install --upgrade Pillow
```

Let's jump right into it.

How to do it...

For our approach with a GAN, we need a generator – a network that takes some input, which could be noise – and a discriminator, an image classifier, such as the one seen in the *Recognizing clothing items* recipe of this chapter.

Both the generator and discriminator are deep neural networks, and the two will be paired together for training. After training, we'll see the training loss, example images over epochs, and a composite image of the final epoch.

First, we will design the discriminator.

This is a classic example of a ConvNet. It's a series of convolution and pooling operations (typically average or max pooling), followed by flattening and an output layer. For more details, please see the *Recognizing clothing items* recipe of this chapter:

```
def discriminator_model():
    model = Sequential([
        Conv2D(
            64, (5, 5),
            padding='same',
            input_shape=(28, 28, 1),
            activation='tanh'
        ),
        MaxPooling2D(pool_size=(2, 2)),
        Conv2D(128, (5, 5), activation='tanh'),
        MaxPooling2D(pool_size=(2, 2)),
        Flatten(),
        Dense(1024, activation='tanh'),
        Dense(1, activation='sigmoid')
    ])
    return model
```

This is very similar to the LeNet convolution block, which we introduced in the *Recognizing clothing items* recipe of this chapter.

Next, we design the generator.

While discriminators downsample their input with convolutions and pooling operations, generators upsample. Our generator takes an input vector of 100 dimensions and generates an image from this by performing the operations in the opposite direction of a ConvNet. For this reason, this type of network is sometimes referred to as a DeconvNet.

The output of the generator is normalized back within the range of -1 to 1 by the Tanh activation. One of the major contributions of the DCGAN paper (Alec Radford and others, 2015, *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*) was the introduction of batch normalization after the deconvolutional operation. In Keras, there are two ways to achieve deconvolution: one is by using UpSampling2D (see https://www.tensorflow.org/api_docs/python/tf/keras/layers/UpSampling2D), and the other by using Conv2DTranspose. Here, we chose UpSampling2D:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (
    Dense, Reshape, Activation,
    Flatten, BatchNormalization,
    UpSampling2D, Conv2D, MaxPooling2D
)
```

```
def create_generator_model():
    model = Sequential([
        Dense(input_dim=100, units=1024, activation='tanh'),
        Dense(128*7*7),
        BatchNormalization(),
        Activation('tanh'),
        Reshape((7, 7, 128), input_shape=(128*7*7,)),
        UpSampling2D(size=(2, 2)),
        Conv2D(64, (5, 5), padding='same'),
        Activation('tanh'),
        UpSampling2D(size=(2, 2)),
        Conv2D(1, (5, 5), padding='same'),
        Activation('tanh'),
    ])
    model.summary()
    return model
```

Calling this function, we'll get an output of our network architecture by using `summary()`. We'll see that we have 6,751,233 trainable parameters. We'd recommend running this on a powerful system – for example, Google Colab.

For training the network, we load the MNIST dataset and normalize it:

```
from tensorflow.keras.datasets import mnist

(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = (X_train.astype(np.float32) - 127.5) / 127.5
X_train = X_train[:, :, :, :, None]
X_test = X_test[:, :, :, :, None]
```

The images come in grayscale with pixel values of 0–255. We normalize into the range -1 and +1. We then reshape to give a singleton dimension at the end.

For the error to feed through to the generator, we chain the generator with the discriminator, as follows:

```
def chain_generator_discriminator(g, d):
    model = Sequential()
    model.add(g)
    model.add(d)
    return model
```

As our optimizer, we'll use Keras stochastic gradient descent:

```
from tensorflow.keras.optimizers import SGD

def optim():
    return SGD(
        lr=0.0005,
        momentum=0.9,
        nesterov=True
    )
```

Now, let's create and initialize our models:

```
d = discriminator_model()
g = generator_model()
d_on_g = chain_generator_discriminator(g, d)
d_on_g.compile(loss='binary_crossentropy', optimizer=optim())
d.compile(loss='binary_crossentropy', optimizer=optim())
```

A single training step consists of three steps:

- The generator generates images from noise.
- The discriminator learns to distinguish generated images from real images.
- The generator learns to create better images given the discriminator feedback.

Let's go through these in turn. First is generating images from noise:

```
import numpy as np

def generate_images(g, batch_size):
    noise = np.random.uniform(-1, 1, size=(batch_size, 100))
    image_batch = X_train[index*batch_size:(index+1)*batch_size]
    return g.predict(noise, verbose=0)
```

Then, the discriminator learns when given fake and real images:

```
def learn_discriminate(d, image_batch, generated_images, batch_size):
    X = np.concatenate(
        (image_batch, generated_images)
    )
    y = np.array(
        [1] * batch_size + [0] * batch_size
    )
    loss = d.train_on_batch(X, y)
    return loss
```

We concatenate true, 1, and fake, 0, images for the input to the discriminator.

Finally, the generator learns from the discriminator feedback:

```
def learn_generate(d_on_g, d, batch_size):
    noise = np.random.uniform(-1, 1, (batch_size, 100))
    d.trainable = False
    targets = np.array([1] * batch_size)
    loss = d_on_g.train_on_batch(noise, targets)
    d.trainable = True
    return loss
```

Please note the inversion of the discriminator target in this function. Instead of 0 (for fake, as before), we feed 1s. It is important to note as well that parameters in the discriminator are fixed during the learning of the generator (otherwise, we'd unlearn again).

We can incorporate additional commands into our training in order to save images so that we can appreciate our generator progress visually:

```
from PIL import Image

def save_images(generated_images, epoch, index):
    image = combine_images(generated_images)
    image = image*127.5+127.5
    Image.fromarray(
        image.astype(np.uint8)
    ).save('{ }_{ }.png'.format(epoch, index))
```

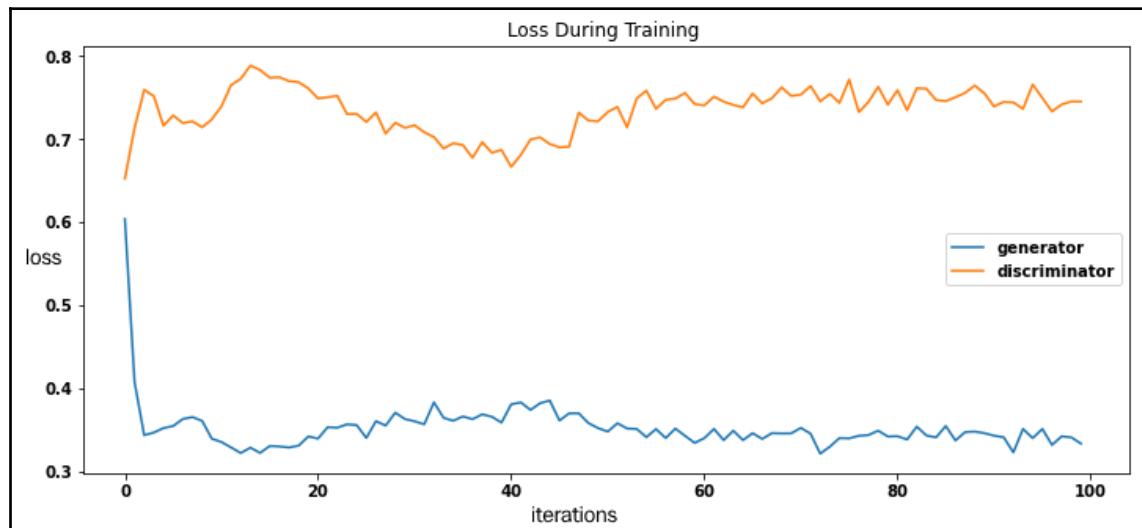
Our training works, then, by alternating the preceding steps:

```
from tqdm.notebook import trange

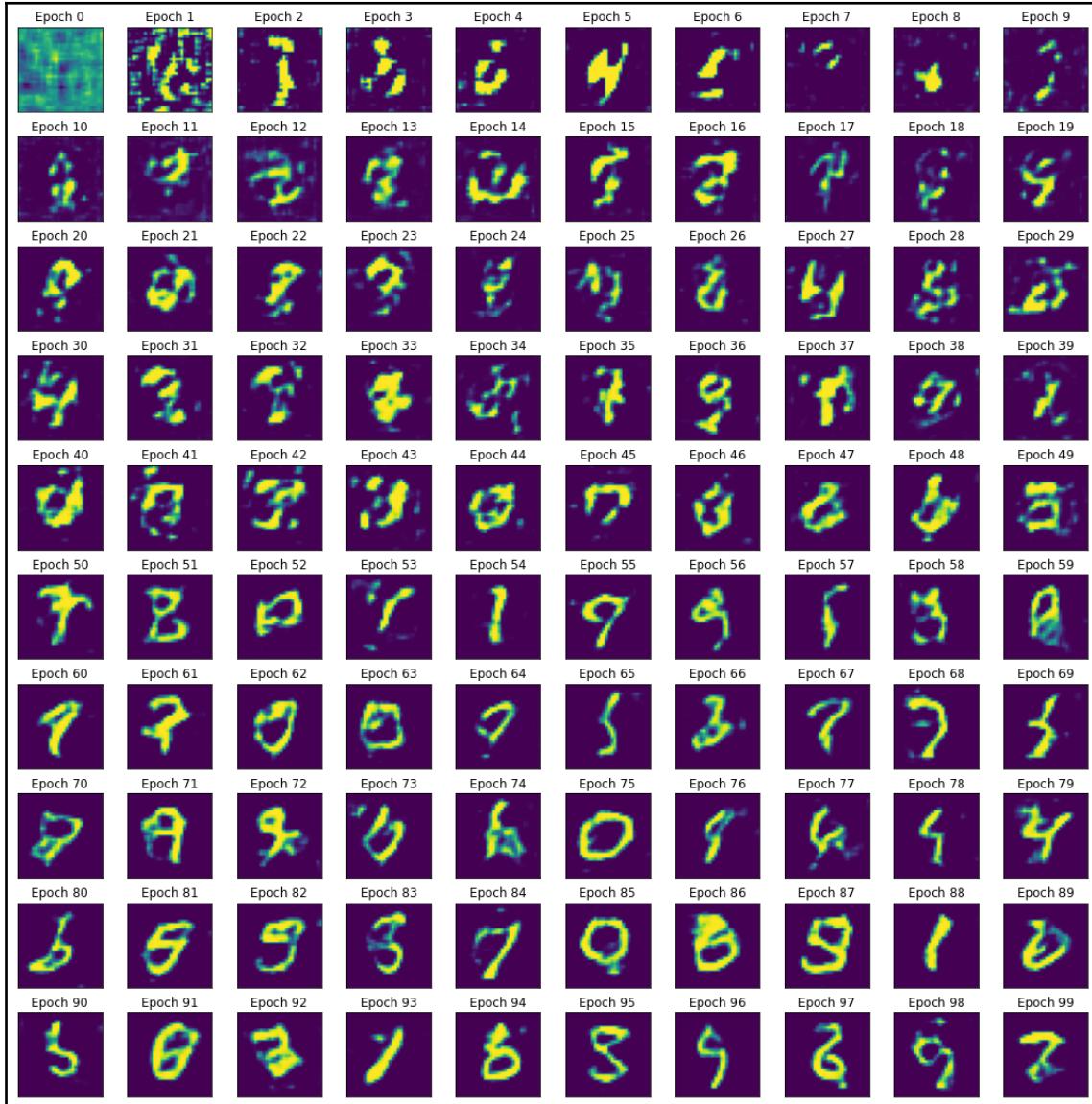
batch_size = 1024
generator_losses = []
discriminator_losses = []
for epoch in trange(100):
    for index in trange(nbatches):
        image_batch = X_train[index*batch_size:(index+1)*batch_size]
        generated_images = generate_images(g, batch_size)
        d_loss = learn_discriminate(
            d, image_batch, generated_images, batch_size
        )
        g_loss = learn_generate(d_on_g, d, batch_size)
        discriminator_losses.append(d_loss)
        generator_losses.append(g_loss)
        if (index % 20) == 0:
            save_images(generated_images, epoch, index)
```

We let it run. The `tqdm` progress bars will show us how much time is left. It might take about an hour on Google Colab.

Over 100 epochs, our training errors look as follows:



We have saved images – so we can also have a look at the generator output over epochs. Here's a gallery of a single randomly generated digit for each of the 100 epochs:



We can see that images generally become sharper and sharper. This is interesting, because the training error for the generator seems to stay at the same baseline after the first couple of epochs. Here's an image that shows the 100 generated images during the last epoch:



The images are not perfect, but most of them are recognizable as digits.

How it works...

Generative models can generate new data with the same statistics as the training set, and this can be useful for semi-supervised and unsupervised learning. GANs were introduced by Ian Goodfellow and others in 2014 (*Generative Adversarial Nets*, in NIPS; <https://papers.nips.cc/paper/5423-generative-adversarial-nets>) and DCGANs by Alec Radford and others in 2015 (*Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*; <https://arxiv.org/abs/1511.06434>). Since the original papers, many incremental improvements have been proposed.

In the GAN technique, the generative network learns to map from a seed – for example, randomized input to the target data distribution – while the discriminative network evaluates and discriminates data produced by the generator from the true data distribution.

The generative network generates data while the discriminative network evaluates them. The two neural networks compete with each other, where the generative network's training increases the error rate of the discriminative network and the discriminator training increases the error of the generator, thereby engaging in a weapons race, forcing each other to become better.

In training, we feed random noise into our generator and then let the discriminator learn how to classify generator output against genuine images. The generator is then trained given the output of the discriminator, or rather the inverse of it. The less likely the discriminator judges an image a fake, the better for the generator, and vice versa.

See also

The original GAN paper, *Generative Adversarial Networks* (Ian Goodfellow and others; 2014), is available from arXiv: <https://arxiv.org/abs/1406.2661>.

The DCGAN paper, *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks* (Alec Radford and others; 2015), is available on arXiv as well: <https://arxiv.org/abs/1511.06434>.

You can find a tutorial on DCGANs on the PyTorch website at https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html.

There are many more GAN architectures that are worth exploring. Erik Linder-Norén implemented dozens of state-of-the-art architectures in both PyTorch and Keras. You can find them in his GitHub repositories (<https://github.com/eriklindernoren/PyTorch-GAN> and <https://github.com/eriklindernoren/Keras-GAN>, respectively).

Encoding images and style

Autoencoders are useful for representing the input efficiently. In their 2016 paper, Makhzani and others showed that adversarial autoencoders can create clearer representations than variational autoencoders, and – similar to the DCGAN that we saw in the previous recipe – we get the added benefit of learning to create new examples, which can help in semi-supervised or supervised learning scenarios, and allow training with less labeled data. Representing in a compressed fashion can also help in content-based retrieval.

In this recipe, we'll implement an adversarial autoencoder in PyTorch. We'll implement both supervised and unsupervised approaches and show the results. There's a nice clustering by classes in the unsupervised approach, and in the supervised approach, our encoder-decoder architecture can identify styles, which gives us the ability to do style transfer. In this recipe, we'll use the *hello world* dataset of computer vision, MNIST.

Getting ready

We'll need `torchvision` for this recipe. This will help us download our dataset. We'll quickly install it:

```
!pip install torchvision
```

For PyTorch, we'll need to get a few preliminaries out of the way, such as to enable CUDA and set `tensor` type and `device`:

```
use_cuda = True
use_cuda = use_cuda and torch.cuda.is_available()
print(use_cuda)
if use_cuda:
    dtype = torch.cuda.FloatTensor
    device = torch.device('cuda:0')
else:
    dtype = torch.FloatTensor
    device = torch.device('cpu')
```

In a break from the style in other recipes, we'll also get the imports out of the way:

```
import numpy as np
import torch
from torch import autograd
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, dataset
from torchvision.datasets import MNIST
import torchvision.transforms as T
from tqdm.notebook import trange
```

Now, let's get to it.

How to do it...

We'll implement an adversarial autoencoder in this recipe and apply it to the MNIST dataset of digits. This code is based on the implementation by Maurits Diephuis and Shideh Rezaifar: <https://github.com/mdiephuis/adversarial-autoencoders>.

We'll first get the imports out of the way. Then, we'll load our dataset, define the model components, including the encoder, decoder, and discriminator, then we'll do our training, and finally, we'll visualize the resulting representations.

First is loading the dataset.

We'll need to set a few global variables that will define training and the dataset. Then, we load our dataset:

```
EPS = torch.finfo(torch.float32).eps
batch_size = 1024
n_classes = 10
batch_size = 1024
n_classes = 10

train_loader = torch.utils.data.DataLoader(
    MNIST(
        'Data/',
        train=True,
        download=True,
        transform=T.Compose([
            T.transforms.ToTensor(),
            T.Normalize((0.1307,), (0.3081,))
        ])
    ),
    batch_size=batch_size,
    shuffle=True
)

val_loader = torch.utils.data.DataLoader(
    MNIST(
        'Val/',
        train=False,
        download=True,
        transform=T.Compose([
            T.transforms.ToTensor(),
            T.Normalize((0.1307,), (0.3081,))
        ])
    ),
    batch_size=batch_size,
    shuffle=False
)
```

The transformation in the normalize corresponds to the mean and standard deviation of the MNIST dataset.

Next is defining the autoencoder model.

The autoencoder consists of an encoder, a decoder, and a discriminator. If you are familiar with autoencoders, this is nothing new for you. You'll find an explanation in the next section, *How it works...*, where this is broken down.

First, we'll define our encoder and decoder:

```
dims = 10
class Encoder(nn.Module):
    def __init__(self, dim_input, dim_z):
        super(Encoder, self).__init__()
        self.dim_input = dim_input # image size
        self.dim_z = dim_z
        self.network = []
        self.network.extend([
            nn.Linear(self.dim_input, self.dim_input // 2),
            nn.Dropout(p=0.2),
            nn.ReLU(),
            nn.Linear(self.dim_input // 2, self.dim_input // 2),
            nn.Dropout(p=0.2),
            nn.ReLU(),
            nn.Linear(self.dim_input // 2, self.dim_z),
        ])
        self.network = nn.Sequential(*self.network)
    def forward(self, x):
        z = self.network(x)
        return z
```

Please note `dim`, the parameter that stands for the size of the representational layer. We choose 10 as the size of our encoding layer.

Then, we'll define our decoder:

```
class Decoder(nn.Module):
    def __init__(self, dim_input, dim_z, supervised=False):
        super(Decoder, self).__init__()
        self.dim_input = dim_input
        self.dim_z = dim_z
        self.supervised = supervised
        self.network = []
        self.network.extend([
            nn.Linear(self.dim_z, self.dim_input // 2) if not
self.supervised
            else nn.Linear(self.dim_z + n_classes, self.dim_input // 2),
            nn.Dropout(p=0.2),
            nn.ReLU(),
            nn.Linear(self.dim_input // 2, self.dim_input // 2),
            nn.Dropout(p=0.2),
            nn.ReLU(),
            nn.Linear(self.dim_input // 2, self.dim_input),
            nn.Sigmoid(),
        ])
        self.network = nn.Sequential(*self.network)
```

```
def forward(self, z):
    x_recon = self.network(z)
    return x_recon
```

While we are at it, we can also define our discriminator to compete against our encoder:

```
class Discriminator(nn.Module):
    def __init__(self, dims, dim_h):
        super(Discriminator, self).__init__()
        self.dim_z = dims
        self.dim_h = dim_h
        self.network = []
        self.network.extend([
            nn.Linear(self.dim_z, self.dim_h),
            nn.Dropout(p=0.2),
            nn.ReLU(),
            nn.Dropout(p=0.2),
            nn.Linear(self.dim_h, self.dim_h),
            nn.ReLU(),
            nn.Linear(self.dim_h, 1),
            nn.Sigmoid(),
        ])
        self.network = nn.Sequential(*self.network)

    def forward(self, z):
        disc = self.network(z)
        return disc
```

Please note that we squash our outputs in order to stay within the range of 0 and 1. This will become important for our loss function.

Then comes training the model.

The adversarial autoencoder can be used in a supervised fashion, where labels are fed into the decoder in addition to the encoding output. In that case, we'll need one more utility function, which one-hot encodes our variables:

```
def one_hot_encoding(labels, n_classes=10):
    cat = np.array(labels.data.tolist())
    cat = np.eye(n_classes)[cat].astype('float32')
    cat = torch.from_numpy(cat)
    return autograd.Variable(cat)
```

We'll show how to use the adversarial autoencoder with and without labels:

```
def train_validate(
    encoder,
    decoder,
    Disc,
    dataloader,
    optim_encoder,
    optim_decoder,
    optim_D,
    train):
    total_rec_loss = 0
    total_disc_loss = 0
    total_gen_loss = 0
    if train:
        encoder.train()
        decoder.train()
        Disc.train()
    else:
        encoder.eval()
        decoder.eval()
        Disc.eval()

    iteration = 0
    for (data, labels) in dataloader:
        # [ training loop here, see next code segment ]

    M = len(dataloader.dataset)
    return total_rec_loss / M, total_disc_loss / M, total_gen_loss / M
```

As you can see in the comment, we've broken out the training loop. The training loop looks as follows:

```
for (data, labels) in dataloader:
    # Reconstruction loss:
    for p in Disc.parameters():
        p.requires_grad = False

    real_data_v = autograd.Variable(data).to(device).view(-1, 784)
    encoding = encoder(real_data_v)

    if decoder.supervised:
        categories = one_hot_encoding(labels, n_classes=10).to(device)
        decoded = decoder(torch.cat((categories, encoding), 1))
    else:
        decoded = decoder(encoding)

    reconstruction_loss = F.binary_cross_entropy(decoded, real_data_v)
```

```
total_rec_loss += reconstruction_loss.item()
if train:
    optim_encoder.zero_grad()
    optim_decoder.zero_grad()
    reconstruction_loss.backward()
    optim_encoder.step()
    optim_decoder.step()

encoder.eval()
z_real_gauss = autograd.Variable(
    torch.randn(data.size()[0], dims) * 5.0
).to(device)
z_fake_gauss = encoder(real_data_v)
D_real_gauss = Disc(z_real_gauss)
D_fake_gauss = Disc(z_fake_gauss)

D_loss = -torch.mean(
    torch.log(D_real_gauss + EPS) +
    torch.log(1 - D_fake_gauss + EPS)
)
total_disc_loss += D_loss.item()

if train:
    optim_D.zero_grad()
    D_loss.backward()
    optim_D.step()

if train:
    encoder.train()
else:
    encoder.eval()
z_fake_gauss = encoder(real_data_v)
D_fake_gauss = Disc(z_fake_gauss)

G_loss = -torch.mean(torch.log(D_fake_gauss + EPS))
total_gen_loss += G_loss.item()

if train:
    optim_encoder_reg.zero_grad()
    G_loss.backward()
    optim_encoder_reg.step()

if (iteration % 100) == 0:
    print(
        'reconstruction loss: %.4f, discriminator loss: %.4f , '
        'generator loss: %.4f' %
        (reconstruction_loss.item(), D_loss.item(), G_loss.item()))
iteration += 1
```

For this code segment, we'll discuss in the *How it works...* section how three different losses are calculated and back-propagated. Please also note the supervised parameter, which defines whether we want to use supervised or unsupervised training.

Now, let's initialize our models and optimizers:

```
encoder = Encoder(784, dims).to(device)
decoder = Decoder(784, dims, supervised=True).to(device)
Disc = Discriminator(dims, 1500).to(device)

lr = 0.001
optim_encoder = torch.optim.Adam(encoder.parameters(), lr=lr)
optim_decoder = torch.optim.Adam(decoder.parameters(), lr=lr)
optim_D = torch.optim.Adam(Disc.parameters(), lr=lr)
optim_encoder_reg = torch.optim.Adam(encoder.parameters(), lr=lr * 0.1)
```

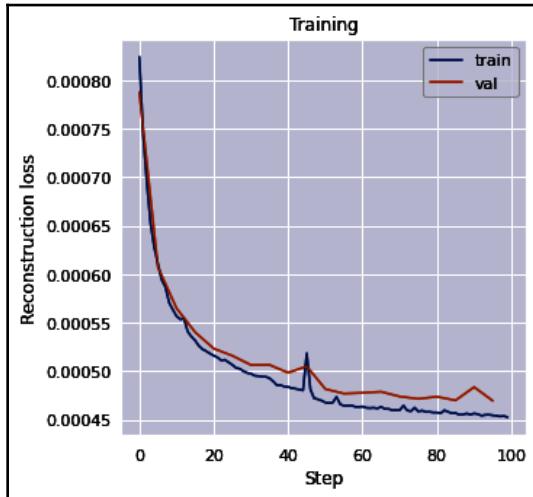
Now we can start training:

```
train_loss = []
val_loss = []
for epoch in range(n_epochs):
    l1, l2, l3 = train_validate(
        encoder, decoder, Disc,
        train_loader, optim_encoder, optim_decoder,
        optim_D, train=True
    )
    print('epoch: {} ---- training loss: {:.8f}'.format(epoch, l1))
    train_loss.append(l1)

    if (epoch % 5) == 0:
        l1, l2, l3 = train_validate(
            encoder, decoder, Disc,
            val_loader, optim_encoder,
            optim_decoder, optim_D, False
        )
        print('epoch: {} ---- validation loss: {:.8f}'.format(epoch, l1))
        val_loss.append(l1)
```

This does nothing much except for calling the `train_validate()` function defined previously, once with the `train=True` option and once with `train=False`. From both calls, we collect the errors for training and validation, respectively.

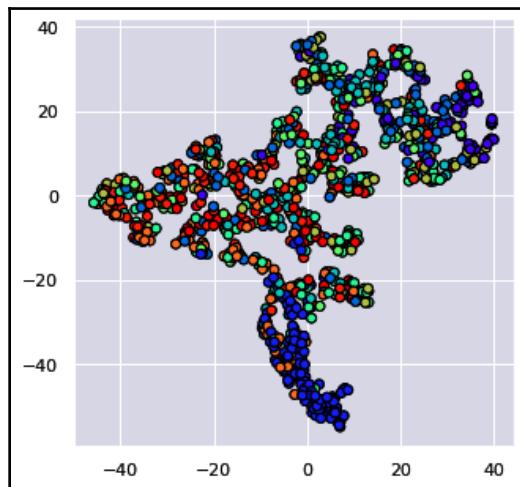
The training and validation errors go consistently down, as we can see in the following graph:



If you run this, compare the generator and discriminator losses – it's interesting to see how the generator and discriminator losses drive each other.

The next step is visualizing representations.

In the supervised condition, the projections of the encoder space do not have much to do with the classes, as you can see in the following `tsne` plot:



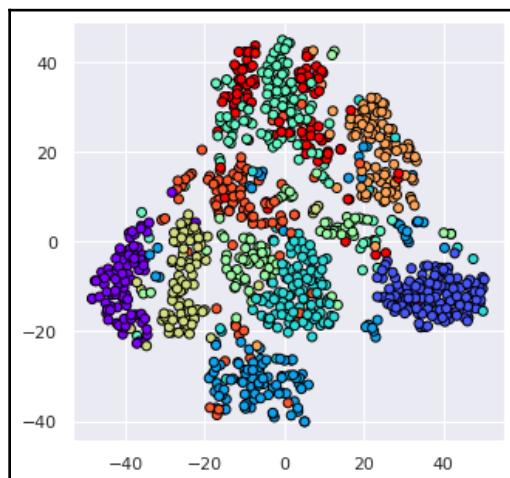
This is a 2D visualization of the encoder's representation space of the digits. The colors (or shades if you are looking at this in black and white), representing different digits, are all grouped together, rather than being separated into clusters. The encoder is not distinguishing between different digits at all.

What is encoded is something else entirely, which is style. In fact, we can vary the input into the decoder on each of the two dimensions separately in order to show this:



The first five rows correspond to a linear range of the first dimension with the second dimension kept constant, then in the next five rows, the first dimension is fixed and the second dimension is varied. We can appreciate that the first dimension corresponds to the thickness and the second to inclination. This is called style transfer.

We can also try the unsupervised training, by setting `supervised=False`. We should see a projection like this, where classes are clustered in the 2D space of the `tsne` projection:



This is a 2D visualization of the encoder's representation space of the digits. Each color (or shade) represents a different digit. We can see different clusters that group together instances of the same digit. The encoder is distinguishing between different digits.

In the next section, we'll discuss how this works.

How it works...

An **autoencoder** is a network of two parts – an encoder and a decoder – where the encoder maps the input into a latent space and the decoder reconstructs the input. Autoencoders can be trained to reconstruct the input by a reconstruction loss, which is often the squared error between the original input and the restored input.

Adversarial autoencoders were introduced in 2016 by Makhzani and others (*Adversarial Autoencoders*; <https://arxiv.org/pdf/1511.05644.pdf>). The publication showed how they could be used in clustering and semi-supervised training, or how they could decorrelate class representations. An adversarial autoencoder is a probabilistic autoencoder that uses a GAN to perform variational inference. Instead of matching input to output directly, the aggregated posterior of the hidden code vector of the autoencoder is matched to an arbitrary prior distribution.

Since adversarial autoencoders are GANs, and therefore rely on the competition between generator and discriminator, the training is a bit more involved than it would be for a vanilla autoencoder. We calculate three different types of errors:

- The standard reconstruction error
- An error of the discriminator that quantifies a failure to distinguish between real random numbers and encoder output
- The encoder error that penalizes the encoder for failing to trick the discriminator

In our case, we force prior distribution and decoder output within the range 0 and 1, and we can therefore use cross-entropy as the reconstruction error.

It might be helpful to highlight the code segments responsible for calculating the different types of error.

The reconstruction error looks like this:

```
if decoder.supervised:  
    categories = one_hot_encoding(labels, n_classes=10).to(device)  
    decoded = decoder(torch.cat((categories, encoding), 1))  
else:  
    decoded = decoder(encoding)  
reconstruction_loss = F.binary_cross_entropy(decoded, real_data_v)
```

There's an extra flag for feeding the labels into the decoder as supervised training. We found that in the supervised setting, the encoder doesn't represent the digits, but rather the style. We argue that this is the case since, in the supervised setting, the reconstruction error doesn't depend on the labels anymore.

The discriminator loss is calculated as follows:

```
# i) latent representation:  
encoder.eval()  
z_real_gauss = autograd.Variable(  
    torch.randn(data.size()[0], dims) * 5.0  
) .to(device)  
z_fake_gauss = encoder(real_data_v)  
# ii) feed into discriminator  
D_real_gauss = Disc(z_real_gauss)  
D_fake_gauss = Disc(z_fake_gauss)  
  
D_loss = -torch.mean(  
    torch.log(D_real_gauss + EPS) +  
    torch.log(1 - D_fake_gauss + EPS)  
)
```

Please note that this is for training the discriminator, not the encoder, hence the `encoder.eval()` statement.

The generator loss is calculated as follows:

```
if train:  
    encoder.train()  
else:  
    encoder.eval()  
z_fake_gauss = encoder(real_data_v)  
D_fake_gauss = Disc(z_fake_gauss)  
  
G_loss = -torch.mean(torch.log(D_fake_gauss + EPS))  
total_gen_loss += G_loss.item()
```

In the next section, we'll look at more material and background.

See also

For a more up-to-date and fuller-featured implementation of adversarial autoencoders, please refer to a repository maintained by Maurits Diephuis at <https://github.com/mdiephuis/generative-models>.

For more historical and mathematical background on autoencoders and adversarial autoencoders, please see Lilian Weng's excellent overview article, *From Autoencoder to Beta-VAE*, on her blog: <https://lilianweng.github.io/lil-log/2018/08/12/from-autoencoder-to-beta-vae.html>.

Finally, please have a look at Maurits Diephuis and others, *Variational Saccading: Efficient Inference for Large Resolution Images* (Ramapuram and others, 2019). They introduce a probabilistic algorithm for focusing on regions of interest in bigger images inspired by the eye's saccade movements. You can find their code on GitHub at https://github.com/jramapuram/variational_saccading.

8

Working with Moving Images

This chapter deals with video applications. While methods applied to images can be applied to single frames of videos, this usually comes with a loss of temporal consistency. We will try to strike a balance between what's possible on consumer hardware and what's interesting enough to show and implement.

Quite a few applications should come to mind when talking about video, such as object tracking, event detection (surveillance), deep fake, 3D scene reconstruction, and navigation (self-driving cars).

A lot of them require many hours or days of computation. We'll try to strike a sensible compromise between what's possible and what's interesting. This compromise might be felt more than in other chapters, where computations are not as demanding as for video. As part of this compromise, we'll work on videos frame by frame, rather than across the temporal domain. Still, as always, we'll try to work on problems by giving examples that are either representative of practical real-world applications, or that are at least similar.

In this chapter, we'll start with image detection, where an algorithm applies an image recognition model to different parts of an image in order to localize objects. We'll then give examples of how to apply this to a video feed. We'll then create videos using a deep fake model, and reference more related models for both creating and detecting deep fakes.

In this chapter, we'll look at the following recipes:

- Localizing objects
- Faking videos

Technical requirements

We'll use many standard libraries, including `keras` and `opencv`, but we'll see a few more libraries that we'll mention at the beginning of each recipe before they'll become relevant.

You can find the notebooks for this chapter's recipes on GitHub at <https://github.com/PacktPublishing/AI-with-Python-Cookbook/tree/master/chapter08>.

Localizing objects

Object detection refers to identifying objects of particular classes in images and videos. For example, in self-driving cars, pedestrians and trees have to be identified in order to be avoided.

In this recipe, we'll implement an object detection algorithm in Keras. We'll apply it to a single image and then to our laptop camera. In the *How it works...* section, we'll discuss the theory and more algorithms for object detection.

Getting ready

For this recipe, we'll need the Python bindings for the **Open Computer Vision Library (OpenCV)** and `scikit-image`:

```
!pip install -U opencv-python scikit-image
```

As our example image, we'll download an image from an object detection toolbox:

```
def download_file(url: str, filename='demo.jpg'):
    import requests
    response = requests.get(url)
    with open(filename, 'wb') as f:
        f.write(response.content)

download_file('https://raw.githubusercontent.com/open-mmlab/mmdetection/master/demo/demo.jpg')
```

Please note that any other image will do.

We'll use a code based on the `keras-yolo3` library, which was quick to set up with only a few changes. We can quickly download this as well:

```
download_file('https://gist.githubusercontent.com/benman1/51b2e4b10365333f0af34f4839f86f27/raw/991b41e5d5d83174d3d75b55915033550e16adf8/kerasyolo3.py', 'keras_yolo3.py')
```

Finally, we also need the weights for the YOLOv3 network, which we can download from the darknet open source implementation:

```
download_file('https://pjreddie.com/media/files/yolov3.weights',  
'yolov3.weights')
```

You should now have the example image, the `yolo3-keras` Python script, and the YOLOv3 network weights in your local directory from which you run your notebook.

How to do it...

In this section, we'll implement an object detection algorithm with Keras.

We'll import the `keras-yolo3` library, load the pretrained weights, and then perform object detection given images or the video feed from a camera:

1. Since we have most of the object detection implemented in the `keras-yolo3` script, we only need to import it:

```
from keras_yolo3 import load_model, detect
```

2. We can then load our network with the pretrained weights as follows. Please note that the weight files are quite big – they'll occupy around 237 MB of disk space:

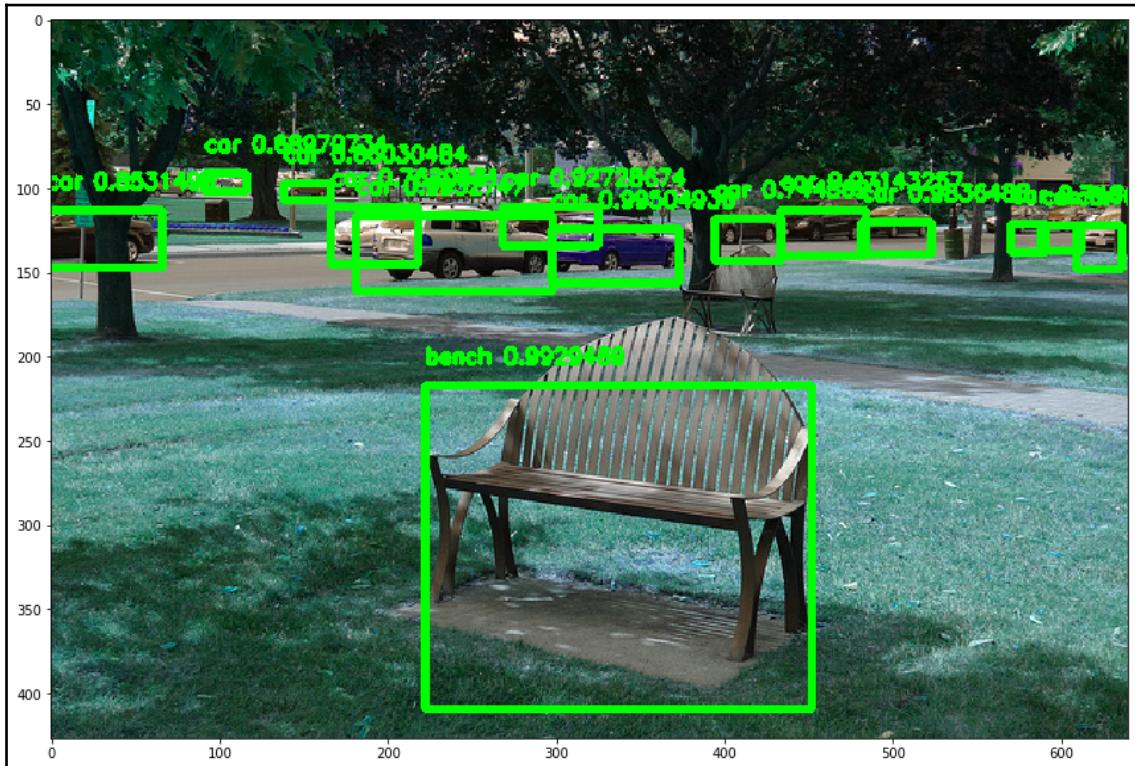
```
yolov3 = load_model('yolov3.weights')
```

Our model is now available as a Keras model.

3. We can then perform the object detection on our example image:

```
from matplotlib import pyplot as plt  
  
plt.imshow(detect(yolov3, 'demo.jpg'))
```

We should see our example image annotated with labels for each bounding box, as can be seen in the following screenshot:



We can extend this for videos using the OpenCV library. We can capture images frame by frame from a camera attached to our computer, run the object detection, and show the annotated image.



Please note that this implementation is not optimized and might run relatively slowly. For faster implementations, please refer to the darknet implementation linked in the *See also* section.

When you run the following code, please know that you can stop the camera by pressing q:

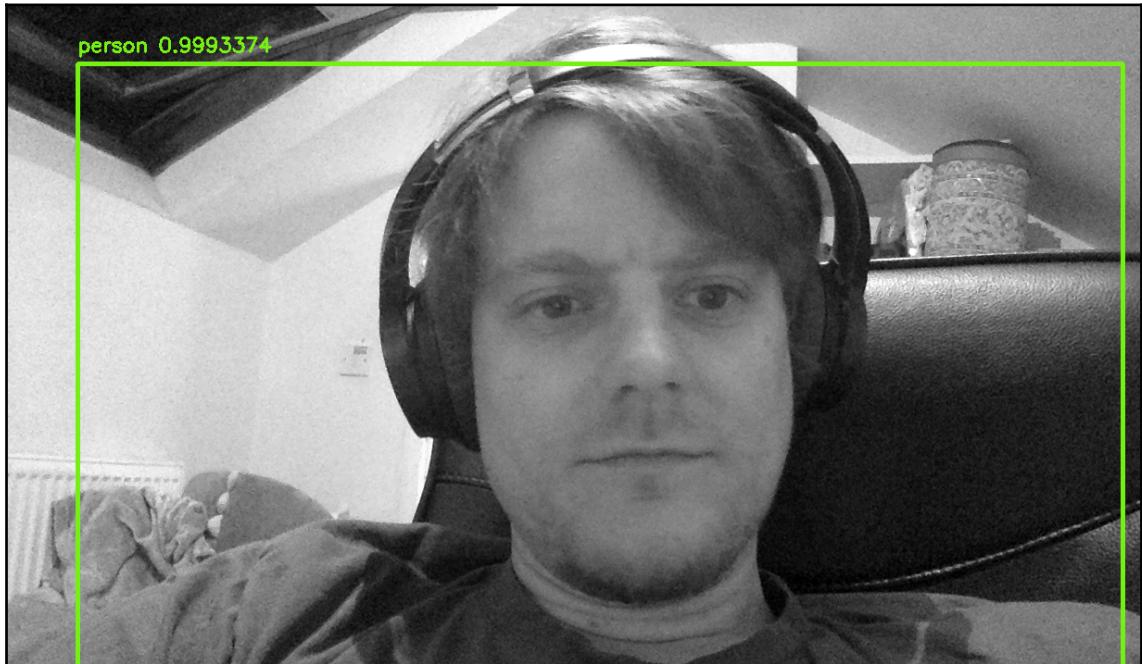
```
import cv2
from skimage import color

cap = cv2.VideoCapture(0)
```

```
while(True):  
    ret, frame = cap.read()  
  
    img = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)  
    img = detect(yolov3, img)  
    cv2.imshow('frame', img)  
    if cv2.waitKey(1) & 0xFF == ord('q'):  
        break  
  
cap.release()  
cv2.destroyAllWindows()
```

We capture our image as grayscale, but then have to convert it back to RGB using scikit-image by stacking the image. Then we detect objects and show the annotated frame.

This is the image we obtained:



In the next section, we'll discuss this recipe with some background explanations.

How it works...

We've implemented an object detection algorithm with Keras. This came out of the box with a standard library, but we connected it to a camera and applied it to an example image.

The main algorithms in terms of image detection are the following:

- Fast R-CNN (Ross Girshick, 2015)
- **Single Shot MultiBox Detector (SSD)**; Liu and others, 2015: <https://arxiv.org/abs/1512.02325>)
- **You Only Look Once (YOLO)**; Joseph Redmon and others, 2016: <https://arxiv.org/abs/1506.02640>)
- YOLOv4 (Alexey Bochkovskiy and others, 2020: <https://arxiv.org/abs/2004.10934>)

One of the main requirements of object detection is speed – you don't want to wait to hit the tree before recognizing it.

Image detection is based on image recognition with the added complexity of searching through the image for candidate locations.

Fast R-CNN is an improvement over R-CNN by the same author (2014). Each region of interest, a rectangular image patch defined by a bounding box, is scale normalized by image pyramids. The convolutional network can then process these object proposals (from a few thousand to as many as many thousands) through a single forward pass of a convolutional neural network. As an implementation detail, Fast R-CNN compresses fully connected layers with singular value decomposition for speed.

YOLO is a single network that proposed bounding boxes and classes directly from images in a single evaluation. It was much faster than other detection methods at the time; in their experiments, the author ran different versions of YOLO at 45 frames per second and 155 frames per second.

The SSD is a single-stage model that does away with the need for a separate object proposal generation, instead of opting for a discrete set of bounding boxes that are passed through a network. Predictions are then combined across different resolutions and bounding box locations.

As a side note, Joseph Redmon published and maintained several incremental improvements of his YOLO architecture, but he has since left academia. The latest instantiation of the YOLO series by Bochkovskiy and others is in the same spirit, however, and is endorsed on Redmon's GitHub repository: <https://github.com/AlexeyAB/darknet>.

YOLOv4 introduces several new network features to their CNN and they exhibit fast processing speed, while maintaining a level of accuracy significantly superior to YOLOv3 (43.5% **average precision (AP)**, for the MS COCO dataset at a real-time speed of about 65 frames per seconds on a Tesla V100 GPU).

There's more...

There are different ways of interacting with a web camera, and there are even some mobile apps that allow you to stream your camera feed, meaning you can plug it into applications that run on the cloud (for example, Colab notebooks) or on a server.

One of the most common libraries is `matplotlib`, and it is also possible to live update a `matplotlib` figure from the web camera, as shown in the following code block:

```
%matplotlib notebook
import cv2
import matplotlib.pyplot as plt

def grab_frame(cap):
    ret, frame = cap.read()
    if not ret:
        print('No image captured!')
        exit()
    return cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

cap = cv2.VideoCapture(0)
fig, ax = plt.subplots(1, 1)
im = ax.imshow(grab_frame(cap))

plt.tick_params(
    top=False, bottom=False, left=False, right=False,
    labelleft=False, labelbottom=False
)
plt.show()

while True:
    try:
        im.set_data(grab_frame(cap))
        fig.canvas.draw()
    except KeyboardInterrupt:
        cap.release()
        break
```

This is the basic template for initiating your video feed, and showing it in a matplotlib subfigure. We can stop by interrupting the kernel.

We'll mention a few more libraries to play with in the next section.

See also

We recommend having a look at the YOLOv4 paper available on arxiv: <https://arxiv.org/abs/2004.10934>.

A few libraries are available regarding object detection:

- The first author of the YOLOv4 paper is maintaining an open source framework supporting object detection, darknet (originally developed by Joseph Redmon) at <https://github.com/AlexeyAB/darknet>.
- Detectron2 is a library by Facebook AI Research implementing many algorithms for object detection: <https://github.com/facebookresearch/detectron2>.
- The Google implementations in TensorFlow for object detection, including the recently published SpineNet (<https://ai.googleblog.com/2020/06/spinenet-novel-architecture-for-object.html>), are available on GitHub: <https://github.com/tensorflow/models/tree/master/official/vision/detection>.
- Valkka Live is an open source Python video surveillance platform: https://elsamps.github.io/valkka-live/_build/html/index.html.
- MMDetection is an open detection toolbox that covers many popular detection methods and comes with pretrained weights for about 200 network models: <https://mmdetection.readthedocs.io/en/latest/>.
- SpineNet is a new model, found using a massive exploration of hyperparameters, for object detection in TensorFlow: <https://github.com/tensorflow/models/tree/master/official/vision/detection>.
- PyTracking is a library for object tracking and video object segmentation with many powerful, state-of-the-art models based on PyTorch, which can be directly plugged on top of the webcam input: <https://github.com/visionml/pytracking>.
- PySlowFast supplies many pretrained models for video classification and detection tasks: <https://github.com/facebookresearch/SlowFast>.
- An implementation of models for real-time hand gesture recognition with PyTorch is available here: <https://github.com/ahmetgunduz/Real-time-GesRec>.

Let's move on to the next recipe!

Faking videos

A deep fake is a manipulated video produced by the application of deep learning. Potential unethical uses have been around in the media for a while. You can imagine how this would end up in the hands of a propaganda mechanism trying to destabilize a government. Please note that we are advising against producing deep fakes for nefarious purposes.

There are ethical applications of the deep fake technology, and some of them are a lot of fun. Have you ever wondered how Sylvester Stallone may have looked in Terminator? Today you can!

In this recipe, we'll learn how to create a deep fake with Python. We'll download public domain videos of two films, and we'll produce a deep fake by replacing one face with another. *Charade* was a 1963 film directed by Stanley Donen in a style reminiscent of a Hitchcock film. It pairs off Cary Grant in his mid-fifties and Audrey Hepburn in her early 30s. We thought we'd make the pairing more age-appropriate. After some searching, what we found was Maureen O'Hara in the 1963 John Wayne vehicle *McLintock!* to replace Audrey Hepburn.

Getting ready

Faceit is a wrapper around the `faceswap` library, which facilitates many of the tasks that we'll need to perform for deep fake. We've forked the faceit repository at <https://github.com/benman1/faceit>.

What we have to do is download the faceit repository and install the requisite library.

You can download (`clone`) the repository with `git` (add an exclamation mark if you are typing this in an `ipython` notebook):

```
git clone https://github.com/benman1/faceit
```

We found that a Docker container was well-suited for the installation of dependencies (for this you need Docker installed). We can create a Docker container like this:

```
./dockerbuild.sh
```

This should take a while to build. Please note that the Docker image is based on Nvidia's container, so you can use your GPU from within the container.

Please note that, although there is a lightweight model that we could use, we'd highly recommend you run the deep fake on a machine with a GPU.

Finally, we can enter our container as follows:

```
./dockerrun.sh
```

Inside the container, we can run Python 3.6. All of the following commands assume we are inside the container and in the /project directory.

How to do it...

We need to define videos and faces as inputs to our deep fake process.

1. We define our model in Python (Python 3.6) like this:

```
from faceit import *

faceit = FaceIt('hepburn_to_ohara', 'hepburn', 'ohara')
```

This makes it clear that we want to replace `hepburn` with `ohara` (this is how we name them inside our process). We have to put images inside the `data/persons` directories named `hepburn.jpg` and `ohara.jpg`, accordingly. We have provided these images for convenience as part of the repository.



If we don't provide the images, `faceit` will extract all face images irrespective of whom they show. We can then place two of these images for the `persons` directory, and delete the directories with faces under `data/processed/`.

2. We then need to define the videos that we want to use. We have the choice of using the complete films or short clips. We didn't find good clips for the *McLintock!* film, so we are using the whole film. As for *Charade*, we've focused on the clip of a single scene. We have these clips on disk as `mclintock.mp4` and `who_trust.mp4`.

Please note that you should only download videos from sites that permit or don't disallow downloading, even of public domain videos:

```
faceit.add('ohara', 'mclintock.mp4')
faceit.add('hepburn', 'who_trust.mp4')
FaceIt.add_model(faceit)
```

This defines the data used by our model as a couple of videos. Faceit allows an optional third parameter that can be a link to a video, from where it can be downloaded automatically. However, before you are downloading videos from YouTube or other sites, please make sure this is permitted in their terms of service and legal within your jurisdiction.

3. The creation of the deep fake is then initiated by a few more lines of code (and a lot of tweaking and waiting):

```
faceit.preprocess()  
faceit.train()  
faceit.convert('who_trust.mp4', face_filter=True, photos=False)
```

The preprocess step consists of downloading the videos, extracting all the frames as images, and finally extracting the faces. We are providing the faces already, so you don't have to perform the preprocess step.

The following image shows Audrey Hepburn on the left, and Maureen O'Hara playing Audrey Hepburn on the right:



The changes might seem subtle. If you want something clearer, we can use the same model to replace Cary Grant with Maureen O'Hara:



In fact, we could produce a film, *Being Maureen O'Hara*, by disabling the face filter in the conversion.

We could have used more advanced models, more training to improve the deep fake, or we could have chosen an easier scene. However, the result doesn't look bad at all sometimes. We've uploaded our fake video to YouTube, where you can view it: <https://youtu.be/vDLxg5qXz4k>.

How it works...

The typical deep fake pipeline consists of a number of steps that we conveniently glossed over in our recipe, because of the abstractions afforded in faceit. These steps are the following, given person A and person B, where A is to be replaced by B:

- Choose videos of A and B, and split the videos into frames.
- Extract faces from these frames for A and B using face recognition. These faces, or the facial expressions and face postures, should ideally be representative of the video you are going to fake. Make sure they are not blurred, not occluded, don't show anyone else other than A and B, and that they are not too repetitive.
- You can train a model on these faces that can take face A and replace it with B. You should let the training run for a while. This can take several days, or even weeks or months, depending on the number of faces and the complexity of the model.

- We can now convert the video by running the face recognition and the model on top of it.

In our case, the face recognition library (`face-recognition`) has a very good performance in terms of detection and recognition. However, it still suffers from high false positives, but also false negatives. This can result in a poor experience, especially in frames where there are several faces.

In the current version of the `faceswap` library, we would extract frames from our target video in order to get landmarks for all the face alignments. We can then use the GUI in order to manually inspect and clean up these alignments in order to make sure they contain the right faces. These alignments will then be used for the conversion: <https://forum.faceswap.dev/viewtopic.php?t=27#align>.

Each of these steps requires a lot of attention. At the heart of the whole operation is the model. There can be different models, including a generative adversarial autoencoder and others. The original model in `faceswap` is an autoencoder with a twist. We've used autoencoders before in [Chapter 7, Advanced Image Applications](#). This one is relatively conventional, and we could have taken our autoencoder implementation from there. However, for the sake of completeness, we'll show its implementation, which is based on `keras/tensorflow` (shortened):

```
from keras.models import Model
from keras.layers import Input, Dense, Flatten, Reshape
from keras.layers.advanced_activations import LeakyReLU
from keras.layers.convolutional import Conv2D
from keras.optimizers import Adam
from lib.PixelShuffler import PixelShuffler

IMAGE_SHAPE = (64, 64, 3)
ENCODER_DIM = 1024

def conv(self, filters):
    def block(x):
        x = Conv2D(filters, kernel_size=5, strides=2, padding='same')(x)
        x = LeakyReLU(0.1)(x)
        return x
    return block

def upscale(self, filters):
    def block(x):
        x = Conv2D(filters * 4, kernel_size=3, padding='same')(x)
        x = LeakyReLU(0.1)(x)
        x = PixelShuffler()(x)
        return x
```

```
    return block

def Encoder():
    input_ = Input(shape=IMAGE_SHAPE)
    x = input_
    x = conv(128)(x)
    x = conv(256)(x)
    x = conv(512)(x)
    x = conv(1024)(x)
    x = Dense(ENCODER_DIM)(Flatten()(x))
    x = Dense(4 * 4 * 1024)(x)
    x = Reshape((4, 4, 1024))(x)
    x = upscale(512)(x)
    return Model(input_, x)

def Decoder():
    input_ = Input(shape=(8, 8, 512))
    x = input_
    x = upscale(256)(x)
    x = upscale(128)(x)
    x = upscale(64)(x)
    x = Conv2D(3, kernel_size=5, padding='same', activation='sigmoid')(x)
    return Model(input_, x)
```

This code, in itself, is not terribly interesting. We have two functions, `Decoder()` and `Encoder()`, which return decoder and encoder models, respectively. This is an encoder-decoder architecture with convolutions. The `PixelShuffle` layer in the upscale operation of the decoder rearranges data from depth into blocks of spatial data through a permutation.

Now, the more interesting part of the autoencoder is in how the training is performed as two models:

```
optimizer = Adam(lr=5e-5, beta_1=0.5, beta_2=0.999)
x = Input(shape=IMAGE_SHAPE)

encoder = Encoder()
decoder_A, decoder_B = Decoder(), Decoder()
autoencoder_A = Model(x, decoder_A(encoder(x)))
autoencoder_B = Model(x, decoder_B(encoder(x)))

autoencoder_A.compile(optimizer=optimizer, loss='mean_absolute_error')
autoencoder_B.compile(optimizer=optimizer, loss='mean_absolute_error')
```

We have two autoencoders, one to be trained on A faces and one on B faces. Both autoencoders are minimizing the reconstruction error (measured in mean absolute error) of output against input. As mentioned, we have a single encoder that forms part of the two models, and is therefore going to be trained both on faces A and faces B. The decoder models are kept separate between the two faces. This architecture ensures that we have a common latent representation between A faces and B faces. In the conversion, we can take a face from A, represent it, and then apply the decoder for B in order to get a B face corresponding to the latent representation.

See also

We've put together some further references relating to playing around with videos and deep fakes, as well as detecting deep fakes.

Deep fakes

We've collated a few links relevant to deep fakes and some more links that are relevant to the process of creating deep fakes.

The face recognition library has been used in this recipe to select image regions for training and application of the transformations. It is available on GitHub at https://github.com/ageitgey/face_recognition.

There are some nice examples of simple video faking applications:

- OpenCV masks can be used to selectively manipulate parts of an image, for example, for an invisibility cloak: <https://www.geeksforgeeks.org/invisibility-cloak-using-opencv-python-project/>.
- A similar effort has been made to add mustaches based on detected face landmarks: <http://sublimerobots.com/2015/02/dancing-mustaches/>.

As for more complex video manipulations with deep fakes, quite a few tools are available, of which we'll highlight two:

- The faceswap library has a GUI and even a few guides: <https://github.com/deepfakes/faceswap>.
- DeepFaceLab is a GUI application for creating deep fakes: <https://github.com/iperov/DeepFaceLab>.

Many different models have been proposed and implemented, including the following:

- **ReenactGAN – Learning to Reenact Faces via Boundary Transfer** (2018, <https://arxiv.org/abs/1807.11079>)
 - Official implementation: <https://github.com/wywu/ReenactGAN>.
- **DiscoGAN** – Taeksoo Kim and others, *Learning to Discover Cross-Domain Relations with Generative Adversarial Networks* (2017, <https://arxiv.org/abs/1703.05192>)
 - Demonstrated for live video feeds here: <https://github.com/ptrblck/DiscoGAN>.
- A denoising adversarial autoencoder with attention mechanisms for face swapping can be seen here: <https://github.com/shaoanlu/faceswap-GAN>.

The faceit live repository (https://github.com/alew3/faceit_live) is a fork of faceit that can operate on live video feeds and comes with a hack to feed the video back to prank participants in video conferences.

Detection of deep fakes

The following links are relevant to detecting deep fakes:

- Yuchen Luo has collected lots of links relating to the detection of deep fakes: <https://github.com/592McAvoy/fake-face-detection>.
- Of particular interest is detection via adversarial attacks, as can be found here: <https://github.com/natanielruiz/disrupting-deepfakes>.
- Google, Google Jigsaw, the Technical University of Munich, and the University Federico II of Naples provide an extensive dataset of deep fakes for the study of detection algorithms: <https://github.com/ondyari/FaceForensics/>.

The paper *DeepFakes and Beyond: A Survey of Face Manipulation and Fake Detection* (Ruben Tolosana and others, 2020) provides more links and more resources to datasets and methods.

9

Deep Learning in Audio and Speech

In this chapter, we'll deal with sounds and speech. Sound data comes in the form of waves, and therefore requires different preprocessing than other types of data.

Machine learning on audio signals finds commercial applications in speech enhancement (for example, in hearing aids), speech-to-text and text-to-speech, noise cancellation (as in headphones), recommending music to users based on their preferences (such as Spotify), and generating audio. Many fun problems can be encountered in audio, including the classification of music genres, the transcription of music, generating music, and many more besides.

We'll implement several applications with sound and speech in this chapter. We'll first do a simple example of a classification task, where we try to distinguish different words. This would be a typical application in a smart home device to distinguish different commands. We'll then look at a text-to-speech architecture. You could apply this to create your own audio books from text, or for the voice output of your home-grown smart home device. We'll close with a recipe for generating music. This is perhaps more of a niche application in the commercial sense, but you could build your own music for fun or to entertain users of your video game.

In this chapter, we'll look at the following recipes:

- Recognizing voice commands
- Synthesizing speech from text
- Generating melodies

Technical requirements

You can find the source code for the notebooks associated with the recipes in this chapter on GitHub at <https://github.com/PacktPublishing/Artificial-Intelligence-with-Python-Cookbook/tree/master/chapter09>.

We'll use the `librosa` audio processing library (<https://librosa.org/doc/latest/index.html>) in this chapter, which you can install as follows:

```
!pip install librosa
```

Librosa comes installed by default in Colab.

For the recipes in this chapter, please make sure you have a GPU available. On Google Colab, make sure you activate a GPU runtime.

Recognizing voice commands

In this recipe, we will look at a simple sound recognition problem on Google's Speech Commands dataset. We'll classify sound commands into different classes. We'll then set up a deep learning model and train it.

Getting ready

For this recipe, we'll need the `librosa` library as mentioned at the start of the chapter. We'll also need to download the Speech Commands dataset, and for that we'll need to install the `wget` library first:

```
!pip install wget
```

Alternatively, we could use the `!wget` system command in Linux and macOS. We'll create a new directory, download the archive with the dataset, and extract the `tarfile`:

```
import os
import wget
import tarfile

DATA_DIR = 'sound_commands'
DATASET_URL =
'http://download.tensorflow.org/data/speech_commands_v0.01.tar.gz'
ARCHIVE = os.path.basename(DATASET_URL)
os.mkdir(DATA_DIR)
```

```
os.chdir(DATA_DIR)
wget.download(DATASET_URL)
with tarfile.open(ARCHIVE, 'r:gz') as tar:
    tar.extractall(path='data/train')
os.remove(ARCHIVE)
```

This gives us a number of files and directories within the `data/train` directory:

```
_background_noise_  five      marvin      right      tree
bed                four      nine       seven      two
bird               go        no        sheila     up
cat                happy     off       six
validation_list.txt
dog                house     on        stop       wow
down               left      one       testing_list.txt yes
eight              LICENSE   README.md three     zero
```

Most of these refer to speech commands; for example, the `bed` directory contains examples of the `bed` command.

With all of this available, we are now ready to start.

How to do it...

In this recipe, we'll train a neural network to recognize voice commands. This recipe is inspired by the TensorFlow tutorial on speech commands at https://www.tensorflow.org/tutorials/audio/simple_audio.

We'll first perform data exploration, then we'll import and preprocess our dataset for training, and then we will create a model, train it, and check its performance in validation:

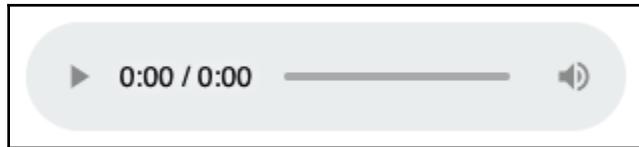
1. Let's start with some data exploration: we'll listen to a command, look at its waveform, and then at its spectrum. The `librosa` library provides functionality to load sound files into a vector:

```
import librosa
x, sr = librosa.load('data/train/bed/58df33b5_nohash_0.wav')
```

We can also get a Jupyter widget for listening to sound files or to the loaded vector:

```
import IPython.display as ipd  
ipd.Audio(x, rate=sr)
```

The widget looks like this:

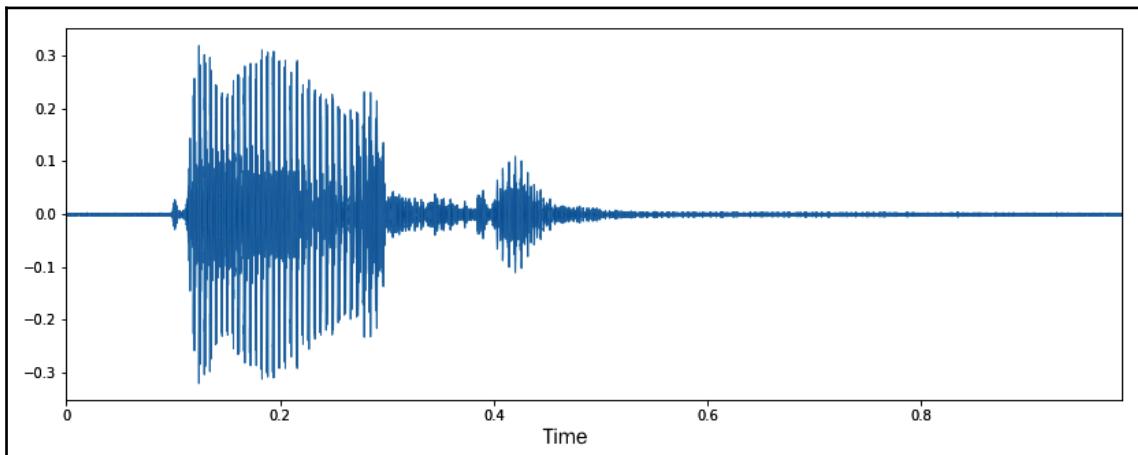


Pressing play, we hear the sound. Note that this works even over a remote connection, for example, if we use Google Colab.

Let's look at the sound waveform now:

```
%matplotlib inline  
import matplotlib.pyplot as plt  
import librosa.display  
  
plt.figure(figsize=(14, 5))  
librosa.display.waveplot(x, sr=sr, alpha=0.8)
```

The waveform looks like this:

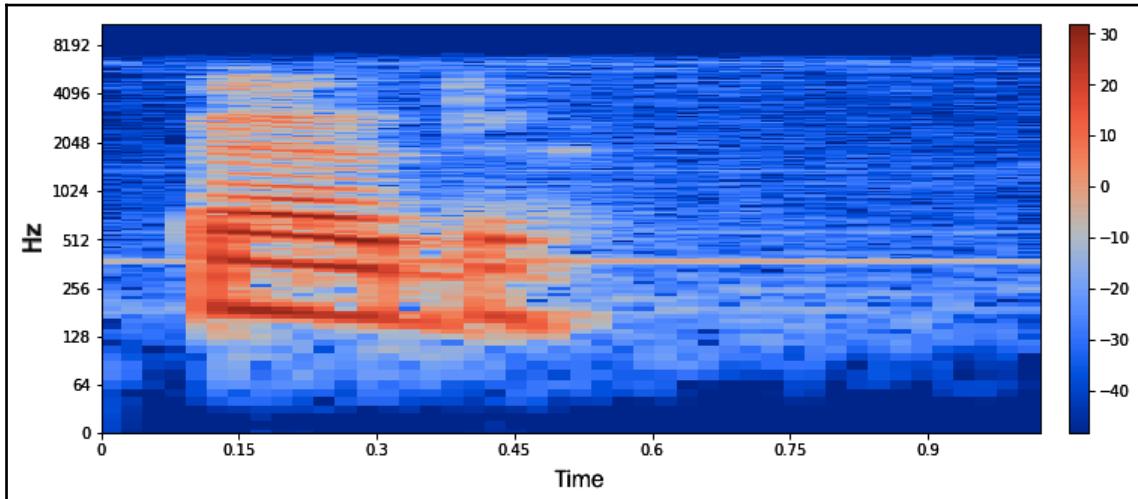


This is also called the pressure-time plot, and shows the (signed) amplitude over time.

We can plot the spectrum as follows:

```
X = librosa.stft(x)
Xdb = librosa.amplitude_to_db(abs(X))
plt.figure(figsize=(14, 5))
librosa.display.specshow(Xdb, sr=sr, x_axis='time', y_axis='log')
plt.colorbar()
```

The spectrum looks like this:



Please note that we've used a log scale on the y -axis.

2. Now, let's get to the data importing and preprocessing. We have to iterate over files, and store them as a vector:

```
from tqdm.notebook import tqdm

def vectorize_directory(dirpath, label=0):
    features = []
    labels = [0]
    files = os.listdir(dirpath)
    for filename in tqdm(files):
        x, _ = librosa.load(
            os.path.join(dirpath, filename)
        )
        if len(x) == 22050:
            features.append(x)
    return features, [label] * len(features)
```

```
features, labels = vectorize_directory('data/train/bed/')
f, l = vectorize_directory('data/train/bird/', 1)
features.extend(f)
labels.extend(l)
f, l = vectorize_directory('data/train/tree/', 2)
features.extend(f)
labels.extend(l)
```

For simplicity, we are only taking three commands here: `bed`, `bird`, and `tree`. This is enough to illustrate the problems and the application of a deep neural network to sound classification, and is simple enough that it won't take very long. This process can, however, still take a while. It took about an hour on Google Colab.

Finally, we need to convert the Python list of features to a NumPy array, and we need to split the training and validation data:

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

features = np.concatenate([f.reshape(1, -1) for f in features],
axis=0)
labels = np.array(labels)
X_train, X_test, y_train, y_test = train_test_split(
    features, labels, test_size=0.33, random_state=42
)
```

Now we need to do something with our training data. We'll need a model that we can train.

3. Let's create a deep learning model and then train and test it. First we need to create our model and normalization. Let's do the normalization first:

```
import tensorflow.keras as keras
from tensorflow.keras.layers import *
from tensorflow.keras.regularizers import l2
from tensorflow.keras.models import Model
import tensorflow.keras.backend as K

def preprocess(x):
    x = (x + 0.8) / 7.0
    x = K.clip(x, -5, 5)
    return x

Preprocess = Lambda(preprocess)
```

This is followed by the following:

```
def relu6(x):
    return K.relu(x, max_value=6)

def conv_layer(x, num_filters=100, k=3, strides=2):
    x = Conv1D(
        num_filters,
        (k),
        padding='valid',
        use_bias=False,
        kernel_regularizer=l2(1e-6)
    )(x)
    x = BatchNormalization()(x)
    x = Activation(relu6)(x)
    x = MaxPool1D(pool_size=num_filters, strides=None,
padding='valid')(x)
    return x

def create_model(classes, nlayers=1, filters=100, k=100):
    input_layer = Input(shape=[features.shape[1]])
    x = Preprocess(input_layer)
    x = Reshape([features.shape[1], 1])(x)
    for _ in range(nlayers):
        x = conv_layer(x, num_filters=filters, k=k)
        x = Reshape([219 * filters])(x)
        x = Dense(
            units=len(classes), activation='softmax',
            kernel_regularizer=l2(1e-2)
        )(x)
    model = Model(input_layer, x, name='conv1d_sound')
    model.compile(
        optimizer=keras.optimizers.Adam(lr=3e-4),
        loss=keras.losses.SparseCategoricalCrossentropy(),
        metrics=[keras.metrics.sparse_categorical_accuracy])
    model.summary()
    return model

model = create_model(classes)
```

Please note the `conv_layer()` function, which provides the core of the network. Very similar convolutional modules can be used in vision, it is just that we use 1D convolutions here.

This gives us a relatively small model of only about 75,000 parameters:

Layer (type)	Output Shape	Param #
input_46 (InputLayer)	[(None, 22050)]	0
lambda_44 (Lambda)	(None, 22050)	0
reshape_86 (Reshape)	(None, 22050, 1)	0
conv1d_56 (Conv1D)	(None, 21951, 100)	10000
batch_normalization_43 (BatchNormalization)	(None, 21951, 100)	400
activation_43 (Activation)	(None, 21951, 100)	0
max_pooling1d_29 (MaxPooling1D)	(None, 219, 100)	0
reshape_87 (Reshape)	(None, 21900)	0
dense_33 (Dense)	(None, 3)	65703
<hr/>		
Total params:	76,103	
Trainable params:	75,903	
Non-trainable params:	200	

You'll note that the biggest layer (in terms of parameters) is the final dense layer. We could have further reduced the number of parameters by changing the convolutional or maxpooling operations before the dense layer.

We can now perform training and validation:

```
import sklearn

model.fit(X_train, y_train, epochs=30)
predicted = model.predict(X_test)
print('accuracy: {:.3f}'.format(
    sklearn.metrics.accuracy_score(y_test, predicted.argmax(axis=1))
))
```

We should see something like 0.805 as the output for the model accuracy in the validation set.

How it works...

Sound is not that different from other domains, except for the preprocessing. It's important to have at least a basic understanding about how sound is stored in a file. At their most basic level, sounds are stored as amplitude over time and frequency. Sounds are sampled at discrete intervals (this is the *sampling rate*). 48 kHz would be a typical recording quality for a DVD, and refers to a sampling frequency of 48,000 times per second. The *bit depth* (also known as the *dynamic range*) is the resolution for the amplitude of the signal (for example, 16 bits means a range of 0-65,535).

For machine learning, we can do feature extraction from the waveform, and use 1D convolutions on the raw waveforms, or 2D convolutions on the spectrogram representation (for example, Mel spectrograms – Davis and Mermelstein, *Experiments in syllable-based recognition of continuous speech*, 1980). We've dealt with convolutions before, in Chapter 7, *Advanced Image Applications*. Briefly, convolutions are feedforward filters that are applied to rectangular patches over the layer input. The resulting maps are usually followed by subsampling by pooling layers.

The convolutional layers can be stacked very deeply (for example, Dai and others, 2016: <https://arxiv.org/abs/1610.00087>). We've made it easy for the reader to experiment with stacked layers. The number of layers, `nlayers`, is one of the parameters in `create_model()`.

Interestingly, many speech recognition models use recurrent neural networks. However, some models, such as Facebook's wav2letter (<https://github.com/facebookresearch/wav2letter>), for example, use a fully convolutional model instead, which is not too dissimilar to the approach taken in this recipe.

See also

Apart from `librosa`, useful libraries for audio processing in Python include `pydub` (<https://github.com/jiaaro/pydub>) and `scipy`. The `pyAudioProcessing` library comes with feature extraction and classification functionality for audio: <https://github.com/jsingh811/pyAudioProcessing>.

There are a few more libraries and repositories that are interesting to explore:

- `wav2letter++` is an open source speech processing toolkit from the speech team at Facebook AI Research with Python bindings: <https://github.com/facebookresearch/wav2letter>.

- A project from a master's thesis – *Structured Autoencoder with Application to Music Genre Recognition*: <https://github.com/mdeff/dlaudio>.
- Erdene-Ochir Tuguldur maintains a GitHub repository for Mongolian speech recognition with PyTorch that includes training from scratch: <https://github.com/tugstugi/mongolian-speech-recognition>.

Synthesizing speech from text

A text-to-speech program, easily intelligible by humans, can allow people with visual or reading impairments to listen to written words on a home computer, or can allow you to enjoy a book while driving a car. In this recipe, we'll work through loading a text-to-speech model, and having it read a text to us. In the *How it works...* section, we'll go through the model implementation and the model architecture.

Getting ready

For this recipe, please make sure you have a GPU available. On Google Colab, make sure you activate a GPU runtime. We'll also need the `wget` library, which we can install from the notebook as follows:

```
!pip install wget
```

We also need to clone the `pytorch-dc-tts` repository from GitHub and install its requirements. Please run this from the notebook (or run it from the terminal without the leading exclamation marks):

```
from os.path import exists  
  
if not exists('pytorch-dc-tts'):  
    !git clone --quiet https://github.com/tugstugi/pytorch-dc-tts  
  
!pip install --ignore-installed librosa
```

Please note that you need to have Git installed in order for this to work. If you don't have Git installed, you can download the repository directly from within your web browser.

We are ready to tackle the main recipe.

How to do it...

We'll download the Torch model files, load them up in Torch, and then we'll synthesize speech from sentences:

1. **Downloading the model files:** We'll download the dataset from dropbox:

```
import wget

if not exists('ljspeech-text2mel.pth'):
    wget.download(
        'https://www.dropbox.com/s/4t13ugxzzgnocbj/step-300K.pth',
        'ljspeech-text2mel.pth'
    )

if not exists('ljspeech-ssrn.pth'):
    wget.download(
        'https://www.dropbox.com/s/gw4aqrgcvccmg0g/step-100K.pth',
        'ljspeech-ssrn.pth'
    )
```

Now we can load the model in torch.

2. **Loading the model:** Let's get the dependencies out of the way:

```
import sys
sys.path.append('pytorch-dc-tts')
import numpy as np
import torch
import IPython
from IPython.display import Audio
from hparams import HParams as hp
from audio import save_to_wav
from models import Text2Mel, SSRN
from datasets.lj_speech import vocab, idx2char, get_test_data
```

Now we can load the model:

```
torch.set_grad_enabled(False)
text2mel = Text2Mel(vocab)
text2mel.load_state_dict(torch.load('ljspeech-
text2mel.pth').state_dict())
text2mel = text2mel.eval()
ssrn = SSRN()
ssrn.load_state_dict(torch.load('ljspeech-ssrn.pth').state_dict())
ssrn = ssrn.eval()
```

Finally, we can read sentences out loud.

3. Synthesizing speech: We've chosen a few garden-path sentences. These are sentences that are grammatically correct, but mislead the reader regarding their initial understanding of it.

The following sentences are examples of garden-path sentences – sentences that mislead listeners about how words relate to one another. We chose them because they are short and fun. You can find these and more garden-path sentences in the academic literature, such as in *Up the Garden Path* (Tomáš Gráf; published in Acta Universitatis Carolinae Philologica, 2013):

```
SENTENCES = [
    'The horse raced past the barn fell.',
    'The old man the boat.',
    'The florist sent the flowers was pleased.',
    'The cotton clothing is made of grows in Mississippi.',
    'The sour drink from the ocean.',
    'Have the students who failed the exam take the supplementary.',
    'We painted the wall with cracks.',
    'The girl told the story cried.',
    'The raft floated down the river sank.',
    'Fat people eat accumulates.'
]
```

We can generate speech from these sentences as follows:

```
for i in range(len(SENTENCES)):
    sentences = [SENTENCES[i]]
    max_N = len(sentences[0])
    L = torch.from_numpy(get_test_data(sentences, max_N))
    zeros = torch.from_numpy(np.zeros((1, hp.n_mels, 1),
                                    np.float32))
    Y = zeros
    A = None

    for t in range(hp.max_T):
        _, Y_t, A = text2mel(L, Y, monotonic_attention=True)
        Y = torch.cat((zeros, Y_t), -1)
        _, attention = torch.max(A[0, :, -1], 0)
        attention = attention.item()
        if L[0, attention] == vocab.index('E'): # EOS
            break

    _, Z = ssrn(Y)
    Z = Z.cpu().detach().numpy()
    save_to_wav(Z[0, :, :].T, '%d.wav' % (i + 1))
    IPython.display.display(Audio('%d.wav' % (i + 1), rate=hp.sr))
```

In the *There's more...* section, we'll have a look at how to train a model for a different dataset.

How it works...

Speech synthesis is the production of human speech by a program, called a speech synthesizer. A synthesis from natural language to speech is called **text-to-speech (TTS)**. Synthesized speech can be generated by concatenating audio from recorded pieces that come in units such as distinct sounds, phones, and pairs of phones (diphones).

Let's look a bit into the details of two methods.

Deep Convolutional Networks with Guided Attention

In this recipe, we've loaded the model published by Hideyuki Tachibana and others, *Efficiently Trainable Text-to-Speech System Based on Deep Convolutional Networks with Guided Attention* (2017; <https://arxiv.org/abs/1710.08969>). We used the implementation at <https://github.com/tugstugi/pytorch-dc-tts>.

Published in 2017, the novelty of this method is to work without recurrency in the network, instead relying on convolutions, a decision that results in much faster training and inference compared to other models. In fact, they claim that training their deep convolutional TTS networks only took about 15 hours on a gaming PC equipped with two off-the-shelf GPUs. Crowdsourced mean opinion scores didn't seem to increase after 15 hours of training on a dataset of readings from the **librivox** public domain audiobook project. The authors furnish a demo page to showcase audio samples at different stages of the training, where you can hear sentences spoken, such as *the two-player zero-sum game of wasserstein gan is derived by considering kantorovich-rubinstein duality*: https://tachi-hi.github.io/tts_samples/.

The architecture consists of two sub-networks, which can be trained separately, one to synthesize spectrograms from text, and another to create waveforms from spectrograms. The text-to-spectrogram part consists of these modules:

- Text encoder
- Audio encoder
- Attention
- Audio decoder

The interesting part of this is the guided attention mentioned in the title of the paper, which is responsible for the alignment of characters with time. They constrain this attention matrix to be nearly linear with time, as opposed to reading characters in random order given a **guided attention loss**:

$$\mathcal{L}_{\text{att}}(A) = \mathbb{E}_{nt} [A_{nt} W_{nt}], \text{ where}$$

$$W_{nt} = 1 - \exp - \frac{(n/N - t/T)^2}{2g^2},$$

with parameter $g = 0.2$, N the number of characters,

T the number of time frames, and $A \in \mathbb{R}^{N \times T}$.

This favors values on the diagonal of the matrix rather than off it. They argue that this constraint helps to speed up the training time considerably.

WaveGAN

In the *There's more...* section, we'll be loading up a different model, WaveGAN, published as *WaveGAN: Learn to synthesize raw audio with generative adversarial networks*, by Chris Donahue and others (2018; <https://arxiv.org/abs/1802.04208>).

Donahue and others train a GAN in an unsupervised setting for the synthesis of raw audio waveforms. They try two different strategies:

- A **spectrogram-strategy (SpecGAN)**, where they use a DCGAN (please refer to the *Generating images* recipe in Chapter 7, *Advanced Image Applications*), and apply it to spectrograms (frequency over time)
- A **waveform-strategy (WaveGAN)**, where they flatten the architecture (1D convolutions)

For the first strategy, they had to develop a spectrogram that they could convert back to text.

For the WaveGAN, they flattened the 2D convolutions into 1D while keeping the size (for example, a kernel of 5x5 became a 1D kernel of 25). Strides of 2x2 became 4. They removed the batch normalization layers. They trained using a Wasserstein GAN-GP strategy (Ishaan Gulrajani and others, 2017; *Improved training of Wasserstein GANs*; <https://arxiv.org/abs/1704.00028>).

Their WaveGAN performed notably worse in human judgments (mean opinion scores) than their SpecGAN. You can find a few examples of generated sounds at https://chrismdonahue.com/wavegan_examples/.

There's more...

We can also use the WaveGAN model to synthesize speech from text.

We'll download the checkpoints of a model trained on the speech commands we encountered in the previous recipe, *Recognizing voice commands*. Then we'll run the model to generate speech:

1. **Downloading the TensorFlow model checkpoints:** We'll download the model data as follows:

```
import wget

wget.download(
    'https://s3.amazonaws.com/wavegan-v1/models/timit.ckpt.index',
    'model.ckpt.index'
)
wget.download(
    'https://s3.amazonaws.com/wavegan-v1/models/timit.ckpt.data-00000-of-00001',
    'model.ckpt.data-00000-of-00001'
)
wget.download(
    'https://s3.amazonaws.com/wavegan-v1/models/timit_infer.meta',
    'infer.meta'
);
```

Now we can load the computation graph into memory:

```
import tensorflow as tf

tf.reset_default_graph()
saver = tf.train.import_meta_graph('infer.meta')
graph = tf.get_default_graph()
sess = tf.InteractiveSession()
saver.restore(sess, 'model.ckpt');
```

We can now generate speech.

2. **Generating speech:** The model architecture involves a latent representation of the letters. We can listen to what the model constructs based on random initializations of the latent representation:

```
import numpy as np
import PIL.Image
from IPython.display import display, Audio
import time as time

_z = (np.random.rand(2, 100) * 2.) - 1.
z = graph.get_tensor_by_name('z:0')
G_z = graph.get_tensor_by_name('G_z:0')[ :, :, 0]
G_z_spec = graph.get_tensor_by_name('G_z_spec:0')

start = time.time()
_G_z, _G_z_spec = sess.run([G_z, G_z_spec], {z: _z})
print('Finished! (Took {} seconds)'.format(time.time() - start))

for i in range(2):
    display(Audio(_G_z[i], rate=16000))
```

This should show us two examples of generated sounds, each with a Jupyter widget:



If these don't sound particularly natural, don't be afraid. After all, we've used a random initialization of the latent space.

See also

Efficiently Trainable Text-to-Speech System Based on Deep Convolutional Networks with Guided Attention (<https://arxiv.org/abs/1710.08969>). on Erdene-Ochir Tuguldur's GitHub repository, you can find a PyTorch implementation of that paper. The Mongolian text-to-speech was trained on 5 hours of audio from the Mongolian Bible: <https://github.com/tugstugi/pytorch-dc-tts>.

On Chris Donahue's GitHub repository of WaveGAN, you can see the WaveGAN implementation and examples for training from audio files in formats such as MP3, WAV, OGG, and others without preprocessing (<https://github.com/chrisdonahue/wavegan>).

Mozilla open sourced their TensorFlow implementation of Baidu's Deep Speech architecture (2014), which you can find here: <https://github.com.mozilla/DeepSpeech>.

Generating melodies

Artificial intelligence (AI) in music is a fascinating topic. Wouldn't it be cool if your favorite group from the 70s was bringing out new songs, but maybe more modern? Sony did this with the Beatles, and you can hear a song on YouTube, complete with automatically generated lyrics, called *Daddy's car*: https://www.youtube.com/watch?v=LSHZ_b05W7o.

In this recipe, we'll be generating a melody. More specifically, we'll be continuing a song using functionality in the Magenta Python library.

Getting ready

We need to install the Magenta library, and a few system libraries as dependencies. Please note that you need admin privileges in order to install system dependencies. If you are not on Linux (or *nix), you'll have to find the ones corresponding to your system.

On macOS, this should be relatively straightforward. Otherwise, it might be easier to run this in a Colab environment:

```
!apt-get update -qq && apt-get install -qq libfluidsynth1 fluid-soundfont-gm build-essential libasound2-dev libjack-dev  
!pip install -qU pyfluidsynth pretty_midi  
!pip install -qU magenta
```

If you are on Colab, you need another tweak to allow Python to find your system libraries:

```
import ctypes.util  
orig_ctypes_util_find_library = ctypes.util.find_library  
def proxy_find_library(lib):  
    if lib == 'fluidsynth':  
        return 'libfluidsynth.so.1'  
    else:  
        return orig_ctypes_util_find_library(lib)  
ctypes.util.find_library = proxy_find_library
```

This is a clever workaround for Python's foreign library import system, taken from the original Magenta tutorial, at https://colab.research.google.com/notebooks/magenta/hello_magenta/hello_magenta.ipynb.

It's time to get creative!

How to do it...

We'll first put together the start of a melody, and then we will load the `MelodyRNN` model from Magenta and let it continue the melody:

1. Let's put a melody together. We'll take *Twinkle Twinkle Little Star*. The Magenta project works with a note sequence representation called `NoteSequence`, which comes with many utilities, including conversion to and from MIDI. We can add notes to a sequence like this:

```
from note_seq.protobuf import music_pb2

twinkle_twinkle = music_pb2.NoteSequence()
twinkle_twinkle.notes.add(pitch=60, start_time=0.0, end_time=0.5,
velocity=80)
twinkle_twinkle.notes.add(pitch=60, start_time=0.5, end_time=1.0,
velocity=80)
twinkle_twinkle.notes.add(pitch=67, start_time=1.0, end_time=1.5,
velocity=80)
twinkle_twinkle.notes.add(pitch=67, start_time=1.5, end_time=2.0,
velocity=80)
twinkle_twinkle.notes.add(pitch=69, start_time=2.0, end_time=2.5,
velocity=80)
twinkle_twinkle.notes.add(pitch=69, start_time=2.5, end_time=3.0,
velocity=80)
twinkle_twinkle.notes.add(pitch=67, start_time=3.0, end_time=4.0,
velocity=80)
twinkle_twinkle.notes.add(pitch=65, start_time=4.0, end_time=4.5,
velocity=80)
twinkle_twinkle.notes.add(pitch=65, start_time=4.5, end_time=5.0,
velocity=80)
twinkle_twinkle.notes.add(pitch=64, start_time=5.0, end_time=5.5,
velocity=80)
twinkle_twinkle.notes.add(pitch=64, start_time=5.5, end_time=6.0,
velocity=80)
twinkle_twinkle.notes.add(pitch=62, start_time=6.0, end_time=6.5,
velocity=80)
twinkle_twinkle.notes.add(pitch=62, start_time=6.5, end_time=7.0,
velocity=80)
```

```

twinkle_twinkle.notes.add(pitch=60, start_time=7.0, end_time=8.0,
velocity=80)
twinkle_twinkle.total_time = 8
twinkle_twinkle.tempos.add(qpm=60);

```

We can visualize the sequence using Bokeh, and then we can play the note sequence:

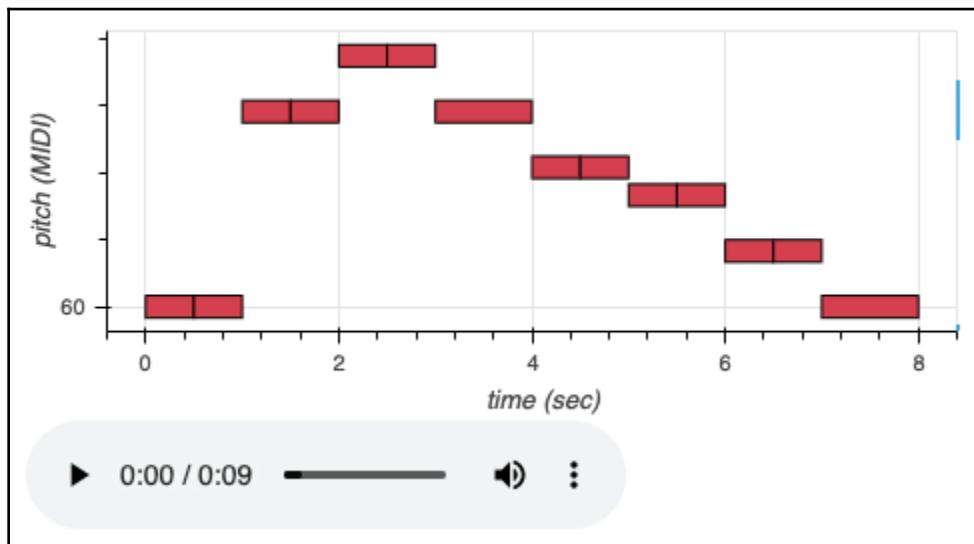
```

import note_seq

note_seq.plot_sequence(twinkle_twinkle)
note_seq.play_sequence(twinkle_twinkle, synth=note_seq.fluidsynth)

```

This looks as follows:



We can listen to the first 9 seconds of the song.

2. Let's load the MelodyRNN model from magenta:

```

from magenta.models.melody_rnn import melody_rnn_sequence_generator
from magenta.models.shared import sequence_generator_bundle
from note_seq.protobuf import generator_pb2
from note_seq.protobuf import music_pb2

note_seq.notebook_utils.download_bundle('attention_rnn.mag',
'/content/')
bundle =
sequence_generator_bundle.read_bundle_file('/content/basic_rnn.mag')

```

```
)  
generator_map = melody_rnn_sequence_generator.get_generator_map()  
melody_rnn = generator_map['basic_rnn'](checkpoint=None,  
bundle=bundle)  
melody_rnn.initialize()
```

This should only take a few seconds. The Magenta model is remarkably small compared to some other models we've encountered in this book.

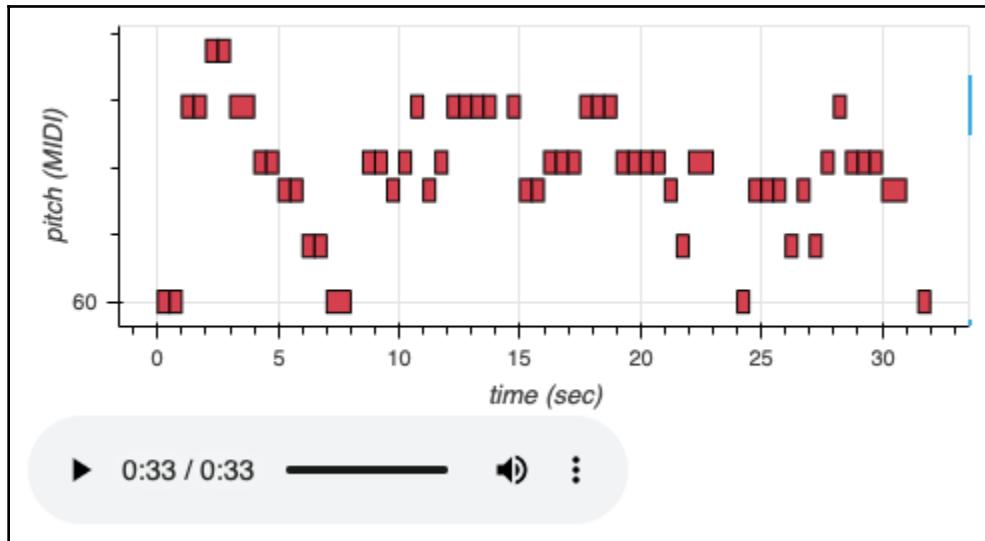
We can now feed in our previous melody, along with a few parameters in order to continue the song:

```
def get_options(input_sequence, num_steps=128, temperature=1.0):  
    last_end_time = (max(n.end_time for n in input_sequence.notes)  
                      if input_sequence.notes else 0)  
    qpm = input_sequence.tempos[0].qpm  
    seconds_per_step = 60.0 / qpm / melody_rnn.steps_per_quarter  
    total_seconds = num_steps * seconds_per_step  
  
    generator_options = generator_pb2.GeneratorOptions()  
    generator_options.args['temperature'].float_value = temperature  
    generate_section = generator_options.generate_sections.add()  
    start_time=last_end_time + seconds_per_step,  
    end_time=total_seconds)  
    return generator_options  
  
sequence = melody_rnn.generate(input_sequence,  
get_options(twinkle_twinkle))
```

We can now plot and play the new music:

```
note_seq.plot_sequence(sequence)  
note_seq.play_sequence(sequence, synth=note_seq.fluidsynth)
```

Once again, we get the Bokeh library plot and a play widget:



We can create a MIDI file from our note sequence like this:

```
note_seq.sequence_proto_to_midi_file(sequence, 'twinkle_continued.mid')
```

This creates a new MIDI file on disk.

On Google Colab, we can download the MIDI file like this:

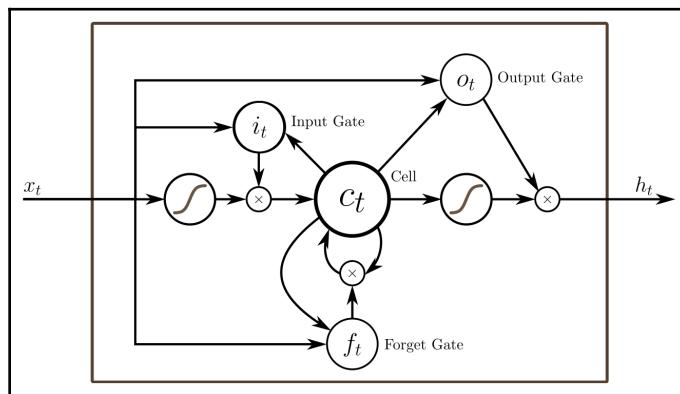
```
from google.colab import files  
files.download('twinkle_continued.mid')
```

We can feed different melodies via MIDI files into the model, or we can try with other parameters; we can increase or decrease the randomness (the `temperature` parameter), or let the sequence continue for longer periods (the `num_steps` parameter).

How it works...

MelodyRNN is an LSTM-based language model for musical notes. In order to understand MelodyRNN, we first need to understand how **Long Short-Term Memory (LSTM)** works. Published in 1997 by Sepp Hochreiter and Jürgen Schmidhuber (*Long short-term memory*: <https://doi.org/10.1162%2Fneco.1997.9.8.1735>), and updated numerous times since, LSTM is the most well-known example of a **Recurrent Neural Network (RNN)** and represents a state-of-the-art model for image recognition and machine learning tasks with sequences such as speech recognition, natural language processing, and time series. LSTMs were, or have been, behind popular tools by Google, Amazon, Microsoft, and Facebook for voice recognition and language translation.

The basic unit of an LSTM layer is an LSTM cell, which consists of several regulators, which we can see in the following schematic:



This diagram is based on Alex Graves and others, *Speech recognition with deep recurrent neural networks*, (2013), taken from the English language Wikipedia article on LSTMs at https://en.wikipedia.org/wiki/Long_short-term_memory.

The regulators include the following:

- An input gate
- An output gate
- A forget gate

We can explain the intuition behind these gates without getting lost in the equations. An input gate regulates how strongly the input influences the cell, an output gate dampens the outgoing cell activation, and the forget gate is a decay on the cell activity.

LSTMs have the advantage of being able to handle sequences of different lengths. However, their performance deteriorates with longer sequences. In order to learn even longer sequences, the Magenta library provides a model that includes an attention mechanism (Dzmitry Bahdanau and others, 2014, *Neural Machine Translation by Jointly Learning to Align and Translate*; <https://arxiv.org/abs/1409.0473>). Bahdanau and others showed that their attention mechanism leads to a much improved performance on longer sequences.

In MelodyRNN, an attention mask a is applied as follows:

$$u_i^t = v^T \tanh(W_1 h_i + W_2 c_t)$$

$$a_i^t = \text{softmax}(u_i^t)$$

$$\hat{h}_t = \sum_{i=t-n}^{t-1} a_i^t h_i$$

over hidden states h and current RNN state c_t
with model parameters v , W_1 , and W_2 .

You can find more details in the Magenta documentation at <https://magenta.tensorflow.org/2016/07/15/lookback-rnn-attention-rnn/>.

See also

Please note that Magenta has different variations of the MelodyRNN model available (https://github.com/magenta/magenta/tree/master/magenta/models/melody_rnn). Apart from MelodyRNN, Magenta provides further models, including a variational autoencoder for music generation, and many browser-based tools for exploring and generating music: <https://github.com/magenta/magenta>.

DeepBeat is a project for hip-hop beat generation: <https://github.com/nicholaschiang/deepbeat>.

Jukebox is an open sourced project based on the paper *Jukebox: A Generative Model for Music*, by Dhariwal and others (2020; <https://arxiv.org/abs/2005.00341>). You can find many audio samples at <https://openai.com/blog/jukebox/>.

You can find the original implementation of Parag K. Mital's NIPS paper, *Time Domain Neural Audio Style Transfer* (2017; <https://arxiv.org/abs/1711.11160>), at <https://github.com/pkmital/time-domain-neural-audio-style-transfer>.

10

Natural Language Processing

Natural language processing (NLP) is about analyzing texts and designing algorithms to process texts, making predictions from texts, or generating more text. NLP covers anything related to language, often including speech similar to what we saw in the *Recognizing voice commands* recipe in Chapter 9, *Deep Learning in Audio and Speech*. You might also want to refer to the *Battling algorithmic bias* recipe in Chapter 2, *Advanced Topics in Supervised Machine Learning*, or the *Representing for similarity search* recipe in Chapter 3, *Patterns, Outliers, and Recommendations*, for more traditional approaches. Most of this chapter will deal with the deep learning models behind the breakthroughs in recent years.

Language is often seen to be intrinsically linked to human intelligence, and machines mastering communication capabilities have long been seen as closely intertwined with the goal of achieving **Artificial General Intelligence (AGI)**. Alan Turing, in his 1950 article *Computing Machinery and Intelligence*, suggested a test, since then called the **Turing test**, in which interrogators have to find out whether their interlocutor (in a different room) is a computer or a human. It has been argued, however, that successfully tricking interrogators into thinking they are dealing with humans is not a proof of true understanding (or intelligence), but rather of manipulating symbols (the **Chinese room argument**; John Searle, *Minds, Brains, and Programs*, 1980). Whichever is the case, in recent years, with the availability of parallel computing devices such as GPUs, NLP has been making impressive progress in many benchmark tests, for example, in text classification: http://nlpprogress.com/english/text_classification.html.

We'll first do a simple supervised task, where we determine the sentiment of paragraphs, then we'll set up an Alexa-style chatbot that responds to commands. Next, we'll translate a text using sequence-to-sequence models. Finally, we'll attempt to write a popular novel using state-of-the-art text generation models.

In this chapter, we'll be doing these recipes:

- Classifying newsgroups
- Chatting to users
- Translating a text from English to German
- Writing a popular novel

Technical requirements

As in most chapters so far, we'll try both PyTorch and TensorFlow-based models. We'll apply different, more specialized libraries in each recipe.

As always, you can find the recipe notebooks on GitHub: <https://github.com/PacktPublishing/Artificial-Intelligence-with-Python-Cookbook/tree/master/chapter10>.

Classifying newsgroups

In this recipe, we'll do a relatively simple supervised task: based on texts, we'll train a model to determine what an article is about, from a selection of topics. This is a relatively common task with NLP; we'll try to give an overview of different ways to approach this.

You might also want to compare the *Battling algorithmic bias* recipe in Chapter 2, *Advanced Topics in Supervised Machine Learning*, on how to approach this problem using a bag-of-words approach (`CountVectorizer` in scikit-learn). In this recipe, we'll be using approaches with word embeddings and deep learning models using word embeddings.

Getting ready

In this recipe, we'll be using scikit-learn and TensorFlow (Keras), as in so many other recipes of this book. Additionally, we'll use word embeddings that we'll have to download, and we'll use utility functions from the Gensim library to apply them in our machine learning pipeline:

```
!pip install gensim
```

We'll be using a dataset from scikit-learn, but we still need to download the word embeddings. We'll use Facebook's fastText word embeddings trained on Wikipedia:

```
!pip install wget
import wget
wget.download(
    'https://dl.fbaipublicfiles.com/fasttext/vectors-wiki/wiki.en.vec',
    'wiki.en.vec'
)
```



Please note that the download can take a while and should take around 6 GB of disk space. If you are running on Colab, you might want to put the embedding file into a directory of your Google Drive, so you don't have to download it again when you restart your notebook.

How to do it...

The newsgroups dataset is a collection of around 20,000 newsgroup documents divided into 20 different groups. The 20 newsgroups collection is a popular dataset for testing machine learning techniques in NLP, such as text classification and text clustering.

We'll be classifying a selection of newsgroups into three different topics, and we'll be approaching this task with three different techniques that we can compare. We'll first get the dataset, and then apply a bag-of-words technique, using word embeddings, training custom word embeddings in a deep learning model.

First, we'll download the dataset using scikit-learn functionality. We'll download the newgroup dataset in two batches, for training and testing, respectively:

```
from sklearn.datasets import fetch_20newsgroups

categories = ['alt.atheism', 'soc.religion.christian', 'comp.graphics',
    'sci.med']
twenty_train = fetch_20newsgroups(
    subset='train',
    categories=categories,
    shuffle=True,
    random_state=42
)
twenty_test = fetch_20newsgroups(
    subset='test',
    categories=categories,
    shuffle=True,
    random_state=42
)
```

This conveniently gives us training and test datasets, which we can use in the three approaches.

Let's begin with covering the first one, using a bag-of-words approach.

Bag-of-words

We'll build a pipeline of counting words and reweighing them according to their frequency. The final classifier is a random forest. We train this on our training dataset:

```
import numpy as np
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.ensemble import RandomForestClassifier

text_clf = Pipeline([
    ('vect', CountVectorizer()),
    ('tfidf', TfidfTransformer()),
    ('clf', RandomForestClassifier()),
])
text_clf.fit(twenty_train.data, twenty_train.target)
```

CountVectorizer counts tokens in texts and TfidfTransformer reweights the counts. We'll discuss the **term frequency-inverse document frequency (TFIDF)** reweighting in the *How it works...* section.

After the training, we can test the accuracy on the test dataset:

```
predicted = text_clf.predict(twenty_test.data)
np.mean(predicted == twenty_test.target)
```

We get an accuracy of about 0.805. Let's see how our other two methods will do. Using word embeddings is next.

Word embeddings

We'll load up our previously downloaded word embeddings:

```
from gensim.models import KeyedVectors

model = KeyedVectors.load_word2vec_format(
    'wiki.en.vec',
    binary=False, encoding='utf8'
)
```

The most straightforward strategy to vectorize a text of several words is to average word embeddings across words. This works usually at least reasonably well for short texts:

```
import numpy as np
from tensorflow.keras.preprocessing.text import text_to_word_sequence

def embed_text(text: str):
    vector_list = [
        model.wv[w].reshape(-1, 1) for w in text_to_word_sequence(text)
        if w in model.wv
    ]
    if len(vector_list) > 0:
        return np.mean(
            np.concatenate(vector_list, axis=1),
            axis=1
        ).reshape(1, 300)
    else:
        return np.zeros(shape=(1, 300))

assert embed_text('training run').shape == (1, 300)
```

We'll apply this vectorization to our dataset and then train a random forest classifier on top of these vectors:

```
train_transformed = np.concatenate(
    [embed_text(t) for t in twenty_train.data]
)
rf = RandomForestClassifier().fit(train_transformed, twenty_train.target)
```

We can then test the performance of our approach:

```
test_transformed = np.concatenate(
    [embed_text(t) for t in twenty_test.data]
)
predicted = rf.predict(test_transformed)
np.mean(predicted == twenty_test.target)
```

We get an accuracy of about 0.862.

Let's see whether our last method does any better than this. We'll build customized word embeddings using Keras' embedding layer.

Custom word embeddings

An embedding layer is a way to create customized word embeddings on the fly in neural networks:

```
from tensorflow.keras import layers

embedding = layers.Embedding(
    input_dim=5000,
    output_dim=50,
    input_length=500
)
```

We have to tell the embedding layer how many words you want to store, how many dimensions your word embeddings should have, and how many words are in each text. We feed in arrays of integers that each refer to words in a dictionary. We can delegate the job of creating the input for the embedding layer to TensorFlow utility functions:

```
from tensorflow.keras.preprocessing.text import Tokenizer

tokenizer = Tokenizer(num_words=5000)
tokenizer.fit_on_texts(twenty_train.data)
```

This creates the dictionary. Now we need to tokenize the text and pad sequences to the right length:

```
from tensorflow.keras.preprocessing.sequence import pad_sequences

X_train = tokenizer.texts_to_sequences(twenty_train.data)
X_test = tokenizer.texts_to_sequences(twenty_test.data)
X_train = pad_sequences(X_train, padding='post', maxlen=500)
X_test = pad_sequences(X_test, padding='post', maxlen=500)
```

Now we are ready to build our neural network:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.losses import SparseCategoricalCrossentropy
from tensorflow.keras import regularizers

model = Sequential()
model.add(embedding)
model.add(layers.Flatten())
model.add(layers.Dense(
    10,
    activation='relu',
    kernel_regularizer=regularizers.l1_l2(l1=1e-5, l2=1e-4)
))
model.add(layers.Dense(len(categories), activation='softmax'))
```

```
model.compile(optimizer='adam',
              loss=SparseCategoricalCrossentropy(),
              metrics=['accuracy'])
model.summary()
```

Our model contains half a million parameters. Approximately half of them sit in the embedding, and the other half in the feedforward fully connected layer.

We fit our networks for a few epochs, and then we can test our accuracy on the test data:

```
model.fit(X_train, twenty_train.target, epochs=10)
predicted = model.predict(X_test).argmax(axis=1)
np.mean(predicted == twenty_test.target)
```

We get about 0.902 accuracy. We haven't tweaked the model architecture yet.

This concludes our newsgroup classification using bag-of-words, pre-trained word embeddings, and custom word embeddings. We'll now come to some background.

How it works...

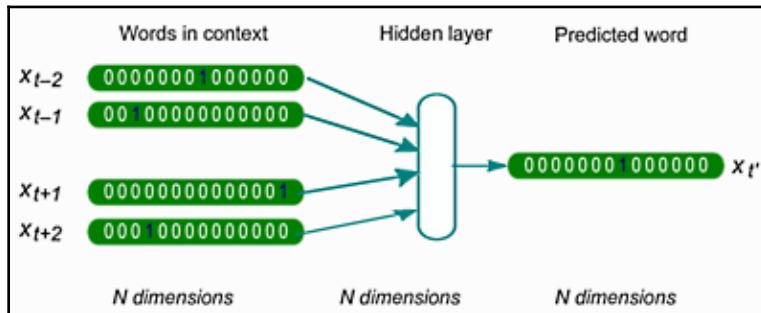
We've classified texts based on three different approaches of featurization: bag-of-words, pre-trained word embeddings, and custom word embeddings. Let's briefly delve into word embeddings and TFIDF.

We've already talked about the Skipgram and the **Continuous Bag of Words (CBOW)** algorithms in the *Making decisions based on knowledge* recipe in Chapter 5, *Heuristic Search Techniques and Logical Inference* (within the *Graph embedding with Walklets* subsection).

Very briefly, word vectors are a simple machine learning model that can predict the next word based on the context (the CBOW algorithm) or can predict the context based on a single word (the Skipgram algorithm). Let's quickly look at the CBOW neural network.

The CBOW algorithm

The CBOW algorithm is a two-layer feedforward neural network that predicts words (rather, the sparse index vector) from their context:



This illustration shows how, in the CBOW model, words are predicted based on the surrounding context. Here, words are represented as bag-of-words vectors. The hidden layer is composed of a weighted average of the context (linear projection). The output word is a prediction based on the hidden layer. This is adapted from an image on the French-language Wikipedia page on word embeddings: https://fr.wikipedia.org/wiki/Word_embedding.

What we haven't talked about is the implication of these word embeddings, which created such a stir when they came out. The embeddings are the network activations for a single word, and they have a compositional property that gave a title to many talks and a few papers. We can combine vectors to do semantic algebra or make analogies. The best-known example of this is the following:

$$\text{king} - \text{man} + \text{woman} = \text{queen}$$

Intuitively, a king and a queen are similar societal positions, only one is taken up by a man, the other by a woman. This is reflected in the embedding space learned on billions of words. Starting with the vector of king, subtracting the vector of man, and finally adding the vector of woman, the closest word that we end up at is queen.

The embedding space can tell us a lot about how we use language, some of it a bit concerning, such as when the word vectors exhibit gender stereotypes:

$$\text{man} - \text{woman} \sim \text{computer programmer} - \text{homemaker}$$

This can actually be corrected to some degree using affine transformations as shown by Tolga Bolukbasi and others (*Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings*, 2016; <https://arxiv.org/abs/1607.06520>).

Let's have a quick look at the reweighting employed in the bag-of-words approach of this recipe.

TFIDF

In the *Bag-of-words* section, we counted words using `CountVectorizer`. This gives us a vector of $1 \times N$, where N is the number of words in the vocabulary. The vocabulary has to be created during the `fit()` stage of `CountVectorizer`, before `transform()` will be able to create the (sparse) vector based on the positions of tokens (words) in the vocabulary.

By applying `CountVectorizer` for a number of documents, we get a sparse matrix of shape $|D| \times N$, where D is the corpus (the collection of documents), and $|D|$ the number of documents. Each position in this matrix accounts for the number of times a certain token occurs in a document. In this recipe, a token corresponds to a word, however, it can equally be a character or any collection of characters.

Some words might occur in every document; others might occur only in a small subset of documents, suggesting they are more specific and precise. That's the intuition of TFIDF, where the importance of counts (columns in the matrix) is raised if a word frequency across the corpus (the collection of documents) is low.

The inverse-term given a term for a set of documents D is defined as follows:

$$\text{tfidf}(t, D) = (1 + \log f_{t,d}) \cdot \log \frac{N}{n_t},$$

Here $f_{t,d}$ is the count of a term t in a document d , and n_t is the number of documents where t appears. You should see that the TFIDF value decreases with n_t . As a term occurs in more documents, the logarithm and the TFIDF value approach 0.

In the next recipes of this chapter, we'll go beyond the encodings of single words and study more complex language models.

There's more...

We'll briefly look at learning your own word embeddings using Gensim, building more complex deep learning models, and using pre-trained word embeddings in Keras:

1. We can easily train our own word embeddings on texts in Gensim.

Let's read in a text file in order to feed it as the training dataset for fastText:

```
from gensim.utils import tokenize
from gensim.test.utils import datapath

class FileIter(object):
```

```
def __init__(self, filepath: str):
    self.path = datapath(filepath)

def __iter__(self):
    with utils.open(self.path, 'r', encoding='utf-8') as fin:
        for line in fin:
            yield list(tokenize(line))
```

This can be useful for transfer learning, search applications, or for cases when learning the embeddings would take too long. Using Gensim, this is only a few lines of code (adapted from the Gensim documentation).

The training itself is straightforward and, since our text file is small, relatively quick:

```
from gensim.models import FastText

model = FastText(size=4, window=3, min_count=1)
model.build_vocab(
    sentences=FileIter(
        'crime-and-punishment.txt'
    ))
model.train(sentences=common_texts,
total_examples=len(common_texts), epochs=10)
```



You can find the *Crime and Punishment* novel at Project Gutenberg, where there are many more classic novels: <http://www.gutenberg.org/ebooks/2554>.

You can retrieve vectors from the trained model like this:

```
model.wv['axe']
```

Gensim comes with a lot of functionality, and we recommend you have a read through some of its documentation.

2. Building more complex deep learning models: for more difficult problems, we can use stacked `Conv1D` layers on top of the embedding, as follows:

```
x = Conv1D(128, 5, activation='relu')(embedded_sequences)
x = MaxPooling1D(5)(x)
```

Convolutional layers come with very few parameters, which is another advantage of using them.

3. Using pretrained word embeddings in a Keras model: If we want to use downloaded (or previously customized word embeddings), we can do this as well. We first need to create a dictionary, which we can easily do ourselves after loading them in, for example, with Gensim:

```
word_index = {i: w for i, w in enumerate(model.wv.vocab.keys())}
```

We can then feed these vectors into the embedding layer:

```
from tensorflow.keras.layers import Embedding

embedding_layer = Embedding(
    len(word_index) + 1,
    300,
    weights=[list(model.wv.vectors)],
    input_length=500,
    trainable=False
)
```

For training and testing, you have to feed in the word indices by looking them up in our new dictionary and pad them to the same length as we've done before.

This concludes our recipe on classifying newsgroups. We've applied three different types of featurization: bag-of-words, a pre-trained word embedding, and a custom word embedding in a simple neural network.

See also

We used word embeddings in this recipe. A lot of different embedding methods have been introduced, and quite a few word embedding matrices have been published that were trained on hundreds of billions of words from many millions of documents. Such large-scale training could cost as much as hundreds of thousands of dollars if done on rented hardware. The most popular word embeddings are the following:

- GloVe: <https://nlp.stanford.edu/projects/glove/>
- fastText: <https://fasttext.cc/docs/en/crawl-vectors.html>
- Word2vec: <https://code.google.com/archive/p/word2vec/>

Popular libraries for dealing with word embeddings are these:

- Gensim: <https://radimrehurek.com/gensim/>
- fastText: <https://fasttext.cc/>
- spaCy: <https://spacy.io/>

Kashgari is a library built on top of Keras for text labeling and text classification and includes Word2vec and more advanced models such as BERT and GPT2 language embeddings: <https://github.com/BrikerMan/Kashgari>.

The Hugging Face transformer library (<https://github.com/huggingface/transformers>) includes many advanced architectures and pre-trained weights for many transformer models that can be used for text embedding. These models can achieve state-of-the-art performance in many NLP tasks. For instance, companies such as Google have moved many of their language applications to the BERT architecture. We'll learn more about transformer architectures in the *Translating a text from English to German* recipe in this chapter.

fast.ai provides a compendium of tutorials and courses on deep learning with PyTorch; it includes many resources about NLP as well: <https://nlp.fast.ai/>.

Finally, in NLP, often there can be thousands or even millions of different labels in classification tasks. This has been termed an **eXtreme MultiLabel (XML)** scenario. You can find a notebook tutorial on XML here: <https://github.com/pfontisso/Extreme-Multi-Label-Classification>.

Chatting to users

In 1966, Joseph Weizenbaum published an article about his chatbot ELIZA, called *ELIZA—a computer program for the study of natural language communication between man and machine*. Created with a sense of humor to show the limitations of technology, the chatbot employed simplistic rules and vague, open-ended questions as a way of giving an impression of empathic understanding in the conversation, and was in an ironic twist often seen as a milestone of artificial intelligence. The field has moved on, and today, AI assistants are around us: you might have an Alexa, a Google Echo, or any of the other commercial home assistants in the market.

In this recipe, we'll be building an AI assistant. The difficulty with this is that there are an infinite amount of ways for people to express themselves and that it is simply impossible to anticipate everything your users might say. In this recipe, we'll be training a model to infer what they want and we'll respond accordingly in consequence.

Getting ready

For this recipe, we'll be using a framework developed by Fariz Rahman called **Eywa**. We'll install it with pip from GitHub:

```
!pip install git+https://www.github.com/farizrahman4u/eywa.git
```

Eywa has the main capabilities of what's expected from a conversational agent, and we can look at its code for some of the modeling that's behind its power.

We are also going to be using the OpenWeatherMap Web API through the `pyOWM` library, so we'll install this library as well:

```
!pip install pyOWM
```

With this library, we can request weather data in response to user requests as part of our chatbot functionality. If you want to use this in your own chatbot, you should register a free user account and get your API key on OpenWeatherMap.org for up to 1,000 requests a day.

Let's see how we implement this.

How to do it...

Our agent will process sentences by the user, interpret them, and respond accordingly. It will first predict the intent of user queries, and then extract entities in order to know more precisely what the query is about, before returning an answer:

1. Let's start with the intent classes – based on a few samples of phrases each, we'll define intents such as `greetings`, `taxi`, `weather`, `datetime`, and `music`:

```
from eywa.nlu import Classifier

CONV_SAMPLES = {
    'greetings' : ['Hi', 'hello', 'How are you', 'hey there',
    'hey'],
    'taxi' : ['book a cab', 'need a ride', 'find me a cab'],
    'weather' : ['what is the weather in tokyo', 'weather germany',
                 'what is the weather like in kochi',
                 'what is the weather like', 'is it hot
    outside'],
    'datetime' : ['what day is today', 'todays date', 'what time is
    it now',
                 'time now', 'what is the time'],
    'music' : ['play the Beatles', 'shuffle songs', 'make a sound']
}
```

```
CLF = Classifier()
for key in CONV_SAMPLES:
    CLF.fit(CONV_SAMPLES[key], key)
```

We've created a classifier based on conversation samples. We can quickly test how this works using the following code block:

```
print(CLF.predict('will it rain today')) # >>> 'weather'
print(CLF.predict('play playlist rock n\roll')) # >>> 'music'
print(CLF.predict('what\'s the hour?')) # >>> 'datetime'
```

We can successfully predict whether the required action is regarding the weather, a hotel booking, music, or about the time.

2. As the next step, we need to understand whether there's something more specific to the intent, such as the weather in London versus the weather in New York, or playing the Beatles versus Kanye West. We can use eywa entity extraction for this purpose:

```
from eywa.nlu import EntityExtractor

X_WEATHER = [
    'what is the weather in tokyo',
    'weather germany',
    'what is the weather like in kochi'
]
Y_WEATHER = [
    {'intent': 'weather', 'place': 'tokyo'},
    {'intent': 'weather', 'place': 'germany'},
    {'intent': 'weather', 'place': 'kochi'}
]

EX_WEATHER = EntityExtractor()
EX_WEATHER.fit(X_WEATHER, Y_WEATHER)
```

This is to check for a specific place for the weather prediction. We can test the entity extraction for the weather as well:

```
EX_WEATHER.predict('what is the weather in London')
```

We ask for the weather in London, and, in fact, our entity extraction successfully comes back with the place name:

```
{'intent': 'weather', 'place': 'London'}
```

3. We also need to code the functionality of our conversational agent, such as looking up the weather forecast. Let's first do a weather request:

```
from pyowm import OWM

mgr = OWM('YOUR-API-KEY').weather_manager()

def get_weather_forecast(place):
    observation = mgr.weather_at_place(place)
    return observation.get_weather().get_detailed_status()

print(get_weather_forecast('London'))
```

We can request weather forecasts given a location with the Python OpenWeatherMap library (pyOWM). Calling the new function, `get_weather_forecast()`, with London as its argument results in this as of in the time of writing:

```
overcast clouds
```



Please note that you need to use your own (free) OpenWeatherMap API key if you want to execute this.

No chatbot is complete without a greeting and the date:

```
X_GREETING = ['Hii', 'helllo', 'Howdy', 'hey there', 'hey', 'Hi']
Y_GREETING = [{ 'greet': 'Hii' }, { 'greet': 'helllo' }, { 'greet': 'Howdy' },
                { 'greet': 'hey' }, { 'greet': 'hey' }, { 'greet': 'Hi' }]
EX_GREETING = EntityExtractor()
EX_GREETING.fit(X_GREETING, Y_GREETING)

X_DATETIME = ['what day is today', 'date today', 'what time is it
now', 'time now']
Y_DATETIME = [{ 'intent' : 'day', 'target': 'today' }, { 'intent' :
'date', 'target': 'today' },
                { 'intent' : 'time', 'target': 'now' }, { 'intent' :
'time', 'target': 'now' }]

EX_DATETIME = EntityExtractor()
EX_DATETIME.fit(X_DATETIME, Y_DATETIME)
```

4. Let's create some interaction based on the classifier and entity extraction. We'll write a response function that can greet, tell the date, and give a weather forecast:

```
_EXTRACTORS = {  
    'taxi': None,  
    'weather': EX_WEATHER,  
    'greetings': EX_GREETING,  
    'datetime': EX_DATETIME,  
    'music': None  
}
```

We are leaving out functionality for calling taxis or playing music:

```
import datetime  
  
_EXTRACTORS = {  
    'taxi': None,  
    'weather': EX_WEATHER,  
    'greetings': EX_GREETING,  
    'datetime': EX_DATETIME,  
    'music': None  
}  
  
def question_and_answer(u_query: str):  
    q_class = CLF.predict(u_query)  
    print(q_class)  
    if _EXTRACTORS[q_class] is None:  
        return 'Sorry, you have to upgrade your software!'  
  
    q_entities = _EXTRACTORS[q_class].predict(u_query)  
    print(q_entities)  
    if q_class == 'greetings':  
        return q_entities.get('greet', 'hello')  
    if q_class == 'weather':  
        place = q_entities.get('place', 'London').replace('_', ' ')  
        return 'The forecast for {} is {}'.format(  
            place,  
            get_weather_forecast(place))  
    if q_class == 'datetime':  
        return 'Today\'s date is {}'.format(  
            datetime.datetime.today().strftime('%B %d, %Y'))  
    return 'I couldn\'t understand what you said. I am sorry.'
```

The `question_and_answer()` function answers a user query.

We can now have a limited conversation with our agent if we ask it questions:

```
while True:  
    query = input('\nHow can I help you?')  
    print(question_and_answer(query))
```

This wraps up our recipe. We've implemented a simple chatbot that first predicts intent and then extracts entities. Based on intent and entities, a user query is answered based on rules.

You should be able to ask for the date and the weather in different places, however, it will tell you to upgrade your software if you ask for taxis or music. You should be able to implement and extend this functionality by yourself if you are interested.

How it works...

We've implemented a very simple, though effective, chatbot for basic tasks. It should be clear how this can be extended and customized for more or other tasks.

Before we go through some of this, it might be of interest to look at the ELIZA chatbot mentioned in the introduction of this recipe. This will hopefully shed some light on what improvements we need to understand a broader set of languages.

How did ELIZA work?

ELIZA

The original ELIZA mentioned in the introduction has many statement-response pairs, such as the following:

```
[r'Is there (.*)', [  
    "Do you think there is %1?",  
    "It's likely that there is %1.",  
    "Would you like there to be %1?"  
],
```

Given a match of the regular expression, one of the possible responses is chosen randomly, while verbs are transformed if necessary, including contractions, using logic like this, for example:

```
gReflections = {  
    #...  
    "i'd" : "you would",  
}
```



These are excerpts from Jez Higgins' ELIZA knock-off on GitHub: <https://github.com/jezhiggins/eliza.py>.

Sadly perhaps experiences with call centers might seem similar. They often employ scripts as well, such as the following:

<Greeting>

"Thank you for calling, my name is _. How can I help you today?"

...

"Do you have any other questions or concerns that I can help you with today?"

While for machines, in the beginning, it is easier to hardcode some rules, if you want to handle more complexity, you'll be building models that interpret intentions and references such as locations.

Eywa

Eywa, a framework for conversational agents, comes with three main functionalities:

- **A classifier** – to decide what class the user input belongs to from a choice of a few
- **An entity extractor** – to extract named entities from sentences
- **Pattern matching** – for variable matching based on parts of speech and semantic meaning

All three are very simple to use, though quite powerful. We've seen the first two functionalities in action in the *How to do it...* section. Let's see the pattern matching for food types based on semantic context:

```
from eywa.nlu import Pattern

p = Pattern('I want to eat [food: pizza, banana, yogurt, kebab]')
p('i\'d like to eat sushi')
```

We create a variable food with sample values: pizza, banana, yogurt, and kebab. Using food terms in similar contexts will match our variables. The expression should return this:

```
{'food' : 'sushi'}
```

The usage looks very similar to regular expressions, however, while regular expressions are based on words and their morphology, eywa.nlu.Pattern works semantically, anchored in word embeddings.



A **regular expression** (short: regex) is a sequence of characters that define a search pattern. It was first formalized by Steven Kleene and implemented by Ken Thompson and others in Unix tools such as QED, ed, grep, and sed in the 1960s. This syntax has entered the POSIX standard and is therefore sometimes referred to as **POSIX regular expressions**. A different standard emerged in the late 1990s with the Perl programming language, termed **Perl Compatible Regular Expressions (PCRE)**, which has been adopted in different programming languages, including Python.

How do these models work?

First of all, the eywa library relies on sense2vec word embeddings from explosion.ai. Sense2vec word embeddings were introduced by Andrew Trask and others (*sense2vec – A Fast and Accurate Method for Word Sense Disambiguation In Neural Word Embeddings*, 2015). This idea was taken up by explosion.ai, who trained part-of-speech disambiguated word embeddings on Reddit discussions. You can read up on these on the explosion.ai website: <https://explosion.ai/blog/sense2vec-reloaded>.

The classifier goes through the stored conversational items and picks out the match with the highest similarity score based on these embeddings. Please note that eywa has another model implementation based on recurrent neural networks.

See also

Libraries and frameworks abound for creating chatbots with different ideas and integrations:

- ParlAI is a library for training and testing dialog models. It comes with more than 80 dialog datasets out of the box as well as, integration with Facebook Messenger and Mechanical Turk: <https://github.com/facebookresearch/ParlAI>.

- NVIDIA has its own toolkit for conversational AI applications and comes with many modules providing additional functionality such as automatic speech recognition and speech synthesis: <https://github.com/NVIDIA/NeMo>.
- Google Research open sourced their code for an open-domain dialog system: <https://github.com/google-research/google-research/tree/master/meena>.
- Rasa incorporates feedback on every interaction to improve the chatbot: <https://rasa.com/>.
- Chatterbot, a spaCy-based library: <https://spacy.io/universe/project/Chatterbot>.

Translating a text from English to German

In this recipe, we'll be implementing a transformer network from scratch, and we'll be training it for translation tasks from English to German. In the *How it works...* section, we'll go through a lot of the details.

Getting ready

We recommend using a machine with a **GPU**. The Colab environment is highly recommended, however, please make sure you are using a runtime with GPU enabled. If you want to check that you have access to a GPU, you can call the NVIDIA System Management Interface:

```
!nvidia-smi
```

You should see something like this:

```
Tesla T4: 0MiB / 15079MiB
```

This tells you you are using an NVIDIA Tesla T4 with 0 MB of about 1.5 GB used (1 MiB corresponds to approximately 1.049 MB).

We'll need a relatively new version of `torchtext`, a library with text datasets and utilities for `pytorch`:

```
!pip install torchtext==0.7.0
```

For the part in the *There's more...* section, you might need to install an additional dependency:

```
!pip install hydra-core
```

We are using spaCy for tokenization. This comes preinstalled in Colab. In other environments, you might have to pip-install it. We do need to install the German core functionality, such as tokenization for spacy, which we'll rely on in this recipe:

```
!python -m spacy download de
```

We'll load up this functionality in the main part of the recipe.

How to do it...

In this recipe, we'll be implementing a transformer model from scratch, and we'll be training it for a translation task. We've adapted this notebook from Ben Trevett's excellent tutorials on implementing a transformer sequence-to-sequence model with PyTorch and TorchText: <https://github.com/bentrevett/pytorch-seq2seq>.

We'll first prepare the dataset, then implement the transformer architecture, then we'll train, and finally test:

1. Preparing the dataset – let's import all the required modules upfront:

```
import torch
import torch.nn as nn
import torch.optim as optim

import torchtext
from torchtext.datasets import Multi30k
from torchtext.data import Field, BucketIterator

import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

import spacy
import numpy as np

import math
```

The dataset we'll be training on is the Multi30k dataset. This is a dataset of about 30,000 parallel English, German, and French short sentences.

We'll load the spacy functionality and we'll implement functions to tokenize German and English text, respectively:

```
spacy_de = spacy.load('de')
spacy_en = spacy.load('en')

def tokenize_de(text):
    return [tok.text for tok in spacy_de.tokenizer(text)]

def tokenize_en(text):
    return [tok.text for tok in spacy_en.tokenizer(text)]
```

These functions tokenize German and English text from a string into a list of strings.

Field defines operations for converting text to tensors. It provides interfaces to common text processing tools and holds a Vocab that maps tokens or words to a numerical representation. We are passing our preceding tokenization methods:

```
SRC = Field(
    tokenize=tokenize_en,
    init_token='<sos>',
    eos_token='<eos>',
    lower=True,
    batch_first=True
)

TRG = Field(
    tokenize=tokenize_de,
    init_token='<sos>',
    eos_token='<eos>',
    lower=True,
    batch_first=True
)
```

We'll create a train-test-validation split from the dataset. The exts parameter specifies which languages to use as the source and target, and fields specifies which fields to feed. After that, we build the vocabulary from the training dataset:

```
train_data, valid_data, test_data = Multi30k.splits(
    exts=('.en', '.de'),
    fields=(SRC, TRG)
)
SRC.build_vocab(train_data, min_freq=2)
TRG.build_vocab(train_data, min_freq=2)
```

Then we can define our data iterator over the train, validation, and test datasets:

```
device = torch.device('cuda' if torch.cuda.is_available() else
    'cpu')
BATCH_SIZE = 128

train_iterator, valid_iterator, test_iterator =
    BucketIterator.splits(
        (train_data, valid_data, test_data),
        batch_size=BATCH_SIZE,
        device=device
    )
```

We can build our transformer architecture now before we train it with this dataset.

2. When implementing the transformer architecture, important parts are the multi-head attention and the feedforward connections. Let's define them, first starting with the attention:

```
class MultiHeadAttentionLayer(nn.Module):
    def __init__(self, hid_dim, n_heads, dropout, device):
        super().__init__()
        assert hid_dim % n_heads == 0
        self.hid_dim = hid_dim
        self.n_heads = n_heads
        self.head_dim = hid_dim // n_heads
        self.fc_q = nn.Linear(hid_dim, hid_dim)
        self.fc_k = nn.Linear(hid_dim, hid_dim)
        self.fc_v = nn.Linear(hid_dim, hid_dim)
        self.fc_o = nn.Linear(hid_dim, hid_dim)
        self.dropout = nn.Dropout(dropout)
        self.scale =
            torch.sqrt(torch.FloatTensor([self.head_dim])).to(device)

    def forward(self, query, key, value, mask = None):
        batch_size = query.shape[0]
        Q = self.fc_q(query)
        K = self.fc_k(key)
        V = self.fc_v(value)
        Q = Q.view(batch_size, -1, self.n_heads,
        self.head_dim).permute(0, 2, 1, 3)
        K = K.view(batch_size, -1, self.n_heads,
        self.head_dim).permute(0, 2, 1, 3)
        V = V.view(batch_size, -1, self.n_heads,
        self.head_dim).permute(0, 2, 1, 3)
        energy = torch.matmul(Q, K.permute(0, 1, 3, 2)) /
        self.scale
```

```

        if mask is not None:
            energy = energy.masked_fill(mask == 0, -1e10)
        attention = torch.softmax(energy, dim = -1)
        x = torch.matmul(self.dropout(attention), v)
        x = x.permute(0, 2, 1, 3).contiguous()
        x = x.view(batch_size, -1, self.hid_dim)
        x = self.fc_o(x)
        return x, attention
    
```

The feedforward layer is just a single forward pass with a non-linear activation, and a dropout, and a linear read-out. The first projection is much larger than the original hidden dimension. In our case, we use a hidden dimension of 512 and a pf dimension of 2048:

```

class PositionwiseFeedforwardLayer(nn.Module):
    def __init__(self, hid_dim, pf_dim, dropout):
        super().__init__()
        self.fc_1 = nn.Linear(hid_dim, pf_dim)
        self.fc_2 = nn.Linear(pf_dim, hid_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        x = self.dropout(torch.relu(self.fc_1(x)))
        x = self.fc_2(x)
        return x
    
```

We'll need Encoder and Decoder parts, each with their own layers. Then we'll connect these two with the Seq2Seq model.

This is how the encoder looks:

```

class Encoder(nn.Module):
    def __init__(self, input_dim, hid_dim,
                 n_layers, n_heads, pf_dim,
                 dropout, device,
                 max_length=100):
        super().__init__()
        self.device = device
        self.tok_embedding = nn.Embedding(input_dim, hid_dim)
        self.pos_embedding = nn.Embedding(max_length, hid_dim)
        self.layers = nn.ModuleList([
            EncoderLayer(
                hid_dim,
                n_heads,
                pf_dim,
                dropout,
                device
            ) for _ in range(n_layers)])
    
```

```

        self.dropout = nn.Dropout(dropout)
        self.scale =
        torch.sqrt(torch.FloatTensor([hid_dim])).to(device)

    def forward(self, src, src_mask):
        batch_size = src.shape[0]
        src_len = src.shape[1]
        pos = torch.arange(
            0, src_len
        ).unsqueeze(0).repeat(
            batch_size, 1
        ).to(self.device)
        src = self.dropout(
            (self.tok_embedding(src) * self.scale) +
            self.pos_embedding(pos)
        )
        for layer in self.layers:
            src = layer(src, src_mask)
        return src

```

It consists of a number of encoder layers. These look as follows:

```

class EncoderLayer(nn.Module):
    def __init__(self, hid_dim, n_heads,
                 pf_dim, dropout, device):
        super().__init__()
        self.self_attn_layer_norm = nn.LayerNorm(hid_dim)
        self.ff_layer_norm = nn.LayerNorm(hid_dim)
        self.self_attention = MultiHeadAttentionLayer(hid_dim,
                                                      n_heads, dropout, device)
        self.positionwise_feedforward =
        PositionwiseFeedforwardLayer(
            hid_dim, pf_dim, dropout
        )
        self.dropout = nn.Dropout(dropout)

    def forward(self, src, src_mask):
        _src, _ = self.self_attention(src, src, src, src_mask)
        src = self.self_attn_layer_norm(src + self.dropout(_src))
        _src = self.positionwise_feedforward(src)
        src = self.ff_layer_norm(src + self.dropout(_src))
        return src

```

The decoder is not too different from the encoder, however, it comes with two multi-head attention layers. The decoder looks like the following:

```

class Decoder(nn.Module):
    def __init__(self, output_dim, hid_dim,

```

```
        n_layers, n_heads, pf_dim,
        dropout, device, max_length=100):
    super().__init__()
    self.device = device
    self.tok_embedding = nn.Embedding(output_dim, hid_dim)
    self.pos_embedding = nn.Embedding(max_length, hid_dim)
    self.layers = nn.ModuleList(
        [DecoderLayer(
            hid_dim, n_heads,
            pf_dim, dropout,
            device
        ) for _ in range(n_layers)])
    self.fc_out = nn.Linear(hid_dim, output_dim)
    self.dropout = nn.Dropout(dropout)
    self.scale =
        torch.sqrt(torch.FloatTensor([hid_dim])).to(device)

    def forward(self, trg, enc_src, trg_mask, src_mask):
        batch_size = trg.shape[0]
        trg_len = trg.shape[1]
        pos = torch.arange(0, trg_len).unsqueeze(0).repeat(
            batch_size, 1
        ).to(self.device)
        trg = self.dropout(
            (self.tok_embedding(trg) * self.scale) +
            self.pos_embedding(pos)
        )
        for layer in self.layers:
            trg, attention = layer(trg, enc_src, trg_mask,
src_mask)
        output = self.fc_out(trg)
        return output, attention
```

In sequence, the decoder layer does the following tasks:

- Self-attention with masking
- Feedforward
- Dropout
- Residual connection
- Normalization



The mask in the self-attention layer is to avoid the model including the next token in its prediction (which would be cheating).

Let's implement the decoder layer:

```
class DecoderLayer(nn.Module):
    def __init__(self, hid_dim, n_heads,
                 pf_dim, dropout, device):
        super().__init__()
        self.self_attn_layer_norm = nn.LayerNorm(hid_dim)
        self.enc_attn_layer_norm = nn.LayerNorm(hid_dim)
        self.ff_layer_norm = nn.LayerNorm(hid_dim)
        self.self_attention = MultiHeadAttentionLayer(hid_dim,
n_heads, dropout, device)
        self.encoder_attention = MultiHeadAttentionLayer(hid_dim,
n_heads, dropout, device)
        self.positionwise_feedforward =
PositionwiseFeedforwardLayer(
            hid_dim, pf_dim, dropout
        )
        self.dropout = nn.Dropout(dropout)

    def forward(self, trg, enc_src, trg_mask, src_mask):
        _trg, _ = self.self_attention(trg, trg, trg, trg_mask)
        trg = self.self_attn_layer_norm(trg + self.dropout(_trg))
        _trg, attention = self.encoder_attention(trg, enc_src,
enc_src, src_mask)
        trg = self.enc_attn_layer_norm(trg + self.dropout(_trg))
        _trg = self.positionwise_feedforward(trg)
        trg = self.ff_layer_norm(trg + self.dropout(_trg))
        return trg, attention
```

Finally, it all comes together in the Seq2Seq model:

```
class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder,
                 src_pad_idx, trg_pad_idx, device):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_pad_idx = src_pad_idx
        self.trg_pad_idx = trg_pad_idx
        self.device = device

    def make_src_mask(self, src):
        src_mask = (src !=
self.src_pad_idx).unsqueeze(1).unsqueeze(2)
        return src_mask

    def make_trg_mask(self, trg):
        trg_pad_mask = (trg !=
```

```
        self.trg_pad_idx).unsqueeze(1).unsqueeze(2)
        trg_len = trg.shape[1]
        trg_sub_mask = torch.tril(torch.ones((trg_len, trg_len),
device=self.device)).bool()
        trg_mask = trg_pad_mask & trg_sub_mask
        return trg_mask

    def forward(self, src, trg):
        src_mask = self.make_src_mask(src)
        trg_mask = self.make_trg_mask(trg)
        enc_src = self.encoder(src, src_mask)
        output, attention = self.decoder(trg, enc_src, trg_mask,
src_mask)
        return output, attention
```

We can now instantiate our model with actual parameters:

```
INPUT_DIM = len(SRC.vocab)
OUTPUT_DIM = len(TRG.vocab)
HID_DIM = 256
ENC_LAYERS = 3
DEC_LAYERS = 3
ENC_HEADS = 8
DEC_HEADS = 8
ENC_PF_DIM = 512
DEC_PF_DIM = 512
ENC_DROPOUT = 0.1
DEC_DROPOUT = 0.1

enc = Encoder(INPUT_DIM,
              HID_DIM, ENC_LAYERS,
              ENC_HEADS, ENC_PF_DIM,
              ENC_DROPOUT, device
              )

dec = Decoder(OUTPUT_DIM,
              HID_DIM, DEC_LAYERS,
              DEC_HEADS, DEC_PF_DIM,
              DEC_DROPOUT, device
              )
SRC_PAD_IDX = SRC.vocab.stoi[SRC.pad_token]
TRG_PAD_IDX = TRG.vocab.stoi[TRG.pad_token]

model = Seq2Seq(enc, dec, SRC_PAD_IDX, TRG_PAD_IDX,
device).to(device)
```

This whole model comes with 9,543,087 trainable parameters.

3. Training the translation model, we can initialize the parameters using Xavier uniform normalization:

```
def initialize_weights(m):
    if hasattr(m, 'weight') and m.weight.dim() > 1:
        nn.init.xavier_uniform_(m.weight.data)

model.apply(initialize_weights);
```

We need to set the learning rate much lower than the default:

```
LEARNING_RATE = 0.0005

optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE)
```

In our loss function, `CrossEntropyLoss`, we have to make sure to ignore padded tokens:

```
criterion = nn.CrossEntropyLoss(ignore_index=TRG_PAD_IDX)
```

Our training function looks like this:

```
def train(model, iterator, optimizer, criterion, clip):
    model.train()
    epoch_loss = 0
    for i, batch in enumerate(iterator):
        src = batch.src
        trg = batch.trg
        optimizer.zero_grad()
        output, _ = model(src, trg[:, :-1])
        output_dim = output.shape[-1]
        output = output.contiguous().view(-1, output_dim)
        trg = trg[:, 1:].contiguous().view(-1)
        loss = criterion(output, trg)
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), clip)
        optimizer.step()
        epoch_loss += loss.item()
    return epoch_loss / len(iterator)
```

The training is then performed in a loop:

```
N_EPOCHS = 10
CLIP = 1
best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):
    train_loss = train(model, train_iterator, optimizer, criterion,
```

CLIP)

```
print(f'\tTrain Loss: {train_loss:.3f} | Train PPL:\n{math.exp(train_loss):7.3f}')
```

We are slightly simplifying things here. You can find the full notebook on GitHub.

This trains for 10 epochs.

4. Testing the model, we'll first have to write functions to encode a sentence for the model and decode the model output back to get a sentence. Then we can run some sentences and have a look at the translations. Finally, we can calculate a metric of the translation performance across the test set.

In order to translate a sentence, we have to encode it numerically using the source vocabulary created before and append stop tokens before feeding this into our model. The model output then has to be decoded from the target vocabulary:

```
def translate_sentence(sentence, src_field, trg_field, model,
device, max_len=50):
    model.eval()
    if isinstance(sentence, str):
        nlp = spacy.load('en')
        tokens = [token.text.lower() for token in nlp(sentence)]
    else:
        tokens = [token.lower() for token in sentence]
    tokens = [src_field.init_token] + tokens +
    [src_field.eos_token]
    src_indexes = [src_field.vocab.stoi[token] for token in tokens]
    src_tensor =
    torch.LongTensor(src_indexes).unsqueeze(0).to(device)
    src_mask = model.make_src_mask(src_tensor)
    with torch.no_grad():
        enc_src = model.encoder(src_tensor, src_mask)
    trg_indexes = [trg_field.vocab.stoi[trg_field.init_token]]
    for i in range(max_len):
        trg_tensor =
        torch.LongTensor(trg_indexes).unsqueeze(0).to(device)
        trg_mask = model.make_trg_mask(trg_tensor)
        with torch.no_grad():
            output, attention = model.decoder(trg_tensor, enc_src,
        trg_mask, src_mask)
            pred_token = output.argmax(2)[:, -1].item()
            trg_indexes.append(pred_token)
            if pred_token == trg_field.vocab.stoi[trg_field.eos_token]:
                break
```

```
trg_tokens = [trg_field.vocab.itos[i] for i in trg_indexes]
return trg_tokens[1:], attention
```

We can look at an example pair and check the translation:

```
example_idx = 8

src = vars(train_data.examples[example_idx])['src']
trg = vars(train_data.examples[example_idx])['trg']

print(f'src = {src}')
print(f'trg = {trg}')
```

We get the following pair:

```
src = ['a', 'woman', 'with', 'a', 'large', 'purse', 'is',
'walking', 'by', 'a', 'gate', '.']
trg = ['eine', 'frau', 'mit', 'einer', 'großen', 'geldbörse',
'geht', 'an', 'einem', 'tor', 'vorbei', '.']
```

We can compare this with the translation we get from our model:

```
translation, attention = translate_sentence(src, SRC, TRG, model,
device)
print(f'predicted trg = {translation}')
```

This is our translated sentence:

```
predicted trg = ['eine', 'frau', 'mit', 'einer', 'großen',
'handtasche', 'geht', 'an', 'einem', 'tor', 'vorbei', '.', '<eos>']
```

Our translation looks actually better than the original translation. A purse is not really a wallet (geldbörse), but a small bag (handtasche).

We can then calculate a metric, the BLEU score, of our model versus the gold standard:

```
from torchtext.data.metrics import bleu_score

def calculate_bleu(data, src_field, trg_field, model, device,
max_len=50):
    trgs = []
    pred_trgs = []
    for datum in data:
        src = vars(datum)['src']
        trg = vars(datum)['trg']
        pred_trg, _ = translate_sentence(src, src_field, trg_field,
model, device, max_len)
        pred_trgs.append(pred_trg)
        trgs.append([trg])
    return bleu_score(pred_trgs, trgs)
```

```
#cut off <eos> token
pred_trg = pred_trg[:-1]
pred_trgs.append(pred_trg)
trgs.append([trg])

return bleu_score(pred_trgs, trgs)

bleu_score = calculate_bleu(test_data, SRC, TRG, model, device)

print(f'BLEU score = {bleu_score*100:.2f}')
```

We get a BLEU score of 33.57, which is not bad while training fewer parameters and the training finishes in a matter of a few minutes.



In translation, a useful metric is the **Bilingual Evaluation Understudy (BLEU)** score, where 1 is the best possible value. It is the ratio

of parts in the candidate translation over parts in a reference translation (gold standard), where parts can be single words or a sequence of words (**n-grams**).

This wraps up our translation model. We can see it's actually not that hard to create a translation model. However, there's quite a lot of theory, part of which we'll cover in the next section.

How it works...

In this recipe, we trained a transformer model from scratch for an English to German translation task. Let's look a bit into what a transformer is and how it works.

Until not long ago, **Long Short-Term Memory networks (LSTMs)** had been the prevalent choice for deep learning models, however, since words are processed sequentially, training can take a long time to converge. We have seen in previous recipes how recurrent neural networks can be used for sequence processing (please compare it with the *Generating melodies* recipe in Chapter 9, *Deep Learning in Audio and Speech*). In yet other recipes, for example, the *Recognizing voice commands* recipe in Chapter 9, *Deep Learning in Audio and Speech*, we discussed how convolutional models have been replacing these recurrent networks with an advantage in speed and prediction performance. In NLP, convolutional networks have been tried as well (for example, Jonas Gehring and others, *Convolutional Sequence to Sequence Learning*, 2017) with improvements in speed and prediction performance with regard to recurrent models, however, the transformer architecture proved more powerful and still faster.

The transformer architecture was originally created for machine translation (Ashish Vaswani and others, *Attention is All you Need*, 2017). Dispensing with recurrence and convolutions, transformer networks are much faster to train and predict since words are processed in parallel. Transformer architectures provide universal language models that have pushed the envelope in a broad set of tasks such as **Neural Machine Translation (NMT)**, **Question Answering (QA)**, **Named-Entity Recognition (NER)**, **Textual Entailment (TE)**, abstractive text summarization, and other tasks. Transformer models are often taken off the shelf and fine-tuned for specific tasks in order to profit from general language understanding acquired through a long and expensive training process.

Transformers come in two parts, similar to an autoencoder:

- **An encoder** – it encodes the input into a series of context vectors (aka hidden states).
- **A decoder** – it takes the context vector and decodes it into a target representation.

The differences between the implementation in our recipe and the original transformer implementation (Ashish Vaswani and others, *Attention is All you Need*, 2017) is the following:

- We use a learned positional encoding instead of a static one.
- We use a fixed learning rate (no warm-up and cool-down steps).
- We don't use label smoothing.

These changes are in sync with modern transformers such as BERT.

First, the input is passed through an embedding layer and a positional embedding layer in order to encode the positions of tokens in the sequence. Token embeddings are scaled by $\sqrt{\text{hidden_dim}}$, the square root of the size of the hidden layers, and added to positional embeddings. Finally, dropout is applied for regularization.

The encoder then passes through stacked modules, each consisting of attention, feedforward fully connected layers, and normalization. Attention layers are linear combinations of scaled multiplicative (dot product) attention layers (**Multi-Head Attention**).

Some transformer architectures only contain one of the two parts. For example, the OpenAI GPT transformer architecture (Alec Radford and others, *Improving Language Understanding by Generative Pre-Training*, 2018), which generates amazingly coherent texts and consists of stacked decoders, while Google's BERT architecture (Jacob Devlin and others, *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*, 2019) also consists of stacked encoders.

There's more...

Both Torch and TensorFlow have a repository for pretrained models. We can download a translation model from the Torch hub and use it straight away. This is what we'll quickly show. For the pytorch model, we need to have a few dependencies installed first:

```
!pip install fairseq fastBPE sacremoses
```

After this, we can download the model. It is quite big, which means it'll take up a lot of disk space:

```
import torch

en2de = torch.hub.load(
    'pytorch/fairseq',
    'transformer.wmt19.en-de',
    checkpoint_file='model1.pt:model2.pt:model3.pt:model4.pt',
    tokenizer='moses',
    bpe='fastbpe'
)
en2de.translate('Machine learning is great!')
```

We should get an output like this:

```
Maschinelles Lernen ist großartig!
```

This model (Nathan Ng and others, *Facebook FAIR's WMT19 News Translation Task Submission*, 2019) is state-of-the-art for translation. It even outperforms human translations in precision (BLEU score). `fairseq` comes with tutorials for training translation models on your own datasets.

The Torch hub provides a lot of different translation models, but also generic language models.

See also

You can find a guide about the transformer architecture complete with PyTorch code (and an explanation on positional embeddings) on the Harvard NLP group website, which can also run on Google Colab: <http://nlp.seas.harvard.edu/2018/04/03/attention.html>.

Lilian Weng of OpenAI has written about language modeling and transformer models, and provides a concise and clear overview:

- Generalized Language Models – about the history of language and **Neural Machine Translation Models (NMTs)**: <https://lilianweng.github.io/lil-log/2019/01/31/generalized-language-models.html>
- The Transformer Family – about the history of transformer models: <https://lilianweng.github.io/lil-log/2020/04/07/the-transformer-family.html>

As for libraries supporting translation tasks, both `pytorch` and `tensorflow` provide pre-trained models, and support architectures useful in translation:

- `fairseq` is a library for sequence-to-sequence models in PyTorch: <https://github.com/pytorch/fairseq>.
- You can find tutorials on TensorFlow by Google Research on GitHub: <https://github.com/tensorflow/nmt>.

Finally, OpenNMT is a framework based on PyTorch and TensorFlow for translation tasks with many tutorials and pre-trained models: <https://opennmt.net/>.

Writing a popular novel

We've mentioned the Turing test before as a way of finding out whether a computer is intelligent enough to trick an interrogator into thinking it is human. Some text generation tools generate essays that could possibly make sense, however, contain no intellectual merit behind the appearance of scientific language.

The same could be said of some human essays and utterances, however. Nassim Taleb, in his book *Fooled by Randomness*, argued a person should be called unintelligent if their writing could not be distinguished from an artificially generated one (a **reverse Turing test**). In a similar vein, Alan Sokal's 1996 hoax article *Transgressing the Boundaries: Towards a Transformative Hermeneutics of Quantum Gravity*, accepted by and published in a well-known social science journal, was a deliberate attempt by the university professor of physics to expose a lack of intellectual rigor and the misuse of scientific terminology without understanding. A possible conclusion could be that imitating humans might not be the way forward toward intellectual progress.

OpenAI GPT-3, with 175 billion parameters, has pushed the field of language models considerably, having learned facts in physics, being able to generate programming code based on descriptions, and being able to compose entertaining and funny prose.

Millions of fans across the world have been waiting for more than 200 years to know how the story of *Pride and Prejudice* continues with Elizabeth and Mr Darcy. In this recipe, we'll be generating *Pride and Prejudice 2* using a transformer-based model.

Getting ready

Project Gutenberg is a digital library of (mostly) public domain e-books hosting more than 60,000 books in different languages and in formats such as plain text, HTML, PDF, EPUB, MOBI, and Plucker. Project Gutenberg also lists the most popular downloads: <http://www.gutenberg.org/browse/scores/top>.

At the time of writing, Jane Austen's romantic early-19th century novel *Pride and Prejudice* had by far the most downloads over the last 30 days (more than 47,000). We'll download the book in plain text format:

```
!wget -O pride_and_prejudice.txt  
http://www.gutenberg.org/files/1342/1342-0.txt
```

We save the text file as `pride_and_prejudice.txt`.

We'll be working in Colab, where you'll have access to Nvidia T4 or Nvidia K80 GPUs. However, you can use your own computer as well, using either GPUs or even CPUs.

If you are working in Colab, you'll need to upload your text file to your Google Drive (<https://drive.google.com>), where you can access it from Colab.

We'll be using a wrapper library for OpenAI's GPT-2 that's called `gpt-2-simple`, which is created and maintained by Max Woolf, a data scientist at BuzzFeed:

```
%tensorflow_version 1.x  
!pip install -q gpt-2-simple
```

This library will make it easy to fine-tune the model to new texts and show us text samples along the way.

We then have a choice of the size of the GPT-2 model. Four sizes of GPT-2 have been released by OpenAI as pretrained models:

- **Small** (124 million parameters; occupies 500 MB)
- **Medium** (355 million parameters; 1.5 GB)
- **Large** (774 million)
- **Extra large** (1,558 million)

The large model cannot currently be fine-tuned in Colab, but can generate text from the pretrained model. The extra large model is too large to load into memory in Colab, and can therefore neither be fine-tuned nor generate text. While bigger models will achieve better performance and have more knowledge, they will take longer to train and to generate text.

We'll choose the small model:

```
import gpt_2_simple as gpt2
gpt2.download_gpt2(model_name='124M')
```

Let's get started!

How to do it...

We've downloaded the text of a popular novel, *Pride and Prejudice*, and we'll first fine-tune the model, then we'll generate similar text to *Pride and Prejudice*:

1. Fine-tuning the model: We'll load a pre-trained model and fine-tune it for our texts.

We'll mount Google Drive. The `gpt-2-simple` library provides a utility function:

```
gpt2.mount_gdrive()
```

At this point, you'd need to authorize the Colab notebook to have access to your Google Drive. We'll use the *Pride and Prejudice* text file that we uploaded to our Google Drive before:

```
gpt2.copy_file_from_gdrive(file_name)
```

We can then start fine-tuning based on our downloaded text:

```
sess = gpt2.start_tf_sess()

gpt2.finetune(
    sess,
    dataset=file_name,
    model_name='124M',
    steps=1000,
    restore_from='fresh',
    run_name='run1',
    print_every=10,
    sample_every=200,
    save_every=500
)
```

We should see the training loss going down over the span of at least a couple of hours. We see samples of generated text during training such as this one:

she will now make her opinions known to the whole of the family, and to all their friends.

"At a time when so many middle-aged people are moving into town, when many couples are making fortunes off each other, when many professions of taste are forming in the society, I am really anxious to find some time here in the course of the next three months to write to my dear Elizabeth, to seek her informed opinion on this happy event, and to recommend it to her husband's conduct as well as her own. I often tell people to be cautious when they see connections of importance here. What is to be done after my death? To go through the world in such a way as to be forgotten?"

Mr. Bennet replied that he would write again at length to write very near to the last lines of his letter. Elizabeth cried out in alarm, and while she remained, a sense of shame had entered her of her being the less attentive companion she had been when she first distinguished her acquaintance. Anxiety increased every moment for the other to return to her senses, and every opportunity for Mr. Bennet to shine any brighter was given by the very first letter.

The gpt-2-simple library is really making it easy to train and continue training. All model checkpoints can be stored on Google Drive, so they aren't lost when the runtime times out. We might have to restart several times, so it's good to always store backups on Google Drive:

```
gpt2.copy_checkpoint_to_gdrive(run_name='run1')
```

If we want to continue training after Colab has restarted, we can do this as well:

```
# 1. copy checkpoint from google drive:  
gpt2.copy_checkpoint_from_gdrive(run_name='run1')  
  
# 2. continue training:  
sess = gpt2.start_tf_sess()  
gpt2.finetune(  
    sess,  
    dataset=file_name,  
    model_name='124M',  
    steps=500,
```

```
        restore_from='latest',
        run_name='run1',
        print_every=10,
        overwrite=True,
        sample_every=200,
        save_every=100
    )
# 3. let's backup the model again:
gpt2.copy_checkpoint_to_gdrive(run_name='run1')
```

We can now generate our new novel.

2. Writing our new bestseller: We might need to get the model from Google Drive and load it up into the GPU:

```
gpt2.copy_checkpoint_from_gdrive(run_name='run1')
sess = gpt2.start_tf_sess()
gpt2.load_gpt2(sess, run_name='run1')
```

Please note that you might have to restart your notebook (Colab) again so that the TensorFlow variables don't clash.

3. Now we can call a utility function in `gpt-2-simple` to generate the text into a file. Finally, we can download the file:

```
gen_file =
'gpt2_gentext_{:%Y%m%d_%H%M%S}.txt'.format(datetime.utcnow())

gpt2.generate_to_file(
    sess,
    destination_path=gen_file,
    temperature=0.7,
    nsamples=100,
    batch_size=20
)
files.download(gen_file)
```

The `gpt_2_simple.generate()` function takes an optional `prefix` parameter, which is the text that is to be continued.

Pride and Prejudice – the saga continues; reading the text, there are sometimes some obvious flaws in the continuity, however, some passages are captivating to read. We can always generate a few samples so that we have a choice of how our novel continues.

How it works...

In this recipe, we've used the GPT-2 model to generate text. This is called **neural story generation** and is a subset of **neural text generation**. Simply put, neural text generation is the process of building a statistical model of a text or of a language and applying this model to generate more text.

XLNet, OpenAI's GPT and GPT-2, Google's Reformer, OpenAI's Sparse Transformers, and other transformer-based models have one thing in common: they are generative because of a modeling choice – they are autoregressive rather than auto-encoding. This autoregressive language generation is based on the assumption that the probability of a token can be predicted given a context sequence of length n like this:

$$P(x_t | P_{t-1}, P_{t-2}, \dots, P_{t-n})$$

This can be approximated via minimizing the cross-entropy of the predicted token versus the actual token. LSTMs, **Generative Adversarial Networks (GANs)**, or autoregressive transformer architectures have been used for this.

One major choice we have to make in our text generation is how to sample, and we have a few choices:

- Greedy search
- Beam search
- Top-k sampling
- Top-p (nucleus) sampling

In greedy search, we take the highest rated choice each time, ignoring other choices. In contrast, rather than taking a high-scoring token, beam search tracks the scores of several choices in parallel in order to take the highest-scored sequence. Top-k sampling was introduced by Angela Fan and others (*Hierarchical Neural Story Generation*, 2018). In top-k sampling, all but the k most likely words are discarded. Conversely, in top-p (also called: nucleus sampling), the highest-scoring tokens surpassing probability threshold p are chosen, while the others are discarded. Top-k and top-p can be combined in order to avoid low-ranking words.

While the `huggingface transformers` library gives us all of these choices, with `gpt-2-simple`, we have the choice of top-k sampling and top-p sampling.

See also

There are many fantastic libraries that make training a model or applying an off-the-shelf model much easier. First of all, perhaps Hugging Face `transformers`, which is a library for language understanding and language generation supporting architectures and pretrained models for BERT, GPT-2, RoBERTa, XLM, DistilBert, XLNet, T5, and CTRL, among others: <https://github.com/huggingface/transformers>.

The Hugging Face `transformers` library comes with a few pre-trained transformer models, including a distilled GPT-2 model, which provides performance at the level of GPT-2, but with about 30% fewer parameters, bringing advantages of higher speed and lower resource demands in terms of memory and processing power. You can find a few notebooks linked from the Hugging Face GitHub repository that describe text generation and the fine-tuning of transformer models: <https://github.com/huggingface/transformers/tree/master/notebooks#community-notebooks>.

Additionally, Hugging Face provides a website called *Write with Transformers* that – according to their slogan – can *autocomplete your thoughts*: <https://transformer.huggingface.co/>.

You can find a tutorial on text generation with **recurrent neural networks** in the TensorFlow documentation: https://www.tensorflow.org/tutorials/text/text_generation.

Such models are also prepackaged in libraries such as `textgenrnn`: <https://github.com/minimaxir/textgenrnn>.

More complex, transformer-based models are also available from TensorFlow Hub, as demonstrated in another tutorial: https://www.tensorflow.org/hub/tutorials/wiki40b_lm.

11

Artificial Intelligence in Production

In the creation of a system that involves **artificial intelligence (AI)**, the actual AI usually only takes a small fraction of the total amount of work, while a major part of the implementation entails the surrounding infrastructure, starting from data collection and verification, feature extraction, analysis, resource management, and serving and monitoring (David Sculley and others. *Hidden technical debt in machine learning systems*, 2015).

In this chapter, we'll deal with monitoring and model versioning, visualizations as dashboards, and securing a model against malicious hacking attacks that could leak user data.

In this chapter, we'll be covering the following recipes:

- Visualizing model results
- Serving a model for live decisioning
- Securing a model against attack

Technical requirements

For Python libraries, we will work with models developed in TensorFlow and PyTorch, and we'll apply different, more specialized libraries in each recipe.

As always, you can find the recipe notebooks on GitHub at <https://github.com/PacktPublishing/Artificial-Intelligence-with-Python-Cookbook/tree/master/chapter11>.

Visualizing model results

Frequent communication with business shareholders is key to getting the buy-in for deploying an AI solution, and should start from the idea and premises to decisions and findings. How results are communicated can make the difference between success and failure in a commercial setting. In this recipe, we'll be building a visualization solution for a **machine learning (ML)** model.

Getting ready

We'll be building our solution in `streamlit` (<https://www.streamlit.io/>) and we'll be using visualizations from `altair`, one of the many Python plotting libraries that `streamlit` integrates with (a list that also includes `matplotlib` and `plotly`). Let's install `streamlit` and `altair`:

```
pip install streamlit altair
```

We won't use the notebook in this recipe. Therefore, we've omitted the exclamation marks in this code block. We'll be running everything from the terminal.

`Altair` has a very pleasant declarative way to plot graphs, which we'll see in the recipe. `Streamlit` is a framework to create data apps – interactive applications in the browser with visualizations.

Let's proceed with building an interactive data app.

How to do it...

We'll be building a simple app for model building. This is meant to show how easy it is to create a visual interactive application for the browser in order to demonstrate findings to non-technical or technical audiences.

For a very quick, practical introduction to `streamlit`, let's look at how a few lines of code in a Python script can be served.

Streamlit hello-world

We'll write our `streamlit` applications as Python scripts, not as notebooks, and we'll execute the scripts with `streamlit` to be deployed.

We'll create a new Python file, let's say `streamlit_test.py`, in our favorite editor, for example, vim, and we'll write these lines:

```
import streamlit as st

chosen_option = st.sidebar.selectbox('Hello', ['A', 'B', 'C'])
st.write(chosen_option)
```

This would show a select box or drop-down menu with the title *Hello* and a choice between options *A*, *B*, and *C*. This choice will be stored in the `chosen_option` variable, which we can output in the browser.

We can run our intro app from the terminal as follows:

```
streamlit run streamlit_test.py --server.port=80
```

The server port option is optional.

This should open our browser in a new tab or window showing our drop-down menu with the three choices. We can change the option, and the new value will be displayed.

This should be enough for an introduction. We'll come to the actual recipe now.

Creating our data app

The main idea of our data app is that we incorporate decisions such as modeling choices into our application, and we can observe the consequences, both summarized in numbers and visually in plots.

We'll start by implementing the core functionality, such as modeling and dataset loading, and then we'll create the interface to it, first in the side panel, and then the main page. We'll write all of the code in this recipe to a single Python script that we can call `visualizing_model_results.py`:

1. Loading the dataset – we'll implement dataset loaders:

Let's begin with a few preliminaries such as imports:

```
import numpy as np
import pandas as pd
import altair as alt
import streamlit as st

from sklearn.datasets import (
    load_iris,
```

```
    load_wine,
    fetch_covtype
)
from sklearn.model_selection import train_test_split
from sklearn.ensemble import (
    RandomForestClassifier,
    ExtraTreesClassifier,
)
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import roc_auc_score
from sklearn.metrics import classification_report
```

If you've followed the hello-world introduction attentively, you might have wondered how the interface communicates with Python. This is handled by streamlit by re-running your script every time the user makes a change by clicking somewhere or entering a field.

We need to load datasets into memory. This can include a download step, and for bigger datasets, downloading could potentially take a long time. Therefore, we are going to cache this step to disk, so instead of downloading every time there's a button-click, we'll retrieve it from the cache on disk:

```
dataset_lookup = {
    'Iris': load_iris,
    'Wine': load_wine,
    'Covertype': fetch_covtype,
}

@st.cache
def load_data(name):
    iris = dataset_lookup[name]()
    X_train, X_test, y_train, y_test = train_test_split(
        iris.data, iris.target, test_size=0.33, random_state=42
    )
    feature_names = getattr(
        iris, 'feature_names',
        [str(i) for i in range(X_train.shape[1])]
    )
    target_names = getattr(
        iris, 'target_names',
        [str(i) for i in np.unique(iris.target)]
    )
    return (
        X_train, X_test, y_train, y_test,
        target_names, feature_names
    )
```

This implements the functions for modeling and dataset loaders.

Please note the use of the streamlit cache decorator, `@st.cache`. It handles the caching of the decorated function, in this case, `load_data()`, in such a way that any number of parameters passed to the function will be stored together with the associated outputs.

Here, the dataset loading might take some time. However, caching means that we only have to load each dataset exactly once, because subsequently the dataset will be retrieved from cache, and therefore loading will be much faster. This caching functionality, which can be applied to long-running functions, is central to making streamlit respond more quickly.

We are using the scikit-learn datasets API to download a dataset. Since scikit-learn's `load_x()` type functions, such as `load_iris()`, which are mostly for toy datasets, include attributes such as `target_names` and `feature_names`, but scikit-learn's `fetch_x()` functions, such as `fetch_covtype()`, are intended for bigger, more serious datasets, we'll generate feature and target names for these separately.

The training procedure is similarly decorated to be cached. However, please note that we have to include our hyperparameters in order to make sure that the cache is unique to the model type, dataset, and all hyperparameters as well:

```
@st.cache
def train_model(dataset_name, model_name, n_estimators, max_depth):
    model = [m for m in models if m.__class__.__name__ ==
model_name][0]
    with st.spinner('Building a {} model for {} ...'.format(
        model_name, dataset_name
    )):
        return model.fit(X_train, y_train)
```

The modeling function takes a list of models, which we'll update based on the choices of hyperparameters. We'll implement this choice now.

2. Exposing key decisions in the side panel:

In the side panel, we'll be presenting the choices of datasets, model type, and hyperparameters. Let's start by choosing the dataset:

```
st.sidebar.title('Model and dataset selection')
dataset_name = st.sidebar.selectbox(
    'Dataset',
    list(dataset_lookup.keys()))
```

```
)  
(X_train, X_test, y_train, y_test,  
 target_names, feature_names) = load_data(dataset_name)
```

This will load the datasets after we've made the choice between iris, wine, and cover type.

For the model hyperparameters, we'll provide slider bars:

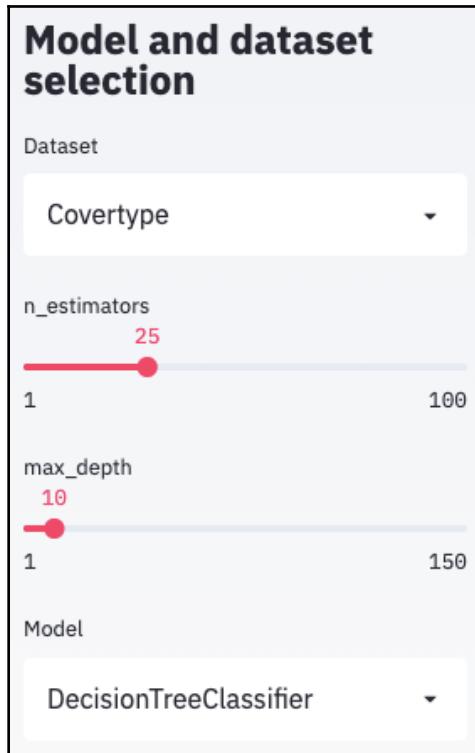
```
n_estimators = st.sidebar.slider(  
    'n_estimators',  
    1, 100, 25  
)  
max_depth = st.sidebar.slider(  
    'max_depth',  
    1, 150, 10  
)
```

Finally, we will expose the model type again as a drop-down menu:

```
models = [  
    DecisionTreeClassifier(max_depth=max_depth),  
    RandomForestClassifier(  
        n_estimators=n_estimators,  
        max_depth=max_depth  
    ),  
    ExtraTreesClassifier(  
        n_estimators=n_estimators,  
        max_depth=max_depth  
    ),  
]  
model_name = st.sidebar.selectbox(  
    'Model',  
    [m.__class__.__name__ for m in models]  
)  
model = train_model(dataset_name, model_name, n_estimators,  
max_depth)
```

In the end, after the choice, we will call the `train_model()` function with the dataset, model type, and hyperparameters as arguments.

This screenshot shows what the side panel looks like:



This shows you the menu options in the browser. We'll show the results of these choices in the main part of the browser page.

3. Reporting classification results in the main panel:

In the main panel, we'll be showing important statistics, including a classification report, a few plots that should give insights into the model strengths and weaknesses, and a view of the data itself, where incorrect decisions by the model will be highlighted.

We first need a title:

```
st.title('{model} on {dataset}'.format(  
    model=model_name,  
    dataset=dataset_name  
)
```

Then we present basic statistics in relation to our modeling result, such as the area under the curve, precision, and many more:

```

predictions = model.predict(X_test)
probs = model.predict_proba(X_test)
st.subheader('Model performance in test')
st.write('AUC: {:.2f}'.format(
    roc_auc_score(
        y_test, probs,
        multi_class='ovo' if len(target_names) > 2 else 'raise',
        average='macro' if len(target_names) > 2 else None
    )
))
st.write(
    pd.DataFrame(
        classification_report(
            y_test, predictions,
            target_names=target_names,
            output_dict=True
        )
    )
)

```

We'll then show a confusion matrix that tabulates the actual and predicted labels for each of the classes:

```

test_df = pd.DataFrame(
    data=np.concatenate([
        X_test,
        y_test.reshape(-1, 1),
        predictions.reshape(-1, 1)
    ], axis=1),
    columns=feature_names + [
        'target', 'predicted'
    ]
)
target_map = {i: n for i, n in enumerate(target_names)}
test_df.target = test_df.target.map(target_map)
test_df.predicted = test_df.predicted.map(target_map)
confusion_matrix = pd.crosstab(
    test_df['target'],
    test_df['predicted'],
    rownames=['Actual'],
    colnames=['Predicted']
)
st.subheader('Confusion Matrix')
st.write(confusion_matrix)

```

We also want to be able to scroll through our test data to be able to inspect samples that were incorrectly classified:

```
def highlight_error(s):
    if s.predicted == s.target:
        return ['background-color: None'] * len(s)
    return ['background-color: red'] * len(s)

if st.checkbox('Show test data'):
    st.subheader('Test data')
    st.write(test_df.style.apply(highlight_error, axis=1))
```

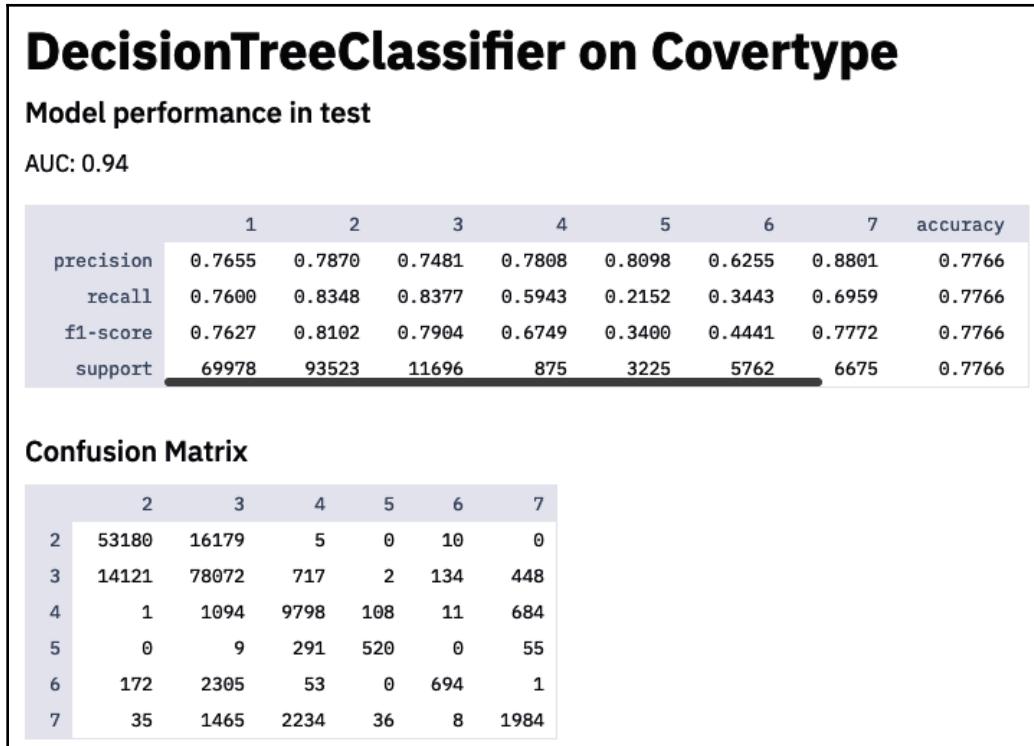
Incorrectly classified samples will be highlighted against a red background. We've made this raw data exploration optional. It needs to be activated by clicking a checkbox.

Finally, we'll show a facet plot of variables plotted against each other in scatter plots. This is the part where we use the altair library:

```
if st.checkbox('Show test distributions'):
    st.subheader('Distributions')
    row_features = feature_names[:len(feature_names)//2]
    col_features = feature_names[len(row_features):]
    test_df_with_error = test_df.copy()
    test_df_with_error['error'] = test_df.predicted ==
    test_df.target
    chart = alt.Chart(test_df_with_error).mark_circle().encode(
        alt.X(alt.repeat("column"), type='quantitative'),
        alt.Y(alt.repeat("row"), type='quantitative'),
        color='error:N'
    ).properties(
        width=250,
        height=250
    ).repeat(
        row=row_features,
        column=col_features
    ).interactive()
    st.write(chart)
```

Incorrectly classified examples are highlighted in these plots. Again, we've made this part optional, activated by marking a checkbox.

The upper part of the main page for the Covtype dataset looks like this:



You can see the classification report and the confusion matrix. Below these (not part of the screenshot) would be the data exploration and the data plots.

This concludes our demo app. Our app is relatively simple, but hopefully this recipe can serve as a guide for building these apps for clear communication.

How it works...

This book is about hands-on learning, and we'd recommend this for streamlit as well. Working with streamlit, you have a quick feedback loop where you implement changes and see the results, and you continue until you are happy with what you see.

Streamlit provides a local server that you can access remotely over the browser if you want. So you can run your streamlit application server on Azure, Google Cloud, AWS, or your company cloud, and see your results in your local browser.

What is important to understand is the streamlit workflow. Values for widgets are stored by streamlit. Other values are recomputed on the fly every time a user interacts with a widget as the Python script is executed again from top to bottom. In order to avoid expensive computations, these can be cached using the `@st.cache` decorator, as we've seen in the recipe.

Streamlit's API has an integration for many plotting and graphing libraries. These include Matplotlib, Seaborn, Plotly, Bokeh, interactive plotting libraries, such as Altair, Vega Lite, deck.gl for maps and 3D charts, and graphviz graphs. Other integrations include Keras models, SymPy expressions, pandas DataFrames, images, audio, and others.

Streamlit also comes with several types of widgets, such as sliders, buttons, and drop-down menus. Streamlit also includes an extensible component system, where each component consists of a browser frontend in HTML and JavaScript and a Python backend, able to send and receive information bi-directionally. Existing components interface with further libraries, including HiPlot, Echarts, Spacy, and D3, to name but a few: <https://www.streamlit.io/components>.

You can play around with different inputs and outputs, you can start from scratch, or you can improve on the code in this recipe. We could extend it to show different results, build dashboards, connect to a database for live updates, or build user feedback forms for subject matter experts to relay their judgment such as, for example, annotation or approval.

See also

Visualization in AI and statistics is a broad field. Fernanda Viégas and Martin Wattenberg gave an overview talk, *Visualization for Machine Learning*, at NIPS 2018, and you can find their slide deck and a recording of their talk.

Here's a list of a few interesting streamlit demos to look at:

- Using a live TensorFlow session to create an interactive face-GAN explorer: <https://github.com/streamlit/demo-face-gan/>.
- An image browser for the Udacity self-driving car dataset and real-time object detection: <https://github.com/streamlit/demo-self-driving>.
- A components demo for maps, audio, images, and many others: <https://fullstackstation.com/streamlit-components-demo>.

Aside from streamlit, there are other libraries and frameworks that can help to create interactive dashboards, presentations, and reports, such as Bokeh, Jupyter Voilà, Panel, and Plotly Dash.

If you are looking for dashboarding and live charting with database integration, tools such as Apache Superset come in handy: <https://superset.apache.org/>.

Serving a model for live decisioning

Often, specialists in AI are asked to model, present, or come up with models. However, even though the solution could be commercially impactful, in practice, productionizing a **proof of concept (POC)** for live decisioning in order to actually act on the insight can be a bigger struggle than coming up with the models in the first place. Once we've created a model based on training data, analyzed it to verify that it's working to an expected standard, and communicated with stakeholders, we want to make that model available so it can provide predictions on data for new decisions. This can mean certain requirements, such as latency (for real-time applications), and bandwidth (for servicing a large volume of customers). Often, a model is deployed as part of a microservice such as an inference server.

In this recipe, we'll build a small inference server from scratch, and we'll focus on the technical challenges around bringing AI into production. We'll showcase how to develop a POC into a software solutions that is fit for production by being robust, scaling to demand, responding timely, and that you can update as fast as needed.

Getting ready

We'll have to switch between the terminal and the Jupyter environment in this recipe. We'll create and log the model from the Jupyter environment. We'll control the `mlflow` server from the terminal. We will note which one is appropriate for each code block.

We'll use `mlflow` in this recipe. Let's install it from the terminal:

```
pip install mlflow
```

We'll assume you have `conda` installed. If not, please refer to the *Setting up a Jupyter environment* recipe in Chapter 1, *Getting Started with Artificial Intelligence in Python*, for detailed instructions.

We can start our local `mlflow` server with a `sqlite` database backend for backend storage from the terminal like this:

```
mlflow server --backend-store-uri sqlite:///mlflow.db --host 0.0.0.0 --  
default-artifact-root file:///$PWD/mlruns
```

We should see a message that our server is listening at `http://0.0.0.0:5000`.

This is where we can access this server from our browser. In the browser, we'll be able to compare and check different experiments, and see the metrics of our models.

In the *There's more...* section, we'll do a quick demo of setting up a custom API using the `FastAPI` library. We'll quickly install this library as well:

```
!pip install fastapi
```

With this, we are ready to go!

How to do it...

We'll build a simple model from a **column-separated value (CSV)** file. We'll try different modeling options, and compare them. Then we'll deploy this model:

1. Downloading and preparing the dataset:

We'll download a dataset as a CSV file and prepare for training. The dataset chosen in this recipe is the Wine dataset, describing the quality of wine samples. We'll download and read the wine-quality CSV file from the UCI ML archive:

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

csv_url = \
    'http://archive.ics.uci.edu/ml/machine-' \
    'learning-databases/wine-quality/winequality-red.csv'
data = pd.read_csv(csv_url, sep=';')
```

We split the data into training and test sets. The predicted column is **column quality**:

```
train_x, test_x, train_y, test_y = train_test_split(
    data.drop(['quality'], axis=1),
    data['quality']
)
```

2. Training with different hyperparameters:

We can track as much as we like in the `mlflow` server. We can create a reporting function for performance metrics like this:

```
from sklearn.metrics import mean_squared_error,
mean_absolute_error, r2_score

def eval_metrics(actual, pred):
    rmse = np.sqrt(mean_squared_error(actual, pred))
    mae = mean_absolute_error(actual, pred)
    r2 = r2_score(actual, pred)
    return rmse, mae, r2
```

Before running our training, we need to register the `mlflow` library with the server:

```
import mlflow

mlflow.set_tracking_uri('http://0.0.0.0:5000')
mlflow.set_experiment('/wine')
```

We set our server URI. We can also give our experiment a name.

Each time we run the training set with different options, MLflow can log the results, including metrics, hyperparameters, the pickled model, and a definition as MLModel that captures library versions and creation time.

In our training function, we train on our training data, extracting metrics of our model over the test data. We need to choose the appropriate hyperparameters and metrics for comparison:

```
from sklearn.linear_model import ElasticNet
import mlflow.sklearn

np.random.seed(40)

def train(alpha=0.5, l1_ratio=0.5):
    with mlflow.start_run():
        lr = ElasticNet(alpha=alpha, l1_ratio=l1_ratio, random_state=42)
        lr.fit(train_x, train_y)
        predicted = lr.predict(test_x)
        rmse, mae, r2 = eval_metrics(test_y, predicted)

        model_name = lr.__class__.__name__
        print('{} (alpha={}, l1_ratio={})'.format(
            model_name, alpha, l1_ratio
```

```
)  
    print(' RMSE: %s' % rmse)  
    print(' MAE: %s' % mae)  
    print(' R2: %s' % r2)  
  
    mlflow.log_params({key: value for key, value in  
lr.get_params().items()})  
    mlflow.log_metric('rmse', rmse)  
    mlflow.log_metric('r2', r2)  
    mlflow.log_metric('mae', mae)  
    mlflow.sklearn.log_model(lr, model_name)
```

We fit the model, extract our metrics, print them to the screen, log them to the `mlflow` server, and store the model artifact on the server as well.

For convenience, we are exposing our model hyperparameter in the `train()` function. We chose to log all the hyperparameters with `mlflow`. Alternatively, we could have logged only store-relevant parameters like this:

```
mlflow.log_param('alpha', alpha).
```

We could also calculate more artifacts to accompany our model, for example, variable importance.

We can also try with different hyperparameters, for example, as follows:

```
train(0.5, 0.5)
```

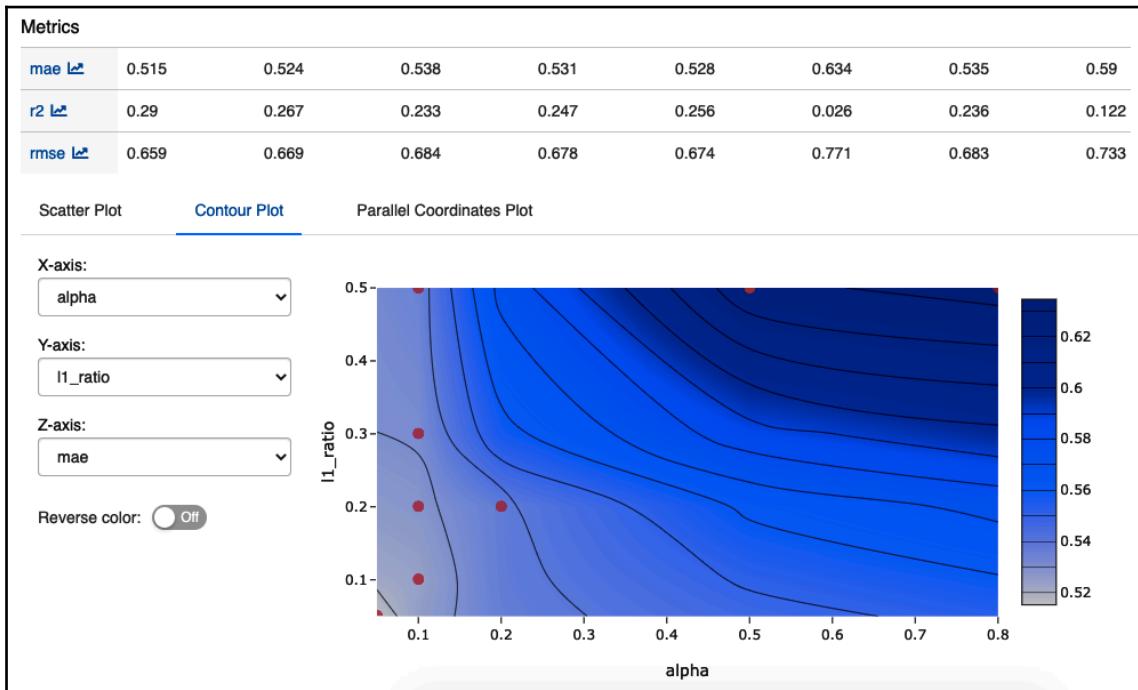
We should get the performance metrics as an output:

```
ElasticNet (alpha=0.5, l1_ratio=0.5):  
RMSE: 0.7325693777577805  
MAE: 0.5895721434715478  
R2: 0.12163690293641838
```

After we've run this for a number of times with different parameters, we can go to our server, compare model runs, and choose a model for deployment.

3. Deploying a model as a local server. We can compare our models in the browser. We should be able to find our wine experiments under the experiments tab on our server.

We can then compare different model runs in the overview table, or get an overview plot for different hyperparameters, such as this:



This contour plot shows us the two hyperparameters we've changed against the **Mean Average Error (MAE)**.

We can then choose a model for deployment. We can see the run ID for our best model. Deployment of a model to a server can be done from the command line, for example, like this:

```
mlflow models serve -m
/Users/ben/mlflow/examples/sklearn_elasticnet_wine/mlruns/1/208e2f525081433
5977b265b328c5c49
/artifacts/ElasticNet/
```

We can pass data as JSON, for example, using curl, again from the terminal. This could look as follows:

```
curl -X POST -H "Content-Type:application/json; format=pandas-split" --data
'{"columns":["alcohol", "chlorides", "citric acid", "density", "fixed
acidity", "free sulfur dioxide", "pH", "residual sugar", "sulphates",
"total sulfur dioxide", "volatile acidity"],"data":[[1.2, 0.231, 0.28,
0.61, 4.5, 13, 2.88, 2.1, 0.26, 63, 0.51]]}'
http://127.0.0.1:1234/invocations
```

With this, we've finished our demo of model deployment with mlflow.

How it works...

The basic workflow for productionizing a model is as follows:

- Train a model on data.
- Package the model itself in a reusable and reproducible model format together with glue code that extracts a prediction from the model.
- Deploy the model in an HTTP server that will enable you to score predictions.

This typically results in a microservice that communicates via JSON (usually this is then called a RESTful service) or GRPC (remote procedure calls via Google's protocol buffers). This has the advantage of being able to disentangle shipping of the decisioning intelligence from the backend, and have ML experts take full ownership of their solution.



A **microservice** is a single service that is independently deployable, maintainable, and testable. Structuring an application as a collection of loosely coupled microservices is called a **microservice architecture**.

Another route would be to package your model and glue code for deployment within the existing enterprise backend of your company. This integration has several alternatives:

- In model interchange formats such as the **Predictive Model Markup Language (PMML)**, a format developed by a group of organizations under the umbrella of the Data Mining Group.

- Some software implementations, such as LightGBM, XGBoost, or TensorFlow have APIs in multiple programming languages, so that models can be developed in Python, and loaded up from languages such as C, C++, or Java.
- Re-engineering your model:
 - Some tools can help to convert models such as decision trees into C or other languages.
 - This can be done manually as well.

MLflow has command-line, Python, R, Java, and REST API interfaces for uploading models to a model repository, for logging model results (**experiments**), for uploading models to a model repository, for downloading them again to use them locally, for controlling the server, and much more. It offers a server, however, that also allows deployment to Azure ML, Amazon Sagemaker, Apache Spark UDF, and RedisAI.

If you want to be able to access your `mlflow` server remotely, such as the case usually when using the model server as an independent service (microservice), we want to set the root to `0.0.0.0`, as we've done in the recipe. By default, the local server will start up at `http://127.0.0.1:5000`.

If we want to access models, we need to switch from the default backend storage (this is where metrics will be stored) to a database backend, and we have to define our artifact storage using a protocol in the URL, such as `file://$PWD/mlruns` for the local `mlruns/` directory. We've enabled a SQLite database for the backend, which is the easiest way (but probably not the best for production). We could have chosen MySQL Postgres, or another database backend as well.

This is only part of the challenge, however, because models become stale or might be unsuitable, facts we can only establish if we are equipped to monitor model and server performance in deployment. Therefore, a note on monitoring is in order.

Monitoring

When monitoring AI solutions, what we are especially interested in are metrics that are either operational or relate to appropriate decision making. Metrics of the former kind are as follows:

- **Latency** – How long does it take to perform a prediction on data?
- **Throughput** – How many data points can we process in any timeframe?
- **Resource usage** – How much CPU, memory, and disk space do we allocate when completing inference?

The following metrics could form part of monitoring the decision-making process:

- Statistical indicators, such as averages, variances of predictions over a certain period of time
- Outlier and drift detection
- The business impact of decisions

For methods to detect outliers, please refer to the *Discovering anomalies* recipe in Chapter 3, *Patterns, Outliers, and Recommendations*.

Standalone monitoring could be built from scratch following a template similar to the *Visualizing model results* recipe in this chapter, or be integrated with more specialist monitoring solutions, such as Prometheus, Grafana, or Kibana (for log monitoring).

See also

This is a very broad topic, and we've mentioned many aspects of productionization in the *How it works...* section of this recipe. There are many competing industrial-strength solutions for ML and **deep learning (DL)** models, and we can only try to give an overview given the space constraint. As always in this book, we'll be mostly concentrating on open source solutions that will avoid a vendor lock-in:

- MLflow aspires to manage the whole ML life cycle, including experimentation, reproducibility, deployment, and a central model registry: <https://mlflow.org/>.
- BentoML creates a high-performance API endpoint serving trained models: <https://github.com/bentoml/bentoml>.

While some tools support only one or a few modeling frameworks, others, particularly BentoML and MLflow, support deploying models trained under all major ML training frameworks such as FastAI, scikit-learn, PyTorch, Keras, XGBoost, LightGBM, H2o, FastText, Spacy, and ONNX. Both of these further provide maximum flexibility for anything created in Python, and they both have a tracking functionality for monitoring.

Our recipe was adapted from the mlflow tutorial example. MLflow has many more examples for different modeling framework integrations on GitHub: <https://github.com/mlflow/mlflow/>.

Other tools include the following:

- Elyra is a cloud-based deployment solution for Jupyter notebooks that comes with a visual data flow editor: <https://elyra.readthedocs.io/en/latest/index.html>.
- RedisAI is a Redis module for executing DL/ML models and managing their data: <https://oss.redislabs.com/redisai/>.
- TFX is a Google production-scale ML platform: <https://www.tensorflow.org/tfx/guide>.
- TorchServe is a tool for serving PyTorch models: <https://pytorch.org/serve/>.

Furthermore, there are many libraries available for creating custom microservices. Two of the most popular such libraries are these:

- Flask: <https://palletsprojects.com/p/flask/>
- FastAPI: <https://fastapi.tiangolo.com/>

Using these, you can create endpoints that would take data such as images or text and return a prediction.

Securing a model against attack

Adversarial attacks in ML refer to fooling a model by feeding input with the purpose of deceiving it. Examples of such attacks include adding perturbations to an image by changing a few pixels, thereby causing the classifier to misclassify the sample, or carrying t-shirts with certain patterns to evade person detectors (**adversarial t-shirts**). One particular kind of adversarial attack is a **privacy attack**, where a hacker can gain knowledge of the training dataset of the model, potentially exposing personal or sensitive information by membership inference attacks and model inversion attacks.

Privacy attacks are dangerous, particularly in domains such as medical or financial, where the training data can involve sensitive information (for example, a health status) and that is possibly traceable to an individual's identity. In this recipe, we'll build a model that is safe against privacy attacks, and therefore cannot be hacked.

Getting ready

We'll implement a PyTorch model, but we'll rely on a script in the TensorFlow/privacy repository created and maintained by Nicolas Papernot and others. We'll clone the repository as follows:

```
!git clone https://github.com/tensorflow/privacy
```

Later during the recipe, we'll use the analysis script to calculate the privacy bounds of our model.

How to do it...

We'll have to define data loaders for teacher models and the student model. The teacher and student architectures are the same in our case. We'll train the teachers, and then we train the student from the aggregates of the teacher responses. We'll close with a privacy analysis executing the script from the privacy repository.

This is adapted from a notebook by Diego Muñoz that is available on GitHub: https://github.com/dimun/pate_torch:

1. Let's start by loading the data. We'll download the data using `torch` utility functions:

```
from torchvision import datasets
import torchvision.transforms as transforms

batch_size = 32

transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.1307,), (0.3081,))]
)

train_data = datasets.MNIST(
    root='data', train=True,
    download=True,
    transform=transform
)
test_data = datasets.MNIST(
    root='data', train=False,
    download=True,
    transform=transform
)
```

This will load the MNIST dataset, and may take a moment. The transform converts data to `torch.FloatTensor.train_data` and `test_data` define the loaders for training and test data, respectively.

Please refer to the *Recognizing clothing items* recipe in Chapter 7, *Advanced Image Applications*, for a brief discussion of the MNIST dataset, and the *Generating images* recipe in the same chapter for another model using the dataset.

Please note that we'll define a few parameters in an ad hoc fashion throughout the recipe. Among these are `num_teachers` and `standard_deviation`. You'll see an explanation of the algorithm in the *How it works...* section and, hopefully, the parameters will make sense then.

Another parameter, `num_workers`, defines the number of subprocesses to use for data loading. `batch_size` defines how many samples per batch to load.

We'll define data loaders for the teachers:

```
num_teachers = 100

def get_data_loaders(train_data, num_teachers=10):
    teacher_loaders = []
    data_size = len(train_data) // num_teachers

    for i in range(num_teachers):
        indices = list(range(i * data_size, (i+1) * data_size))
        subset_data = Subset(train_data, indices)
        loader = torch.utils.data.DataLoader(
            subset_data,
            batch_size=batch_size,
            num_workers=num_workers
        )
        teacher_loaders.append(loader)

    return teacher_loaders

teacher_loaders = get_data_loaders(train_data, num_teachers)
```

The `get_data_loaders()` function implements a simple partitioning algorithm that returns the right portion of the data needed by a given teacher out of a certain number of teachers. Each teacher model will get a disjoint subset of the training data.

We define a training set for the student of 9,000 training samples and 1,000 test samples. Both sets are taken from the teachers' test dataset as unlabeled training points – they will be labeled using the teacher predictions:

```
import torch
from torch.utils.data import Subset

student_train_data = Subset(test_data, list(range(9000)))
student_test_data = Subset(test_data, list(range(9000, 10000)))

student_train_loader = torch.utils.data.DataLoader(
    student_train_data, batch_size=batch_size,
    num_workers=num_workers
)
student_test_loader = torch.utils.data.DataLoader(
    student_test_data, batch_size=batch_size,
    num_workers=num_workers
)
```

2. Defining the models: We are going to define a single model for all the teachers:

```
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x)
```

This is a convolutional neural network for image processing. Please refer to Chapter 7, *Advanced Image Applications*, for more image processing models.

Let's create another utility function for prediction from these models given a dataloader:

```
def predict(model, dataloader):
    outputs = torch.zeros(0, dtype=torch.long).to(device)
    model.to(device)
    model.eval()
    for images, labels in dataloader:
        images, labels = images.to(device), labels.to(device)
        output = model.forward(images)
        ps = torch.argmax(torch.exp(output), dim=1)
        outputs = torch.cat((outputs, ps))
    return outputs
```

We can now start training the teachers.

3. Training the teacher models:

First, we'll implement a function to train the models:

```
device = torch.device('cuda:0' if torch.cuda.is_available() else
'cpu')

def train(model, trainloader, criterion, optimizer, epochs=10,
print_every=120):
    model.to(device)
    steps = 0
    running_loss = 0
    for e in range(epochs):
        model.train()
        for images, labels in trainloader:
            images, labels = images.to(device), labels.to(device)
            steps += 1
            optimizer.zero_grad()
            output = model.forward(images)
            loss = criterion(output, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
```

We are now ready to train our teachers:

```
from tqdm.notebook import trange

def train_models(num_teachers):
    models = []
    for t in trange(num_teachers):
        model = Net()
```

```
        criterion = nn.NLLLoss()
        optimizer = optim.Adam(model.parameters(), lr=0.003)
        train(model, teacher_loaders[t], criterion, optimizer)
        models.append(model)
    return models

models = train_models(num_teachers)
```

This instantiates and trains the models for each teacher.

4. Training the student:

For the student, we require an aggregation function. You can see an explanation of the aggregation function in the *How it works...* section:

```
import numpy as np

def aggregated_teacher(models, data_loader,
standard_deviation=1.0):
    preds = torch=torch.zeros((len(models), 9000),
    dtype=torch.long)
    print('Running teacher predictions...')
    for i, model in enumerate(models):
        results = predict(model, data_loader)
        preds[i] = results
    print('Calculating aggregates...')
    labels = np.zeros(preds.shape[1]).astype(int)
    for i, image_preds in enumerate(np.transpose(preds)):
        label_counts = np.bincount(image_preds,
        minlength=10).astype(float)
        label_counts += np.random.normal(0, standard_deviation,
        len(label_counts))
        labels[i] = np.argmax(label_counts)
    return preds.numpy(), np.array(labels)

standard_deviation = 5.0
teacher_models = models
preds, student_labels = aggregated_teacher(
    teacher_models,
    student_train_loader,
    standard_deviation
)
```

The `aggregated_teacher()` function makes the predictions for all the teachers, counts the votes, and adds noise. Finally, it returns the votes and the results aggregated by `argmax`.

`standard_deviation` defines the standard deviation for noise. This is important for the privacy guarantees.

The student requires a data loader first:

```
def student_loader(student_train_loader, labels):
    for i, (data, _) in enumerate(iter(student_train_loader)):
        yield data,
    torch.from_numpy(labels[i*len(data):(i+1)*len(data)])
```

This student data loader will be fed the aggregated teacher label:

```
student_model = Net()
criterion = nn.NLLLoss()
optimizer = optim.Adam(student_model.parameters(), lr=0.001)
epochs = 10
student_model.to(device)
steps = 0
running_loss = 0
for e in range(epochs):
    student_model.train()
    train_loader = student_loader(student_train_loader,
    student_labels)
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        steps += 1
        optimizer.zero_grad()
        output = student_model.forward(images)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    # <validation code omitted>
```

This runs the student training.

Some parts of this code have been omitted from the training loop for the sake of brevity. The validation looks like this:

```
if steps % 50 == 0:
    test_loss = 0
    accuracy = 0
    student_model.eval()
    with torch.no_grad():
        for images, labels in student_test_loader:
            images, labels = images.to(device), labels.to(device)
            log_ps = student_model(images)
            test_loss += criterion(log_ps, labels).item()
```

```

        ps = torch.exp(log_ps)
        top_p, top_class = ps.topk(1, dim=1)
        equals = top_class == labels.view(*top_class.shape)
        accuracy += torch.mean(equals.type(torch.FloatTensor))
    student_model.train()
    print('Training Loss: {:.3f}..'
        .format(running_loss/len(student_train_loader)),
        'Test Loss: {:.3f}..'
        .format(test_loss/len(student_test_loader)),
        'Test Accuracy: '
        {:.3f}'.format(accuracy/len(student_test_loader)))
    running_loss = 0

```

The final training update reads as follows:

```
Epoch: 10/10.. Training Loss: 0.026.. Test Loss: 0.190.. Test
Accuracy: 0.952
```

We see that it's a good model: 95.2 percent accuracy on the test dataset.

5. Analyzing the privacy:

In Papernot and others (2018), they detail how data-dependent differential privacy bounds can be computed to estimate the cost of training the student.

They provide a script to do this analysis based on the vote counts and the used standard deviation of the noise. We've clone this repository earlier, so we can change into a directory within it, and execute the analysis script:

```
%cd privacy/research/pate_2018/ICLR2018
```

We need to save the aggregated teacher counts as a NumPy file. This can then be loaded by the analysis script:

```

clean_votes = []
for image_preds in np.transpose(preds):
    label_counts = np.bincount(image_preds,
        minlength=10).astype(float)
    clean_votes.append(label_counts)

clean_votes = np.array(counts)
with open('clean_votes.npy', 'wb') as file_obj:
    np.save(file_obj, clean_votes)

```

This puts together the `counts` matrix, and saves it as a file.

Finally, we call the privacy analysis script:

```
!python smooth_sensitivity_table.py --sigma2=5.0 --
counts_file=clean_votes.npy --delta=1e-5
```

The epsilon privacy guarantee is estimated at 34.226 (data-independent) and 6.998 (data-dependent). The epsilon value is not intuitive by itself, but needs an interpretation in the context of the dataset and its dimensions.

How it works...

We've created a set of teacher models from a dataset, and then we bootstrapped from these teachers a student model that gives privacy guarantees. In this section, we'll discuss some background about the problem of privacy in ML, differential privacy, and how PATE works.

Leaking data about customers can bring great reputational damage to a company, not to speak of fees from the regulator for being in violation of data protection and privacy laws such as GDPR. Therefore, considering privacy in the creation of datasets and in ML is as important as ever. As a point in case, data of 500,000 users from the well-known Netflix prize dataset for recommender development was de-anonymized by co-referencing them to publicly available Amazon reviews.

While a combination of a few columns could give too much away about specific individuals, for example, an address or postcode together with the age would be a give-away for anyone trying to trace data, ML models created on top of such datasets can be insecure as well. ML models can potentially leak sensitive information when hit by attacks such as membership inference attacks and model inversion attacks.

Membership attacks consist, roughly speaking, of recognizing differences in the target model's predictions on inputs that it was trained on compared to inputs that it wasn't trained on. You can find out more about membership attacks from the paper *Membership Inference Attacks against Machine Learning Models* (Reza Shokri and others, 2016). They showed that off-the-shelf models provided as a service by Google and others can be vulnerable to these attacks.



In **inversion attacks**, given API or black box access to a model and some demographic information, the samples used in the model training can be reconstructed. In a particularly impressive example, faces used for training facial recognition models were reconstructed. Of even greater concern, Matthew Fredrikson and others showed that models in personalized medicine can expose sensitive genomic information about individuals (*Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing*; 2014)

Differential privacy (DP) mechanisms can prevent model inversion and membership attacks. In the next sections, we'll be talking about DP and then about PATE.

Differential privacy

The concept of DP, first formulated by Cynthia Dwork and others in 2006 (*Calibrating Noise to Sensitivity in Private Data Analysis*), is the gold standard for privacy in ML. It centers around the influence of individual data points on the decisions of an algorithm. Roughly speaking, this implies, in turn, that any output from the model wouldn't give away whether an individual's information was included. In DP, data is perturbed with the noise of a certain distribution. This not only can lead to safety against privacy attacks, but also to less overfitting.

In order to properly explain DP, we need to introduce the notion of a neighboring database (think *dataset*), which is a database that only differs in a single row or, in other words, a single individual. Two datasets, $D, D' \in \mathcal{D}$, differ only in the fact that $i \in D$ and $i \notin D'$.

The key is then to set an upper bound to require nearly identical behavior of the mapping (or mechanism) $M : \mathcal{D} \rightarrow \mathcal{S}$ on neighboring databases:

$$\Pr[M(D) \in S] \leq e^\epsilon \Pr[M(D') \in S] + \delta, \text{ for } D', D \in \mathcal{D}.$$

This is called the epsilon-delta parametrized DP for an algorithm, M , on any neighboring databases $D, D' \in \mathcal{D}$, and any subsets S of outcomes \mathcal{S} .

In this formulation, the epsilon parameter is the multiplicative guarantee, and the delta parameter the additive guarantee of probabilistically almost exact outcomes. This means the DP cost that an individual incurs as a result of their data being used is minimal. Delta privacy can be seen as a subset or the special case, where epsilon is 0, and epsilon privacy as the case, where delta is 0.

These guarantees are achieved by masking small changes in the input data. For example, a simple routine for this masking was described by Stanley L. Warner in 1965 (*Randomized response: A survey technique for eliminating evasive answer bias*). Respondents in surveys answer sensitive questions such as *Have you had an abortion?* either truthfully or deterministically according to coin flips:

1. Flip a coin.
2. If tails, respond truthfully: no.
3. If heads, flip a second coin and respond *yes* if heads, or respond *no* if tails.

This gives plausible deniability.

Private aggregation of teacher ensembles

Based on the paper *Semi-supervised Knowledge Transfer for Deep Learning from Private Training Data*, by Nicolas Papernot and others (2017), the **Private Aggregation of Teacher Ensembles (PATE)** technique relies on the noisy aggregation of teachers. In 2018, Nicolas Papernot and others (*Scalable private learning with PATE*) refined the 2017 framework, improving both the accuracy and privacy of the combined model. They further demonstrated the applicability of the PATE framework to large-scale, real-world datasets.

The PATE training follows this procedure:

1. An ensemble of models (**teacher models**) is created based on different datasets with no training examples in common.
2. A student model is trained based on querying noisy aggregate decisions of the teacher models.
3. Only the student model can be published, not the teacher models.

As mentioned, each teacher is trained on disjointed subsets of the dataset. The intuition is that if teachers agree on how to classify a new input example, then the aggregate decision does not reveal information about any single training example.

The aggregate mechanism in queries includes **Gaussian NoisyMax (GNMax)**, an argmax with Gaussian noise, $\mathcal{N}(0, \sigma^2)$, defined as follows:

$$\mathcal{M}_\sigma(x) \triangleq \operatorname{argmax}_i \{n_i(x) + \mathcal{N}(0, \sigma^2)\},$$

This has a data point x , classes $1, \dots, m$, and the vote $f_j(x) \in [m]$ of teacher j on x .

$n_i(x)$ denotes the vote count for class i :

$$n_i(x) = \sum_j f_j(x) \equiv i.$$

Intuitively, accuracy decreases with the variance of the noise, so the variance has to be chosen tight enough to provide good performance, but wide enough for privacy.

The epsilon value depends on the aggregation, particularly the noise level, but also on the context of the dataset and its dimensions. Please see *How Much is Enough? Choosing ϵ for Differential Privacy* (2011), by Jaewoo Lee and Chris Clifton, for a discussion.

See also

A detailed overview of the concepts in DP can be found in *The Algorithmic Foundations of Differential Privacy*, by Cynthia Dwork and Aaron Roth. Please see the second PATE paper (Nicolas Papernot and others 2018; <https://arxiv.org/pdf/1802.08908.pdf>), the method of which we adopted for this recipe.

As for Python libraries concerning DP, a number are available:

- Opacus enables training PyTorch models with DP: <https://github.com/pytorch/opacus>.
- PySyft works with PyTorch, TensorFlow, and Keras, and includes many mechanisms, including PATE: <https://github.com/OpenMined/PySyft>.
- TensorFlow's cleverhans library provides tools for benchmarking model vulnerability to adversarial attacks: <https://github.com/tensorflow/cleverhans>.

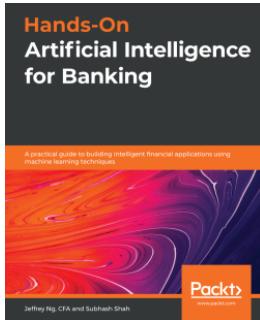
TensorFlow's privacy repository contains optimizers and models related to DP. It also contains tutorials using different mechanisms, such as a DP Stochastic Gradient Descent, DP Adam Optimizer, or PATE, on the adult, IMDB, or other datasets: <https://github.com/tensorflow/privacy>.

There are frameworks for both TensorFlow and PyTorch for encrypted ML:

- tf-encrypted relates to privacy-preserving ML and encryption in TensorFlow: tf-encrypted.io/.
- Facebook's CrypTen also relates to PyTorch, and includes the encryption of models and data: <https://github.com/facebookresearch/CrypTen>.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



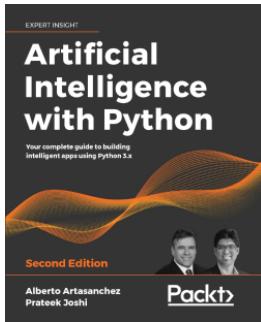
Hands-On Artificial Intelligence for Banking

Jeffrey Ng , Subhash Shah

ISBN: 978-1-78883-078-2

Learn how to clean your data and ready it for analysis

- Implement data preprocessing steps and optimize model hyperparameters
- Work with large amounts of data using distributed and parallel computing techniques
- Get to grips with representational learning from images using InfoGAN
- Delve into deep probabilistic modeling with a Bayesian network
- Create your own artwork using adversarial neural networks
- Understand a model's key performance characteristics and bring solutions to production as APIs
- Go from proof to production covering data loading and visualization as well as modeling and deployment as microservices



Artificial Intelligence with Python - Second Edition

Alberto Artasanchez , Prateek Joshi

ISBN: 978-1-83921-953-5

Learn how to clean your data and ready it for analysis

- Understand what artificial intelligence, machine learning, and data science are
- Explore the most common artificial intelligence use cases
- Learn how to build a machine learning pipeline
- Assimilate the basics of feature selection and feature engineering
- Identify the differences between supervised and unsupervised learning
- Discover the most recent advances and tools offered for AI development in the cloud
- Develop automatic speech recognition systems and chatbots
- Apply AI algorithms to time series data

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

A/B test 260
abductive reasoning 212
accuracy 297
ActionCores
 reference link 215
active learning 101
Adaptive Moment Estimation (ADAM) 92
adjacency matrix
 creating 168
adversarial attacks 421
adversarial t-shirts 421
adverse impact ratio (AIR) 107
agents
 about 239
 watching, in environment 276
AI effect 1
AI solutions
 monitoring 419
Akaike information criterion (AIC) 123
aleatoric uncertainty 196
algorithmic bias
 battling 102, 103, 104, 105, 106, 107, 108,
 109, 110, 111, 112, 113, 114
Allen AI Commonsense Knowledge Graphs
 reference link 215
Amazon Web Services (AWS) 1
AmpliGraph
 reference link 215
annoy library
 reference link 152
anomalies
 discovering 141, 142
outlier detection methods, implementing 142,
 143, 144, 145, 146, 148
working 149

ant colony optimization (ACO)
 about 237
 shortest bus route, finding 235, 236
Apache Superset
 reference link 413
application programming interface (API) 68
area under the curve (AUC) 65, 154, 180
artificial intelligence (AI)
 about 1, 2, 354
 Python, implementing 16, 17
auto-reloading packages 19
autoencoder
 about 149
 working 319, 320
Autoregression (AR) 124
Autoregressive Integrated Moving Average
 (ARIMA)
 about 124
 used, for analyzing CO2 time series 122, 123
Autoregressive Moving Average (ARMA) 124
Ax library
 URL 268

B

bag of words representation 109, 159
bag-of-characters approach
 implementing 154, 155
baseline
 implementing, with string comparison functions
 153, 154
Bayesian regression
 reference link 204
BentoML
 reference link 420
Bernoulli distribution 197
BetaGeo (BD) model 187, 189
Bilingual Evaluation Understudy (BLEU) score 392

binary decision diagrams (BDD) 230
blackjack
 playing 278, 280, 281, 282, 283, 284, 285, 286
Boolean satisfiability problem 216
bottleneck 149
branching programs 230

C

Captum
 URL 95
cartpole
 controlling 268, 269, 271, 272, 273, 274, 275
Census Income dataset 51
chess engine
 writing, with Monte Carlo tree search 250
chess
 playing 254, 255
chromosomes 227
class weighting 101
clothing items
 recognizing 288, 290, 291, 297
 recognizing, with DoG 291, 292
 recognizing, with LeNet5 294, 295
 recognizing, with MLP 292, 293
 recognizing, with MobileNet transfer learning 296
clustering methods
 overview 140
CO2 time series
 analyzing, with ARIMA and SARIMA 122, 123
 forecasting 118, 119, 120, 121, 123, 124, 125
 forecasting, with Prophet 126, 127
code execution
 timing 21, 22
code
 compiling 23
 parallelizing 26, 27
coin-flip distribution 197
collaborative filtering 165
column-separated value (CSV) file 414
Commonsense Reasoning Problem Page at NYU CS
 reference link 215
community detection algorithms 169
Conceptnet5

 reference link 216
conda 11
confidence calibration
 reference link 185
content-based filtering 165
contextual bandit 267
continuous bag-of-words algorithm (CBOW) 214, 367, 368

Convolutional Neural Network (ConvNet) 290
Correctional Offender Management Profiling for Alternative Sanctions (COMPAS) 102

cost function 34
CovidSim microsimulation model
 reference link 250

credit defaults
 epistemic uncertainty 203
 stopping 198, 199, 201, 203
 working 203

credit scoring process 198

cross-entropy 49

custom word embeddings 366, 367

customer lifetime value
 BG/NBD model 189
 datasets, using 186, 187
 Gamma-Gamma model 189
 working 188

customer values
 live decisioning 96, 97, 99, 100
Cython
 URL 23

D

data generators 67, 68
data
 transforming, in scikit-learn 71, 72, 73, 80, 81
 visualizing, in seaborn 31, 32
debugging 20, 21
decisions
 making, based on knowledge 205
deductive reasoning 212
Deep Convolutional Generative Adversarial Network (DCGAN) 301
Deep Convolutional Networks
 with Guided Attention 350, 351
deep fakes 336, 337

deep fakes detection

reference link 337

deep learning models

building 369, 371

Deep Q network (DQN) 259

Difference of Gaussians (DoG)

about 290

clothing items, recognizing with 291, 292

working 298

differential privacy (DP) 430, 431

DiscoGAN 337

distributional hypothesis 214

dummy-transform 133

dynamic range 346

E

easyAI library

reference link 258

edit distance 152

elbow criterion 134

eli5 documentation

reference link 69

ELIZA

reference link 372

working 377, 378

Elyra

reference link 421

epistemic uncertainty 203

epsilon-greedy action selection 281

equal false positive rates 105

equal odds 105

expert system 212

exploratory data analysis (EDA) 51

Extract, Transform, Load (ETL) 25

eXtreme MultiLabel (XML) 372

Eywa

about 373

functionalities 378

working 379

F

Facebook's PlanOut

reference link 268

faceit live repository

reference link 337

faceswap, alignment file

reference link 334

fairseq

reference link 395

false alarm rate 65

Fashion-MNIST 288

FastAPI

reference link 421

fitters 190

Flask

reference link 421

fraudster communities

adjacency matrix, creating 168

community detection algorithms 169

detecting 167, 168

evaluating 170, 171, 173, 174

graph community algorithms 172

implementing 168

information entropy 173

working 172

G

Gamma-Gamma model 189

Gaussian NoisyMax (GNMax) 432

Gaussian process regression (GPR) 119

Generative Adversarial Networks (GANs) 149, 400

genetic algorithm

implementing, for n-queens problem 217, 218, 219, 221

working 227, 228

Girvan-Newman algorithm 173

Google Colab

used, for installing Python libraries 10

graph classification

reference link 175

graph community algorithms

about 172

Girvan-Newman algorithm 173

Louvain algorithm 172

graph convolution networks

reference link 175

graph embedding

with Walklets 213

graphics processing units (GPUs) 1

GraphVite
reference link 215
guided attention loss 351
Guided Attention
Deep Convolutional Networks, using with 350,
351

H

hidden layer 47
higher-order features
deriving 76, 77, 78
hit rate 35, 65
Hoeffding Tree 101
hold-out set 33
house prices
predicting, in PyTorch 82, 84, 85, 86, 87, 89,
91, 92, 93, 94, 95
Hugging Face transformer library
reference link 372
hybrid recommender 165
hyperparameters 33

I

identity function 37
image browser, Udacity self-driving car dataset
reference link 412
ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) 298
images
encoding 309, 310, 311, 312, 313, 314, 316,
317, 318
generating 301, 302, 303, 305, 306, 307, 308
inductive reasoning 212
industrial process monitoring (IPM) 141
information entropy 173
integer encoding 38
inversion attacks 430
iPython-autotime library
reference link 21
isolation forest 149
isotonic regression 184

J

Julia, Python, R (Jupyter)
about 9

outputs 19
Jupyter commands and outputs
history, executing 19
history, obtaining 18
Jupyter environment
options 14
Python libraries, installing 9
reference link 16
setting up 9
working 14
Jupyter Notebook environment
self-hosting 11, 12, 13
just-in-time (JIT) 18

K

K-arm bandit problem 267
k-means algorithm
about 138, 140
reference link 138
k-Nearest Neighbors (kNN) 149
KagNet
reference link 215
Kanren logic programming
reference link 215
KarateClub
reference link 215
Keras
classifying 29, 30
data loading 54, 55, 56, 57, 58, 59, 60, 61
data preprocessing 54, 55, 56, 57, 58, 59, 60,
61
implementing 53
model training 62, 63, 64, 65, 66
modeling 35, 36, 37, 38, 39, 41, 42
modeling with 51, 52, 53
reference link 50
trained word embeddings 369, 371
working 45, 66
kernels 299
key performance indicators (KPIs) 141
knowledge base 212
knowledge embedding (KE) 207, 208, 209, 210,
211, 213
knowledge graph (KG) 213
knowledge-based system 205

L

label encoding 38
LeNet5
 clothing items, recognizing with 294, 295
 working 298, 299
Levenshtein distance 152
libraries and frameworks, for creating chatbots
 reference links 379
libraries, for dealing with word embeddings
 reference links 371
library 9
library functionality, for item ranking
 reference link 167
lightfm model 166
live decisioning
 model, serving for 413, 414, 416, 417
live TensorFlow session, for interactive face-GAN
 explorer
 reference link 412
Local Outlier Factors (LOF) 149
logic provers 213
logical reasoning 207, 212
Long Short-Term Memory networks (LSTMs) 149,
 359, 392
loss function 34
Louvain algorithm 172

M

MAgent
 reference link 250
market segments
 clustering 130, 131
 dataset, visualizing 131, 133, 134, 135, 136
 working 136, 137
matrix decomposition 165
matrix factorization 165
maximal information coefficient (MIC) 58, 67
Mean Average Error (MAE) 417
mean decrease accuracy (MDA) 68
Mean Squared Error (MSE) 48, 89, 113
melodies
 generating 354, 355, 356, 357, 358, 359, 360
membership attacks 430
metrics 197

microservice 418
microservice architecture 418
Microsoft Cognitive Toolkit (CNTK) 35
Microsoft's PDP solver
 reference link 232
MLFlow
 URL 420
MobileNet transfer learning
 clothing items, recognizing with 296
 working 299, 300
model results
 visualizing 403
model
 productionizing 418, 419
 securing, against attack 421, 423, 424, 425,
 427, 428
 serving, for live decisioning 413, 414, 416, 417
module 9
Monte Carlo tree search (MCTS)
 about 256
 chess engine, writing 250
Moving Average (MA) 124
multi-agent system (MAS) 239
multi-agent-based modeling, in Python
 reference link 250
multi-armed bandit (MABP) 267
multilayer perceptron (MLP)
 about 47, 290
 clothing items, recognizing with 292, 293

N

n-queens problem
 solving 216
Naive Bayes 184
Named-Entity Recognition (NER) 393
NapkinML
 reference link 138
National Health Service (NHS) 239
natural language processing (NLP) 213
natural language toolkit (nltk) 206
nbdev
 reference link 28
Negative Binomial Distribution (NBD) model 187,
 189
negative log-likelihood 197

neural fitted Q-learning (NFQ) 286
Neural Machine Translation (NMT) 393
neural network
 training 45, 47, 48, 49
neural story generation 400
neural text generation 400
newsgroups
 about 362, 363
 bag-of-words representation 364
 custom word embeddings 366, 367
 word embeddings 364, 365
 working 367
node
 using, for chess implementation 253, 254
non-public scikit-learn API
 using 114, 116, 117
nondeterministic polynomial time 216
norming group 107

O

object detection, libraries
 reference links 329
objects
 localizing 323, 324, 325, 326, 327, 328
one-hot encoding 38
Open Computer Vision Library (OpenCV) 323
Open Multilingual Wordnet
 reference link 216
OpenAI Gym 268
OpenML
 URL 82
OpenNMT
 reference link 395
ordinal coding 32
osBrain
 reference link 250
outlier detection
 reference link 150

P

p-value 171
package 9
PADE
 reference link 250
pandas DataFrames

speeding up 25, 26
particle swarm optimization
 about 221
 working 229, 230
perceptron 46
Perl Compatible Regular Expressions (PCRE) 379
permutation importance 68
platt scaling 183
popular novel
 writing 395, 398, 399, 400
POSIX regular expressions 379
precision at k 165
Predictive Model Markup Language (PMML) 418
pressure-time plot 341
Principal Component Analysis (PCA) 136
privacy attack 421
Private Aggregation of Teacher Ensembles (PATE)
 431
probabilistic analyses
 reference link 186
progress bars
 displaying 22, 23
proof of concept (POC) 413
Prophet
 URL 127
 used, for forecasting CO2 time series 126, 127
PSO algorithm
 implementing, for n-queens problem 221, 223,
 224
PyDatalog
 reference link 215
pydub
 reference link 346
pykg2vec
 reference link 215
pyRDF2Vec
 reference link 215
Python bindings, to picosat
 reference link 232
Python Debugger (pdb) 21
Python environment
 setting up 10
Python libraries, concerning DP
 reference link 432
Python libraries

installing 9
installing, with Google Colab 10
reference link 11
Python Outlier Detection (pyOD)
about 141
reference link 141
python-Levenshtein library
reference link 152
Python
auto-reloading packages 19
code execution, timing 21, 22
code, compiling 23, 24
code, parallelizing 26, 27
debugging 20, 21
implementing, for AI 16, 17
Jupyter commands and outputs, history obtaining 18
pandas DataFrames, speeding up 25, 26
progress bars, displaying 22, 23

PyTorch BigGraph
reference link 215

PyTorch
classifying 29, 30
house prices, predicting in 82, 84, 85, 86, 87, 89, 91, 92, 93, 94, 95
modeling 42, 44, 45
reference link 50
working 45

Q

Question Answering (QA) 393

R

random decision forest 33
random forest model 33
ranges
encoding, numerically 73, 74, 75
read-eval-print loop (REPL) 9
Receiver Operating Characteristic (ROC) 146
recommendation system
building 160, 161, 162
model, implementing 162, 163, 164
working 164
rectified linear unit activation (ReLU) 89
Recurrent Neural Network (RNN) 359

RedisAI
reference link 421
ReenactGAN 337
regular expression (regex) 379
Relative Risk Ratio (RRR) 107
RLLib library
using 277
Root Mean Square Propogation (RMSProp) 92

S

SAT solver
implementing, for n-queens problem 225, 227
reference link 232
working 231
scaled exponential linear unit (SELU) 49
scikit-learn, feature selection
reference link 82
scikit-learn
classifying 29, 30
data, transforming in 71, 72, 73, 80, 81
modeling 33, 34, 35
reference link 50, 102
working 45
scikit-opt 238
Seasonal Autoregressive Integrated Moving Average (SARIMA)
about 124
used, for analyzing CO2 time series 122, 123
SELU activation function 37, 49
shortest bus route
finding 232, 233
Siamese neural network 151
Siamese neural network approach
implementing 155, 157, 158
SimPy
reference link 250
simulated annealing
about 237
shortest bus route 233
shortest bus route, finding 235
Single Shot MultiBox Detector (SSD) 327
singular value decomposition (SVD) 166
sklearn 33
softmax activation 49
softmax function 38

Spearman rank correlation 264
specialized satisfiability (SAT) 230
spectrogram–strategy (SpecGAN) 351
speech synthesizer 350
speech
 synthesizing, from text 347, 348, 349, 350
spread of disease
 simulating 239, 240, 242, 244, 247, 248
standardization sample 107
statistical significance 171
Statsmodels
 URL 127
Stochastic Gradient Descent (SGD) 92
stock prices
 featurization 183
 isotonic regression 184
 Naive Bayes 184
 platt scaling 183
 predicting 177, 178, 179, 180, 181
 prediction, working 182
streamlit hello-world 403
streamlit
 URL 403
string comparison functions
 used, for implementing baseline 153, 154
string featurization model
 parameters 156
strings
 bag-of-characters approach, implementing 154, 155
 baseline, implementing with string comparison functions 153, 154
 implementing 152
 Siamese neural network approach, implementing 155, 157, 158
 similarity search, representing 150, 151, 152
 working 159, 160
Strongly Connected Components (SCC) 169
style transfer 318
style
 encoding 309, 310, 311, 312, 313, 314, 316, 317, 318
support vector machines (SVMs) 47
SymPy
 reference link 215

T
taxonomy 206
teacher models 431
tensor processing units (TPUs) 1
TensorBoard
 reference link 42
TensorFlow Probability
 aleatoric uncertainty 196
 Bernoulli distribution 197
 implementing 190, 191, 192, 193, 194, 195, 196
 metrics 197
 negative log-likelihood 197
 reference link 198
 working 196
TensorFlow Serving
 reference link 36
TensorFlow, tutorials
 reference link 395
TensorFlow
 reference link 69
term frequency-inverse document frequency (TFIDF) 364, 369
text-to-speech (TTS) 350
text
 speech, synthesizing from 347, 348, 349, 350
 translating, from English to German 380, 381, 382, 383, 384, 385, 386, 387, 388, 389, 390, 391, 392, 393
textgenrnn
 reference link 401
Textual Entailment (TE) 393
TFX
 reference link 421
TorchServe
 reference link 421
trained word embeddings
 in Keras 369, 371
transformations
 combining 78, 79
traveling salesman problem (TSP)
 about 236
 reference link 238
tree search 251, 252
true positive rate 35

twin neural network 151

U

Upper Confidence Bound version 1 (UCB1) 267
Upper Confidence Trees (UCTs)
 about 257
 reference link 258
Upper-Confidence-Bound (UCB) 258
users
 chatting to 372, 374, 375, 376, 377

V

validation set 33
Very Fast Decision Tree (VFDT) 101
videos
 faking 330, 331, 332, 333, 334, 335, 336
visualization solution, for machine learning (ML)
 model
 building 403
 data app, creating 404, 406, 408, 409, 410
 working 411
voice commands
 recognizing 339, 340, 341, 342, 343, 344, 346

W

Walklets
 graph embedding 213
wav2letter++
 reference link 346

waveform–strategy (WaveGAN) 351
WaveGAN model
 about 351, 352
 used, to synthesize speech from text 352, 353
web camera
 interacting with 328, 329
website
 optimizing 260, 261, 262, 263, 265, 266, 267,
 268
Wikidata
 reference link 216
word embeddings
 about 364, 365, 369, 371
 reference links 371
WordNet 213

X

XML, notebook tutorial
 reference link 372

Y

Yago
 reference link 216
YOLOv4 paper
 reference link 329
You Only Look Once (YOLO) 327

Z

Z3
 reference link 232