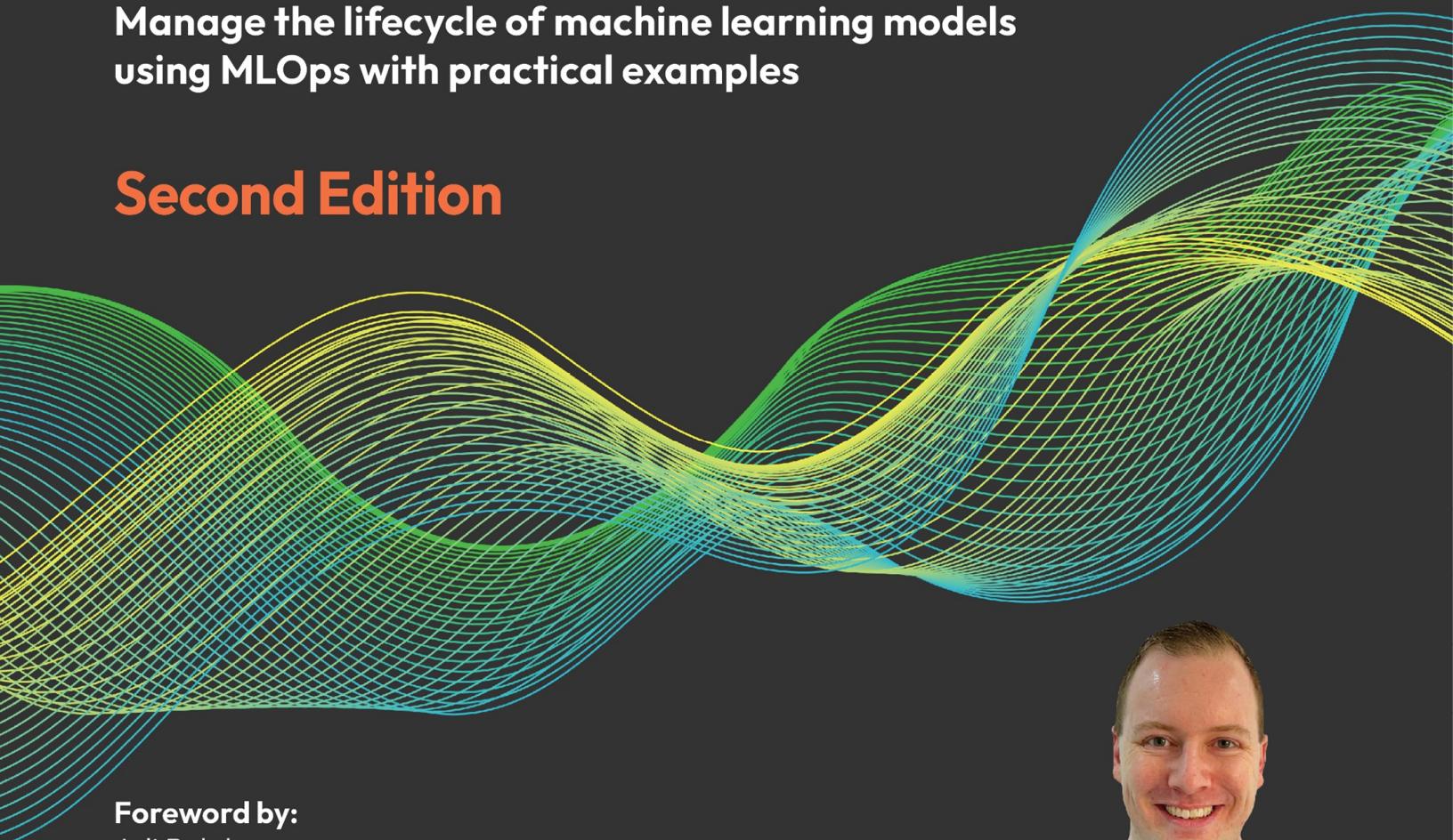


EXPERT INSIGHT

Machine Learning Engineering with Python

Manage the lifecycle of machine learning models
using MLOps with practical examples

Second Edition



Foreword by:

Adi Polak

Author of Scaling Machine Learning with Spark



Andrew P. McMahon

<packt>

Machine Learning Engineering with Python

Second Edition

Manage the lifecycle of machine learning models using MLOps with practical examples

Andrew P. McMahon



BIRMINGHAM—MUMBAI

Machine Learning Engineering with Python

Second Edition

Copyright © 2023 Packt Publishing

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

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

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

Publishing Product Manager: Bhavesh Amin

Acquisition Editor – Peer Reviews: Gaurav Gavas

Project Editor: Amisha Vathare

Content Development Editor: Elliot Dallow

Copy Editor: Safis Editing

Technical Editor: Anjitha Murali

Proofreader: Safis Editing

Indexer: Subalakshmi Govindhan

Presentation Designer: Rajesh Shirasath

Developer Relations Marketing Executive: Monika Sangwan

First published: November 2021

Second edition: August 2023

Production reference: 2280823

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-83763-196-4

www.packtpub.com

Foreword

Throughout my career in Fortune 500 companies and startups, I have witnessed the various cycles of machine learning momentum firsthand. With each cycle, our economy has broadened, opportunities evolved, and the world moved forward with innovations. This is machine learning; welcome! This world is an ever-evolving field that has witnessed unprecedented growth over the past decade. As artificial intelligence has become increasingly integrated into our lives, it is essential to grasp the principles of building robust, scalable, and efficient machine learning systems.

It is spectacular to see how the world has changed since I released my book, *Scaling Machine Learning with Spark: Distributed ML with MLLib, TensorFlow, and PyTorch* (O'Reilly, March 2023).

As an industry, we have noticed a massive adoption of OpenAI services, evolving GPT models, and industry openness, this is a phenomenal time to dive deeper, take notes, and learn from hands-on practitioners.

Machine Learning Engineering with Python is your gateway to mastering the art of turning machine learning models into real-world applications, including all the bits and pieces of building pipelines with Airflow, data processing with Spark, LLMs, CI/CD for machine learning, and working with AWS services.

Andy breaks the space into tangible, easy-to-read, and compelling chapters, each focusing on specific use cases and dedicated technologies to pass his

wisdom to you, including how to approach projects as an ML engineer working with deep learning, large-scale serving and training, and LLMs.

This fantastic resource bridges the gap between theory and practice, offering a hands-on, Python-focused approach to ML engineering. Whether you are an aspiring data scientist, a seasoned machine learning practitioner, or a software engineer venturing into the realm of AI for the first time, this book equips you with the knowledge and tools to thrive in the ML engineering domain.

What I love about this book is that it is very practical. It covers the space well, from complete beginners to hands-on. You can use it to get a bigger picture of the space and then dive deep with code samples and real-world projects.

Another distinguishing aspect of this book is its focus on Python—a popular language in data science and ML engineering. Python's versatility, ease of use, and vast ecosystem of libraries makes it the ideal choice for turning machine learning models into tangible solutions. Throughout the book, Andy leverages Python's powerful libraries like NumPy, pandas, scikit-learn, TensorFlow, and an array of open-source solutions.

Whether you are an individual learner seeking to enhance your skills or an organization striving to harness the potential of machine learning, *Machine Learning Engineering with Python* is your ultimate companion.

Adi Polak

Author of Scaling Machine Learning with Spark

Contributors

About the author

Andrew Peter McMahon has spent years building high-impact ML products across a variety of industries. He is currently Head of MLOps for NatWest Group in the UK and has a PhD in theoretical condensed matter physics from Imperial College London. He is an active blogger, speaker, podcast guest, and leading voice in the MLOps community. He is co-host of the *AI Right* podcast and was named ‘Rising Star of the Year’ at the 2022 British Data Awards and ‘Data Scientist of the Year’ by the Data Science Foundation in 2019.

It takes a village to do anything in this life, and I wish to thank my Mum, Dad, and brothers, David and Paul, wider family and friends for being that village for me. But most of all I want to thank my wife Hayley for being my rock, my best friend, and the only person I could go through this crazy life with; and my sons, Teddy and Alfie, for being all the inspiration I’ll ever need. This book is dedicated to you three.

About the reviewers

Hamza Tahir is a software developer turned ML engineer. An indie hacker by heart, he loves ideating, implementing, and launching data-driven products. He has previously worked on creating machine learning based

products like *PicHance*, *Scrilys*, *BudgetML*, and *you-tldr*. Based on his learnings from deploying ML in production for predictive maintenance use-cases in his previous startup, he co-created *ZenML*, an open-source MLOps framework for easily creating production grade ML pipelines on any infrastructure stack.

I'd like to give my appreciation to all my ZenML colleagues, who are working hard every day to make the most accessible MLOps product on the market. I'd also like to thank Isabel, for her consistent and unwavering support.

Prince Canuma is an accomplished ML engineer with an extensive background in MLOps, ML, data science, computer vision, and NLP. Born in Maputo, Mozambique, Prince embarked on his journey into the realm of ML after earning a diploma in IT and telecommunications from the **Institute of Transports and Communications (ITC)** in 2017. At present, Prince is an integral part of *Neptune.ai*. In his tenure, he has excelled as an ML engineer and carved a niche for himself in developer relations.

Beyond his professional commitments, Prince's expertise shines brightly in his significant research contributions, especially in the realm of **Automatic Speech Recognition (ASR)**. His most groundbreaking work revolved around benchmarking and enhancing OpenAI's Whisper ASR system by introducing speaker diarization and a targeted language model.

On a personal note, I extend my deepest gratitude to Eliseu Canuma and the entire Canuma family for their unwavering support and encouragement throughout the process of producing this book. Your faith in my capabilities has been the cornerstone of my involvement in this project.

Jonah Turner is a student at Toulouse 3 Paul Sabatier University pursuing a master's in artificial intelligence. He was previously a data scientist at Sandhills Global working in computer vision, NLP, and DataOps. He enjoys traveling in southern France, browsing GitHub, and contributing to open-source machine learning projects.

Join our community on Discord

Join our community's Discord space for discussion with the author and other readers:

<https://packt.link/mle>



Contents

Preface

Who this book is for

What this book covers

To get the most out of this book

Get in touch

1. Introduction to ML Engineering

Technical requirements

Defining a taxonomy of data disciplines

Data scientist

ML engineer

ML operations engineer

Data engineer

Working as an effective team

ML engineering in the real world

What does an ML solution look like?

Why Python?

High-level ML system design

Example 1: Batch anomaly detection service

Example 2: Forecasting API

Example 3: Classification pipeline

Summary

2. The Machine Learning Development Process

Technical requirements

Setting up our tools

Setting up an AWS account

Concept to solution in four steps

Comparing this to CRISP-DM

Discover

Using user stories

Play

Develop

Selecting a software development methodology

Package management (conda and pip)

Poetry

Code version control

Git strategies

Model version control

Deploy

Knowing your deployment options

Understanding DevOps and MLOps

Building our first CI/CD example with GitHub Actions

Continuous model performance testing

Continuous model training

Summary

3. From Model to Model Factory

Technical requirements

Defining the model factory

Learning about learning

Defining the target

Cutting your losses

Preparing the data

Engineering features for machine learning

Engineering categorical features
Engineering numerical features

Designing your training system

Training system design options

Train-run

Train-persist

Retraining required

Detecting data drift

Detecting concept drift

Setting the limits

Diagnosing the drift

Remediating the drift

Other tools for monitoring

Automating training

Hierarchies of automation

Optimizing hyperparameters

Hyperopt

Optuna

AutoML

auto-sklearn

AutoKeras

Persisting your models

Building the model factory with pipelines

Scikit-learn pipelines

Spark ML pipelines

Summary

4. Packaging Up

Technical requirements

Writing good Python

Recapping the basics
Tips and tricks
Adhering to standards
Writing good PySpark

Choosing a style

Object-oriented programming
Functional programming

Packaging your code

Why package?
Selecting use cases for packaging
Designing your package

Building your package

Managing your environment with Makefiles
Getting all poetic with Poetry

Testing, logging, securing, and error handling

Testing
Securing your solutions
Analyzing your own code for security issues
Analyzing dependencies for security issues
Logging
Error handling

Not reinventing the wheel
Summary

5. Deployment Patterns and Tools

Technical requirements
Architecting systems

Building with principles

Exploring some standard ML patterns

Swimming in data lakes

Microservices

Event-based designs

Batching

Containerizing

Hosting your own microservice on AWS

Pushing to ECR

Hosting on ECS

Building general pipelines with Airflow

Airflow

Airflow on AWS

Revisiting CI/CD for Airflow

Building advanced ML pipelines

Finding your ZenML

Going with the Kubeflow

Selecting your deployment strategy

Summary

6. Scaling Up

Technical requirements

Scaling with Spark

Spark tips and tricks

Spark on the cloud

AWS EMR example

Spinning up serverless infrastructure

Containerizing at scale with Kubernetes

Scaling with Ray

Getting started with Ray for ML
Scaling your compute for Ray
Scaling your serving layer with Ray
Designing systems at scale
Summary

7. Deep Learning, Generative AI, and LLMOps

Going deep with deep learning
Getting started with PyTorch
Scaling and taking deep learning into production
Fine-tuning and transfer learning
Living it large with LLMs
Understanding LLMs
Consuming LLMs via API
Coding with LLMs
Building the future with LLMOps
Validating LLMs
PromptOps
Summary

8. Building an Example ML Microservice

Technical requirements
Understanding the forecasting problem
Designing our forecasting service
Selecting the tools
Training at scale
Serving the models with FastAPI
Response and request schemas
Managing models in your microservice
Pulling it all together

Containerizing and deploying to Kubernetes

Containerizing the application

Scaling up with Kubernetes

Deployment strategies

Summary

9. Building an Extract, Transform, Machine Learning Use Case

Technical requirements

Understanding the batch processing problem

Designing an ETML solution

Selecting the tools

Interfaces and storage

Scaling of models

Scheduling of ETML pipelines

Executing the build

Building an ETML pipeline with advanced Airflow features

Summary

Other Books You May Enjoy

Index

Preface



“Software is eating the world, but AI is going to eat software.”

— Jensen Huang, CEO of Nvidia

Machine learning (ML), part of the wider field of **Artificial Intelligence (AI)**, is rightfully recognized as one of the most powerful tools available for organizations to extract value from their data. As the capabilities of ML algorithms have grown over the years, it has become increasingly obvious that implementing such algorithms in a scalable, fault-tolerant, and automated way requires the creation of new disciplines. These disciplines, **ML Engineering (MLE)** and **ML Operations (MLOps)**, are the focus of this book.

The book covers a wide variety of topics in order to help you understand the tools, techniques, and processes you can apply to engineer your ML solutions, with an emphasis on introducing the key concepts so that you can build on them in your future work. The aim is to develop fundamentals and a broad understanding that will stand the test of time, rather than just provide a series of introductions to the latest tools, although we do cover a lot of the latest tools!

All of the code examples are given in Python, the most popular programming language in the world (at the time of writing) and the *lingua franca* for data applications. Python is a high-level and object-oriented language with a rich ecosystem of tools focused on data science and ML. For example, packages such as *scikit-learn* and *pandas* have become part of the standard lexicon for data science teams across the world. The central tenet of this book is that knowing how to use packages like these is not enough. In this book, we will use these tools, and many, many more, but focus on how to wrap them up in production-grade pipelines and deploy them using appropriate cloud and open-source tools.

We will cover everything from how to organize your ML team, to software development methodologies and best practices, to automating model building through to packaging your ML code, how to deploy your ML pipelines to a variety of different targets and then on to how to scale your workloads for large batch runs. We will also discuss, in an entirely new chapter for this second edition, the exciting world of applying ML engineering and MLOps to deep learning and generative AI, including how to start building solutions using **Large Language Models (LLMs)** and the new field of **LLM Operations (LLMOps)**.

The second edition of *Machine Learning Engineering with Python* goes into a lot more depth than the first edition in almost every chapter, with updated examples and more discussion of core concepts. There is also a far wider selection of tools covered and a lot more focus on open-source tools and development. The ethos of focusing on core concepts remains, but it is my hope that this wider view means that the second edition will be an excellent resource for those looking to gain practical knowledge of machine learning engineering.

Although a far greater emphasis has been placed on using open-source tooling, many examples will also leverage services and solutions from Amazon Web Services (AWS). I believe that the accompanying explanations and discussions will, however, mean that you can still apply everything you learn here to any cloud provider or even in an on-premises setting.

Machine Learning Engineering with Python, Second Edition will help you to navigate the challenges of taking ML to production and give you the confidence to start applying MLOps in your projects. I hope you enjoy it!

Who this book is for

This book is for machine learning engineers, data scientists, and software developers who want to build robust software solutions with ML components. It is also relevant to anyone who manages or wants to understand the production life cycle of these systems. The book assumes intermediate-level knowledge of Python and some basic exposure to the concepts of machine learning. Some basic knowledge of AWS and the use of Unix tools like bash or zsh will also be beneficial.

What this book covers

Chapter 1, Introduction to ML Engineering, explains the core concepts of machine learning engineering and machine learning operations. Roles within ML teams are discussed in detail and the challenges of ML engineering and MLOps are laid out.

Chapter 2, The Machine Learning Development Process, explores how to organize and successfully execute an ML engineering project. This includes a discussion of development methodologies like Agile, Scrum and CRISP-DM, before sharing a project methodology developed by the author that is referenced to throughout the book. This chapter also introduces **continuous integration/continuous deployment (CI/CD)** and developer tools.

Chapter 3, From Model to Model Factory, shows how to standardize, systematize and automate the process of training and deploying machine learning models. This is done using the author's concept of the model factory, a methodology for repeatable model creation and validation. This chapter also discusses key theoretical concepts important for understanding machine learning models and covers different types of drift detection and model retrain triggering criteria.

Chapter 4, Packaging Up, discusses best practices for coding in Python and how this relates to building your own packages, libraries and components for reuse in multiple projects. This chapter covers fundamental Python programming concepts before moving onto more advanced concepts, and then discusses package and environment management, testing, logging and error handling and security.

Chapter 5, Deployment Patterns and Tools, teaches you some of the standard ways you can design your ML system and then get it into production. This chapter focusses on architectures, system design and deployment patterns first before moving onto using more advanced tools to deploy microservices, including containerization and AWS Lambda. The popular ZenML and Kubeflow pipelining and deployment platforms are then reviewed in detail with examples.

Chapter 6, Scaling Up, is all about developing with large datasets in mind. For this, the Apache Spark and Ray frameworks are discussed in detail with worked examples. The focus for this chapter is on scaling up batch workloads where massive compute is required.

Chapter 7, Deep Learning, Generative AI and LLMOps, covers the latest concepts and techniques for training and deploying deep learning models for production use cases. This chapter includes material discussing the new wave of generative models, with a particular focus on **Large Language Models (LLMs)** and the challenges for ML engineers looking to productionize these models. This leads us onto define the core elements of LLM Operations (LLMOPs).

Chapter 8, Building an Example ML Microservice, walks through the building of a machine learning microservice that serves a forecasting solution using FastAPI, Docker and Kubernetes. This pulls together many of the previous concepts developed throughout the book.

Chapter 9, Building an Extract, Transform, Machine Learning Use Case, builds out an example of a batch processing ML system that leverages standard ML algorithms and augments these with the use of LLMs. This shows a concrete application of LLMs and LLMOPs, as well as providing a more advanced discussion of Airflow DAGs.

To get the most out of this book

- In this book, some previous exposure to Python development is assumed. Many introductory concepts are covered for completeness but in general it will be easier to get through the examples if you have already written at least some Python programs before. The book also assumes some exposure to the main concepts from machine learning, such as what a model is, what training and inference refer to and an understanding of similar concepts. Several of these are recapped in the text but again it will be a smoother ride if you have previously been acquainted with the main ideas behind building a machine learning model, even at a rudimentary level.
- On the technical side, to get the most out of the examples in the book, you will need access to a computer or server where you have privileges to install and run Python and other software packages and applications. For many of the examples, access to a UNIX type terminal, such as bash or zsh, is assumed. The examples in this book were written and tested on both a Linux machine running Ubuntu LTS and an M2 Macbook Pro running macOS. If you use a different setup, for example Windows, the examples may require some translation in order to work for your system. Note that the use of the M2 Macbook Pro means several examples show some additional information to get the examples working on Apple Silicon devices. These sections can comfortably be skipped if your system does not require this extra setup.
- Many of the Cloud based examples leverage **Amazon Web Services (AWS)** and so require an AWS account with billing setup. Most of the

examples will use the free-tier services available from AWS but this is not always possible. Caution is advised in order to avoid large bills. If in doubt, it is recommended you consult the AWS documentation for more information. As a concrete example of this, In *Chapter 5, Deployment Patterns and Tools*, we use the **Managed Workflows with Apache Spark (MWAA)** service from AWS. There is no free tier option for MWAA so as soon as you spin up the example, you will be charged for the environment and any instances. Ensure you are happy to do this before proceeding and I recommend tearing down your MWAA instances when finished.

- **Conda** and **Pip** are used for package and environment management throughout this book, but Poetry is also used in many cases. To facilitate easy reproduction of development environments for each chapters in the book's GitHub repository, (<https://github.com/PacktPublishing/Machine-Learning-Engineering-with-Python-Second-Edition>), each chapter of the book has a corresponding folder and within that folder are `requirements.txt` and Conda `environment.yml` files, as well as helpful `README` files. The commands for replicating the environments and any other requirements are given at the beginning of each chapter within the book.
- If you are using the digital version of this book, I still advise you to type the code yourself or access the code from the book's GitHub repository (<https://github.com/PacktPublishing/Machine-Learning-Engineering-with-Python-Second-Edition>)

[Edition](#)). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

As mentioned above, the code bundle for the book is hosted on GitHub at <https://github.com/PacktPublishing/Machine-Learning-Engineering-with-Python-Second-Edition>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

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

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. For example: “First, we must import the `TabularDrift` detector from the `alibi-detect` package, as well as the relevant packages for loading and splitting the data.”

A block of code is set as follows:

```
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_
import alibi
from alibi_detect.cd import TabularDrift
```

Any command-line input or output is written as follows and are indicated as command-line commands in the main body of the text:

```
pip install tensorflow-macos
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: “Select the **Deploy** button. This will provide a dropdown where you can select **Create** service.”



References to additional resources or background information appear like this.



Helpful tips and important caveats appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com and mention the book’s title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@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 reported this to us. Please visit <http://www.packtpub.com/submit-errata>, click **Submit Errata**, and fill in the form.

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@packtpub.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 <http://authors.packtpub.com>.

Share your thoughts

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

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

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

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

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

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

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781837631964>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

1

Introduction to ML Engineering

Welcome to the second edition of *Machine Learning Engineering with Python*, a book that aims to introduce you to the exciting world of making **machine learning (ML)** systems production-ready.

In the two years since the first edition of this book was released, the world of ML has moved on substantially. There are now far more powerful modeling techniques available, more complex technology stacks, and a whole host of new frameworks and paradigms to keep up to date with. To help extract the signal from the noise, the second edition of this book covers a far larger variety of topics in more depth than the first edition, while still focusing on the critical tools and techniques you will need to build up your ML engineering expertise. This edition will cover the same core topics such as how to manage your ML projects, how to create your own high-quality Python ML packages, and how to build and deploy reusable training and monitoring pipelines, while adding discussion around more modern tooling. It will also showcase and dissect different deployment architectures in more depth and discuss more ways to scale your applications using AWS and cloud-agnostic tooling. This will all be done using a variety of the most popular and latest open-source packages and frameworks, from classics like **Scikit-Learn** and **Apache Spark** to **Kubeflow**, **Ray**, and **ZenML**.

Excitingly, this edition also has new sections dedicated entirely to Transformers and **Large Language Models (LLMs)** like ChatGPT and GPT-4, including examples using Hugging Face and OpenAI APIs to fine-tune and build pipelines using these extraordinary new models. As in the first edition, the focus is on equipping you with the solid foundation you need to go far deeper into each of these components of ML engineering. The aim is that by the end of this book, you will be able to confidently build, scale, and deploy production-grade ML systems in Python using these latest tools and concepts.

You will get a lot from this book even if you do not run the technical examples, or even if you try to apply the main points in other programming languages or with different tools. As already mentioned, the aim is to create a solid conceptual foundation you can build on. In covering the key principles, the aim is that you come away from this book feeling more confident in tackling your own ML engineering challenges, whatever your chosen toolset.

In this first chapter, you will learn about the different types of data roles relevant to ML engineering and why they are important, how to use this knowledge to build and work within appropriate teams, some of the key points to remember when building working ML products in the real world, how to start to isolate appropriate problems for engineered ML solutions, and how to create your own high-level ML system designs for a variety of typical business problems.

We will cover these topics in the following sections:

- Defining a taxonomy of data disciplines
- Assembling your team
- ML engineering in the real world

- What does an ML solution look like?
- High-level ML system design

Now that we have explained what we are going after in this first chapter, let's get started!

Technical requirements

Throughout the book, all code examples will assume the use of Python 3.10.8 unless specified otherwise. Examples in this edition have been run on a 2022 Macbook Pro with an M2 Apple silicon chip, with Rosetta 2 installed to allow backward compatibility with Intel-based applications and packages. Most examples have also been tested on a Linux machine running Ubuntu 22.04 LTS. The required Python packages for each chapter are stored in `conda` environment `.yml` files in the appropriate chapter folder in the book's Git repository. We will discuss package and environment management in detail later in the book. But in the meantime, assuming you have a GitHub account and have configured your environment to be able to pull and push from GitHub remote repositories, to get started you can clone the book repository from the command line:

```
git clone https://github.com/PacktPublishing/Ma
```

Assuming you have Anaconda or Miniconda installed, you can then navigate to the *Chapter01* folder of the Git repository for this book and run:

```
conda env create -f mlewp-chapter01.yml
```

This will set up the environment you can use to run the examples given in this chapter. A similar procedure can be followed for each chapter, but each section will also call out any installation requirements specific to those examples.

Now we have done some setup, we will start to explore the world of ML engineering and how it fits into a modern data ecosystem. Let's begin our exploration of the world of ML engineering!

Note: Before running the `conda` commands given in this section you may have to install a specific library manually. Some versions of the Facebook Prophet library require versions of PyStan that can struggle to build on Macbooks running Apple silicon. If you run into this issue, then you should try to install the package `httpstan` manually. First, go to <https://github.com/stan-dev/httpstan/tags> and select a version of the package to install. Download the `.tar.gz` or `.zip` of that version and extract it. Then you can navigate to the extracted folder and run the following commands:

```
make  
python3 -m pip install poetry  
python3 -m poetry build  
python3 -m pip install dist/*.whl
```



You may also run into an error like the following when you call `model.fit()` in the later example:

```
dyld[29330]: Library not loaded: '@rpath
```

If this is the case you will have to run the following commands, substituting in the correct path for your Prophet installation location in the Conda environment:

```
cd /opt/homebrew/Caskroom/miniforge/k  
install_name_tool -add_rpath @execut&
```

Oh, the joys of doing ML on Apple silicon!

Defining a taxonomy of data disciplines

The explosion of data and the potential applications of it over the past few years have led to a proliferation of job roles and responsibilities. The debate that once raged over how a *data scientist* was different from a *statistician* has now become extremely complex. I would argue, however, that it does not have to be so complicated. The activities that have to be undertaken to get value from data are pretty consistent, no matter what business vertical you are in, so it should be reasonable to expect that the skills and roles you need to perform these steps will also be relatively consistent. In this chapter, we will explore some of the main data disciplines that I think you will always need in any data project. As you can guess, given the name of this book, I will be particularly keen to explore the notion of *ML engineering* and how this fits into the mix.

Let's now look at some of the roles involved in using data in the modern landscape.

Data scientist

After the *Harvard Business Review* declared that being a data scientist was *The Sexiest Job of the 21st Century* (<https://hbr.org/2012/10/data-scientist-the-sexiest-job-of-the-21st-century>), this job role became one of the most sought after, but also hyped, in the mix. Its popularity remains high, but the challenges of taking advanced analytics and ML into production have meant there has been more and more of a shift toward engineering roles within data-driven organizations. The traditional data scientist role can cover an entire spectrum of duties, skills, and responsibilities depending on the business vertical, the organization, or even just personal preference. No matter how this role is defined, however, there are some key areas of focus that should always be part of the data scientist's job profile:

- **Analysis:** A data scientist should be able to wrangle, munge, manipulate, and consolidate datasets before performing calculations on the data that help us to understand it. *Analysis* is a broad term, but it's clear that the end result is knowledge of your dataset that you didn't have before you started, no matter how basic or complex.
- **Modeling:** The thing that gets everyone excited (potentially including you, dear reader) is the idea of modeling phenomena found in your data. A data scientist usually has to be able to apply statistical, mathematical, and ML techniques to data, in order to explain processes

or relationships contained within it and to perform some sort of prediction.

- **Working with the customer or user:** The data scientist role usually has some more business-directed elements so that the results of the previous two points can support decision-making in an organization. This could be done by presenting the results of the analysis in PowerPoint presentations or Jupyter notebooks, or even sending an email with a summary of the key results. It involves communication and business acumen in a way that goes beyond classic tech roles.

ML engineer

The gap between creating ML proof-of-concept and building robust software, what I often refer to in talks as “the chasm,” has led to the rise of what I would now argue is one of the most important roles in technology.

The ML engineer serves an acute need to translate the world of data science modeling and exploration into the world of software products and systems engineering. Since this is no easy feat, the ML engineer has become increasingly sought after and is now a critical piece of the data-driven software value chain. If you cannot get things into production, you are not generating value, and if you are not generating value, well we know that’s not great!

You can articulate the need for this type of role quite nicely by considering a classic voice assistant. In this case, a data scientist would usually focus on translating the business requirements into a working *speech-to-text* model, potentially a very complex neural network, and showing that it can perform the desired voice transcription task *in principle*. ML engineering is then all about how you take that speech-to-text model and build it into a product,

service, or tool that can be used *in production*. Here, it may mean building some software to train, retrain, deploy, and track the performance of the model as more transcription data is accumulated, or as user preferences are understood. It may also involve understanding how to interface with other systems and provide results from the model in the appropriate formats. For example, the results of the model may need to be packaged into a JSON object and sent via a REST API call to an online supermarket, in order to fulfill an order.

Data scientists and ML engineers have a lot of overlapping skillsets and competencies but have different areas of focus and strengths (more on this later), so they will usually be part of the same project team and may have either title, but it will be clear what hat they wear from what they do in that project.

Similar to the data scientist, we can define the key areas of focus for the ML engineer:

- **Translation:** Taking models and research code in a variety of formats and translating them into slicker, more robust pieces of code.
This can be done using OO programming, functional programming, a mix, or something else, but it basically helps to take the *proof-of-concept* work of the data scientist and turn it into something that is far closer to being trusted in a production environment.
- **Architecture:** Deployments of any piece of software do not occur in a vacuum and will always involve lots of integrated parts. This is true of ML solutions as well. The ML engineer has to understand how the appropriate tools and processes link together so that the models built with the data scientist can do their job and do it at scale.

- **Productionization:** The ML engineer is focused on delivering a solution and so should understand the customer's requirements inside out, as well as be able to understand what that means for the project development. The end goal of the ML engineer is not to provide a good model (though that is part of it), nor is it to provide something that *basically works*. Their job is to make sure that the hard work on the data science side of things generates the maximum potential value in a real-world setting.

ML operations engineer

ML engineering will be the focus of this book, but there is an important role now emerging with the aim of enabling ML engineers to do their work with higher quality, at greater pace, and at a larger scale. These are the **Machine Learning Operations (MLOps)** engineers. This role is all about building out the tools and capabilities that enable the tasks of the ML engineer and data scientists. This role focuses more on building out the tooling, platforms, and automation used by the other roles, and so connects them nicely. That is not to say MLOps engineers will not be used in specific projects or builds; it is just that their main value-add comes not from this but from enabling the capabilities used during a specific project or build. If we revisit the example of the speech-to-text solution described in the *ML engineer* section, we can get a flavor of this. Where the ML engineer will be worried about building out a solution that works seamlessly in production, the MLOps engineer will work hard to build out the platform or toolset that the ML engineer uses to do this. The ML engineer will build pipelines, but the MLOps engineer may build pipeline templates; the ML engineer may use **continuous integration/continuous deployment (CI/CD)** practices

(more on this later), but the MLOps engineer will enable that capability and define the best practice to use CI/CD smoothly. Finally, where the ML engineer thinks “How do I solve this specific problem robustly using the proper tools and techniques?”, the MLOps engineer asks “How do I make sure that the ML engineers and data scientists will be able to, in general, solve the types of problems they need to, and how can I continually update and improve that setup?”

As we did with the data scientist and ML engineer, let us define some of the key areas of focus for the MLOps engineer:

- **Automation:** Increasing the level of automation across the data science and ML engineering workflows through the use of techniques such as CI/CD and **Infrastructure-as-Code (IAC)**. Pre-package software that can be deployed to allow for smoother deployments of solutions through these capabilities and more, such as automation scripts or standardized templates.
- **Platform engineering:** Working to integrate a series of useful services together in order to build out the ML platform for the different data-driven teams to use. This can include developing integrations across orchestration tools, compute, and more data-driven services until they become a holistic whole for use by ML engineers and data scientists.
- **Enabling key MLOps capabilities:** MLOps consists of a set of practices and techniques that enable the productionization of ML models by the other engineers in the team. Capabilities such as model management and model monitoring should be enabled by the MLOps engineers in a way that can be used at scale across multiple projects.

It should be noted that some of the topics covered in this book could be carried out by an MLOps engineer and that there is naturally some overlap.

This should not concern us too much, as MLOps is based on quite a generic set of practices and capabilities that can be encompassed by multiple roles (see *Figure 1.1*).

Data engineer

The data engineers are the people responsible for getting the commodity that everything else in the preceding sections is based on from A to B with high fidelity, appropriate latency, and as little effort on the part of the other team members as possible. You cannot create any type of software product, never mind an ML product, without data.

The key areas of focus for a data engineer are as follows:

- **Quality:** Getting data from A to B is a pointless exercise if the data is garbled, fields are missing, or IDs are screwed up. The data engineer cares about avoiding this and uses a variety of techniques and tools, generally to ensure that the data that left the source system is what lands in your data storage layer.
- **Stability:** Similar to the previous point on quality, if the data comes from A to B but it only arrives every second Wednesday if it's not a rainy day, then what's the point?

Data engineers spend a lot of time and effort and use their considerable skills to ensure that data pipelines are robust, reliable, and can be trusted to deliver when promised.

- **Access:** Finally, the aim of getting data from A to B is for it to be used by applications, analyses, and ML models, so the nature of B is important. The data engineer will have a variety of technologies to hand to surface data and should work with the data consumers (our

data scientists and ML engineers, among others) to define and create appropriate data models within these solutions:

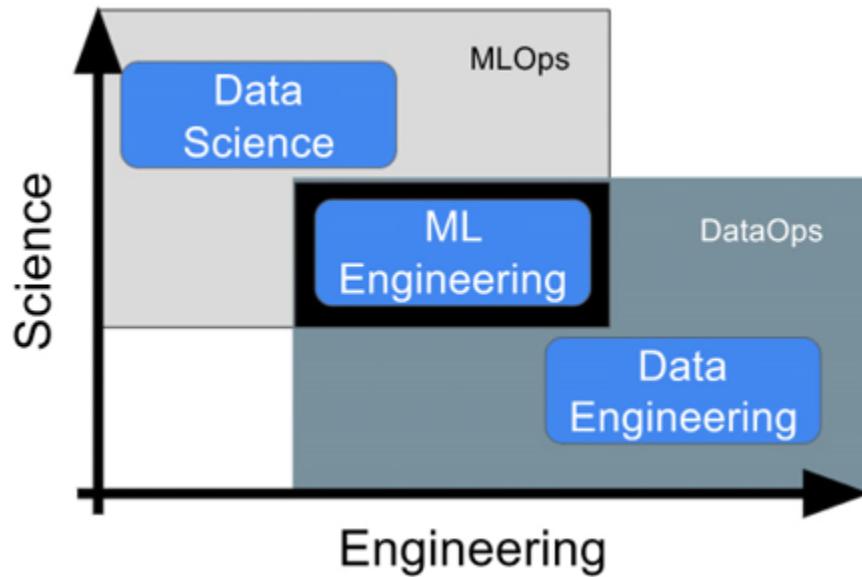


Figure 1.1: A diagram showing the relationships between data science, ML engineering, and data engineering.

As mentioned previously, this book focuses on the work of the ML engineer and how you can learn some of the skills useful for that role, but it is important to remember that you will not be working in a vacuum. Always keep in mind the profiles of the other roles (and many more not covered here that will exist in your project team) so that you work most effectively together. Data is a team sport after all!

Now that you understand the key roles in a modern data team and how they cover the spectrum of activities required to build successful ML products, let's look at how you can put them together to work efficiently and effectively.

Working as an effective team

In modern software organizations, there are many different methodologies to organize teams and get them to work effectively together. We will cover some of the project management methodologies that are relevant in *Chapter 2, The Machine Learning Development Process*, but in the meantime, this section will discuss some important points you should consider if you are ever involved in forming a team, or even if you just work as part of a team, that will help you become an effective teammate or lead.

First, always bear in mind that *nobody can do everything*. You can find some very talented people out there, but do not ever think one person can do everything you will need to the level you require. This is not just unrealistic; it is bad practice and will negatively impact the quality of your products. Even when you are severely resource-constrained, the key is for your team members to have a laser-like focus to succeed.

Second, *blended is best*. We all know the benefits of diversity for organizations and teams in general and this should, of course, apply to your ML team as well. Within a project, you will need mathematics, code, engineering, project management, communication, and a variety of other skills to succeed. So, given the previous point, make sure you cover this in at least some sense across your team.

Third, *tie your team structure to your projects in a dynamic way*. If you work on a project that is mostly about getting the data in the right place and the actual ML models are really simple, focus your team profile on the engineering and data modeling aspects. If the project requires a detailed understanding of the model, and it is quite complex, then reposition your team to make sure this is covered. This is just sensible and frees up team members who would otherwise have been underutilized to work on other projects.

As an example, suppose that you have been tasked with building a system that classifies customer data as it comes into your shiny new data lake, and the decision has been taken that this should be done at the point of ingestion via a streaming application. The classification has already been built for another project. It is already clear that this solution will heavily involve the skills of the data engineer and the ML engineer, but not so much the data scientist, since that portion of the work will have been completed in another project.

In the next section, we will look at some important points to consider when deploying your team on a real-world business problem.

ML engineering in the real world

The majority of us who work in ML, analytics, and related disciplines do so for organizations with a variety of different structures and motives. These could be for for-profit corporations, not-for-profits, charities, or public sector organizations like government or universities. In pretty much all of these cases, we do not do this work in a vacuum and not with an infinite budget of time or resources. It is important, therefore, that we consider some of the important aspects of doing this type of work in the *real world*.

First of all, the ultimate goal of your work is to generate **value**. This can be calculated and defined in a variety of ways, but fundamentally your work has to improve something for the company or its customers in a way that justifies the investment put in. This is why most companies will not be happy for you to take a year to play with new tools and then generate nothing concrete to show for it, or to spend your days only reading the latest

papers. Yes, these things are part of any job in technology, and they can definitely be super-fun, but you have to be strategic about how you spend your time and always be aware of your value proposition.

Secondly, to be a successful ML engineer in the real world, you cannot just understand the technology; you must understand *the business*. You will have to understand how the company works day to day, you will have to understand how the different pieces of the company fit together, and you will have to understand the people of the company and their roles. Most importantly, you have to understand *the customer*, both of the business and your work. If you do not know the motivations, pains, and needs of the people you build for, then how can you be expected to build the right thing?

Finally, and this may be controversial, the most important skill for you to become a successful ML engineer in the real world is one that this book will not teach you, and that is the ability to communicate effectively. You will have to work in a team, with a manager, with the wider community and business, and, of course, with your customers, as mentioned above. If you can do this and you know the technology and techniques (many of which are discussed in this book), then what can stop you?

But what kinds of problems can you solve with ML when you work in the real world? Well, let's start with another potentially controversial statement: *a lot of the time, ML is not the answer*. This may seem strange given the title of this book, but it is just as important to know when *not* to apply ML as when to apply it. This will save you tons of expensive development time and resources.

ML is ideal for cases when you want to do a semi-routine task faster, with more accuracy, or at a far larger scale than is possible with other solutions.

Some typical examples are given in the following table, along with some discussion as to whether or not ML would be an appropriate tool to solve the problem:

Requirement	Is ML Appropriate?	Details
Anomaly detection of energy pricing signals.	Yes	You will want to do this on large numbers of points on potentially varying time signals.
Improving data quality in an ERP system.	No	This sounds more like a process problem. You can try and apply ML to this but often it is better to make the data entry process more automated or the process more robust.
Forecasting item consumption for a warehouse.	Yes	ML will be able to do this more accurately than a human can, so this is a good area of application.
Summarizing data for	Maybe	This can be required at scale but it is not an ML problem – simple queries against your data will do.

business reviews.

Table 1.1: Potential use cases for ML.

As this table of simple examples hopefully starts to make clear, the cases where ML *is* the answer are ones that can usually be very well framed as a mathematical or statistical problem. After all, this is what ML really is – a series of algorithms rooted in mathematics that can iterate some internal parameters based on data. Where the lines start to blur in the modern world are through advances in areas such as deep learning or reinforcement learning, where problems that we previously thought would be very hard to phrase appropriately for standard ML algorithms can now be tackled.

The other tendency to watch out for in the real world (to go along with *let's use ML for everything*) is the worry that people have about ML coming for their job and that it should not be trusted. This is understandable: a report by PwC in 2018 suggested that 30% of UK jobs will be impacted by automation by the 2030s (*Will Robots Really Steal Our Jobs?*:

<https://www.pwc.co.uk/economic-services/assets/international-impact-of-automation-feb-2018.pdf>). What you have to try and make clear when working with your colleagues and customers is that what you are building is there to supplement and augment their capabilities, not to replace them.

Let's conclude this section by revisiting an important point: the fact that you work for a company means, of course, that the aim of the game is to create value appropriate to the investment. In other words, you need to

show a good **Return on Investment (ROI)**. This means a couple of things for you practically:

- You have to understand how different designs require different levels of investment. If you can solve your problem by training a deep neural net on a million images with a GPU running 24/7 for a month, or you know you can solve the same problem with some basic clustering and a few statistics on some standard hardware in a few hours, which should you choose?
- You have to be clear about the *value* you will generate. This means you need to work with experts and try to translate the results of your algorithm into actual dollar values. This is so much more difficult than it sounds, so you should take the time you need to get it right. And never, ever over-promise. *You should always under-promise and over-deliver.*

Adoption is not guaranteed. Even when building products for your colleagues within a company, it is important to understand that your solution will be tested every time someone uses it post-deployment. If you build shoddy solutions, then people will not use them, and the value proposition of what you have done will start to disappear.

Now that you understand some of the important points when using ML to solve business problems, let's explore what these solutions can look like.

What does an ML solution look like?

When you think of ML engineering, you would be forgiven for defaulting to imagining working on voice assistance and visual recognition apps (I fell

into this trap in previous pages – did you notice?). The power of ML, however, lies in the fact that wherever there is data and an appropriate problem, it can help and be integral to the solution.

Some examples might help make this clearer. When you type a text message and your phone suggests the next words, it can very often be using a natural language model under the hood. When you scroll any social media feed or watch a streaming service, recommendation algorithms are working double time. If you take a car journey and an app forecasts when you are likely to arrive at your destination, there is going to be some kind of regression at work. Your loan application often results in your characteristics and application details being passed through a classifier. These applications are not the ones shouted about on the news (perhaps with the exception of when they go horribly wrong), but they are all examples of brilliantly put-together ML engineering.

In this book, the examples we will work through will be more like these – typical scenarios for ML encountered in products and businesses every day. These are solutions that, if you can build them confidently, will make you an asset to any organization.

We should start by considering the broad elements that should constitute any ML solution, as indicated in the following diagram:



Figure 1.2: Schematic of the general components or layers of any ML solution and what they are responsible for.

Your **storage layer** constitutes the endpoint of the data engineering process and the beginning of the ML one. It includes your data for training, your results from running your models, your artifacts, and important metadata. We can also consider this as the layer including your stored code.

The **compute layer** is where the *magic* happens and where most of the focus of this book will be. It is where training, testing, prediction, and transformation all (mostly) happen. This book is all about making this layer as well engineered as possible and interfacing with the other layers.

You can break this layer down to incorporate these pieces as shown in the following workflow:

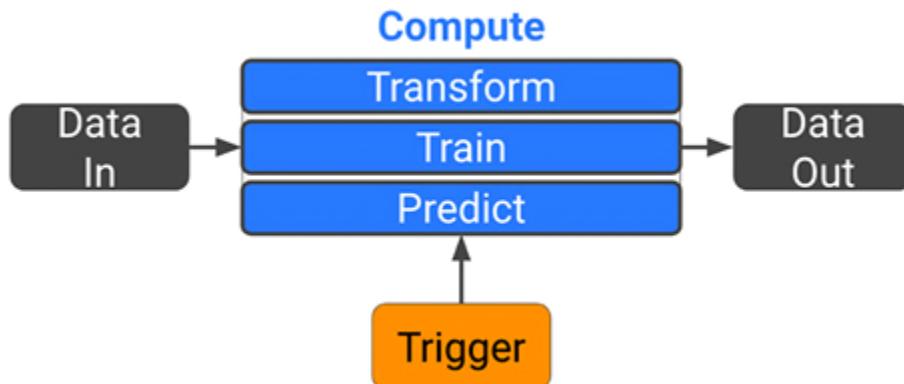


Figure 1.3: The key elements of the compute layer.



IMPORTANT NOTE

The details are discussed later in the book, but this highlights the fact that at a fundamental level, your compute processes for any ML solution are really just about taking some data in and pushing some data out.

The **application layer** is where you share your ML solution's results with other systems. This could be through anything from application database insertion to API endpoints, message queues, or visualization tools. This is the layer through which your customer eventually gets to use the results, so you must engineer your system to provide clean and understandable outputs, something we will discuss later.

And that is it in a nutshell. We will go into detail about all of these layers and points later, but for now, just remember these broad concepts and you will start to understand how all the detailed technical pieces fit together.

Why Python?

Before moving on to more detailed topics, it is important to discuss why Python has been selected as the programming language for this book. Everything that follows that pertains to higher-level topics, such as architecture and system design, can be applied to solutions using any or multiple languages, but Python has been singled out here for a few reasons.

Python is colloquially known as the *lingua franca* of data. It is a non-compiled, not strongly typed, and multi-paradigm programming language that has a clear and simple syntax. Its tooling ecosystem is also extensive, especially in the analytics and ML space.

Packages such as scikit-learn, numpy, scipy, and a host of others form the backbone of a huge amount of technical and scientific development across the world. Almost every major new software library for use in the data world has a Python API. It is the most popular programming language in the world, according to the **TIOBE index**

(<https://www.tiobe.com/tiobe-index/>) at the time of writing (August 2023).

Given this, being able to build your systems using Python means you will be able to leverage all of the excellent ML and data science tools available in this ecosystem, while also ensuring that you build applications that can play nicely with other software.

High-level ML system design

When you get down to the nuts and bolts of building your solution, there are so many options for tools, tech, and approaches that it can be very easy to be overwhelmed. However, as alluded to in the previous sections, a lot of this complexity can be abstracted to understand the bigger picture via some *back-of-the-envelope* architecture and designs. This is always a useful exercise once you know what problem you will try and solve, and it is something I recommend doing before you make any detailed choices about implementation.

To give you an idea of how this works in practice, what follows are a few worked-through examples where a team has to create a high-level ML systems design for some typical business problems. These problems are similar to ones I have encountered before and will likely be similar to ones you will encounter in your own work.

Example 1: Batch anomaly detection service

You work for a tech-savvy taxi ride company with a fleet of thousands of cars. The organization wants to start making ride times more consistent and

understand longer journeys in order to improve the customer experience and, thereby, increase retention and return business. Your ML team is employed to create an anomaly detection service to find rides that have unusual ride time or ride length behaviors. You all get to work, and your data scientists find that if you perform clustering on sets of rides using the features of ride distance and time, you can clearly identify outliers worth investigating by the operations team. The data scientists present the findings to the CTO and other stakeholders before getting the go-ahead to develop this into a service that will provide an outlier flag as a new field in one of the main tables of the company's internal analysis tool.

In this example, we will simulate some data to show how the taxi company's data scientists could proceed. In the repository for the book, which can be found at

<https://github.com/PacktPublishing/Machine-Learning-Engineering-with-Python-Second-Edition>, if you navigate to the folder *Chapter01*, you will see a script called `clustering_example.py`. If you have activated the `conda` environment provided via the `mlewp-chapter01.yml` environment file, then you can run this script with:

```
python3 clustering_example.py
```

After a successful run you should see that three files are created: `taxi-rides.csv`, `taxi-labels.json`, and `taxi-rides.png`. The image in `taxi-rides.png` should look something like that shown in *Figure 1.4*.

We will walk through how this script is built up:

1. First, let's define a function that will simulate some ride distances based on the random distribution given in `numpy` and return a `numpy` array containing the results. The reason for the repeated lines is so that we can create some base behavior and anomalies in the data, and you can clearly compare against the speeds we will generate for each set of taxis in the next step:

```
import numpy as np
from numpy.random import MT19937
from numpy.random import RandomState, SeedSe
rs = RandomState(MT19937(SeedSequence(123456

# Define simulate ride data function
def simulate_ride_distances():
    ride_dists = np.concatenate(
        (
            10 * np.random.random(size=370),
            10 * np.random.random(size=10),
            10 * np.random.random(size=10),
            10 * np.random.random(size=10) #
        )
    )
    return ride_dists
```

2. We can now do the exact same thing for speeds, and again we have split the taxis into sets of `370`, `10`, `10`, and `10` so that we can create some data with “typical” behavior and some sets of anomalies, while allowing for clear matching of the values with the `distances` function:

```
def simulate_ride_speeds():
    ride_speeds = np.concatenate(
        (
            np.random.normal(loc=30, scale=5),
            np.random.normal(loc=30, scale=5),
            np.random.normal(loc=50, scale=1),
            np.random.normal(loc=15, scale=4)
        )
    )
    return ride_speeds
```

3. We can now use both of these helper functions inside a function that will call them and bring them together to create a simulated dataset containing ride IDs, speeds, distances, and times. The result is returned as a `pandas` DataFrame for use in modeling:

```
def simulate_ride_data():
    # Simulate some ride data ...
    ride_dists = simulate_ride_distances()
    ride_speeds = simulate_ride_speeds()
    ride_times = ride_dists / ride_speeds
    # Assemble into Data Frame
    df = pd.DataFrame(
        {
            'ride_dist': ride_dists,
            'ride_time': ride_times,
            'ride_speed': ride_speeds
        }
    )
```

```
    ride_ids = datetime.datetime.now().strftime("%Y-%m-%d %H:%M")
    df.index.astype(str)
    df['ride_id'] = ride_ids
    return df
```

4. Now, we get to the core of what data scientists produce in their projects, which is a simple function that wraps some `sklearn` code to return a dictionary with the clustering run metadata and results. We include the relevant imports here for ease:

```
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import DBSCAN
from sklearn import metrics

def cluster_and_label(data, create_and_show_=True):
    data = StandardScaler().fit_transform(data)
    db = DBSCAN(eps=0.3, min_samples=10).fit(data)
    # Find labels from the clustering
    core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
    core_samples_mask[db.core_sample_indices_] = True
    labels = db.labels_
    # Number of clusters in labels, ignoring noise if present
    n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
    n_noise_ = list(labels).count(-1)
    run_metadata = {
        'nClusters': n_clusters_,
        'nNoise': n_noise_,
        'silhouetteCoefficient': metrics.silhouette_score(data, labels)}
```

```
        'labels': labels,
    }
    if create_and_show_plot:
        plot_cluster_results(data, labels, c
                             n_clusters_)
    else:
        pass
    return run_metadata
```

Note that the function in *step 4* leverages a utility function for plotting that is shown below:

```
import matplotlib.pyplot as plt

def plot_cluster_results(data, labels, core_
                           n_clusters_):
    fig = plt.figure(figsize=(10, 10))
    # Black removed and is used for noise in
    unique_labels = set(labels)
    colors = [plt.cm.cool(each) for each in
              len(unique_labels))]
    for k, col in zip(unique_labels, colors):
        if k == -1:
            # Black used for noise.
            col = [0, 0, 0, 1]
        class_member_mask = (labels == k)
        xy = data[class_member_mask & core_s
                  plt.plot(xy[:, 0], xy[:, 1], 'o',
                           markerfacecolor=tuple(col),
                           markeredgecolor='k', marker
```

```
xy = data[class_member_mask & ~core_
plt.plot(xy[:, 0], xy[:, 1], '^',
          markerfacecolor=tuple(col),
          markeredgecolor='k', marker
plt.xlabel('Standard Scaled Ride Dist.')
plt.ylabel('Standard Scaled Ride Time')
plt.title('Estimated number of clusters:')
plt.savefig('taxi-rides.png')
```

Finally, this is all brought together at the entry point of the program, as shown below:

```
import logging
logging.basicConfig()
logging.getLogger().setLevel(logging.INFO)

if __name__ == "__main__":
    import os
    # If data present, read it in
    file_path = 'taxi-rides.csv'
    if os.path.exists(file_path):
        df = pd.read_csv(file_path)
    else:
        logging.info('Simulating ride data')
        df = simulate_ride_data()
        df.to_csv(file_path, index=False)
    X = df[['ride_dist', 'ride_time']]

    logging.info('Clustering and labelling')
    results = cluster_and_label(X, create_an
```

```
df['label'] = results['labels']

logging.info('Outputting to json ...')
df.to_json('taxi-labels.json', orient='r
```

This script, once run, creates a dataset showing each simulated taxi journey with its clustering label in `taxi-labels.json`, as well as the simulated dataset in `taxi-rides.csv` and the plot showing the results of the clustering in `taxi-rides.png`, as shown in *Figure 1.4*.

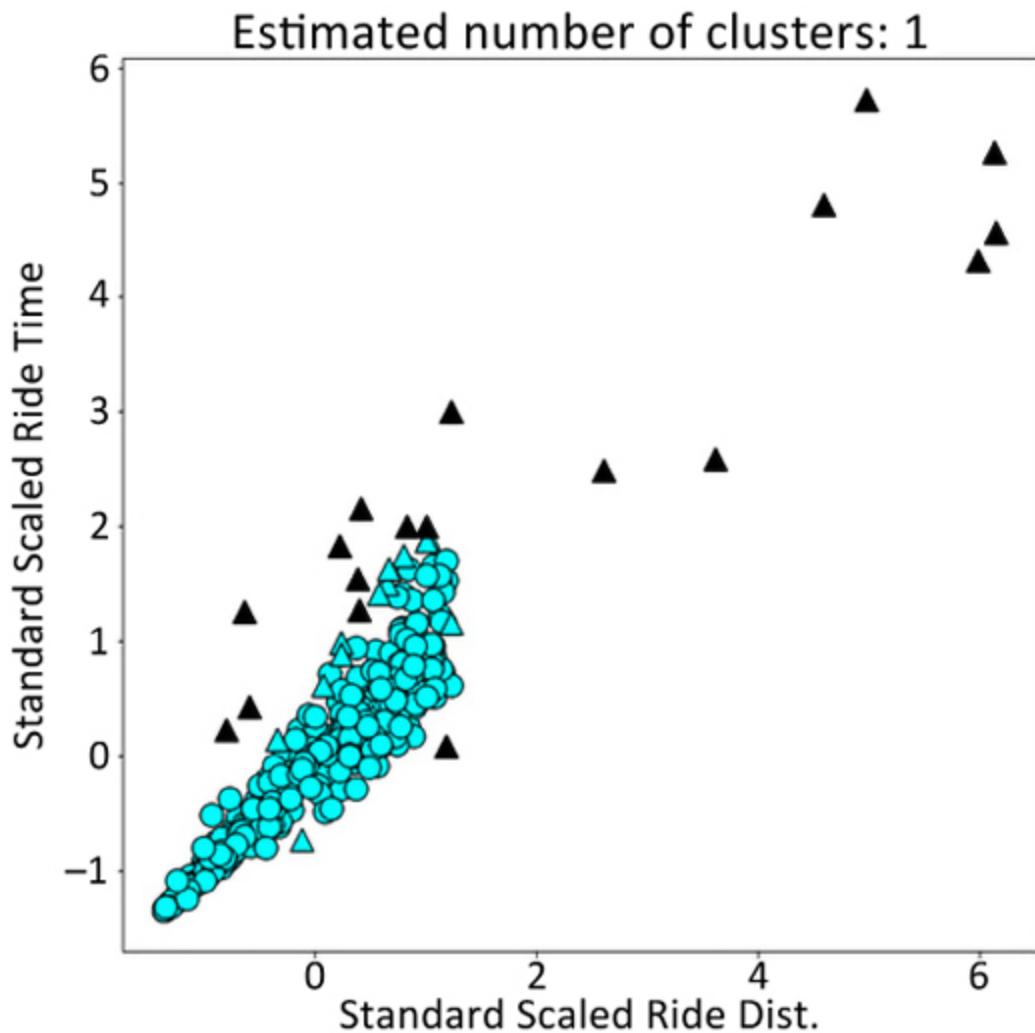


Figure 1.4: An example set of results from performing clustering on some taxi ride data.

Now that you have a basic model that works, you have to start thinking about how to pull this into an engineered solution – how could you do it?

Well, since the solution here will support longer-running investigations by another team, there is no need for a very low-latency solution. The stakeholders agree that the insights from clustering can be delivered at the end of each day. Working with the data science part of the team, the ML engineers (led by you) understand that if clustering is run daily, this provides enough data to give appropriate clusters, but doing the runs any more frequently could lead to poorer results due to smaller amounts of data. So, a daily batch process is agreed upon.

The next question is, how do you schedule that run? Well, you will need an orchestration layer, which is a tool or tools that will enable you to schedule and manage pre-defined jobs. A tool like Apache Airflow would do exactly this.

What do you do next? Well, you know the frequency of runs is daily, but the volume of data is still very high, so it makes sense to leverage a distributed computing paradigm. Two options immediately come to mind and are skillsets that exist within the team, Apache Spark and Ray. To provide as much decoupling as possible from the underlying infrastructure and minimize the refactoring of your code required, you decide to use Ray. You know that the end consumer of the data is a table in a SQL database, so you need to work with the database team to design an appropriate handover of the results. Due to security and reliability concerns, it is not a good idea to write to the production database directly. You, therefore, agree that another database in the cloud will be used as an intermediate staging area for the data, which the main database can query against on its daily builds.

It might not seem like we have done anything technical here, but actually, you have already performed the high-level system design for your project. The rest of this book tells you how to fill in the gaps in the following diagram!

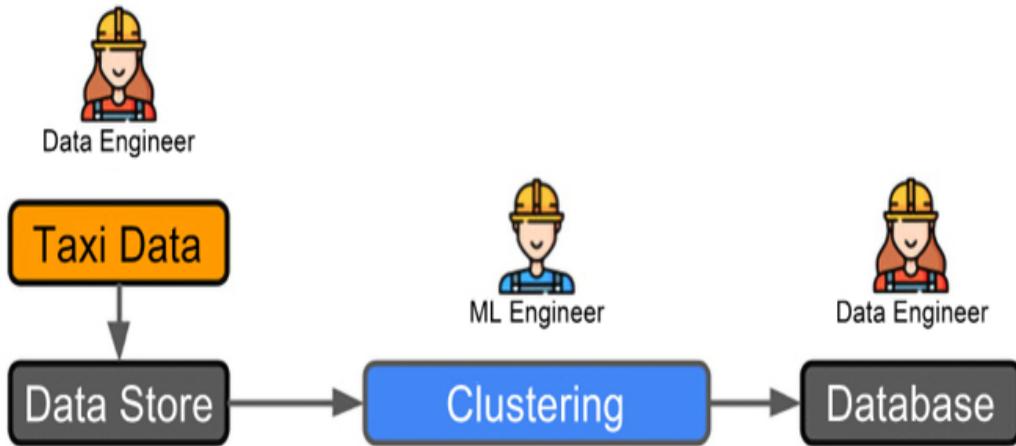


Figure 1.5: Example 1 workflow.

Let's now move on to the next example!

Example 2: Forecasting API

In this example, you work for the logistics arm of a large retail chain. To maximize the flow of goods, the company would like to help regional logistics planners get ahead of particularly busy periods and avoid product sell-outs. After discussions with stakeholders and subject matter experts across the business, it is agreed that the ability for planners to dynamically request and explore forecasts for particular warehouse items through a web-hosted dashboard is optimal. This allows the planners to understand likely future demand profiles before they make orders.

The data scientists come good again and find that the data has very predictable behavior at the level of any individual store. They decide to use

the Facebook Prophet library for their modeling to help speed up the process of training many different models. In the following example we will show how they could do this, but we will not spend time optimizing the model to create the best predictive performance, as this is just for illustration purposes.

This example will use the Kaggle API in order to retrieve an exemplar dataset for sales in a series of different retail stores. In the book repository under *Chapter01/forecasting* there is a script called `forecasting_example.py`. If you have your Python environment configured appropriately you can run this example with the following command at the command line:

```
python3 forecasting_example.py
```

The script downloads the dataset, transforms it, and uses it to train a Prophet forecasting model, before running a prediction on a test set and saving a plot. As mentioned, this is for illustration purposes only and so does not create a validation set or perform any more complex hyperparameter tuning than the defaults provided by the Prophet library.

To help you see how this example is pieced together, we will now break down the different components of the script. Any functionality that is purely for plotting or logging is excluded here for brevity:

1. If we look at the main block of the script, we can see that the first steps all concern reading in the dataset if it is already in the correct directory, or downloading and then reading it in otherwise:

```
import pandas as pd

if __name__ == "__main__":
    import os
    file_path = train.csv
    if os.path.exists(file_path):
        df = pd.read_csv(file_path)
    else:
        download_kaggle_dataset()
        df = pd.read_csv(file_path)
```

2. The function that performed the download used the Kaggle API and is given below; you can refer to the Kaggle API documentation to ensure this is set up correctly (which requires a Kaggle account):

```
import kaggle

def download_kaggle_dataset( kaggle_dataset:
                                rossmann-store-
                                api = kaggle.api
                                kaggle.api.dataset_download_files(kaggle
                                unzip=
```

3. Next, the script calls a function to transform the dataset called `prep_store_data`. This is called with two default values, one for a store ID and the other specifying that we only want to see data for when the store was open. The definition of this function is given below:

```
def prep_store_data(df: pd.DataFrame,
                    store_id: int = 4,
                    store_open: int = 1) ->
    df['Date'] = pd.to_datetime(df['Date'])
    df.rename(columns= {'Date':'ds', 'Sales': 'y'})
    df_store = df[
        (df['Store'] == store_id) &
        (df['Open'] == store_open)
    ].reset_index(drop=True)
    return df_store.sort_values('ds', ascending=True)
```

4. The Prophet forecasting model is then trained on the first 80% of the data and makes a prediction on the remaining 20% of the data. Seasonality parameters are provided to the model in order to guide its optimization:

```
seasonality = {
    'yearly': True,
    'weekly': True,
    'daily': False
}
predicted, df_train, df_test, train_index =
    df = df,
    train_fraction = 0.8,
    seasonality=seasonality
)
```

The definition of the `train_predict` method is given below, and you can see that it wraps some further data prep and the main calls to

the Prophet package:

```
def train_predict(df: pd.DataFrame, train_fraction: float) -> tuple[pd.DataFrame, pd.DataFrame]:
    train_index = int(train_fraction * df.shape[0])
    df_train = df.copy().iloc[0:train_index]
    df_test = df.copy().iloc[train_index:]
    model = Prophet(
        yearly_seasonality=seasonality['year'],
        weekly_seasonality=seasonality['week'],
        daily_seasonality=seasonality['daily'],
        interval_width=0.95
    )
    model.fit(df_train)
    predicted = model.predict(df_test)
    return predicted, df_train, df_test, train_index
```

5. Then, finally, a utility plotting function is called, which when run will create the output shown in *Figure 1.6*. This shows a zoomed-in view of the prediction on the test dataset. The details of this function are not given here for brevity, as discussed above:

```
plot_forecast(df_train, df_test, predicted)
```

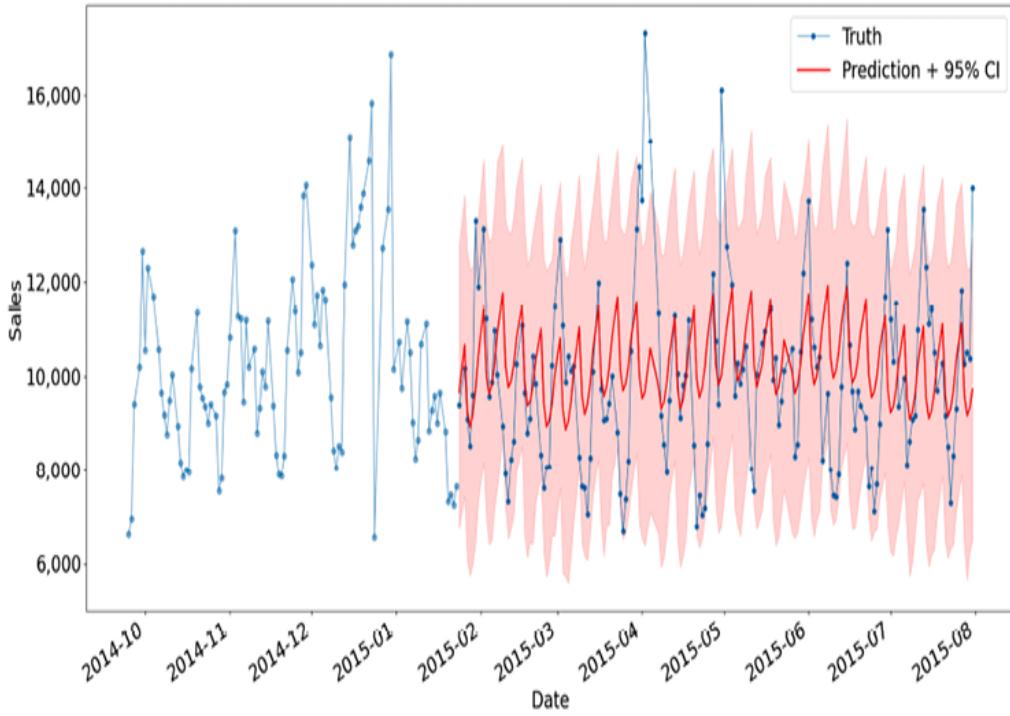


Figure 1.6: Forecasting store sales.

One issue here is that implementing a forecasting model like the one above for every store can quickly lead to hundreds or even thousands of models if the chain gathers enough data. Another issue is that not all stores are on the resource planning system used at the company yet, so some planners would like to retrieve forecasts for other stores they know are similar to their own. It is agreed that if users like this can explore regional profiles they believe are similar to their own data, then they can still make the optimal decisions.

Given this and the customer requirements for dynamic, ad hoc requests, you quickly rule out a full batch process. This wouldn't cover the use case for regions not on the core system and wouldn't allow for dynamic retrieval of up-to-date forecasts via the website, which would allow you to deploy models that forecast at a variety of time horizons in the future. It also means you could save on compute as you don't need to manage the storage and

updating of thousands of forecasts every day and your resources can be focused on model training.

Therefore, you decide that, actually, a web-hosted API with an endpoint that can return forecasts as needed by the user makes the most sense. To give efficient responses, you have to consider what happens in a typical user session. By workshopping with the potential users of the dashboard, you quickly realize that although the requests are dynamic, most planners will focus on particular items of interest in any one session. They will also not look at many regions. You then decide that it makes sense to have a caching strategy, where you take certain requests that you think might be common and cache them for reuse in the application.

This means that after the user makes their first selections, results can be returned more quickly for a better user experience. This leads to the rough system sketch in *Figure 1.7*:

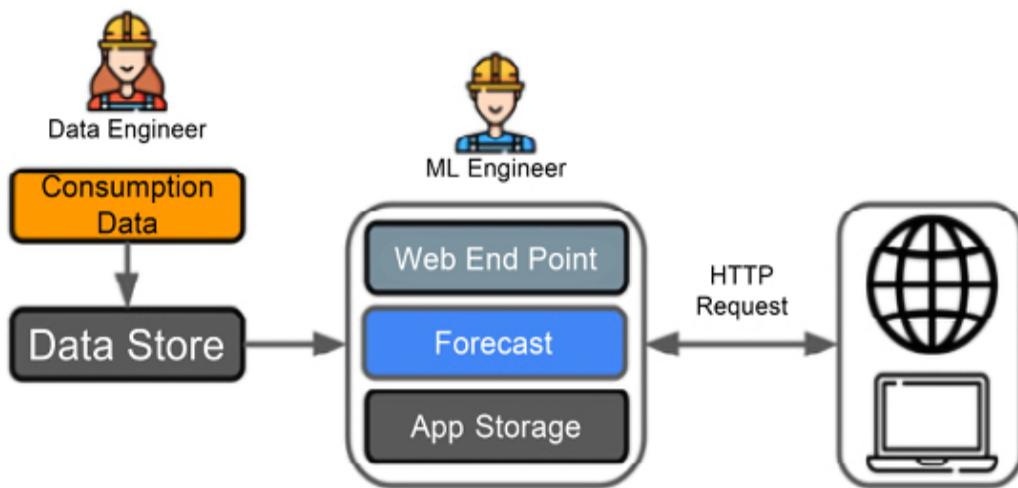


Figure 1.7: Example 2 workflow.

Next, let's look at the final example.

Example 3: Classification pipeline

In this final example, you work for a web-based company that wants to classify users based on their usage patterns as targets for different types of advertising, in order to more effectively target marketing spend. For example, if the user uses the site less frequently, we may want to entice them with more aggressive discounts. One of the key requirements from the business is that the end results become part of the data landed in a data store used by other applications.

Based on these requirements, your team determines that a pipeline running a classification model is the simplest solution that ticks all the boxes. The data engineers focus their efforts on building the ingestion and data store infrastructure, while the ML engineer works to wrap up the classification model the data science team has trained on historical data. The base algorithm that the data scientists settle on is implemented in `sklearn`, which we will work through below by applying it to a marketing dataset that would be similar to that produced in this use case.

This hypothetical example aligns with a lot of classic datasets, including the Bank Marketing dataset from the UCI ML repository:

<https://archive.ics.uci.edu/ml/datasets/Bank+Marketing#>. As in the previous example, there is a script you can run from the command line, this time in the *Chapter01/classifying* folder and called `classify_example.py`:

```
python3 classify_example.py
```

Running this script will read in the downloaded bank data, rebalance the training dataset, and then execute a hyperparameter optimization run on a randomized grid search for a random forest classifier. Similarly to before, we will show how these pieces work to give a flavor of how a data science team might have approached this problem:

1. The main block of the script contains all the relevant steps, which are neatly wrapped up into methods we will dissect over the next few steps:

```
if __name__ == "__main__":
    X_train, X_test, y_train, y_test = ingest_and_prep_data()
    X_balanced, y_balanced = rebalance_class()
    rf_random = get_randomised_rf_cv(
        random_grid=get_hyperparameter_grid())
    rf_random.fit(X_balanced, y_balanced)
```

2. The `ingest_and_prep_data` function is given below, and it does assume that the `bank.csv` data is stored in a directory called `bank_data` in the current folder. It reads the data into a `pandas` DataFrame, before performing a train-test split on the data and one-hot encoding the training features, before returning all the train and test features and targets. As in the other examples, most of these concepts and tools will be explained throughout the book, particularly in *Chapter 3, From Model to Model Factory*:

```
def ingest_and_prep_data(
    bank_dataset: str = 'bank_data/bank.'
```

```
) -> tuple[pd.DataFrame, pd.DataFrame]:  
    df = pd.read_csv('bank_data/bank.csv',  
                      decimal=',')  
  
    feature_cols = ['job', 'marital', 'education',  
                    'housing', 'loan', 'default']  
    X = df[feature_cols].copy()  
    y = df['y'].apply(lambda x: 1 if x == 'yes' else 0)  
    X_train, X_test, y_train, y_test = train_test_split(X, y,  
                                                       test_size=0.3)  
    enc = OneHotEncoder(handle_unknown='ignore')  
    X_train = enc.fit_transform(X_train)  
    return X_train, X_test, y_train, y_test
```

3. Because the data is imbalanced, we need to rebalance the training data with an oversampling technique. In this example, we will use the **Synthetic Minority Over-Sampling Technique (SMOTE)** from the `imblearn` package:

```
def rebalance_classes(X: pd.DataFrame, y: pd.Series) -> tuple[pd.DataFrame, pd.Series]:  
    sm = SMOTE()  
    X_balanced, y_balanced = sm.fit_resample(X, y)  
    return X_balanced, y_balanced
```

4. Now we will move on to the main ML components of the script. We will perform a hyperparameter search (there'll be more on this in

Chapter 3, From Model to Model Factory), so we have to define a grid to search over:

```
def get_hyperparam_grid() -> dict:
    n_estimators = [int(x) for x in np.linspace(2, 2000, num=10)]
    max_features = ['auto', 'sqrt']
    max_depth = [int(x) for x in np.linspace(1, 100, 10)]
    max_depth.append(None)
    min_samples_split = [2, 5, 10]
    min_samples_leaf = [1, 2, 4]
    bootstrap = [True, False] # Create the random_grid = {
        'n_estimators': n_estimators,
        'max_features': max_features,
        'max_depth': max_depth,
        'min_samples_split': min_samples_split,
        'min_samples_leaf': min_samples_leaf,
        'bootstrap': bootstrap
    }
    return random_grid
```

5. Then finally, this grid of hyperparameters will be used in the definition of a `RandomisedSearchCV` object that allows us to optimize an estimator (here, a `RandomForestClassifier`) over the hyperparameter values:

```
def get_randomised_rf_cv(random_grid: dict)
    selection._se
```

```
rf = RandomForestClassifier()
rf_random = RandomizedSearchCV(
    estimator=rf,
    param_distributions=random_grid,
    n_iter=100,
    cv=3,
    verbose=2,
    random_state=42,
    n_jobs=-1,
    scoring='f1'
)
return rf_random
```

The example above highlights the basic components of creating a typical classification model, but the question we have to ask ourselves as engineers is, “Then what?” It is clear that we have to actually run predictions with the model that has been produced, so we’ll need to persist it somewhere and read it in later. This is similar to the other use cases discussed in this chapter. Where things are more challenging here is that in this case, the engineers may actually consider not running in a batch or request-response scenario but in a streaming context. This means we will have to consider new technologies like **Apache Kafka** that enable you to both publish and subscribe to “topics” where packets of data called “events” can be shared. Not only that, but we will also have to make decisions about how to interact with data in this way using an ML model, raising questions about the appropriate model hosting mechanism. There will also be some subtleties around how often you want to retrain your algorithm to make sure that the classifier does not go stale. This is before we consider questions of latency or of monitoring the model’s performance in this very different setting. As

you can see, this means that the ML engineer's job here is quite a complex one. *Figure 1.8* subsumes all this complexity into a very high-level diagram that would allow you to start considering the sort of system interactions you would need to build if you were the engineer on this project.

We will not cover streaming in that much detail in this book, but we will cover all of the other key components that would help you build out this example into a real solution in a lot of detail. For more details on streaming ML applications please see the book *Machine Learning for Streaming Data with Python* by Joose Korstanje, Packt, 2022.

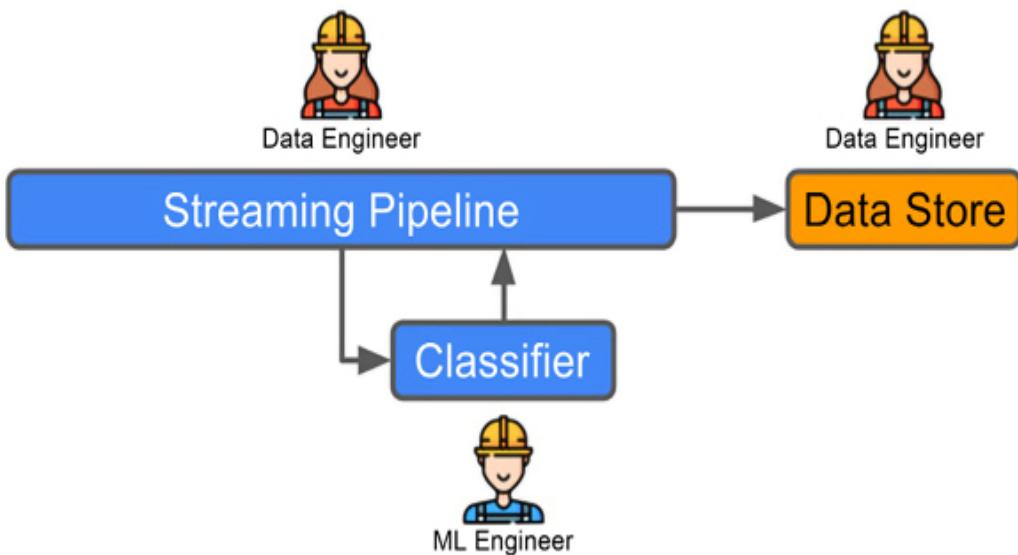


Figure 1.8: Example 3 workflow.

We have now explored three high-level ML system designs and discussed the rationale behind our workflow choices. We have also explored in detail the sort of code that would often be produced by data scientists working on modeling, but which would act as input to future ML engineering work. This section should, therefore, have given us an appreciation of where our engineering work begins in a typical project and what types of problems we

aim to solve. And there you go. You are already on your way to becoming an ML engineer!

Summary

In this chapter, we introduced the idea of ML engineering and how that fits within a modern team building valuable solutions based on data. There was a discussion of how the focus of ML engineering is complementary to the strengths of data science and data engineering and where these disciplines overlap. Some comments were made about how to use this information to assemble an appropriately resourced team for your projects.

The challenges of building ML products in modern real-world organizations were then discussed, along with pointers to help you overcome some of these challenges. In particular, the notions of reasonably estimating value and effectively communicating with your stakeholders were emphasized.

This chapter then rounded off with a taster of the technical content to come in later chapters, through a discussion of what typical ML solutions look like and how they should be designed (at a high level) for some common use cases.

These topics are important to cover before we dive deeper into the rest of the book, as they will help you to understand why ML engineering is such a critical discipline and how it ties into the complex ecosystem of data-focused teams and organizations. It also helps to give a taster of the complex challenges that ML engineering encompasses, while giving you some of the conceptual tools to start reasoning about those challenges. My hope is that this not only motivates you to engage with the material in the rest of this edition, but it also sets you down the path of exploration and

self-study that will be required to have a successful career as an ML engineer.

The next chapter will focus on how to set up and implement your development processes to build the ML solutions you want, providing some insight into how this is different from standard software development processes. Then there will be a discussion of some of the tools you can use to start managing the tasks and artifacts from your projects without creating major headaches. This will set you up for the technical details of how to build the key elements of your ML solutions in later chapters.

Join our community on Discord

Join our community's Discord space for discussion with the author and other readers:

<https://packt.link/mle>



2

The Machine Learning Development Process

In this chapter, we will define how the work for any successful **machine learning (ML)** software engineering project can be divided up. Basically, we will answer the question of how you *actually organize the doing* of a successful ML project. We will not only discuss the process and workflow but we will also set up the tools you will need for each stage of the process and highlight some important best practices with real ML code examples.

In this edition, there will be more details on an important data science and ML project management methodology: **Cross-Industry Standard Process for Data Mining (CRISP-DM)**. This will include a discussion of how this methodology compares to traditional Agile and Waterfall methodologies and will provide some tips and tricks for applying it to your ML projects. There are also far more detailed examples to help you get up and running with **continuous integration/continuous deployment (CI/CD)** using GitHub Actions, including how to run ML-focused processes such as automated model validation. The advice on getting up and running in an **Interactive Development Environment (IDE)** has also been made more tool-agnostic, to allow for those using any appropriate IDE. As before, the chapter will focus heavily on a “four-step” methodology I propose that encompasses a *discover, play, develop, deploy* workflow for your ML

projects. This project workflow will be compared with the CRISP-DM methodology, which is very popular in data science circles. We will also discuss the appropriate development tooling and its configuration and integration for a successful project. We will also cover version control strategies and their basic implementation, and setting up CI/CD for your ML project. Then, we will introduce some potential execution environments as the target destinations for your ML solutions. By the end of this chapter, you will be set up for success in your Python ML engineering project. This is the foundation on which we will build everything in subsequent chapters.

As usual, we will conclude the chapter by summarizing the main points and highlighting what this means as we work through the rest of the book.

Finally, it is also important to note that although we will frame the discussion here in terms of ML challenges, most of what you will learn in this chapter can also be applied to other Python software engineering projects. My hope is that the investment in building out these foundational concepts in detail will be something you can leverage again and again in all of your work.

We will explore all of this in the following sections and subsections:

- Setting up our tools
- Concept to solution in four steps:
 - Discover
 - Play
 - Develop
 - Deploy

There is plenty of exciting stuff to get through and lots to learn – so let's get started!

Technical requirements

As in *Chapter 1, Introduction to ML Engineering* if you want to run the examples provided here, you can create a Conda environment using the environment YAML file provided in the `Chapter02` folder of the book's GitHub repository:

```
conda env create -f mlewp-chapter02.yml
```

On top of this, many of the examples in this chapter will require the use of the following software and packages. These will also stand you in good stead for following the examples in the rest of the book:

- Anaconda
- PyCharm Community Edition, VS Code, or another Python-compatible IDE
- Git

You will also need the following:

- An Atlassian Jira account. We will discuss this more later in the chapter, but you can sign up for one for free at <https://www.atlassian.com/software/jira/free>.
- An AWS account. This will also be covered in the chapter, but you can sign up for an account at <https://aws.amazon.com/>. You will need to add payment details to sign up for AWS, but everything we do in this book will only require the free tier solutions.

The technical steps in this chapter were all tested on both a Linux machine running Ubuntu 22.04 LTS with a user profile that had admin rights and on a Macbook Pro M2 with the setup described in *Chapter 1, Introduction to*

ML Engineering. If you are running the steps on a different system, then you may have to consult the documentation for that specific tool if the steps do not work as planned. Even if this is the case, most of the steps will be the same, or very similar, for most systems. You can also check out all of the code for this chapter in the book's repository at <https://github.com/PacktPublishing/Machine-Learning-Engineering-with-Python-Second-Edition/tree/main/Chapter02>. The repo will also contain further resources for getting the code examples up and running.

Setting up our tools

To prepare for the work in the rest of this chapter, and indeed the rest of the book, it will be helpful to set up some tools. At a high level, we need the following:

- Somewhere to code
- Something to track our code changes
- Something to help manage our tasks
- Somewhere to provision infrastructure and deploy our solution

Let's look at how to approach each of these in turn:

- **Somewhere to code:** First, although the weapon of choice for coding by data scientists is of course Jupyter Notebook, once you begin to make the move toward ML engineering, it will be important to have an IDE to hand. An IDE is basically an application that comes with a series of built-in tools and capabilities to help you to develop the best software that you can. **PyCharm** is an excellent example for Python developers and comes with a wide variety of plugins, add-ons, and

integrations useful to ML engineers. You can download the Community Edition from JetBrains at <https://www.jetbrains.com/pycharm/>. Another popular development tool is the lightweight but powerful source code editor VS Code. Once you have successfully installed PyCharm, you can create a new project or open an existing one from the **Welcome to PyCharm** window, as shown in *Figure 2.1*:

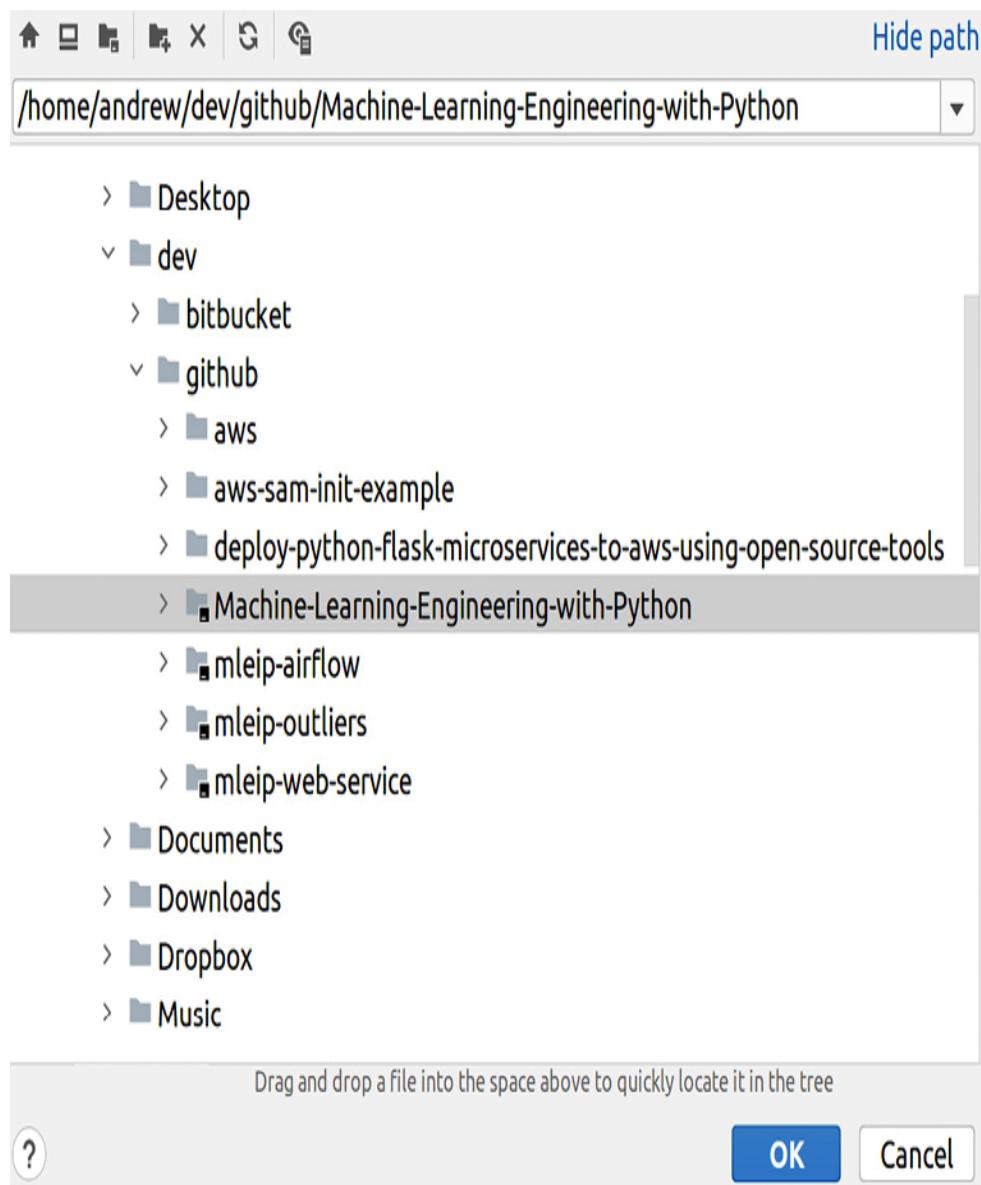


Figure 2.1: Opening or creating your PyCharm project.

- **Something to track code changes:** Next on the list is a code version control system. In this book, we will use **GitHub** but there are a variety of solutions, all freely available, that are based on the same underlying open-source **Git** technology. Later sections will discuss how to use these as part of your development workflow, but first, if you do not have a version control system set up, you can navigate to github.com and create a free account. Follow the instructions on the site to create your first repository, and you will be shown a screen that looks something like *Figure 2.2*. To make your life easier later, you should select **Add a README file** and **Add .gitignore** (then select **Python**). The README file provides an initial Markdown file for you to get started with and somewhere to describe your project. The **.gitignore** file tells your Git distribution to ignore certain types of files that in general are not important for version control. It is up to you whether you want the repository to be public or private and what license you wish to use. The repository for this book uses the **MIT license**:

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository](#).

Owner * Repository name *

 AndyMc629 /

Great repository names are short and memorable. Need inspiration? How about **bookish-goggles**?

Description (optional)

 **Public**
Anyone on the internet can see this repository. You choose who can commit.

 **Private**
You choose who can see and commit to this repository.

Initialize this repository with:
Skip this step if you're importing an existing repository.

Add a README file
This is where you can write a long description for your project. [Learn more](#).

Add .gitignore
Choose which files not to track from a list of templates. [Learn more](#).

Figure 2.2: Setting up your GitHub repository.

Once you have set up your IDE and version control system, you need to make them talk to each other by using the Git plugins provided with PyCharm. This is as simple as navigating to **VCS | Enable Version Control Integration** and selecting **Git**. You can edit the version

control settings by navigating to **File | Settings | Version Control**; see *Figure 2.3*:

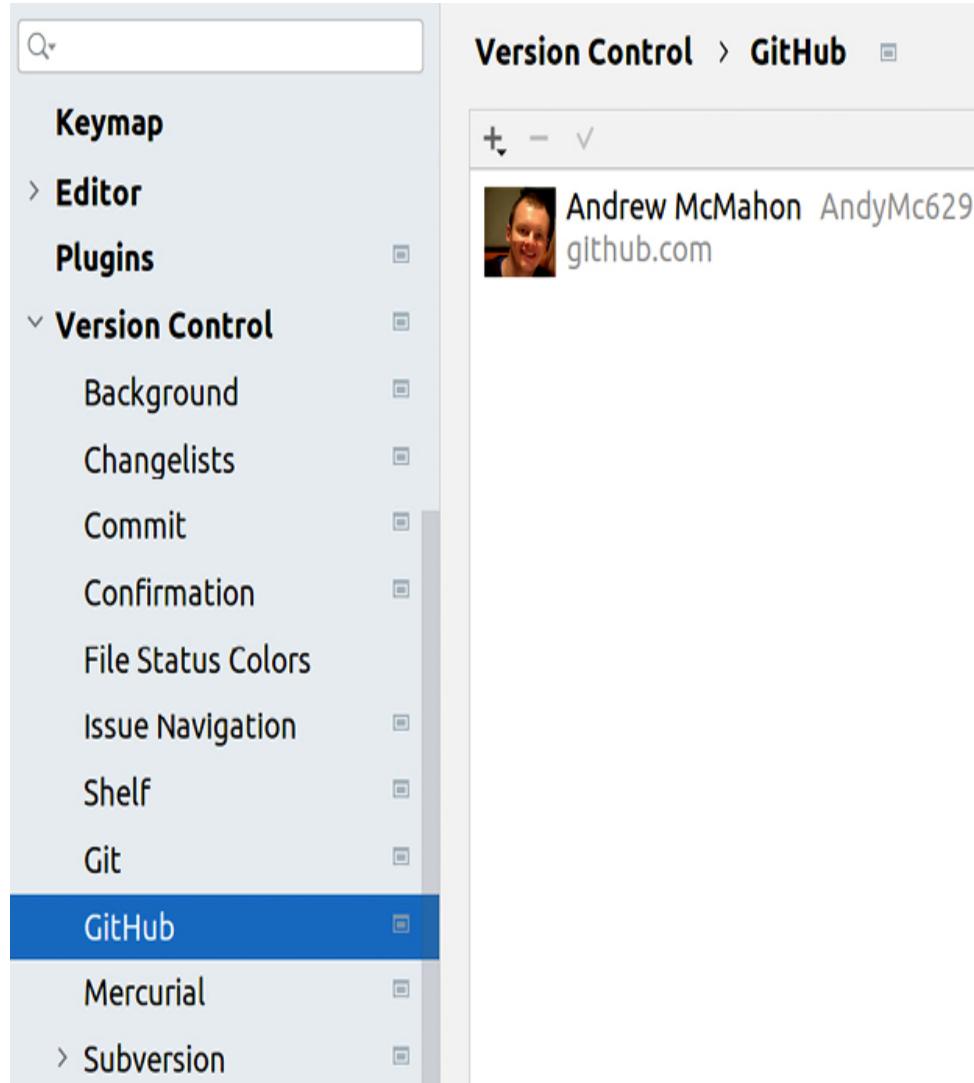


Figure 2.3: Configuring version control with PyCharm.

- **Something to help manage our tasks:** You are now ready to write Python and track your code changes, but are you ready to manage or participate in a complex project with other team members? For this, it is often useful to have a solution where you can track tasks, issues, bugs, user stories, and other documentation and items of work. It also helps if this has good integration points with the other tools you will

use. In this book, we will use **Jira** as an example of this. If you navigate to <https://www.atlassian.com/software/jira>, you can create a free cloud Jira account and then follow the interactive tutorial within the solution to set up your first board and create some tasks. *Figure 2.4* shows the task board for this book project, called **Machine Learning Engineering in Python (MEIP)**:

The screenshot shows a Jira task board titled "MEIP board". At the top, there are search and filter icons, followed by a "Type" dropdown. The board is divided into three columns: "TO DO 4 ISSUES", "IN PROGRESS 5 ISSUES", and "DONE".

TO DO 4 ISSUES	IN PROGRESS 5 ISSUES	DONE ✓
As a store demand planner, I want to see forecasts for items split by region to anticipate extra orders that need to be made MEIP-8	As a store demand planner, I want to be able to trigger retraining of models I think are out of date to improve forecast performance MEIP-9	See all Done issues
Add DBSCAN functionality to DetectionModels in outliers package MEIP-26	Build forecasting algorithm: build basic prophet algorithm (use code already developed) MEIP-6	

Figure 2.4: The task board for this book in Jira.

- **Somewhere to provision infrastructure and deploy our solution:** Everything that you have just installed and set up is tooling that will really help take your workflow and software development practices to the next level. The last piece of the puzzle is having the tools,

technologies, and infrastructure available for deploying the end solution. The management of computing infrastructure for applications was (and often still is) the provision of dedicated infrastructure teams, but with the advent of public clouds, there has been real democratization of this capability for people working across the spectrum of software roles. In particular, modern ML engineering is very dependent on the successful implementation of cloud technologies, usually through the main public cloud providers such as **Amazon Web Services (AWS)**, **Microsoft Azure**, or **Google Cloud Platform (GCP)**. This book will utilize tools found in the AWS ecosystem, but all of the tools and techniques you will find here have equivalents in the other clouds.

The flip side of the democratization of capabilities that the cloud brings is that teams who own the deployment of their solutions have to gain new skills and understanding. I am a strong believer in the principle that “*you build it, you own it, you run it*” as far as possible, but this means that as an ML engineer, you will have to be comfortable with a host of potential new tools and principles, as well as *owning* the performance of your deployed solution. *With great power comes great responsibility* and all that. In *Chapter 5, Deployment Patterns and Tools*, we will dive into this topic in detail.

Let’s talk through setting this up.

Setting up an AWS account

As previously stated, you don’t have to use AWS, but that’s what we’re going to use throughout this book. Once it’s set up here, you can use it for everything we’ll do:

1. To set up an AWS account, navigate to aws.amazon.com and select **Create Account**. You will have to add some payment details but everything we mention in this book can be explored through the *free tier* of AWS, where you do not incur a cost below a certain threshold of consumption.
2. Once you have created your account, you can navigate to the AWS Management Console, where you can see all the services that are available to you (see *Figure 2.5*):

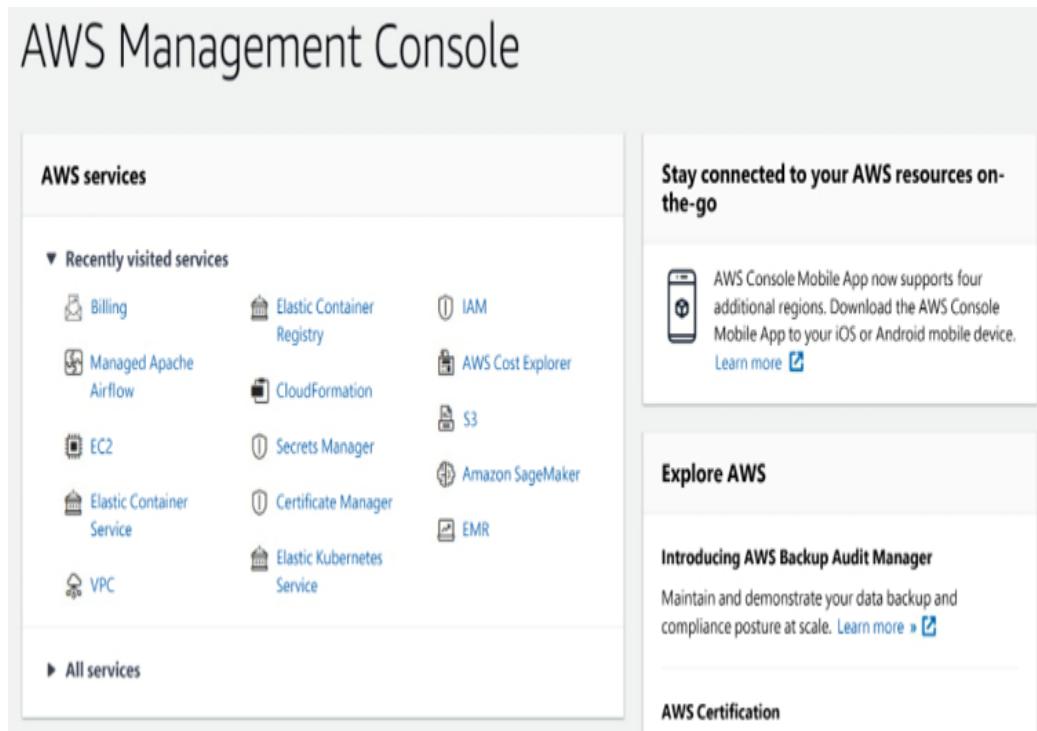


Figure 2.5: The AWS Management Console.

With our AWS account ready to go, let's look at the four steps that cover the whole process.

Concept to solution in four steps

All ML projects are unique in some way: the organization, the data, the people, and the tools and techniques employed will never be exactly the same for any two projects. This is good, as it signifies progress as well as the natural variety that makes this such a fun space to work in.

That said, no matter the details, broadly speaking, all successful ML projects actually have a good deal in common. They require the translation of a business problem into a technical problem, a lot of research and understanding, proofs of concept, analyses, iterations, the consolidation of work, the construction of the final product, and its deployment to an appropriate environment. That is ML engineering in a nutshell!

Developing this a bit further, you can start to bucket these activities into rough categories or stages, the results of each being necessary inputs for later stages. This is shown in *Figure 2.6*:

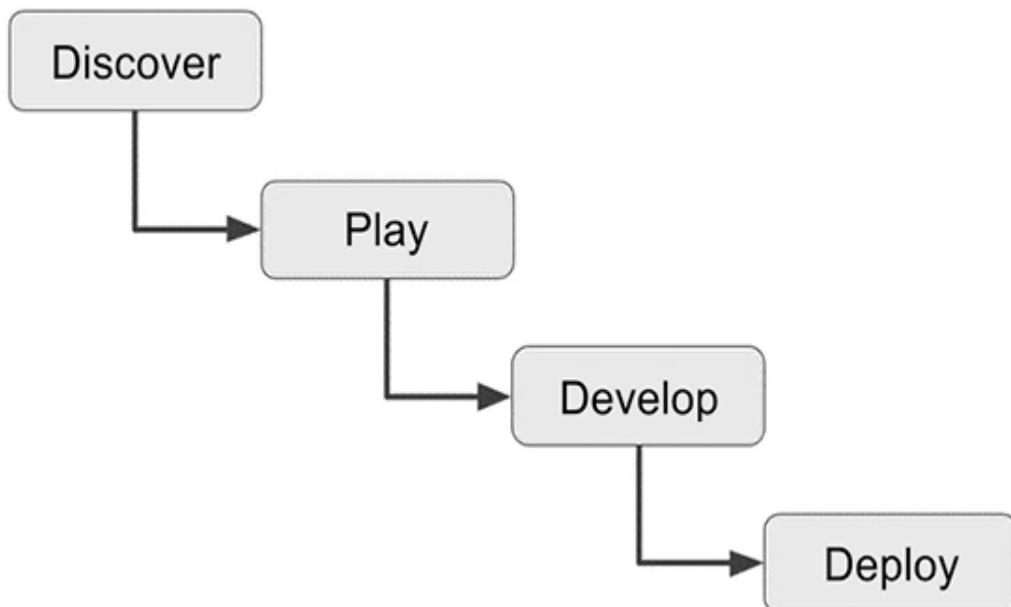


Figure 2.6: The stages that any ML project goes through as part of the ML development process.

Each category of work has a slightly different flavor, but taken together, they provide the backbone of any good ML project. The next few sections will develop the details of each of these categories and begin to show you how they can be used to build your ML engineering solutions. As we will discuss later, it is also not necessary for you to tackle your entire project in four steps like this; you can actually work through each of these steps for a specific feature or part of your overall project. This will be covered in the *Selecting a software development methodology* section.

Let's make this a bit more real. The main focus and outputs of every stage can be summarized as shown in *Table 2.1*:

Stage	Outputs
Discover	Clarity on the business question. Clear arguments for ML over another approach. Definition of the KPIs and metrics you want to optimize. A sketch of the route to value.
Play	Detailed understanding of the data. Working proof of concept. Agreement on the model/algorithim/logic that will solve the problem. Evidence that a solution is doable within realistic resource scenarios.

	Evidence that good ROI can be achieved.
Develop	<p>A working solution that can be hosted on appropriate and available infrastructure.</p> <p>Thorough test results and performance metrics (for algorithms and software).</p> <p>An agreed retraining and model deployment strategy.</p> <p>Unit tests, integration tests, and regression tests.</p> <p>Solution packaging and pipelines.</p>
Deploy	<p>A working and tested deployment process.</p> <p>Provisioned infrastructure with appropriate security and performance characteristics.</p> <p>Mode retraining and management processes.</p> <p>An end-to-end working solution!</p>

Table 2.1: The outputs of the different stages of the ML development process.



IMPORTANT NOTE

You may think that an ML engineer only really needs to consider the latter two stages, *develop*, and *deploy*, and that earlier stages are owned by the data scientist or even a business analyst. We will indeed focus mainly on these stages throughout this book and this division of labor can work very well. It is, however, crucially important that if you are going to build an ML solution, you understand all of

the motivations and development steps that have gone before – you wouldn't build a new type of rocket without understanding where you want to go first, would you?

Comparing this to CRISP-DM

The high-level categorization of project steps that we will outline in the rest of this chapter has many similarities to, and some differences from, an important methodology known as CRISP-DM. This methodology was published in 1999 and has since gathered a large following as a way to understand how to build any data project. In CRISP-DM, there are six different phases of activity, covering similar ground to that outlined in the four steps described in the previous section:

1. **Business understanding:** This is all about getting to know the business problem and domain area. This becomes part of the *Discover* phase in the four-step model.
2. **Data understanding:** Extending the knowledge of the business domain to include the state of the data, its location, and how it is relevant to the problem. Also included in the *Discover* phase.
3. **Data preparation:** Starting to take the data and transform it for downstream use. This will often have to be iterative. Captured in the *Play* stage.
4. **Modeling:** Taking the prepared data and then developing analytics on top of it; this could now include ML of various levels of sophistication. This is an activity that occurs both in the *Play* and *Develop* phases of the four-step methodology.
5. **Evaluation:** This stage is concerned with confirming whether the solution will meet the business requirements and performing a holistic

review of the work that has gone before. This helps confirm if anything was overlooked or could be improved upon. This is very much part of the *Develop* and *Deploy* phases; in the methodology we will describe in this chapter, these tasks are very much more baked in across the project.

6. Deployment: In CRISP-DM, this was originally focused on deploying simple analytics solutions like dashboards or scheduled ETL pipelines that would run the decided-upon analytics models.

In the world of model ML engineering, this stage can represent, well, anything talked about in this book! CRISP-DM suggests sub-stages around planning and then reviewing the deployment.

As you can see from the list, many steps in CRISP-DM cover similar topics to those outlined in the four steps I propose. CRISP-DM is extremely popular across the data science community and so its merits are definitely appreciated by a huge number of data professionals across the world. Given this, you might be wondering, “Why bother developing something else then?” Let me convince you of why this is a good idea.

The CRISP-DM methodology is just another way to group the important activities of any data project in order to give them some structure. As you can perhaps see from the brief description of the stages I gave above and if you do further research, CRISP-DM has some potential drawbacks for use in a modern ML engineering project:

- The process outlined in CRISP-DM is relatively rigid and quite linear. This can be beneficial for providing structure but might inhibit moving fast in a project.
- The methodology is very big on documentation. Most steps detail writing some kind of report, review, or summary. Writing and

maintaining good documentation is absolutely critical in a project but there can be a danger of doing too much.

- CRISP-DM was written in a world before “big data” and large-scale ML. It is unclear to me whether its details still apply in such a different world, where classic extract-transform-load patterns are only one of so many.
- CRISP-DM definitely comes from the data world and then tries to move toward the idea of a deployable solution in the last stage. This is laudable, but in my opinion, this is not enough. ML engineering is a different discipline in the sense that it is far closer to classic software engineering than not. This is a point that this book will argue time and again. It is therefore important to have a methodology where the concepts of deployment and development are aligned with software and modern ML techniques all the way through.

The *four-step* methodology attempts to alleviate some of these challenges and does so in a way that constantly makes reference to software engineering and ML skills and techniques. This does not mean that you should never use CRISP-DM in your projects; it might just be the perfect thing! As with many of the concepts introduced in this book, the important thing is to have many tools in your toolkit so that you can select the one most appropriate for the job at hand.

Given this, let’s now go through the four steps in detail.

Discover

Before you start working to build any solution, it is vitally important that you understand the problem you are trying to solve. This activity is often

termed **discovery** in business analysis and is crucial if your ML project is going to be a success.

The key things to do during the discovery phase are the following:

- *Speak to the customer! And then speak to them again:* You must understand the end user requirements in detail if you are to design and build the right system.
- *Document everything:* You will be judged on how well you deliver against the requirements, so make sure that all of the key points from your discussion are documented and signed off by members of your team and the customer or their appropriate representative.
- *Define the metrics that matter:* It is very easy at the beginning of a project to get carried away and to feel like you can solve any and every problem with the amazing new tool you are going to build. Fight this tendency as aggressively as you can, as it can easily cause major headaches later on. Instead, steer your conversations toward defining a single or very small number of metrics that define what success will look like.
- *Start finding out where the data lives!: If you can start working out what kind of systems you will have to access to get the data you need, this saves you time later and can help you find any major issues before they derail your project.*

Using user stories

Once you have spoken to the customer (a few times), you can start to define some **user stories**. User stories are concise and consistently formatted expressions of what the user or customer wants to see and the acceptance criteria for that feature or unit of work. For example, we may want to define

a user story based on the taxi ride example from *Chapter 1, Introduction to ML Engineering*: “As a user of our internal web service, I want to see anomalous taxi rides and be able to investigate them further.”

Let’s begin!

1. To add this in Jira, select the **Create** button.
2. Next, select **Story**.
3. Then, fill in the details as you deem appropriate.

You have now added a user story to your work management tool! This allows you to do things such as create new tasks and link them to this user story or update its status as your project progresses:

As a taxi ride analyst, I want to see anomalous journeys so that I can tailor our offers to customers



Description

Acceptance criteria (scenario):

Given I have access to data and/or visualizations of ML results,

where there are anomalous taxi rides,

then the system accurately identifies and labels these rides and I can see them

GitHub integration for Jira

[Branches](#) [Pull Requests](#) [Commits](#) [Tags](#)

Figure 2.7: An example user story in Jira.

The data sources you use are particularly crucial to understand. As you know, *garbage in, garbage out*, or even worse, *no data, no go!* The particular questions you have to answer about the data are mainly centered around **access, technology, quality, and relevance**.

For access and technology, you are trying to pre-empt how much work the data engineers have to do to start their pipeline of work and how much this will hold up the rest of the project. It is therefore crucial that you get this one right.

A good example would be if you find out quite quickly that the main bulk of data you will need lives in a legacy internal financial system with no real modern APIs and no access request mechanism for non-finance team members. If its main backend is on-premises and you need to migrate locked-down financial data to the cloud, but this makes your business nervous, then you know you have a lot of work to do before you type a line of code. If the data already lives in an enterprise data lake that your team has access to, then you are obviously in a better position. Any challenge is surmountable if the value proposition is strong enough, but finding all this out early will save you time, energy, and money later on.

Relevance is a bit harder to find out before you kick off, but you can begin to get an idea. For example, if you want to perform the inventory forecast we discussed in *Chapter 1, Introduction to ML Engineering*, do you need to pull in customer account information? If you want to create the classifier of *premium* or *non-premium* customers as marketing targets, also mentioned in *Chapter 1, Introduction to ML Engineering*, do you need to have data on social media feeds? The question as to what is relevant will often be less clear-cut than for these examples but an important thing to remember is that you can always come back to it if you really missed something important. You are trying to capture the most important design decisions early, so common sense and lots of stakeholder and subject-matter expert engagement will go a long way.

Data quality is something that you can try to anticipate a little before moving forward in your project with some questions to current users or consumers of the data or those involved in its entry processes. To get a more quantitative understanding though, you will often just need to get your data scientists working with the data in a hands-on manner.

In the next section, we will look at how we develop proof-of-concept ML solutions in the most research-intensive phase, *Play*.

Play

In the **play** stage of the project, your aim is to work out whether solving the task even at the proof-of-concept level is feasible. To do this, you might employ the usual data science bread-and-butter techniques of exploratory data analysis and explanatory modeling we mentioned in the last chapter before moving on to creating an ML model that does what you need.

In this part of the process, you are not overly concerned with details of implementation, but with exploring the realms of possibility and gaining an in-depth understanding of the data and the problem, which goes beyond initial discovery work. Since the goal here is not to create *production-ready* code or to build reusable tools, you should not worry about whether or not the code you are writing is of the highest quality, or using sophisticated patterns. For example, it will not be uncommon to see code that looks something like the following examples (taken, in fact, from the repo for this book):

Prep for Prophet

```
df.rename(columns= {'Datetime': 'ds', 'AEP_MW': 'y'}, inplace=True)
```

```
df['ds']=df['ds'].astype('datetime64[ns]')
```

```
df.dtypes
```

```
#Initialize Split Class, we'll split our data 5 times for cv
ts_splits = TimeSeriesSplit(n_splits=5)
```

Train and Forecast

```
tmp = time_split_train_test(df.sort_values('ds', ascending=True).iloc[-1000:], ts_splits)
```

```
tmp.head()
```

Plot

```
nrow = 5; ncol = 1;
fig, axs = plt.subplots(nrows=nrow, ncols=ncol, figsize=(20,30))
fig.subplots_adjust(hspace=0.4, wspace=0.4)
for i, ax in enumerate(fig.axes):
    split_rmse = tmp[(tmp['split']==i) & (tmp['train']==False)]['rmse'].iloc[0]
    ax.set_title('Split '+str(i)+ ' - RMSE: '+'{:.2f}'.format(split_rmse))

    tmp[(tmp['split']==i) & (tmp['train']==True)].plot(x='ds', y='y', ax=ax, color='blue', marker='o')
    tmp[(tmp['split']==i) & (tmp['train']==False)].plot(x='ds', y='y', ax=ax, color='red', marker='o')
    tmp[(tmp['split']==i) & (tmp['train']==False)].plot(x='ds', y='yhat', ax=ax, color='orange', marker='^')
```

Figure 2.8: Some example prototype code that will be created during the play stage.

Even a quick glance at these screenshots tells you a few things:

- The code is in a Jupyter notebook, which is run by a user interactively in a web browser.
- The code sporadically calls methods to simply check or explore elements of the data (for example, `df.head()` and `df.dtypes`).
- There is ad hoc code for plotting (and it's not very intuitive!).
- There is a variable called `tmp`, which is not very descriptive.

All of this is absolutely fine in this more exploratory phase, but one of the aims of this book is to help you understand what is required to take code like this and make it into something suitable for your production ML pipelines. The next section starts us along this path.

Develop

As we have mentioned a few times already, one of the aims of this book is to get you thinking about the fact that you are building software products that just happen to have ML in them. This means a steep learning curve for some of us who have come from more mathematical and algorithmic backgrounds. This may seem intimidating but do not despair! The good news is that we can reuse a lot of the best practices and techniques honed through the software engineering community over several decades. There is nothing new under the sun.

This section explores several of those methodologies, processes, and considerations that can be employed in the development phase of our ML engineering projects.

Selecting a software development methodology

One of the first things we could and should shamelessly replicate as ML engineers is the software development methodologies that are utilized in projects across the globe. One category of these, often referred to as **Waterfall**, covers project workflows that fit quite naturally with the idea of building something complex (think a building or a car). In Waterfall methodologies, there are distinct and sequential phases of work, each with a clear set of outputs that are needed before moving on to the next phase. For

example, a typical Waterfall project may have phases that broadly cover requirements-gathering, analysis, design, development, testing, and deployment (sound familiar?). The key thing is that in a Waterfall-flavored project, when you are in the *requirements-gathering* phase, you should *only* be working on gathering requirements, when in the testing phase, you should *only* be working on testing, and so on. We will discuss the pros and cons of this for ML in the next few paragraphs after introducing another set of methodologies.

The other set of methodologies, termed **Agile**, began its life after the introduction of the **Agile Manifesto** in 2001

(<https://agilemanifesto.org/>). At the heart of Agile development are the ideas of flexibility, iteration, incremental updates, failing fast, and adapting to changing requirements. If you are from a research or scientific background, this concept of flexibility and adaptability based on results and new findings may sound familiar.

What may not be so familiar to you if you have this type of scientific or academic background is that you can still embrace these concepts within a relatively strict framework that is centered around delivery outcomes. Agile software development methodologies are all about finding the balance between experimentation and delivery. This is often done by introducing the concepts of **ceremonies** (such as **Scrums** and **Sprint Retrospectives**) and **roles** (such as **Scrum Master** and **Product Owner**).

Further to this, within Agile development, there are two variants that are extremely popular: **Scrum** and **Kanban**. Scrum projects are centered around short units of work called **Sprints** where the idea is to make additions to the product from ideation through to deployment in that small timeframe. In Kanban, the main idea is to achieve a steady **flow** of tasks

from an organized backlog into work in progress through to completed work.

All of these methodologies (and many more besides) have their merits and their detractions. You do not have to be married to any of them; you can chop and change between them. For example, in an ML project, it may make sense to do some *post-deployment* work that has a focus on maintaining an already existing service (sometimes termed a *business-as-usual* activity) such as further model improvements or software optimizations in a Kanban framework. It may make sense to do the main delivery of your core body of work in Sprints with very clear outcomes. But you can chop and change and see what fits best for your use cases, your team, and your organization.

But what makes applying these types of workflows to ML projects different? What do we need to think about in this world of ML that we didn't before? Well, some of the key points are the following:

- *You don't know what you don't know:* You cannot know whether you will be able to solve the problem until you have seen the data.
Traditional software engineering is not as critically dependent on the data that will flow through the system as ML engineering is. We can know how to solve a problem in principle, but if the appropriate data does not exist in sufficient quantity or is of poor quality, then we can't solve the problem in practice.
- *Your system is alive:* If you build a classic website, with its backend database, shiny frontend, amazing load-balancing, and other features, then realistically, if the resource is there, it can just run forever.
Nothing fundamental changes about the website and how it runs over time. Clicks still get translated into actions and page navigation still

happens the same way. Now, consider putting some ML-generated advertising content based on typical user profiles in there. What is a *typical user profile* and does that change with time? With more traffic and more users, do behaviors that we never saw before become *the new normal*? Your system is learning all the time and that leads to the problems of *model drift* and *distributional shift*, as well as more complex update and rollback scenarios.

- *Nothing is certain*: When building a system that uses rule-based logic, you know what you are going to get each and every time. *If X, then Y* means just that, always. With ML models, it is often much harder to know what the answer is when you ask the question, which is in fact why these algorithms are so powerful.

But it does mean that you can have unpredictable behavior, either for the reasons discussed previously or simply because the algorithm has learned something that is not obvious about the data to a human observer, or, because ML algorithms can be based on probabilistic and statistical concepts, results come attached to some uncertainty or *fuzziness*. A classic example is when you apply logistic regression and receive the probability of the data point belonging to one of the classes. It's a probability so you cannot say with certainty that it is the case; just how likely it is! This is particularly important to consider when the outputs of your ML system will be leveraged by users or other systems to make decisions.

Given these issues, in the next section, we'll try and understand what development methodologies can help us when we build our ML solutions. In *Table 2.2*, we can see some advantages and disadvantages of each of these Agile methodologies for different stages and types of ML engineering projects:

Methodology	Pros	Cons
Agile	<p>Flexibility is expected.</p> <p>Faster dev to deploy cycles.</p>	<p>If not well managed, can easily have scope drift.</p> <p>Kanban or Sprints may not work well for some projects.</p>
Waterfall	<p>Clearer path to deployment.</p> <p>Clear staging and ownership of tasks.</p>	<p>Lack of flexibility.</p> <p>Higher admin overheads.</p>

Table 2.2: Agile versus Waterfall for ML development.

Let's move on to the next section!

Package management (`conda` and `pip`)

If I told you to write a program that did anything in data science or ML without using any libraries or packages and just pure Python, you would probably find this quite difficult to achieve in any reasonable amount of time, and incredibly boring! This is a good thing. One of the really powerful features of developing software in Python is that you can leverage an extensive ecosystem of tools and capabilities relatively easily. The flip side of this is that it would be very easy for managing the dependencies of your code base to become a very complicated and hard-to-replicate task. This is where package and environment managers such as `pip` and `conda` come in.

`pip` is the standard package manager in Python and the one recommended for use by the Python Package Authority.

It retrieves and installs Python packages from `PyPI`, the `Python Package Index`. `pip` is super easy to use and is often the suggested way to install packages in tutorials and books.

`conda` is the *package and environment* manager that comes with the Anaconda and Miniconda Python distributions. A key strength of `conda` is that although it comes from the Python ecosystem, and it has excellent capabilities there, it is actually a more general package manager. As such, if your project requires dependencies outside Python (the NumPy and SciPy libraries being good examples), then although `pip` can install these, it can't track all the non-Python dependencies, nor manage their versions. With `conda`, this is solved.

You can also use `pip` within `conda` environments, so you can get the best of both worlds or use whatever you need for your project. The typical workflow that I use is to use `conda` to manage the environments I create and then use that to install any packages I think may require non-Python dependencies that perhaps are not captured well within `pip`, and then I can use `pip` most of the time within the created `conda` environment. Given this, throughout the book, you may see `pip` or `conda` installation commands used interchangeably. This is perfectly fine.

To get started with Conda, if you haven't already, you can download the **Individual** distribution installer from the Anaconda website (<https://www.anaconda.com/products/individual>).

Anaconda comes with some Python packages already installed, but if you want to start from a completely empty environment, you can download

Miniconda from the same website instead (they have the exact same functionality; you just start from a different base).

The Anaconda documentation is very helpful for getting you up to speed with the appropriate commands, but here is a quick tour of some of the key ones.

First, if we want to create a `conda` environment called `mleng` with Python version 3.8 installed, we simply execute the following in our terminal:

```
conda env --name mleng python=3.10
```

We can then activate the `conda` environment by running the following:

```
source activate mleng
```

This means that any new `conda` or `pip` commands will install packages in this environment and not system-wide.

We often want to share the details of our environment with others working on the same project, so it can be useful to export all the package configurations to a `.yml` file:

```
conda export env > environment.yml
```

The GitHub repository for this book contains a file called `mleng-environment.yml` for you to create your own instance of the `mleng`

environment. The following command creates an environment with this configuration using this file:

```
conda env create --file environment.yml
```

This pattern of creating a `conda` environment from an environment file is a nice way to get your environments set up for running the examples in each of the chapters in the book. So, the *Technical requirements* section in each chapter will point to the name of the correct environment YAML file contained in the book's repository.

These commands, coupled with your classic `conda` or `pip install` command, will set you up for your project quite nicely!

```
conda install <package-name>
```

Or

```
pip install <package-name>
```

I think it's always a good practice to have many options for doing something, and in general, this is good engineering practice. So given that, now that we have covered the classic Python environment and package managers in `conda` and `pip`, we will cover one more package manager. This is a tool that I like for its ease of use and versatility. I think it provides a nice extension of the capabilities of `conda` and `pip` and can be used to complement them nicely. This tool is called Poetry and it is what we turn to now.

Poetry

Poetry is another package manager that has become very popular in recent years. It allows you to manage your project’s dependencies and package information into a single configuration file in a similar way to the environment YAML file we discussed in the section on Conda. Poetry’s strength lies in its far superior ability to help you manage complex dependencies and ensure “deterministic” builds, meaning that you don’t have to worry about the dependency of a package updating in the background and breaking your solution. It does this via the use of “lock files” as a core feature, as well as in-depth dependency checking. This means that reproducibility can often be easier in Poetry. It is important to call out that Poetry is focused on Python package management specifically, while Conda can also install and manage other packages, for example, C++ libraries. One way to think of Poetry is that it is like an upgrade of the `pip` Python installation package, but one that also has some environment management capability. The next steps will explain how to set up and use Poetry for a very basic use case.

We will build on this with some later examples in the book. First, follow these steps:

1. First, as usual, we will install Poetry:

```
pip install poetry
```

2. After Poetry is installed, you can create a new project using the `poetry new` command, followed by the name of your project:

```
poetry new mlen-with-python
```

3. This will create a new directory named `mleng-with-python` with the necessary files and directories for a Python project. To manage your project's dependencies, you can add them to the `pyproject.toml` file in the root directory of your project. This file contains all of the configuration information for your project, including its dependencies and package metadata.

For example, if you are building a ML project and want to use the `scikit-learn` library, you would add the following to your `pyproject.toml` file:

```
[tool.poetry.dependencies]
scikit-learn = "*"
```

4. You can then install the dependencies for your project by running the following command. This will install the `scikit-learn` library and any other dependencies specified in your `pyproject.toml` file:

```
poetry install
```

5. To use a dependency in your project, you can simply import it in your Python code like so:

```
from sklearn import datasets
from sklearn.model_selection import train_te
from sklearn.linear_model import LogisticRe
```

As you can see, getting started with Poetry is very easy. We will return to using Poetry throughout the book in order to give you examples that complement the knowledge of Conda that we will develop. *Chapter 4, Packaging Up*, will discuss this in detail and will show you how to get the most out of Poetry.

Code version control

If you are going to write code for real systems, you are almost certainly going to do it as part of a team. You are also going to make your life easier if you can have a clean audit trail of changes, edits, and updates so that you can see how the solution has developed. Finally, you are going to want to cleanly and safely separate out the stable versions of the solution that you are building and that can be deployed versus more transient developmental versions. All of this, thankfully, is taken care of by source code version control systems, the most popular of which is **Git**.

We will not go into how Git works under the hood here (there are whole books on the topic!) but we will focus on understanding the key practical elements of using it:

1. You already have a GitHub account from earlier in the chapter, so the first thing to do is to create a repository with Python as the language and initialize `README.md` and `.gitignore` files. The next thing to do is to get a local copy of this repository by running the following command in Bash, Git Bash, or another terminal:

```
git clone <repo-name>
```

2. Now that you have done this, go into the `README.md` file and make some edits (anything will do). Then, run the following commands to tell Git to *monitor* this file and to save your changes locally with a message briefly explaining what these are:

```
git add README.md  
git commit -m "I've made a nice change ..."
```

This now means that your local Git instance has stored what you've changed and is ready to share that with the remote repo.

3. You can then incorporate these changes into the `main` branch by doing the following:

```
git push origin main
```

If you now go back to the GitHub site, you will see that the changes have taken place in your remote repository and that the comments you added have accompanied the change.

4. Other people in your team can then get the updated changes by running the following:

```
git pull origin main
```

These steps are the absolute basics of Git and there is a ton more you can learn online. What we will do now, though, is start setting up our repo and workflow in a way that is relevant to ML engineering.

Git strategies

The presence of a strategy for using version control systems can often be a key differentiator between the data science and ML engineering aspects of a project. It can sometimes be overkill to define a strict Git strategy for exploratory and basic modeling stages (*Discover* and *Play*) but if you want to engineer something for deployment (and you are reading this book, so this is likely where your head is at), then it is fundamentally important.

Great, but what do we mean by a Git strategy?

Well, let's imagine that we just try to develop our solution without a shared direction on how to organize the versioning and code.

ML engineer A wants to start building some of the data science code into a Spark ML pipeline (more on this later) so creates a branch from `main` called `pipeline1spark`:

```
git checkout -b pipeline1spark
```

They then get to work on the branch and writes some nice code in a new file called `pipeline.py`:

```
# Configure an ML pipeline, which consists of t
tokenizer = Tokenizer(inputCol="text", outputCo
hashingTF = HashingTF(inputCol=tokenizer.getOut
                           outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.
pipeline = Pipeline(stages=[tokenizer, hashingT
```

Great, they've made some excellent progress in translating some previous `sklearn` code into Spark, which was deemed more appropriate for the use case. They then keep working in this branch because it has all of their additions, and they think it's better to do everything in one place. When they want to push the branch to the remote repository, they run the following commands:

```
git push origin pipeline1spark
```

ML engineer *B* comes along, and they want to use ML engineer *A*'s pipeline code and build some extra steps around it. They know engineer *A*'s code has a branch with this work, so they know enough about Git to create another branch with *A*'s code in it, which *B* calls `pipeline`:

```
git pull origin pipeline1spark  
git checkout pipeline1spark  
git checkout -b pipeline
```

They then add some code to read the parameters for the model from a variable:

```
lr = LogisticRegression(maxIter=model_config["m  
regParam=model_config[""
```

Cool, engineer *B* has made an update that is starting to abstract away some of the parameters. They then push their new branch to the remote repository:

```
git push origin pipeline
```

Finally, ML engineer C joins the team and wants to get started on the code. Opening up Git and looking at the branches, they see there are three:

```
main  
pipeline1spark  
pipeline
```

So, which one should be taken as the most up to date? If they want to make new edits, where should they branch from? It isn't clear, but more dangerous than that is if they are tasked with pushing deployment code to the execution environment, they may think that `main` has all the relevant changes. On a far busier project that's been going on for a while, they may even branch off from `main` and duplicate some of *B* and *C*'s work! In a small project, you would waste time going on this wild goose chase; in a large project with many different lines of work, you would have very little chance of maintaining a good workflow:

```
# Branch pipeline1spark - Commit 1 (Engineer A)  
lr = LogisticRegression(maxIter=10, regParam=0.  
pipeline = Pipeline(stages=[tokenizer, hashingT  
# Branch pipeline - Commit 2 (Engineer B)  
lr = LogisticRegression(maxIter=model_config["m  
                                regParam=model_config["  
pipeline = Pipeline(stages=[tokenizer, hashingT
```

If these commits both get pushed to the `main` branch at the same time, then we will get what is called a **merge conflict**, and in each case, the engineer will have to choose which piece of code to keep, the current or new example. This would look something like this if engineer A pushed their changes to `main` first:

```
<<<<< HEAD
lr = LogisticRegression(maxIter=10, regParam=0.
pipeline = Pipeline(stages=[tokenizer, hashingT
=====
lr = LogisticRegression(maxIter=model_config["m
                    regParam=model_config["m
pipeline = Pipeline(stages=[tokenizer, hashingT
>>>>> pipeline
```

The delimiters in the code show that there has been a merge conflict and that it is up to the developer to select which of the two versions of the code they want to keep.



IMPORTANT NOTE

Although, in this simple case, we could potentially trust the engineers to select the *better* code, allowing situations like this to occur very frequently is a huge risk to your project. This not only wastes a huge amount of precious development time but it could also mean that you actually end up with worse code!

The way to avoid confusion and extra work like this is to have a very clear strategy for the use of the version control system in place, such as the one we will now explore.

The Gitflow workflow

The biggest problem with the previous example was that all of our hypothetical engineers were actually working on the same piece of code in different places. To stop situations like this, you have to create a process that your team can all follow – in other words, a version control strategy or workflow.

One of the most popular of these strategies is the **Gitflow workflow**. This builds on the basic idea of having branches that are dedicated to features and extends it to incorporate the concept of releases and hotfixes, which are particularly relevant to projects with a continuous deployment element.

The main idea is we have several types of branches, each with clear and specific reasons for existing:

- **Main** contains your official releases and should only contain the stable version of your code.
- **Dev** acts as the main point for branching from and merging to for most work in the repository; it contains the ongoing development of the code base and acts as a staging area before **main**.
- **Feature** branches should not be merged straight into the **main** branch; everything should branch off from **dev** and then be merged back into **dev**.
- **Release** branches are created from **dev** to kick off a build or release process before being merged into **main** and **dev** and then deleted.

- **Hotfix** branches are for removing bugs in deployed or production software. You can branch this from `main` before merging into `main` and `dev` when done.

This can all be summarized diagrammatically as in *Figure 2.9*, which shows how the different branches contribute to the evolution of your code base in the Gitflow workflow:

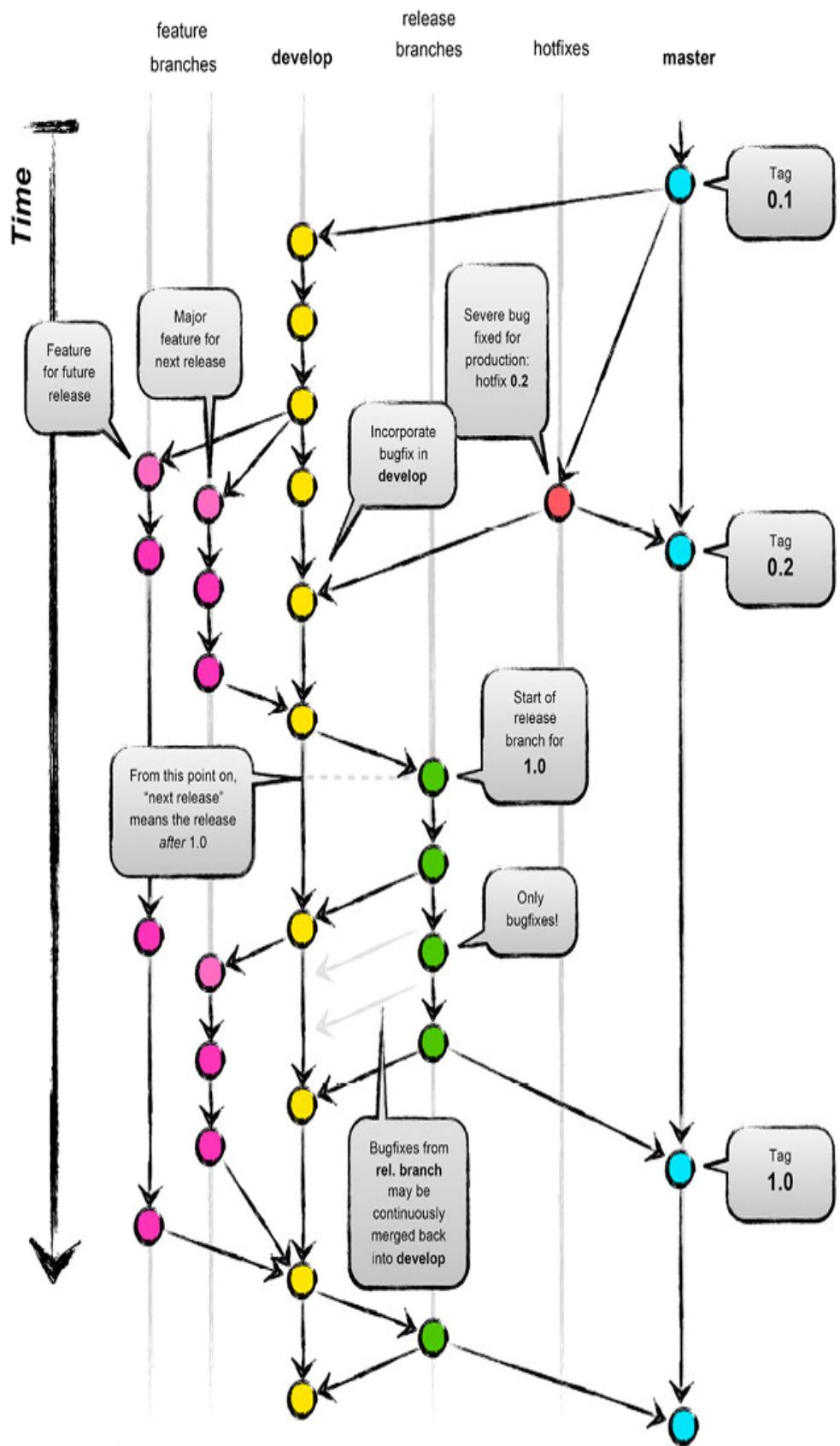


Figure 2.9: The Gitflow workflow.

This diagram is taken from

<https://lucamezzalira.com/2014/03/10/git-flow-vs-github-flow/>. More details can be found at <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>.

If your ML project can follow this sort of strategy (and you don't need to be completely strict about this if you want to adapt it), you will likely see a drastic improvement in productivity, code quality, and even documentation:

```

v ⇩ 9 chapter1/bad-git/pipeline.py ⌂
... ... @@ -1,9 +1,14 @@
1 - # EXAMPLE BELOW TAKEN FROM THE SPARK API DOCS, BEFORE BEING UPDATE FOR THE BOOK
1 + # EXAMPLE BELOW ADAPTED FROM THE SPARK DOCS
2 2 # https://spark.apache.org/docs/latest/ml-pipeline.html#pipeline
3 3 from pyspark.ml import Pipeline
4 4 from pyspark.ml.classification import LogisticRegression
5 5 from pyspark.ml.feature import HashingTF, Tokenizer
6 6
7 + import json
8 +
9 + with open("model_config.json") as f:
10 |     model_config = json.load(f)
11 +
12 # Prepare training documents from a list of (id, text, label) tuples.
13 training = spark.createDataFrame([
14     (0, "a b c d e spark", 1.0),
15 @@ -15,7 +20,7 @@
16     # Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.
17     tokenizer = Tokenizer(inputCol="text", outputCol="words")
18     hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
19 -     lr = LogisticRegression(maxIter=10, regParam=0.001)
20 +     lr = LogisticRegression(maxIter=model_config['maxIter'], regParam=model_config['regParam'])
21     pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

```

Figure 2.10: Example code changes upon a pull request in GitHub.

One important aspect we haven't discussed yet is the concept of code reviews. These are triggered in this process by what is known as a **pull request**, where you make known your intention to merge into another branch and allow another team member to review your code before this executes. This is the natural way to introduce code review to your workflow. You do this whenever you want to merge your changes and

update them into dev or main branches. The proposed changes can then be made visible to the rest of the team, where they can be debated and iterated on with further commits before completing the merge.

This enforces code review to improve quality, as well as creating an audit trail and safeguards for updates. *Figure 2.10* shows an example of how changes to code are made visible for debate during a pull request in GitHub.

Now that we have discussed some of the best practices for applying version control to your code, let's explore how to version control the models you produce during your ML project.

Model version control

In any ML engineering project, it is not only code changes that you have to track clearly but also changes in your models. You want to track changes not only in the modeling approach but also in performance when new or different data is fed into your chosen algorithms. One of the best tools for tracking these kinds of changes and providing version control of ML models is **MLflow**, an open-source platform from **Databricks** under the stewardship of the Linux Foundation.

To install MLflow, run the following command in your chosen Python environment:

```
pip install mlflow
```

The main aim of MLflow is to provide a platform via which you can log model experiments, artifacts, and performance metrics. It does this through some very simple APIs provided by the Python `mlflow` library, interfaced

to selected storage solutions through a series of centrally developed and community plugins. It also comes with functionality for querying, analyzing, and importing/exporting data via a **Graphical User Interface (GUI)**, which will look something like *Figure 2.11*:

The screenshot shows the MLflow tracking server interface. At the top, there's a navigation bar with 'mlflow' logo, 'Experiments', 'Models', 'GitHub', and 'Docs'. Below the navigation bar, the 'Experiments' tab is selected. A sidebar on the left has 'Default' selected. The main area shows experimental details: 'Experiment ID: 0' and 'Artifact Location: file:///home/andrew/dev/github/Machine-Learning-Engineering-with-Python/chapter1/mlflow/mlruns/0'. There's a 'Notes' section with 'None'. Below that is a search bar with the query 'metrics.rmse < 1 and params.model = "tree" and tags.mlflow.source.type = "LOCAL"'. The results table shows three matching runs:

	Start Time	Run Name	User	Source	Version	Models	param1	foo	rmse
<input type="checkbox"/>	2021-02-22 13:12:57	-	andrew	mlflow_fore	c6790e	-	-	-	4.853
<input type="checkbox"/>	2021-02-22 10:23:15	-	andrew	mlflow_fore	c6790e	-	-	-	4.853
<input type="checkbox"/>	2021-02-17 19:23:53	-	andrew	mlflow_bas	ed8a02	-	20	2.762	-

Buttons for 'Compare', 'Delete', and 'Download CSV' are available above the table. To the right of the table are 'Parameters' and 'Metrics' filters, and 'Columns' selection buttons. A 'Load more' button is at the bottom of the table.

Figure 2.11: The MLflow tracking server UI with some forecasting runs.

The library is extremely easy to use. In the following example, we will take the sales forecasting example from *Chapter 1, Introduction to ML Engineering*, and add some basic MLflow functionality for tracking performance metrics and saving the trained Prophet model:

1. First, we make the relevant imports, including MLflow's `pyfunc` module, which acts as a general interface for saving and loading models that can be written as Python functions. This facilitates working with libraries and tools not natively supported in MLflow (such as the `fbprophet` library):

```
import pandas as pd
from fbprophet import Prophet
from fbprophet.diagnostics import cross_vali
from fbprophet.diagnostics import performance
import mlflow
import mlflow.pyfunc
```

2. To create a more seamless integration with the forecasting models from `fbprophet`, we define a small wrapper class that inherits from the `mlflow.pyfunc.PythonModel` object:

```
class FbProphetWrapper(mlflow.pyfunc.PythonModel):
    def __init__(self, model):
        self.model = model
        super().__init__()
    def load_context(self, context):
        from fbprophet import Prophet
        return
    def predict(self, context, model_input):
        future = self.model.make_future_dataframe(
            periods=model_input["periods"])[C]
        return self.model.predict(future)
```

We now wrap the functionality for training and prediction into a single helper function called `train_predict()` to make running multiple times simpler. We will not define all of the details inside this function here but let's run through the main pieces of MLflow functionality contained within it.

3. First, we need to let MLflow know that we are now starting a training run we wish to track:

```
with mlflow.start_run():
    # Experiment code and mlflow logging goe
```

4. Inside this loop, we then define and train the model, using parameters defined elsewhere in the code:

```
# create Prophet model
model = Prophet(
    yearly_seasonality=seasonality_params['y'
    weekly_seasonality=seasonality_params['w'
    daily_seasonality=seasonality_params['da
)
# train and predict
model.fit(df_train)
```

5. We then perform some cross-validation to calculate some metrics we would like to log:

```
# Evaluate Metrics
df_cv = cross_validation(model, initial="730
                                         period="180 days",
df_p = performance_metrics(df_cv)
```

6. We can log these metrics, for example, the **Root Mean Squared Error (RMSE)** here, to our MLflow server:

```
# Log parameter, metrics, and model to MLflow
mlflow.log_metric("rmse", df_p.loc[0, "rmse"]
```

7. Then finally, we can use our model wrapper class to log the model and print some information about the run:

```
mlflow.pyfunc.log_model("model", python_model)
print(
    "Logged model with URI: runs:/{}{}/run_id={}/"
    .format(run_id=mlflow.active_run().info.run_id))
)
```

8. With only a few extra lines, we have started to perform version control on our models and track the statistics of different runs!

There are many different ways to save the ML model you have built to MLflow (and in general), which is particularly important when tracking model versions. Some of the main options are as follows:

- **pickle**: `pickle` is a Python library for object serialization that is often used for the export of ML models that are written in `scikit-learn` or pipelines in the wider `scipy` ecosystem (<https://docs.python.org/3/library/pickle.html#module-pickle>). Although it is extremely easy to use and often very fast, you must be careful when exporting your models to `pickle` files because of the following:

- **Versioning:** When you pickle an object, you have to unpickle it in other programs using the *same version of pickle* for stability reasons. This adds more complexity to managing your project.
- **Security:** The documentation for `pickle` states clearly that it is *not secure* and that it is very easy to construct malicious pickles, which will execute dangerous code upon unpickling. This is a very important consideration, especially as you move toward production.

In general, as long as the lineage of the `pickle` files you use is known and the source is trusted, they are OK to use and a very simple and fast way to share your models!

- **joblib:** `joblib` is a general-purpose pipelining library in Python that is very powerful but lightweight. It has a lot of really useful capabilities centered around caching, parallelizing, and compression that make it a very versatile tool for saving and reading in your ML pipelines. It is also particularly fast for storing large `NumPy` arrays, so is useful for data storage. We will use `joblib` more in later chapters. It is important to note that `joblib` suffers from the same security issues as `pickle`, so knowing the lineage of your `joblib` files is incredibly important.
- **JSON:** If `pickle` and `joblib` aren't appropriate, you can serialize your model and its parameters in JSON format. This is good because JSON is a standardized text serialization format that is commonly used across a variety of solutions and platforms. The caveat to using JSON serialization of your models is that you often have to manually define the JSON structure with the relevant parameters you want to store. So, it can create a lot of extra work. Several ML libraries in Python have

their own export to JSON functionality, for example, the deep learning package Keras, but they can all result in quite different formats.

- **MLeap**: MLeap is a serialization format and execution engine based on the **Java Virtual Machine (JVM)**. It has integrations with Scala, PySpark, and Scikit-Learn but you will often see it used in examples and tutorials for saving Spark pipelines, especially for models built with Spark ML. This focus means it is not the most flexible of formats but is very useful if you are working in the **Spark ecosystem**.
- **ONNX**: The **Open Neural Network Exchange (ONNX)** format is aimed at being completely cross-platform and allowing the exchange of models between the main ML frameworks and ecosystems. The main downside of ONNX is that (as you can guess from the name) it is mainly aimed at neural network-based models, with the exception of its `scikit-learn` API. It is an excellent option if you are building a neural network though.

In *Chapter 3, From Model to Model Factory*, we will export our models to MLflow using some of these formats, but they are all compatible with MLflow and so you should feel comfortable using them as part of your ML engineering workflow.

The final section of this chapter will introduce some important concepts for planning how you wish to deploy your solution, prefacing more detailed discussions later in the book.

Deploy

The final stage of the ML development process is the one that really matters: how do you get the amazing solution you have built out into the

real world and solve your original problem? The answer has multiple parts, some of which will occupy us more thoroughly later in this book but will be outlined in this section. If we are to successfully deploy our solution, first of all, we need to know our deployment options: what infrastructure is available and is appropriate for the task? We then need to get the solution from our development environment onto this production infrastructure so that, subject to appropriate orchestration and controls, it can execute the tasks we need it to and surface the results where it has to. This is where the concepts of **DevOps** and **MLOps** come into play.

Let's elaborate on these two core concepts, laying the groundwork for later chapters and exploring how to begin deploying our work.

Knowing your deployment options

In *Chapter 5, Deployment Patterns and Tools*, we will cover in detail what you need to get your ML engineering project from the **develop** to **deploy** stage, but to pre-empt that and provide a taster of what is to come, let's explore the different types of deployment options we have at our disposal:

- **On-premises deployment:** The first option we have is to ignore the public cloud altogether and deploy our solutions in-house on owned infrastructure. This option is particularly popular and necessary for a lot of large institutions with a lot of legacy software and strong regulatory constraints on data location and processing. The basic steps for deploying on-premises are the same as deploying on the cloud but often require a lot more involvement from other teams with particular specialties. For example, if you are in the cloud, you often do not need to spend a lot of time configuring networking or implementing load balancers, whereas on-premises solutions will require these.

The big advantage of on-premises deployment is security and peace of mind that none of your data is going to traverse your company firewall. The big downsides are that it requires a larger investment upfront for hardware and that you have to expend a lot of effort to successfully configure and manage that hardware effectively. We will not be discussing on-premises deployment in detail in this book, but all of the concepts we will employ around software development, packaging, environment management, and training and prediction systems still apply.

- **Infrastructure-as-a-Service (IaaS):** If you are going to use the cloud, one of the lowest levels of abstraction you have access to for deployment is IaaS solutions. These are typically based on the concept of virtualization, such that servers with a variety of specifications can be spun up at the user's will. These solutions often abstract away the need for maintenance and operations as part of the service. Most importantly, they allow extreme scalability of your infrastructure as you need it. Have to run 100 more servers next week? No problem, just scale up your IaaS request and there it is. Although IaaS solutions are a big step up from fully managed on-premises infrastructure, there are still several things you need to think about and configure. The balance in cloud computing is always over how easy you want things to be versus what level of control you want to have. IaaS maximizes control but minimizes (relative) ease compared to some other solutions. In **AWS**, **Simple Storage Service (S3)** and **Elastic Compute Cloud (EC2)** are good examples of IaaS offerings.
- **Platform-as-a-Service (PaaS):** PaaS solutions are the next level up in terms of abstraction and usually provide you with a lot of capabilities without needing to know exactly what is going on under the hood. This

means you can focus solely on the development tasks that the platform is geared up to support, without worrying about underlying infrastructure at all. One good example is **AWS Lambda** functions, which are serverless functions that can scale almost without limit.

All you are required to do is enter the main piece of code you want to execute inside the function. Another good example is **Databricks**, which provides a very intuitive UI on top of the **Spark cluster** infrastructure, with the ability to provision, configure, and scale up these clusters almost seamlessly.

Being aware of these different options and their capabilities can help you design your ML solution and ensure that you focus your team's engineering effort where it is most needed and will be most valuable. If your ML engineer is working on configuring routers, for example, you have definitely gone wrong somewhere.

But once you have selected the components you'll use and provisioned the infrastructure, how do you integrate these together and manage your deployment and update cycles? This is what we will explore now.

Understanding DevOps and MLOps

A very powerful idea in modern software development is that your team should be able to continuously update your code base as needed, while testing, integrating, building, packaging, and deploying your solution should be as automated as possible. This then means these processes can happen on an almost continual basis without big pre-planned **buckets** of time being assigned to update cycles. This is the main idea behind **CI/CD**. CI/CD is a core part of **DevOps** and its ML-focused cousin **MLOps**, which both aim to bring together software development and post-deployment

operations. Several of the concepts and solutions we will develop in this book will be built up so that they naturally fit within an MLOps framework.

The CI part is mainly focused on the stable incorporation of ongoing changes to the code base while ensuring functionality remains stable. The CD part is all about taking the resultant stable version of the solution and pushing it to the appropriate infrastructure.

Figure 2.12 shows a high-level view of this process:

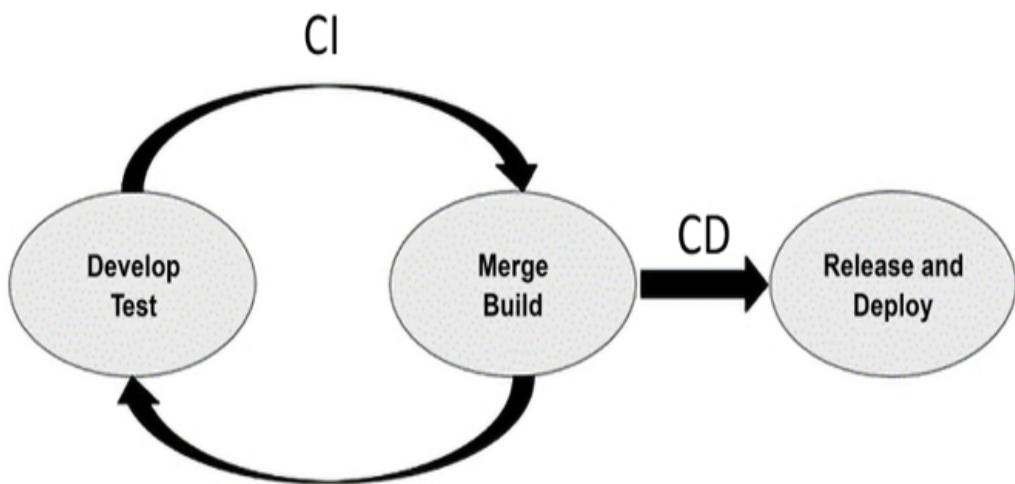


Figure 2.12: A high-level view of CI/CD processes.

In order to make CI/CD a reality, you need to incorporate tools that help automate tasks that you would traditionally perform manually in your development and deployment process. For example, if you can automate the running of tests upon merging of code, or the pushing of your code artifacts/models to the appropriate environment, then you are well on your way to CI/CD.

We can break this out further and think of the different types of tasks that fall into the DevOps or MLOps lifecycles for a solution. Development tasks will typically cover all of the activities that take you from a blank screen on

your computer to a working piece of software. This means that development is where you spend most of your time in a DevOps or MLOps project. This covers everything from writing the code to formatting it correctly and testing it.

Table 2.3 splits out these typical tasks and provides some details on how they build on each other, as well as typical tools you could use in your Python stack for enabling them.

Lifecycle Stage	Activity	Details	Tools
Dev	Testing	Unit tests: tests aimed at testing the functionality smallest pieces of code.	pytest or unittest
		Integration tests: ensure that interfaces within the code and to other solutions work.	Selenium
		Acceptance tests: business focused tests.	Behave
		UI tests: ensuring any frontends behave as expected.	
	Linting	Raise minor stylistic errors and bugs.	flake8 or bandit

Formatting	Enforce well-formatted code automatically.	black or sort
Building	The final stage of bringing the solution together.	Docker, twine, or pip

Table 2.3: Details of the development activities carried out in any DevOps or MLOps project.

Next, we can think about the ML activities within MLOps, which this book will be very concerned with. This covers all of the tasks that a classic Python software engineer would not have to worry about, but that are crucially important to get right for ML engineers like us. This includes the development of capabilities to automatically train the ML models, to run the predictions or inferences the model should generate, and to bring that together inside code pipelines. It also covers the staging and management of the versions of your models, which heavily complements the idea of versioning your application code, as we do using tools like Git. Finally, an ML engineer also has to consider that they have to build out specific monitoring capabilities for the operational mode of their solution, which is not covered in traditional DevOps workflows. For an ML solution, you may have to consider monitoring things like precision, recall, the f1-score, population stability, entropy, and data drift in order to know if the model component of your solution is behaving within a tolerable range. This is very different from classic software engineering as it requires a knowledge of how ML models work, how they can go wrong, and a real appreciation of the importance of data quality to all of this. This is why ML engineering is such an exciting place to be! See *Table 2.4* for some more details on these types of activities.

Lifecycle Stage	Activity	Details	Tools
ML	Training	Train the model .	Any ML package.
	Predicting	Run the predictions or inference steps.	Any ML package.
	Building	Creating the pipelines and application logic in which the model is embedded.	sklearn pipelines, Spark ML pipelines, ZenML.
	Staging	Tag and release the appropriate version of your models and pipelines.	MLflow or Comet.ml.
	Monitoring	Track the solution performance and raise alerts when necessary.	Seldon, Neptune.ai, Evidently.ai, or Arthur.ai.

Table 2.4: Details on the ML-centered activities carried out during an MLOps project.

Finally, in either DevOps or MLOps, there is the Ops piece, which refers to Operations. This is all about how the solution will actually run, how it will alert you if there is an issue, and if it can recover successfully. Naturally

then, operations will cover activities relating to the final packaging, build, and release of your solution. It also has to cover another type of monitoring, which is different from the performance monitoring of ML models. This monitoring has more of a focus on infrastructure utilization, stability, and scalability, on solution latency, and on the general running of the wider solution. This part of the DevOps and MLOps lifecycle is quite mature in terms of tooling, so there are many options available. Some information to get you started is presented in *Table 2.5*.

Lifecycle Stage	Activity	Details	Tools
Ops	Releasing	Taking the software you have built and storing it somewhere central for reuse.	Twine, pip, GitHub, or BitBucket.
	Deploying	Pushing the software you have built to the appropriate target location and environment.	Docker, GitHub Actions, Jenkins, TravisCI, or CircleCI.
	Monitoring	Tracking the performance and utilization of the underlying infrastructure and general software performance, alerting where necessary.	DataDog, Dynatrace, or Prometheus.

Table 2.5: Details of the activities carried out in order to make a solution operational in a DevOps or MLOps project.

Now that we have elucidated the core concepts needed across the MLOps lifecycle, in the next section, we will discuss how to implement CI/CD practices so that we can start making this a reality in our ML engineering projects. We will also extend this to cover automated testing of the performance of your ML models and pipelines, and to perform automated retraining of your ML models.

Building our first CI/CD example with GitHub Actions

We will use GitHub Actions as our CI/CD tool in this book, but there are several other tools available that do the same job. GitHub Actions is available to anyone with a GitHub account, has a very useful set of documentation, <https://docs.github.com/en/actions>, and is extremely easy to start using, as we will show now.

When using GitHub Actions, you have to create a `.yml` file that tells GitHub when to perform the required actions and, of course, what actions to perform. This `.yml` file should be put in a folder called `.github/workflows` in the root directory of your repository. You will have to create this if it doesn't already exist. We will do this in a new branch called `feature/actions`. Create this branch by running:

```
git checkout -b feature/actions
```

Then, create a `.yml` file called `github-actions-basic.yml`. In the following steps, we will build up this example `.yml` file for a Python project where we automatically install dependencies, run a **linter** (a solution to check for bugs, syntax errors, and other issues), and then run some unit tests. This example comes from the GitHub Starter Workflows repository (<https://github.com/actions/starter-workflows/blob/main/ci/python-package-conda.yml>). Open up `github-actions-basic.yml` and then execute the following:

1. First, you define the name of the GitHub Actions workflow and what Git event will trigger it:

```
name: Python package
on: [push]
```

2. You then list the jobs you want to execute as part of the workflow, as well as their configuration. For example, here we have one job called `build`, which we want to run on the latest Ubuntu distribution, and we want to attempt the build using several different versions of Python:

```
jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: [3.9, 3.10]
```

3. You then define the steps that execute as part of the job. Each step is separated by a hyphen and is executed as a separate command. It is important to note that the `uses` keyword grabs standard GitHub Actions; for example, in the first step, the workflow uses the **v2** version of the `checkout` action, and the second step sets up the Python versions we want to run in the workflow:

```
steps:
  - uses: actions/checkout@v3
  - name: Set up Python ${{ matrix.python-version}}
    uses: actions/setup-python@v4
    with:
      python-version: ${{ matrix.python-version }}
```

4. The next step installs the relevant dependencies for the solution using `pip` and a `requirements.txt` file (but you can use `conda` of course!):

```
- name: Install dependencies
run:
  - python -m pip install --upgrade pip
  - pip install flake8 pytest
  - if [ -f requirements.txt ]; then pip install -r requirements.txt;
  - name: Lint with flake8
    run: flake8
```

5. We then run some linting:

```
- name: Lint with flake8
run: |
    # stop the build if there are Python syntax
    # errors or warnings.
    flake8 . --count --select=E9,F63,F7,F82 --
    # exit-zero treats all errors as warnings.
    127 chars wide
    flake8 . --count --exit-zero --max-complex
    length=127 --statistics
```

6. Finally, we run our tests using our favorite Python testing library. For this step, we do not want to run through the entire repository, as it is quite complex, so for this example, we use the `working-directory` keyword to only run `pytest` in that directory. Since it contains a simple test function in `test_basic.py`, this will automatically pass:

```
- name: Test with pytest
run: pytest
working-directory: Chapter02
```

We have now built up the GitHub Actions workflow; the next stage is to show it running. This is taken care of automatically by GitHub, all you have to do is push to the remote repository. So, add the edited `.yml` file, commit it, and then push it:

```
git add .github/workflows/github-actions-basic.yml
git commit -m "Basic CI run with dummy test"
```

```
git push origin feature/actions
```

After you have run these commands in the terminal, you can navigate to the GitHub UI and then click on **Actions** in the top menu bar. You will then be presented with a view of all action runs for the repository like that in *Figure 2.13*.

The screenshot shows a list of workflow runs for a Python package. At the top, there is a search bar labeled "Filter workflow runs" and a "..." button. Below the search bar, there are filters for "Event", "Status", "Branch", and "Actor". A summary row indicates "1 workflow run". The main list contains one item: "Basic CI run with dummy test". This item includes details: "Python package #12: Commit e293632 pushed by AndyMc629", a "feature/actions" badge, a timestamp "9 minutes ago", and a duration "17s". There is also a "..." button next to the timestamp.

Figure 2.13: The GitHub Actions run as viewed from the GitHub UI.

If you then click on the run, you will be presented with details of all jobs that ran within the **Actions** run, as shown in *Figure 2.14*.

The screenshot shows the details of the "Basic CI run with dummy test" workflow run. At the top, it says "Triggered via push 10 minutes ago" and shows "AndyMc629 pushed -o- e29a632 feature/actions". It displays the status as "Success", "Total duration" as "17s", and "Artifacts" as "-". Below this, the configuration file "github-actions-basic.yml" is shown with the "on: push" section. A "Matrix: build" section is expanded, showing "2 jobs completed". A "Show all jobs" link is visible at the bottom of this section.

Figure 2.14: GitHub Actions run details from the GitHub UI.

Finally, you can go into each job and see the steps that were executed, as shown in *Figure 2.15*. Clicking on these will also show the outputs from each of the steps. This is extremely useful for analyzing any failures in the run.

The screenshot shows the GitHub Actions run details interface. At the top left is a back arrow labeled "Python package". Below it is a green checkmark icon followed by the text "Basic CI run with dummy test #12". On the left side, there's a sidebar with the following options: "Summary" (disabled), "Jobs" (disabled), "build (3.9)" (disabled), "build (3.10)" (selected and highlighted with a blue border), "Run details" (disabled), "Usage" (disabled), and "Workflow file" (disabled). The main content area has a dark background. It displays a summary for the "build (3.10)" job, which succeeded 11 minutes ago in 8s. Below this is a list of execution steps, each with a green checkmark icon:

- > Set up job
- > Run actions/checkout@v3
- > Set up Python 3.10
- > Install dependencies
- > Lint with flake8
- > Test with pytest
- > Post Set up Python 3.10
- > Post Run actions/checkout@v3
- > Complete job

Figure 2.15: The GitHub Actions run steps as shown on the GitHub UI.

What we have shown so far is an example of CI. For this to be extended to cover CD, we need to include steps that push the produced solution to its

target host destination. Examples are building a Python package and publishing it to `pip`, or creating a pipeline and pushing it to another system for it to be picked up and run. This latter example will be covered with an **Airflow DAG** in *Chapter 5, Deployment Patterns and Tools*. And that, in a nutshell, is how you start building your CI/CD pipelines. As mentioned, later in the book, we will build workflows specific to our ML solutions.

Now we will look at how we take CI/CD concepts to the next level for ML engineering and build some tests for our model performance, which can then also be triggered as part of continuous processes.

Continuous model performance testing

As ML engineers, we not only care about the core functional behavior of the code we are writing; we also have to care about the models that we are building. This is an easy thing to forget, as traditional software projects do not have to consider this component.

The process I will now walk you through shows how you can take some base reference data and start to build up some different flavors of tests to give confidence that your model will perform as expected when you deploy it.

We have already introduced how to test automatically with Pytest and GitHub Actions, the good news is that we can just extend this concept to include the testing of some model performance metrics. To do this, you need a few things in place:

1. Within the action or tests, you need to retrieve the reference data for performing the model validation. This can be done by pulling from a

remote data store like an object store or a database, as long as you provide the appropriate credentials. I would suggest storing these as secrets in Github. Here, we will use a dataset generated in place using the `sklearn` library as a simple example.

2. You need to retrieve the model or models you wish to test from some location as well. This could be a full-fledged model registry or some other storage mechanism. The same points around access and secrets management as in *point 1* apply. Here we will pull a model from the `Hugging Face Hub` (more on Hugging Face in *Chapter 3*), but this could equally have been an MLflow Tracking instance or some other tool.
3. You need to define the tests you want to run and that you are confident will achieve the desired outcome. You do not want to write tests that are far too sensitive and trigger failed builds for spurious reasons, and you also want to try and define tests that are useful for capturing the types of failures you would want to flag.

For *point 1*, here we grab some data from the `sklearn` library and make it available to the tests through a `pytest fixture`:

```
@pytest.fixture
def test_dataset() -> Union[np.array, np.array]
    # Load the dataset
    X, y = load_wine(return_X_y=True)
    # create an array of True for 2 and False otherwise
    y = y == 2
    # Train and test split
    X_train, X_test, y_train, y_test = train_te
```

```
    return X_test, y_test
```

For *point 2*, I will use the `Hugging Face Hub` package to retrieve the stored model. As mentioned in the bullets above, you will need to adapt this to whatever model storage mechanism you are accessing. The repository in this case is public so there is no need to store any secrets; if you did need to do this, please use the GitHub Secrets store.

```
@pytest.fixture
def model() -> sklearn.ensemble._forest.RandomForestClassifier:
    REPO_ID = "electricweegie/mlewp-sklearn-win"
    FILENAME = "rfc.joblib"
    model = joblib.load(hf_hub_download(REPO_ID))
    return model
```

Now, we just need to write the tests. Let's start simple with a test that confirms that the predictions of the model produce the correct object types:

```
def test_model_inference_types(model, test_data):
    assert isinstance(model.predict(test_data), np.ndarray)
    assert isinstance(test_dataset[0], np.ndarray)
    assert isinstance(test_dataset[1], np.ndarray)
```

We can then write a test to assert some specific conditions on the performance of the model on the test dataset is met:

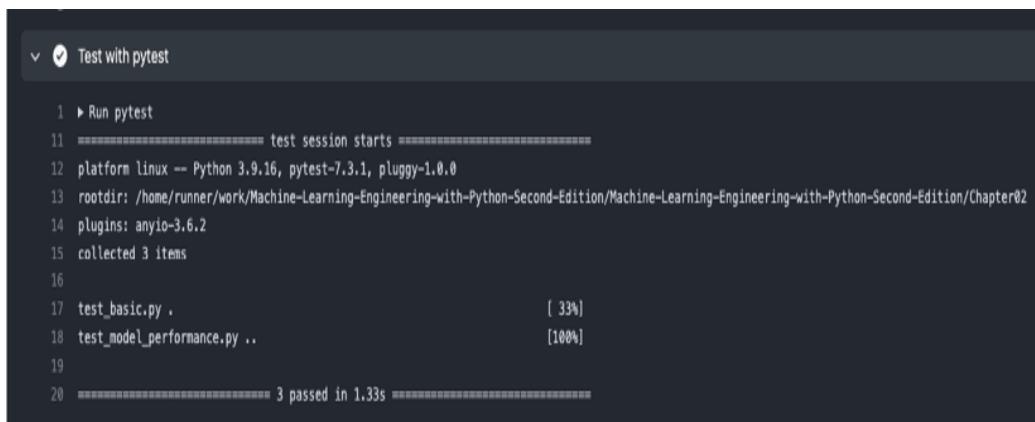
```

def test_model_performance(model, test_dataset)
    metrics = classification_report(y_true=test
                                      y_pred=mode
                                      output_dict=True)
    assert metrics['False']['f1-score'] > 0.95
    assert metrics['False']['precision'] > 0.9
    assert metrics['True']['f1-score'] > 0.8
    assert metrics['True']['precision'] > 0.8

```

The previous test can be thought of as something like a data-driven unit test and will make sure that if you change something in the model (perhaps you change some feature engineering step in the pipeline or you change a hyperparameter), you will not breach the desired performance criteria. Once these tests have been successfully added to the repo, on the next push, the GitHub action will be triggered and you will see that the model performance test runs successfully.

This means we are performing some continuous model validation as part of our CI/CD process!



```

Test with pytest
Run pytest
=====
platform linux -- Python 3.9.16, pytest-7.3.1, pluggy-1.0.0
rootdir: /home/runner/work/Machine-Learning-Engineering-with-Python-Second-Edition/Machine-Learning-Engineering-with-Python-Second-Edition/Chapter02
plugins: anyio-3.6.2
collected 3 items

test_basic.py . [ 33%]
test_model_performance.py .. [100%]

=====
3 passed in 1.33s

```

Figure 2.16: Successfully executing model validation tests as part of a CI/CD process using GitHub Actions.

More sophisticated tests can be built upon this simple concept, and you can adapt the environment and packages used to suit your needs.

Continuous model training

An important extension of the “continuous” concept in ML engineering is to perform continuous training. The previous section showed how to trigger some ML processes for testing purposes when pushing code; now, we will discuss how to extend this for the case where you want to trigger retraining of the model based on a code change. Later in this book, we will learn a lot about training and retraining ML models based on a variety of different triggers like data or model drift in *Chapter 3, From Model to Model Factory*, and about how to deploy ML models in general in *Chapter 5, Deployment Patterns and Tools*. Given this, we will not cover the details of deploying to different targets here but instead show you how to build continuous training steps into your CI/CD pipelines.

This is actually simpler than you probably think. As you have hopefully noticed by now, CI/CD is really all about automating a series of steps, which are triggered upon particular events occurring during the development process. Each of these steps can be very simple or more complex, but fundamentally it is always just other programs we are executing in the specified order upon activating the trigger event.

In this case, since we are concerned with continuous training, we should ask ourselves, when would we want to retrain during code development? Remember that we are ignoring the most obvious cases of retraining on a schedule or upon a drift in model performance or data quality, as these are touched on in later chapters. If we only consider that the code is changing

for now, the natural answer is to train only when there is a substantial change to the code.

For example, if a trigger was fired every time we committed our code to version control, this would likely result in a lot of costly compute cycles being used for not much gain, as the ML model will likely not perform very differently in each case. We could instead limit the triggering of retraining to only occur when a pull request is merged into the main branch. In a project, this is an event that signifies a new software feature or functionality has been added and has now been incorporated into the core of the solution.

As a reminder, when building CI/CD in GitHub Actions, you create or edit `YAML` files contained in the `.github` folder of your Git repository. If we want to trigger a training process upon a pull request, then we can add something like:

```
name: Continuous Training Example
on: [pull_request]
```

And then we need to define the steps for pushing the appropriate training script to the target system and running it. First, this would likely require some fetching of access tokens. Let's assume this is for AWS and that you have loaded your appropriate AWS credentials as GitHub Secrets; for more information, see *Chapter 5, Deployment Patterns and Tools*. We would then be able to retrieve these in the first step of a `deploy-trainer` job:

```
jobs:
  deploy-trainer
    runs-on: [ubuntu-latest]
```

```
steps:
  - name: Checkout      uses: actions/checko
  - name: Configure AWS Credentials
    uses: aws-actions/configure-aws-credentia
    with:
      aws-access-key-id: ${{ secrets.AWS_ACCE
      aws-secret-access-key: ${{ secrets.AWS_
      aws-region: us-east-2
      role-to-assume: ${{ secrets.AWS_ROLE_TO
      role-external-id: ${{ secrets.AWS_ROLE_
      role-duration-seconds: 1200
      role-session-name: TrainingSession
```

You may then want to copy your repository files to a target **S3** destination; perhaps they contain modules that the main training script needs to run. You could then do something like this:

```
- name: Copy files to target destination
run: aws s3 sync . s3://<S3-BUCKET-NAME>
```

And finally, you would want to run some sort of process that uses these files to perform the training. There are so many ways to do this that I have left the specifics out for this example. Many ways for deploying ML processes will be covered in *Chapter 5, Deployment Patterns and Tools*:

```
- name: Run training job
run: |
  # Your bespoke run commands go in here
```

And with that, you have all the key pieces you need to run continuous ML model training to complement the other section on continuous model performance testing. This is how you bring the DevOps concept of CI/CD to the world of MLOps!

Summary

This chapter was all about building a solid foundation for future work. We discussed the development steps common to all ML engineering projects, which we called “*Discover, Play, Develop, Deploy,*” and contrasted this way of thinking against traditional methodologies like CRISP-DM. In particular, we outlined the aim of each of these steps and their desired outputs.

This was followed by a high-level discussion of tooling and a walkthrough of the main setup steps. We set up the tools for developing our code, keeping track of the changes to that code, managing our ML engineering project, and finally, deploying our solutions.

In the rest of the chapter, we went through the details for each of the four steps we outlined previously, with a particular focus on the *Develop* and *Deploy* stages. Our discussion covered everything from the pros and cons of Waterfall and Agile development methodologies to environment management and then software development best practices. We explored how to package your ML solution and what deployment infrastructure is available for you to use, and outlined the basics of setting up your DevOps and MLOps workflows. We finished up the chapter by discussing, in some detail, how to apply testing to our ML code, including how to automate this testing as part of CI/CD pipelines. This was then extended into the concepts of continuous model performance testing and continuous model training.

In the next chapter, we will turn our attention to how to build out the software for performing the automated training and retraining of your models using a lot of the techniques we have discussed here.

Join our community on Discord

Join our community's Discord space for discussion with the author and other readers:

<https://packt.link/mle>



3

From Model to Model Factory

This chapter is all about one of the most important concepts in ML engineering: how do you take the difficult task of training and fine-tuning your models and make it something you can automate, reproduce, and scale for production systems?

We will recap the main ideas behind training different ML models at a theoretical and practical level, before providing motivation for retraining, namely the idea that ML models will not perform well forever. This concept is also known as **drift**. Following this, we will cover some of the main concepts behind feature engineering, which is a key part of any ML task. Next, we will deep dive into how ML works and how it is, at heart, a series of optimization problems. We will explore how when setting out to tackle these optimization problems, you can do so with a variety of tools at various levels of abstraction. In particular, we will discuss how you can provide the direct definition of the model you want to train, which I term *hand cranking*, or how you can perform hyperparameter tuning or **automated ML (AutoML)**. We will look at examples of using different libraries and tools that do all of these, before exploring how to implement them for later use in your training workflow. We will then build on the introductory work we did in *Chapter 2, The Machine Learning*

Development Process, on MLflow by showing you how to interface with the different MLflow APIs to manage your models and update their status in MLflow’s Model Registry.

We will end this chapter by discussing the utilities that allow you to chain all of your ML model training steps into single units known as **pipelines**, which can help act as more compact representations of all the steps we have discussed previously. The summary at the end will recap the key messages and also outline how what we have done here will be built upon further in *Chapter 4, Packaging Up*, and *Chapter 5, Deployment Patterns and Tools*.

In essence, this chapter will tell you *what* you need to stick together in your solution, while later chapters will tell you *how* to stick them together robustly. We will cover this in the following sections:

- Defining the model factory
- Learning about learning
- Engineering features for machine learning
- Designing your training system
- Retraining required
- Persisting your models
- Building the model factory with pipelines

Technical requirements

As in the previous chapters, the required packages for this chapter are contained within a conda environment `.yml` file in the repository folder for **Chapter03**, so to create the conda environment for this chapter, simply run the following command:

```
conda env create -f mlewp-chapter03.yml
```

This will install packages including MLflow, AutoKeras, Hyperopt Optuna, auto-sklearn, Alibi Detect, and Evidently.

Note that if you are running these examples on a Macbook with Apple Silicon, a straight `pip` or `conda` install of TensorFlow and `auto-sklearn` may not work out of the box. Instead, you will need to install the following packages to work with TensorFlow:

```
pip install tensorflow-macos
```

And then



```
pip install tensorflow-metal
```

To install `auto-sklearn`, you will need to run

```
brew install swig
```

Or install `swig` using whatever Mac package manager you use, then you can run

```
pip install auto-sklearn
```

Defining the model factory

If we want to develop solutions that move away from ad hoc, manual, and inconsistent execution and toward ML systems that can be automated, robust, and scalable, then we have to tackle the question of how we will create and curate the star of the show: the models themselves.

In this section, we will discuss the key components that have to be brought together to move toward this vision and provide some examples of what these may look like in code. These examples are not the only way to implement these concepts, but they will enable us to start building up our ML solutions toward the level of sophistication we will need if we want to deploy in the *real world*.

The main components we are talking about here are as follows:

- **Training system:** A system for robustly training our models on the data we have in an automated way. This consists of all the code we have developed to train our ML models on data.
- **Model store:** A place to persist successfully trained models and a place to share production-ready models with components that will run the predictions.
- **Drift detector:** A system for detecting changes in model performance to trigger training runs.

These components, combined with their interaction with the deployed prediction system, encompass the idea of a model factory. This is shown schematically in the following diagram:

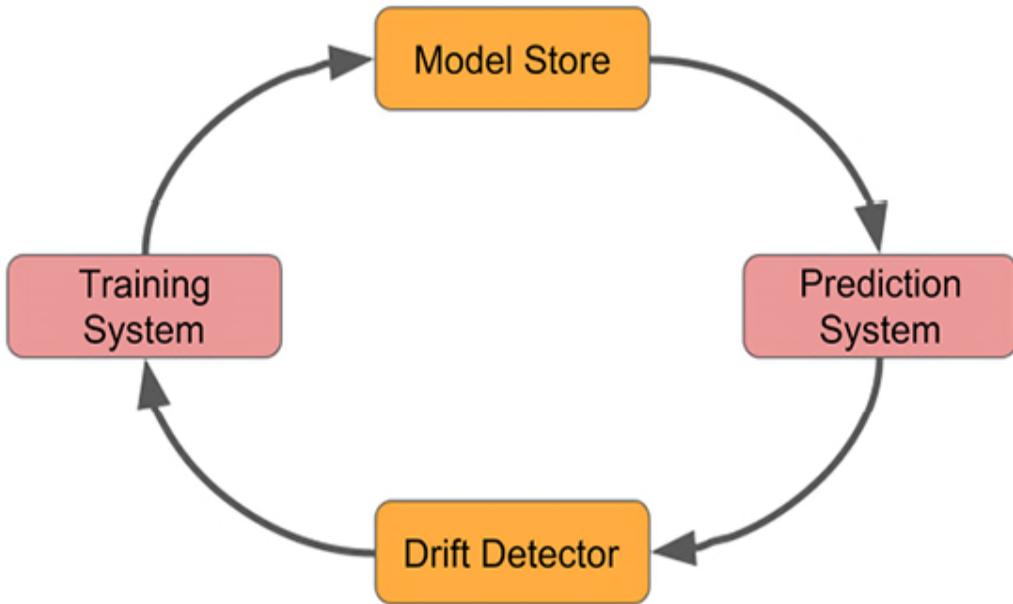


Figure 3.1: The components of the model factory.

For the rest of this chapter, we will explore the three components we mentioned previously in detail. **Prediction systems** will be the focus of later chapters, especially *Chapter 5, Deployment Patterns and Tools*.

First, let's explore what it means to train an ML model and how we can build systems to do so.

Learning about learning

At their heart, ML algorithms all contain one key feature: an optimization of some kind. The fact that these algorithms *learn* (meaning that they iteratively improve their performance concerning an appropriate metric upon exposure to more observations) is what makes them so powerful and exciting. This process of learning is what we refer to when we say *training*.

In this section, we will cover the key concepts underpinning training, the options we can select in our code, and what these mean for the potential

performance and capabilities of our training system.

Defining the target

We have just stated that training is an optimization, but what exactly are we optimizing? Let's consider supervised learning. In training, we provide the labels or values that we would want to predict for the given feature so that the algorithms can learn the relationship between the features and the target. To optimize the internal parameters of the algorithm during training, it needs to know how *wrong* it would be with its current set of parameters. The optimization is then all about updating the parameters so that this measure of *wrongness* gets smaller and smaller. This is exactly what is captured by the concept of a loss function.

Loss functions come in a variety of forms, and you can even define your own if you need to with a lot of packages, but there are some standard ones that it helps to be aware of. The names of some of these are mentioned here.

For regression problems, you can use the following:

- Mean squared error/L2 loss
- Mean absolute error/L1 loss

For binary classification problems, you can use the following:

- Log loss/logistic loss/cross-entropy loss
- Hinge loss

For multi-class classification problems, you can use the following:

- Multi-class cross entropy loss
- Kullback Leibler divergence loss

In unsupervised learning, the concept of a loss function still applies but now the target is the correct distribution of the input data. After defining your loss function, you then need to optimize it. This is what we will look at in the next section.

Cutting your losses

At this point, we know that training is all about optimizing, and we know what to optimize, but we have not covered *how* to optimize yet.

As usual, there are plenty of options to choose from. In this section, we will look at some of the main approaches.

The following are the **constant learning rate** approaches:

- **Gradient descent:** This algorithm works by calculating the derivative of our loss function regarding our parameters, and then uses this to construct an update that moves us in the direction of decreasing loss.
- **Batch gradient descent:** The gradient that we use to make our move in the parameter space is found by taking the average of all the gradients found. It does this by looking at each data point in our training set and checking that the dataset is not too large and the loss function is relatively smooth and convex. This can pretty much reach the global minimum.
- **Stochastic gradient descent:** The gradient is calculated using one randomly selected data point at each iteration. This is faster at getting to the global minimum of the loss function, but it is more susceptible to sudden fluctuations in the loss after each optimization step.
- **Mini-batch gradient descent:** This is a mixture of both the batch and stochastic cases. In this case, updates to the gradient for each update to

the parameters use several points greater than one but smaller than the entire dataset. This means that the size of the batch is now a parameter that needs to be tuned. The larger the batch, the more we approach batch gradient descent, which provides a better gradient estimate but is slower. The smaller the batch, the more we approach stochastic gradient descent, which is faster but not as robust. Mini-batch allows us to decide where in between the two we want to be. Batch sizes may be selected with a variety of criteria in mind. These can take on a range of memory considerations. Batches processed in parallel and larger batches will consume more memory while providing improved generalization performance for smaller batches. See *Chapter 8* of the book *Deep Learning* by Ian Goodfellow, Yoshua Bengio, and Aaron Courville at <https://www.deeplearningbook.org/> for more details.

Then, there are the **adaptive learning rate methods**. Some of the most common are as follows:

- **AdaGrad**: The learning rate parameters are dynamically updated based on the properties of the learning updates during the optimization process.
- **AdaDelta**: This is an extension of AdaGrad that does not use all the previous gradient updates. Instead, it uses a rolling window on the updates.
- **RMSprop**: This works by maintaining a moving average of the square of all the gradient steps. It then divides the latest gradient by the square root of this.
- **Adam**: This is an algorithm that is supposed to combine the benefits of AdaGrad and RMSprop.

The limits and capabilities of all these optimization approaches are important for us, as ML engineers, because we want to ensure that our training systems use the right tool for the job and are optimal for the problem at hand. Just having the awareness that there are multiple options for your internal optimization will also help you focus your efforts and increase performance.

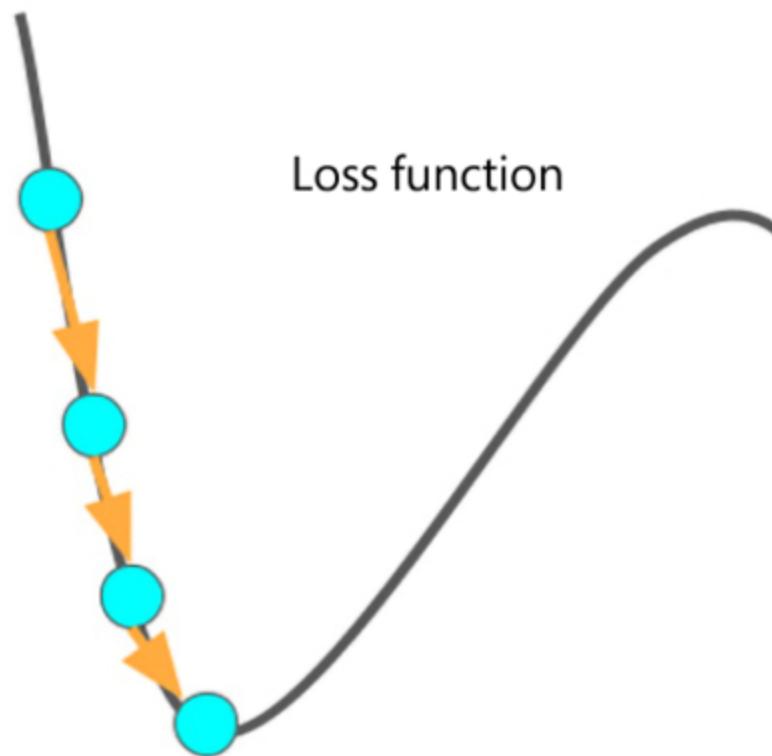


Figure 3.2: Simple representation of training as the optimization of a loss function.

Now, let's discuss how we prepare the raw material that the model factory needs to do its work, the data, through the process of feature engineering.

Preparing the data

Data can come in all varieties of types and quality. It can be tabular and from a relational database, unstructured text from a crawled website, a

formatted response from a REST API, an image, an audio file, or any other form you can think of.

If you want to run machine learning algorithms on this data though, the first thing you have to do is make it readable by these algorithms. This process is known as *feature engineering*, and that is what the next few sections will discuss to give you some grounding in the main principles. There are many excellent resources on feature engineering that can go into a lot of depth, so we will only touch on some of the main concepts here. For more information, you could check out a book like *Feature Engineering Cookbook* by Soledad Galli, Packt, 2022.

Engineering features for machine learning

Before we feed any data into an ML model, it has to be transformed into a state that can be *understood* by our models. We also need to make sure we only do this on the data we deem useful for improving the performance of the model, as it is far too easy to explode the number of features and fall victim to the *curse of dimensionality*. This refers to a series of related observations where, in high-dimensional problems, data becomes increasingly sparse in the feature space, so achieving statistical significance can require exponentially more data. In this section, we will not cover the theoretical basis of feature engineering. Instead, we will focus on how we, as ML engineers, can help automate some of the steps in production. To this end, we will quickly recap the main types of feature preparation and feature engineering steps so that we have the necessary pieces to add to our pipelines later in this chapter.

Engineering categorical features

Categorical features are those that form a non-numerical set of distinct objects, such as the day of the week or hair color. They can be distributed in a variety of ways throughout your data.

For an ML algorithm to be able to *digest* a categorical feature, we need to translate the feature into something numerical, while also ensuring that the numerical representation *does not produce bias or weigh our values inappropriately*. An example of this would be if we had a feature that contained different products sold in a supermarket:

```
data = [['Bleach'], ['Cereal'], ['Toilet Roll']]
```

Here, we can map each to a positive integer using `sklearn`'s `OrdinalEncoder`:

```
from sklearn import preprocessing
ordinal_enc = preprocessing.OrdinalEncoder()
ordinal_enc.fit(data)
# Print returns [[0.]
#                [1.]
#                [2.]]
print(ordinal_enc.transform(data))
```

This is what is called **ordinal encoding**. We have mapped these features to numbers, so there's a big tick there, but is the representation appropriate?

Well, if you think about it for a second, not really. These numbers seem to suggest that cereal is to bleach as toilet roll is to cereal, and that the average of toilet roll and bleach is cereal. These statements don't make sense (and I don't want bleach and toilet roll for breakfast), so this suggests we should try a different approach. This representation would be appropriate, however, in cases where we wanted to maintain the notion of ordering in the categorical features. An excellent example would be if we had a survey and the participants were asked their opinion of the statement *breakfast is the most important meal of the day*. If the participants were then told to select one option from the list *Strongly Disagree*, *Disagree*, *Neither Disagree nor Agree*, *Agree*, and *Strongly Agree* and we ordinally encoded this data to map to the numerical list of 1, 2, 3, 4, and 5, then we could more intuitively answer questions such as *Was the average response more in agreement or disagreement?* and *How widespread was the opinion on this statement?* Ordinal encoding would help here, but as we mentioned previously, it's not necessarily correct in this case.

What we could do is consider the list of items in this feature, and then provide a binary number to represent whether the value is or isn't that particular value in the original list. So, here, we will decide to use `sklearn's OneHotEncoder`:

```
onehot_enc = preprocessing.OneHotEncoder()
onehot_enc.fit(data)
# Print returns [[1. 0. 0.]
#               [0. 1. 0.]
#               [0. 0. 1.]]
print(onehot_enc.transform(data).toarray())
```

This representation is known as a **one-hot encoding**. There are a few benefits to this method of encoding, including the following:

- There are no enforced orderings of the values.
- All the feature vectors have unit norms (more on this later).
- Every unique feature is orthogonal to the others, so there are no weird averages or distance statements that are implicit in the representation.

One of the disadvantages of this approach is that if your categorical list contains a lot of instances, then the size of your feature vector will easily blow up, and we have to both store and work with extremely sparse vectors and matrices at the algorithmic level. This can very easily lead to issues in several implementations and is another manifestation of the dreaded curse of dimensionality.

In the next section, numerical features are discussed.

Engineering numerical features

Preparing numerical features is slightly easier since we already have numbers, but there are a few steps we still need to take to prepare for many algorithms. For most ML algorithms, the features must be all on similar scales; for example, they must have a magnitude between -1 and 1 or 0 and 1. This is for the relatively obvious reason that some algorithms taking in a feature for house price values of up to a million dollars and another for the square footage of the house will automatically weigh the larger dollar values more. This also means that we lose the helpful notion of where specific values sit in their distributions. For example, some algorithms will benefit from scaling features so that the median dollar value and the median square footage value are both represented by 0.5 rather than 500,000 and

350. Or we may want all of our distributions to have the same meaning if they were normally distributed, which allows our algorithms to focus on the shape of the distributions rather than their locations.

So, what do we do? Well, as always, we are not starting from scratch and there are some standard techniques we can apply. Some very common ones are listed here, but there are far too many to include all of them:

- **Standardization:** This is a transformation of a numerical feature and assumes that the distribution of values is normal or Gaussian before scaling the variance to be 1 and the average to be 0. If your data is indeed normal or Gaussian, then this is a good technique to use. The mathematical formula for standardization is very simple, so I've provided it here, where z represents the transformed value, x is the original value, and μ and σ are the average and standard deviation, respectively:

$$z_i = \frac{x_i - \mu}{\sigma}$$

- **Min-max normalization:** In this case, we want to scale the numerical features so that they're always between 0 and 1, irrespective of the type of distribution that they follow.

This is intuitively easy to do, as you just need to subtract the minimum of the distribution from any given value and then divide by the range of the data (maximum minus minimum). You can think of this first step as making sure that all the values are greater than or equal to 0. The second step involves making sure that their maximum size is 1. This can be written with a simple formula, where the transformed number, x_i is the original number, and x_i' represents the entire distribution of that feature:

$$x'_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

- **Feature vector normalization:** Here, you scale every single sample in your dataset so that they have norms equal to 1. This can be very important if you are using algorithms where the distance or cosine similarity between features is an important component, such as in clustering. It is also commonly used in text classification in combination with other feature engineering methods, such as the **TF-IDF statistic**. In this case, assuming your entire feature is numerical, you just calculate the appropriate norm for your feature vector and then divide every component by that value. For example, if we use the Euclidean or L2-norm of the feature vector, $\|x\|$, then we would transform each component, x_j via the following formula:

$$x'_j = \frac{x_j}{\|x\|}$$

To highlight the improvements these simple steps can make to your model's performance, we will look at a simple example from the `sklearn` wine dataset. Here, we will be training a Ridge classifier on data that has not been standardized and then on data that has been standardized. Once we've done this, we will compare the results:

1. First, we must import the relevant libraries and set up our training and test data:

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import RidgeClassifier
```

```
from sklearn import metrics
from sklearn.datasets import load_wine
from sklearn.pipeline import make_pipeline
X, y = load_wine(return_X_y=True)
```

2. Then, we must make a typical 70/30 train/test split:

```
X_train, X_test, y_train, y_test =\
train_test_split(X, y, test_size=0.30, randc
```

3. Next, we must train a model without any standardization in the features and predict on the test set:

```
no_scale_clf = make_pipeline(RidgeClassifier

no_scale_clf.fit(X_train, y_train)
y_pred_no_scale = no_scale_clf.predict(X_tes
```

4. Finally, we must do the same but with a standardization step added in:

```
std_scale_clf = make_pipeline(StandardScaler
std_scale_clf.fit(X_train, y_train)
y_pred_std_scale = std_scale_clf.predict(X_t
```

5. Now, if we print some performance metrics, we will see that without scaling, the accuracy of the predictions is at 0.76, while the other

metrics, such as the weighted averages of `precision`, `recall`, and `f1-score`, are `0.83`, `0.76`, and `0.68`, respectively:

```
print('\nAccuracy [no scaling]')
print('{:.2%}\n'.format(metrics.accuracy_scc
    scale)))
print('\nClassification Report [no scaling]')
print(metrics.classification_report(y_test,
```

6. This produces the following output:

```
Accuracy [no scaling]75.93%
Classification Report [no scaling]
      precision    recall   f1-score
          0         0.90     1.00     0.95
          1         0.66     1.00     0.79
          2         1.00     0.07     0.13
accuracy                      0.76
macro avg                  0.85     0.69     0.63
weighted avg                0.83     0.76     0.68
```

7. In the case where we standardized the data, the metrics are far better across the board, with the accuracy and weighted averages of the `precision`, `recall`, and `f1-score` all at `0.98`:

```
print('\nAccuracy [scaling]')
print('{:.2%}\n'.format(metrics.accuracy_scc
```

```
print('Classification Report [scaling]')
print(metrics.classification_report(y_test,
```

8. This produces the following output:

```
Accuracy [scaling]
98.15%
Classification Report [scaling]
precision    recall   f1-score
0            0.95    1.00    0.97
1            1.00    0.95    0.98
2            1.00    1.00    1.00
accuracy                           0.98
macro avg       0.98    0.98    0.98
weighted avg    0.98    0.98    0.98
```

Here, we can see a significant jump in performance, just by adding one simple step to our ML training process.

Now, let's look at how training is designed and works at its core. This will help us make sensible choices for our algorithms and training approaches.

Designing your training system

Viewed at the highest level, ML models go through a life cycle with two stages: a **training** phase and an **output** phase. During the training phase, the model is fed data to learn from the dataset. In the prediction phase, the

model, complete with its optimized parameters, is fed new data in order and returns the desired output.

These two phases have very different computational and processing requirements. In the training phase, we have to expose the model to as much data as we can to gain the best performance, all while ensuring subsets of data are kept aside for testing and validation. Model training is fundamentally an optimization problem, which requires several incremental steps to get to a solution.

Therefore, this is computationally demanding, and in cases where the data is relatively large (or compute resources are relatively low), it can take a long time. Even if you had a small dataset and a lot of computational resources, training is still not a low-latency process. Also, it is a process that is often run in batches and where small additions to the dataset will not make that much difference to model performance (there are exceptions to this). Prediction, on the other hand, is a more straightforward process and can be thought of in the same way as running any calculation or function in your code: inputs go in, a calculation occurs, and the result comes out. This (in general) is not computationally demanding and is low latency.

Taken together, this means that, firstly, it makes sense to separate these two steps (training and prediction) both logically and in code. Secondly, it means we have to consider the different execution requirements for these two stages and build this into our solution designs. Finally, we need to make choices about our training regime, including whether we schedule training in batches, use incremental learning, or should trigger training based on model performance criteria. These are the key parts of your training system.

Training system design options

Before we create any detailed designs of our training system, some general questions will always apply:

- Is there infrastructure available that is appropriate to the problem?
- Where is the data and how will we feed it to the algorithm?
- How am I testing the performance of the model?

In terms of infrastructure, this can be very dependent on the model and data you are using for training. If you are going to train a linear regression on data with three features and your dataset contains only 10,000 tabular records, you can likely run this on laptop-scale hardware without much thought. This is not a lot of data, and your model does not have a lot of free parameters. If you are training on a far larger dataset, such as one that contains 100 million tabular records, then you could benefit from parallelization across something such as a Spark cluster. If, however, you are training a 100-layer deep convolutional neural network on 1,000 images, then you are likely going to want to use a GPU. There are plenty of options, but the key is choosing the right thing for the job.

Regarding the question of how we feed data to the algorithm, this can be non-trivial. Are we going to run a SQL query against a remotely hosted database? If so, how are we connecting to it? Does the machine we're running the query on have enough RAM to store the data?

If not, do we need to consider using an algorithm that can learn incrementally? For classic algorithmic performance testing, we need to employ the well-known tricks of the ML trade and perform train/test/validation splits on our data. We also need to decide what cross-validation strategies we may want to employ. We then need to select our model performance metric of choice and calculate it appropriately. As ML engineers, however, we will also be interested in *other* measures of

performance, such as training time, efficient use of memory, latency, and (dare I say it) cost. We will need to understand how we can measure and then optimize these as well.

So long as we bear these things in mind as we proceed, we will be in a good position. Now, onto the design.

As we mentioned in the introduction to this section, we have two fundamental pieces to consider: the training and output processes. There are two ways in which we can put these together for our solution. We will discuss this in the next section.

Train-run

Option 1 is to perform training and prediction in the same process, with training occurring in either batch or incremental mode. This is shown schematically in the following diagram. This pattern is called *train-run*:

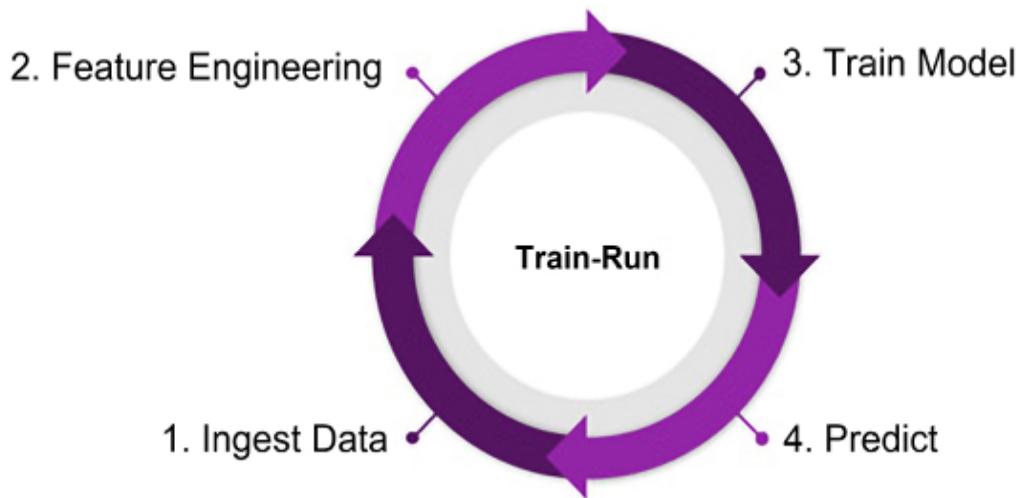


Figure 3.3: The train-run process.

This pattern is the simpler of the two but also the least desirable for real-world problems since it does not embody the *separation of concerns*

principle we mentioned previously. This does not mean it is an invalid pattern, and it does have the advantage of often being simpler to implement. Here, we run our entire training process before making our predictions, with no real *break* in between. Given our previous discussions, we can automatically rule out this approach if we have to serve prediction in a very low-latency fashion; for example, through an event-driven or streaming solution (more on these later).

Where this approach *could* be completely valid, though (and I've seen this a few times in practice), is either in cases where the algorithms you are applying are actually very lightweight to train and you need to keep using very recent data, or where you are running a large batch process relatively infrequently.

Although this is a simple approach and does not apply to all cases, it does have distinct advantages:

- Since you are training as often as you predict, you are doing everything you can to protect against modern performance degradation, meaning that you are combatting *drift* (see later sections in this chapter).
- You are significantly reducing the complexity of your solution. Although you are tightly coupling two components, which should generally be avoided, the training and prediction stages may be so simple to code that if you just stick them together, you will save a lot of development time. This is a non-trivial point because *development time costs money*.

Now, let's look at the other case.

Train-persist

Option 2 is that training runs in batch, while prediction runs in whatever mode is deemed appropriate, with the prediction solution reading in the trained model from a store. We will call this design pattern *train-persist*. This is shown in the following diagram:

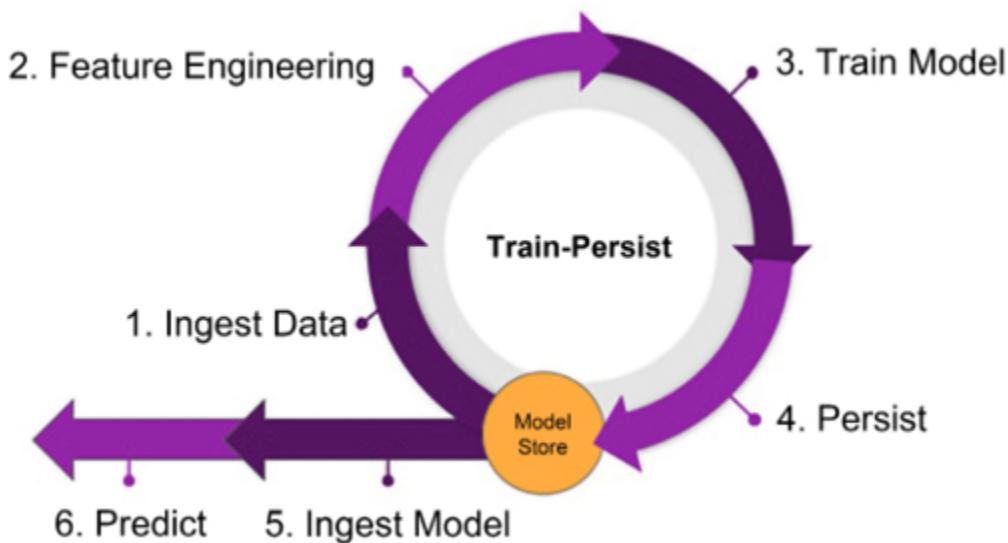


Figure 3.4: The train-persist process.

If we are going to train our model and then persist the model so that it can be picked up later by a prediction process, then we need to ensure a few things are in place:

- What are our model storage options?
- Is there a clear mechanism for accessing our model store (writing to and reading from)?
- How often should we train versus how often will we predict?

In our case, we will solve the first two questions by using MLflow, which we introduced in *Chapter 2, The Machine Learning Development Process*, but will revisit in later sections. There are also lots of other solutions

available. The key point is that no matter what you use as a model store and *handover* point between your train and predict processes, it should be used in a way that is robust and accessible.

The third point is trickier. You could potentially just decide at the outset that you want to train on a schedule, and you stick to that. Or you could be more sophisticated and develop trigger criteria that must be met before training occurs. Again, this is a choice that you, as an ML engineer, need to make with your team. Later in this chapter, we will discuss mechanisms for scheduling your training runs.

In the next section, we will explore what you have to do if you want to trigger your training runs based on how your model's performance could be degrading over time.

Retraining required

You wouldn't expect that after finishing your education, you never read a paper or book or speak to anyone again, which would mean you wouldn't be able to make informed decisions about what is happening in the world. So, you shouldn't expect an ML model to be trained once and then be performant forever afterward.

This idea is intuitive, but it represents a formal problem for ML models known as **drift**. Drift is a term that covers a variety of reasons for your model's performance dropping over time. It can be split into two main types:

- **Concept drift:** This happens when there is a change in the fundamental relationship between the features of your data and the outcome you are trying to predict. Sometimes, this is also known as

covariate drift. An example could be that at the time of training, you only have a subsample of data that seems to show a linear relationship between the features and your outcome. If it turns out that, after gathering a lot more data post-deployment, the relationship is non-linear, then concept drift has occurred. The mitigation against this is retraining with data that is more representative of the correct relationship.

- **Data drift:** This happens when there is a change in the statistical properties of the variables you are using as your features. For example, you could be using *age* as a feature in one of your models but at training time, you only have data for 16–24-year-olds.

If the model gets deployed and your system starts ingesting data for a wider age demographic, then you have data drift.

The truth is that drift is part of life as an ML engineer, so we will spend a good bit of time getting to know how to detect and mitigate against it. But why does it happen? As you would expect, there are a variety of reasons for drift that it is important to consider. Let us consider some examples. Say the mechanism you used for sampling your training data is not appropriate in some way; perhaps you have subsampled for a specific geographic region or demographic, but you want the model to be applied in more general circumstances.

There may be seasonal effects in the problem domain in which we are operating, as can be expected in sales forecasting or weather prediction. Anomalies could be introduced by “black swan” or rare events, like geopolitical events or even the Covid-19 pandemic. The data-gathering process may at some point introduce errors, for example, if there is a bug in

an upstream system or the process itself is not being followed or has changed. This last example can be particularly prevalent in processes where manual inputs of data are required. If a salesperson is to be trusted with correctly labeling the state of a sale in the **Customer Resource Management (CRM)** system, then salespeople with less training or experience may not label the data as accurately or in as timely a manner. Despite advances in so many areas of software development, this sort of data-gathering process is still very prevalent and so you must guard against this in your own machine learning system development. It can be mitigated slightly by trying to enforce more automation of data gathering or in providing guides to those entering data (think drop-down menus), but it is almost certain that a lot of data is still gathered in this way and will be for the foreseeable future.

That drift is an important aspect of your system to consider should be clear now, but dealing with it is actually a multi-step process. We first need to detect the drift. Detecting drift in your deployed models is a key part of MLOps and should be at the forefront of your mind as an ML engineer. We then need to diagnose the source of the drift; this will usually involve some sort of offline investigation by those responsible for monitoring. The tools and techniques we will mention will help you to define workflows that start to automate this, though, so that any repeatable tasks are taken care of when an issue is detected. Finally, we need to implement some action to remediate the effects of the drift: this will often be retraining the model using an updated or corrected dataset but may require a redevelopment or rewrite of key components of your model. In general, if you can build your training systems so that retraining is triggered based on an informed understanding of the drift in your models, you will save a lot of computational resources by only training when required.

The next section will discuss some of the ways we can detect drift in our models. This will help us start building up a smart retraining strategy in our solution.

Detecting data drift

So far, we have defined drift, and we know that detecting it is going to be important if we want to build sophisticated training systems. The next logical question is, *how do we do this?*

The definitions of drift we gave in the previous section were very qualitative; we can start to make these statements a bit more quantitative as we explore the calculations and concepts that can help us detect drift.

In this section, we will rely heavily on the `alibi-detect` Python package from Seldon, which, at the time of writing, is not available from [Anaconda.org](#) but is available on PyPI. To acquire this package, use the following commands:

```
pip install alibi
pip install alibi-detect
```

It is very easy to use the `alibi-detect` package. In the following example, we will work with the `wine` dataset from `sklearn`, which will be used elsewhere in this chapter. In this first example, we will split the data 50/50 and call one set the *reference* set and the other the *test* set. We will then use the Kolmogorov-Smirnov test to show that there hasn't been data drift between these two datasets, as expected, and then artificially add some drift to show that it has been successfully detected:

1. First, we must import the `TabularDrift` detector from the `alibi-detect` package, as well as the relevant packages for loading and splitting the data:

```
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
import alibi
from alibi_detect.cd import TabularDrift
```

2. Next, we must get and split the data:

```
wine_data = load_wine()
feature_names = wine_data.feature_names
X, y = wine_data.data, wine_data.target
X_ref, X_test, y_ref, y_test = train_test_split(X, y, test_size=0.2)
```

3. Next, we must initialize our drift detector using the reference data and by providing the `p-value` we want to be used by the statistical significance tests. If you want to make your drift detector trigger when smaller differences occur in the data distribution, you must select a larger `p_val`:

```
cd = TabularDrift(X_ref=X_ref, p_val=.05 )
```

4. We can now check for drift in the test dataset against the reference dataset:

```
preds = cd.predict(X_test)
labels = ['No', 'Yes']
print('Drift: {}'.format(labels[preds['data']]
```

5. This returns 'Drift: No'.
6. So, we have not detected drift here, as expected (see the following *IMPORTANT NOTE* for more on this).
7. Although there was no drift in this case, we can easily simulate a scenario where the chemical apparatus being used for measuring the chemical properties experienced a calibration error, and all the values are recorded as 10% higher than their true values. In this case, if we run drift detection again on the same reference dataset, we will get the following output:

```
X_test_cal_error = 1.1*X_test
preds = cd.predict(X_test_cal_error)
labels = ['No', 'Yes']
print('Drift: {}'.format(labels[preds['data']]
```

8. This returns 'Drift: Yes', showing that the drift has been successfully detected.

IMPORTANT NOTE

This example is very artificial but is useful for illustrating the point. In a standard dataset like this, there won't be data drift between 50% of the randomly sampled data and the



other 50% of the data. This is why we have to artificially *shift* some of the points to show that the detector does indeed work. In real-world scenarios, data drift can occur naturally due to everything from updates to sensors being used for measurements; to changes in consumer behavior; all the way through to changes in database software or schemas. So, be on guard as many drift cases won't be as easy to spot as in this case!

This example shows how, with a few simple lines of Python, we can detect a change in our dataset, which means our ML model may start to degrade in performance if we do not retrain to take the new properties of the data into account. We can also use similar techniques to track when the performance metrics of our model, for example, accuracy or mean squared error, are drifting as well. In this case, we have to make sure we periodically calculate performance on new test or validation datasets.

The first drift detection example was very simple and showed us how to detect a basic case of one-off data drift, specifically feature drift. We will now show an example of detecting **label drift**, which is basically the same but now we simply use the labels as the reference and comparison dataset. We will ignore the first few steps as they are identical, and resume from the point where we have reference and test datasets available.

1. As in the example for the drift in the features, we can configure the tabular drift detector, but now we will use the initial label as our baseline dataset:

```
cd = TabularDrift(X_ref=y_ref, p_val=.05 )
```

2. We can now check for drift in the test labels against the reference dataset:

```
preds = cd.predict(y_test)
labels = ['No', 'Yes']
print('Drift: {}'.format(labels[preds['data']]
```

3. This returns 'Drift: No'.

4. So, we have not detected drift here, as expected. Note that this method can also be used as a good sanity check that training and test data labels follow similar distributions and our sampling of test data is representative.

5. As in the previous example, we can simulate some drift in the data, and then check that this is indeed detected:

```
y_test_cal_error = 1.1*y_test
preds = cd.predict(y_test_cal_error)
labels = ['No', 'Yes']
print('Drift: {}'.format(labels[preds['data']]
```

We will now move on to a far more complex scenario, which is detecting concept drift.

Detecting concept drift

Concept drift was described in this section, and there it was emphasized that this type of drift is really all about a change in the relationships between the variables in our model. This means by definition that it is far more likely

that cases of this type will be complex and potentially quite hard to diagnose.

The most common way that you can catch concept drift is by monitoring the performance of your model through time. For example, if we are working with the `wine` classification problem again, we can look at metrics that tell us the model's classification performance, plot these through time, and then build logic around the trends and outliers that we might see in these values.

The `alibi_detect` package, which we have already been using, has several useful methods for online drift detection that can be used to find concept drift as it happens and impacts model performance. Online here refers to the fact that the drift detection takes place at the level of a single data point, so this can happen even if data comes in completely sequentially in production. Several of these methods assume that either PyTorch or TensorFlow are available as backends since the methods use **Untrained AutoEncoders (UAEs)** as out-of-the-box pre-processing methods.

As an example, let us walk through an example of creating and using one of these online detectors, the Online Maximum Mean Discrepancy method. The following example assumes that in addition to the reference dataset, `X_ref`, we have also defined variables for the expected run time, `ert`, and the window size, `window_size`. The expected run time is a variable that states the average number of data points the detector should run before it raises false positive detection. The idea here is that you want the expected run time to be larger but as it gets larger the detector becomes more insensitive to actual drift, so a balance must be struck. The `window_size` is the size of the sliding window of data used in order to calculate the appropriate drift test statistic. A smaller `window_size` means you are

tuning the detector to find sharp changes in the data or performance in a small time-frame, whereas longer window sizes will mean you are tuning to look for more subtle drift effects over longer periods of time.

1. First we import the method:

```
from alibi_detect.cd import MMDDriftOnline
```

2. We then initialize the drift detector with some variable settings as discussed in the previous paragraph. We also include the number of bootstrapped simulations we want to apply in order for the method to calculate some thresholds for detecting the drift.

Depending on your hardware settings for the deep learning library used and the size of the data, this may take some time.

```
ert = 50
window_size = 10
cd = MMDDriftOnline(X_ref, ert, window_size,
                     n_bootstraps=2500)
```

3. We can then simulate the drift detection in a production setting by taking the test data from the **Wine** dataset and feeding it in one feature vector at a time. If the feature vector for any given instance of data is given by `x`, we can then call the `predict` method of the drift detector and retrieve the '`is_drift`' value from the returned metadata like so:

```
cd.predict(x)['data'] ['is_drift']
```

4. Performing step 2 on all of the rows of the test data and plotting a vertical orange bar wherever we find drift detected gives the plot in *Figure 3.5*.

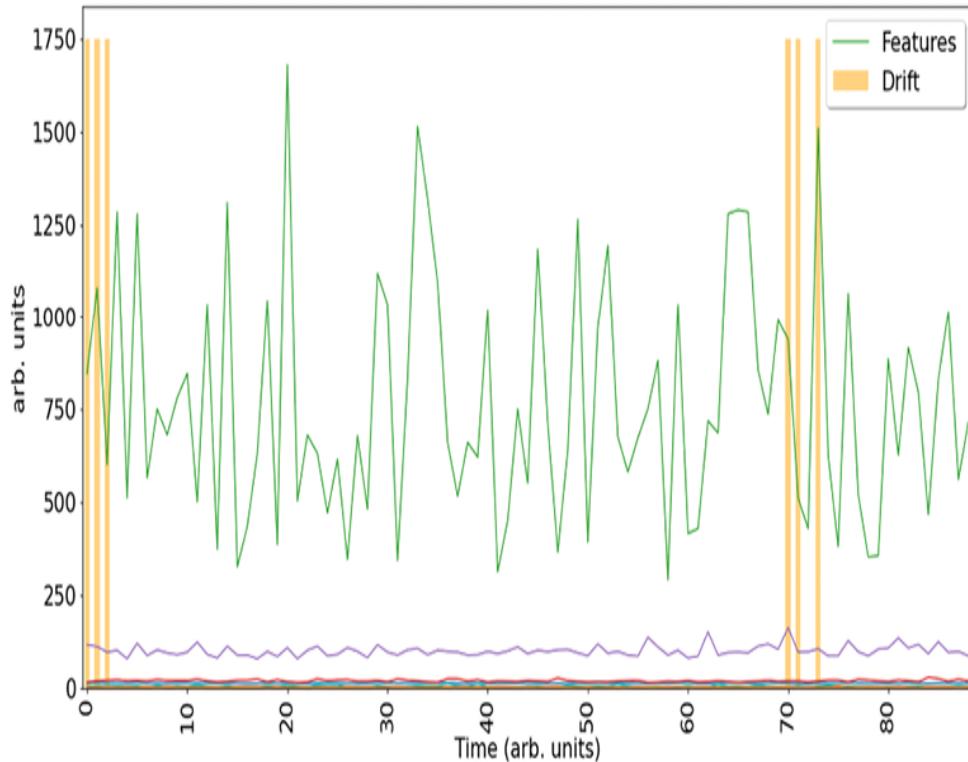


Figure 3.5: The features of the test set from the wine dataset, which we have used to run some simulated drift detection.

In this example, we can see in the plots of the simulated data that the accuracy of the data has changed over time. If we want to automate the detection of behaviors like this though, we will need to not simply plot this data but start to analyze it in a systematic way that we can fold into our model monitoring processes running in production.

Note: The test data for the **Wine** dataset was used in the drift example only as an example. In production, this drift





detection will be running on data that has never been seen before, but the principle is the same.

Now that you know drift is happening, we'll move on to discuss how you can start to decide which limits to set on your drift detectors and then cover some processes and techniques for helping you to diagnose the type and source of your drift.

Setting the limits

Many of the techniques we have been describing in this section on drift are very much aligned with standard techniques from statistics and machine learning. You can get very far using these techniques almost “out of the box” to diagnose a series of different types of issues, but we have not discussed how we can bring these together into a coherent set of drift detection mechanisms. One of the most important things to consider before setting out to do this is setting the boundaries of acceptable behavior of the data and the model so that you know when your system should raise an alarm or take some action. We will call this “setting the limits” for your drift detection system.

So, where do you start? This is where things become a bit less technical and definitely more centered around operating within a business environment, but let's cover some of the key points. First, it is important to understand what is important to alert on. Alerting on deviations in all of the metrics that you can think of might sound like a good idea, but it may just create a super noisy system where it is hard to find issues that are genuinely of concern. So, we have to be judicious in our selection of what we want to track and monitor. Next, we need to understand the timeliness required for detecting

issues. This relates very strongly to the notion in software of **Service-Level Agreements (SLAs)**, which write down the demanded and expected performance of the system in question. If your business is running real-time anomaly detection and predictive maintenance models on equipment used in hazardous conditions, it may be that the requirement for timeliness in alarms being raised and action being taken is quite high. However, if your machine learning system is performing a financial forecast once a week, then it could be that the timeliness constraints are a lot less severe. Finally, you need to set the limits. This means that you need to think carefully about the metrics you are tracking and think “What constitutes bad here?” or “What do we want to be notified of?” It may be that as part of your Discovery phase in the project, you know that the business is happy with a regression model that can have wide variability in the accuracy of its prediction, as long as it provides suitable confidence intervals.

In another scenario, it could be that the classification model you are building must have a recall that fluctuates only within a relatively tight band; otherwise, it will jeopardize the efficacy of processes downstream.

Diagnosing the drift

Although we have discussed in another section how there can be a variety of reasons for drift in our model, when it comes down to it, we must remember that machine learning models only act on features to create predictions. This then means that if we want to diagnose the source of the drift, we need to look no further than our features.

So, where do we start? The first thing we should consider is that any feature could realistically have drifted, but not all the features will be equally important in terms of the model. This means we need to understand how

important the features are before prioritizing which ones need remedial action.

Feature importance can be calculated in ways that are either model dependent or model independent. The model-dependent methods refer specifically to tree-based models, such as decision trees or random forests. In these cases, feature importance can often be extracted from the model for inspection, depending on the package used for developing the model. As an example, if we take a random forest classifier trained in Scikit-Learn, we can extract its feature importances using syntax like that given below. In this example, we retrieve the default feature importances for the random forest model, which are calculated using **Mean Decrease in Impurity (MDI)**, equivalently known as “Gini importance,” and put them in an ordered pandas series for later analysis:

```
import pandas as pd
feature_names = rf[:-1].get_feature_names_out()
mdi_importances = pd.Series(rf[-1].feature_impo
                            index=feature_names
```

Although this is extremely simple, it can sometimes give erroneous results for a couple of reasons. The feature importances here have been calculated using an impurity measure, which is a class of measures that can exhibit bias toward features with high cardinality (e.g., numerical) and are computed only on training set data, meaning they do not take into account any generalizability of the model onto unseen test data. This should always be kept in mind when using this sort of importance measure.

Another standard measure of feature importance, which is model agnostic and alleviates some of the issues for MDI or Gini importance, is the permutation importance.

This works by taking the feature we are interested in, shuffling it (i.e., moving the values in the column of the feature matrix up, down, or via some other method of reorganization), and then recalculating the model accuracy or error. The change in the accuracy or error can then be used as a measure of the importance of this feature, as fewer important features should mean less change in model performance upon shuffling. Below is an example of this method, again using Scikit-Learn, on the same model we used in the previous example:

```
from sklearn.inspection import permutation_impo
result = permutation_importance(
    rf, X_test, y_test, n_repeats=10, random_st
)
sorted_importances_idx = result.importances_mean
importances = pd.DataFrame(
    result.importances[sorted_importances_idx].
    columns=X.columns[sorted_importances_idx])
```

Finally, one other very popular method for determining feature importance is the calculation of **SHAP (SHapley Additive exPlanation)** values for the features. This uses ideas from game theory to consider how the features combine to inform the prediction. SHAP values are calculated by training the model on all permutations of features that include or exclude the considered feature and then calculating the marginal contribution to the predicted value of that feature. This is different from permutation

importance because we are no longer simply permuting the feature values; we are now actually running through a series of different potential sets of features including or excluding the feature.

You can start calculating SHAP values on your models by installing the `shap` package:

```
pip install shap
```

And then we can execute syntax like the following, using the same random forest model in the previous examples to define a *shap explainer* object and calculate the SHAP values for the features in the test dataset. We assume here the `X_test` is a pandas DataFrame with the feature names as the column names:

```
explainer = shap.Explainer(rf, predict, X_test)
shap_values = explainer(X_test)
```

Note that the calculation of the SHAP values can take some time due to running all the permutations. The `shap_values` themselves are not feature importances, but contain the SHAP values calculated for each feature in all the different feature combination experiments. In order to determine feature importances, you should take the average of the absolute magnitudes of the `shap_values` for each feature. This is done for you and the result plotted if you use the following command:

```
shap.plots.bar(shap_values)
```

We have now covered three different ways to calculate feature importances for your models, two of them completely model agnostic. Feature importance is extremely helpful to help you get to the root of drift very quickly. If you see the performance of your model drifting or breaching a threshold you have set, you can use the feature importances to focus your diagnostic efforts on where the most important features are and ignore drift in features that are not as critical.

Now that we have covered a useful way to help dig into the drift, we will now discuss how you can go about remediating it once you spot the feature or features that seem to be causing the most trouble.

Remediating the drift

There are a few ways we can take action against drift in order to maintain the performance of our system:

- **Remove features and retrain:** If certain features are drifting or exhibiting degradation of some other kind, we can try removing them and retraining the model. This can become time consuming as our data scientists potentially need to re-run some analysis and testing to ensure that this approach still makes sense from a modeling point of view. We also have to take into account things like the importance of the features we are removing.
- **Retrain with more data:** If we are seeing concept drift, we may simply be noticing that the model has become stale with respect to the distributions and the relationships between these distributions in the data. It could be that retraining the model and including more recent data may create an uptick in performance. There is also the option of retraining the model on some selected portion of more recent data.

This can be especially useful if you are able to diagnose some dramatic event or shift in the data, for example, the introduction of Covid-19 lockdowns. This approach can be hard to automate though, so sometimes it is also an option to introduce a time-windowed approach, where you train on some preselected amount of data up to the present time.

- **Roll back the model:** We can replace the current model with a previous version or even a go-to baseline model. This can be a very good approach if your baseline model is more simple but also more predictable in terms of performance, because it applies some simple business logic, for example. The ability to roll back to previous versions of models requires that you have built up a good set of automated processes around your model registry. This is very reminiscent of rollbacks in general software engineering, a key component of building robust systems.
- **Rewrite or debug the solution:** It may be the case that the drift we are dealing with is so substantial that the model as it stands cannot cope with any of the above approaches. The idea of rewriting the model may seem drastic but this can be more common than you think. As an example, consider that initially you deploy a well-tuned LightGBM model that performs binary classification on a set of five features daily. After running the solution for months, it could be that after detecting drift in the model performance several times, you decide that it is better to perform an investigation to see if there is a better approach. This can be especially helpful in this scenario as now you know more about the data you will see in production. You may then discover that actually, a random forest classifier is not as performant on the same production data scenarios on average but that *it is* more stable,

behaving more consistently and triggering drift alarms less often. You may then decide that actually, it is better for the business to deploy this different model into the same system as it will reduce operational overheads from dealing with the drift alarms and it will be something the business can trust more. It is important to note that if you need to write a new pipeline or model, it is often important to roll back to a previous model while the team does this work.

- **Fix the data source:** Sometimes, the most challenging issues do not actually have anything to do with the underlying model but are more to do with changes in how data is collected and fed downstream to your system. There are so many business scenarios where the collection of data, the transformation of data, or the characteristics of data may be changed due to the introduction of new processes, updates to systems, or even due to changes in the personnel responsible for entering some source data. A great example from the author's own experience is when it comes to **customer resource management (CRM)** systems, the quality of the data being input from the sales team can depend on so many factors that it can be reasonable to expect slow or sudden changes in data quality, consistency, and timeliness.

In this case, the right answer may not actually be an engineering one, but a process one, working with the appropriate teams and stakeholders to ensure that data quality is maintained, and standard processes are followed. This will benefit customers and the business, but it can still be a hard sell.

Now, we can start to build this into solutions that will automatically trigger our ML model being retrained, as shown in *Figure 3.6*:

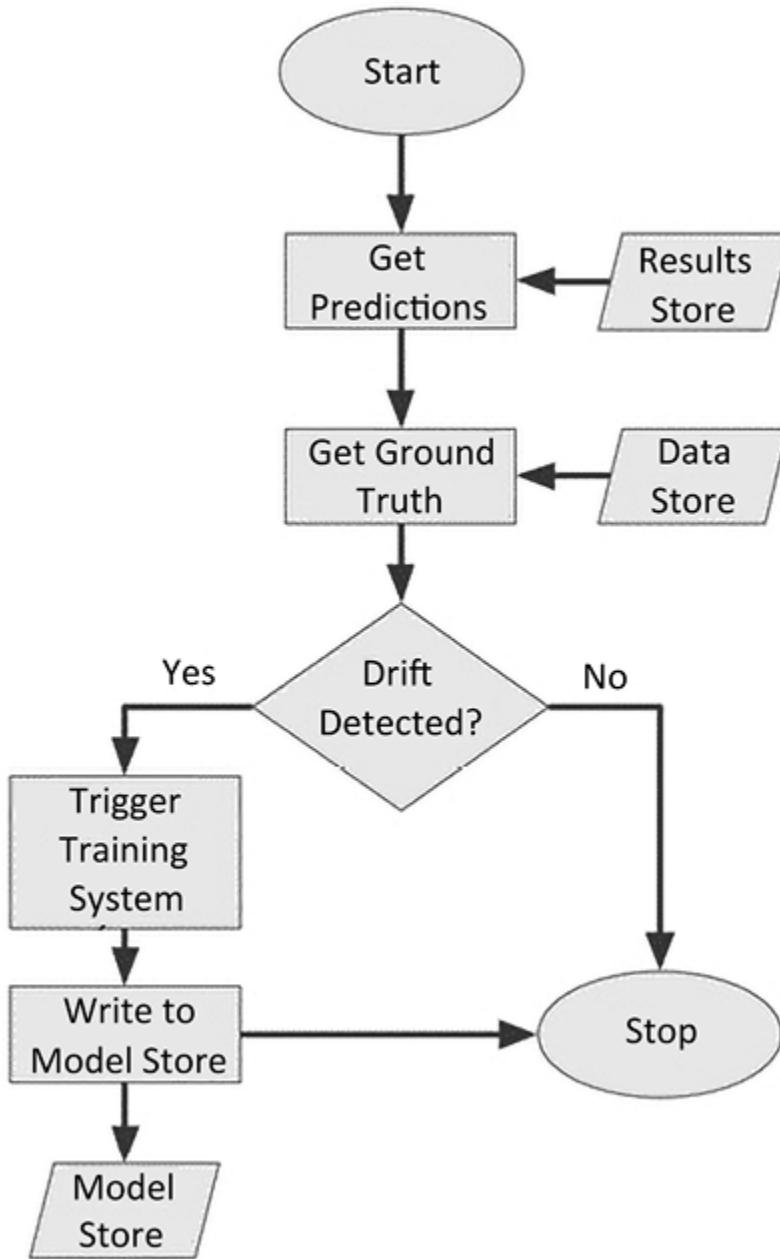


Figure 3.6: An example of drift detection and the training system process.

Other tools for monitoring

The examples in this chapter have mainly used the alibi-detect package, but we are now in somewhat of a golden age of open source **MLOps** tools.

There are several different packages and solutions available that you can start using to build your monitoring solutions without spending a penny.

In this section, we will quickly cover some of these tools and show some basic points on their syntax, so that if you want to develop monitoring pipelines, then you can just get started right away and know where is best to use these different tools.

First, we will cover **Evidently AI**

(<https://www.evidentlyai.com/>), which is a very easy-to-use Python package that allows users to not only monitor their models but also create customizable dashboards in a few lines of syntax. Below is an adaptation of the getting started guide from the documentation.

1. First, install Evidently:

```
pip install evidently
```

2. Import the `Report` functionality. The `Report` is an object that collects calculations across several metrics to allow for visualizations or outputs as a JSON object. We will show this latter behavior later:

```
from evidently.report import Report
```

3. Next, import what is known as a metric preset, in this case for data drift. We can think of this as a templated report object that we can later customize:

```
from evidently.metric_preset import DataDrif
```

4. Next, assuming you have the data to hand, you can then run the data drift report. Let's assume you have the **Wine** dataset from the previous examples to hand. If we split the wine data 50/50 using `scikit-learn`'s `train_test_split()` method, we will have two datasets, which we again use to simulate the reference dataset, `X_ref`, and the current dataset, `X_curr`:

```
data_drift_report = Report(metrics=[  
    DataDriftPreset(),  
    ])  
report.run(  
    reference_data=X_ref,  
    current_data=X_ref  
)
```

5. Evidently then provides some really nice functionality for visualizing the results in the report. You can export or view these using a few different methods. You can export the report to JSON or HTML objects for consumption or to review downstream or in other applications. *Figures 3.7 and 3.8* show snippets of the results when you create these outputs with the following commands:

```
data_drift_report.save_json('data_drift_repc  
data_drift_report.save_html('data_drift_repc
```

```
{
  "version": "0.3.1",
  "timestamp": "2023-05-01 14:55:28.221095",
  "metrics": [
    {
      "metric": "DatasetDriftMetric",
      "result": {
        "drift_share": 0.5,
        "number_of_columns": 13,
        "number_of_drifted_columns": 0,
        "share_of_drifted_columns": 0.0,
        "dataset_drift": false
      }
    },
    {
      "metric": "DataDriftTable",
      "result": {
        "number_of_columns": 13,
        "number_of_drifted_columns": 0,
        "share_of_drifted_columns": 0.0,
        "dataset_drift": false,
        "drift_by_columns": {
          "alcalinity_of_ash": {
            "column_name": "alcalinity_of_ash",
            "column_type": "num",
            "stattest_name": "K-S p_value",
            "stattest_threshold": 0.05,
            "drift_score": 0.39530698914758006,
            "drift_detected": false,
            "current": {
              "small_distribution": {
                "x": [
                  14.6,
                  15.99,
                  17.38,
                  18.77,
                  20.16,
                  21.55,
                  22.939999999999998,
                  24.33,
                  25.72,
                  27.11,
                  28.5
                ],
                "y": [
                  0.06466736722981163,
                  0.16975183897825574
                ]
              }
            }
          }
        }
      }
    }
  ]
}
```

Figure 3.7: JSON output from the Evidently report on the 50/50 split Wine feature set.



Figure 3.8: HTML version of the drift report generated by Evidently on the 50/50 split Wine feature set.

One of the nice things about the rendered HTML report is that you can dynamically drill down into some useful information. As an example, *Figure 3.9* shows that if you click down into any of the features, you are provided with a plot of data drift through time, and *Figure 3.10* shows that you can also get a plot of the distributions of the features in the same way.



Figure 3.9: The automatically generated data drift plot when you drill down in the Evidently report for the Wine features.

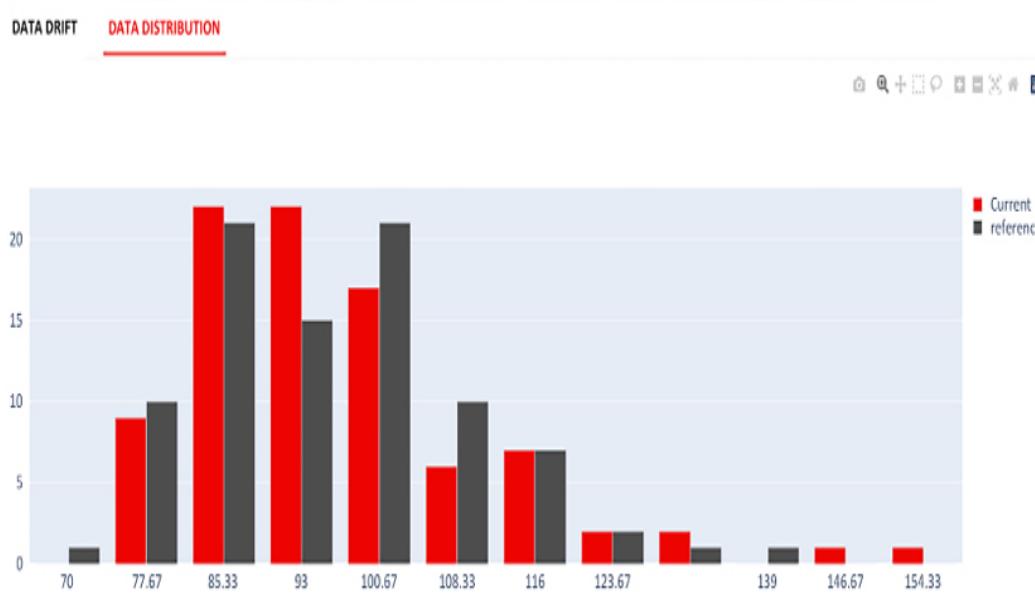


Figure 3.10: The automatically generated histogram showing the distribution of the feature when you drill down in the Evidently report for the Wine feature set.

This has only scratched the surface of what you can do with Evidently. There is a lot of functionality available for generating your own model test

suites and monitoring functionality as well as visualizing it all nicely like we have seen.

Now that we have explored the concepts of model and data drift and how to detect them, we can now move on to a discussion about how we can take a lot of the concepts we covered earlier in the chapter and automate them.

The next couple of sections will provide deep dives into different aspects of the training process and, in particular, how this process can be automated using a variety of tools.

Automating training

The training process is an integral part of the model factory and one of the main differentiators between ML engineering and traditional software engineering. The next few sections will discuss in detail how we can start to use some excellent open source tooling to streamline, optimize, and, in some cases, fully automate elements of this process.

Hierarchies of automation

One of the main reasons that ML is now a common part of software development, as well as a major business and academic activity, is because of the plethora of tools available. All of the packages and libraries containing working and optimized implementations of sophisticated algorithms have allowed people to build on top of these, rather than have to reimplement the basics every time there is a problem to solve.

This is a powerful expression of the idea of **abstraction** in software development, where lower-level units can be leveraged and engaged with at higher levels of implementation.

This idea can be extended even further to the entire enterprise of training itself. At the lowest level of implementation (but still a very high level in the sense of the underlying algorithms), we can provide details about how we want the training process to go. We can manually define the exact set of hyperparameters (see the next section on *Optimizing hyperparameters*) to use in the training run in our code. I call this **hand cranking**. We can then move one level of abstraction up and supply ranges and bounds for our hyperparameters to tools designed to efficiently sample and test our model's performance for each of these; for instance, *automated hyperparameter tuning*. Finally, there is one higher level of abstraction that has created a lot of media excitement over the past few years, where we optimize over which algorithm to run. This is known as **automated ML** or **AutoML**.

There can be a lot of hype surrounding AutoML, with some people proclaiming the eventual automation of all ML development job roles. In my opinion, this is just not realistic, as selecting your model and hyperparameters is only one aspect of a hugely complex engineering challenge (hence this being a book and not a leaflet!). AutoML is, however, a very powerful tool that should be added to your arsenal of capabilities when you go into your next ML project.

We can summarize all of this quite handily as a *hierarchy of automation*; basically, how much control do you, as the ML engineer, want in the training process? I once heard this described in terms of gear control in a car (credit: *Databricks at Spark AI 2019*). Hand cranking is the equivalent of driving a manual car with full control over the gears: there's more to think about, but it can be very efficient if you know what you're doing. One level up, you have automatic cars: there's less to worry about so that you can focus more on getting to your destination, traffic, and other challenges. This is a good option for a lot of people but still requires you to have

sufficient knowledge, skills, and understanding. Finally, we have self-driving cars: sit back, relax, and don't even worry about how to get where you're going. You can focus on what you are going to do once you get there.

This *hierarchy of automation* is shown in the following diagram:

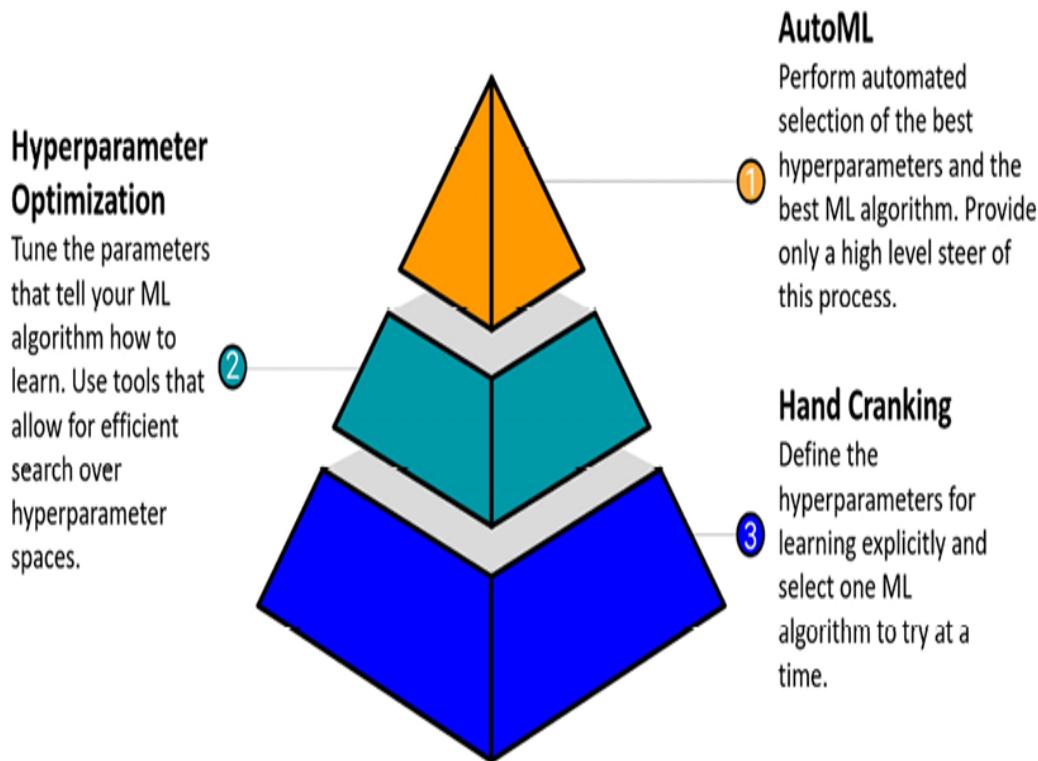


Figure 3.11: The hierarchy of automation of ML model optimization, with AutoML as the most automated possibility.

That, in a nutshell, is how the different levels of training abstraction link together.

In the next few sections, we will discuss how to get started building out implementations of hyperparameter optimization and AutoML. We will not cover “hand cranking” as that is self-explanatory.

Optimizing hyperparameters

When you fit some sort of mathematical function to data, some values are tuned during the fitting or training procedure: these are called **parameters**. For ML, there is a further level of abstraction where we have to define the values that tell the algorithms we are employing *how they should update the parameters*. These values are called **hyperparameters**, and their selection is one of the important *dark arts* of training ML algorithms.

The following tables list some hyperparameters that are used for common ML algorithms to show you the different forms they may take. These lists are not exhaustive but are there to highlight that hyperparameter optimization is not a trivial exercise:

Algorithm	Hyperparameters	What This Controls
Decision Trees and Random Forests	<ul style="list-style-type: none">Tree depth.Min/max leaves.	<ul style="list-style-type: none">How many levels are in your trees.How much branching can occur at each level.
Support Vector Machines	<ul style="list-style-type: none">CGamma	<ul style="list-style-type: none">Penalty for misclassification.The radius of influence of the training points for Radial Basis Function (RBF) kernels.
Neural Networks	<ul style="list-style-type: none">Learning rate.	<ul style="list-style-type: none">Update step sizes.How deep your network is.

(numerous architectures)	<ul style="list-style-type: none"> Number of hidden layers. Activation function. Many more. 	<ul style="list-style-type: none"> The firing conditions of your neurons.
Logistic Regression	<ul style="list-style-type: none"> Solver Regularization type. Regularization prefactor. 	<ul style="list-style-type: none"> How to minimize the loss. How to prevent overfitting/make the problem well behaved. The strength of the regularization type.

Table 3.1: Some hyperparameters and what they control for some supervised algorithms.

Further examples can be seen in the following table:

Algorithm	Hyperparameters	What This Controls
K-Nearest Neighbors	<ul style="list-style-type: none"> K Distance metric. 	<ul style="list-style-type: none"> The number of clusters. How to define the distance between points.
DBSCAN	<ul style="list-style-type: none"> Epsilon Minimum number of samples. Distance metric. 	<ul style="list-style-type: none"> The max distance to be considered neighbors. How many neighbors are required to be considered core. How to define the distance between points.

Table 3.2: Some hyperparameters and what they control for some unsupervised algorithms.

All of these hyperparameters have their own specific set of values they can take. This range of hyperparameter values for the different potential algorithms you want to apply to your ML solution means that there are a lot of ways to define a *working* model (meaning one that doesn't break the implementation you are using), but how do you find the *optimal* model?

This is where hyperparameter search comes in. The concept is that for a finite number of hyperparameter value combinations, we want to find the set that gives the best model performance. This is another optimization problem that's similar to that of training in the first place!

In the following sections, we will discuss two very popular hyperparameter optimization libraries and show you how to implement them in a few lines of Python.

IMPORTANT NOTE

It is important to understand which algorithms are being used for optimization in these hyperparameter libraries, as you may want to use a couple of different implementations from each to compare different approaches and assess performance. If you didn't look at how they were working under the hood, you could easily make unfair comparisons – or worse, you could be comparing almost the same thing without knowing it! If you have some deeper knowledge of how these solutions work, you will also be able to make better judgment calls as to when they will be beneficial and when they will be overkill. Aim to have a working knowledge of a few of these algorithms and approaches, since this will help you design more holistic training



systems with algorithm-tuning approaches that complement one another.

Hyperopt

Hyperopt is an open source Python package that bills itself as being *for serial and parallel optimization over awkward search spaces, which may include real-valued, discrete, and conditional dimensions*. Check out the following link for more information:

<https://github.com/Hyperopt/Hyperopt>. At the time of writing, version 0.2.5 comes packaged with three algorithms for performing optimization over user-provided search spaces:

- **Random search:** This algorithm essentially selects random numbers within your provided ranges of parameter values and tries them. It then evaluates which sets of numbers provide the best performance according to your chosen objective function.
- **Tree of Parzen Estimators (TPE):** This is a Bayesian optimization approach that models distributions of hyperparameters below and above a threshold for the objective function (roughly *good* and *bad* scorers), and then aims to draw more values from the *good* hyperparameter distribution.
- **Adaptive TPE:** This is a modified version of TPE that allows for some optimization of the search, as well as the ability to create an ML model to help guide the optimization process.

The Hyperopt repository and documentation contain several nice and detailed worked examples. We will not go through these here. Instead, we will learn how to use this for a simple classification model, such as the one we defined in *Chapter 1, Introduction to ML Engineering*. Let's get started:

1. In Hyperopt, we must define the hyperparameters that we want to optimize across. For example, for a typical logistic regression problem, we could define the space of hyperparameters to cover, whether we want to reuse parameters that were learned from the previous model runs each time (`warm_start`), whether we want the model to include a bias in the decision function (`fit_intercept`), the tolerance set for deciding when to stop the optimization (`tol`), the regularization parameter (`C`), which `solver` we want to try, and the maximum number of iterations, `max_iter`, in any training run:

```
from Hyperopt import hp

space = {
    'warm_start' : hp.choice('warm_start', [
        'fit_intercept' : hp.choice('fit_intercept', [
            'tol' : hp.uniform('tol', 0.00001, 0.0001),
            'C' : hp.uniform('C', 0.05, 2.5),
            'solver' : hp.choice('solver', [
                'newton-cg',
                'lbfgs',
                'liblinear',
                'sag',
                'saga'
            ])
        ])
    ])
}
```

2. Then, we have to define an objective function to optimize. In the case of our classification algorithm, we can simply define the `loss` function we want to minimize as 1 minus the `f1-score`. Note that Hyperopt allows your objective function to supply run statistics and metadata via your return statement if you are using the `fmin` functionality. The only requirement if you do this is that you return a

value labeled `loss` and a valid status value from the list of `Hyperopt.STATUS_STRING` (`ok` by default and `fail` if there is an issue in the calculation that you want to call out as a failure):

```
def objective(params, n_folds, X, y):
    # Perform n_fold cross validation with h
    clf = LogisticRegression(**params, random_state=42)
    scores = cross_val_score(clf, X, y, cv=n_folds,
                             scoring='f1_macro')

    # Extract the best score
    max_score = max(scores)
    # Loss must be minimized
    loss = 1 - max_score
    # Dictionary with information for evaluation
    return {'loss': loss, 'params': params}
```

3. Now, we must optimize using the `fmin` method with the **TPE** algorithm:

```
# Trials object to track progress
trials = Trials()
# Optimize
best = fmin(
    fn=partial(objective, n_folds=n_folds, X=X, y=y),
    space=space,
    algo=tpe.suggest,
    max_evals=16,
    trials=trials
)
```

4. The content of `best` is a dictionary containing all the best hyperparameters in the search space you defined. So, in this case, we have the following:

```
{'C': 0.26895003542493234,  
 'fit_intercept': 1,  
 'max_iter': 452,  
 'solver': 2,  
 'tol': 1.863336145787027e-05,  
 'warm_start': 1}
```

You can then use these hyperparameters to define your model for training on the data.

Optuna

Optuna is a software package that has an extensive series of capabilities based on some core design principles, such as its **define-by-run** API and modular architecture. *Define-by-run* here refers to the fact that, when using Optuna, the user does not have to define the full set of parameters to test, which is *define-and-run*. Instead, they can provide some initial values and ask Optuna to suggest its own set of experiments to run. This saves the user time and reduces the code footprint (two big pluses for me!).

Optuna contains four basic search algorithms: **grid search**, **random search**, **TPE**, and the **Covariance Matrix Adaptation Evolution Strategy (CMA-ES)** algorithm. We covered the first three previously, but CMA-ES is an important addition to the mix. As its name suggests, this is based on an evolutionary algorithm and draws samples of hyperparameters from a multivariate Gaussian distribution. Then, it uses the rankings of the

evaluated scores for the given objective function to dynamically update the parameters of the Gaussian distribution (the covariance matrix being one set of these) to help find an optimum over the search space quickly and robustly.

The key thing that makes Optuna’s optimization process different from Hyperopt, however, is in its application of **pruning** or **automated early stopping**. During optimization, if Optuna detects evidence that a trial of a set of hyperparameters will not lead to a better overall trained algorithm, it terminates that trial. The developers of the package suggest that this leads to overall efficiency gains in the hyperparameter optimization process by reducing unnecessary computation.

Here, we're looking at the same example we looked at previously, but we are now using Optuna instead of Hyperopt:

1. First, when using Optuna, we can work using an object known as `Study`, which provides us with a convenient way to fold our search space into our `objective` function:

```
def objective(trial, n_folds, X, y):
    """Objective function for tuning logistic
    params = {
        'warm_start':
            trial.suggest_categorical('warm_star',
        'fit_intercept':
            trial.suggest_categorical('fit_inter',
        'tol': trial.suggest_uniform('tol',
        'C': trial.suggest_uniform('C', 0.05,
        'solver': trial.suggest_categorical(
```

```
'max_iter': trial.suggest_categorical  
    }  
    # Perform n_fold cross validation with h  
    clf = LogisticRegression(**params, random_state=42)  
    scores = cross_val_score(clf, X, y, cv=n_folds,  
                             scoring='f1_macro')  
    # Extract the best score  
    max_score = max(scores)  
    # Loss must be minimized  
    loss = 1 - max_score  
    # Dictionary with information for evaluation  
    return loss
```

2. Now, we must set up the data in the same way as we did in the Hyperopt example:

```
n_folds = 5  
X, y = datasets.make_classification(n_samples=100, n_features=20, n_informative=2, n_redundant=10, n_clusters_per_class=1, class_sep=0.8)  
train_samples = 100 # Samples used for training  
X_train = X[:train_samples]  
X_test = X[train_samples:]  
y_train = y[:train_samples]  
y_test = y[train_samples:]
```

3. Now, we can define this `Study` object that we mentioned and tell it how we wish to optimize the value that's returned by our `objective` function, complete with guidance on how many trials to

run in the `study`. Here, we will use the TPE sampling algorithm again:

```
from optuna.samplers import TPESampler
study = optuna.create_study(direction='minimize')
study.optimize(partial(objective, n_folds=n_train), n_trials=16)
```

4. Now, we can access the best parameters via the `study.best_trial.params` variable, which gives us the following values for the best case:

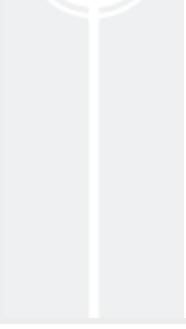
```
{'warm_start': False,
'fit_intercept': False,
'tol': 9.866562116436095e-05,
'C': 0.08907657649508408,
'solver': 'newton-cg',
'max_iter': 108}
```

As you can see, Optuna is also very simple to use and very powerful. Now, let's look at the final level of the hierarchy of automation: AutoML.

IMPORTANT NOTE

You will notice that these values are different from the ones returned by Hyperopt. This is because we have only run 16 trials in each case, so we are not effectively subsampling the space. If you run either of the Hyperopt or Optuna samples a few times in a row, you can get quite different results for





the same reason. The example given here is just to show the syntax, but if you are keen, you can set the number of iterations to be very high (or create smaller spaces to sample), and the results of the two approaches should roughly converge.

AutoML

The final level of our hierarchy is the one where we, as the engineer, have the least direct control over the training process, but where we also potentially get a good answer for very little effort!

The development time that's required to search through many hyperparameters and algorithms for your problem can be large, even when you code up reasonable-looking search parameters and loops.

Given this, the past few years have seen the deployment of several **AutoML** libraries and tools in a variety of languages and software ecosystems. The hype surrounding these techniques has meant they have had a lot of airtime, which has led to several data scientists questioning when their jobs will be automated away. As we mentioned previously in this chapter, in my opinion, declaring the death of data science is extremely premature and also dangerous from an organizational and business performance standpoint. These tools have been given such a pseudo-mythical status that many companies could believe that simply using them a few times will solve all their data science and ML problems.

They are wrong, but they are also right.

These tools and techniques *are* very powerful and *can* help make some things better, but they are not a magical *plug-and-play* panacea. Let's

explore these tools and start to think about how to incorporate them into our ML engineering workflow and solutions.

auto-sklearn

One of our favorite libraries, good old Scikit-Learn, was always going to be one of the first targets for building a popular AutoML library. One of the very powerful features of auto-sklearn is that its API has been designed so that the main objects that optimize and section models and hyperparameters can be swapped seamlessly into your code.

As usual, an example will show this more clearly. In the following example, we will assume that the `Wine` dataset (a favorite for this chapter) has already been retrieved and split into train and test samples in line with other examples, such as the one in the *Detecting drift* section:

1. First, since this is a classification problem, the main thing we need to get from `auto-sklearn` is the `autosklearn.classification` object:

```
import numpy as np
import sklearn.datasets
import sklearn.metrics
import autosklearn.classification
```

2. We must then define our `auto-sklearn` object. This provides several parameters that help us define how the model and hyperparameter tuning process will proceed. In this example, we will provide an upper time limit in seconds for running the overall

optimization and an upper time limit in seconds for any single call to the ML model:

```
automl = autosklearn.classification.AutoSklearnClassifier(  
    time_left_for_this_task=60,  
    per_run_time_limit=30  
)
```

3. Then, just like we would fit a normal `sklearn` classifier, we can fit the `auto-sklearn` object. As we mentioned previously, the `auto-sklearn` API has been designed so that this looks familiar:

```
automl.fit(X_train, y_train, dataset_name='WIKIART')
```

4. Now that we've fit the object, we can start to dissect what has been achieved by the object during its optimization run.
5. First, we can see which models were tried and which were kept in the object as part of the final ensemble:

```
print(automl.show_models())
```

6. We can then get a readout of the main statistics from the run:

```
print(automl.print_statistics())
```

7. Then, we can predict some text features, as expected:

```
predictions = automl.predict(X_test)
```

- Finally, we can check how well we did by using our favorite metric calculators – in this case, the `sklearn metrics` module:

```
sklearn.metrics.accuracy_score(y_test, predi
```

- As you can see, it is very straightforward to start using this powerful library, especially if you are already comfortable working with `sklearn`.

Next, let's discuss how we extend this concept to neural networks, which have an extra layer of complexity due to their different potential model architectures.

AutoKeras

A particular area where AutoML has been a big hit is neural networks. This is because for a neural network, the question of *what is the best model?* is a very complicated one. For our typical classifiers, we can usually think of a relatively short, finite list of algorithms to try. For a neural network, we don't have this finite list. Instead, we have an essentially infinite set of possible neural network *architectures*; for instance, for organizing the neurons into layers and the connections between them. Searching for the optimal neural network architecture is a problem in which powerful optimization can make your life, as an ML engineer or data scientist, a whole lot easier.

In this instance, we are going to explore an AutoML solution built on top of the very popular neural network API library known as Keras. Unbelievably, the name of this package is – you guessed it – AutoKeras!

For this example, we will, once again, assume that the `Wine` dataset has been loaded so that we can focus on the details of the implementation. Let's get started:

1. First, we must import the `autokeras` library:

```
import autokeras as ak
```

2. Now, it's time for the fun and, for `autokeras`, the extremely simple bit! Since our data is structured (tabular with a defined schema), we can use the `StructuredDataClassifier` object, which wraps the underlying mechanisms for automated neural network architecture and hyperparameter search:

```
clf = ak.StructuredDataClassifier(max_trials
```

3. Then, all we have to do is fit this classifier object, noticing its similarity to the `sklearn` API. Remember that we assume that the training and test data exist in `pandas DataFrames`, as in the other examples in this chapter:

```
clf.fit(x=X_train, y=y_train)
```

4. The training objects in AutoKeras have a convenient evaluation method wrapped within them. Let's use this to see how accurate our

solution was:

```
accuracy=clf.evaluate(x=X_train, y=y_train)
```

5. With that, we have successfully performed a neural network architecture and hyperparameter search in a few lines of Python. As always, read the solution documentation for more information on the parameters you can provide to the different methods.

Now that we've covered how to create performant models, in the next section, we will learn how to persist these models so that they can be used in other programs.

Persisting your models

In the previous chapter, we introduced some of the basics of model version control using MLflow. In particular, we discussed how to log metrics for your ML experiments using the MLflow Tracking API. We are now going to build on this knowledge and consider the touchpoints our training systems should have with model control systems in general.

First, let's recap what we're trying to do with the training system. We want to automate (as far as possible) a lot of the work that was done by the data scientists in finding the first working model, so that we can continually update and create new model versions that still solve the problem in the future. We would also like to have a simple mechanism that allows the results of the training process to be shared with the part of the solution that will carry out the prediction when in production. We can think of our model version control system as a bridge between the different stages of the ML

development process we discussed in *Chapter 2, The Machine Learning Development Process*. In particular, we can see that the ability to track experiment results allows us to keep the results of the **Play** phase and build on these during the **Develop** phase. We can also track more experiments, test runs, and hyperparameter optimization results in the same place during the **Develop** phase. Then, we can start to tag the performant models as ones that are good candidates for deployment, thus bridging the gap between the **Develop** and **Deploy** development phases.

If we focus on MLflow for now (though plenty of other solutions are available that fulfill the need for a model version control system), then MLflow's Tracking and Model Registry functionalities nicely slot into these bridging roles. This is represented schematically in the following diagram:

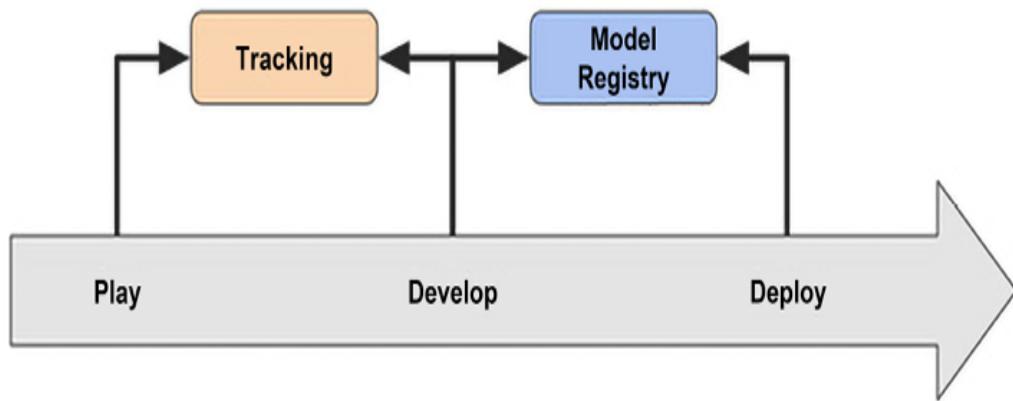


Figure 3.12: How the MLflow Tracking and Model Registry functionalities can help us progress through the different stages of the ML development process.

In *Chapter 2, The Machine Learning Development Process*, we only explored the basics of the MLflow Tracking API for storing experimental model run metadata. Now, we will briefly dive into how to store production-ready models in a very organized way so that you can start to perform model staging. This is the process whereby models can be

progressed through stages of readiness, and you can swap models in and out of production if you wish to. This is an extremely important part of any training system that supplies models and will run as part of a deployed solution, which is what this book is all about!

As alluded to previously, the functionality that we need in MLflow is called **Model Registry**, which enables you to manage the staging of models across your development life cycle. Here, we will walk through examples of how to take a logged model and push it to the registry, how to update information such as the model version number in the registry, and then how to progress your model through different life cycle stages. We will finish this section by learning how to retrieve a given model from the registry in other programs – a key point if we are to share our models between separate training and prediction services.

Before we dive into the Python code for interacting with Model Registry, we have one important piece of setup to perform. The registry only works if a database is being used to store the model metadata and parameters. This is different from the basic Tracking API, which works with just a file backend store. This means that before pushing models to Model Registry, we have to fire up an MLflow server with a database backend. You can do this with a **SQLite** database running locally by executing the following command in your terminal.

You will have to run this before the code snippets in the rest of this section (this command is stored in a short Bash script in this book's GitHub repository, under

<https://github.com/PacktPublishing/Machine-Learning-Engineering-with->

Python/blob/main/Chapter03/mlflow-advanced/start-mlflow-server.sh):

```
mlflow server \
    --backend-store-uri sqlite:///mlflow.db \
    --default-artifact-root ./artifacts \
    --host 0.0.0.0
```

Now that the backend database is up and running, we can use it as part of our model workflow. Let's get started:

1. Let's begin by logging some metrics and parameters for one of the models we trained earlier in this chapter:

```
    metrics.recall_score(y_test, y
                          average=""
                        })
mlflow.log_params(params)
```

- Inside the same code block, we can now log the model to Model Registry, providing a name for the model to reference later:

```
mlflow.sklearn.log_model(
    sk_model=std_scale_clf,
    artifact_path="sklearn-model",
    registered_model_name="sk-learn-"
)
```

- Now, let's assume we are running a prediction service and we want to retrieve the model and predict using it. Here, we have to write the following:

```
model_name = "sk-learn-std-scale-clf"
model_version = 1
model = mlflow.pyfunc.load_model(
    model_uri=f"models:{model_name}/{model_"
)
model.predict(X_test)
```

- By default, newly registered models in Model Registry are assigned the '**'Staging'** stage value. Therefore, if we want to retrieve the

model based on knowing the stage but not the model version, we could execute the following code:

```
stage = 'Staging'  
model = mlflow.pyfunc.load_model(  
    model_uri=f"models:{model_name}/{stage}"  
)
```

5. Based on all of our discussions in this chapter, the result of our training system must be able to produce a model we are happy to deploy to production. The following piece of code promotes the model to a different stage, called "Production":

```
client = MlflowClient()  
client.transition_model_version_stage(  
    name="sk-learn-std-scale-clf",  
    version=1,  
    stage="Production"  
)
```

6. These are the most important ways to interact with Model Registry and we have covered the basics of how to register, update, promote, and retrieve your models in your training (and prediction) systems.

Now, we will learn how to chain our main training steps together into single units called **pipelines**. We will cover some of the standard ways of doing this inside single scripts, which will allow us to build our first training pipelines. In *Chapter 5, Deployment Patterns and Tools*, we will cover tools

for building more generic software pipelines for your ML solution (of which your training pipeline may be a single component).

Building the model factory with pipelines

The concept of a software pipeline is intuitive enough. If you have a series of steps chained together in your code, so that the next step consumes or uses the output of the previous step or steps, then you have a pipeline.

In this section, when we refer to a pipeline, we will be specifically dealing with steps that contain processing or calculations that are appropriate to ML. For example, the following diagram shows how this concept may apply to some of the steps the marketing classifier mentioned in *Chapter 1, Introduction to ML Engineering*:

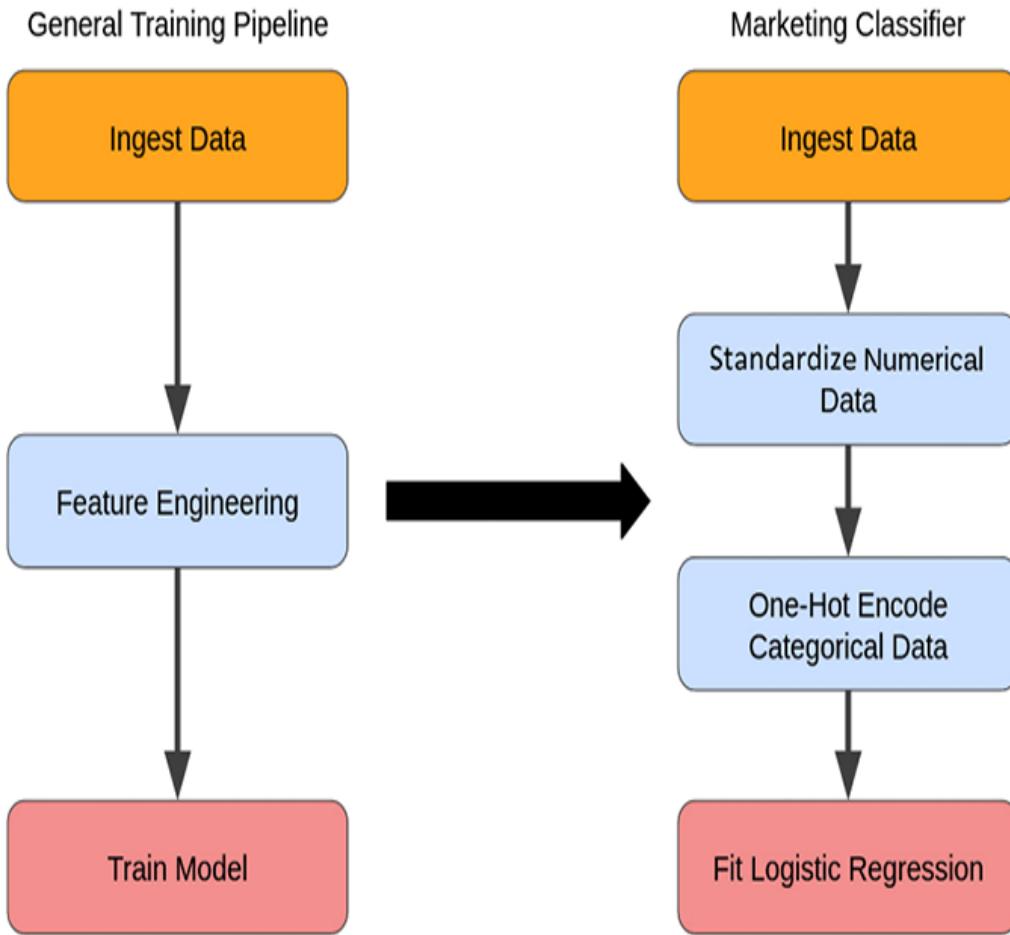


Figure 3.13: The main stages of any training pipeline and how this maps to a specific case from Chapter 1, Introduction to ML Engineering.

Let's discuss some of the standard tools for building up your ML pipelines in code.

Scikit-learn pipelines

Our old friend **Scikit-Learn** comes packaged with some nice pipelining functionality. The API is extremely easy to use, as you would expect from **Scikit-Learn**, but has some concepts we should understand before proceeding:

- **The pipeline object:** This is the object that will bring together all the steps we require, in particular, `sklearn` demands that instantiated pipeline objects are composed of sequences of transformers and estimators, with all intermediate objects having the `.fit()` and `.transform()` methods and the last step being an estimator with at least the `.fit()` method. We will explain these terms in the next two points. The reason for this condition is that the `pipeline` object will inherit the methods from the last item in the sequence provided, so we must make sure to have `.fit()` present in the last object.
- **Estimators:** The estimator class is the base object in `scikit-learn` and anything in the package that can be fit on data and then predict on data, therefore the `.fit()` and `.predict()` methods, is a subclass of the estimator class.
- **Transformers:** In **Scikit-Learn**, transformers are any estimators that have a `.transform()` or `.fit_transform()` method and, as you can guess, are mainly focused on transforming datasets from one form to another rather than performing predictions.

The use of the `pipeline` object really helps facilitate the simplification of your code, as rather than writing several different fitting, transforming, and predicting steps as their own function calls with datasets and then managing the flow of that data, you can simply compose them all in one object that manages this for you and uses the same simple API.

There are new transformers and features being added to Scikit-Learn all the time, which means that it become possible to build more and more useful pipelines. For example, at the time of writing, Scikit-Learn versions greater than 0.20 also contain the `ColumnTransformer` object, which allows you to build pipelines that perform different actions on specific columns.

This is exactly what we want to do with the logistic regression marketing model example we were discussing previously, where we want to standardize our numerical values and one-hot encode our categorical variables. Let's get started:

1. To create this pipeline, you need to import the `ColumnTransformer` and `Pipeline` objects:

```
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
```

2. To show you how to chain steps inside the transformers that make up the pipeline, we will add some imputation later. For this, we need to import the `SimpleImputer` object:

```
from sklearn.impute import SimpleImputer
```

3. Now, we must define the numerical transformer sub-pipeline, which contains the two steps for imputation and scaling. We must also define the names of the numerical columns this will apply to so that we can use them later:

```
numeric_features = ['age', 'balance']
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())])
```

4. Next, we must perform similar steps for the categorical variables, but here, we only have one transformation step to define for the `one-hot` encoder:

```
categorical_features = ['job', 'marital', 'education', 'housing', 'loan', 'categorical_transformer = OneHotEncoder(handle_unknown='ignore')]
```

5. We must bring all of these preprocessing steps together into a single object, called `preprocessor`, using the `ColumnTransformer` object. This will apply our `transformers` to the appropriate columns of our DataFrame:

```
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])
```

6. Finally, we want to add the ML model step at the end of the previous steps and finalize the pipeline. We will call this `clf_pipeline`:

```
clf_pipeline = Pipeline(steps=[('preprocessor', preprocessor),
                               ('classifier', LogisticRegression())])
```

7. This is our first ML training pipeline. The beauty of the `scikit-learn` API is that the `clf_pipeline` object can now be called as

if it were a standard algorithm from the rest of the library. So, this means we can write the following:

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

This will run the `fit` methods of all of the pipeline steps in turn.

The previous example was relatively basic, but there are a few ways you can make this sort of pipeline more sophisticated if your use case requires it. One of the simplest and most extensible is the ability in Scikit-Learn to create custom transformer objects that inherit from the base classes. You can do this for a class transformer by inheriting from the `BaseEstimator` and `TransformerMixin` classes and defining your own transformation logic. As a simple example, let's build a transformer that takes in the specified columns and adds a float. This is just a simple schematic to show you how it's done; I can't imagine that adding a single float to your columns will be that helpful in most cases!

```
from sklearn.base import BaseEstimator, Transfo

class AddToColumnsTransformer(BaseEstimator, Tra
    def __init__(self, addition = 0.0, columns=
        self.addition = addition
        self.columns = columns

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

    def transform(self, X, y=None):
        transform_columns = list(X.columns)
```

```
if self.columns:  
    transform_columns = self.columns  
X[transform_columns] = X[transform_colu  
return X
```

You could then add this transformer to your `pipeline`:

```
pipeline = Pipeline(  
    steps=[  
        ("add_float", AddToColumnsTransformer(0  
  
    ]  
)
```

This example of adding a number is actually not the best use case for using this class-based transformer definition, as this operation is stateless. Since there is no training or complex manipulation of the values being fed in that requires the class to retain and update its state, we have actually just wrapped a function. The second way of adding your own custom steps takes advantage of this and uses the `FunctionTransformer` class to wrap any function you provide:

```
from sklearn.pipeline import Pipeline  
from sklearn.preprocessing import FunctionTrans  
  
def add_number(X, columns=None, number=None):  
    if columns == None and number == None:  
        return X
```

```
X[columns] = X[columns] + number
pipeline = Pipeline(
    steps=[
        (
            "add_float",
            FunctionTransformer(
                add_number, kw_args={"columns": "number":}
            )
        )
    ]
)
```

By building on these examples, you can start to create complex pipelines that can perform any feature engineering task you want.

To conclude this section, we can clearly see that the ability to abstract the steps that are performing feature engineering and training your model into a single object is very powerful, as it means you can reuse this object in various places and build even more complex workflows with it without constantly recoding the details of the implementation. Abstraction is a good thing!

We will now turn to another way of writing pipelines, using Spark ML.

Spark ML pipelines

There is another toolset we have been using throughout this book that will be particularly important when we discuss scaling up our solutions: Apache Spark and its ML ecosystem. We will see that building a similar pipeline

with Spark ML requires a slightly different set of syntax, but the key concepts look very similar to the Scikit-Learn case.

There are a few important points to mention about PySpark pipelines. Firstly, in line with good programming practices in Scala, which Spark is written in, objects are treated as **immutable**, so transformations do not occur *in place*. Instead, new objects are created. This means that the output of any transformation will require new columns to be created in your original DataFrame (or indeed new columns in a new DataFrame).

Secondly, the Spark ML estimators (that is, the ML algorithms) all require the features to be assembled into one tuple-like object in a single column. This contrasts with Scikit-Learn, where you can keep all the features in their columns in your data object. This means that you need to become comfortable with the use of **assemblers**, which are utilities for pulling disparate feature columns together, especially when you are working with mixed categorical and numerical features that must be transformed in different ways before being invested by the algorithm.

Thirdly, Spark has many functions that use **lazy evaluation**, meaning that they are only executed when they're triggered by specific actions. This means that you can build up your entire ML pipeline and not have to transform any data. The reason for lazy evaluation is that the computational steps in Spark are stored in a **Directed Acyclic Graph (DAG)** so that the execution plan can be optimized before you perform the computational steps, making Spark very efficient.

Finally – and this is a small point – it is commonplace to write PySpark variables using *camel case* rather than the common *snake case*, which is often used for Python variables (for instance, **variableName** versus **variable_name**). This is done to keep the code in line with the PySpark

functions that inherit this convention from the underlying **Scala** code behind Spark.

The Spark ML pipelines API utilizes concepts of Transformer and Estimator in a similar way to how the Scikit-Learn pipeline API did, with some important differences. The first difference is that Transformers in Spark ML implement `.transform()` but not the `.fit_transform()` method. Secondly, the Transformer and Estimator objects in Spark ML are stateless, so once you have trained them they do not change and they only contain model metadata. They don't store anything about the original input data. One similarity is that pipelines are treated as Estimators in Spark ML as well.

We will now build a basic example to show how to build a training pipeline using the Spark ML API.

Let's take a look:

1. First, we must one-hot encode the categorical features for the previous example using the following syntax:

```
from pyspark.ml import Pipeline, PipelineModel
categoricalColumns = ["job", "marital", "education",
                      "housing", "loan", "default"]

for categoricalCol in categoricalColumns:
    stringIndexer = StringIndexer(inputCol=categoricalCol,
                                   outputCol=categoricalCol + "Index").setInputCol(categoricalCol)

    encoder = OneHotEncoder(
        inputCols=[stringIndexer.getOutputCol()],
        outputCols=[categoricalCol + "Index"])
```

```
)  
stages += [stringIndexer, encoder]
```

2. For the numerical columns, we must perform imputation:

```
numericalColumns = ["age", "balance"]  
numericalColumnsImputed = [x + "_imputed" for x in numericalColumns]  
imputer = Imputer(inputCols=numericalColumns)  
stages += [imputer]
```

3. Then, we must perform standardization. Here, we need to be a bit clever about how we apply **StandardScaler** as it only applies to one column at a time. Therefore, we need to create a scaler for each numerical feature after pulling our numerically imputed features into a single feature vector:

```
from pyspark.ml.feature import StandardScaler  
  
numericalAssembler = VectorAssembler(  
    inputCols=numericalColumnsImputed,  
    outputCol='numerical_cols_imputed')  
stages += [numericalAssembler]  
scaler = StandardScaler(inputCol='numerical_cols_imputed',  
                        outputCol="numerical_cols_imputed")  
stages += [scaler]
```

4. Then, we have to assemble the numerical and categorical transformed features into one feature column:

```
assemblerInputs = [c + "classVec" for c in c
assembler = VectorAssembler(inputCols=assemb
                                outputCol="featu
stages += [assembler]
```

5. Finally, we can define our model step, add this to the `pipeline`, and then train on and transform:

```
lr = LogisticRegression(labelCol="label", fe
                           maxIter=10)
stages += [lr]
(trainingData, testData) = data.randomSplit(
clfPipeline = Pipeline().setStages(stages).f
clfPipeline.transform(testData)
```

You can then persist the model pipeline as you would any `Spark` object, for example, by using:

```
clfPipeline.save(path)
```

Where `path` is the path to your target destination. You would then read this pipeline into memory by using:

```
from pyspark.ml import Pipeline
clfPipeline = Pipeline().load(path)
```

And that is how we can build a training pipeline in PySpark using **Spark ML**. This example shows you enough to get started with the API and build out your own, more sophisticated pipelines.

We will now conclude this chapter with a brief summary of everything we have covered.

Summary

In this chapter, we learned about the important topic of how to build up our solutions for training and staging the ML models that we want to run in production. We split the components of such a solution into pieces that tackled training the models, the persistence of the models, serving the models, and triggering retraining for the models. I termed this the “Model Factory.”

We got into the more technical details of some important concepts with a deep dive into what training an ML model really means, which we framed as learning about how ML models learn. Some time was then spent on the key concepts of feature engineering, or how you transform your data into something that a ML model can understand during this process. This was followed by sections on how to think about the different modes your training system can run in, which I termed “train-persist” and “train-run.”

We then discussed how you can perform drift detection on your models and the data they are consuming using a variety of techniques. This included some examples of performing drift detection using the Alibi Detect and Evidently packages and a discussion of how to calculate feature importances.

We then covered the concept of how the training process can be automated at various levels of abstraction, before explaining how to programmatically manage the staging of your models with MLflow Model Registry. The final section covered how to define training pipelines in the Scikit-Learn and Spark ML packages.

In the next chapter, we will find out how to package up some of these concepts in a Pythonic way so that they can be deployed and reused seamlessly in other projects.

Join our community on Discord

Join our community's Discord space for discussion with the author and other readers:

<https://packt.link/mle>



4

Packaging Up

In previous chapters, we introduced a lot of the tools and techniques you will need to use to successfully build working **machine learning (ML)** products. We also introduced a lot of example pieces of code that helped us to understand how to implement these tools and techniques. So far, this has all been about *what* we need to program, but this chapter will focus on *how* to program. In particular, we will introduce and work with a lot of the techniques, methodologies, and standards that are prevalent in the wider Python software development community and apply them to ML use cases. The conversation will be centered around the concept of developing *user-defined libraries and packages*, reusable pieces of code that you can use to deploy your ML solutions or develop new ones. It is important to note that everything we discuss here can be applied to all of your Python development activities across your ML project development life cycle. If you are working on some exploratory data analysis in a notebook or some modeling scripts for the research portion of your project, your work will still benefit immensely from the concepts we are about to introduce.

In this chapter, we will recap some of the basic points of programming in Python, before discussing the concept of coding standards and some pointers for writing high-quality Python code. We will also touch upon the difference between **object-oriented** and **functional** programming in Python, and where this has strengths and points of synergy with other tools

that you may want to use in your solution. We will discuss some good use cases for writing your own ML packages and go through the options for packaging up. Next will be a discussion on testing, logging, and error handling in your code, which are important concepts for building code that can be trusted not just to work but also to be diagnosable when it doesn't. This will be followed by a deep dive into the logical flow of our package. Finally, we will perform an exploration of how we ensure we do not reinvent the wheel and use functionality that already exists elsewhere.

In this chapter, we will cover the following topics:

- Writing good Python
- Choosing a style
- Packaging your code
- Building your package
- Testing, logging, and error handling
- Not reinventing the wheel

IMPORTANT NOTE

There isn't a clearly defined difference between a package and a library in Python. The general consensus seems to be that *library* often refers to any collection of code you want to reuse in other projects, whereas *package* refers to a collection of Python modules (covered in this chapter). We will often use the two interchangeably here with the understanding that when we say library, we are usually referring to a bunch of code that is cleanly put together and contains at least one package. This means that we won't



count single scripts with some code you reuse later as a library for our purposes here.

Who doesn't want to write more robust, clean, readable, testable, and performant code that can be used by our colleagues, the ML community, or even our customers? Let's get started!

Technical requirements

As with the other chapters, the dependencies required to run the examples in this chapter can be installed by navigating to the `Chapter 04` folder of the book repository and creating a new Conda environment:

```
conda env create -f mlewp-chapter04.yml
```

You should note that this chapter mainly focuses on Python fundamentals around packaging, so the requirements are a bit lighter than usual!

Writing good Python

As discussed throughout this book, Python is an extremely popular and very versatile programming language. Some of the most widely used software products in the world, and some of the most widely used ML engineering solutions in the world, use Python as a core language.

Given this scope and scale, it is clear that if we are to write similarly amazing pieces of ML-driven software, we should once again follow the best practices and standards already adopted by these solutions. In the

following sections, we will explore what packaging up means in practice, and start to really level up our ML code in terms of quality and consistency.

Recapping the basics

Before we get stuck into some more advanced concepts, let's make sure we are all on the same page and go over some of the basic terminology of the Python world. If you feel quite confident in the fundamentals of Python, then you can skip this section and carry on with the rest of the chapter.

However, going over these fundamentals if you are a bit newer to Python or have not revised them in a while will ensure that you apply the right thought processes to the right things and that you can feel confident when writing your code.

In Python, we have the following objects:

- **Variable:** An object that stores data of one of a variety of types. In Python, variables can be created through **assignment** without specifying the type, for example:

```
numerical_variable = 10
string_variable = 'string goes here'
```

- **Function:** A unit of code that is self-contained and performs logical steps on variables (or another object). Defined by the **def** keyword in Python and can return any Python object. Functions are *first-class citizens* in Python, which means you can reference them using their object name (and re-reference them), and that functions can pass and return functions. So, for example, if we create a function that

calculates some simple statistics from a pandas DataFrame, we can do the following. First, define it:

```
def calculate_statistics(df):
    return df.describe()
```

Then run it using the original name and a DataFrame called `X_train`:

```
calculate_statistics(X_train)
```

Then you can re-assign the function using a new name and similarly call it:

```
new_statistics_calculator = calculate_statistics
new_statistics_calculator(X_train)
```

You can then pass the function around even more. For example, if you pass the function into a new function that takes the result and returns a JSON object, then you can call that!

```
def make_func_result_json(func ,df):
    return func(df).to_json
make_func_result_json(calculate_statistics,
```

This can help build up some simple pieces of code into something relatively complex quite quickly.

- **Module:** This is a file containing definitions and statements of functions, variables, and other objects where the contents can be imported into other Python code. For example, if we put the functions defined in the previous example into a file called `module.py`, we can then type the following in another Python program (or the Python interpreter) in order to use the functionality contained within it:

```
import module
module.calculate_statistics(df)
module.make_func_result_json(module.calcula
```

- **Class:** We will discuss classes in detail in the *Object-oriented programming* section, but for now, just know that these are the basic units of object-oriented programming, and act as a nice way of containing logically related functionality.
- **Package:** This is a collection of modules that are coupled together via their directory structure and is built such that modules in the package are accessed through the `dot` syntax. For example, if we have a package called `feature` that contains modules to help us to do feature engineering, it could be organized as follows:

```
feature/
| -- numerical/
|   | -- analyze.py
|   | -- aggregate.py
|   | -- transform.py
| -- categorical/
|   | -- analyze.py
```

```
| -- aggregate.py  
| -- transform.py
```

Then, if we wanted to use the functionality contained within the `numerical` or `categorical` sub-modules, we would use the `dot` syntax like so:

```
import feature.categorical.analyze  
import feature.numerical.transform
```

Now let's move on to discuss some general Python tips and tricks.

Tips and tricks

Let's now discuss some tips and tricks for using Python that can often be overlooked, even by those quite familiar with the language. The following concepts can help you write more compact and performant code, so it's good to have them to hand. Note that this list is definitely not exhaustive:

- **Generators:** These are convenience functions for helping us create a syntax that iterates in some sense. They save us from writing a lot of boilerplate code, are memory efficient, and have very useful properties, such as the ability to pause execution and save the internal state automatically. Then you can resume iterating with it later in your program. Generators are created in Python whenever we define a function that uses the `yield` statement. For example, here we can define a generator that will filter a given list of values based on a predicate called `condition`:

```
def filter_data(data, condition):
    for x in data:
        if condition(x):
            yield x
```

In action, we could apply this to a simple list of the integers from zero to ninety-nine called `data_vals` and filter out values below a certain threshold:

```
for x in filter_data(data_vals, lambda x: x
                      >= 50)
    print(x)
```

This will return the integers from fifty to ninety-nine.

The other way to define a generator expression is by using an iterative statement in round brackets. For example, here we can define a generator that iterates over the squares from zero to nine:

```
gen1 = (x**2 for x in range(10))
for i in gen1:
    print(i)
```

Note that you can only execute your generators once; after that, they are *empty*. This is because they only store what they need in memory for each step of the iteration, so once it is complete, nothing is stored!

Generators are really powerful ways of creating data manipulation steps that are memory efficient and can be used to define custom pipelines in frameworks such as Apache Beam. We will not cover this

here, but it is definitely worth checking out. As an example, take a look at the article at <https://medium.com/analytics-vidhya/building-a-data-pipeline-with-python-generators-a80a4d19019e>.

- **List comprehension:** This is a syntax that allows us to take any iterable we have to hand (a `dict`, a `list`, a `tuple`, and a `str` are all examples) and build a list from it in an extremely compact way. This can save you from writing long, clunky loops and can help create some more elegant code. List comprehensions create the entire list in memory, so they are not as efficient as generators. So use them wisely, and only create small lists if you can. You perform list comprehension by writing your iteration logic in square brackets, as opposed to the round brackets of generators. As an example, we can create the data used in the first generator example:

```
data_vals = [x for x in range(100)]
```

- **Containers and collections:** Python has a useful set of built-in types that are known as **containers**, these being `dict`, `set`, `list`, and `tuple`. Beginners in Python learn how to use these from their first time playing with the language, but what we can often forget is their augmented counterparts: **collections**. These allow for additional behavior on top of the standard containers, which can be useful. The table shown in *Figure 4.1* summarizes some useful containers mentioned in the Python 3 documentation on [python.org](https://docs.python.org/3/library/collections.html) at <https://docs.python.org/3/library/collections.html>.

These are useful to have to hand when you are working through some data manipulations and can often save you a couple of lines of code:

Container	Description
deque	This is a double-ended queue and allows you to add and remove elements to either end of the object in a scalable way. It's useful if you want to add to the beginning or end of large data lists or if you want to search for the last occurrences of X in your data.
Counter	Counters take in iterables such as dicts or lists and return the count of each of the elements. They're really useful to get quick summaries of the content of these objects.
OrderedDict	The standard dict object does not maintain order, so OrderedDict introduces this functionality. This can be really useful if you need to loop back over a dictionary you have created in the same order as it was created for new processing.

Table 4.1: Some useful types in the collections module in Python 3.

- ***args and **kwargs:** When we want to call a function in Python, we often supply it with arguments. We have seen plenty of examples of this in this book already. But what happens if you define a function for which you would like to apply to a varying number of arguments? This is where the ***args** and ****kwargs** patterns come in. For example,

imagine we want to initialize a class called `Address` that uses information gathered from an online web form to create a single string giving an address.

We may not know how many elements are going to be in each text box used by the user for the address ahead of time. We could then use the `*args` pattern (you don't have to call it `args`, so here we've called it `address`). Here's the class:

```
class Address(object):
    def __init__(self, *address):
        if not address:
            self.address = None
            print('No address given')
        else:
            self.address = ' '.join(str(x) for
```

Then your code will work absolutely fine in both of these cases, even though there are a variable number of arguments to the constructor:

```
address1 = Address('62', 'Lochview', 'Crescent')
address2 = Address('The Palm', '1283', 'Royston Road')
```

Then `address1.address` will be given by '`62 Lochview Crescent`' and `address2.address` will be given by '`The Palm 1283 Royston Road`'.

`**kwargs` extends this idea to allow a variable number of keyword arguments. This is particularly useful if you have functions where you

may want to define a variable number of parameters, but you need names attached to those parameters. For example, we may want to define a class for containing ML model hyperparameter values, the number and names of which will vary by algorithm. We can therefore do something like the following:

```
class ModelHyperparameters(object):
    def __init__(self, **hyperparams):
        if not hyperparams:
            self.hyperparams = None
        else:
            self.hyperparams = hyperparams
```

Then the code will allow us to define instances such as the following:

```
hyp1 = ModelHyperparameters(eps=3, distance='euclidean')
hyp2 = ModelHyperparameters(n_clusters=4, max_iter=100)
```

And then `hyp1.hyperparams` will be given by `{'eps': 3, 'distance': 'euclidean'}` and `hyp2.hyperparams` by `{'n_clusters': 4, 'max_iter': 100}`.

There are many more concepts that are important to understand for a detailed understanding of how Python works. For now, these pointers will be enough for us to build upon throughout the chapter.

Now we will consider how to define and organize these elements in a way that makes your code readable and consistent.

Adhering to standards

When you say something like *adhering to standards*, in most contexts, you would be forgiven for half-expecting a sigh and a gigantic eye roll from whoever you were talking to. Standards sound boring and tedious, but they are in fact an extremely important part of making sure that your work is consistent and high quality.

In Python, the *de facto* standard for coding style is **Python Enhancement Proposal 8 (PEP-8)**, written by Guido Van Rossum (the creator of Python), Barry Warsaw, and Nick Coghlan (<https://www.python.org/dev/peps/pep-0008/>). It is essentially a collection of guidelines, tips, tricks, and suggestions for making code that is consistent and readable. Some of the benefits of adhering to the PEP-8 style guide in your Python projects are as follows:

- **Greater consistency:** This will help you write code that is less likely to break once you have deployed it, as it is much easier to follow the flow of your programs and identify errors and bugs. Consistency also helps simplify the design of extensions and interfaces to your code.
- **Improved readability:** This begets efficiency, as colleagues and even users of your solutions can understand what is being done and how to use it more effectively.

So, what is in the PEP-8 style guide? And how should you think about applying it to your ML project? For the full details, I recommend you read the PEP-8 documentation given earlier. But in the next few paragraphs, we will go into some of the details that will give you the greatest improvement to your code for the least effort.

First, let's cover **naming conventions**. When you write a piece of code, you will have to create several variables, files, and other objects, such as classes, and these all have to have a name. Making sure that these names are readable and consistent is the first part of making your code of a very high standard.

Some of the key pointers from PEP-8 are as follows:

- **Variables and function names:** It is recommended that these consist of all lowercase words, separated by underscores. They should also help us understand what they are for. As an example, if you are building a regression model and you want to put some of your feature engineering steps inside a function to simplify reuse and readability elsewhere in the code, you may call it something like

`Makemydata()`:

```
def Makemydata():
    # steps go here ...
    return result
```

Calling your function `Makemydata()` is not a great idea, whereas naming it something like `transform_features` is better:

```
def transform_features():
    # steps go here ...
    return result
```

This function name is compliant with PEP-8.

- **Modules and packages:** The recommendation is that these have all short lowercase names. Some great examples are ones you are familiar with, such as `pandas`, `numpy`, and `scipy`. `Scikit-learn` may seem like it breaks this rule, but it actually doesn't as the package name is `sklearn`. The style guide mentions that modules can have underscores to improve readability, but packages should not. If we had a module in a package called `transform_helpers`, then this is acceptable, but an entire package called `marketing_outlier_detection` would be terrible!
- **Classes:** Classes should have names such as `OutlierDetector`, `Transformer`, or `PipelineGenerator`, which clearly specify what they do and also follow the upper CamelCase or PascalCase (both mean the same thing) style.

These are some of the most commonly used naming conventions you should be aware of. The PEP-8 document also covers a lot of good points on whitespace and the formatting of lines that we will not go into here. We will finish this section with a discussion on some of the author's favorite suggestions from the *programming recommendations* of PEP-8. These are often overlooked and, if forgotten, can make for some code that is both horrible to read and likely to break, so take heed!

A good point to remember in all of this talk about style is that at the top of the PEP-8 document, which states that *Foolish Consistency is the Hobgoblin of Little Minds* and that there are good reasons to ignore these style suggestions in certain circumstances. Again, read the PEP-8 document for the full works, but if you follow these points, then in general, you will write clean and readable code.

Next, we will cover how some of these rules do not really apply when we are using the Python API for Apache Spark.

Writing good PySpark

In this section, we draw attention to one particular flavor of Python that is very important in the world of data science and ML. PySpark code has already been used in examples throughout this book since it is the go-to tool for distributing your data workloads, including your ML models. In *Chapter 6, Scaling Up*, we will learn more about PySpark, but here we will just briefly mention some points on coding style.

As mentioned in the section on *Spark ML pipelines* in *Chapter 3, From Model to Model Factory*, since Spark is written in Scala, the syntax of PySpark (which is just the Python API for Spark) has inherited a lot of the syntactical style from that underlying language. This means in practice that many of the methods you use will be written in CamelCase, meaning that it also makes sense to define your variables using CamelCase rather than the standard Python PEP-8 naming convention of words separated by underscores. This is behavior that we should encourage as it helps people reading our code to clearly see which sections are PySpark code and which are (more) vanilla Python. To emphasize this, when we used the `StringIndexer` object from the `pyspark.ml` package before, we used `StringIndexer` instead of the more idiomatic Python, `string_indexer`:

```
from pyspark.ml.feature import StringIndexer
stringIndexer = StringIndexer(inputCol=categori
                               outputCol=categor
```

Another important point about PySpark code is that because Spark is written in a functional paradigm, it also makes sense that your code also follows this style. We will understand a bit more about what this means in the next section.

Choosing a style

This section will provide a summary of two coding styles or paradigms, which make use of different organizational principles and capabilities of Python. Whether you write your code in an object-oriented or functional style could just be an aesthetic choice.

This choice, however, can also provide other benefits, such as code that is more aligned with the logical elements of your problem, code that is easier to understand, or even more performant code.

In the following sections, we will outline the main principles of each paradigm and allow you to choose for yourself based on your use case.

Object-oriented programming

Object-oriented programming (OOP) is a style where the code is organized around, you guessed it, abstract objects with relevant attributes and data instead of around the logical flow of your solution. The subject of OOP is worth a book (or several books!) in itself, so we will focus on the key points that are relevant to our ML engineering journey.

First, in OOP, you have to define your **objects**. This is done in Python through the core OOP principle of classes, which are definitions of structures in your program that keep together related data and logical

elements. A class is a template for defining the objects in OOP. As an example, consider a very simple class that groups together some methods for calculating numerical outliers on a dataset. For example, if we consider the pipelines that we looked into in *Chapter 3, From Model to Model Factory*, we may want to have something that makes this even easier to apply in a production setting. We may therefore want to wrap up some of the functionality provided by tools such as Scikit-Learn into a class of its own that could have bespoke steps specific to our problem. In the simplest case, if we wanted a class to wrap the standardization of our data and then apply a generic outlier detection model, it could look something like this:

```
class OutlierDetector(object):
    def __init__(self, model=None):
        if model is not None:
            self.model = model
        self.pipeline = make_pipeline(Standards
    def detect(self, data):
        return self.pipeline.fit(data).predict()
```

All this example does is allow a user to skip writing out some of the steps that they may have to otherwise write to get the job done. The code doesn't disappear; it just gets placed inside a handy object with a clear logical definition. In this case, the pipeline shown is extremely simple, but we can imagine extending this to something very complex and containing logic that's specific to our use case.

Therefore, if we have already defined an outlier detection model (or retrieved it from a model store, such as MLflow, as discussed in *Chapter 3, From Model to Model Factory*), we can then feed this into this class and run

quite complex pipelines just with a single line of code, no matter the complexity contained within the class:

```
model = IsolationForest(contamination=outliers_
                         random_state=42)
detector = OutlierDetector(model=model)
result = detector.detect(data)
```

As you can see from the example, this pattern of implementation seems familiar, and it should! **Scikit-learn** has a lot of OOP in it, and you use this paradigm every time you create a model. The act of creating a model is a case of you instantiating a class object, and the process of you calling `fit` or `predict` on your data are examples of calling class methods. So, the reason the preceding code may not seem alien is that it shouldn't! We've already been using OOP when working with ML with Scikit-Learn.

Despite what we have just said, using objects and understanding how to build them are obviously two different challenges. So, let's go through the core concepts of building your own classes. This will set us up later for building more classes of relevance for our own ML solutions.

First, you see from the preceding code snippet that a class is defined with the `class` keyword and that the PEP-8 convention is to use upper CamelCase for the class name. It is also good practice to make your class names clear definitions of *things that do stuff*. For example, `OutlierDetector`, `ModelWrapper`, and `DataTransformer` are good class names, but `Outliers` and `Calculation` are not. You will also notice that we have something in brackets after the name of the class. This tells the class which objects to inherit functionality from. In the

preceding example, we can see that this class inherits from something called `object`. This is actually the built-in base class in Python from which *all other objects inherit*. Therefore, since the class we defined does not inherit from anything more complex than `object`, you can think of this as essentially saying *the class we are about to build will have all of the functionality it needs defined within it; we do not need to use more complex functionality already defined in other objects for this class*. The syntax showing the inheritance from `object` is actually superfluous as you can just omit the brackets and write `OutlierDetector`, but it can be good practice to make the inheritance explicit.

Next, you can see that the functionality we want to group together is defined inside the class. Functions that live inside a class are called methods. You can see that `OutlierDetector` has only one method, called `detect`, but you are not limited in how many methods your class can have.

Methods contain your class's abilities to interact with data and other objects, so their definition is where most of the work of building up your class goes.

You might think we have missed a method, the one called `__init__()`. This is in fact not a method (or you can think of it as a very special method) and is called the *constructor*. The constructor does what it says—it constructs! Its job is to perform all of the relevant setup tasks (some of which occur in the background, such as memory allocation) for your class when it gets initialized as an object. When the example defines `detector`, the constructor is called. As you can see, you can pass variables and then these variables can be used within the class. Classes in Python can be created without defining an explicit constructor, but one will

be created in the background. The final point we will make on constructors is that they are not allowed to return anything other than `None`, so it's common to leave the `return` statement unwritten.

You will also have seen in the example that there are variables inside the class and there is a somewhat mysterious `self` keyword. This allows methods and operations inside the class to refer to the particular instance of the class. So, if you define two or a hundred instances of the `OutlierDetector` object, it is possible for them all to have different values for their internal attributes but still have the same functionality.

We will create some more involved OOP styles for your ML solution later, but for now, let's discuss the other programming paradigm that we may want to use – functional programming.

Functional programming

Functional programming is based on the concept of, you guessed it, functions. At its core, this programming paradigm is about trying to write pieces of code that only take in data and output data, doing so without creating any internal state that can be changed. One of the goals of functional programming is to write code that has no unintended side effects due to mismanagement of state. It also has the benefit of making sure that the data flow in your programs can be understood completely by looking at the `return` statements of the functions you have written.

It uses the idea of the data in your program not being allowed to change in place. This concept is known as **immutability**. If your data (or any object) is immutable, it means that there is no internal state to modify, and if you want to do something with the data, you actually have to create new data.

For example, in the section on *Object-oriented programming*, we again revisited the concept of standardizing data. In a functional program, standardized data cannot overwrite unstandardized data; you would need to store this new data somewhere, for example, in a new column in the same data structure.

Some programming languages are designed with functional principles at their core, such as F# and Haskell, but Python is a general-purpose language that can accommodate both paradigms quite nicely.

You will likely have seen some other functional programming concepts in other Python code. For example, if you have ever used a lambda function, then this can be a powerful aspect of a functionally programmed piece of code as it is how you define *anonymous functions* (those without a specified name). So, you may have seen code that looks something like this:

```
df['data_squared'] = df['data'].apply(lambda x:
```

In the preceding code block, `df` is a pandas DataFrame and `data` is just a column of numbers. This is one of the tools that help make functional programming in Python easier. Other such tools are the built-in functions `map()`, `reduce()`, and `filter()`.

As an example, imagine that we have some address data similar to that in the *Recapping the basics* section, where we discussed the concepts of `args` and `**kwargs`:

```
data = [  
    ['The', 'Business', 'Centre', '15', 'Steven
```

```
[ '6', 'Mossvale', 'Road'],
[ 'Studio', '7', 'Tottenham', 'Court', 'Road'
]
```

Now, we might want to write some code that returns a list of lists with the same shape as this data, but every entry now contains the number of characters in each string. This could be a stage in a data preparation step in one of our ML pipelines. If we wanted to write some code to do this functionally, we could define a function that takes a list and returns a new list with the string lengths for the entries like this:

```
def len_strings_in_list(data_list):
    return list(map(lambda x: len(x), data_list
```

This embodies functional programming because the data is immutable (there is no change of internal state) and the function is pure (it only uses data within the scope of the function). We can then use another concept from functional programming called higher-order functions, where you supply functions as the arguments of other functions. For example, we may want to define a function that can apply any list-based function but to a list of lists:

```
def list_of_list_func_results(list_func, list_o
    return list(map(lambda x: list_func(x), lis
```

Note that this is completely generic; as long as the `list_func()` can be applied to a list, this will work on a list of lists. We can therefore get the

original result we wanted by calling the following:

```
list_of_list_func_results(len_strings_in_list,
```

This returns the desired result:

```
[[3, 8, 6, 2, 9, 4], [1, 8, 4], [6, 1, 9, 5, 4]]
```

Spark, a tool that's already been used multiple times in this book, is written in the Scala language, which is also general-purpose and can accommodate both object-oriented and functional programming. Spark is predominantly written in a functional style; its aim of distributing computation is more easily accommodated if principles such as immutability are respected. This means that when we have been typing PySpark code through this book, we have subtly been picking up some functional programming practices (did you notice?).

In fact, in *Chapter 3, From Model to Model Factory*, the example PySpark pipeline we built had code like this:

```
data = data.withColumn('label', f.when((f.col("1)).otherwise(0)))
```

This is functional since the `data` object we create is actually a new DataFrame with the new column added—we can't just add a column in place. There was also code that formed part of our pipelines from the Spark ML library:

```
scaler = StandardScaler(inputCol='numerical_col  
outputCol="numerical_co
```

This is defining how to take a series of columns in a DataFrame and perform a scaling transformation on them. Note how you define input columns and output columns, and *these cannot be the same*. That's immutability in action—you have to create new data rather than transform it in place.

Hopefully, this gives you a taste of functional programming in Python. This is not the main paradigm we will use in this book, but it will be used for some pieces of code, and, in particular, remember that when we use PySpark, we are often implicitly using functional programming.

We will now discuss ways of packaging the code that you have written.

Packaging your code

In some ways, it is interesting that Python has taken the world by storm. It is dynamically typed and non-compiled, so it can be quite different to work with compared to Java or C++. This particularly comes to the fore when we think about packaging our Python solutions. For a compiled language, the main target is to produce a compiled artifact that can run on the chosen environment – a Java `jar`, for example. Python requires that the environment you run in has an appropriate Python interpreter and the ability to install the libraries and packages you need. There is also no single compiled artifact created, so you often need to deploy your whole code base as is.

Despite this, Python has indeed taken the world by storm, especially for ML. As we are ML engineers thinking about taking models to production, we would be remiss to not understand how to package and share Python code in a way that helps others to avoid repetition, trust in the solution, and be able to easily integrate it with other projects.

In the following sections, we are first going to discuss what we mean by a user-defined library and some of the advantages of packaging your code this way. We are then going to define the main ways you can do this so that you can run your ML code in production.

Why package?

Before we discuss in detail exactly what a package or library is in Python, we can articulate the advantages by using a working definition of *a collection of Python code that can be run without detailed knowledge of its implementation*.

You will have already picked up from this definition the nature of the first reason to do this: **abstraction**.

Bringing together your code into a library or package that can be reused by other developers and data scientists in your team, organization, or the wider community allows these user groups to solve problems more quickly. Since the details of the work are abstracted away, anyone using your code can focus on implementing the capabilities of your solution, rather than trying to understand and dissect every line. This will lead to reduced development and deployment time in projects, as well as encourage the usage of your code in the first place!

The second advantage is that, by consolidating the functionality you need into a library or package, you bring all of the implementation details to one place and therefore *improvements scale*. What we mean by this is if 40 projects are using your library and someone discovers a minor bug, you only need to patch it *once* and then redeploy or update the package in those 40 implementations.

This is way more scalable than explaining the issue to the relevant teams and getting 40 different fixes at the implementation end. This consolidation also means that once you have thoroughly tested all the components, you can more confidently assume that this solution will be running smoothly in those 40 different projects, without knowing anything about the details under the hood.

Figure 4.1 helps to show how packages helpfully allow a *write once, use many* philosophy for your code, which is incredibly important if you want to engineer ML solutions that can solve multiple problems in a scalable fashion:

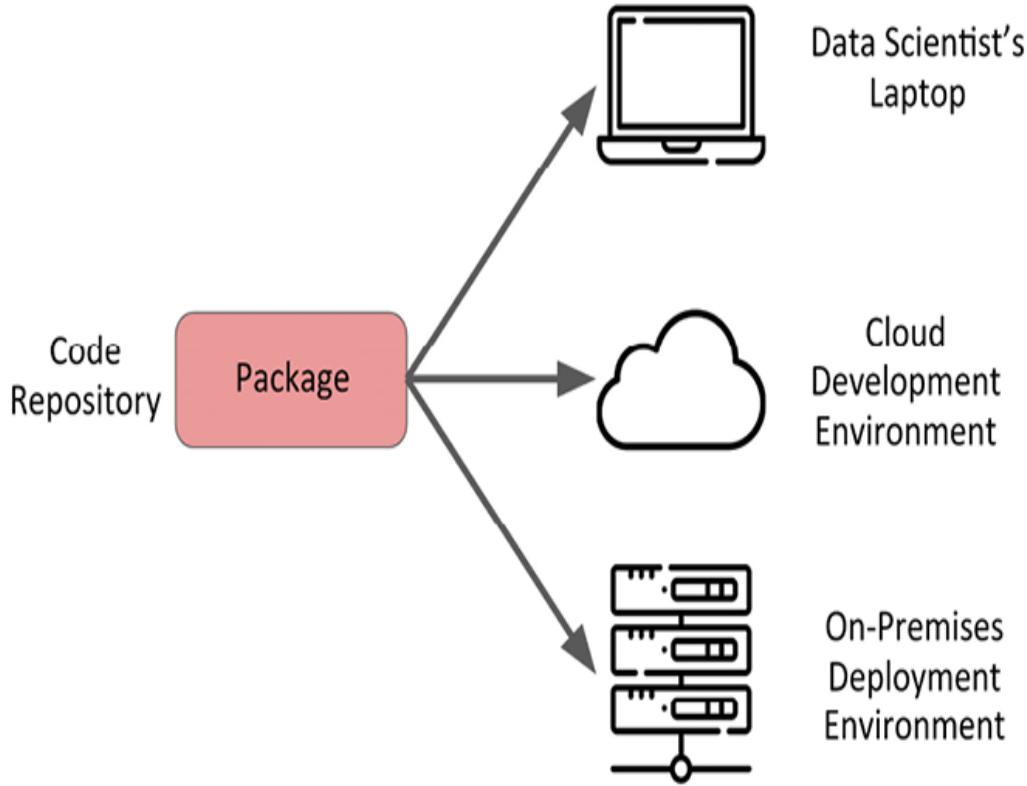


Figure 4.1: Developing packages for your ML solutions allows you to write the code once but use it many times in different environments.

The next section will build on these main ideas about packaging to discuss specified use cases in which packaging our code can be beneficial.

Selecting use cases for packaging

First things first, not all of your solutions should be libraries. If you have an extremely simple use case, you may only need one simple script to run on a schedule for the core of your ML solution. You can still write a well-engineered system and performant code in this case, but it's not a library. Similarly, if your problem is best solved by a web app, then although there will be lots of components, it will not naturally be a library.

Some good reasons you may want to write up your solution as a library or package are as follows:

- The problem your code solves is a common one that may come up in multiple projects or environments.
- You want to abstract away implementation details so that execution and development are decoupled, making it easier for others to use your code.
- To minimize the number of places and the number of times you need to change code to implement bug fixes.
- To make testing simpler.
- To simplify your **continuous integration/continuous deployment (CI/CD)** pipeline.

We will now dive into how we might go about designing our packages.

Designing your package

The layout of your code base is far more than just a stylistic consideration. It is something that will determine how your code is used in every instance of the project – no pressure!

This means that it is important to put some thought into how you want to lay out your code and how this influences usage patterns. You need to ensure that all of the main components you need have a presence in the code base and are easy to find.

Let's work this through with an example based on the outlier detection case we worked through in the previous sections.

First, we need to decide what kind of solution we want to create. Are we building something that will run a web application or a standalone

executable with lots of functionality, or are we building a library for others to use in their ML projects? In fact, we can choose to do more than one thing! For this case, let's build a package that can be imported for use in other projects but can also run in a standalone execution mode.

To set the context for the development of our package, imagine we have been asked to start building a solution that can run a set of selected unsupervised outlier detection models. The data scientists have found that, for the problem at hand, `Isolation Forest` models are the most performant, but they must be retrained on every run and the users of the package should be able to edit the configuration of the models through a config file. Only `sklearn` models have been studied so far, but the business and users of the package would like this functionality to be extensible to other modeling tools if needed. The technical requirements for this project mean we cannot use MLflow.

Don't worry; in later chapters when we build more examples, we will relax this constraint to show how it all fits together:

1. The package we are going to build is all about outliers, so let's call it `outliers` (I know, inventive, right?). Just to make it clear how everything hangs together, we will start to build the `outliers` package in a folder called `outlier_package`:

```
outlier_package/  
    └── outliers/
```

2. Our package design will be based on the functionality we want the solution to have; in this case, we want something that detects outliers, so let's create a sub-package called `detectors`:

```
outlier_package/
└── outliers/
    └── detectors
```

3. Within this, we will put some code that wraps (more on this later) around some basic models from external libraries. We will also want some code that gets data for us to analyze, so we will add a sub-package for that too:

```
outlier_package/
└── outliers/
    ├── detectors
    └── data
```

4. We can already see our package taking shape. Finally, we will want to have somewhere to store configuration information and somewhere to store helper functions that may be used across the package, so let's add a directory and sub-package for those too:

```
outlier_package/
└── outliers/
    ├── detectors
    ├── data
    ├── configs
    └── utils
```

Now, this layout is not sacrosanct or dictated in any way. We can create the layout however we want and do whatever we think makes sense.

It is important when doing this, though, to always remember the principles of **Don't Repeat Yourself (DRY)**, **Keep It Simple, Stupid (KISS)**, and the Python mantra of *there should preferably be only one way to do something*. If you stick to these principles, you will be fine.

For more information on these principles, see

<https://code.tutsplus.com/tutorials/3-key-software-principles-you-must-understand--net-25161> and <https://www.python.org/dev/peps/pep-0020/>.

So, what actually goes in each of these sub-packages? Well, the underlying code of course!

5. In this case, we will want something to provide an interface between our detector implementations and the syntax for creating a pipeline and calling them, so we will build a simple class and keep it in `pipelines.py`. The `pipelines.py` file contains the following code:

```
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

class OutlierDetector(object):
    def __init__(self, model=None):
        if model is not None:
            self.model = model
        self.pipeline = make_pipeline(StandardScaler(), model)
    def detect(self, data):
        return self.pipeline.fit(data).predict(data)
```

6. We then also need something to define the models we want to interface with. In this case, we will create code that uses information stored in a configuration file to decide which of a select few models to instantiate. We put all this functionality in a class called `DetectionModels`. For brevity, we omit the details of each of the functions in the class in this first instance:

```
import json
from sklearn.ensemble import IsolationForest

class DetectionModels(object):
    def __init__(self, model_config_path=None):
        ...
    def create_model(self, model_name=None):
        ...
    def get_models(self):
        ...
```

7. The initialization method is expanded here. Notice that we wrote this code so that we could define a series of models in the `config` file:

```
class DetectionModels(object):
    def __init__(self, model_config_path=None):
        if model_config_path is not None:
            with open(model_config_path) as f:
                self.model_def = json.load(f)
```

8. Then the `create_model` method is able to instantiate the model based on parameter and model name information. We have also built this so that we can actually pull in configuration information for models from different libraries if we wanted to; we would just need to add the appropriate implementation logic in this `create_model` function, checking that `sklearn` or another model was defined and running the appropriate syntax in each case. We would also have to make sure the pipeline generated in `OutlierDetector` was appropriate in each case as well:

```
def create_model(self, model_name=None,
                 if model_name is None and params is
                     return None
                 if model_name == 'IsolationForest' a
                     return IsolationForest(**params)
```

9. Finally, we bring the preceding methods together through the `get_models` method, which returns a list of all models defined in the appropriate config file, instantiated as a `sklearn` object via the `create_model` method:

```
def get_models(self):
    models = []
    for model_definition in self.model_c
        defined_model = self.create_mode
            model_name=model_definition[
                params=model_definition[ 'par
            )
```

```
        models.append(defined_model)
    return models
```

You may be thinking *why not just read in the appropriate model and apply it, no matter what it is?* That could be a viable solution, but what we have done here means that only model types and algorithms that have been approved by the team working on the project can make it through to production, as well as permitting the use of heterogeneous model implementations.

10. To see how this could all work in practice, let's define a script called `__main__.py` at the uppermost level of the package, which can act as the main entry point for the execution of modeling runs:

```
from utils.data import create_data
from detectors.detection_models import DetectionModels
import detectors.pipelines
from definitions import MODEL_CONFIG_PATH

if __name__ == "__main__":
    data = create_data()
    models = DetectionModels(MODEL_CONFIG_PATH)
    for model in models:
        detector = detectors.pipelines.OutlierDetector(model)
        result = detector.detect(data)
        print(result)
```

11. The `model_config.json` file referred to here is given by the following code:

```
[  
  {  
    "model": "IsolationForest",  
    "params": {  
      "contamination": 0.15,  
      "random_state": 42  
    }  
  }  
]
```

12. The `definitions.py` file is a file that holds relevant paths and other variables that we want to make globally accessible in the package without polluting the namespace:

```
import os  
  
ROOT_DIR = os.path.dirname(__file__)  
MODEL_CONFIG_PATH = os.path.join(ROOT_DIR, "json")
```

We can see that we don't really do anything with the results; we just print them to show that output is produced. But in reality, you will either push these results elsewhere or calculate statistics on them.

This script can be run by typing this in your terminal:

```
python __main__.py
```

Alternatively, you could type the following:

```
python -m outliers
```

And that is how you can package functionality into classes, modules, and packages. The example given was relatively constrained, but it does give us an awareness of how the different pieces can be brought together and executed.

IMPORTANT NOTE



The example given here has been built up to show you how to hang your code together by using some of the techniques discussed in this chapter. It is not necessarily the only way to bring all of these bits together, but it does act as a good illustration of how to create your own package. So, just remember that if you see a way to improve this implementation or adapt it to your own purposes, then brilliant!

In the next section, we will explore how to build distributions of this code and how to allow ourselves and users to install the `outliers` package as a normal Python package that we can use in other projects.

Building your package

In our example, we can package up our solution using the `setuptools` library. In order to do this, you must create a file called `setup.py` that contains the important metadata for your solution, including the location of the relevant packages it requires. An example of `setup.py` is shown in

the following code block. This shows how to do this for a simple package that wraps some of the outlier detection functionality we have been mentioning in this chapter:

```
from setuptools import setup

setup(name='outliers',
      version='0.1',
      description='A simple package to wrap some
                  functionality',
      author='Andrew McMahon',
      license='MIT',
      packages=['outliers'],
      zip_safe=False)
```

We can see that `setuptools` allows you to supply metadata such as the name of the package, the version number, and the software license. Once you have this file in the root directory of your project, you can then do a few things:

1. First, you can install the package locally as an executable. This will mean you can import your library like any other Python library in the code you want to run:

```
pip install .
```

2. You can create a source distribution of the package so that all of the code is bundled together efficiently. For example, if you run the

following command at the root of your project, a `gzipped tarball` is created in a folder called `dist`:

```
python setup.py sdist
```

3. You can create a built distribution of the package, which is an object that can be unpacked and used immediately by the user without them having to run the `setup.py` script as in a source distribution. The most appropriate built distribution is what is known as a Python `wheel`. Running the following command in the root directory of your project creates the `wheel` and puts it in the `dist` folder:

```
python setup.py bdist_wheel
```

4. If you are going to distribute your code using pip, then it makes sense to package both a `source` distribution and a `wheel` and let the user decide what to do. So, you can build both and then use a package called `twine` to upload both distributions to PyPI. If you want to do this, then you need to register for a PyPI account at <https://pypi.org/account/register/>. Just run the previous two commands together in the root directory of your project and use the `twine upload` command:

```
python setup.py sdist bdist_wheel  
twine upload dist/*
```

For a lot more information on packaging, you can read through the information and tutorials at <https://www.pypa.io/en/latest/>, provided by the **Python Packaging Authority (PyPA)**.

The next section touches briefly on how we can automate a few of the steps around building and testing our packages using Makefiles.

Managing your environment with Makefiles

If we are on a UNIX system and we have the `make` utility installed, then we can further automate a lot of the steps we want to run for our solution in different scenarios using Makefiles. For example, in the following code block, we have a Makefile that allows us to run our module's main entry point, run our test suite, or clean up any artifacts using the `run`, `test`, and `clean` targets:

```
MODULE := outliers
run:
    @python -m $(MODULE)
test:
    @pytest
.PHONY: clean test
clean:
    rm -rf .pytest_cache .coverage .pytest_cach
```

This is a very simple Makefile, but we can make it as complex as needed by layering more and more commands. If we want to `run` a specific target set

of commands, we simply call `make`, then the target name:

```
make test  
make run
```

This is a powerful way to abstract out a lot of terminal commands you would otherwise have to manually enter in each case. It also acts as documentation for other users of the solution!

The example we have just gone through is quite simple; let's now make things more sophisticated. We can actually use Makefiles to manage our environments and help streamline our development process so that it does not require lots of cognitive effort just to keep track of the state of our environment.

The following examples leverage a lot of great work by Kjell Wooding, or *hackalog* on GitHub, specifically his repository

https://github.com/hackalog/make_better_defaults.

This repository formed the basis of his talk at the 2021 PyData Global conference titled “Makefiles: One Great Trick for Making Your Conda Environments More Manageable.”

First, the inclusion of a `Makefile.help` file allows for customizable help prompts when using the `make` command. If we run `make` in the terminal, assuming we are still in the main project directory, you will see the output in *Figure 4.2*.

To get started:

```
>>> make create_environment  
>>> conda activate mlewp-ed2-ch4-outliers  
>>> make update_environment
```

Figure 4.2: The help presented from the Makefile example.

This help message has been customized by using the `PROJECT_NAME` variable in the main Makefile, which has been set as `mlewp-ed2-ch4-outliers`. In fact, the top of the Makefile has several variables set for this project:

```
MODULE := outliers  
PROJECT_NAME := mlewp-ed2-ch4-outliers  
PYTHON_INTERPRETER := python3  
ARCH := $(shell $(PYTHON_INTERPRETER) -c "import platform; print(platform.platform())")  
VIRTUALENV := conda  
CONDA_EXE ?= ~/anaconda3/bin/conda  
EASYDATA_LOCKFILE := environment.$(ARCH).lock.y
```

The `MODULE` variable is referring to the name of the package as before. `PYTHON_INTERPRETER`, `CONDA_EXE`, and `VIRTUALENV` are hopefully self-explanatory. `ARCH` is grabbing architecture information from the local system. `EASYDATA_LOCKFILE` refers to a file that will be created as we work that helps us track the full list of dependencies in our project.

You can see that the help message clearly refers to different targets for the Makefile, so let's explore each of these in turn. First, in order to standardize the creation of a new Conda environment for the project, if one is required, there are a few steps that can be brought together:

```
$(EASYDATA_LOCKFILE): environment.yml
ifeq (conda, $(VIRTUALENV))
    $(CONDA_EXE) env update -n $(PROJECT_NAME)
    $(CONDA_EXE) env export -n $(PROJECT_NAME)
    # pip install -e . # uncomment for conda <
else
    $(error Unsupported Environment `$(VIRTUALE
endif

.PHONY: create_environment
# Set up virtual (conda) environment for this p
create_environment: $(EASYDATA_LOCKFILE)
ifeq (conda,$(VIRTUALENV))
    @rm -f $(EASYDATA_LOCKFILE)
    @echo
    @echo "New conda env created. Activate with"
    @echo ">>> conda activate $(PROJECT_NAME)"
    @echo ">>> make update_environment"
    ifneq ("X$(wildcard .post-create-enviro
            @cat .post-create-environment.txt
    endif
else
    $(error Unsupported Environment `$(VIRTUALE
endif
```

Walking through this step by step, this code states that if `conda` is the virtual environment, then proceed to create or update a Conda environment with the project name and then export the environment into the `environment.yml` file; then after that, export the environment configuration into the lock file. It works this way because `$<` refers to the first prerequisite (in this case, the `environment.yml` file), and `$@` refers to the name of the target (in this case, the `EASYDATA_LOCKFILE` variable). After this is triggered, the second block is checking if `conda` is the virtual env manager before removing the lock file and then providing some guides to the user in the terminal. Note that `@` here is referring to terminal commands.

The next important block in the Makefile is the one that handles updating the environment for you if required, which will often be the case during the “Develop” phase of your project:

```
.PHONY: update_environment
## Install or update Python Dependencies in the
update_environment: environment_enabled $(EASYD
    ifneq ("X$(wildcard .post-update-environmen
            @cat .post-update-environment.txt
    endif
```

This block ensures that if you run the following command in your terminal:

```
make update_environment
```

Then you will create a new `lockfile.yml` with all of the details of the latest version of the environment. There is also a `delete_environment` target that will clear out the lockfiles and remove the Conda environment, as well as some other helper targets that need not concern us here, but that you can explore in the book repository.

Bringing this all together, the workflow using this Makefile-based approach would be:

1. Create a starter `environment.yml` file for the project. This could be very simple; as an example, for the `outliers` package that we are building in the chapter, I started with an `environment.yml` file that looked something like this:

```
name: mlewp-ed2-ch4-outliers
channels:
  - conda-forge
dependencies:
  - python=3.10.8
  - scikit-learn
  - pandas
  - numpy
  - pytest
  - pytest-cov
  - pip
```

2. Create the environment with:

```
make create_environment
```

3. Update the environment, which will create the first lockfile:

```
make update_environment
```

4. As you develop your solution, if you need a new package, go into the `environment.yml` file and add the dependency you require before running `make update_environment`. The idea here is that by not installing the packages manually but by mandating them in the `environment.yml` file, you are creating a more repeatable and robust workflow.

It becomes impossible for you to forget what you installed and what you haven't! For example, if I wanted to add the `bandit` package to this environment, I would go into the `environment.yml` file using my text editor or IDE and would simply add that dependency in either the `conda` or `pip` dependency:

```
name: mlewp-ed2-ch4-outliers
channels:
  - conda-forge
dependencies:
  - python=3.10.8
  - scikit-learn
  - pandas
  - numpy
  - pytest
  - pytest-cov
  - bandit
  - pip
```

And that's it! This is how you can use Makefiles to manage your Conda environments in a far more repeatable way. As mentioned above, if you wanted to start again, you can delete the environment by running:

```
make delete_environment
```

That covers this particular method for managing your Python development environment when developing your packages. We will now move on to discuss one of the most popular tools for Python dependency management and packaging in use today, **Poetry**.

Getting all poetic with Poetry

Python package management is one of the things about the language that definitely does not have people screaming praise from the rooftops. It has been widely admitted by even the most ardent supporters of the language (myself included) that Python's package management is, to put it bluntly, a bit of a mess. The examples we have walked through using `setup.py` and the production of wheels are some of the most accepted ways and, as mentioned, recommended by the PyPA. But they are still not the simplest or most intuitive approaches you would expect from a language that otherwise holds these as key design principles.

Thankfully, over the past few years, there have been a few major developments, one of which we will cover in detail here. This is the creation of the Python packaging and dependency management tool, Poetry. Poetry's benefits include its ease of use and its drastic simplification of the packaging up and dependency management of your solution. Most visibly, it does this by requiring only one configuration file, the

`pyproject.toml` file, rather than a potential setup including `setup.py`, `setup.cfg`, `MANIFEST.in`, or `Pipfile` configuration files. There is also a big advantage in the fact that the dependency file is locked so that auto-updates do not occur, which means the admin (you) has to explicitly call out changes in dependencies. This helps to make the project more stable.

So, this sounds great, but how do we get started? Well, no surprise, we first install the tool using `pip`:

```
pip install poetry
```

Then, if you wanted to start a new project that leveraged Poetry, you would go into the appropriate directory where you want your package to be and run the command:

```
poetry new my-ml-package
```

This would then create a sub-directory structure like:

```
├── README.md
├── my_ml_package
│   └── __init__.py
├── poetry.lock
└── pyproject.toml
└── tests
    └── __init__.py
```

The `tests` folder will be where we place our unit tests, as covered in the *Testing* section of this chapter. `pyproject.toml` is the most important file in the directory. It specifies the main metadata concerning the project and is organized into blocks covering package dependencies for production and for development and testing.

The file generated when I ran the previous commands was the one shown in *Figure 4.3*:

```
[tool.poetry]
name = "my-ml-package"
version = "0.1.0"
description = ""
authors = ["AndyMc629 <andrewpmcmahon629@gmail.com>"]
readme = "README.md"
)packages = [{include = "my_ml_package"}]

)[tool.poetry.dependencies]
)python = "^3.10"

)[build-system]
requires = ["poetry-core"]
)build-backend = "poetry.core.masonry.api"
```

Figure 4.3: The pyproject.toml file created by Poetry when we create a new project.

In the first case, this is given in a block under `[tool.poetry]`, which covers high-level information about the package. Then there is `[tool.poetry.dependencies]`, which currently only contains

Python version 3.10 and nothing else, as I have not used it to install anything else yet. The `[build-system]` section contains details of the build-time dependencies, here only listing `poetry-core`.

If we then want to add a new dependency, such as `pytest`, we can run a command like:

```
poetry add pytest
```

This will output in the terminal something like that shown in *Figure 4.4*.

```
Using version ^7.2.1 for pytest

Updating dependencies
Resolving dependencies... Downloading https://files.pythonhosted.org/packages/ed/35/a31aed2993e398f6b09a790a18
Resolving dependencies... (1.0s)

Writing lock file

Package operations: 4 installs, 0 updates, 0 removals

  • Installing exceptiongroup (1.1.0)
  • Installing iniconfig (2.0.0)
  • Installing pluggy (1.0.0)
  • Installing pytest (7.2.1)
```

Figure 4.4: Output from adding a new package to a Poetry-managed project.

This will also update the `pyproject.toml` file with the new dependency, as shown in *Figure 4.5*.

```
[tool.poetry]
name = "my-ml-package"
version = "0.1.0"
description = ""
authors = ["AndyMc629 <andrewpmcmahon629@gmail.com>"]
readme = "README.md"
packages = [{include = "my_ml_package"}]

[tool.poetry.dependencies]
python = "^3.10"
pytest = "^7.2.1"

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

Figure 4.5: Updated pyproject.toml file after adding a new dependency.

Now, the `[tool.poetry.dependencies]` section is the place where you should define all of the packages you need to be installed at runtime for your package, so you do not necessarily want to bloat this with lots of testing packages.

Instead, Poetry allows you to define a block that lists your development dependencies by specifying a

`[tool.poetry.group.dev.dependencies]` block, like that shown in *Figure 4.6*.

```
[tool.poetry]
name = "my-ml-package"
version = "0.1.0"
description = ""
authors = ["AndyMc629 <andrewpmcmahon629@gmail.com>"]
readme = "README.md"
packages = [{include = "my_ml_package"}]

[tool.poetry.dependencies]
python = "^3.10"

[tool.poetry.group.dev.dependencies]
pytest = "^7.2.1"
pytest-mock = "*"

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

Figure 4.6: The pyproject.toml file with a set of development dependencies.

When you install Poetry, it creates its own virtual environment in order to create appropriate isolation from the rest of your system. You can activate this environment if you are on a Linux system by running:

```
source /path/to/venv/bin/activate
```

Or you can also run:

```
poetry shell
```

If you already have a Python virtual environment running through Conda, `venv`, or some other tool, then Poetry is actually aware of this and works within it. This can be very helpful as you can use Poetry to manage that virtual environment, rather than starting completely from scratch. In this case, you may get some output in the terminal like that shown in *Figure 4.7*.

```
|Virtual environment already activated: /home/andrew/anaconda3/envs/mlewp-ed2-ch4
```

Figure 4.7: Output from poetry shell command if you are already working in a Python virtual environment.

To close down this environment but not the shell you are running, you can use the command:

```
deactivate
```

If you want to close the environment and the shell (be warned, this will likely close your terminal), you can type the command:

```
exit
```

To install the dependencies you have been adding in the `pyproject.toml` file, you can run:

```
poetry install
```

This will either download and install all the dependencies listed in your `pyproject.toml` file, grabbing the latest versions from `pip`, or it will grab and install the versions of these packages as they are listed in the `.lock` file. This is to ensure that even if multiple people have been working in the environment and running `poetry install` commands, the environment is kept stable with consistent package versions. This is exactly the same reason for using a `.lock` file in the section on Makefiles earlier in this chapter. When the `install` command was run for this `my-ml-package` project, for example, the output was that shown in *Figure 4.8*.

```
Installing dependencies from lock file
```

```
Package operations: 1 install, 0 updates, 0 removals
```

- Installing `pytest-mock (3.10.0)`

```
Installing the current project: my-ml-package (0.1.0)
```

Figure 4.8: Poetry installing packages from the `.lock` file in order to maintain environment stability.

All of the preceding commands are around the basic management of the environment, but what about when we want to do something with this environment? Well, if you have a script called `main.py`, you can run this using the Poetry-configured environment via the command:

```
poetry run python main.py
```

We do not have anything like this in the `my-ml-package`. Instead, since we are building a library, we can package and deploy the package by running:

```
poetry build
```

This gives the output shown in *Figure 4.9*.

```
Building my_ml_package (0.1.0)
- Building sdist
- Built my_ml_package-0.1.0.tar.gz
- Building wheel
- Built my_ml_package-0.1.0-py3-none-any.whl
```

Figure 4.9: The output when Poetry builds our simple package.

If you want to publish to PyPI, which only works if you are a registered user with the correct credentials configured, you can just run the command `poetry publish`. If you want to publish to some other private repository, you can run:

```
poetry publish -r private-repository-location
```

After all of this, you can probably see how Poetry can make things a lot clearer when it comes to package development. We have been able to manage stable development and production (like Python environments that can be worked on by multiple developers without fear of corruption), build and publish our package, as well as run any scripts and processes we want to – all in a few commands!

Next, let's cover some of the steps we can take to ensure that our packages are robust and can be trusted to work or fail gracefully and be diagnosable if there is an issue.

Testing, logging, securing, and error handling

Building code that performs an ML task may seem like the end goal, but it is only one piece of the puzzle. We also want to be confident that this code will work, and if it doesn't, we will be able to fix it. This is where the concepts of testing, logging, and error handling come in, which the next few sections cover at a high level.

Testing

One of the most important features that sets your ML-engineered code apart from typical research scripts is the presence of robust testing. It is critical that any system you are designing for deployment can be trusted not to fall down all the time and that you can catch issues during the development process.

Luckily, since Python is a general-purpose programming language, it is replete with tools for performing tests on your software. In this chapter, we will use **pytest**, which is one of the most popular, powerful, and easy-to-use testing toolsets for Python code available. pytest is particularly useful if you are new to testing because it focuses on building tests as standalone Python functions that are quite readable, whereas other packages can sometimes lead to the creation of clunky testing classes and complex `assert` statements. Let's dive into an example.

First, let's start by writing tests for some pieces of code defined in the rest of this chapter from our `outliers` package. We can define a simple test to ensure that our data helper function actually creates some numerical data that can be used for modeling. To run this sort of test in pytest, we first

create a file with `test_` or `_test` in the name somewhere in our test's directory—pytest will automatically find files that have this in their name. So, for example, we may write a test script called `test_create_data.py` that contains the logic we need to test all of the functions that refer to creating data within our solution. Let's make this explicit with an example:

1. Import the relevant modules we will need from the package and anything else we need for testing. Here, we import `pytest` because we will use some functionality from it in later steps but, in general, you don't need to import this:

```
import numpy
import pytest
import outliers.utils.data
```

2. Then, since we want to test the function for creating data, it would be good to only generate the data once, then test its attributes in a variety of ways. To do this, we employ the `fixture` decorator from pytest, which allows us to define an object that can be read into several of our tests. Here, we use this so that we can apply our tests using `dummy_data`, which is just the output of the `create_data` function:

```
@pytest.fixture()
def dummy_data():
    data = outliers.utils.data.create_data()
    return data
```

3. Finally, we can actually write the tests. Here are two examples that test if the dataset created by the function is a `numpy` array and if it has more than `100` rows of data:

```
def test_data_is_numpy(dummy_data):
    assert isinstance(dummy_data, numpy.ndarray)
def test_data_is_large(dummy_data):
    assert len(dummy_data) > 100
```

We can write as many of these tests and as many of these types of test modules as we like. This allows us to create a high degree of **test coverage** across our package.

4. You can then enter the following command in the terminal at the top level of your project in order to run all the tests in the package:

```
$ pytest
```

5. Then you will see a message like this, telling us what tests have run and which have passed and failed:

```
===== test session starts =====
platform linux -- Python 3.8.5, pytest-6.1.1, py-1.9.0, pluggy-0.13.1 -- /home/andrew/anaconda3/envs/mleng/bin/python
cachedir: .pytest_cache
rootdir: /home/andrew/dev/github/Machine-Learning-Engineering-with-Python/chapter4/outlier_package
collected 2 items

outliers/tests/test_create_data.py::test_data_is_numpy PASSED
outliers/tests/test_create_data.py::test_data_is_large PASSED

===== 2 passed in 0.45s =====
```

Figure 4.10: The output of a successful unit test in pytest.

The previous example showed how to write and execute some basic tests on our data utilities. We can now expand on this by testing some

of the more sophisticated functionality in the package – namely, the model creation process.

6. Similarly to the previous case, we create a script for holding our tests in `tests/test_detectors.py`. Since we are testing more complex functionality, we will have to import more pieces of the package into the script:

```
import pytest
from outliers.detectors.detection_models import DetectionModels
from outliers.detectors.pipelines import OutlierDetector
from outliersdefinitions import MODEL_CONFIG_PATH
import outliers.utils.data
import numpy as np
```

7. We will have the same fixture for dummy data created as in *Step 2*, but now we also have a fixture for creating some example models to use in tests:

```
@pytest.fixture()
def example_models():
    models = DetectionModels(MODEL_CONFIG_PATH)
    return models
```

8. Our final fixture creates an example detector instance for us to use, based on the previous model's fixture:

```
@pytest.fixture()
def example_detector(example_models):
```

```
model = example_models.get_models()[0]
detector = OutlierDetector(model=model)
return detector
```

9. And now we are ready to test some of the model creation functionality. First, we can test that the models we created are not empty **objects**:

```
def test_model_creation(example_models):
    assert example_models is not None
```

10. We can then test that we can successfully retrieve models using the instance of **DetectionModels** created in *Step 6*:

```
def test_model_get_models(example_models):
    example_models.get_models() is not None
```

11. Finally, we can test that the results found by applying the model pass some simple tests. This shows that the main pieces of our package are working for an end-to-end application:

```
def test_model_evaluation(dummy_data, example_detector):
    result = example_detector.detect(dummy_data)
    assert len(result[result == -1]) == 39 #S
    assert len(result) == len(dummy_data) #S
    assert np.unique(result)[0] == -1
    assert np.unique(result)[1] == 1
```

12. As in *Step 4*, we can run the full test suite from the command line. We add a verbosity flag to return more information and show the individual tests that pass. This helps confirm that both our data utility and our model tests are being triggered:

```
pytest --verbose
```

13. The output is shown in the following screenshot:

```
===== test session starts =====
platform linux -- Python 3.8.5, pytest-6.1.1, py-1.9.0, pluggy-0.13.1 -- /home/andrew/anaconda3/envs/mleng/bin/python
cachedir: .pytest_cache
rootdir: /home/andrew/dev/github/Machine-Learning-Engineering-with-Python/chapter4/outlier_package
collected 5 items

outliers/tests/test_create_data.py::test_data_is_numpy PASSED
outliers/tests/test_create_data.py::test_data_is_large PASSED
outliers/tests/test_detectors.py::test_model_creation PASSED
outliers/tests/test_detectors.py::test_model_get_models PASSED
outliers/tests/test_detectors.py::test_model_evaluation PASSED
```

Figure 4.11: Output of successful tests on both data and model functionality.

The running of these tests can be automated either via the inclusion of **githooks** in our repository or through the use of other tools, such as the **Makefile** used for the project.

We'll now move on to consider how we can log information about our code as it runs, which can help with debugging and general monitoring of your solution.

Securing your solutions

As software engineers of any kind, we should always be very cognizant of the fact that there is a flip-side to the joy of building products that people use. This flip-side is that it is then your job to make sure the solution is

secure and safe for those users. In the words of Uncle Ben, “*With great power comes great responsibility.*”

Now, cybersecurity is a huge discipline in its own right, so we cannot do it justice here. The following sections will simply aim to introduce some useful tools and explain the basics of using them in order to make your solutions more secure and trustworthy.

First, we need to understand the different ways we can create secure solutions:

- Testing the application and code itself for internal bugs.
- Screening packages and scanning other code used for security vulnerabilities.
- Testing for data leaks and data exposure.
- Developing robust monitoring techniques, specifically with respect to the above points and less so about the monitoring of your ML models, which has been covered elsewhere in this book.

In the first case, this refers mostly to things like our unit-testing approaches, which, again, we have covered elsewhere, but in brief, this refers to the act of testing the functionality of the code you write in order to ensure it works as expected. In the section on model monitoring, it was mentioned that performing standard tests for the expected performance of a machine learning model can be difficult and so requires specific techniques.

Here, we are focused more on the general application code and the solution wrapping the main model.

In the case of screening packages and code, this is a very pertinent and, thankfully, easy-to-implement challenge. The reader may recall that in 2022, there was a wave of activity across the world as organizations and

software engineers tried to deal with the discovery of a bug in the Java-based Log4j library. Bugs and security flaws will always happen and not always be detected, but this point is all about having some system in place to automatically scan the code and packages you are using in your solution to find these proactively, saving major headaches (and far worse) for the users of your code.

Data leakage is an incredibly important topic now. Regulations like the **General Data Protection Regulation (GDPR)** in the European Union have placed a massive emphasis on the management and curation of customers' data. Since machine learning systems are fundamentally data-driven applications, this means that concerns around privacy, usage, storage, and many other points become extremely important to consider in designs and implementations. It is important to note that what we are discussing here goes far beyond the “garbage in, garbage out” question of data quality, and really is about how securely you are holding and transferring the data required to make your machine learning system work.

Analyzing your own code for security issues

As you will have noticed throughout this book, the Python open-source community has almost every challenge you can think of covered in at least some way, and when it comes to security, this is no different. To perform static analysis of your own code and check for vulnerabilities that may have been introduced during development, you can use the open-source Bandit package, <https://bandit.readthedocs.io/en/latest/>. This is a linter that is focused on finding security issues in source code, and it is extremely easy to run.

First, as always, we need to install Bandit. We can now do this using the Makefile magic we learned about in the earlier section on *Building your package*, so we add the Bandit package to the `pip` dependencies in the `environment.yml` file and run the command:

```
make update_environment
```

Then, to run Bandit on your source code, you simply run:

```
bandit -r outliers
```

As we mentioned in *Chapter 2, The Machine Learning Development Process*, it is always useful to automate any development steps that you will want to run again and again. We can do this with Bandit by adding the following to our `.pre-commit-config.yaml` in our Git directory:

```
repos:
- repo: https://github.com/PyCQA/bandit
  rev: '' # Update me!
  hooks:
    - id: bandit
```

This means that after every commit, we will run the `bandit` command as outlined in the previous two steps.

The output from running Bandit on some example code is given by a series of blocks like the following in *Figure 4.12*.

```

[main] INFO profile include tests: None
[main] INFO profile exclude tests: None
[main] INFO cli include tests: None
[main] INFO cli exclude tests: None
[main] INFO running on Python 3.10.9
RUN started:2023-01-20 21:00:55.255019

Test results:
>> Issue: [B101:assert_used] Use of assert detected. The enclosed code will be removed when compiling to optimised byte code.
    Severity: Low Confidence: High
    CWE: CWE-703 (https://cwe.mitre.org/data/definitions/703.html)
    Location: outliers/tests/test_create_data.py:12:4
    More Info: https://bandit.readthedocs.io/en/1.7.4/plugins/b101\_assert\_used.html
11     def test_data_is_numpy(dummy_data):
12         assert isinstance(dummy_data, numpy.ndarray)
13
-----
```

Figure 4.12: Bandit output on a typical piece of code.

This is followed by a small summary report at the end of the output, shown in *Figure 4.13*.

```

Code scanned:
    Total lines of code: 98
    Total lines skipped (#nosec): 0

Run metrics:
    Total issues (by severity):
        Undefined: 0
        Low: 7
        Medium: 0
        High: 0
    Total issues (by confidence):
        Undefined: 0
        Low: 0
        Medium: 0
        High: 7
Files skipped (0):
```

Figure 4.13: The Bandit tool provides a high-level summary at the end of its output for diagnosing the state of your code base.

There are many more features of Bandit but this shows how easy it is to get started and start analyzing the potential issues in your Python code base.

Analyzing dependencies for security issues

As we have outlined, we don't just want to scan the code we have written for security vulnerabilities; it is also important that we try and find any issues in the packages that we are using in our solution. This can be done using something like the Python `safety` tool,

<https://pypi.org/project/safety/>. Safety uses a standardized database containing known Python security issues and then compares any packages found in your solution against this database. Please note that, as the documentation for safety calls out:



By default it uses the open Python vulnerability database Safety DB, which is licensed for non-commercial use only.

For all commercial projects, Safety must be upgraded to use a PyUp API using the key option.

Below is an example of using this on the same source code tree as in the example for using Bandit:

1. If you do not have safety installed, install it:

```
pip install safety
```

2. You then need to change into the top folder of your source code tree before running:

```
safety check
```

When I ran this on the folder containing the `outlier_package` we have been building, I got the terminal output shown in *Figure 4.14*:

```
+=====+  
          /$$$$$$$/      /$$/  
         /$$_  $$ |  |  $$\  /$$$$$/ |  /$$/  
        /$$/_/ |  |  $$\ /$$$_/ |  /$$/  
       |  $$_/ |  |  $$\ /$$$_/ |  /$$/  
      |  $$_/ |  |  $$\ /$$$_/ |  /$$/  
     |  $$_/ |  |  $$\ /$$$_/ |  /$$/  
    |  $$_/ |  |  $$\ /$$$_/ |  /$$/  
   |  $$_/ |  |  $$\ /$$$_/ |  /$$/  
by pyup.io  
+=====+
```

REPORT

You are using Safety's free vulnerability database. This data is outdated, limited, and licensed for non-commercial use only.
All commercial projects must sign up and get an API key at <https://pyup.io>

Safety v2.3.5 is scanning for Vulnerabilities...
Scanning dependencies in your environment:

```
-> /usr/local/spark/python  
-> /home/andrew/anaconda3/envs/mlewp-ed2-ch4-outliers/lib/python3.10/site-packages
```

Using non-commercial database
Found and scanned 17 packages
Timestamp 2023-01-20 20:41:05
1 vulnerability found
0 vulnerabilities ignored

VULNERABILITIES FOUND

```
-> Vulnerability found in wheel version 0.37.1  
  Vulnerability ID: 51499  
  Affected spec: <0.38.1  
  ADVISORY: Wheel 0.38.1 includes a fix for CVE-2022-40898: An issue discovered in Python Packaging Authority (PyPA) Wheel 0.37.1 and earlier...  
  CVE-2022-40898  
  For more information, please visit https://pyup.io/v/51499/f17
```

Scan was completed. 1 vulnerability was found.

REMEDIATIONS

1 vulnerability was found in 1 package. For detailed remediation & fix recommendations, upgrade to a commercial license.

```
You are using Safety's free vulnerability database. This data is outdated, limited, and licensed for non-commercial use only.  
All commercial projects must sign up and get an API key at https://pyup.io
```

Figure 4.14: The output of safety on the outliers package.

As you can see from *Figure 4.15*, we are being warned that the current version cannot be used for scanning commercial software and that if this was needed, you should get an API key. For the project here, this is fine. One vulnerability has been found by the tool relating to the version of the `wheel` package. Upon inspection of the `environment.yml` file in the project, we see that we can update this to version 0.38.1 as the advisory note suggests. This is shown in *Figure 4.15*.

```
- sqlite=3.40.1=h5082296_0
- tk=8.6.12=h1ccaba5_0
- tzdata=2022g=h04d1e81_0
- wheel=0.37.1=pyhd3eb1b0_0
- xz=5.2.10=h5eee18b_1
- zlib=1.2.13=h5eee18b_0
- pip
```

Figure 4.15: Updating the environment.yml file in order to avoid the error produced by the safety tool.

Note that the Conda channels used in this `environment.yml` file did not have the `wheel` package in version 0.38.1 or greater so this was added to the `pip` dependencies instead, as shown in *Figure 4.16*.

```
pip:
- certifi==2022.12.7
- pip==22.3.1
- setuptools==65.6.3
- safety
- wheel>=0.38.1
```

Figure 4.16: Updating the pip dependencies of the environment.yml file in the outliers package.

After doing this and re-running the command:

```
safety check
```

The solution is given a clean bill of health, as shown in the report in *Figure 4.17*.

```
+=====  
          /$$$$$$$           /$$  
         /$$__  $$           | $$  
        /$$$$$$$/ | $$ \_/_//$$$$$$$/ /$$$$$$$/ /$$ /$$  
 /$$____/ | ____ $$| $$$$_ /$$__ $$| _ $$/_ | $$ /$$ | $$  
 | $$$$_/ /$$$$$$$/| $$/_ | $$$$_/ /$$$$$_/ | $$ | $$ | $$  
 \____ $$ /$$__ $$| $$ | $$____/ | $$ /$$| $$ | $$ | $$  
 /$$$$$$$/| $$$$_/ | $$ | $$$$_/ | $$$$/| $$$$_/ | $$  
 | ____/ \____/| / \____/ | \____/ \____/ | \____/ | $$  
          /$$ | $$  
          | $$$$_$/|  
          \____/  
+=====
```

REPORT

You are using Safety's free vulnerability database. This data is outdated, limited, and licensed for non-commercial use only.
All commercial projects must sign up and get an API key at <https://pyup.io>

Safety v2.3.5 is scanning for Vulnerabilities...
Scanning dependencies in your environment:

```
-> /home/andrew/anaconda3/envs/mlewp-ed2-ch4-outliers/lib/python3.10/site-packages  
-> /usr/local/spark/python
```

```
Using non-commercial database
Found and scanned 17 packages
Timestamp 2023-01-20 20:55:44
0 vulnerabilities found
0 vulnerabilities ignored
```

No known security vulnerabilities found.

You are using Safety's free vulnerability database. This data is outdated, limited, and licensed for non-commercial use only.

All commercial projects must sign up and get an API key at <https://pyup.io>

Figure 4.17: The safety tool returns zero security vulnerabilities after updating the identified package.

Although safety does require a commercial license in order to get the full set of features, it can still be extremely helpful for sniffing out issues in your dependencies.

Logging

Next, it is important to ensure that as your code is running, the status of the different operations is reported, as well as any errors that occur. This helps make your code more maintainable and helps you debug when there is an issue. For this, you can use the Python `logging` library.

Loggers can be instantiated in your code via logic like this:

```
import logging
logging.basicConfig(filename='outliers.log',
                    level=logging.DEBUG,
                    format='%(asctime)s | %(name)s | %(message)s')
```

This code defines our format for the logging messages and specifies that logging messages of level `DEBUG` or higher will go to the `outliers.log` file. We can then log output and information relevant to our code's running status using the very easy-to-use syntax that comes with the `logging` library:

```
logging.debug('Message to help debug ...')
logging.info('General info about a process that ...
logging.warning('Warn, but no need to error ...
With the settings shown in the first logging sn
```

```
2021-08-02 19:58:53,501 | root | DEBUG | Message
2021-08-02 19:58:53,501 | root | INFO | General
2021-08-02 19:58:53,501 | root | WARNING | Warn
```

What we have shown so far is really the basics of logging and it so far assumes that, although things may not be going perfectly, nothing has errored. This is obviously not always the case! So, what if we want to log an exception or error to our logging file?

Well, this is typically done with the `logging.error` syntax but with an important point we must consider, which is that it is often not enough to just log the fact we've raised an error; we would also like to log the details of the error. So, as discussed in the *Error handling* section, we know that we can execute a `try except` clause on some code and then raise an exception. What we want to do in this case is log the details of that exception to our logging target. To do this we need to know that the `logging.error` method (and the `logging.debug` method) have some important keyword arguments we can use. For more on keyword arguments, see the section on *Tips and tricks* in this chapter. According to the logging documentation,

<https://docs.python.org/3/library/logging.xhtml#logging.debug>, the keyword arguments `exc_info` and `stack_info` are given as Booleans, and `extra` is a dictionary. The keyword argument `exc_info = True` specifies that we wish to return exception information in the logging call, `stack_info = True` will return far more detailed stack trace information for the exception (including the logger call), and `extra` can be set equal to a dictionary with extra information that the developer has defined.

The extra information in this case is then provided with the initial part of the record, as part of the identifier of the event. This is a good way to provide some bespoke information to your logging calls.

As an example, let us consider a bespoke feature transformation function, which, in this case, will actually not do anything useful, just return the original DataFrame like so:

```
def feature_transform(df: pd.DataFrame) -> pd.D
    """Transform a dataframe by not doing anyth
    :param df: a dataframe.
    :return: df.mean(), the same dataframe with
    """
    return df.mean()
```

If we want to raise an exception when this function failed and log details about the error, we could write something like this:

```
try:
    df_transformed = feature_transform(df)
    logging.info("df successfully transformed")
except Exception as err:
    logging.error("Unexpected error", exc_info=
```

If we run this code on a simple dummy pandas DataFrame like the one below, the code will execute without issue:

```
df = pd.DataFrame(data={'col1': [1, 2, 3, 4], 'col
```

If, however, we do the same, but this time, we run the code on something that does not have the pandas DataFrame `mean()` syntax, like a list:

```
list_of_nums = [1, 2, 3, 4, 5, 6, 7, 8]
try:
    df_transformed = feature_transform(list_of_nums)
    logging.info("df successfully transformed")
except Exception as err:
    logging.error("Unexpected error", exc_info=True)
```

We will activate the error and log the details to the logging file. Note that in the below code snippet, the file path was shortened for the screenshot, shown in *Figure 4.18*.

```
2023-01-22 21:31:57,317 | root | ERROR | Unexpected error
Traceback (most recent call last):
  File "/home/andrew/dev/github/Machine-Learning-Engineering-with-Python-Second-Edition
    df_transformed = feature_transform(list_of_nums)
  File "/home/andrew/dev/github/Machine-Learning-Engineering-with-Python-Second-Edition
    return df.mean()
AttributeError: 'list' object has no attribute 'mean'
```

Figure 4.18: The output to the log file when we use the `exc_info = True` flag.

Error handling

The last piece of *housekeeping* to cover in this section is error handling. It is important to remember that when you are an ML engineer, your aim is to build products and services that work, but an important part of this is

recognizing that things do not always work! It is therefore important that you build in patterns that allow for the escalation of (inevitable) errors during runtime. In Python, this is typically done via the concept of *exceptions*. Exceptions can be raised by the core Python functions and methods you are using. For example, imagine you ran the following code without defining the variable `x`:

```
y = 10*x
```

The following exception would be raised:

```
NameError: name 'x' is not defined
```

The important point for us as engineers is that we should build solutions in which we can confidently control the flow of errors. We may not always want our code to break when an error occurs, or we may want to ensure that very specific messages and logging occur upon certain expected edge cases. The simplest technique for doing this is via `try except` blocks, as seen in the following code block:

```
try:  
    do_something()  
except:  
    do_something_else()
```

In this case, `do_something_else()` is executed if `do_something()` runs into an error.

We will now finish with a comment on how to be efficient when building your solutions.

Error handling in Python is often built around the idea of “exceptions,” which are just events that disrupt the expected functioning of the program.

The full list of exceptions in Python contains around 50 different types.

Below is an excerpt taken from the Python documentation, with ellipses highlighting where I have not shown the full details:

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
        |    +-- FloatingPointError
        |    +-- OverflowError
        |    +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError

...
+-- Warning
    +-- DeprecationWarning
    +-- PendingDeprecationWarning
    +-- RuntimeWarning
    +-- SyntaxWarning
```

```
+-- UserWarning
```

```
...
```

You can see from the exception list that these are organized in a hierarchy. This means that raising an exception at a lower level is simply a more specific instance of an exception at a higher level, so you can actually raise it at a higher level of the hierarchy and everything still works correctly. As a quick example, we can see from the `ArithmeticError` sub-hierarchy that there are three exceptions at a lower level in the hierarchy:

```
+-- ArithmeticError
|   +-- FloatingPointError
|   +-- OverflowError
|   +-- ZeroDivisionError
```

This then means that if we were to raise an exception for a piece of code we think may divide a number by zero, we can legally use `ZeroDivisionError` or `ArithmeticError`:

```
n = 4.0
try:
    result = n/0.0
except ZeroDivisionError:
    print("Division by zero not allowed!")
n = 4.0
try:
    result = n/0.0
except ArithmeticError:
    print("Division by zero not allowed!")
```

In general, when you catch an exception, there are a few different routes you can go down to handle it. First, you could handle the exception and continue the program flow. You should do this when it is clear what will be causing the error and it can be handled within the logic of the code. Second, you could raise the exception again. You may want to do this for a few reasons:

1. You want to log the error but still allow the exception to propagate up the call stack. This can be useful for debugging purposes, as it allows you to log the error message and other details about the exception, while still allowing the calling code to handle the exception as appropriate. An example may look something like this:

```
import logging

def process_data(data):
    try:
        # Do some processing on the data
        result = process(data)
    except Exception as e:
        # Log the exception
        logging.exception("Exception occurred while processing data")
        # Re-raise the exception
        raise
    return result
```

In this example, the `process_data` function tries to process some data and returns the result. If an exception occurs while processing the

data, the exception is logged using the `logging.exception` function, which logs the exception along with a stack trace. The exception is then re-raised using the `raise` statement, which allows the calling code to handle the exception as appropriate.

2. You want to add additional context to the exception. For example, you might want to add information about the state of your application when the exception occurred, or about the input that led to the exception being raised. Adapting the previous example, we may then have something like:

```
def process_data(data):
    try:
        # Do some processing on the data
        result = process(data)
    except Exception as e:
        # Add additional context to the exception message
        message = f"Exception occurred while processing {data}"
        # Create a new exception with the modified message
        raise Exception(message) from e
    return result
```

In this example, if an exception occurs while processing the data, the exception message is modified to include the data that was being processed. A new exception is then raised using the modified message and the original exception as the cause (using the `from e` syntax).

3. You want to handle an exception in a higher level of your code but still allow lower-level code to handle the exception if it is not appropriate

to handle it at the higher level. This is a bit more complex, so we will try and walk through another adapted example step by step. First, this time, when we raise the exception, we call a function that handles the exception in a bespoke way, called `handle_exception`:

```
def process_data(data):
    try:
        # Do some processing on the data
        result = process(data)
    except Exception as e:
        # Handle the exception
        handle_exception(e)
        # Re-raise the exception
        raise
    return result
```

The code for `handle_exception` would look something like the following, where we have to determine if we want to handle the exception at this level of abstraction or pass it up the call stack, using another function called `should_handle`:

```
def handle_exception(e):
    # Log the exception
    logging.exception("Exception occurred")
    # Check if the exception should be handled
    if should_handle(e):
        # Handle the exception
        ...
    else:
```

```
# Allow the exception to propagate up
raise
```

The `should_handle` function would then be where we define our bespoke logic for deciding if we handle the exception at the current level or use the `raise` syntax to escalate up the call stack. For example, if we want to handle an `ArithmeticError` at this level and otherwise we want to raise up the call stack, the logic would look like this:

```
def should_handle(e):
    # Check the type of the exception
    if isinstance(e, ArithmeticError):
        # Handle the exception
        return True
    else:
        # Allow the exception to propagate
        return False
```

4. Finally, you may raise a different exception, perhaps because you need to bring together a few different exceptions and deal with them together at a higher level of abstraction. Once again, adapting the previous examples, this may mean that you write some code that looks like this:

```
def process_data(data):
    try:
        # Do some processing on the data
        result = process(data)
```

```
        except ValueError as e:
            # Raise a different exception with t
            raise MyCustomException(str(e))
        except MyCustomException as e:
            # Raise a different exception with a
            message = f"Exception occurred while
                        {data}"
            raise MyCustomException(message)
    return result
```

In this example, if a `ValueError` exception occurs while processing the data, it is caught and a new `MyCustomException` is raised with the same message. If a `MyCustomException` exception occurs, it is caught and a new `MyCustomException` is raised with a modified message that includes the `data` that was being processed. This allows you to deal with different types of exceptions together at a higher level of abstraction, by raising a single, custom exception type that can be handled in a consistent way.

A third program flow that we can use is that we can raise a new exception from within the original exception. This can be helpful because we can provide more detailed information about the type of error that has occurred, and we can give more contextual information that will help us debug any issues down the line. To make this clearer, let's define an example function to stand in for the function we've been using in the previous examples:

```
def process(data_to_be_processed):
    '''Dummy example that returns original data
    return data_to_be_processed + 1
```

We will call this in the same function as we had in the first example of the list above but now we will add a new piece of syntax to raise an exception from the original exception:

```
def process_data(data):
    try:
        # Do some processing on the data
        result = process(data)
    except Exception as e:
        # Log the exception
        logging.exception("Exception occurred w
        # Raise a new exception from the overall
        new_exception = ValueError("Error proce
        raise new_exception from e
    return result
```

The raising from the original exception now ensures that we have logged the fact that this was specifically a `ValueError` related to the input data, and it allows us to log a higher-level message that can give additional context in the stack trace. For example, if you use the above functions and run this function call:

```
process_data('3')
```

We get an error, as expected, since we are supplying a string and then trying to add an integer to it. A snippet from the stack trace I got when I ran this is given below:

```
ERROR:root:Exception occurred while processing
  File "exception_handling_examples.py", line 5
    return data_to_be_processed + 1
TypeError: can only concatenate str (not "int")
```

The above exception was the direct cause of the following exception:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "exception_handling_examples.py", line 1
    raise new_exception from e
ValueError: Error processing data
```

You can see how the exception we raised from the original exception calls out where in the `process_data` function the error has occurred, and it has also given us the information that this issue relates to the processing of the data. Both pieces of information help provide context and can help us debug. The more technical original exception, the `TypeError` referring to the operand types, is still useful but could be hard to digest and fully debug on its own.

This only scratches the surface of what is possible when it comes to logging, but this will allow you to get started.

Now, we move on to what we need to do in our code to handle scenarios where things go wrong!

Not reinventing the wheel

You will already have noticed through this chapter (or I hope you have!) that a lot of the functionality that you need for your ML and Python project has already been built. One of the most important things you can learn as an ML engineer is that you are not supposed to build everything from scratch. You can ensure you do not do this in a variety of ways, the most obvious of which is to use other packages in your own solution and then build a functionality that enriches what is already there. As an example, you do not need to build basic regression modeling capabilities since they exist in a variety of packages, but you might have to add a new type of regressor or use some specific domain knowledge or trick you have developed. In this case, you would be justified in writing your own code on top of the existing solution. You can also use a variety of concepts from Python, such as wrapper classes or decorators. The key message is that although there is a lot of work for you to do when building your ML solutions, it is important that you do not feel the need to build everything from scratch. It is far more efficient to focus on where you can create added value and build on what has gone before!

Summary

This chapter has been all about best practices for when you write your own Python packages for your ML solutions. We went over some of the basic concepts of Python programming as a refresher before covering some tips and tricks and good techniques to bear in mind. We covered the importance of coding standards in Python and PySpark. We then performed a comparison between object-oriented and functional programming paradigms for writing your code. We moved on to the details of taking the high-quality code you have written and packaging it up into something you

can distribute across multiple platforms and use cases. To do this, we looked into different tools, designs, and setups you could use to make this a reality, including the use of Makefiles and Poetry. We continued with a summary of some housekeeping tips for your code, including how to test, log, and monitor your solution. This also included some detailed examples of exception handling and how you can develop more sophisticated control flows in your programs and packages. We finished with a brief *philosophical* point on the importance of not reinventing the wheel.

In the next chapter, we will take a deep dive into the world of deployment. This will be all about how you take scripts, packages, libraries, and apps that you have written and run them on appropriate infrastructure and tools.

Join our community on Discord

Join our community's Discord space for discussion with the author and other readers:

<https://packt.link/mle>



5

Deployment Patterns and Tools

In this chapter, we will dive into some important concepts around the deployment of your **machine learning (ML)** solution. We will begin to close the circle of the ML development lifecycle and lay the groundwork for getting your solutions out into the world.

The act of deploying software, of taking it from a demo you can show off to a few stakeholders to a service that will ultimately impact customers or colleagues, is a very exhilarating but often challenging exercise. It also remains one of the most difficult aspects of any ML project and getting it right can ultimately make the difference between generating value or just hype.

We are going to explore some of the main concepts that will help your ML engineering team cross the chasm between a fun proof-of-concept to solutions that can run on scalable infrastructure in an automated way. This will require us to first cover questions of how to design and architect your ML systems, particularly if you want to develop solutions that can be scaled and extended seamlessly. We will then discuss the concept of containerization and how this allows your application code to be abstracted from the specific infrastructure it is being built or run on, allowing for portability in many different cases. We will then move on to a concrete

example of using these ideas to deploy an ML microservice on AWS. The rest of the chapter will then return to the question of how to build effective and robust pipelines for your end-to-end ML solution, which was introduced in *Chapter 4, Packaging Up*. We will introduce and explore **Apache Airflow** for building and orchestrating any generic Python process, including your data preparation and ML pipelines. Then we will finish up with a similar deep dive on **ZenML** and **Kubeflow**, two open-source advanced ML pipelining tools that are now extensively used in industry. This collection of tools means that you should finish this chapter feeling very confident that you can deploy and orchestrate quite complex ML solutions using a variety of software.

This will all be broken down into the following sections:

- Architecting systems
- Exploring some standard ML patterns
- Containerizing
- Hosting your own microservice on **Amazon Web Services (AWS)**
- Building general pipelines with Airflow
- Building advanced ML pipelines

The next section will kick things off with a discussion of how we can architect and design our ML systems with deployment in mind. Let's go!

Technical requirements

As with the other chapters, you can set up your Python development environment to be able to run the examples in this chapter by using the supplied Conda environment `yml` file or the `requirements.txt` files from the book repository, under *Chapter05*.

```
conda env create -f mlewp-chapter05.yml
```

You will also require some non-Python tools to be installed to follow the examples from end to end. Please see the respective documentation for each tool:

- AWS CLI v2
- Postman
- Docker

Architecting systems

No matter how you are working to build your software, it is always important to have a design in mind. This section will highlight the key considerations we must bear in mind when architecting ML systems.

Consider a scenario where you are contracted to organize the building of a house. We would not simply go out and hire a team of builders, buy all the supplies, hire all the equipment, and just tell everyone to *start building*. We would also not assume we knew exactly what the client who hired us wants without first speaking to them.

Instead, we would likely try to understand what the client wanted in detail, and then try to design the solution that would fit their requirements. We would potentially iterate this plan a few times with them and with appropriate experts who knew the details of pieces that fed into the overall design. Although we are not interested in building houses (or maybe you are, but there will not be any in this book!), we can still see the analogy with software. Before building anything, we should create an effective and clear design. This design provides the direction of travel for the solution

and helps the build team know exactly what components they will work on. This means that we will be confident that what we build will solve the end user's problem.

This, in a nutshell, is what software architecture is all about.

If we did the equivalent of the above example for our ML solution, some of the following things may happen. We could end up with a very confusing code base, with some ML engineers in our team building elements and functionality that are already covered by the work that other engineers have done. We may also build something that fundamentally cannot work later in the project; for example, if we have selected a tool that has specific environmental requirements we cannot meet due to another component. We may also struggle to anticipate what infrastructure we need to be provisioned ahead of time, leading to a disorganized scramble within the project to get the correct resource. We may also underestimate the amount of work required and miss our deadline. All of these are outcomes we wish to avoid and can be avoided if we are following a good design.

In order to be effective, the architecture of a piece of software should provide at least the following things to the team working on building the solution:

- It should define the functional components required to solve the problem in totality.
- It should define how these functional components will interact, usually through the exchange of some form of data.
- It should show how the solution can be extended in the future to include further functionality the client may require.
- It should provide guidance on which tools should be selected to implement each of the components outlined in the architecture.

- It should stipulate the process flow for the solution, as well as the data flow.

This is what a piece of good architecture should do, but what does this actually mean in practice?

There is no strict definition of how an architecture has to be compiled. The key point is that it acts as a design against which building can progress. So, for example, this might take the form of a nice diagram with boxes, lines, and some text, or it could be a several-page document. It might be compiled using a formal modeling language such as **Unified Modeling Language (UML)**, or not. This often depends on the business context in which you operate and what requirements are placed on the people writing the architecture. The key is that it checks off the points above and gives the engineers clear guidance on what to build and how it will all stick together.

Architecture is a vast and fascinating subject in itself, so we will not go much further into the details of this here, but we will now focus on what architecture means in an ML engineering context.

Building with principles

The field of architecture is vast but no matter where you look, like any mature discipline, there are always consistent principles that are presented. The good news is that some of them are actually the same as some of the principles we met when discussing good Python programming in *Chapter 4, Packaging Up*. In this section, we will discuss some of these and how they can be used for architecting ML systems.

Separation of Concerns has already been mentioned in this book as a good way to ensure that software components inside your applications are not

unnecessarily complex and that your solutions are extensible and can be easily interfaced with. This principle holds true of systems in their entirety and as such is a good architecture principle to bear in mind. In practice, this often manifests in the idea of separate “layers” within your applications that have distinct responsibilities. For example, let’s look at the architecture shown in *Figure 5.1*. This shows how to use tools to create an automated deployment and orchestration process for ML pipelines and is taken from the AWS Solutions Library,

<https://aws.amazon.com/solutions/implementations/ml-ops-workload-orchestrator/>. We can see that there are distinct “areas” within the architecture corresponding to provisioning, pipeline deployment, and pipeline serving. These blocks show that there are distinct pieces of the solution that have specific functionalities and that the interaction between these different pieces is handled by an interface.

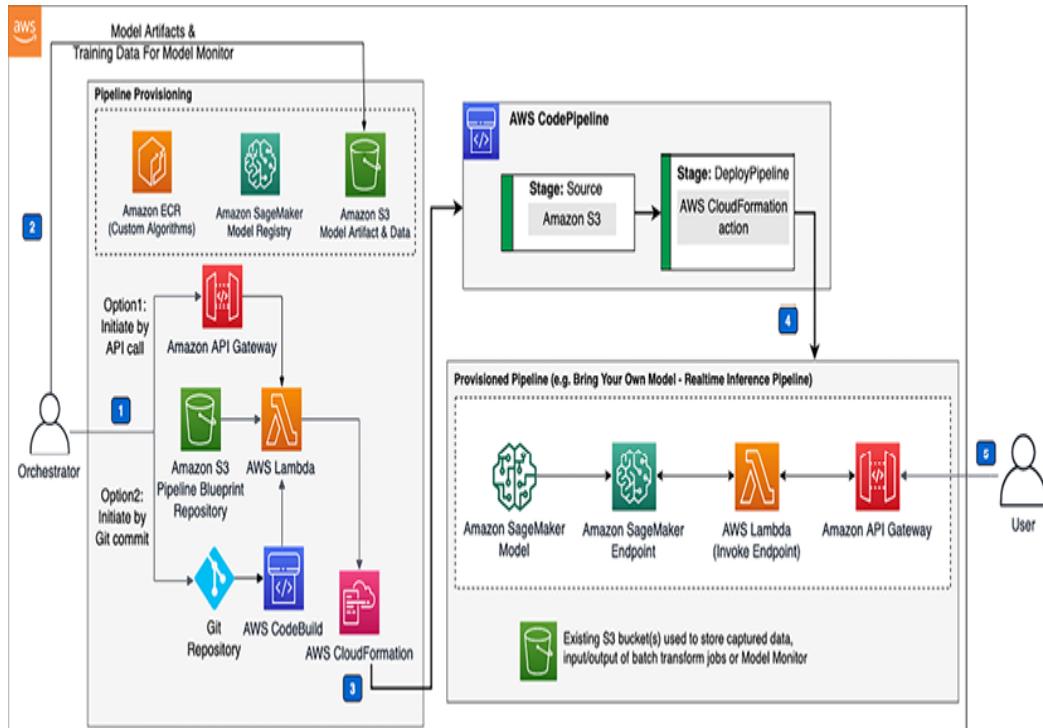


Figure 5.1: An ML workload orchestrator architecture from the AWS Solutions Library MLOps Workload Orchestrator.

The **Principle of Least Surprise** is a rule of thumb that essentially captures the fact that the first time any reasonably knowledgeable person in your domain, such as a developer, tester, or data scientist, encounters your architecture, it should not have anything within it that should stand out as unorthodox or surprising. This may not always be possible, but it is a good principle to keep in mind as it forces you to consider what those who are likely to be working with your architecture already know and how you can leverage that to both make a good design and have it followed. Using *Figure 5.1* as an example again, the architecture embodies the principle very nicely, as the design has clear logic building blocks for provisioning, promoting, and running the ML pipelines. At a lower level in the architecture, we can see that data is consistently being sourced from S3 buckets, that Lambdas are interacting with API gateways, and so on and so forth. This means that ML engineers, data scientists, and cloud platform engineers will both understand and leverage this architecture well when implementing it.

The **Principle of Least Effort** is a bit more subtle than the previous one, in that it captures the idea that developers, being human, will follow the path of least resistance and not create more work unless necessary. I interpret this principle as emphasizing the importance of taking the time to consider your architecture thoughtfully and building it with care, as it could be used for a long time after it has been developed, by engineer after engineer!

So far, we have only discussed high-level architecture principles. Now we will look at some design principles that – while they can still be used at the system design level – are also very powerful when used at the level of your code.

The **SOLID** principles (**Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion**) are a set that is often applied to the code base but can also be extrapolated up to system design and architecture quite nicely. Once we adapt these principles to the architecture level, they can be explained in the following way:

- **Single Responsibility:** This is very similar, perhaps identical, to the idea of separation of concerns. Specifically, this states that if a module only has one reason to change at any one time, or it only has one job to do, then this makes it more resilient and easier to maintain. If you have one box in your architecture diagram that is going to have to do ten different things, then you have violated this principle and it means that whenever any one of those processes or interfaces has to change, you have to go into that box and poke around, potentially creating more issues or drastically increasing the likelihood of downtime.
- **Open/Closed:** This refers to the fact that it is a really good idea to architect in a way that components are “open for extension but closed for modification.” This also works at the level of the entire design. If you design your system so that new functionality can be tagged on and does not require going back and rewiring the core, then you will likely build something that will stand the test of time. A great example from ML would be that if we try and build our system so that if we want to add in new processing pipelines we can just do that, and we don’t have to go back into some obscure section of the code and severely modify things.
- **Liskov Substitution:** When the SOLID principles were written, they originally referred to object-oriented programming in languages like Java. This principle then stated that objects should be able to be replaced by their subtypes and still maintain application behavior. At

the system level, this now basically states that if two components are supposed to have the same interface and contract with other components, you can swap them for one another.

- **Interface Segregation:** I interpret this one as “don’t have multiple ways for components to talk to one another.” So, in your application, try and ensure that the ways of handing off between different pieces of the solution are pretty narrow. Another way of phrasing this is that making your interfaces as client specific as possible is a good idea.
- **Dependency Inversion:** This is very similar to the Liskov Substitution principle but is a bit more general. The idea here is that the communications between modules or parts of your solution should be taken care of by abstractions and not by a concrete, specific implementation. A good example would be that instead of calling an ML microservice directly from another process, you instead place the requisite job data in a queue, for example, AWS Simple Queue Service, and then the microservice picks up the work from the queue. This ensures that the client and the serving microservice do not need to know details about each other’s interface, and also that the downstream application can be extended with more services reading from the queue. This would then also embody the Open/Closed principle, and can be seen in the architecture in *Figure 5.1* through the use of the Lambda function calling into AWS CloudFormation.

A final favorite of mine is the concept of **Bounded Contexts**, where we have to seek to ensure that data models, or other important data or metadata, are aligned within specific conceptual models and are not a “free-for-all.” This applies particularly well to Domain-Driven Design and applies very well to large, complex solutions. A great example would be if you have a large organization with multiple business units and they want a series of

very similar services that run ML on business data stored in a database. It would be better for there to be several databases hosting the information, one for each business unit, rather than having a shared data layer across multiple applications. More concretely, your data model shouldn't contain information specific to the sales and marketing function and the engineering function and the human resources function, and so on. Instead, each should have their own database with their own models, and there should be explicit contracts for joining any information between them later if needed. I believe that this idea can still be applied to Data Lakes, which are discussed later in this chapter. In this case, the bounded contexts could apply to specific folders within the lake, or they could actually refer to the context of entire lakes, each segregated into different domains. This is very much the idea behind the so-called Data Mesh.

We have just mentioned some of the most used ML patterns, so let's now move on to explore this concept in a bit more detail as we look to apply the principles we have been discussing.

Exploring some standard ML patterns

In this book, we have already mentioned a few times that we should not attempt to *reinvent* the wheel and we should reuse, repeat, and recycle what works according to the wider software and ML community. This is also true about your deployment architectures.

When we discuss architectures that can be reused for a variety of different use cases with similar characteristics, we often refer to these as *patterns*. Using standard (or at least well-known) patterns can really help you speed

up the time to value of your project and help you engineer your ML solution in a way that is robust and extensible.

Given this, we will spend the next few sections summarizing some of the most important architectural patterns that have become increasingly successful in the ML space over the past few years.

Swimming in data lakes

The single most important asset for anyone trying to use ML is, of course, the data that we can analyze and train our models on. The era of **big data** meant that the sheer size and variability in the format of this data became an increasing challenge. If you are a large organization (or even not so large), it is not viable to store all of the data you will want to use for ML applications in a structured relational database. Just the complexity of modeling the data for storage in such a format would be very high. So, what can you do?

Well, this problem was initially tackled with the introduction of **data warehouses**, which let you bring all of your relational data storage into one solution and create a single point of access. This helps alleviate, to some extent, the problem of data volumes, as each database can store relatively small amounts of data even if the total is large. These warehouses were designed with the integration of multiple data sources in mind. However, they are still relatively restrictive as they usually bundle together the infrastructure for compute and storage. This means they can't be scaled very well, and they can be expensive investments that create vendor lock-in. Most importantly for ML, data warehouses cannot store raw and semi-structured or unstructured data (for example, images). This automatically rules out a lot of good ML use cases if warehouses are used as your main

data store. Now, with tools such as **Apache Spark**, which we've already used extensively throughout this book, if we have the clusters available, we can feasibly analyze and model any size or structure of data. The question then becomes, how should we store it?

Data lakes are technologies that allow you to store any type of data at any scale you feasibly need. There are a variety of providers of data lake solutions, including the main public cloud providers, such as **Microsoft Azure**, **Google Cloud Platform (GCP)**, and AWS. Since we have met AWS before, let's focus on that.

The main storage solution in AWS is called the **Simple Storage Service**, or **S3**. Like all of the core data lake technologies, you can effectively load anything into it since it is based on the concept of *object storage*. This means that every instance of data you load is treated as its own object with a unique identifier and associated metadata.

It allows your S3 bucket to simultaneously contain photographs, JSON files, `.txt` files, Parquet files, and any other number of data formats.

If you work in an organization that does not have a data lake, this does not automatically exclude you from doing ML, but it can definitely make it a more difficult journey since with a lake you always know how you can store the data you need for your problem, no matter the format.

Microservices

Your ML project's code base will start small – just a few lines at first. But as your team expends more and more effort in building the solution required, this will quickly grow. If your solution has to have a few different capabilities and perform some quite distinct actions and you keep all of this

in the same code base, your solution can become incredibly complex. In fact, software in which the components are all tightly coupled and non-separable like this is called **monolithic**, as it is akin to single big blocks that can exist independently of other applications. This sort of approach may fit the bill for your use case, but as the complexity of solutions continues to increase, a much more resilient and extensible design pattern is often required.

Microservice architectures are those in which the functional components of your solution are cleanly separated, potentially in completely different code bases or running on completely different infrastructure. For example, if we are building a user-facing web application that allows users to browse, select, and purchase products, we may have a variety of ML capabilities we wish to deploy in quick succession. We may want to recommend new products based on what they have just been looking at, we may want to retrieve forecasts of when their recently ordered items will arrive, and we may want to highlight some discounts we think they will benefit from (based on our analysis of their historic account behavior). This would be a very tall order, maybe even impossible, for a monolithic application. However, it is something that quite naturally falls into microservice architecture like that in *Figure 5.2*:

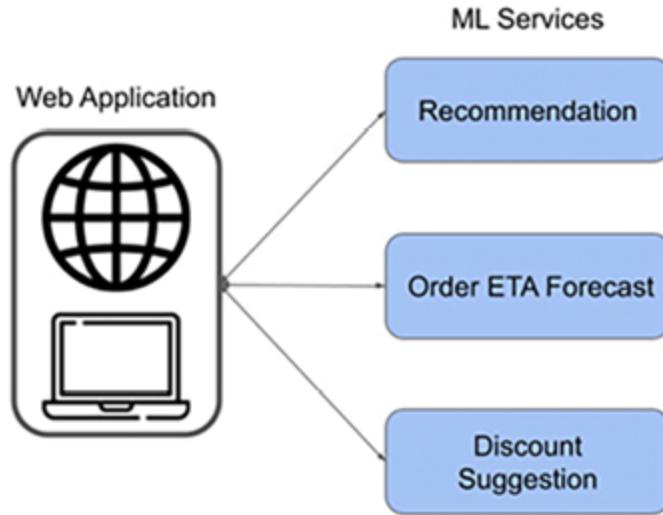


Figure 5.2: An example of some ML microservices.

The implementation of a microservice architecture can be accomplished using a few tools, some of which we will cover in the *Hosting your own microservice on AWS* section. The main idea is that you always separate out the elements of your solution into their own services that are not tightly coupled together.

Microservice architectures are particularly good at allowing our development teams to achieve the following:

- Independently debug, patch, or deploy individual services rather than tearing down the whole system.
- Avoid a single point of failure.
- Increase maintainability.
- Allow separate services to be owned by distinct teams with clearer responsibilities.
- Accelerate the development of complex products.

Like every architecture pattern or design style, it is, of course, not a silver bullet, but we would do well to remember the microservice architecture

when designing our next solution.

Next, we will discuss event-based designs.

Event-based designs

You do not always want to operate in scheduled batches. As we have seen, even just in the previous section, *Microservices*, not all use cases align with running a large batch prediction from a model on a set schedule, storing the results, and then retrieving them later. What happens if the data volumes you need are not there for a training run? What if no new data to run predictions on has arrived? What if other systems could make use of a prediction based on individual data points at the earliest time they become available rather than at a specific time every day?

In an event-based architecture, individual actions produce results that then trigger other individual actions in the system, and so on and so forth. This means that processes can happen as early as they can and no earlier. It also allows for a more dynamic or stochastic data flow, which can be beneficial if other systems are not running on scheduled batches either.

Event-based patterns could be mixed with others, for example, microservices or batch processing. The benefits still stand, and, in fact, event-based components allow for more sophisticated orchestration and management of your solution.

There are two types of event-based patterns:

- **Pub/sub:** In this case, event data is published to a message broker or event bus to be consumed by other applications. In one variant of the pub/sub pattern, the broker or buses used are organized by some

appropriate classification and are designated as **topics**. An example of a tool that does this is **Apache Kafka**.

- **Event streaming:** Streaming use cases are ones where we want to process a continuous flow of data in something very close to real time. We can think of this as working with data as it *moves through* the system. This means it is not persisted *at rest* in a database but processed as it is created or received by the streaming solution. An example tool to use for event streaming applications is **Apache Storm**.

Figure 5.3 shows an example event-based architecture applied to the case of **IoT** and mobile devices that have their data passed into classification and anomaly detection algorithms:

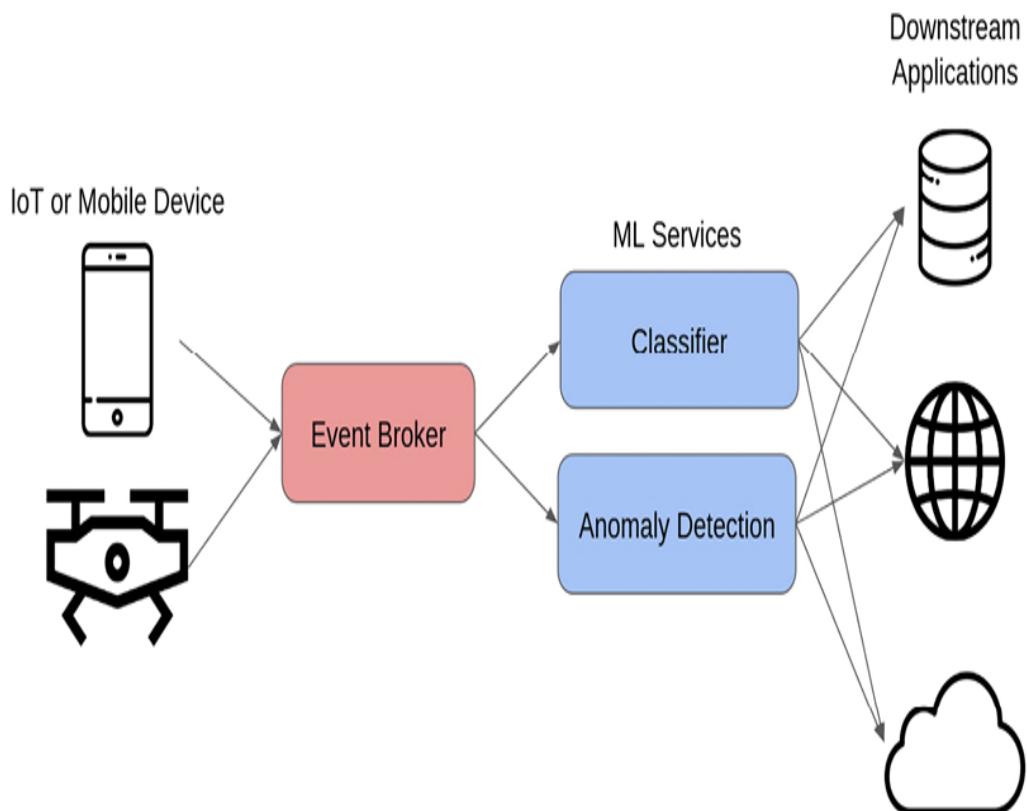


Figure 5.3: A basic event-based high-level design where a stream of data is accessed by different services via a broker.

The next section will touch on designs where we do the opposite of processing one data point at a time and instead work with large chunks or batches at any one time.

Batching

Batches of work may not sound like the most sophisticated concept, but it is one of the most common pattern flavors out there in the world of ML.

If the data you require for prediction comes in at regular time intervals in batches, it can be efficient to schedule your prediction runs with a similar cadence. This type of pattern can also be useful if you do not have to create a low-latency solution.

This concept can also be made to run quite efficiently for a few reasons:

- Running in scheduled batches means that we know exactly when we will need compute resources, so we can plan accordingly. For example, we may be able to shut down our clusters for most of the day or repurpose them for other activities.
- Batches allow for the use of larger numbers of data points at runtime, so you can run things such as anomaly detection or clustering at the batch level if desired.
- The size of your batches of data can often be chosen to optimize some criterion. For example, using large batches and running parallelized logic and algorithms on it could be more efficient.

Software solutions where ML algorithms are run in batches often look very similar to classic **Extract, Transform, Load (ETL)** systems. These are systems where data is extracted from a source or sources, before being processed en route to a target system where it is then uploaded. In the case

of an ML solution, the processing is not standard data transformation such as joins and filters but is instead the application of feature engineering and ML algorithm pipelines. This is why, in this book, we will term these designs **Extract, Transform, Machine Learning (ETML)** patterns. ETML will be discussed more in *Chapter 9, Building an Extract, Transform, Machine Learning Use Case*.

We will now discuss a key piece of technology that is critical to making modern architectures applicable to a wide range of platforms – containers.

Containerizing

If you develop software that you want to deploy somewhere, which is the core aim of an ML engineer, then you have to be very aware of the environmental requirements of your code, and how different environments might affect the ability of your solution to run. This is particularly important for Python, which does not have a core capability for exporting programs as standalone executables (although there are options for doing this). This means that Python code needs a Python interpreter to run and needs to exist in a general Python environment where the relevant libraries and supporting packages have been installed.

A great way to avoid headaches from this point of view is to ask the question: *Why can't I just put everything I need into something that is relatively isolated from the host environment, which I can ship and then run as a standalone application or program?* The answer to this question is that you can and that you do this through **containerization**. This is a process whereby an application and its dependencies can be packaged together in a standalone unit that can effectively run on any computing platform.

The most popular container technology is **Docker**, which is open-source and very easy to use. Let's learn about it by using it to containerize a simple **Flask** web application that could act as an interface to a forecasting model like that created in the *Example 2: Forecasting API* section in *Chapter 1, Introduction to ML Engineering*.

The next few sections will use a similar simple Flask application that has a forecast serving endpoint. As a proxy for a full ML model, we will first work with a skeleton application that simply returns a short list of random numbers when requested for a forecast. The detailed code for the application can be found in this book's GitHub repo at

<https://github.com/PacktPublishing/Machine-Learning-Engineering-with-Python-Second-Edition/tree/main/Chapter05/microservices/mlewp2-web-service>.

The web application creates a basic app where you can supply a store ID and forecast a start date for the system and it will return the dummy forecast. To get this, you hit the `/forecast` endpoint.

An example is shown in *Figure 5.4*:

HTTP mlewp2 / New Request

POST http://127.0.0.1:8080/forecast

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL **JSON**

```
1 {
2   "store_number": 10,
3   "forecast_start_date": "2023-07-01T00:00:00"
4 }
```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize **JSON**

```
1 [
2   "result": [
3     0.6043513740449178,
4     0.039012484283529325,
5     0.4893184826596272,
6     0.31851042354330683,
7     0.09421735231755868,
8     0.7291520357848239,
9     0.22508165205615693,
10    0.07317010122711154,
11    0.7551697651805447,
12    0.11153561883299301
13  ],
14  "store_number": 10
15 ]
```

Figure 5.4: The result of querying our skeleton ML microservice.

Now, we'll move on to discuss how to containerize this application. First, you need to install Docker on your platform by using the documentation at <https://docs.docker.com/engine/install/>:

1. Once you have Docker installed, you need to tell it how to build the container image, which you do by creating a `Dockerfile` in your project. The `Dockerfile` specifies all of the build steps in text so that the process of building the image is automated and easily configurable. We will now walk through building a simple example `Dockerfile`, which will be built on in the next section, *Hosting your own microservice on AWS*. First, we need to specify the base image we are working from. It usually makes sense to use one of the official Docker images as a base, so here we will use the `python:3.10-slim` environment to keep things lean and mean. This base image will be used in all commands following the `FROM` keyword, which signifies we are entering a build stage. We can actually name this stage for later use, calling it `builder` using the `FROM ... as` syntax:

```
FROM python:3.10-slim as builder
```

2. Then, we copy all the files we need from the current directory to a directory labeled `src` in the build stage and install all of our requirements using our `requirements.txt` file (if you want to run this step without specifying any requirements, you can just use an empty `requirements.txt` file):

```
COPY . /src  
RUN pip install --user --no-cache-dir -r requirements.txt
```

3. The next stage involves similar steps but is aliased to the word `app` since we are now creating our application. Notice the reference to the `builder` stage from steps 1 and 2 here:

```
FROM python:3.10-slim as app  
COPY --from=builder /root/.local /root/.local  
COPY --from=builder /src .
```

4. We can define or add to environment variables as we are used to in a bash environment:

```
ENV PATH=/root/.local:$PATH
```

5. Since in this example we are going to be running a simple Flask web application, we need to tell the system which port to expose:

```
EXPOSE 5000
```

6. We can execute commands during the Docker build using the `CMD` keyword. Here, we use this to run `app.py`, which is the main entry point to the Flask app, and will start the service we will call via REST API to get ML results later:

```
CMD ["python3", "app.py"]
```

7. Then we can build the image with the `docker build` command. Here, we create an image named `basic-ml-microservice` and tag it with the `latest` label:

```
docker build -t basic-ml-microservice:latest
```

8. To check the build was successful, run the following command in the Terminal:

```
docker images --format "table {{.ID}}\t{{.Cr
```

You should see an output like that in *Figure 5.5*:

IMAGE ID	CREATED AT	REPOSITORY
b4cff53f44287	2023-07-02 22:17:37 +0100 BST	basic-ml-webservice
dad5d3d3ced3	2023-06-29 13:13:54 +0100 BST	inferencefunction
25b59787560e	2023-06-29 12:10:49 +0100 BST	508972911348.dkr.ecr.eu-west-2.amazonaws.com/mlewpsamm
25b59787560e	2023-06-29 12:10:49 +0100 BST	inferencefunction
16cd75dd36ad	2023-06-29 10:20:07 +0100 BST	<none>
ab560c318c87	2023-06-28 14:26:40 +0100 BST	public.ecr.aws/lambda/python
5e71c7186e71	2023-04-10 15:00:27 +0100 BST	electricweegie/custom-kserve-endpoint
5e71c7186e71	2023-04-10 15:00:27 +0100 BST	custom-kserve-endpoint
1465debfd6f	2023-04-03 22:47:45 +0100 BST	gcr.io/k8s-minikube/kicbase
b4faebd75645	2023-03-22 22:17:40 +0000 GMT	vsc-anaconda-1c27a4749abc4e72305ca51f3243da64
b2e19fd3d045	2023-03-22 22:13:08 +0000 GMT	vsc-volume-bootstrap
f60702d4ce22	2023-02-08 22:31:07 +0000 GMT	electricweegie/docker101tutorial
f60702d4ce22	2023-02-08 22:31:07 +0000 GMT	docker101tutorial
9793ee61fc75	2022-11-20 05:16:44 +0000 GMT	alpine/git
2b4661558fb8	2022-11-12 03:39:38 +0000 GMT	alpine
476b7007f4f5	2022-10-25 20:41:01 +0100 BST	kindest/node
df953f7b867a	2020-02-07 01:06:27 +0000 GMT	kindest/node

Figure 5.5: Output from the docker images command.

9. Finally, you can run your Docker image with the following command in your Terminal:

```
docker run --rm -it -p 8080:5000 basic-ml-mi
```

Now that you have containerized some basic applications and can run your Docker image, we need to answer the question of how we can use this to build an ML solution hosted on an appropriate platform. The next section covers how we can do this on AWS.

Hosting your own microservice on AWS

A classic way to surface your ML models is via a lightweight web service hosted on a server. This can be a very flexible pattern of deployment.

You can run a web service on any server with access to the internet (roughly) and, if designed well, it is often easy to add further functionality to your web service and expose it via new endpoints.

In Python, the two most used web frameworks have always been **Django** and **Flask**. In this section, we will focus on Flask as it is the simpler of the two and has been written about extensively for ML deployments on the web, so you will be able to find plenty of material to build on what you learn here.

On AWS, one of the simplest ways you can host your Flask web solution is as a containerized application on an appropriate platform. We will go through the basics of doing this here, but we will not spend time on the detailed aspects of maintaining good web security for your service. To fully discuss this may require an entire book in itself, and there are excellent, more focused resources elsewhere.

We will assume that you have your AWS account set up from *Chapter 2, The Machine Learning Development Process*. If you do not, then go back and refresh yourself on what you need to do.

We will need the AWS **Command Line Interface (CLI)**. You can find the appropriate commands for installing and configuring the AWS CLI, as well as a lot of other useful information, on the AWS CLI documentation pages at <https://docs.aws.amazon.com/cli/index.html>.

Specifically, configure your Amazon CLI by following the steps in this tutorial:

<https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-quickstart.html>.

The documentation specifies how to install the CLI for a variety of different computer architectures, so follow along for your given platform and then you will be ready to have fun with the AWS examples used in the rest of the book!

In the following example, we will use Amazon **Elastic Container Registry (ECR)** and **Elastic Container Service (ECS)** to host a skeleton containerized web application. In *Chapter 8, Building an Example ML Microservice*, we will discuss how to build and scale an ML microservice in more detail and using a lower-level implementation based on Kubernetes. These two approaches complement each other nicely and will help you widen your ML engineering toolkit.

Deploying our service on ECS will require a few different components, which we will walk through in the next few sections:

- Our container hosted inside a repository on ECR
- A cluster and service created on ECS

- An application load balancer created via the **Elastic Compute Cloud (EC2)** service

First, let's tackle pushing the container to ECR.

Pushing to ECR

Let's look at the following steps:

1. We have the following Dockerfile defined within the project directory from the *Containerizing* section:

```
FROM python:3.10-slim as builder
COPY . /src
RUN pip install --user --no-cache-dir -r requirements.txt
FROM python:3.10-slim as app
COPY --from=builder /root/.local /root/.local
COPY --from=builder /src .
ENV PATH=/root/.local:$PATH
EXPOSE 5000
CMD ["python3", "app.py"]
```

2. We can then use the AWS **CLI** to create an ECR repository for hosting our container. We will call the repository **basic-ml-microservice** and will set the region as **eu-west-1**, but this should be changed to what region seems most appropriate for your account. The command below will return some metadata about your ECR repository; keep this for later steps:

```
aws ecr create-repository  
  --repository-name basic-ml-microservice  
  --image-scanning-configuration scanOnPush  
  --region eu-west-1
```

3. We can then log in to the container registry with the following command in the Terminal. Note that the repository URI will have been in the metadata provided after running step 2. You can also retrieve this by running `aws ecr describe-repositories --region eu-west-1`:

```
aws ecr get-login-password --region eu-west-
```

4. Then, if we navigate to the directory containing the `Dockerfile` (`app`), we can run the following command to build the container:

```
docker build --tag basic-ml-microservice:loc
```

5. The next step tags the image:

```
docker tag basic-ml-microservice:local <ECR_
```

6. We then deploy the Docker image we have just built to the container registry with the following command:

```
docker push <YOUR_AWS_ID>.dkr.ecr.eu-west-1.
```

If successful, this last command will have pushed the locally built Docker image to your remotely hosted ECR repository. You can confirm this by navigating to the AWS management console, going to the ECR service, and selecting the basic-ml-microservice repository. You should then see something like what is shown in *Figure 5.6*.

The screenshot shows the AWS ECR interface. At the top, it says 'Amazon ECR > Repositories > basic-ml-microservice'. Below that, the repository name 'basic-ml-microservice' is displayed with a 'View' button. Underneath, there's a section titled 'Images (1)' with a 'Find images' search bar. A table lists the single image: 'latest' (Image type), pushed at '02 July 2023, 23:23:29 (UTC+01)', size '70.97 MB', and digest 'sha256:f1e5a76b1e1e7df...'. There are 'Copy URI' and 'Delete' buttons for this entry.

Image tag	Artifact type	Pushed at	Size (MB)	Image URI	Digest	Scan status
latest	Image	02 July 2023, 23:23:29 (UTC+01)	70.97	Copy URI	sha256:f1e5a76b1e1e7df...	Complete

Figure 5.6: Successful push of the locally built Docker image to the ECR repository.

The steps we have just gone through are actually quite powerful in general, as you are now able to build cross-platform Docker images and share them in a central repository under your AWS account. You can share Docker containers and images via DockerHub as well, <https://hub.docker.com/>, but this gives you more control if you want to do this inside your own organization.

Now that we have built the container that hosts the Flask app, we will now look to deploy this on scalable infrastructure. To do this, in the next section, we will set up our cluster on ECS.

Hosting on ECS

Now, let's start with the setup! At the time of writing in mid-2023, AWS has recently introduced a revamped ECS console that allows for a far smoother setup than previously. So, if you read the first edition of this book, you will find this a far smoother experience:

1. First, navigate to **ECS** on the AWS Management Console and click **Create Cluster**. You will be provided with a form that asks for details about networking, infrastructure, monitoring, and the provision of any tags on the resources we are about to create. This should look like *Figure 5.7.*

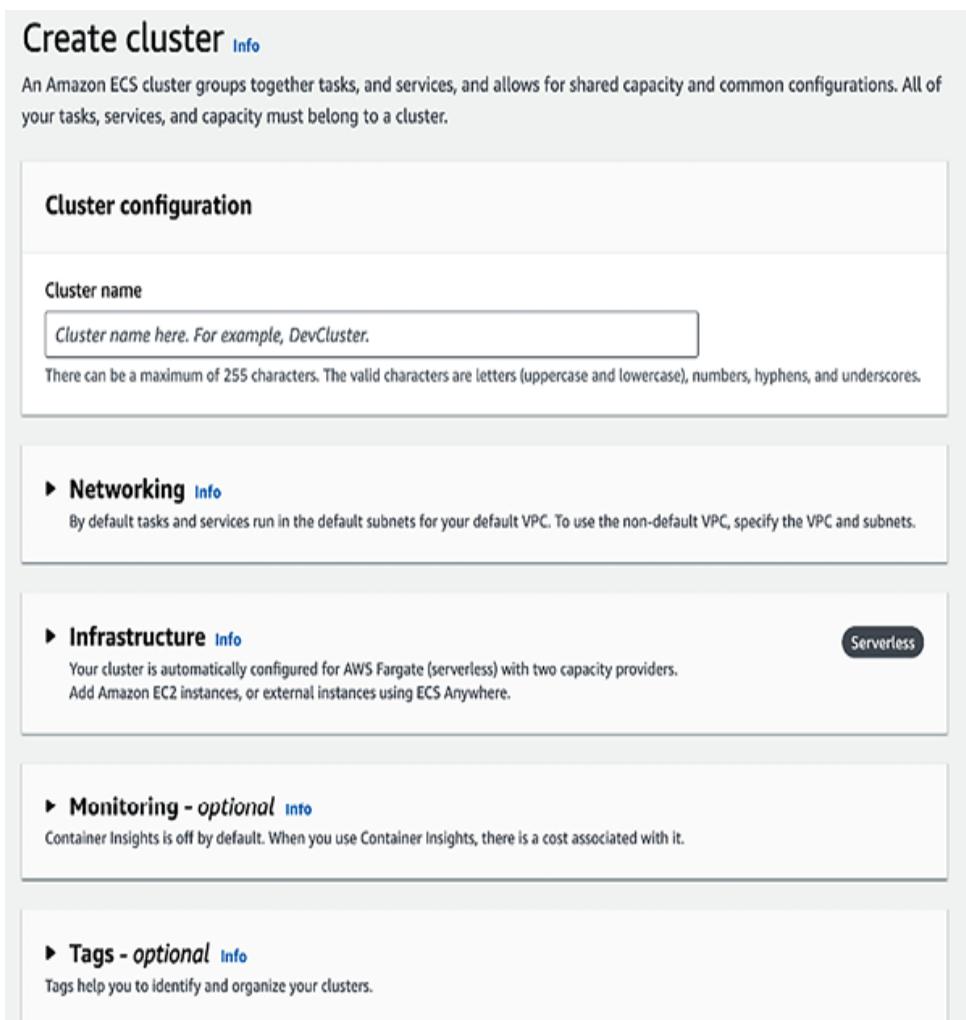


Figure 5.7: Creating a cluster in Elastic Container Service.

2. First, we can name the cluster `mlewp2-ecs-cluster`, or indeed whatever you want! Then when you expand the **Networking** section, you should see that many of the **VPC** and subnet details are auto-populated with defaults based on your AWS account setup. If you need to set these up, the form points to the relevant documentation. See *Figure 5.8* for an example.

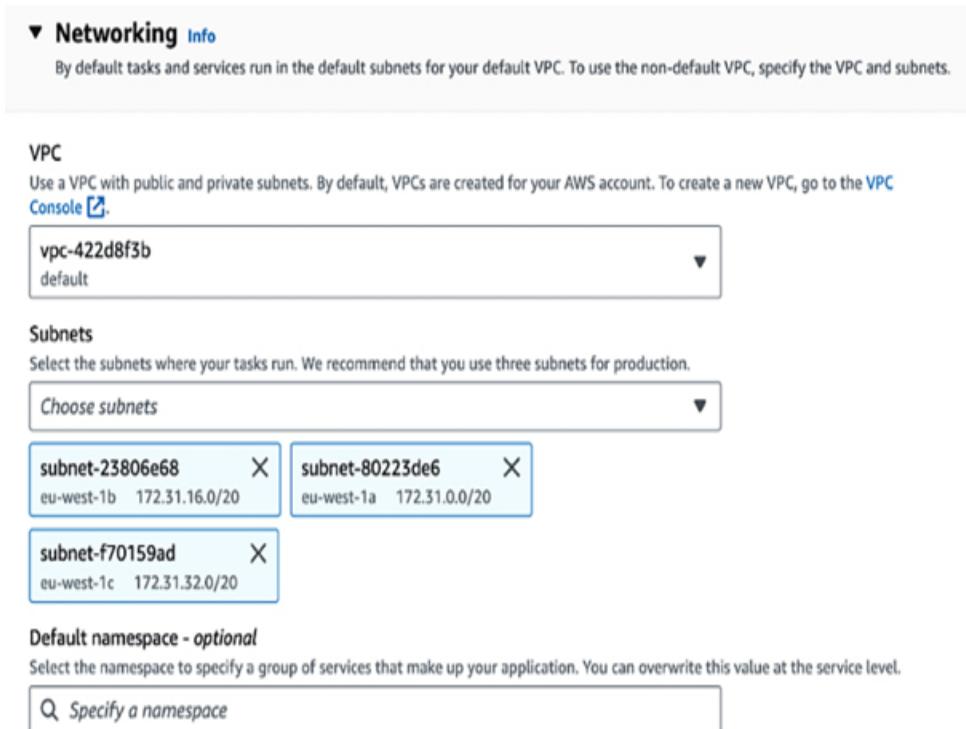


Figure 5.8: Networking configuration for our cluster in AWS ECS.

3. The **Infrastructure** section contains three options, with the use of **AWS Fargate** being the pre-selected default option. We do not need to know the details of how Fargate works but suffice it to say that this provides a very high-level abstraction for managing container workloads across multiple servers. The introduction of Fargate has meant that you do not need to worry about details of the provisioning and running of clusters of virtual machines to run your container

workloads. According to the AWS documentation, the Fargate service is ideal for dynamic bursts of work or large workloads with low operational overhead. If you know you are going to be running large jobs that have to be price optimized, you can then look to the other infra options provided, for example, **EC2 instances**. We will not need these for the purposes of this example. *Figure 5.9* shows the **Infrastructure** section for reference.

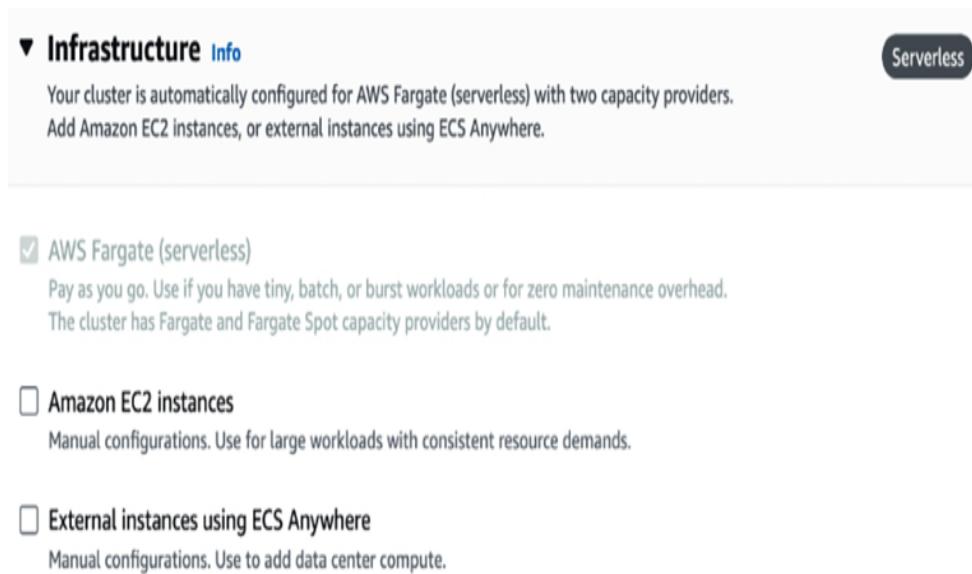


Figure 5.9: Configuring the infrastructure options in the ECS service.

4. The **Monitoring** and **Tags** sections are relatively self-explanatory and allow you to toggle on **container insights** and provide your own string tags for the ECS resources that will be created. Let's leave these as the default for now and click **Create** at the bottom of the page. You should then see that the cluster was successfully created after a few minutes, as shown in *Figure 5.10*.

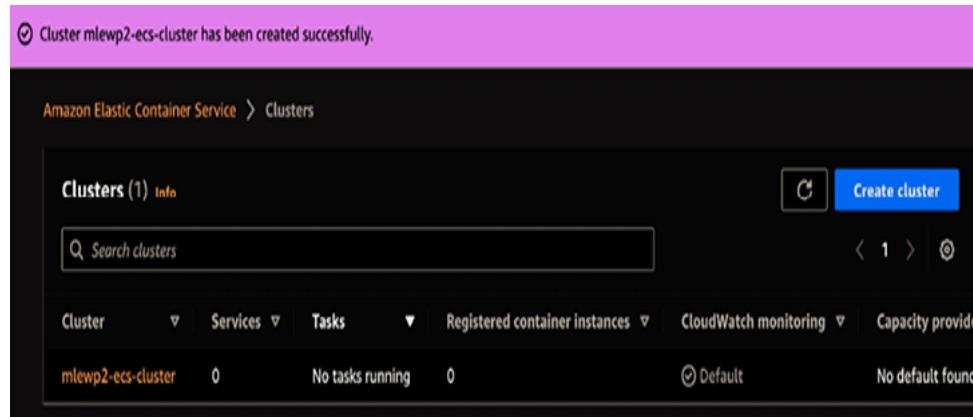


Figure 5.10: The successful creation of the ECS cluster.

The previous steps were all about setting up the ECS cluster, the infrastructure on which our containerized application can run. To actually tell ECS how to run the solution, we need to define **tasks**, which are simply processes we wish to be executed on the cluster. There is a related concept of **Services** in ECS, which refers to a process for managing your tasks, for example, by ensuring a certain number of tasks are always running on the cluster. This is useful if you have certain uptime requirements for your solution, such as, if it needs to be available for requests 24/7. We can create the task definition in the cluster by first navigating to the cluster review page in the AWS management console, and then selecting **Task Definitions** on the left-hand side. We will then click on **Create New Task Definition**. Follow the steps below to create this task definition.

1. We have to name the task definition family, which is just the collection of versions of the task definition. Let's call ours **basic-ml-microservice-tasks** for simplicity. We then need to provide some container details such as the URI for the image we want to use. This is the URI for the image we pushed to the ECR repository previously, which is formatted something like **<YOUR_AWS_ID>.dkr.ecr.eu-west-**

`1.amazonaws.com/basic-ml-microservice:latest`

You can give the container a new name. Here, I have called it **mlmicro**. Finally, you need to supply appropriate port mappings to allow the container and the application it contains to be accessible to external traffic. I have mapped `port 5000`, which you may recall is the port we exposed in the original Dockerfile using the TCP protocol. This is all shown in *Figure 5.11*. You can leave the rest of the optional settings for this first section as the default just now and click **Next** to move on to the next page of settings.

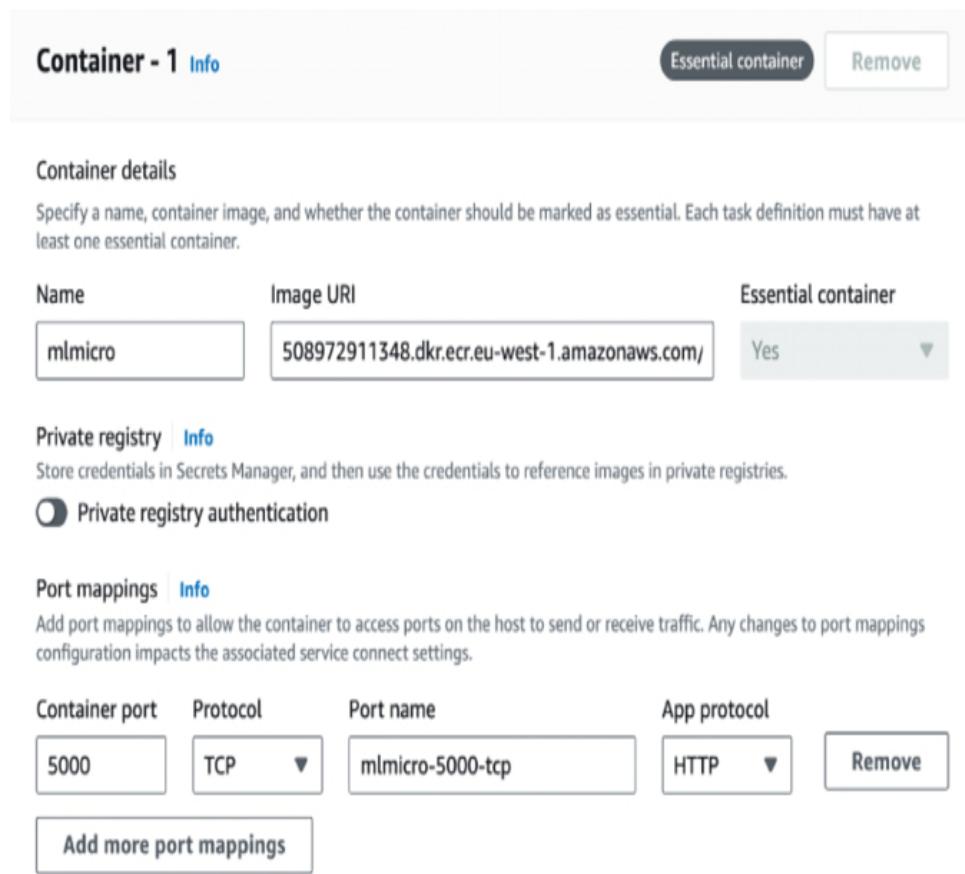


Figure 5.11: Defining the container image to use for the task definition in ECS.

2. The next page in the console asks for information about the environment and infrastructure you will be running the solution on. Based on the settings we used for the ECS cluster, we will be using

Fargate as the infrastructure option, running on a **Linux x86_64** environment. The tasks we are running are very small in this case (we're just returning some numbers for demo purposes) so we can keep the default options of **1 vCPU** with **3 GB** memory. You can also add container-level memory and CPU requirements if necessary, but we can leave this blank for now. This is particularly useful if you have a computationally heavy service, or it contains an application that is pre-loaded with some large model or configuration data. You can see this in *Figure 5.12*.

▼ Environment

Specify the infrastructure requirements for the task definition.

App environment [Info](#)

Specify the infrastructure for the task definition.

Add an option ▾

AWS Fargate (serverless) X

Operating system/Architecture [Info](#)

Linux/X86_64 ▾

Task size [Info](#)

Specify the amount of CPU and memory to reserve for your task.

CPU

Memory

1 vCPU ▾

3 GB ▾

▼ Container size - optional [Info](#)

For each container associated with the task, specify the container-level CPU and memory.

Add

Figure 5.12: Configuring our application environment for the AWS ECS task definition used for our ML microservice.

3. Next, IAM roles need to be configured. We will not be calling other AWS services from our application, so at this point, we do not need an IAM task role, but you can create one if you need this at a later point,

for example, if you wish to call another data or ML service. For executing the tasks we need an execution role, which by default is created for you, so let's use that. The IAM configuration section is shown in *Figure 5.13*.

▼ Task roles, network mode - conditional

Task role | [Info](#)

A task IAM role allows containers in the task to make API requests to AWS services. You can create a task IAM role from the [IAM console](#).



Task execution role | [Info](#)

A task execution IAM role is used by the container agent to make AWS API requests on your behalf. If you don't already have a task execution IAM role created, we can create one for you.



Network mode | [Info](#)

The network mode that's used for your tasks. When the AWS Fargate (serverless) launch type is selected, you must use the awsvpc network mode. If you select the Amazon EC2 instance launch type, you can use different network modes in Linux or Windows. On Linux, you can choose between bridge, awsvpc, host, or none. On Windows, you can choose between default or awsvpc.



Figure 5.13: The IAM roles defined for use by the AWS ECS task definition.

4. The rest of this section contains optional sections for storage, monitoring, and tagging. The storage subsection refers to ephemeral storage used to decompress and host your Docker container. Again, for larger containers, you may need to consider increasing this size from the default 21 GiB. Monitoring can be enabled using **Amazon**

CloudWatch, which is useful when you need infrastructure monitoring as part of your solution, but we will not cover that here and focus more on the core deployment. Keep these sections as is for now and click **Next** at the bottom of the page.

5. We are almost there. Now we'll review and create the task definition. If you are happy with the selections upon reviewing, then create the task definition and you will be taken to a summary page like that shown in *Figure 5.14*.

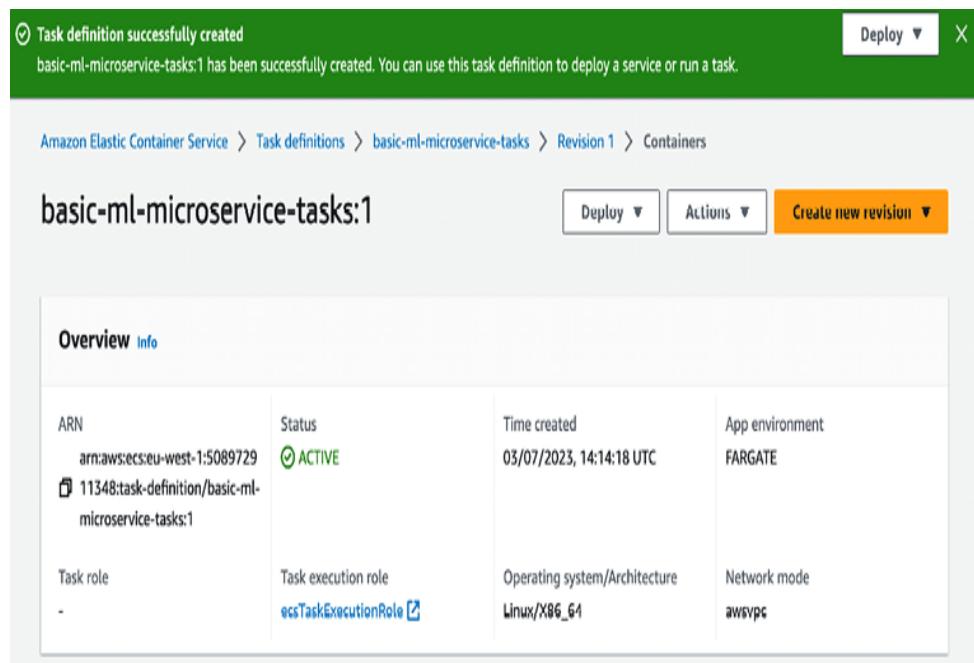


Figure 5.14: Successful creation of the ML microservice task definition.

Now, the final step of setting up our ECS-hosted solution is the creation of a service. We will now walk through how to do this:

1. First, navigate to the task definition we have just created in the previous steps and select the **Deploy** button. This will provide a dropdown where you can select **Create service**. *Figure 5.15* shows you what this looks like as it may be easy to miss.

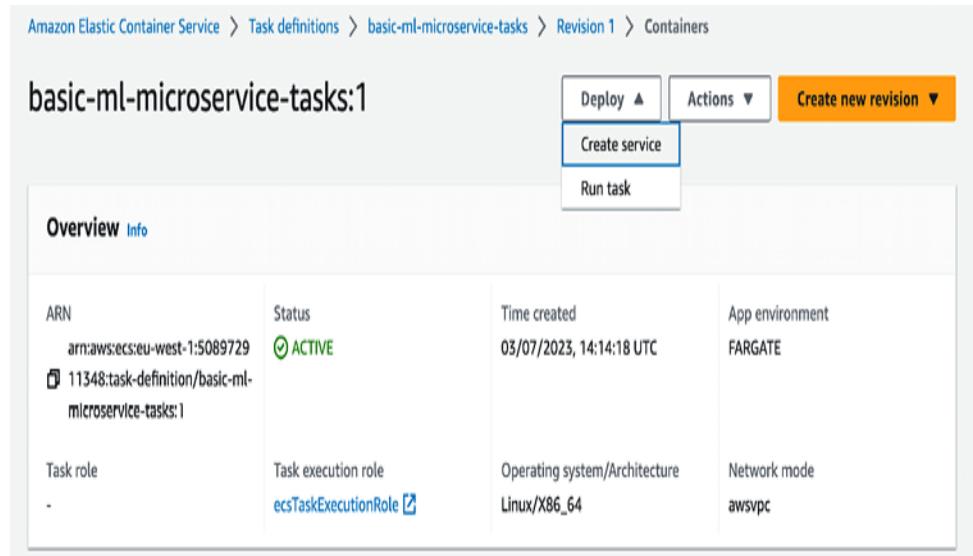


Figure 5.15: Selecting the *Create service* option for the task definition we have just created in the previous steps.

2. You will then be taken to another page where we need to fill in the details of the service we wish to create. For **Existing cluster**, select the ECS cluster we defined before, which for this example was called **mlewp2-ecs-cluster**. For **Compute configuration**, we will just use the **Launch type** option, which means we can just allow Fargate to manage the infrastructure requirements. If you have multiple infrastructure options that you want to blend together, then you can use the **Capacity provider strategy** option. Note that this is more advanced and so I encourage you to read more in the AWS documentation about your options here if you need to use this route. For reference, my selections are shown in *Figure 5.16*.

Existing cluster
Select an existing cluster. To create a new cluster, go to [Clusters](#).

mlewp2-ecs-cluster ▾

▼ Compute configuration (advanced)

Compute options | [Info](#)
To ensure task distribution across your compute types, use appropriate compute options.

Capacity provider strategy
Specify a launch strategy to distribute your tasks across one or more capacity providers.

Launch type
Launch tasks directly without the use of a capacity provider strategy.

Launch type | [Info](#)
Select either managed capacity (Fargate), or custom capacity (EC2 or user-managed, External instances). External instances are registered to your cluster using the ECS Anywhere capability.

FARGATE ▾

Platform version | [Info](#)
Specify the platform version on which to run your service.

LATEST ▾

Figure 5.16: AWS ECS selections for the environment that we will run our ECS service on. This service will enable the task definition we defined before, and therefore our application, to run continuously.

3. Next is the deployment configuration, which refers to how the service runs in terms of the number of replicas and what actions to take upon failures of the solution. I have defined the service name simply as **basic-ml-microservice-service**, and have used the **Replica** service type, which specifies how many tasks should be maintained across the cluster. We can leave this as **1** for now as we only have one task in our task definition. This is shown in *Figure 5.17*.

Task definition

Select an existing task definition. To create a new task definition, go to [Task definitions](#).

Specify the revision manually

Manually input the revision instead of choosing from the 100 most recent revisions for the selected task definition family.

Family	Revision
basic-ml-microservice-tasks	▼ 1 (LATEST)

Service name

Assign a unique name for this service.

basic-ml-microservice-service

Service type [Info](#)

Specify the service type that the service scheduler will follow.

Replica
Place and maintain a desired number of tasks across your cluster.

Daemon
Place and maintain one copy of your task on each container instance.

Desired tasks

Specify the number of tasks to launch.

1

Figure 5.17: Configuring the AWS ECS service name and type.

4. The Deployment options and Deployment failure detection

subsections will be auto-populated with some defaults. A rolling deployment type refers to the replacement of the container with the latest version when that is available. The failure detection options ensure that deployments that run into errors fail to proceed and that

rollbacks to previous versions are enabled. We do not need to enable **CloudWatch alarms** at this stage as we have not configured CloudWatch, but this could be added in future iterations of your project. See *Figure 5.18* for reference.

▼ Deployment options

Deployment type | [Info](#)

Select a deployment controller type for the service.

Rolling update ▾

Min running tasks % | [Info](#)

Specify the minimum percent of running tasks allowed during a service deployment.

100

values in %

Max running tasks % | [Info](#)

Specify the maximum percent of running tasks allowed during a service deployment.

200

values in %

▼ Deployment failure detection [Info](#)

Use the Amazon ECS deployment circuit breaker

If the service can't reach a steady state because a task failed to launch, the deployment fails.

Rollback on failures

If the current deployment fails, the service is rolled back to the last completed deployment state.

Use CloudWatch alarm(s)

If the CloudWatch alarm or alarms that you specify transition to the ALARM state, the deployment fails.

Figure 5.18: Deployment and failure detection options for the AWS ECS service we are about to deploy.

5. As in the other examples, there is a **Networking** section that should be prepopulated with the VPC and subnet information appropriate for your account. As before, you can switch these out for specific VPCs and subnets according to your requirements. *Figure 5.19* shows what this looks like for reference.

▼ Networking

VPC [Info](#)

Choose the Virtual Private Cloud to use.

vpc-422d8f3b

default

Subnets

Choose the subnets within the VPC that the task scheduler should consider for placement.

Choose subnets

subnet-23806e68



eu-west-1b 172.31.16.0/20

subnet-80223de6



eu-west-1a 172.31.0.0/20

subnet-f70159ad



eu-west-1c 172.31.32.0/20

Security group [Info](#)

Choose an existing security group or create a new security group.

Use an existing security group

Create a new security group

Security group name

Choose an existing security group.

Choose security groups

sg-e52346b4



default

Public IP [Info](#)

Choose whether to auto-assign a public IP to the task's elastic network interface (ENI).

Turned on

Figure 5.19: The networking section for the AWS ECS service that we are defining for hosting the ML microservice.

6. The remaining sections are optional and contain configuration elements for load balancing, auto-scaling, and tagging. Although we do not necessarily need it for such a simple application, we will use this section to create an application load balancer, which is one of the options available. An application load balancer routes HTTP and HTTPS requests and supports useful capabilities like path-based routing and dynamic host port mapping, which allows for multiple tasks from a single service to run on the same container. We can name the load balancer `basic-ml-microservice-lb` and configure the **listener** for this load balancer to listen on `port 80` with the HTTP protocol, as shown in *Figure 5.20*. This listener checks for connection requests at the given port and uses the specified protocol so that requests can then be routed by the load balancer to the downstream system.

▼ Load balancing - optional

Load balancer type [Info](#)

Configure a load balancer to distribute incoming traffic across the tasks running in your service.

Application Load Balancer

Application Load Balancer

Specify whether to create a new load balancer or choose an existing one.

Create a new load balancer

Use an existing load balancer

Load balancer name

Assign a unique name for the load balancer.

basic-ml-microservice-lb

Choose container to load balance

mlmicro 5000:5000

Listener [Info](#)

Specify the port and protocol that the load balancer will listen for connection requests on.

Create new listener

Use an existing listener

You need to select an existing load balancer.

Port

80

Protocol

HTTP

Figure 5.20: Defining the load balancer name and listener details for the AWS ECS service.

- Finally, we must specify a target group for the load balancer, which as the name suggests is basically the collection of target endpoints for the

tasks in your service. AWS ECS ensures that this updates as task definitions are updated through the lifetime of your service. *Figure 5.21* shows the configurations for the target group, which just specifies the HTTP protocol and home path for health checks.

Target group | [Info](#)

Specify whether to create a new target group or choose an existing one that the load balancer will use to route requests to the tasks in your service.

Create new target group
 Use an existing target group
You need to select an existing load balancer.

Target group name	Protocol
basic-ml-microservice-lb-tg	HTTP ▾
Health check path Info	Health check protocol
/	HTTP ▾
Health check grace period Info	
4	
seconds	

Figure 5.21: Target group definition for the application load balancer.

8. After filling in these details, hit the **Create** button. This will then deploy your service. If all has gone well, then you should be able to see the service in your cluster details on the AWS ECS console page. You can navigate to this service and find the load balancer. This will have a **Domain Name System (DNS)** address that will be the root of

the target URL for sending requests. *Figure 5.22* shows what this page with the DNS looks like. Copy or save this DNS value.

Figure 5.22: The deployed load balancer for our service with the DNS name in the bottom right-hand corner.

9. Finally, to test the service, we can run the same request we had for local testing in Postman, but now update the URL to contain the load balancer DNS name and the port we have specified that the load balancer will receive oncoming traffic with . For us, this is port 80. This is shown with the application response in *Figure 5.23*.

Figure 5.23: A valid result is returned by our simple forecasting service from the hosted application AWS ECS.

And that's it! We have now successfully built and deployed a simplified forecasting service using Flask, Docker, AWS Elastic Container Registry, AWS Elastic Container Service, and an application load balancer. These components can all be adapted for deploying your future ML microservices.

The first half of this chapter has been about architectural and design principles that apply at the system and code level, as well as showing you how some of this comes together in one mode of deployment that is very common for ML systems, that of the ML microservice. Now that we have done this, we will move on to discuss some tools and techniques that allow us to build, deploy, and host complex ML workflows as pipelines, a concept we briefly introduced earlier in the book. The tools and concepts we will cover in the second half of this chapter are crucial for any modern ML engineer to have a strong grasp of, as they are starting to form the backbone of so many deployed ML systems.

The next section will start this discussion with an exploration of how we can use Airflow to create and orchestrate flexible, general-purpose, production-ready pipelines, before we move on to some tools aimed specifically at advanced ML pipelining and orchestration.

Building general pipelines with Airflow

In *Chapter 4, Packaging Up*, we discussed the benefits of writing our ML code as pipelines. We discussed how to implement some basic ML pipelines using tools such as `sklearn` and `Spark ML`. The pipelines we were concerned with there were very nice ways of streamlining your code and making several processes available to use within a single object to simplify an application. However, everything we discussed then was very much focused on one Python file and not necessarily something we could extend very flexibly outside the confines of the package we were using. With the techniques we discussed, for example, it would be very difficult to create pipelines where each step was using a different package or even where they were entirely different programs. They did not allow us to build much sophistication into our data flows or application logic either, as if one of the steps failed, the pipeline failed, and that was that.

The tools we are about to discuss take these concepts to the next level. They allow you to manage the workflows of your ML solutions so that you can organize, coordinate, and orchestrate elements with the appropriate level of complexity to get the job done.

Airflow

Apache Airflow is the workflow management tool that was initially developed by **Airbnb** in the 2010s and has been open-source since its inception. It gives data scientists, data engineers, and ML engineers the capability of programmatically creating complex pipelines through Python scripts. Airflow's task management is based on the definition and then execution of a **Directed Acyclic Graph (DAG)** with nodes as the tasks to be run. DAGs are also used in **TensorFlow** and **Spark**, so you may have heard of them before.

Airflow contains a variety of default operators to allow you to define DAGs that can call and use multiple components as tasks, without caring about the specific details of a task. It also provides functionality for scheduling your pipelines. As an example, let's build an Apache Airflow pipeline that will get data, perform some feature engineering, train a model, and then persist the model. We won't cover the detailed implementation of each command, but simply show you how your ML processes hang together in an Airflow DAG. In *Chapter 9, Building an Extract, Transform, Machine Learning Use Case*, we will build out a detailed end-to-end example discussing these lower-level details. This first example is more concerned with understanding the high level of how to write, deploy, and manage your DAGs in the cloud:

1. First, in a file called `classification_pipeline_dag.py`, we can import the relevant Airflow packages and any utility packages we need:

```
import datetime
from datetime import timedelta
from airflow import DAG
```

```
from airflow.operators.python import PythonOperator
from airflow.utils.dates import days_ago
```

2. Next, Airflow allows you to define default arguments that can be referenced by all of the following tasks, with the option to overwrite at the same level:

```
default_args = {
    'owner': 'Andrew McMahon',
    'depends_on_past': False,
    'start_date': days_ago(31),
    'email': ['example@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=2)
}
```

3. We then have to instantiate our DAG and provide the relevant metadata, including our scheduling interval:

```
with DAG(
    dag_id="classification_pipeline",
    start_date=datetime.datetime(2021, 10, 1),
    schedule_interval="@daily",
    catchup=False,
) as dag:
```

4. Then, all that is required is to define your tasks within the `DAG` definition. First, we define an initial task that gets our dataset. This next piece of code assumes there is a Python executable, for example, a function or class method, called `get_data` that we can pass to the task. This could have been imported from any submodule or package we want. Note that *steps 3-5* assume we are inside the code block of the DAG instantiation, so we assume another indent that we don't show here to save space:

```
get_data_task = PythonOperator(  
    task_id="get_data",  
    python_callable=get_data  
)
```

5. We then perform a task that takes this data and performs our model training steps. This task could, for example, encapsulate one of the pipeline types we covered in *Chapter 3, From Model to Model Factory*; for example, a Spark ML pipeline, **Scikit-Learn** pipeline, or any other ML training pipeline we looked at. Again, we assume there is a Python executable called `train_model` that can be used in this step:

```
train_model_task = PythonOperator(  
    task_id="train_model",  
    python_callable=train_model  
)
```

6. The final step of this process is a placeholder for taking the resultant trained model and persisting it to our storage layer. This means that

other services or pipelines can use this model for prediction:

```
persist_model_task = PythonOperator(  
    task_id="persist_model",  
    python_callable=persist_model  
)
```

7. Finally, we define the running order of the task nodes that we have defined in the DAG using the `>>` operator. The tasks above could have been defined in any order, but the following syntax stipulates how they must run:

```
get_data_task >> train_model_task >> persist
```

In the next sections, we will briefly cover how to set up an Airflow pipeline on AWS using the **Managed Workflows for Apache Airflow (MWAA)** service. The section after will then show how you can use **CI/CD** principles to continuously develop and update your Airflow solutions. This will bring together some of the setup and work we have been doing in previous chapters of the book.

Airflow on AWS

AWS provides a cloud-hosted service called **Managed Workflows for Apache Airflow (MWAA)** that allows you to deploy and host your Airflow pipelines easily and robustly. Here, we will briefly cover how to do this.

Complete the following steps:

1. Select **Create an environment** on the MWAA landing page. You can find this by searching for MWAA in the AWS Management Console.
2. You will then be provided with a screen asking for the details of your new Airflow environment. *Figure 5.24* shows the high-level steps that the website takes you through:



Figure 5.29 – The high-level steps for setting up an MWAA environment and associated managed Airflow runs

Figure 5.24: The high-level steps for setting up an MWAA environment and associated managed Airflow runs.

Environment details, as shown in *Figure 5.25*, is where we specify our environment name. Here, we have called it **mlewp2-airflow-dev-env**:

Figure 5.25: Naming your MWAA environment.

3. For MWAA to run, it needs to be able to access code defining the DAG and any associated requirements or plugin files. The system then asks for an AWS S3 bucket where these pieces of code and configuration reside. In this example, we create a bucket called **mlewp2-ch5-airflow-example** that will contain these pieces. *Figure 5.26* shows the creation of the bucket:

Figure 5.26: The successful creation of our AWS S3 bucket for storing our Airflow code and supporting configuration elements.

Figure 5.27 shows how we point MWAA to the correct bucket, folders, and plugins or requirement files if we have them too:

Figure 5.27: We reference the bucket we created in the previous step in the configuration of the MWAA instance.

4. We then have to define the configuration of the network that the managed instance of Airflow will use, similar to the other AWS examples in this chapter. This can get a bit confusing if you are new to networking, so it might be good to read around the topics of subnets, IP addresses, and VPCs. Creating a new MWAA VPC is the easiest approach for getting started in terms of networking here, but your organization will have networking specialists who can help you use the appropriate settings for your situation. We will go with this simplest route and click **Create MWAA VPC**, which opens a new window where we can quickly spin up a new VPC and network setup based on a standard stack definition provided by AWS. You will be asked for a stack name. I have called mine **MLEWP - 2 - MWAA - VPC**. The networking information will be populated with something like that shown in *Figure 5.28*:

Figure 5.28: An example stack template for creating your new VPC.

5. We are then taken to a page where we are asked for more details on networking. We can select **Public network (No additional setup)** for this example as we will not be too concerned with creating an organizationally aligned security model. For deployments in an organization, work with your security team to understand what additional security you need to put in place. We can also select **Create new security group**. This is shown in *Figure 5.29*.

Figure 5.29: Finalizing the networking for our MWAA setup.

6. Next, we have to define the **Environment class** that we want to spin up. Currently, there are three options. Here, we'll use the smallest, but you can choose the environment that best suits your needs (always ask the billpayer's permission!). *Figure 5.30* shows that we can select the **mw1.small** environment class with a min to max worker count of 1-10. MWAA does allow you to change the environment class after instantiating if you need to, so it can often be better to start small and scale up as needed from a cost point of view. You will also be asked about the number of schedulers you want for the environment. Let's leave this as the default, **2**, for now, but you can go up to 5.

Figure 5.30: Selecting an environment class and worker sizes.

7. Now, if desired, we confirm some optional configuration parameters (or leave these blank, as done here) and confirm that we are happy for AWS to create and use a new execution role. We can also just proceed with the default monitoring settings. *Figure 5.31* shows an example of this (and don't worry, the security group will have long been deleted by the time you are reading this page!):

Figure 5.31: The creation of the execution role used by AWS for the MWAA environment.

8. The next page will supply you with a final summary before allowing you to create your MWAA environment. Once you do this, you will be able to see your newly created environment in the MWAA service, as in *Figure 5.32*. This process can take some time, and for this example it took around 30 minutes:

Figure 5.32: Our newly minted MWAA environment.

Now that you have this MWAA environment and you have supplied your DAG to the S3 bucket that it points to, you can open the Airflow UI and see the scheduled jobs defined by your DAG. You have now deployed a basic running service that we can build upon in later work.

Now we will want to see the DAGs in the Airflow UI so that we can orchestrate and monitor the jobs. To do this, you may need to configure access for your own account to the MWAA UI using the details outlined on the AWS documentation pages. As a quick summary, you need to go to the IAM service on AWS. You will need to be logged in as a root user, and then create a new policy title, **AmazonMWAAWebServerAccess**. Give this policy the following JSON body:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "airflow:CreateWebLoginTo",  
            "Resource": [  
                "arn:aws:airflow:{your-region}:",  
            ]  
        }  
    ]  
}
```

For this definition, the Airflow role refers to one of the five roles of **Admin**, **Op**, **Viewer**, **User**, or **Public**, as defined in the Airflow documentation at

<https://airflow.apache.org/docs/apache-airflow/stable/security/access-control.xhtml>. I have used the Admin role for this example. If you add this policy to the permissions of your account, you should be able to access the Airflow UI by clicking the **Open Airflow UI** button in the MWAA service. You will then be directed to the Airflow UI, as shown in *Figure 5.33*.

Figure 5.33: The Airflow UI accessed via the AWS MWAA service. This view shows the classification DAG that we wrote earlier in the example.

The Airflow UI allows you to trigger DAG runs, manage the jobs that you have scheduled, and monitor and troubleshoot your pipelines. As an example, upon a successful run, you can see summary information for the runs, as shown in *Figure 5.34*, and can use the different views to understand the time taken for each of the pipeline steps and diagnose where any issues have arisen if there are errors raised.

Figure 5.34: Example run summary for our simple classification DAG in the Airflow UI.

The pipeline we have built and run in this example is obviously very simple, with only core Python functionality being used. If you want to leverage other AWS services, for example, by submitting a Spark job to an EMR cluster, then you will need to configure further access policies like the one we did above for the UI access. This is covered in the MWAA documentation.

IMPORTANT NOTE



Once you have created this MWAA environment, you cannot pause it, as it costs a small amount to run per hour (around 0.5 USD per hour for the environment configuration above). MWAA does not currently contain a feature for pausing and resuming an environment, so you will have to delete the environment and re-instantiate a new one with the same configuration when required. This can be automated using tools such as **Terraform** or **AWS CloudFormation**, which we will not cover here. So, a word of warning – *DO NOT ACCIDENTALLY LEAVE YOUR ENVIRONMENT RUNNING*. For example, definitely do not leave it running for a week, like I may or may not have done.

Revisiting CI/CD for Airflow

We introduced the basics of CI/CD in *Chapter 2, The Machine Learning Development Process*, and discussed how this can be achieved by using **GitHub Actions**. We will now take this a step further and start to set up CI/CD pipelines that deploy code to the cloud.

First, we will start with an important example where we will push some code to an AWS S3 bucket. This can be done by creating a `.yml` file in your GitHub repo under your `.github./workflows` directory called `aws-s3-deploy.yml`. This will be the nucleus around which we will form our CI/CD pipeline.

The `.yml` file, in our case, will upload the Airflow DAG and contain the following pieces:

1. We name the process using the syntax for `name` and express that we want the deployment process to be triggered on a push to the main branch or a pull request to the main branch:

```
name: Upload DAGS to S3
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
```

2. We then define the jobs we want to occur during the deployment process. In this case, we want to upload our DAG files to an S3 bucket we have already created, and we want to use the appropriate AWS credentials we have configured in our GitHub secrets store:

```
jobs:
  deploy:
    name: Upload DAGS to Amazon S3
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v2
      - name: Configure AWS credentials from GitHub secrets
        uses: aws-actions/configure-aws-credentials@v1
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: us-east-1
```

Then, as part of the job, we run the step that copies the relevant files to our specified AWS S3 bucket. In this case, we are also specifying some details about how to make the copy using the AWS CLI. Specifically, here we want to copy over all the Python files to the `dags` folder of the repo:

```
- name: Copy files to bucket with the
  run: |
    aws s3 cp ./dags s3://github-actic
    --recursive--include "*.py"
```

3. Once we perform a `git push` command with updated code, this will then execute the action and push the `dag` Python code to the specified S3 bucket. In the GitHub UI, you will be able to see something like *Figure 5.35* on a successful run:

 *Figure 5.38 – A successful CI/CD process run via GitHub Actions and using the AWS CLI*

Figure 5.35: A successful CI/CD process run via GitHub Actions and using the AWS CLI.

This process then allows you to successfully push new updates to your Airflow service into AWS to be run by your MWAA instance. This is real CI/CD and allows you to continually update the service you are providing without downtime.

Building advanced ML pipelines

We have already discussed in this chapter how **SciKit-learn** and **Spark ML** provide mechanisms for creating ML pipelines. You can think of these as the basic way to do this and to get started. There are a series of tools now available, both open-source and enterprise, that take this concept to the next level.

For awareness, the three main public cloud providers have tools in this area you may want to be aware of and try out. **Amazon SageMaker** is one of the giants of this space and contains within it a large ecosystem of tools and capabilities to help take your ML models into production. This book could have been entirely about Amazon SageMaker, but since that was done elsewhere, in *Learn Amazon SageMaker*,

<https://tinyurl.com/mr48rsxp>, we will leave the details for the reader to discover. The key thing you need to know is that this is AWS's managed service for building up ML pipelines, as well as monitoring, model registry, and a series of other capabilities in a way that lets you develop and promote your models all the way through the ML lifecycle.

Google Vertex AI is the Google Cloud Platform ML pipelining, development, and deployment tool. It brings tons of functionality under one UI and API, like Sagemaker, but seems to have less flexibility on the types of models you can train. **Azure ML** is the Microsoft cloud provider's offering.

These are all enterprise-grade solutions that you can try for free, but you should be prepared to have your credit card ready when things scale up. The solutions above are also naturally tied into specific cloud providers and therefore can create “vendor lock-in,” where it becomes difficult to switch later. Thankfully, there are solutions that help with this and allow ML engineers to work with a less complex setup and then migrate to more

complex infrastructure and environments later. The first one of these that we will discuss is **ZenML**.

Finding your ZenML

ZenML is a completely open-source framework that helps you write ML pipelines in a way that is totally abstracted from the underlying infrastructure. This means that your local development environment and your eventual production environment can be very different, and can be changed through changes in configuration without altering the core of your pipelines. This is a very powerful idea and is one of ZenML’s key strengths.

ZenML has some core concepts that you need to understand in order to get the best out of the tool:

- **Pipelines:** As you might expect given the discussion in the rest of this chapter, these are the definitions of the steps in the ML workflow. Pipelines consist of “steps” chained together in a specified order.
- **Stacks:** Configurations specifying the environment and infrastructure that the pipeline is to run on.
- **Orchestrator:** Within the stack definition, there are two key components, the first of which is an orchestrator. Its job is to coordinate the steps in the pipeline that are executed on the infrastructure. This could be the default orchestrator that comes with the distribution or it could be something like Airflow or the Kubeflow orchestrator. Airflow is described in the *Building general pipelines with Airflow* section in this chapter and Kubeflow is covered in the *Going with the Kubeflow* section.

- **Artifact store:** This is the stack component responsible for data and metadata storage. ZenML has a series of different compatible artifact stores out of the box, specifically AWS S3, Azure Blob Storage, and Google Cloud Storage. The assumption here is that the artifact store is really just a storage layer, and nothing too complex on top of that.

So far, so straightforward. Let's get on and start setting ZenML up. You can install it with:

```
pip install zenml
```

We will also want to use the React dashboard that comes with ZenML, but to run this locally you also need to install a different repository:

```
pip install zenml[server]
```

ZenML also comes with a series of existing templates you can leverage, which you can install with:

```
pip install zenml[templates]
```

We can then start working with a template by running:

```
zenml init --template
```

This will then start a terminal-based wizard to help you generate the ZenML template. See *Figure 5.36*.

Figure 5.36: The ZenML template wizard.

Hit *Enter*; then you will be asked a series of questions to help configure the template. Some are shown with their answers in *Figure 5.37*.

Figure 5.37: Providing responses for the ZenML template definition.

The next series of questions start to get very interesting as we are asked about the details of the information we wish to be logged and made visible in the CLI, as well as selecting the dataset and model type. Here we will work with the `Wine` dataset, again using a `RandomForestClassifier`, as can be seen in *Figure 5.38*.

Figure 5.38: Selecting a model for the ZenML template instantiation.

ZenML will then start initializing the template for you. You can see that this process generates a lot of new files to use, as shown in *Figure 5.39*.

Figure 5.39: File and folder structure generated after using the ZenML template generation wizard.

Let's start to explore some of these elements for the ZenML solution. First, let's look at `pipelines/model_training.py`. This is a short script that is there to give you a starting point. Omitting the comments in the file, we have the following code present:

```
from zenml.pipelines import pipeline

@pipeline()
```

```
def model_training_pipeline(
    data_loader,
    data_processor,
    data_splitter,
    model_trainer,
    model_evaluator,
):
    dataset = data_loader()
    processed_dataset = data_processor(dataset=
        train_set, test_set = data_splitter(dataset))
    model = model_trainer(train_set=train_set)
    model_evaluator(
        model=model,
        train_set=train_set,
        test_set=test_set,
    )
```

We can already start to appreciate some of the features that are available in ZenML and how it works. First, we see that the use of the `@pipeline` decorator signals that the function following will contain the main pipeline logic. We can also see that the pipeline is actually written in pure Python syntax; all you need is the decorator to make it “Zen.” This is a very powerful feature of ZenML as it provides you the flexibility to work as you want but still leverage the downstream abstraction we will see soon for deployment targets. The steps inside the pipeline are just dummy function calls created when the template was initialized to help guide you in what you should develop.

Now, we will look at the pipeline steps, which have been defined in the `steps/data_loaders.py` and `steps/model_trainers.py`

files. In our discussions of these modules, we will not discuss the helper classes and utility functions used; these are left for the reader to play around with. Instead, we will focus on the pieces that show the most important ZenML functionality. Before we do that, let us briefly discuss some important ZenML modules that are imports at the top of the module:

```
from zenml.enums import StrEnum
from zenml.logger import get_logger
from zenml.steps import (
    BaseParameters,
    Output,
    step,
)
```

The first import brings in `StrEnum` from the `enums` module of ZenML. This is a collection of Python enumerations that have been defined to help with specific elements of building ZenML workflows.

IMPORTANT NOTE

Recall that a Python enumeration (or `enum`) is a collection of members with unique values that can be iterated over to return the values in their order of definition. You can think of these as somewhere between a class and a dictionary.

First, in the `data_loaders.py` module, we can see that the first step wraps simple logic for pulling in different datasets from `scikit-learn`, depending on the parameters supplied. This is a very basic example but can be updated to incorporate much more sophisticated behavior

like calling out to databases or pulling from cloud-hosted object storage. The method looks like the following:



```
@step
def data_loader(
    params: DataLoaderStepParameters,
) -> pd.DataFrame:
    # Load the dataset indicated in t
    # pandas DataFrame
    if params.dataset == SklearnDatasets.WINE:
        dataset = load_wine(as_frame=True)
    elif params.dataset == SklearnDatasets.IRIS:
        dataset = load_iris(as_frame=True)
    elif params.dataset == SklearnDatasets.BREAST_CANCER:
        dataset = load_breast_cancer(as_frame=True)
    elif params.dataset == SklearnDatasets.DIABETES:
        dataset = load_diabetes(as_frame=True)
    logger.info(f"Loaded dataset {params.dataset.name}")
    logger.info(dataset.head())
    return dataset
```

Note that the output of this function is a pandas DataFrame, and in the language of ZenML this is an artifact. The next important step given is data processing. The example given in the template looks like the following:

```
@step
def data_processor(
```

```
params: DataProcessorStepParameters,
dataset: pd.DataFrame,
) -> pd.DataFrame:
if params.drop_na:
    # Drop rows with missing values
    dataset = dataset.dropna()
if params.drop_columns:
    # Drop columns
    dataset = dataset.drop(columns=params.d
if params.normalize:
    # Normalize the data
    target = dataset.pop("target")
    dataset = (dataset - dataset.mean()) /
    dataset["target"] = target
return dataset
```

We can see that, here, the processing is relatively standard and will drop `NULL` values in the dataset, remove columns we have labeled in the `DataProcessingStepParameters` classes (not shown here), and apply some normalization using standard scaling – the steps given are in fact identical to applying the `sklearn.preprocessing.StandardScaler` method.

The final method in the data loaders module performs train/test splitting of the data, using methods we have already seen in this book:

```
@step
def data_splitter(
    params: DataSplitterStepParameters,
    dataset: pd.DataFrame,
```

```
    ) -> Output(train_set=pd.DataFrame, test_se
# Split the dataset into training and dev s
train_set, test_set = train_test_split(
    dataset,
    test_size=params.test_size,
    shuffle=params.shuffle,
    stratify=dataset["target"] if params.st
    random_state=params.random_state,
)
return train_set, test_set
```

Now, moving back into the `steps` folder, we can see that there is also a module entitled `model_trainers.py`. At the top of this folder are some more important imports we should understand before we proceed:

```
from zenml.enums import StrEnum
from zenml.logger import get_logger
from zenml.steps import (
    BaseParameters,
    Output,
    step,
)
from artifacts import ModelMetadata
from materializers import ModelMetadataMaterial
logger = get_logger(__name__)
```

In particular, we can see that ZenML provides a wrapper to the Python logging library and that there are two modules being used here, called `artifacts` and `materializers`. These are defined within the

template repo and show how you can create custom code to work with the artifact store. Specifically, in the `artifacts/model_metadata.py` module, there is a class that allows you to store model metadata in a format of your choosing for later serialization and deserialization. Once again, all docstrings and most imports are omitted for brevity:

```
from typing import Any, Dict
from sklearn.base import ClassifierMixin

class ModelMetadata:
    def __init__(self) -> None:
        self.metadata: Dict[str, Any] = {}
    def collect_metadata(
        self,
        model: ClassifierMixin,
        train_accuracy: float,
        test_accuracy: float,
    ) -> None:
        self.metadata = dict(
            model_type = model.__class__.__name__,
            train_accuracy = train_accuracy,
            test_accuracy = test_accuracy,
        )

    def print_report(self) -> None:
        """Print a user-friendly report from the
        print(f"""
        Model type: {self.metadata.get('model_t
        Accuracy on train set: {self.metadata.g
```

```
Accuracy on test set: {self.metadata.ge  
    """})
```

In ZenML, materializers are the objects that contain the logic for the serialization and deserialization of the artifacts. They define how your pipelines interact with the artifact store. When defining materializers, you can create custom code but you have to inherit from the `BaseMaterializer` class in order to ensure that ZenML knows how to persist and read in data between steps and at the beginning and end of pipelines. This is shown below in important code from `materializers/model_metadata.py`:

```
from zenml.materializers.base_materializer import  
  
class ModelMetadataMaterializer(BaseMaterializer):  
    # This needs to point to the artifact data  
    # materializer  
    ASSOCIATED_TYPES = (ModelMetadata,)  
    ASSOCIATED_ARTIFACT_TYPE = ArtifactType.DATASOURCES  
  
    def save(self, model_metadata: ModelMetadata):  
        super().save(model_metadata)  
        # Dump the model metadata directly into  
        # YAML file  
        with fileio.open(os.path.join(self.uri,  
                                      'w')) as f:  
            f.write(yaml.dump(model_metadata.me  
def load(self, data_type: Type[ModelMetadata]):  
    super().load(data_type)
```

```
with fileio.open(os.path.join(self.uri,
    model_metadata = ModelMetadata()
    model_metadata.metadata = yaml.safe
return model_metadata
```

Note that the naming convention of the path to this module shows that the materializer is paired with the `model_metadata` artifact we walked through in the previous code snippet. This is good practice and is similar to the organization of tests that we discussed in *Chapter 4, Packaging Up*.

Now that we have discussed all the key pieces of the ZenML template, we want to run the pipeline. This is done via runner `run.py` at the utmost level of the repository. You can then run the pipeline with:

```
python run.py
```

After the pipeline successfully runs (you will see a series of outputs in the terminal), you can run the following command to spin up a locally hosted ZenML dashboard:

```
zenml up
```

Now, if you navigate to the URL that is returned as output, usually something like `http://127.0.0.1:8237/login`, you will see a home screen like that shown in *Figure 5.40*.

Figure 5.40: The ZenML UI login page.

In the output that gave you the URL is also a default username and password, conveniently **default** and a blank. Fill these in and you will see the home page shown in *Figure 5.41*.

Figure 5.41: The ZenML UI home page.

If you then click through into the **Pipelines** section on the left and then click the pipeline created by your first run, you will be able to see all of the times that it has been run since then. This view is shown in *Figure 5.42*.

Figure 5.42: The pipelines view in the ZenML UI.

You can then also get really detailed information about the specifics of each run by clicking through. This gives you information like a graphical representation of the pipeline as a DAG at the time of the run. See *Figure 5.43*.

Figure 5.43: An example DAG for a ZenML pipeline shown in the UI.

If you click through on the pipeline name in any of these views, you can also retrieve the configuration of the run at the time of its execution in YAML format, which you can download and then use in subsequent pipeline runs:

Figure 5.44: An example YAML configuration for a ZenML pipeline run, shown in the ZenML UI.

This has only begun to scratch the surface of what is possible with ZenML, but hopefully, you can already see how it is a very flexible way to define and execute your ML pipelines. This becomes even more powerful when you leverage its ability to deploy the same pipeline across different stacks and different artifact stores.

In the next section, we will discuss another pipelining tool that focuses on creating cross-platform compatibility and standardization for your ML pipelines, **Kubeflow**.

Going with the Kubeflow

Kubeflow is an open-source solution aimed at providing portable methods for building end-to-end ML systems. This tool has a particular focus on giving developers the ability to quickly create pipelines for data processing, ML model training, prediction, and monitoring that are platform agnostic. It does all this by leveraging Kubernetes, allowing you to develop your solution on very different environments from where you eventually deploy. Kubeflow is agnostic about the particular programming and ML frameworks you use, so you can leverage everything you like out there in the open-source community but still stitch it together in a way you can trust.

The Kubeflow documentation provides a great wealth of detail on the architecture and design principles behind the tool at

<https://www.kubeflow.org/docs/>. We will focus instead on understanding the most salient points and getting started with some practical examples. This will allow you to compare to the other tools we have discussed in this chapter and make your own decisions around which to take forward in future projects.

Kubeflow is a platform that consists of multiple modular components, each one playing a role in the ML development lifecycle. Specifically, there are:

- The Jupyter Notebook web app and controller for exploratory data analysis and initial modeling.
- Training operators like PyTorch, TFJob, and XGBoost operators, among others, to build a variety of models.
- Hyperparameter tuning and neural network architecture search capabilities using Katib.
- Spark operators for data transformation, including an option for AWS EMR clusters.
- Dashboard for interfacing with your Kubernetes cluster and for managing your Kubeflow workloads.
- Kubeflow Pipelines: its own platform for building, running, and managing end-to-end ML workflows. This includes an orchestration engine for workflows with multiple steps and an SDK for working with your pipelines. You can install Kubeflow Pipelines as a standalone platform.

The installation steps for getting Kubeflow up and running can be quite involved and so it is best to look at the official documentation and run the appropriate steps for your platform and needs. We will proceed via the following steps:

1. Install Kind, a tool that facilitates easy building and running of local Kubernetes clusters. On Linux, this is done with:

```
curl -Lo ./kind https://kind.sigs.k8s.io/dl/
chmod +x ./kind && \
mv ./kind /{YOUR_KIND_DIRECTORY}/kind
```

And on MacOS this is done by:

```
brew install kind
```

2. Install the Kubernetes command-line tool `kubectl`, which allows you to interact with your cluster. On Linux, this is done with:

```
sudo apt-get install kubectl
```

Or on MacOS:

```
brew install kubernetes-cli
```

3. To check this has worked, you can run the following command in the terminal:

```
kubectl version --client --output=yaml
```

4. And you should receive an output like this:

```
clientVersion:  
  buildDate: "2023-01-18T15:51:24Z"  
  compiler: gc  
  gitCommit: 8f94681cd294aa8cf3407b8191f6c7  
  gitTreeState: clean  
  gitVersion: v1.26.1  
  goVersion: go1.19.5
```

```
major: "1"
minor: "26"
platform: darwin/arm64
kustomizeVersion: v4.5.7
```

5. Use Kind to create your local cluster. As the default, the name of the cluster will be `kind`, but you can provide your own name as a flag:

```
kind create cluster --name=mlewp
```

6. You will then see output that is something like this:

```
Creating cluster "mlewp" ...
[?] Ensuring node image (kindest/node:v1.25.3)
[?] Preparing nodes
[?] Writing configuration
[?] Starting control-plane
[?] Installing CNI
[?] Installing StorageClass
Set kubectl context to "kind-mlewp"
You can now use your cluster with:
kubectl cluster-info --context kind-mlewp
Thanks for using kind!
```

7. You then have to deploy Kubeflow pipelines to the cluster. The commands for doing this have been brought into a script called `deploy_kubeflow_pipelines.zsh` in the book's GitHub repository and it contains the following code (the

`PIPELINE_VERSION` number can be updated as needed to match your installation):

```
export PIPELINE_VERSION=1.8.5
kubectl apply -k "github.com/kubeflow/pipeline"
kubectl wait --for condition=established --t
kubectl apply -k "github.com/kubeflow/pipeline"
```

After running these commands, you can verify that the installation was a success through port forwarding and opening the Kubeflow Pipelines UI at `http://localhost:8080/`:

```
kubectl port-forward -n kubeflow svc/ml-pipeline
```

This should then give you a landing page like the one shown in *Figure 5.45*.

Figure 5.45: The Kubeflow UI landing page.

Now that you have initiated port-forwarding with the previous command, you will use this to allow the Kubeflow Pipelines SDK to talk to the cluster via the following Python code (note that you cannot do this until you have installed the Kubeflow Pipelines SDK, which is covered in the next step):

```
import kfp
client = kfp.Client(host="http://localhost:8080")
```

To install the Kubeflow Pipelines SDK, run:

```
pip install kfp -upgrade
```

To check that everything is in order, you can run this command:

```
pip list | grep kfp
```

Which gives output that should be similar to:

kfp	1.8.19
kfp-pipeline-spec	0.1.16
kfp-server-api	1.8.5

And that's it! We are now ready to start building some Kubeflow pipelines. Let's get started with a basic example.

We can now start building out some basic pipelines using the SDK and then we can deploy them to our cluster. Let's assume for the next few steps we are working in a file called `pipeline_basic.py`:

1. First, we import what is known as the **KFP Domain-Specific Language (DSL)**, which is a set of Python packages with various utilities for defining KFP steps. We also import the client package for interacting with the cluster. We'll also import several DSL sub-modules that we will use later. An important point to note here is that some functionality we will leverage is in fact contained in the **V2** of the Kubeflow pipelines SDK and so we will need to import some of those specific modules as well:

```
from kfp import Client
import kfp.dsl
from kfp.v2 import dsl
from kfp.v2.dsl import Dataset
from kfp.v2.dsl import Input
from kfp.v2.dsl import Model
from kfp.v2.dsl import Output
```

2. The next step is to define the steps in the pipeline. These are called “components” and are functions wrapped with `dsl` decorators. In this first step, we retrieve the Iris dataset and write it to CSV. In the first line, we will use the `dsl` decorator and also define what packages need to be installed in the container that will run that step:

```
@dsl.component(packages_to_install=['pandas'])
def create_dataset(iris_dataset: Output[Data]):
    import pandas as pd
    csv_url = "https://archive.ics.uci.edu/ml/databases/iris/iris.data"
    col_names = ["Sepal_Length", "Sepal_Width", "Petal_Length", "Petal_Width", "Labels"]
    df = pd.read_csv(csv_url)
    df.columns = col_names
    with open(iris_dataset.path, 'w') as f:
        df.to_csv(f)
```

3. Now that we have retrieved a dataset, and remembering what we learned in *Chapter 3, From Model to Model Factory*, we want to feature engineer this data. So, we will normalize the data in another

component. Most of the code should be self-explanatory, but note that we have had to add the `scikit-learn` dependency in the `packages_to_install` keyword argument and that we have again had to write the result of the component out to a CSV file:

```
@dsl.component(packages_to_install=['pandas'])
def normalize_dataset(
    input_iris_dataset: Input[Dataset],
    normalized_iris_dataset: Output[Dataset]
    standard_scaler: bool,
    min_max_scaler: bool,
):
    if standard_scaler is min_max_scaler:
        raise ValueError(
            'Exactly one of standard_scaler
            be True.')
    import pandas as pd
    from sklearn.preprocessing import MinMaxScaler
    from sklearn.preprocessing import StandardScaler
    with open(input_iris_dataset.path) as f:
        df = pd.read_csv(f)
    labels = df.pop('Labels')
    if standard_scaler:
        scaler = StandardScaler()
    if min_max_scaler:
        scaler = MinMaxScaler()
    df = pd.DataFrame(scaler.fit_transform(df))
    df['Labels'] = labels
    with open(normalized_iris_dataset.path,
              'w') as f:
        df.to_csv(f)
```

4. We will now train a K-nearest neighbors classifier on the data. Instead of outputting a dataset in this component, we will output the trained model artifact, a `.pkl` file:

```
@dsl.component(packages_to_install=['pandas=learn==1.0.2'])
def train_model(
    normalized_iris_dataset: Input[Dataset],
    model: Output[Model],
    n_neighbors: int,
):
    import pickle
    import pandas as pd
    from sklearn.model_selection import train_test_split
    from sklearn.neighbors import KNeighborsClassifier
    with open(normalized_iris_dataset.path)
        df = pd.read_csv(f)
    y = df.pop('Labels')
    X = df
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
    clf = KNeighborsClassifier(n_neighbors=n_neighbors)
    clf.fit(X_train, y_train)
    with open(model.path, 'wb') as f:
        pickle.dump(clf, f)
```

5. We now have all the components for the work we want to do, so now we can finally bring it together into a Kubeflow pipeline. To do this,

we use the `@dsl.pipeline` decorator and as an argument to that decorator, we provide the name of the pipeline:

```
@dsl.pipeline(name='iris-training-pipeline')
def my_pipeline(
    standard_scaler: bool,
    min_max_scaler: bool,
    neighbors: List[int],
):
    create_dataset_task = create_dataset()
    normalize_dataset_task = normalize_data
        input_iris_dataset=create_dataset_ta
        .outputs['iris_dataset'],
        standard_scaler=True,
        min_max_scaler=False)
    with dsl.ParallelFor(neighbors) as n_nei
        train_model(
            normalized_iris_dataset=normaliz
            .outputs['normalized_iris_dataset'],
            n_neighbors=n_neighbors)
```

6. The final stage is to submit the pipeline to run. This is done by instantiating a Kubeflow Pipelines client class and feeding in the appropriate arguments. `<KFP_UI_URL>` is the URL for the host of your instance of Kubeflow Pipelines – in this case, the one that we got from performing port-forwarding earlier. It is also important to note that since we are using several features from the `V2` Kubeflow Pipelines API, we should pass in the

```
kfp.dsl.PipelineExecutionMode.V2_COMPATIBLE flag
```

for the mode argument:

```
endpoint = '<KFP_UI_URL>'  
kfp_client = Client(host=endpoint)  
run = kfp_client.create_run_from_pipeline_fn  
mode=kfp.dsl.PipelineExecutionMode.V2_CC  
arguments={  
    'min_max_scaler': True,  
    'standard_scaler': False,  
    'neighbors': [3, 6, 9]  
},  
)  
url = f'{endpoint}#/runs/details/{run.run_id}'  
print(url)
```

7. To build and deploy this pipeline and run it, you can then execute:

```
python basic_pipeline.py
```

After running this last step, you will see the URL of the run is printed to the terminal, and should look something like this:

```
http://localhost:8080#/runs/details/<UID>
```

If you navigate to that link and the pipeline has successfully run, you should see a view in the Kubeflow dashboard showing the steps of the pipeline, with a sidebar that allows you to navigate through a series of metadata

about your pipeline and its run. An example from running the above code is shown in *Figure 5.46*.

Figure 5.46: The Kubeflow UI showing the successful run of the training pipeline defined in the main text.

And that's it, you have now built and run your first Kubeflow pipeline!

IMPORTANT NOTE

You can also compile your Kubeflow pipelines to serialized YAML, which can then be read by the Kubeflow backend. You would do this by running a command like the following, where `pipeline` is the same pipeline object used in the previous example:



```
cmplr = compiler.Compiler()  
cmplr.compile(my_pipeline, package_pa
```

One reason to do this is it is then super easy to run the pipeline. You can just upload it to the Kubeflow Pipelines UI, or you can send the YAML to the cluster programmatically.

As in the *Finding your ZenML* section, we have only begun to scratch the surface of this tool and have focused initially on getting to know the basics in a local environment. The beauty of Kubeflow being based on Kubernetes

is that platform agnosticism is very much at its core and so these pipelines can be effectively run anywhere that supports containers.



IMPORTANT NOTE

That although I have presented ZenML and Kubeflow as two different pipelining tools, they can actually be viewed as complementary, so much so that ZenML provides the ability to deploy Kubeflow pipelines through the use of the ZenML Kubeflow orchestrator. This means you can leverage the higher-level abstractions provided by ZenML but still get the scaling behavior and robustness of Kubeflow as a deployment target. We will not cover the details here but the ZenML documentation provides an excellent guide: <https://docs.zenml.io/stacks-and-components/component-guide/orchestrators/kubeflow>.

The next section will finish the chapter with a brief note on some different deployment strategies that you should be aware of when you aim to put all of these tools and techniques into practice with real solutions.

Selecting your deployment strategy

We have discussed many of the technical details of ways to take ML solutions into production in this chapter. The missing piece, however, is that we have not defined how you deal with existing infrastructure and how you

introduce your solution to real traffic and requests. This is what is defined by your deployment strategy, and selecting an appropriate one is an important part of being an ML engineer.

Most deployment strategies are, like many of the concepts in this book, inherited from the world of **software engineering** and **DevOps**. Two of the most important to know about are listed below with some discussion about when they can be particularly useful in an ML context.

Blue/green deployments are deployments where the new version of your software runs alongside your existing solution until some predefined criteria are met. After this point, you then switch all incoming traffic/requests to the new system before decommissioning the old one or leave it there for use as a potential rollback solution. The method was originally developed by two developers, Daniel North and Jez Humble, who were working on an e-commerce site in 2005.

The origin of the name is described in this GitHub Gist, <https://gitlab.com/-/snippets/1846041>, but essentially boils down to the fact that any other naming convention they could come up with always implied one of the candidate solutions or environments was “better” or “worse” than the other, for example with “A and B” or “Green and Red.” The strategy has since become a classic.

In an ML engineering context, this is particularly useful in scenarios where you want to gather model and solution performance data over a known period of time before trusting full deployment. It also helps with giving stakeholders evidence that the ML solution will perform as expected “in the wild.” It also plays particularly well with batch jobs, as you are just effectively running another batch at the same time. This may have some

cost implications for you to consider if the job is big or complex, or if your production environment is costly to maintain.

The next strategy is known as **canary deployments** and involves a similar setup to the blue/green method but involves a more gradual switching of traffic between the two solutions. Here the idea is that the new system is deployed and receives some percentage of the traffic initially, say 5% or 10%, before stability and performance are confirmed, and then the next increment of traffic is added. The total always remains at 100% so as the new system gains more traffic, the old system receives less. The name originates from the old coal mining technique of using canaries as a test of toxicity in the atmosphere in mines. Release the canaries and if they survive, all is well. Thankfully, no birds are harmed in the usage of this deployment technique. This strategy makes a lot of sense when you are able to divide the data you need to score and still get the information you need for progression to the next stage. As an example, an ML microservice that is called in the backend of a website would fit the bill nicely, as you can just gradually change the routing to the new service on your load balancer. This may make less sense for large batch jobs, as there may be no natural way to split your data into different increments, whereas with web traffic there definitely is.

Chapter 8, Building an Example ML Microservice, will show you how to use these strategies when building a custom ML endpoint using Kubernetes.

No matter what deployment strategy you use, always remember that the key is to strike the balance between cost-effectiveness, the uptime of your solution, and trust in the outputs it produces. If you can do all of these, then you will have deployed a winning combination.

Summary

In this chapter, we have discussed some of the most important concepts when it comes to deploying your ML solutions. In particular, we focused on the concepts of architecture and what tools we could potentially use when deploying solutions to the cloud. We covered some of the most important patterns used in modern ML engineering and how these can be implemented with tools such as containers and AWS Elastic Container Registry and Elastic Container Service, as well as how to create scheduled pipelines in AWS using Managed Workflows for Apache Airflow. We also explored how to hook up the MWAA example with GitHub Actions, so that changes to your code can directly trigger updates of running services, providing a template to use in future CI/CD processes.

We then moved on to a discussion of more advanced pipelining tools to build on the discussion in *Chapter 4, Packaging Up*. This focused on how to use Apache Airflow to build and orchestrate your generic pipelines for running your data engineering, ML, and MLOps pipelines. We then moved on to a detailed introduction to ZenML and Kubeflow, two powerful tools for developing and deploying ML and MLOps pipelines at scale.

In the next chapter, we will look at the question of other ways to scale up our solutions so that we can deal with large volumes of data and high-throughput calculations.

Join our community on Discord

Join our community's Discord space for discussion with the author and other readers:

<https://packt.link/mle>

6

Scaling Up

The previous chapter was all about starting the conversation around how we get our solutions out into the world using different deployment patterns, as well as some of the tools we can use to do this. This chapter will aim to build on that conversation by discussing the concepts and tools we can use to scale up our solutions to cope with large volumes of data or traffic.

Running some simple **machine learning (ML)** models on a few thousand data points on your laptop is a good exercise, especially when you're performing the discovery and proof-of-concept steps we outlined previously at the beginning of any ML development project. This approach, however, is not appropriate if we have to run millions upon millions of data points at a relatively high frequency, or if we have to train thousands of models of a similar scale at any one time. This requires a different approach, mindset, and toolkit.

In the following pages, we will cover some details of two of the most popular frameworks for distributing data computations in use today:

Apache Spark and **Ray**. In particular, we will discuss some of the key points about how these frameworks tick under the hood so that, in development, we can make some good decisions about how to use them. We will then move onto a discussion of how to use these in your ML workflows with some concrete examples, these examples being specifically

aimed to help you when it comes to processing large batches of data. Next, a brief introduction to creating serverless applications that allow you to scale out inference endpoints will be provided. Finally, we will cover an introduction to scaling containerized ML applications with Kubernetes, which complements the work we did in *Chapter 5, Deployment Patterns and Tools*, and will be built upon in detail with a full end-to-end example in *Chapter 8, Building an Example ML Microservice*.

This will help you build on some of the practical examples we already looked at earlier in this book, when we used Spark to solve our ML problems, with some more concrete theoretical understanding and further detailed practical examples. After this chapter, you should feel confident in how to use some of the best frameworks and techniques available to scale your ML solutions to larger and larger datasets.

In this chapter, we will cover all of this in the following sections:

- Scaling with Spark
- Spinning up serverless infrastructure
- Containerizing at scale with Kubernetes
- Scaling with Ray
- Designing systems at scale

Technical requirements

As with the other chapters, you can set up your Python development environment to be able to run the examples in this chapter by using the supplied Conda environment `yml` file or the `requirements.txt` files from the book repository, under *Chapter06*:

```
conda env create -f mlewp-chapter06.yml
```

This chapter's examples will also require some non-Python tools to be installed to follow the examples end to end; please see the respective documentation for each tool:

- AWS CLI v2
- Docker
- Postman
- Ray
- Apache Spark (version 3.0.0 or higher)

Scaling with Spark

Apache Spark, or just Spark, came from the work of some brilliant researchers at the *University of California, Berkeley* in 2012 and since then, it has revolutionized how we tackle problems with large datasets.

Spark is a cluster computing framework, which means it works on the principle that several computers are linked together in a way that allows computational tasks to be shared. This allows us to coordinate these tasks effectively. Whenever we discuss running Spark jobs, we always talk about *the cluster* we are running on.

This is the collection of computers that perform the tasks, the worker nodes, and the computer that hosts the organizational workload, known as the head node.

Spark is written in Scala, a language with a strong functional flavor and that compiles down to **Java Virtual Machines (JVMs)**. Since this is a book

about ML engineering in Python, we won't discuss too much about the underlying Scala components of Spark, except where they help us use it in our work. Spark has several popular APIs that allow programmers to develop with it in a variety of languages, including Python. This gives rise to the PySpark syntax we have used in several examples throughout this book.

So, how is this all put together?

Well, first of all, one of the things that makes Apache Spark so incredibly popular is the large array of connectors, components, and APIs it has available. For example, four main components interface with *Spark Core*:

- **Spark SQL, DataFrames, and Datasets:** This component allows you to create very scalable programs that deal with structured data. The ability to write SQL-compliant queries and create data tables that leverage the underlying Spark engine through one of the main **structured APIs** of Spark (Python, Java, Scala, or R) gives very easy access to the main bulk of Spark's functionality.
- **Spark Structured Streaming:** This component allows engineers to work with streaming data that's, for example, provided by a solution such as Apache Kafka. The design is incredibly simple and allows developers to simply work with streaming data as if it is a growing Spark structured table, with the same querying and manipulation functionality as for a standard one. This provides a low entry barrier for creating scalable streaming solutions.
- **GraphX:** This is a library that allows you to implement graph parallel processing and apply standard algorithms to graph-based data (for example, algorithms such as PageRank or Triangle Counting). The **GraphFrames** project from Databricks makes this functionality even

easier to use by allowing us to work with DataFrame-based APIs in Spark and still analyze graph data.

- **Spark ML:** Last but not least, we have the component that's most appropriate for us as ML engineers: Spark's native library for ML. This library contains the implementation of many algorithms and feature engineering capabilities we have already seen in this book. Being able to use **DataFrame** APIs in the library makes this extremely easy to use, while still giving us a route to creating very powerful code.

The potential speedups you can gain for your ML training by using Spark ML on a Spark cluster versus running another ML library on a single thread can be tremendous. There are other tricks we can apply to our favorite ML implementations and then use Spark to scale them out; we'll look at this later.

Spark's architecture is based on the driver/executor architecture. The driver is the program that acts as the main entry point for the Spark application and is where the **SparkContext** object is created. **SparkContext** sends tasks to the executors (which run on their own JVMs) and communicates with the cluster manager in a manner appropriate to the given manager and what mode the solution is running in. One of the driver's main tasks is to convert the code we write into a logical set of steps in a **Directed Acyclic Graph (DAG)** (the same concept that we used with Apache Airflow in *Chapter 5, Deployment Patterns and Tools*), and then convert that DAG into a set of tasks that needs to be executed across the available compute resources.

In the pages that follow, we will assume we are running Spark with the **Hadoop YARN** resource manager, which is one of the most popular options and is also used by the **AWS Elastic MapReduce (EMR)** solution by default (more on this later). When running with YARN in *cluster mode*, the driver program runs in a container on the YARN cluster, which allows a client to submit jobs or requests through the driver and then exit (rather than requiring the client to remain connected to the cluster manager, which can happen when you're running in so-called *client mode*, which we will not discuss here).

The cluster manager is responsible for launching the executors across the resources that are available on the cluster.

Spark's architecture allows us, as ML engineers, to build solutions with the same API and syntax, regardless of whether we are working locally on our laptop or a cluster with thousands of nodes. The connection between the driver, the resource manager, and the executors is what allows this magic to happen.

Spark tips and tricks

In this subsection, we will cover some simple but effective tips for writing performant solutions with Spark. We will focus on key pieces of syntax for data manipulation and preparation, which are always the first steps in any ML pipeline. Let's get started:

1. First, we will cover the basics of writing good Spark SQL. The entry point for any Spark program is the `SparkSession` object, which we need to import an instance of in our application.
It is often instantiated with the `spark` variable:

```
from pyspark.sql import SparkSession

spark = SparkSession\
    .builder\
    .appName("Spark SQL Example")\
    .config("spark.some.config.option", "some_value")\
    .getOrCreate()
```

2. You can then run Spark SQL commands against your available data using the `spark` object and the `sql` method:

```
spark.sql('''select * from data_table''')
```

There are a variety of ways to make the data you need available inside your Spark programs, depending on where they exist. The following example has been taken from some of the code we went through in *Chapter 3, From Model to Model Factory*, and shows how to pull data into a DataFrame from a `csv` file:

```
data = spark.read.format("csv")\
    .option("sep", ";")\
    .option("inferSchema", "true")\
    .option("header", "true").load(
    "data/bank/bank.csv")
```

3. Now, we can create a temporary view of this data using the following syntax:

```
data.createOrReplaceTempView('data_view')
```

4. Then, we can query against this data using the methods mentioned previously to see the records or to create new DataFrames:

```
new_data = spark.sql('''select ...''')
```

When writing Spark SQL, some standard practices help your code to be efficient:

- Try not to join big tables on the left with small tables on the right as this is inefficient. In general, try and make datasets used for joins as lean as possible, so, for example, do not join using unused columns or rows as much as possible.
- Avoid query syntax that will scan full datasets if they are very large; for example, `select max(date_time_value)`.

In this case, try and define logic that can filter the data more aggressively before finding min or max values and in general allow the solution to scan over a smaller dataset.

Some other good practices when working with Spark are as follows:

- **Avoid data skew:** Do what you can to understand how your data will be split across executors. If your data is partitioned on a date column, this may be a good choice if volumes of data are comparable for each day but bad if some days have most of your data and others very little. Repartitioning using a more appropriate column (or on a Spark-generated ID from the `repartition` command) will be required.

- **Avoid data shuffling:** This is when data is redistributed across different partitions. For example, we may have a dataset that is partitioned at the day level and then we ask Spark to sum over one column of the dataset for all of time. This will cause all of the daily partitions to be accessed and the result to be written to a new partition. For this to occur, disk writes and a network transfer have to occur, which can often lead to performance bottlenecks for your Spark job.
- **Avoid actions in large datasets:** For example, when you run the `collect()` command, you will bring all of your data back onto the driver node. This can be very bad if it is a large dataset but may be needed to convert the result of a calculation into something else. Note that the `toPandas()` command, which converts your Spark `DataFrame` into a pandas `DataFrame`, also collects all the data in the driver's memory.
- **Use UDFs when they make sense:** Another excellent tool to have in your arsenal, as an ML engineer using Apache Spark, is the **User-Defined Function (UDF)**. UDFs allow you to wrap up more complex and bespoke logic and apply it at scale in a variety of ways. An important aspect of this is that if you write a standard PySpark (or Scala) UDF, then you can apply this *inside* Spark SQL syntax, which allows you to efficiently reuse your code and even simplify the application of your ML models. The downside is that these are sometimes not the most efficient pieces of code, but if it helps to make your solution simpler and more maintainable, it may be the right choice.

As a concrete example, let's build a UDF that looks at the banking data we worked with in *Chapter 3, From Model to Model Factory*, to create a new

column called ‘month_as_int’ that converts the current string representation of the month into an integer for processing later. We will not concern ourselves with train/test splits or what this might be used for; instead, we will just highlight how to apply some logic to a PySpark UDF.

Let’s get started:

1. First, we must read the data. Note that the relative path given here is consistent with the `spark_example_udfs.py` script, which can be found in this book’s GitHub repository at
https://github.com/PacktPublishing/Machine-Learning-Engineering-with-Python-Second-Edition/blob/main/Chapter06/mlewp2-spark/spark_example_udfs.py:

```
from pyspark.sql import SparkSession
from pyspark import SparkContext
from pyspark.sql import functions as f

sc = SparkContext("local", "Ch6BasicExampleA")
# Get spark session
spark = SparkSession.builder.getOrCreate()
# Get the data and place it in a spark dataframe
data = spark.read.format("csv").option("sep",
                                         "data/bank/bank.csv")
```

If we show the current data with the `data.show()` command, we will see something like this:

age	job marital education default housing loan contact day month duration campaign pdays previous poutcome y
30 unemployed married primary no 1787 no no cellular 19 oct 79 1 -1 0 unknown no	
33 services married secondary no 4789 yes yes cellular 11 may 220 1 339 4 failure no	
35 management single tertiary no 1350 yes no cellular 16 apr 185 1 330 1 failure no	
30 management married tertiary no 1476 yes yes unknown 3 jun 199 4 -1 0 unknown no	
59 blue-collar married secondary no 0 yes no unknown 5 may 226 1 -1 0 unknown no	
35 management single tertiary no 747 yes no cellular 23 feb 141 2 176 3 failure no	
36 self-employed married tertiary no 307 yes no cellular 14 may 341 1 330 2 other no	
39 technician married secondary no 147 yes no cellular 6 may 151 2 -1 0 unknown no	
41 entrepreneur married tertiary no 221 yes no unknown 14 may 57 2 -1 0 unknown no	
43 services married primary no -88 yes yes cellular 17 apr 313 1 147 2 failure no	

Figure 6.1: A sample of the data from the initial DataFrame in the banking dataset.

2. Now, we can double-check the schema of this DataFrame using the `data.printSchema()` command. This confirms that `month` is stored as a string currently, as shown here:

```
| -- age: integer (nullable = true)
| -- job: string (nullable = true)
| -- marital: string (nullable = true)
| -- education: string (nullable = true)
| -- default: string (nullable = true)
| -- balance: integer (nullable = true)
| -- housing: string (nullable = true)
| -- loan: string (nullable = true)
| -- contact: string (nullable = true)
| -- day: integer (nullable = true)
| -- month: string (nullable = true)
| -- duration: integer (nullable = true)
| -- campaign: integer (nullable = true)
| -- pdays: integer (nullable = true)
| -- previous: integer (nullable = true)
| -- poutcome: string (nullable = true)
| -- y: string (nullable = true)
```

3. Now, we can define our UDF, which will use the Python `datetime` library to convert the string representation of the month into an integer:

```
import datetime

def month_as_int(month):
    month_number = datetime.datetime.strptime(month, '%B')
    return month_number
```

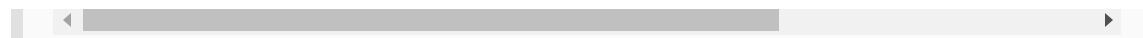
4. If we want to apply our function inside Spark SQL, then we must register the function as a UDF. The arguments for the `register()` function are the registered name of the function, the name of the Python function we have just written, and the return type. The return type is `StringType()` by default, but we have made this explicit here:

```
from pyspark.sql.types import StringType

spark.udf.register("monthAsInt", month_as_ir
```

5. Finally, now that we have registered the function, we can apply it to our data. First, we will create a temporary view of the bank dataset and then run a Spark SQL query against it that references our UDF:

```
data.createOrReplaceTempView('bank_data_view'
spark.sql('''
select *, monthAsInt(month) as month_as_int
'''').show()
```



Running the preceding syntax with the `show()` command shows that we have successfully calculated the new column. The last few columns of the resulting `DataFrame` are shown here:

pdays	previous	poutcome	y	month_as_int
-1	0	unknown	no	10
339	4	failure	no	5
330	1	failure	no	4
-1	0	unknown	no	6
-1	0	unknown	no	5
176	3	failure	no	2
330	2	other	no	5
-1	0	unknown	no	5
-1	0	unknown	no	5
147	2	failure	no	4
-1	0	unknown	no	5
-1	0	unknown	no	4
-1	0	unknown	no	8
-1	0	unknown	yes	4
241	1	failure	no	1
-1	0	unknown	no	8
-1	0	unknown	no	8
152	2	failure	no	4
-1	0	unknown	no	5
152	1	other	no	7

Figure 6.2: The new column has been calculated successfully by applying our UDF.

6. Alternatively, we can create our UDF with the following syntax and apply the result to a Spark `DataFrame`. As mentioned before, using a UDF can sometimes allow you to wrap up relatively complex syntax quite simply. The syntax here is quite simple but I'll show you it anyway. This gives us the same result that's shown in the preceding screenshot:

```
from pyspark.sql.functions import udf

month_as_int_udf = udf(month_as_int, StringType)
df = spark.table("bank_data_view")
df.withColumn('month_as_int', month_as_int_u
```

7. Finally, PySpark also provides a nice decorator syntax for creating our UDF, meaning that if you are indeed building some more complex functionality, you can just place this inside the Python function that is being decorated. The following code block also gives the same results as the preceding screenshot:

```
@udf("string")
def month_as_int_udf(month):
    month_number = datetime.datetime.strptime(month, "%m")
    return month_number
df.withColumn('month_as_int', month_as_int_u
```

This shows how we can apply some simple logic in a UDF, but for us to deploy a model at scale using this approach, we have to put the ML logic inside the function and apply it in the same manner. This can become a bit tricky if we want to work with some of the standard tools we are used to from the data science world, such as Pandas and **Scikit-learn**. Luckily, there is another option we can use that has a few benefits. We will discuss this now.

The UDFs currently being considered have a slight issue when we are working in Python in that translating data between the JVM and Python can

take a while. One way to get around this is to use what is known as **pandas UDFs**, which use the Apache Arrow library under the hood to ensure that the data is read quickly for the execution of our UDFs. This gives us the flexibility of UDFs without any slowdown.

pandas UDFs are also extremely powerful because they work with the syntax of – you guessed it – pandas **Series** and **DataFrame** objects. This means that a lot of data scientists who are used to working with pandas to build models locally can easily adapt their code to scale up using Spark.

As an example, let's walk through how to apply a simple classifier to the wine dataset that we used earlier in this book. Note that the model was not optimized for this data; we are just showing an example of applying a pre-trained classifier:

1. First, let's create a simple **Support Vector Machine (SVM)**-based classifier on the wine dataset. We are not performing correct training/test splits, feature engineering, or other best practices here as we just want to show you how to apply any `sklearn` model:

```
import sklearn.svm
import sklearn.datasets

clf = sklearn.svm.SVC()
X, y = sklearn.datasets.load_wine(return_X_y
```

2. We can then bring the feature data into a Spark DataFrame to show you how to apply the pandas UDF in later stages:

```
df = spark.createDataFrame(X.tolist())
```

3. pandas UDFs are very easy to define. We just write our logic in a function and then add the `@pandas_udf` decorator, where we also have to provide the output type for the function. In the simplest case, we can just wrap the (normally serial or only locally parallelized) process of performing a prediction with the trained model:

```
import pandas as pd
from pyspark.sql.types import IntegerType
from pyspark.sql.functions import pandas_udf

@pandas_udf(returnType=IntegerType())
def predict_pd_udf(*cols):
    X = pd.concat(cols, axis=1)
    return pd.Series(clf.predict(X))
```

4. Finally, we can apply this to the Spark `DataFrame` containing the data by passing in the appropriate inputs we needed for our function. In this case, we will pass in the column names of the features, of which there are 13:

```
col_names = ['_{}'.format(x) for x in range(13)]
df_pred = df.select('*', predict_pd_udf(*col_names))
```

Now, if you look at the results of this, you will see the following for the first few rows of the `df_pred` DataFrame:

	<code>_1</code>	<code>_2</code>	<code>_3</code>	<code>_4</code>	<code>_5</code>	<code>_6</code>	<code>_7</code>	<code>_8</code>	<code>_9</code>	<code>_10</code>	<code>_11</code>	<code>_12</code>	<code>_13</code>	class
14.23	1.71	2.43	15.6	127.0	2.8	3.06	0.28	2.29	5.64	1.04	3.92	1065.0	0	
13.2	1.78	2.14	11.2	100.0	2.65	2.76	0.26	1.28	4.38	1.05	3.4	1050.0	0	
13.16	2.36	2.67	18.6	101.0	2.8	3.24	0.3	2.81	5.68	1.03	3.17	1185.0	0	
14.37	1.95	2.5	16.8	113.0	3.85	3.49	0.24	2.18	7.8	0.86	3.45	1480.0	0	
13.24	2.59	2.87	21.0	118.0	2.8	2.69	0.39	1.82	4.32	1.04	2.93	735.0	2	
14.2	1.76	2.45	15.2	112.0	3.27	3.39	0.34	1.97	6.75	1.05	2.85	1450.0	0	
14.39	1.87	2.45	14.6	96.0	2.5	2.52	0.3	1.98	5.25	1.02	3.58	1290.0	0	

Figure 6.3: The result of applying a simple pandas UDF.

And that completes our whirlwind tour of UDFs and pandas UDFs in Spark, which allow us to take serial Python logic, such as data transformations or our ML models, and apply them in a manifestly parallel way.

In the next section, we will focus on how to set ourselves up to perform Spark-based computations in the cloud.

Spark on the cloud

As should be clear from the preceding discussion, writing and deploying PySpark-based ML solutions can be done on your laptop, but for you to see the benefits when working at scale, you must have an appropriately sized computing cluster to hand. Provisioning this sort of infrastructure can be a long and painful process but as discussed already in this book, a plethora of options for infrastructure are available from the main public cloud providers.

For Spark, AWS has a particularly nice solution called **AWS Elastic MapReduce (EMR)**, which is a managed big data platform that allows you to easily configure clusters of a few different flavors across the big data

ecosystem. In this book, we will focus on Spark-based solutions, so we will focus on creating and using clusters that have Spark tooling to hand.

In the next section, we will go through a concrete example of spinning up a Spark cluster on EMR and then deploying a simple Spark ML-based application onto it.

So, with that, let's explore Spark on the cloud with **AWS EMR**!

AWS EMR example

To understand how EMR works, we will continue in the practical vein that the rest of this book will follow and dive into an example. We will begin by learning how to create a brand-new cluster before discussing how to write and deploy our first PySpark ML solution to it. Let's get started:

1. First, navigate to the **EMR** page on AWS and find the **Create Cluster** button. You will then be brought to a page that allows you to input configuration data for your cluster. The first section is where you specify the name of the cluster and the applications you want to install on it. I will call this cluster `mlewp2-cluster`, use the latest EMR release available at the time of writing, 6.11.0, and select the **Spark** application bundle.
2. All other configurations can remain as default in this first section. This is shown in *Figure 6.4*:

Name

Amazon EMR release | [Info](#)

A release contains a set of applications which can be installed on your cluster.

[emr-6.11.0](#) ▾

Application bundle

Spark	Core Hadoop	Flink	HBase	Presto	Trino	Custom

Applications included in bundle

Spark 3.3.2 on Hadoop 3.3.3 YARN with and Zeppelin 0.10.1

AWS Glue Data Catalogue settings

Use the AWS Glue Data Catalog to provide an external metastore for your application.

Use for Spark table metadata

Operating system options | [Info](#)

Amazon Linux release

Customised Amazon Machine Image (AMI)

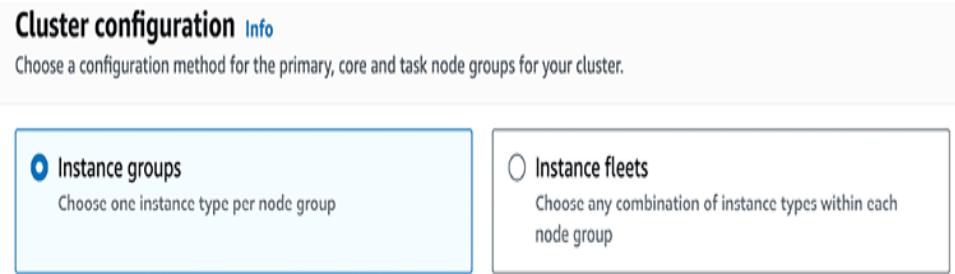
Automatically apply the latest Amazon Linux updates

Figure 6.4: Creating our EMR cluster with some default configurations.

3. Next comes the configuration of the compute used in the cluster. You can just use the defaults here again but it is important to understand what is going on. First, there is the selection of whether to use “instance groups” or “instance fleets,” which refers to the strategy deployed for scaling up your compute given some constraints you provide. Instance groups are simpler and define the specific servers you want to run for each node type, more on this in a second, and you

can choose between “on-demand” or “spot instances” for acquiring more servers if needed during the lifetime of the cluster. Instance fleets allow for a lot more complex acquisition strategies and for blends of different server instance types for each node type. For more information, read the AWS documentation to make sure you get a clear view of the different options,

<https://docs.aws.amazon.com/emr/index.xhtml>; we will proceed by using an instance group with default settings. Now, onto nodes. There are different nodes in an EMR cluster; primary, core and task. The primary node will run our YARN Resource Manager and will track job status and instance group health. The core nodes run some daemons and the Spark executors. Finally, the task nodes perform the actual distributed calculations. For now, let’s proceed with the defaults provided for the instance groups option, as shown for the **Primary** nodes in *Figure 6.5*.



Instance groups

Primary

Choose EC2 instance type

m5.xlarge

4 vCore 16 GiB memory EBS only storage

On-demand price: USD 0.214 per instance/hour

Lowest spot price: \$0.087 (eu-west-1a)

Actions ▾

Use multiple primary nodes

To improve cluster availability, use three primary nodes with the same configuration and bootstrap actions. You cannot use multiple primary nodes with instance fleets.

► Node configuration - optional

Figure 6.5: Compute configuration for our EMR cluster. We have selected the simpler “instance groups” option for the configuration and have gone with the server type defaults.

4. Next, we move onto defining the explicit cluster scaling behavior that we mentioned is used in the instance groups and instance fleet compute options in *step 2*. Again, select the defaults for now, but you can play around to make the cluster larger here in terms of numbers of nodes or you can define auto-scaling behaviour that will dynamically increase the cluster size upon larger workloads. *Figure 6.6* shows what this should look like.

Cluster scaling and provisioning option Info

The Amazon EMR console only supports EMR-managed scaling. To create a cluster with auto-scaling, use CLI or SDK.

Choose an option

Set cluster size manually

Use this option if you know your workload patterns in advance.

Use EMR-managed scaling

Monitor key workload metrics so that EMR can optimise the cluster size and optimise its resource utilisation.

Provisioning configuration

Set the size of your core and task instance groups. Amazon EMR attempts to provision this capacity when you launch your cluster.

Name	Instance type	Size	Use spot purchasing option
Core	m5.xlarge	<input type="text" value="1"/>	<input type="checkbox"/>
Task - 1	m5.xlarge	<input type="text" value="1"/>	<input type="checkbox"/>

Figure 6.6: The cluster provisioning and scaling strategy selection. Here we have gone with the defaults of a specific, small cluster size, but you can increase these values to have a bigger cluster or use the auto-scaling option to provide min and max size limits.

- Now, there is a **Networking** section, which is easier if you have already created some **virtual private clouds (VPCs)** and subnets for the other examples in the book; see *Chapter 5, Deployment Patterns and Tools* and the AWS documentation for more information. Just remember that VPCs are all about keeping the infrastructure you are provisioning isolated from other services in your own AWS account and even from the wider internet, so it's definitely good to become familiar with them and their application.
- For completeness, *Figure 6.7* shows the setup I used for this example.

The screenshot shows a user interface for networking configuration. At the top, there's a header 'Networking' with a 'Info' link. Below it, a section for 'Virtual private cloud (VPC)' has a 'Info' link and a text input field containing 'vpc-422d8f3b' with a 'Browse' button next to it. A 'Create VPC' button with a plus icon is also present. Below this, another section for 'Subnet' has an 'Info' link and a text input field containing 'subnet-23806e68' with a 'Browse' button next to it. A 'Create subnet' button with a plus icon is also present. A horizontal line separates these sections from a section labeled '▶ EC2 security groups (firewall)'.

Figure 6.7: The networking configuration requires the use of a VPC; it will automatically create a subnet for the cluster if there is not one selected.

7. We only have a couple more sections we need to input into to define our cluster. The next mandatory section is around cluster termination policies. I would always recommend having an automated teardown policy for infrastructure where possible as this helps to manage costs. There have been many stories across the industry of teams leaving unused servers running and racking up huge bills! *Figure 6.8* shows that we are using such an automated cluster termination policy where the cluster will terminate after 1 hour of not being utilized.

Cluster termination Info

Manually terminate cluster
 Terminate cluster after idle time (recommended)

Idle time
Enter the time until your cluster terminates.

Choose a time that is greater than 1 minute (00:01:00) and less than 7 days. The time is in hh:mm:ss (24-hour) format.

Terminate cluster after last step has been completed
You cannot edit this selection after you have created your cluster.

Use termination protection
Protect your EC2 instances from accidental termination.

Figure 6.8: Defining a cluster termination policy like this one is considered best practice and can help avoid unnecessary costs.

8. The final required section for completion is the definition of the appropriate **Identity and Access Management (IAM)** roles, which defines what accounts can access the resources we are creating. If you already have IAM roles that you are happy to reuse as your EMR service role, then you can do this; for this example, however, let's create a new service role specifically for this cluster. *Figure 6.9* shows that selecting the option to create a new role pre-populates the VPC, subnet, and security group with values matching what you have already selected through this process. You can add more to these. *Figure 6.10* shows that we can also select to create an "instance profile", which is just the name given to a service role that applies to all server instances in an EC2 cluster at launch.

Amazon EMR service role Info

The service role is an IAM role that Amazon EMR assumes to provision resources and perform service-level actions with other AWS services.

Choose an existing service role

Select a default service role or a custom role with IAM policies attached so that your cluster can interact with other AWS services.

Create a service role

Let Amazon EMR create a new service role so that you can grant and restrict access to resources in other AWS services.

Networking resources

We've already added the resources that you configured in the [Networking](#) section. Choose the VPC, subnet and security groups that the service role can access.

Virtual Private Cloud (VPC)

Choose one or more VPCs ▾

- X
vpc-422d8f3b

Subnet

Choose one or more subnets ▾

- X
subnet-23806e68

Security group

Choose one or more security groups ▾

ElasticMapReduce-Primary X
sg-051deff292498b3c4

ElasticMapReduce-Core X
sg-029348bc3ae9c3e02

Figure 6.9: Creating an AWS EMR service role.

EC2 instance profile for Amazon EMR

The instance profile assigns a role to every EC2 instance in a cluster. The instance profile must specify a role that can access the resources for your steps and bootstrap actions.

Choose an existing instance profile

Select a default role or a customised instance profile with IAM policies attached so that your cluster can interact with your resources in Amazon S3.

Create an instance profile

Let Amazon EMR create a new instance profile so that you can specify a customised set of resources for it to access in Amazon S3.

S3 bucket access [Info](#)

Specific S3 buckets or prefixes in your account [Info](#)

Choose the buckets or prefixes that you want this instance profile to access.

All S3 buckets in this account with read and write access

Grant the instance profile access to all buckets that have read and write access enabled in your account.

Figure 6.10: Creating an instance profile for the EC2 servers being used in this EMR cluster. An instance profile is just the name for the service role that is assigned to all cluster EC2 instances at spin-up time.

9. The sections discussed are all the mandatory ones required to create your cluster, but there are also some optional sections that I would like to briefly mention to guide your further exploration. There is the option to specify **steps**, which is where you can define shell scripts, JAR applications, or Spark applications to run in sequence. This means you can spin up your cluster with your applications ready to start processing data in the sequence you desire, rather than submitting jobs after the infrastructure deployment. There is a section on **Bootstrap actions**, which allows you to define custom installation or configuration steps that should run before any applications are installed or any data is processed on the EMR cluster. Cluster logs locations, tags, and some basic software considerations are also available for configuration. The final important point to mention is on the security configuration. *Figure 6.11* shows the options. Although we will deploy this cluster without specifying any EC2 key pair or

security configuration, it is crucially important that you understand the security requirements and norms of your organization if you want to run this cluster in production. Please consult your security or networking teams to ensure this is all in line with expectations and requirements. For now, we can leave it blank and proceed to create the cluster.

Security configuration and EC2 key pair – optional [Info](#)

Security configuration

Select your cluster encryption, authentication, authorisation and instance metadata service settings.

Create a security configuration



Browse

Create security configuration

Amazon EC2 key pair for SSH to the cluster [Info](#)

Enter a key name or choose Browse to select an Amazon EC2 ...

Browse

Create key pair

You haven't entered an EC2 key. If you're outside a VPN and want to enable SSH or use Hue SQL assistant with this cluster, you must enter an EC2 key.

Figure 6.11: The security configuration for the cluster shown here is optional but should be considered carefully if you aim to run the cluster in production.

10. Now that we have selected all of the mandatory options, click the **Create cluster** button to launch. Upon successful creation, you should see a review page like that shown in *Figure 6.12*. And that's it; you have now created your own Spark cluster in the cloud!

The screenshot shows the Amazon EMR console with a success message at the top: "Your cluster 'mlewp2-cluster' has been successfully created." Below this, the breadcrumb navigation shows: Amazon EMR > EMR on EC2: Clusters > mlewp2-cluster. The main content area displays the cluster details for 'mlewp2-cluster'. The cluster ID is j-2MFY5DTL16MYA. The Amazon EMR version is emr-6.11.0. The log destination in Amazon S3 is Logging not configured. The primary node public DNS is listed as '-'. The creation time is 1 July 2023 23:14 (UTC+01:00). The status is Starting. The cluster was updated less than a minute ago. There is an 'Actions' dropdown menu.

Cluster info	Applications	Cluster management	Status and time
Cluster ID j-2MFY5DTL16MYA	Amazon EMR version emr-6.11.0	Log destination in Amazon S3 Logging not configured	Status Starting
Cluster configuration Instance groups	Installed applications Spark 3.3.2, Zeppelin 0.10.1	Primary node public DNS -	Creation time 1 July 2023 23:14 (UTC+01:00)
Capacity 1 Primary 1 Core 1 Task			Elapsed time 0 seconds

Figure 6.12: EMR cluster creation review page shown upon successful launch.

After spinning up our EMR cluster, we want to be able to submit work to it. Here, we will adapt the example Spark ML pipeline we produced in *Chapter 3, From Model to Model Factory*, to analyze the banking dataset and submit this as a step to our newly created cluster. We will do this as a standalone single PySpark script that acts as the only step in our application, but it is easy to build on this to make far more complex applications:

1. First, we will take the code from *Chapter 3, From Model to Model Factory*, and perform some nice refactoring based on our discussions around good practices. We can more effectively modularize the code so that it contains a function that provides all our modeling steps (not all of the steps have been reproduced here, for brevity). We have also included a final step that writes the results of the modeling to a **parquet** file:

```
def model_bank_data(spark, input_path, output_path):
    data = spark.read.format("csv")\
        .option("sep", ";")\
        .option("inferSchema", "true")\
        .option("header", "true")\
        .load(input_path)
    data = data.withColumn('label', f.when((# ...
        data.write.format('parquet')\
            .mode('overwrite')\
            .save(output_path)
```

2. Building on this, we will wrap all of the main boilerplate code into a `main` function that can be called at the `if __name__ == "__main__":` entry point to the program:

```
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument(
        '--input_path', help='S3 bucket path
        Assume to be csv for this case.')
    parser.add_argument(
        '--output_path', help='S3 bucket path
        Assume to be parquet for this case')
    args = parser.parse_args()
    # Create spark context
    sc = SparkContext("local", "pipelines")
```

```
# Get spark session
spark = SparkSession\
    .builder\
    .appName('MLEWP Bank Data Classifier')
    .getOrCreate()
model_bank_data(
    spark,
    input_path=args.input_path,
    output_path=args.output_path
)
```

3. We put the preceding functions into a script called

`spark_example_emr.py`, which we will submit to our EMR cluster later:

```
import argparse
from pyspark.sql import SparkSession
from pyspark import SparkContext
from pyspark.sql import functions as f
from pyspark.mllib.evaluation import BinaryC
from pyspark.ml.feature import StandardScale
from pyspark.ml import Pipeline, PipelineMod
from pyspark.ml.classification import Logis

def model_bank_data(spark, input_path, output_
    ...
def main():
    ...
if __name__ == "__main__":
    main()
```

4. Now, to submit this script to the EMR cluster we have just created, we need to find out the cluster ID, which we can get from the AWS UI or by running the following command:

```
aws emr list-clusters --cluster-states WAITI
```

5. Then, we need to send the `spark_example_emr.py` script to S3 to be read in by the cluster. We can create an S3 bucket called `s3://mlewp-ch6-emr-examples` to store this and our other artifacts using either the CLI or the AWS console (see *Chapter 5, Deployment Patterns and Tools*). Once copied over, we are ready for the final step.
6. Now, we must submit the script using the following command, with `<CLUSTER_ID>` replaced with the ID of the cluster we just created. Note that if your cluster has been terminated due to the automated termination policy we set, you can't restart it but you can clone it. After a few minutes, the step should have completed and the outputs should have been written to the `results.parquet` file in the same S3 bucket:

```
aws emr add-steps\
--region eu-west-1 \
--cluster-id <CLUSTER_ID> \
--steps Type=Spark,Name="Spark Application S
Args=[--files,s3://mlewp-ch6-emr-examples/sp
--input_path,s3://mlewp-ch6-emr-examples/bar
--output_path,s3://mleip-emr-ml-simple/resul
```

And that is it – that is how we can start developing PySpark ML pipelines on the cloud using **AWS EMR**!

You will see that this previous process has worked successfully by navigating to the appropriate S3 bucket and confirming that the `results.parquet` file was created successfully; see *Figure 6.13*.

The screenshot shows the AWS S3 console interface for the bucket `mlewp-ch6-emr-examples`. The `Objects` tab is active, showing three items:

Name	Type	Last modified	Size	Storage class
<code>bank.csv</code>	csv	July 2, 2023, 07:06:07 (UTC+01:00)	450.7 KB	Standard
<code>results.parquet/</code>	Folder	-	-	-
<code>spark_example_emr.py</code>	py	July 2, 2023, 07:38:31 (UTC+01:00)	4.6 KB	Standard

Figure 6.13: Successful creation of the `results.parquet` file upon submission of the EMR script.

In the next section, we will explore another method of scaling up our solutions by using so-called serverless tools.

Spinning up serverless infrastructure

Whenever we do any ML or software engineering, we have to run the requisite tasks and computations on computers, often with appropriate networking, security, and other protocols and software already in place, which we have often referred to already as constituting our *infrastructure*. A big part of our infrastructure is the servers we use to run the actual computations. This might seem a bit strange, so let's start by talking about *serverless* infrastructure (how can there be such a thing?). This section will explain this concept and show you how to use it to scale out your ML solutions.

Serverless is a bit misleading as a term as it does not mean that no physical servers are running your programs. It does mean, however, that the programs you are running should not be thought of as being statically hosted on one machine, but as ephemeral instances on another layer on top of the underlying hardware.

The benefits of serverless tools for your ML solution include (but are not limited to) the following:

- **No servers:** Don't underestimate the savings in time and energy you can get by offloading infrastructure management to your cloud provider.
- **Simplified scaling:** It's usually very easy to define the scaling behavior of your serverless components by using clearly defined maximum instances, for example.
- **Low barrier to entry:** These components are usually extremely easy to set up and run, allowing you and your team members to focus on

writing high-quality code, logic, and models.

- **Natural integration points:** Serverless tools are often nice to use for handovers between other tools and components. Their ease of setup means you can be up and running with simple jobs that pass data or trigger other services in no time.
- **Simplified serving:** Some serverless tools are excellent for providing a serving layer to your ML models. The scaling and low barrier to entry mentioned previously mean you can quickly create a very scalable service that provides predictions upon request or upon being triggered by some other event.

One of the best and most widely used examples of serverless functionality is **AWS Lambda**, which allows us to write programs in a variety of languages with a simple web browser interface or through our usual development tools, and then have them run completely independently of any infrastructure that's been set up.

Lambda is an amazing low-entry-barrier solution to getting some code up and running and scaling it up. However, it is very much aimed at creating simple APIs that can be hit over an HTTP request. Deploying your ML model with Lambda is particularly useful if you are aiming for an event- or request-driven system.

To see this in action, let's build a basic system that takes incoming image data as an HTTP request with a JSON body and returns a similar message containing the classification of the data using a pre-built Scikit-Learn model. This walkthrough is based on the AWS example at

<https://aws.amazon.com/blogs/compute/deploying-machine-learning-models-with-serverless-templates/>.

For this, we can save a lot of time by leveraging templates already built and maintained as part of the AWS **Serverless Application Model (SAM)** framework (<https://aws.amazon.com/about-aws/what-s-new/2021/06/aws-sam-launches-machine-learning-inference-templates-for-aws-lambda/>).

To install the AWS SAM CLI on your relevant platform, follow the instructions at <https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-sam-cli-install.xhtml>.

Now, let's perform the following steps to set up a template serverless deployment for hosting and serving a ML model that classifies images of handwritten digits:

1. First, we must run the `sam init` command and select the AWS Quick Start Templates option:

```
Which template source would you like to use?  
 1 - AWS Quick Start Templates  
 2 - Custom Template Location  
Choice: 1
```

2. You will then be offered a choice of AWS Quick Start Application templates to use; select option 15, Machine Learning:

```
Choose an AWS Quick Start application template
1 - Hello World Example
2 - Data processing
3 - Hello World Example with Powertools
4 - Multi-step workflow
5 - Scheduled task
6 - Standalone function
7 - Serverless API
8 - Infrastructure event management
9 - Lambda Response Streaming
10 - Serverless Connector Hello World Example
11 - Multi-step workflow with Connectors
12 - Full Stack
13 - Lambda EFS example
14 - DynamoDB Example
15 - Machine Learning
```

Template:



3. Next are the options for the Python runtime you want to use; in line with the rest of the book, we will use the Python 3.10 runtime:

```
Which runtime would you like to use?
1 - python3.9
2 - python3.8
3 - python3.10
```

Runtime:

4. At the time of writing, the SAM CLI will then auto-select some options based on these choices, first the package type and then the

dependency manager. You will then be asked to confirm the ML starter template you want to use. For this example, select **XGBoost Machine Learning API**:

```
Based on your selections, the only Package type is XGBoost Machine Learning API
We will proceed to selecting the Package type
Based on your selections, the only dependency is xgboost
We will proceed copying the template using package manager
Select your starter template
  1 - PyTorch Machine Learning Inference API
  2 - Scikit-learn Machine Learning Inference API
  3 - Tensorflow Machine Learning Inference API
  4 - XGBoost Machine Learning Inference API
Template: 4
```

5. The SAM CLI then helpfully asks about some options for configuring request tracing and monitoring; you can select yes or no depending on your own preferences. I have selected no for the purposes of this example. You can then give the solution a name; here I have gone with **mlewp-sam-ml-api**:

```
Would you like to enable X-Ray tracing on this project? [y/N]: N
Would you like to enable monitoring using CloudWatch Metrics? [y/N]: N
For more info, please view: https://docs.aws.amazon.com/serverless/latest/developerguide/xray-tracing.html
Project name [sam-app]: mlewp-sam-ml-api
Cloning from https://github.com/aws/aws-sam-cli-template-xgboost
```

6. Finally, your command line will provide some helpful information about the installation and next steps:

```
-----  
Generating application:  
-----  
Name: mlewp-sam-ml-api  
Base Image: amazon/python3.10-base  
Architectures: x86_64  
Dependency Manager: pip  
Output Directory: .  
Configuration file: mlewp-sam-ml-api/sam.toml  
Next steps can be found in the README file.  
Commands you can use next  
=====  
[*] Create pipeline: cd mlewp-sam-ml-api &&  
[*] Validate SAM template: cd mlewp-sam-ml-api && sam validate  
[*] Test Function in the Cloud: cd mlewp-sam-ml-api && sam invoke function
```

Note that the preceding steps have created a template for an XGBoost-based system that classifies handwritten digits. For other applications and project use cases, you will need to adapt the source code of the template as you require. If you want to deploy this example, follow the next few steps:

1. First, we must build the application container provided with the template. First, navigate to the top directory of your project, you can see the directory structure should be something like below. I have used the `tree` command to provide a clean outline of the directory structure in the command line:

```
cd mlewp-sam-ml-api
ls
tree
├── README.md
├── __init__.py
└── app
    ├── Dockerfile
    ├── __init__.py
    ├── app.py
    ├── model
    └── requirements.txt
├── events
│   └── event.json
└── samconfig.toml
└── template.yaml
3 directories, 10 files
```

2. Now that we are in the top directory, we can run the `build` command. This requires that Docker is running in the background on your machine:

```
 sam build
```

3. Upon a successful build, you should receive a success message similar to the following in your terminal:

```
Build Succeeded
Built Artifacts  : .aws-sam/build
Built Template  : .aws-sam/build/template.y
```

```
Commands you can use next
```

```
=====
[*] Validate SAM template: sam validate
[*] Invoke Function: sam local invoke
[*] Test Function in the Cloud: sam sync --s
[*] Deploy: sam deploy --guided
```

4. Now, we can test the service locally to ensure that everything is working well with the mock data that's supplied with the repository. This uses a JSON file that encodes a basic image and runs the inference step for the service. If this has worked, you will see an output that looks something like the following for your service:

```
sam local invoke --event events/event.json
Invoking Container created from inferencefun
Building image.....
Using local image: inferencefunction:rapid-x
START RequestId: de4a2fe1-be86-40b7-a59d-151
END RequestId: de4a2fe1-be86-40b7-a59d-151aa
REPORT RequestId: de4a2fe1-be86-40b7-a59d-15
{"statusCode": 200, "body": "{\"predicted_la

```

5. In a real project, you would edit the source code for the solution in the `app.py` and other files as required before deploying up to the cloud. We will do this using the SAM CLI , with the understanding that if you want to automate this process, you can use the CI/CD processes and tools we discussed in several places in this book, especially in *Chapter 4, Packaging Up*. To deploy, you can use the guided deployment

wizard with the CLI by running the `deploy` command, which will return the below output:

```
 sam deploy --guided
Configuring SAM deploy
=====
    Looking for config file [samconfig.toml]
    Reading default arguments : Success
    Setting default arguments for 'sam deplo
=====
Stack Name [mlewp-sam-ml-api]:
```

6. We then have to configure the application for each of the provided elements. I have selected the defaults in most cases, but you can refer to the AWS documentation and make the choices most relevant to your project:

```
Configuring SAM deploy
=====
    Looking for config file [samconfig.toml]
    Reading default arguments : Success
    Setting default arguments for 'sam deplo
=====
Stack Name [mlewp-sam-ml-api]:
AWS Region [eu-west-2]:
#Shows you resources changes to be deplo
Confirm changes before deploy [Y/n]: y
#SAM needs permission to be able to crea
Allow SAM CLI IAM role creation [Y/n]: y
```

```
#Preserves the state of previously provi  
Disable rollback [y/N]: y  
InferenceFunction has no authentication.  
Save arguments to configuration file [Y/  
SAM configuration file [samconfig.toml]:  
SAM configuration environment [default]:
```

7. The previous step will generate a lot of data in the terminal; you can monitor this to see if there are any errors or issues. If the deployment was successful, then you should see some final metadata about the application that looks like this:

```
CloudFormation outputs from deployed stack  
-----  
Outputs  
-----  
Key           InferenceApi  
Description    API Gateway endpoint URL  
Value          https://8qg87m9380.execute.  
amazonaws.com/Prod/classify_digit/  
Key           InferenceFunctionIamRole  
Description    Implicit IAM Role create  
Value          arn:aws:iam::50897291134  
Key           InferenceFunction  
Description    Inference Lambda Function  
Value          arn:aws:lambda:eu-west-2
```

8. As a quick test to confirm the cloud-hosted solution is working, we can use a tool such as Postman to hit our shiny new ML API. Simply copy the **InferenceApi** URL from the output screen from *step 8* as the destination for the request, select **POST** for the request type, and then choose **binary** as the body type. Note that if you need to get the inference URL, again you can run the **sam list endpoints --output json** command in your terminal. Then, you can choose an image of a handwritten digit, or any other image for that matter, to send up to the API . You can do this in Postman either by selecting the **binary** body option and attaching an image file or you can copy in the encoded string of an image. In *Figure 6.14*, I have used the encoded string for the **body** key-value pair in the **events/event.json** file we used to test the function locally:

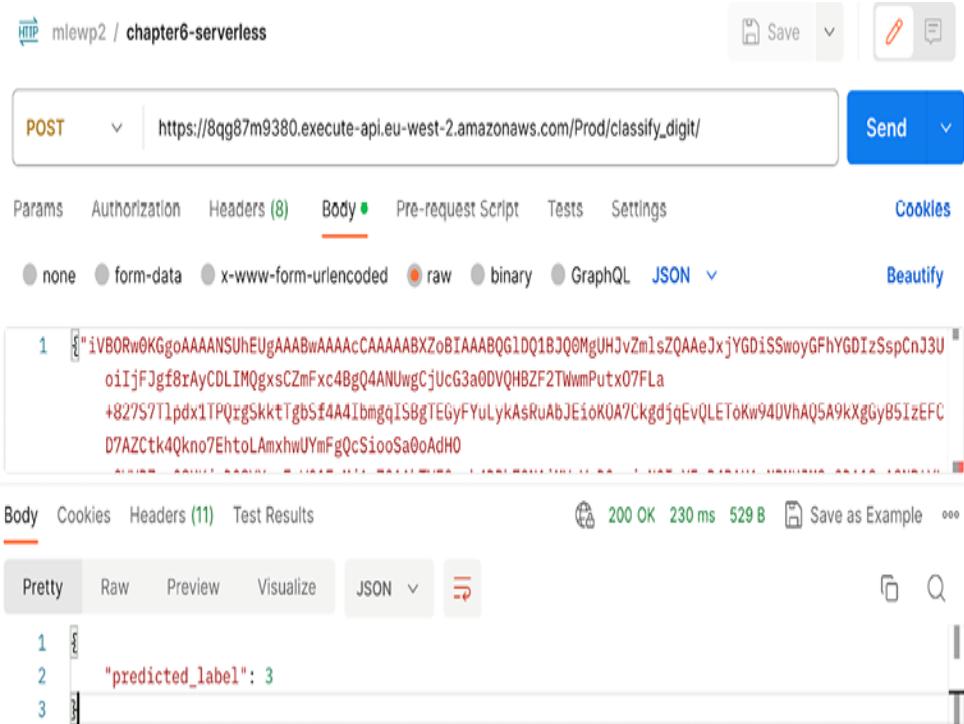


Figure 6.14: Calling our serverless ML endpoint with Postman. This uses an encoded example image as the body of the request that is provided with the SAM XGBoost ML API template.

9. You can also test this more programmatically with a `curl` command like the following – just replace the encoded binary string of the image with the appropriate values, or indeed edit the command to point to a data binary if you wish, and you are good to go:

```
curl --location --request POST 'https://8qg8  
--header 'Content-Type: raw/json' \  
--data '<ENCODED_IMAGE_STRING>'
```

In this step and *step 9*, the body of the response from the Lambda function is as follows:

```
{  
    "predicted_label": 3  
}
```

And that's it – we have just built and deployed a simple serverless ML inference service on AWS!

In the next section, we will touch upon the final way of scaling our solutions that we will discuss in this chapter, which is using Kubernetes (K8s) and Kubeflow to horizontally scale containerized applications.

Containerizing at scale with Kubernetes

We have already covered how to use containers for building and deploying our ML solutions. The next step is understanding how to orchestrate and

manage several containers to deploy and run applications at scale. This is where the open source tool **Kubernetes (K8s)** comes in.

K8s is an extremely powerful tool that provides a variety of different functionalities that help us create and manage very scalable containerized applications, including (but not limited to) the following:

- **Load Balancing:** K8s will manage routing incoming traffic to your containers for you so that the load is split evenly.
- **Horizontal Scaling:** K8s provides simple interfaces so that you can control the number of container instances you have at any one time, allowing you to scale massively if needed.
- **Self Healing:** There is built-in management for replacing or rescheduling components that are not passing health checks.
- **Automated Rollbacks:** K8s stores the history of your system so that you can revert to a previous working version if something goes wrong.

All of these features help ensure that your deployed solutions are robust and able to perform as required under all circumstances.

K8s is designed to ensure the preceding features are embedded from the ground up by using a microservice architecture, with a control plane interacting with nodes (servers), each of which host pods (one or more containers) that run the components of your application.

The key thing that K8s gives you is the ability to scale your application based on load by creating replicas of the base solution. This is extremely useful if you are building services with API endpoints that could feasibly face surges in demand at different times. To learn about some of the ways you can do this, see

<https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#scaling-a-deployment>:

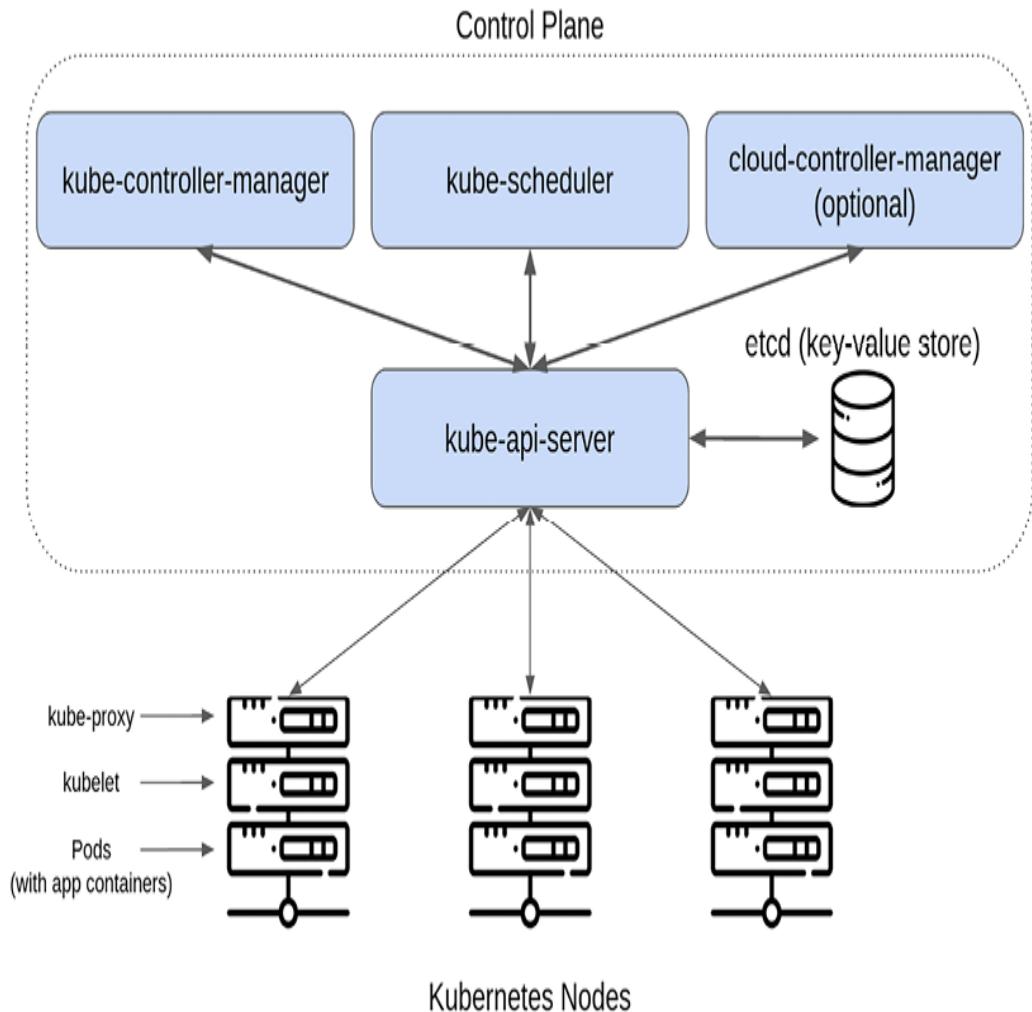


Figure 6.15: The K8s architecture.

But what about ML? In this case, we can look to a newer piece of the K8s ecosystem: **Kubeflow**, which we learned how to use in *Chapter 5, Deployment Patterns and Tools*.

Kubeflow styles itself as the *ML toolkit for K8s* (<https://www.kubeflow.org/>), so as ML engineers, it makes sense

for us to be aware of this rapidly developing solution. This is a very exciting tool and an active area of development.

The concept of horizontal scaling for K8s generally still applies here, but Kubeflow provides some standardized tools for converting the pipelines you build into standard K8s resources, which can then be managed and resourced in the ways described previously. This can help reduce *boilerplate* and lets us, as ML engineers, focus on building our modelling logic rather than setting up the infrastructure. We leveraged this when we built some example pipelines in *Chapter 5*.

We will explore Kubernetes in far more detail in *Chapter 8, Building an Example ML Microservice*, where we use it to scale out our own wrapped ML model in a REST API. This will complement nicely the work we have done in this chapter on higher-level abstractions that can be used for scaling out, especially in the *Spinning up serverless infrastructure* section. We will only touch on K8s and Kubeflow very briefly here, to make sure you are aware of these tools for your exploration. For more details on K8s and Kubeflow, consult the documentation. I would also recommend another Packt title called *Kubernetes in Production Best Practices* by Aly Saleh and Murat Karslioglu.

Now, we will move on and discuss another very powerful toolkit for scaling out compute-intensive Python workloads, which has now become extremely popular across the ML engineering community and been used by organizations such as Uber, Amazon and even used by OpenAI for training their large language **Generative Pre-trained Transformer (GPT)** models, which we discuss at length in *Chapter 7, Deep Learning, Generative AI, and LLM Ops*. Let's meet **Ray**.

Scaling with Ray

Ray is a Python native distributed computing framework that was specifically designed to help ML engineers meet the needs of massive data and massively scalable ML systems. Ray has an ethos of making scalable compute available to every ML developer, and in doing this in a way such that you can run anywhere by abstracting out all interactions with underlying infrastructure. One of the unique features of Ray that is particularly interesting is that it has a distributed scheduler, rather than a scheduler or DAG creation mechanism that runs in a central process, like in Spark. At its core, Ray has been developed with compute-intensive tasks such as ML model training in mind from the beginning, which is slightly different from Apache Spark, which has data intensity in mind. You can therefore think about this in a simplified manner: if you need to process lots of data a couple of times, Spark; if you need to process one piece of data lots of times, Ray. This is just a heuristic so should not be followed strictly, but hopefully it gives you a helpful rule of thumb. As an example, if you need to transform millions and millions of rows of data in a large batch process, then it makes sense to use Spark, but if you want to train an ML model on the same data, including hyperparameter tuning, then Ray may make a lot of sense.

The two tools can be used together quite effectively, with Spark transforming the feature set before feeding this into a Ray workload for ML training. This is taken care of in particular by the **Ray AI Runtime (AIR)**, which has a series of different libraries to help scale different pieces of an ML solution. These include:

- **Ray Data:** Focused on providing data pre-processing and transformation primitives.

- **Ray Train:** Facilitates large model training.
- **Ray Tune:** Helps with scalable hyperparameter training.
- **Ray RLib:** Supports methods for the development of reinforcement learning models.
- **Ray Batch Predictor:** For batch inference.
- **Ray Serving:** For real-time inference.

The AIR framework provides a unified API through which to interact with all of these capabilities and nicely integrates with a huge amount of the standard ML ecosystem that you will be used to, and that we have leveraged in this book.

Ray AIR in Ray 2.0

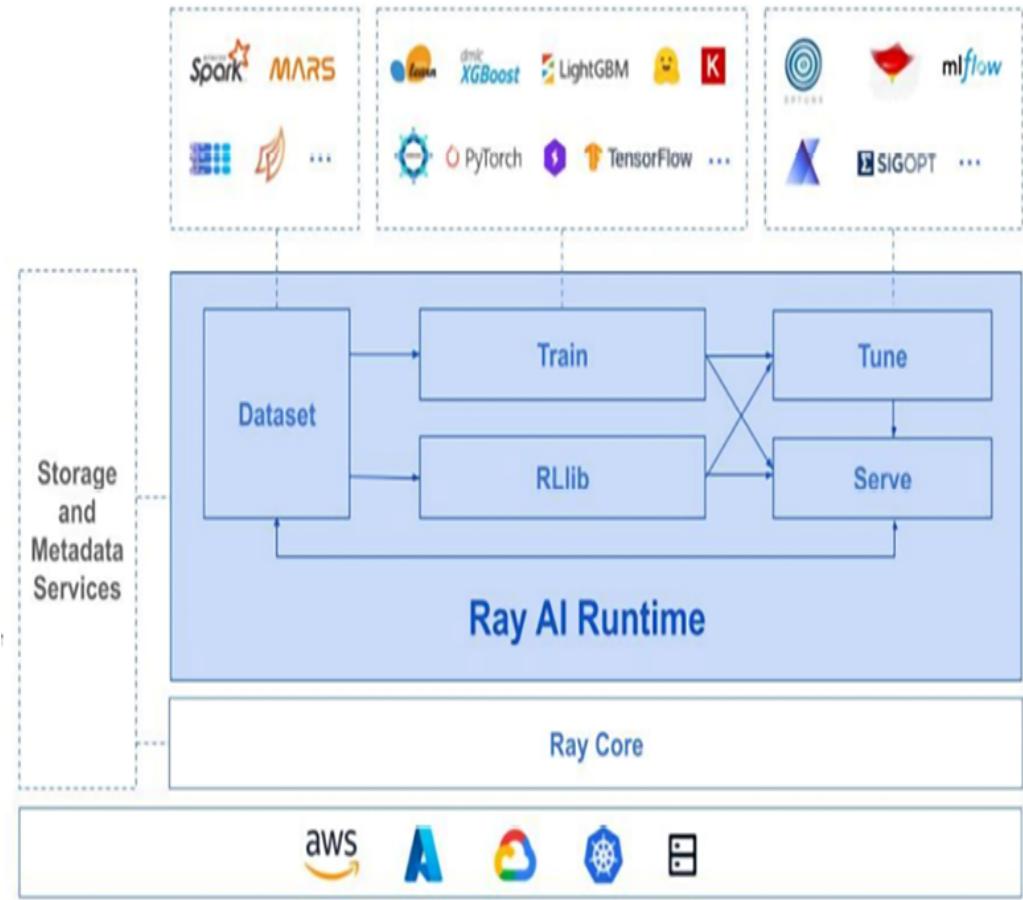


Figure 6.16: The Ray AI runtime, from a presentation by Jules Damji from Anyscale:
<https://microsites.databricks.com/sites/default/files/2022-07/Scaling%20AI%20Workloads%20with%20the%20Ray%20Ecosystem.pdf>. Reproduced with permission.

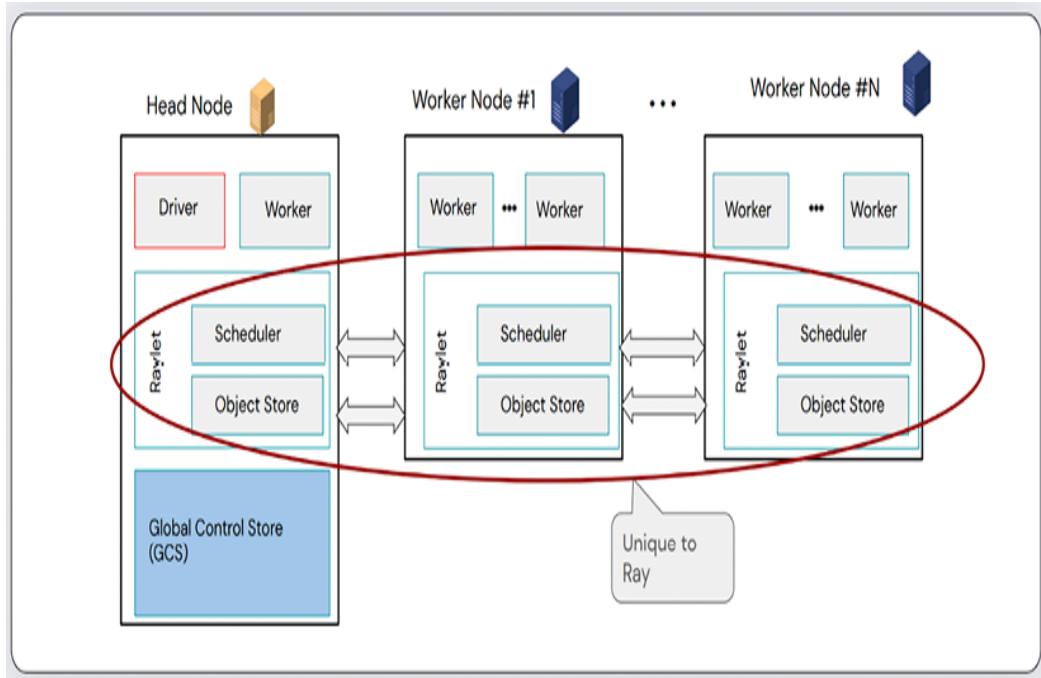


Figure 6.17: The Ray architecture including the Raylet scheduler. From a presentation by Jules Damji: <https://microsites.databricks.com/sites/default/files/2022-07/Scaling%20AI%20Workloads%20with%20the%20Ray%20Ecosystem.pdf>. Reproduced with permission.

The Ray Core API has a series of different objects that you leverage when using Ray in order to distribute your solution. The first is tasks, which are asynchronous items of work for the system to perform. To define a task, you can take a Python function like:

```
def add(int: x, int: y) -> int:
    return x+y
```

And then add the `@remote` decorator and then use the `.remote()` syntax in order to submit this task to the cluster. This is not a blocking function so will just return an ID that Ray uses to refer to the task in later computation steps

([https://www.youtube.com/live/XME90SGL6Vs?
feature=share&t=832](https://www.youtube.com/live/XME90SGL6Vs?feature=share&t=832)):

```
import ray

@remote
def add(int: x, int: y) -> int:
    return x+y
add.remote()
```

In the same vein, the Ray API can extend the same concepts to classes as well; in this case, these are called **Actors**:

```
import ray

@ray.remote
class Counter(object):
    def __init__(self):
        self.value = 0
    def increment(self):
        self.value += 1
        return self.value
    def get_counter(self):
        return self.value
# Create an actor from this class.
counter = Counter.remote()
```

Finally, Ray also has a distributed immutable object store. This is a smart way to have one shared data store across all the nodes of the cluster without

shifting lots of data around and using up bandwidth. You can write to the object store with the following syntax:

```
import ray

numerical_array = np.arange(1,10e7)
obj_numerical_array = ray.put(numerical_array)
new_numerical_array = 0.5*ray.get(obj_numerical
```



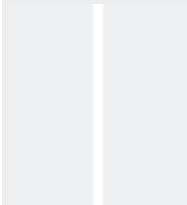
IMPORTANT NOTE

An Actor in this context is a service or stateful worker, a concept used in other distributed frameworks like Akka, which runs on the JVM and has bindings to Java and Scala.

Getting started with Ray for ML

To get started you can install Ray with AI Runtime, as well as some the hyperparameter optimization package, the central dashboard and a Ray enhanced XGBoost implementation, by running:

```
pip install "ray[air, tune, dashboard]"
pip install xgboost
pip install xgboost_ray
```



IMPORTANT NOTE



a reminder here that whenever you see `pip install` in this book, you can also use Poetry as outlined in *Chapter 4, Packaging Up*. So, in this case, you would have the following commands after running `poetry new project_name`:

```
poetry add "ray[air, tune, dashboard]"
poetry add xgboost
poetry add pytorch
```

Let's start by looking at Ray Train, which provides an API to a series of `Trainer` objects that helps facilitate distributed training. At the time of writing, Ray 2.3.0 supports trainers across a variety of different frameworks including:

- **Deep learning:** Horovod, Tensorflow and PyTorch.
- **Tree based:** LightGBM and XGBoost.
- **Other:** Scikit-learn, HuggingFace, and Ray's reinforcement learning library RLlib.

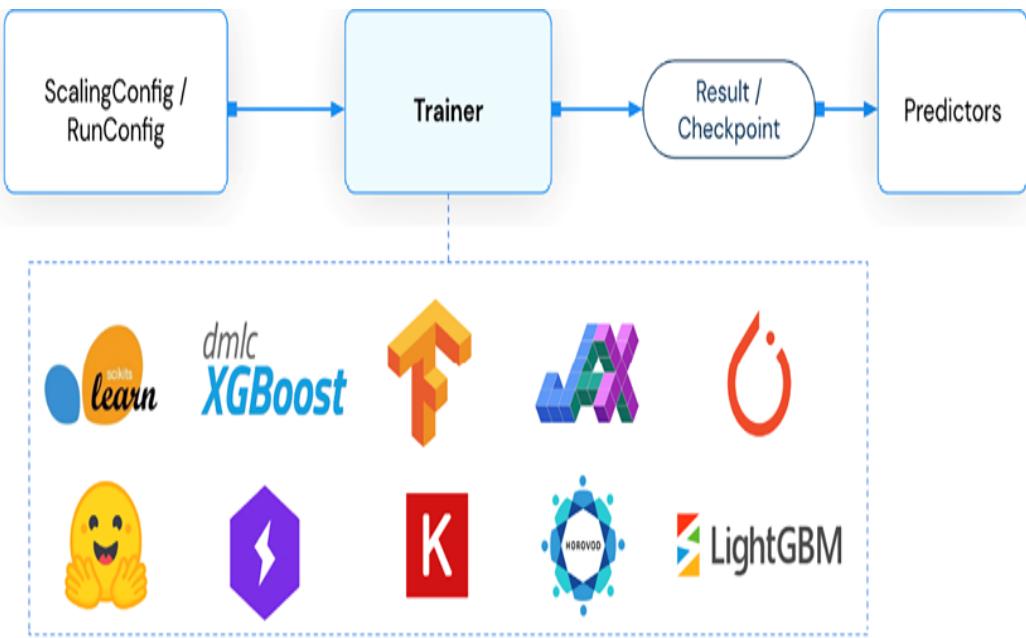


Figure 6.18: Ray Trainers as shown in the Ray docs at <https://docs.ray.io/en/latest/train/train.xhtml>.

We will first look at a tree-based learner example using XGBoost. Open up a script and begin adding to it; in the repo, this is called `getting_started_with_ray.py`. What follows is based on an introductory example given in the Ray documentation. First, we can use Ray to download one of the standard datasets; we could also have used `sklearn.datasets` or another source if we wanted to, like we have done elsewhere in the book:

```

import ray

dataset = ray.data.read_csv("s3://anonymous@air
                            cancer.csv")
train_dataset, valid_dataset = dataset.train_te
test_dataset = valid_dataset.drop_columns(cols=

```

Note that here we use the `ray.data.read_csv()` method, which returns a `PyArrow` dataset. The Ray API has methods for reading from other data formats as well such as JSON or Parquet, as well as from databases like MongoDB or your own custom data sources.

Next, we will define a preprocessing step that will standardize the features we want to use; for more information on feature engineering, you can check out *Chapter 3, From Model to Model Factory*:

```
from ray.data.preprocessors import StandardScal  
preprocessor = StandardScaler(columns=[ "mean ra
```

Then is the fun part where we define the `Trainer` object for the XGBoost model. This has several different parameters and inputs we will need to define shortly:

```
from ray.air.config import ScalingConfig  
from ray.train.xgboost import XGBoostTrainer  
  
trainer = XGBoostTrainer(  
    scaling_config=ScalingConfig(...),  
    label_column="target",  
    num_boost_round=20,  
    params={...},  
    datasets={"train": train_dataset, "valid":  
        preprocessor=preprocessor,  
    })  
result = trainer.fit()
```

You'll then see something like that shown in *Figure 6.19* as output if you run this code in a Jupyter notebook or Python script.

Tune Status		System Info						
Current time:		Using FIFO scheduling algorithm.						
Resources requested:		0/8 CPUs, 0/0 GPUs, 0.0/2.36 GiB heap, 0.0/1.18 GiB objects						
Running for:		00:00:07.47						
Memory:		5.2/8.0 GiB						
Trial Status								
Trial name	status	loc	iter	total time (s)	train-logloss	train-error	valid-logloss	
XGBoostTrainer_6ecab_00000	TERMINATED	127.0.0.1:1713	21	6.22244	0.0184957	0	0.0897979	
<pre>(XGBoostTrainer pid=1713) 2023-03-14 20:33:11,212 INFO bulk_executor.py:39 -- Executing DAG InputDataBuffer[Input] -> AllToAllOperator[aggregate] (XGBoostTrainer pid=1713) 2023-03-14 20:33:12,242 INFO bulk_executor.py:39 -- Executing DAG InputDataBuffer[Input] -> TaskPoolMapOperator[StandardScaler] (XGBoostTrainer pid=1713) 2023-03-14 20:33:12,264 INFO bulk_executor.py:39 -- Executing DAG InputDataBuffer[Input] -> TaskPoolMapOperator[StandardScaler] (XGBoostTrainer pid=1713) 2023-03-14 20:33:12,289 INFO bulk_executor.py:39 -- Executing DAG InputDataBuffer[Input] -> AllToAllOperator[repartition] (XGBoostTrainer pid=1713) 2023-03-14 20:33:12,325 INFO bulk_executor.py:39 -- Executing DAG InputDataBuffer[Input] -> AllToAllOperator[repartition] (XGBoostTrainer pid=1713) 2023-03-14 20:33:15,807 INFO tracker.py:218 -- start listen on 127.0.0.1:49620 (_RemoteRayXGBoostActor pid=1725) [20:33:15] task [xgboost.ray]:5221498736 got new rank 1 (_RemoteRayXGBoostActor pid=1724) [20:33:15] task [xgboost.ray]:5747720096 got new rank 0 (XGBoostTrainer pid=1713) 2023-03-14 20:33:15,821 INFO tracker.py:382 -- @tracker All of 2 nodes getting started (XGBoostTrainer pid=1713) 2023-03-14 20:33:16,247 INFO tracker.py:388 -- @tracker All nodes finishes job</pre>								
Trial Progress								
Trial name	date done	episodes_total		experiment_id	experiment_tag	hostname	iterations	
XGBoostTrainer_6ecab_00000	2023-03-14_20:33:17	True		2df66fa1a6b14717bed8b31470d386d4		0	Andrews-MacBook-Pro.local	
2023-03-14 20:33:17,439 INFO tune.py:798 -- Total run time: 8.19 seconds (7.47 seconds for the tuning loop).								

Figure 6.19: Output from parallel training of an XGBoost model using Ray.

The `result` object contains tons of useful information; one of the attributes of it is called `metrics` and you can print this to reveal details about the end state of the run. Execute `print(result.metrics)` and you will see something like the following:

```
{'train-logloss': 0.01849572773292735,
'train-error': 0.0, 'valid-logloss': 0.08979789
'valid-error': 0.04117647058823529,
'time_this_iter_s': 0.019704103469848633,
'should_checkpoint': True,
'done': True,
'timesteps_total': None,
'episodes_total': None,
'training_iteration': 21,
'trial_id': '6ecab_00000',
'experiment_id': '2df66fa1a6b14717bed8b31470d38
'date': '2023-03-14_20-33-17',
'timestamp': 1678825997,
'time_total_s': 6.222438812255859,
'pid': 1713,
'hostname': 'Andrews-MacBook-Pro.local',
'node_ip': '127.0.0.1',
'config': {},
'time_since_restore': 6.222438812255859,
'timesteps_since_restore': 0,
'iterations_since_restore': 21,
'warmup_time': 0.003551006317138672, 'experimen
```

In the instantiation of the `XGBoostTrainer`, we defined some important scaling information that was omitted in the previous example; here it is:

```
scaling_config=ScalingConfig(
    num_workers=2,
    use_gpu=False,
```

```
    _max_cpu_fraction_per_node=0.9,  
)
```

The `num_workers` parameter tells Ray how many actors to launch, with each actor by default getting one CPU. The `use_gpu` flag is set to false since we are not using GPU acceleration here. Finally, by setting the `_max_cpu_fraction_per_node` parameter to `0.9` we have left some spare capacity on each CPU, which can be used for other operations.

In the previous example, there were also some XGBoost specific parameters we supplied:

```
params={  
    "objective": "binary:logistic",  
    "eval_metric": ["logloss", "error"],  
}
```

If you wanted to use GPU acceleration for the XGBoost training you would add `tree_method: gpu_hist` as a key-value pair in this `params` dictionary.

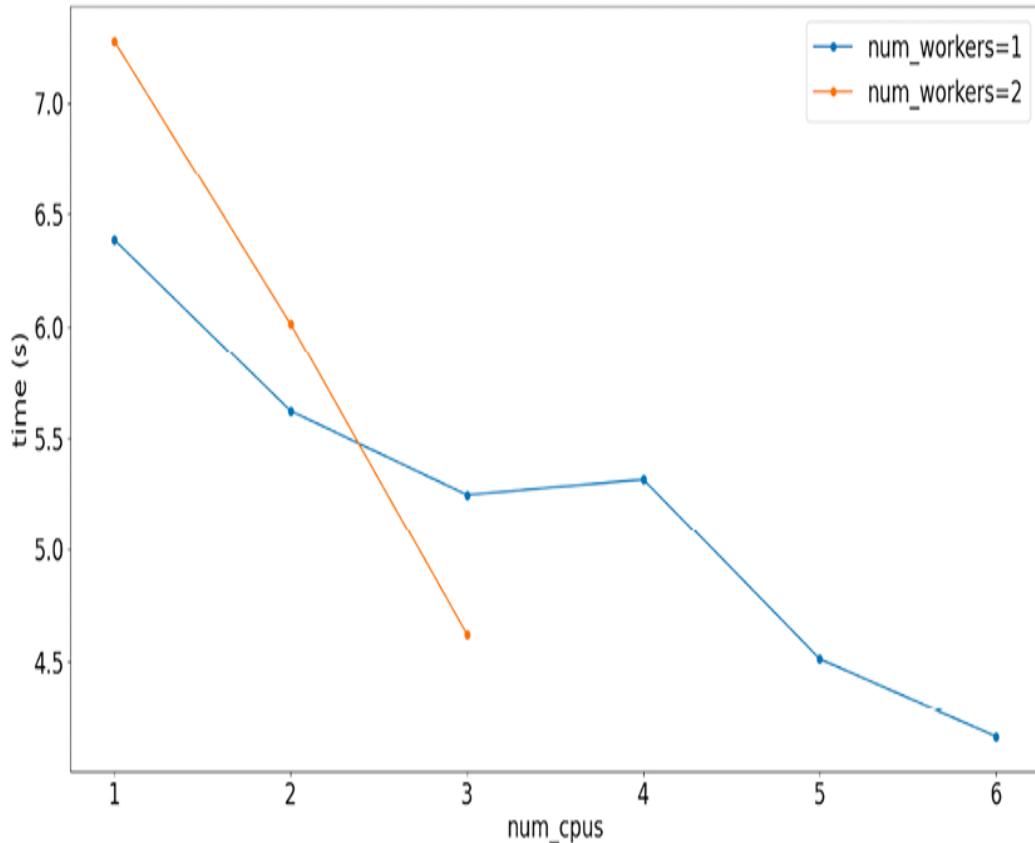


Figure 6.20: A few experiments show how changing the number of workers and CPUs available per worker results in different XGBoost training times on the author’s laptop (an 8 core Macbook Pro).

We will now discuss briefly how you can scale compute with Ray when working in environments other than your local machine.

Scaling your compute for Ray

The examples we’ve seen so far use a local Ray cluster that is automatically set up on the first call to the Ray API. This local cluster grabs all the available CPUs on your machine and makes them available to execute work. Obviously, this will only get you so far. The next stage is to work with clusters that can scale to far larger numbers of available workers in order to get more speedup. You have a few options if you want to do this:

- **On the cloud:** Ray provides the ability to deploy on to Google Cloud Platform and AWS resources, with Azure deployments handled by a community maintained solution. For more information on deploying and running Ray on AWS, you can check out its online documentation.
- **Using Kubernetes:** We have already met Kubeflow in *Chapter 5, Deployment Patterns and Tools*, which is used to build Kubernetes enabled ML pipelines. And we have also discussed Kubernetes in the Containerizing at Scale with Kubernetes section in this chapter.. As mentioned there, Kubernetes is a container orchestration toolkit designed to create massively scalable solutions based on containers. If you want to work with Ray on Kubernetes, you can use the **KubeRay** project, <https://ray-project.github.io/kuberay/>.

The setup of Ray on either the cloud or Kubernetes mainly involves defining the cluster configuration and its scaling behaviour. Once you have done this, the beauty of Ray is that scaling your solution is as simple as editing the `ScalingConfig` object we used in the previous example, and you can keep all your other code the same. So, for example, if you have a 20-node CPU cluster, you could simply change the definition to the following and run it as before:

```
scaling_config=ScalingConfig(  
    num_workers=20,  
    use_gpu=False,  
    _max_cpu_fraction_per_node=0.9,  
)
```

Scaling your serving layer with Ray

We have discussed the ways you can use Ray to distributed ML training jobs but now let's have a look at how you can use Ray to help you scale your application layer. As mentioned before, Ray AIR provides some nice functionality for this that is badged under **Ray Serve**.

Ray Serve is a framework-agnostic library that helps you easily define ML endpoints based on your models. Like with the rest of the Ray API that we have interacted with, it has been built to provide easy interoperability and access to scaling without large development overheads.

Building on the examples from the previous few sections, let us assume we have trained a model, stored it in our appropriate registry, such as MLflow, and we have retrieved this model and have it in memory.

In Ray Serve, we create **deployments** by using the `@ray.serve.deployments` decorator. These contain the logic we wish to use to process incoming API requests, including through any ML models we have built. As an example, let's build a simple wrapper class that uses an XGBoost model like the one we worked with in the previous example to make a prediction based on some pre-processed feature data that comes in via the request object. First, the Ray documentation encourages the use of the Starlette requests library:

```
from starlette.requests import Request
import ray
from ray import serve
```

Next we can define the simple class and use the `serve` decorator to define the service. I will assume that logic for pulling from MLflow or any other

model storage location is wrapped into the utility function `get_model` in the following code block:

```
@serve.deployment
class Classifier:
    def __init__(self):
        self.model = get_model()
    async def __call__(self, http_request: Request):
        request_payload = await http_request.json()
        input_vector = [
            request_payload["mean_radius"],
            request_payload["mean_texture"]
        ]
        classification = self.model.predict([in])
    return {"result": classification}
```

You can then deploy this across an existing Ray cluster.

This concludes our introduction to Ray. We will now finish with a final discussion on *designing systems at scale* and then a summary of everything we have learned.

Designing systems at scale

To build on the ideas presented in *Chapter 5, Deployment Patterns and Tools*, and in this chapter, we should now consider some of the ways in which the scaling capabilities we have discussed can be employed to maximum effect in your ML engineering projects.

The whole idea of scaling should be thought of in terms of providing an increase in the throughput of analyses or inferences or ultimate size of data that can be processed. There is no real difference in the kind of analyses or solution you can develop, at least in most cases. This means that applying scaling tools and techniques successfully is more dependent on selecting the correct processes that will benefit from them, even when we include any overheads that come from using these tools. That is what we will discuss now in this section, so that you have a few guiding principles to revisit when it comes to making your own scaling decisions.

As discussed in several places throughout this book, the pipelines you develop for your ML projects will usually have to have stages that cover the following tasks:

- Ingestion/pre-processing
- Feature engineering (if different from above)
- Model training
- Model inference
- Application layer

Parallelization or distribution can help in many of these steps but usually in some different ways. For ingestion/pre-processing, if you are operating in a large scheduled batch setting, then the ability to scale to larger datasets in a distributed manner is going to be of huge benefit. In this case, the use of Apache Spark will make sense. For feature engineering, similarly the main bottleneck is in processing large amounts of data once as we perform the transformations, so again Spark is useful for this. The compute-intensive steps for training ML models that we discussed in detail in *Chapter 3, From Model to Model Factory*, are very amenable to frameworks that are optimized for this intensive computation, irrespective of the data size. This

is where Ray comes in as discussed in the previous sections. Ray will mean that you can also neatly parallelize your hyperparameter tuning if you need to do that too. Note that you could run these steps in Spark as well but Ray's low task overheads and its distributed state management mean that it is particularly amenable to splitting up these compute-intensive tasks. Spark on the other hand has centralized state and schedule management. Finally, when it comes to the inference and application layers, where we produce and surface the results of the ML model, we need to think about the requirements for the specific use case. As an example, if you want to serve your model as a REST API endpoint, we showed in the previous section how Ray's distribution model and API can help facilitate this very easily, but this would not make sense to do in Spark. If, however, the model results are to be produced in large batches, then Spark or Ray may make sense. Also, as alluded to in the feature engineering and ingestion steps, if the end result should be transformed in large batches as well, perhaps into a specific data model such as a star schema, then performing that transformation in Spark may again make sense due to the data scale requirements of this task.

Let's make this a bit more concrete by considering a potential example taken from industry. Many organizations with a retail element will analyze transactions and customer data in order to determine whether the customer is likely to churn. Let's explore some of the decisions we can make to design and develop this solution with a particular focus on the questions of scaling up using the tools and techniques we have covered in this chapter.

First, we have the ingestion of the data. For this scenario, we will assume that the customer data, including interactions with different applications and systems, is processed at the end of the business day and numbers millions of records. This data contains numerical and categorical values and these need to be processed in order to feed into the downstream ML algorithm. If

the data is partitioned by date, and maybe some other feature of the data, then this plays really naturally into the use of Spark, as you can read this into a Spark DataFrame and use the partitions to parallelize the data processing steps.

Next, we have the feature engineering. If we are using a Spark DataFrame in the first step, then we can apply our transformation logic using the base PySpark syntax we have discussed earlier in this chapter. For example, if we want to apply some feature transformations available from Scikit-Learn or another ML library, we can wrap these in UDFs and apply at the scale we need to. The data can then be exported in our chosen data format using the PySpark API. For the customer churn model, this could mean a combination of encoding of categorical variables and scaling of numerical variables, in line with the techniques explored in *Chapter 3, From Model to Model Factory*.

Switching into the training of the model, now are moving from the data-intensive to the compute-intensive tasks. This means it is natural to start using Ray for model training, as you can easily set up parallel tasks to train models with different hyperparameter settings and distribute the training steps as well. There are particular benefits to using Ray for training deep learning or tree-based models as these are algorithms that are amenable to parallelization. So, if we are performing classification using one of the available models in Spark ML, then this can be done in a few lines, but if we are using something else, we will likely need to start wrapping in UDFs. Ray is far more library-agnostic but again the benefits really come if we are using a neural network in PyTorch or TensorFlow or using XGBoost or LightGBM, as these more naturally parallelize.

Finally, onto the model inference step. In a batch setting, it is less clear who the winner is in terms of suggested framework here. Using UDFs or the core PySpark APIs, you can easily create a quite scalable batch prediction stage using Apache Spark and your Spark cluster. This is essentially because prediction on a large batch is really just another large-scale data transformation, where Spark excels. If, however, you wish to serve your model as an endpoint that can scale across a cluster, this is where Ray has very easy-to-use capabilities as shown in the *Scaling your serving layer with Ray* section. Spark does not have a facility for creating endpoints in this way and the scheduling and task overheads required to get a Spark job up and running mean that it would not be worth running Spark on small packets of data coming in as requests like this.

For the customer churn example, this may mean that if we want to perform a churn classification on the whole customer base, Spark provides a nice way to process all of that data and leverage concepts like the underlying data partitions. You can still do this in Ray, but the lower-level API may mean it is slightly more work. Note that we can create this serving layer using many other mechanisms, as discussed in *Chapter 5, Deployment Patterns and Tools*, and the section on *Spinning up serverless infrastructure* in this chapter. *Chapter 8, Building an Example ML Microservice*, will also cover in detail how to use Kubernetes to scale out a deployment of an ML endpoint.

Finally, I have called the last stage the *application layer* to cover any “last mile” integrations between the output system and downstream systems in the solution. In this case, Spark does not really have a role to play since it can really be thought of as a large-scale data transformation engine. Ray, on the other hand, has more of a philosophy of general Python acceleration, so if there are tasks that would benefit from parallelization in the backend

of your applications, such as data retrieval, general calculations, simulation, or some other process, then the likelihood is you can still use Ray in some capacity, although there may be other tools available. So, in the customer churn example, Ray could be used for performing analysis at the level of individual customers and doing this in parallel before serving the results through a **Ray Serve** endpoint.

The point of going through this high-level example was to highlight the points along your ML engineering project where you can make choices about how to scale effectively. I like to say that there are often *no right answers, but very often wrong answers*. What I mean by this is that there are often several ways to build a good solution that are equally valid and may leverage different tools. The important thing is to avoid the biggest pitfalls and dead ends. Hopefully, the example gives some indication of how you can apply this thinking to scaling up your ML solutions.

IMPORTANT NOTE



Although I have presented a lot of questions here in terms of Spark vs. Ray, with a nod to Kubernetes as a more *base infrastructure* scaling option, there is now the ability to combine Spark and Ray through the use of **RayDP**. This toolkit now allows you to run Spark jobs on Ray clusters, so it nicely allows you to still use Ray as your base scaling layer but then leverage the Spark APIs and capabilities where it excels. RayDP was introduced in 2021 and is in active development, so this is definitely a capability to watch. For more information, see the project repository here: <https://github.com/oap-project/raydp>.

This concludes our look at how we can start to apply some of the scaling techniques we have discussed to our ML use cases.

We will now finish the chapter with a brief summary of what we have covered in the last few pages.

Summary

In this chapter, we looked at how to take the ML solutions we have built in the past few chapters and thought about how to scale them up to larger data volumes or higher numbers of requests for predictions. To do this, we mainly focused on **Apache Spark** as this is the most popular general-purpose engine for distributed computing. During our discussion of Apache Spark, we revisited some coding patterns and syntax we used previously in this book. By doing so, we developed a more thorough understanding of how and why to do certain things when developing in PySpark. We discussed the concept of **UDFs** in detail and how these can be used to create massively scalable ML workflows.

After this, we explored how to work with Spark on the cloud, specifically through the **EMR** service provided by AWS. Then, we looked at some of the other ways we can scale our solutions; that is, through serverless architectures and horizontal scaling with containers. In the former case, we walked through how to build a service for serving an ML model using **AWS Lambda**. This used standard templates provided by the AWS SAM framework. We provided a high-level view of how to use K8s and Kubeflow to scale out ML pipelines horizontally, as well as some of the other benefits of using these tools. A section covering the Ray parallel computing framework then followed, showing how you can use its relatively simple API to scale compute on heterogenous clusters to

supercharge your ML workflows. Ray is now one of the most important scalable computing toolkits for Python and has been used to train some of the largest models on the planet, including the GPT-4 model from OpenAI.

In the next chapter, we are going to build on the ideas of scale here by discussing the largest ML models you can build: deep learning models, including **large language models (LLMs)**. Everything we will discuss in this next chapter could only have been developed, and can often only be effectively utilized, by considering the techniques we have covered here. The question of scaling up your ML solutions will also be revisited in *Chapter 8, Building an Example ML Microservice*, where we will focus on the use of Kubernetes to horizontally scale an ML microservice. This complements nicely the work we have done here on scaling large batch workloads by showing you how to scale more real-time workloads. Also, in *Chapter 9, Building an Extract, Transform, Machine Learning Use Case*, many of the scaling discussions we have had here are prerequisites; so, everything we have covered here puts you in a good place to get the most from the rest of the book. So, armed with all this new knowledge, let's go and explore the world of the largest models known.

Join our community on Discord

Join our community's Discord space for discussion with the author and other readers:

<https://packt.link/mle>



7

Deep Learning, Generative AI, and LLM Ops

The world is changing. Fast. At the time of writing in mid-2023, **machine learning (ML)** and **artificial intelligence (AI)** have entered the public consciousness in a way that even a few months ago seemed impossible. With the rollout of ChatGPT in late 2022, as well as a wave of new tools from labs and organizations across the world, hundreds of millions of people are now using ML solutions every day to create, analyze, and develop. On top of this, innovation seems to only be speeding up, with what seems like a new announcement of a record-beating model or new tool every day. ChatGPT is only one example of a solution that uses what is now known as **generative artificial intelligence (generative AI or GenAI)**. While ChatGPT, Bing AI, and Google Bard are examples of text-based generative AI tools, there is also DALL-E and Midjourney in the image space and now a whole suite of multi-modal models combining these and other types of data. Given the complexity of the ecosystem that is evolving and the models that are being developed by leading AI labs around the world, it would be easy to feel overwhelmed. But fear not, as this chapter is all about answering the question, “What does this mean for me as a budding ML engineer?”

In this chapter, we will take the same strategy as in the other chapters of the book and focus on the core concepts and on building solid foundations that

you can use in your own projects for years to come. We will start with the fundamental algorithmic approach that has been at the heart of many cutting-edge developments in ML since the 2010s with a review of **deep learning**.

We will then discuss how you can build and host your own deep learning models before moving on to GenAI, where we will explore the general landscape before going into a deep dive into the approach behind ChatGPT and other powerful text models, **Large Language Models (LLMs)**.

This will then transition smoothly into an exploration of how ML engineering and MLOps can be applied to LLMs, including a discussion of the new challenges this brings. This is such a new area that much of what we will discuss in this chapter will reflect my views and understanding at the time of writing. As an ML community, we are only now starting to define what best practice means for these models, so we are going to be contributing to this brave new world together in the next few pages. I hope you enjoy the ride!

We will cover all of this in the following sections:

- Going deep with deep learning
- Going big with LLMs
- Building the future with LLMOps

Going deep with deep learning

In this book, we have worked with relatively “classical” ML models so far, which rely on a variety of different mathematical and statistical approaches to learn from data. These algorithms in general are not modeled on any biological theory of learning and are at their heart motivated by finding

procedures to explicitly optimize the loss function in different ways. A slightly different approach that the reader will likely be aware of, and that we met briefly in the section on *Learning about learning* in *Chapter 3, From Model to Model Factory*, is that taken by **Artificial Neural Networks (ANNs)**, which originated in the 1950s and were based on idealized models of neuronal activity in the brain. The core concept of an ANN is that through connecting relatively simple computational units called neurons or nodes (modeled on biological neurons), we can build systems that can effectively model any mathematical function (see the information box below for more details). The neuron in this case is a small component of the system that will return an output based on an input and the transformation of that input using some pre-determined mathematical formula. They are inherently non-linear and when acting in combination can very quickly begin to model quite complex data. Artificial neurons can be thought of as being arranged in layers, where neurons from one layer have connections to neurons in the next layer. At the level of small neural networks with not many neurons and not many layers, many of the techniques we have discussed in this book around retraining and drift detection still apply without modification. When we get to ANNs with many layers and neurons, so-called **Deep Neural Networks (DNNs)**, then we have to consider some additional concepts, which we will cover in this section.

The ability of neural networks to represent a huge variety of functions has a theoretical basis in what are known as **Universal Approximation Theorems**. These are rigorous mathematical results that prove that multilayer neural networks can approximate classes of mathematical functions to arbitrary levels of precision. These results don't say which



specific neural networks will do this, but they tell us that with enough hidden neurons or nodes, we can be sure that with enough data we should be able to represent our target function. Some of the most important results for these theorems were established in the late 1980s in papers like *Hornik, K., Stinchcombe, M. and White, H. (1989) “Multilayer feedforward networks are universal approximators”, Neural Networks, 2(5), pp. 359–366* and *Cybenko, G. (1989) “Approximation by superpositions of a sigmoidal function”, Mathematics of Control, Signals, and Systems, 2(4), pp. 303–314.*

DNNs have taken the world by storm in the last few years. From computer vision to natural language processing and from StableDiffusion to ChatGPT, there are now countless amazing examples of DNNs doing what was once considered the sole purview of humans. The in-depth mathematical details of deep learning models are covered in so much literature elsewhere, such as in the classic *Deep Learning* by Goodfellow, Bengio, Courville, MIT Press, 2016, that we would never be able to do them justice here. Although covering the detailed theory is far beyond the scope of this chapter, I will attempt to provide an overview of the main concepts and techniques you need in order to have a good working knowledge and to be able to start using these models in your ML engineering projects.

As mentioned, ANNs are based on ideas borrowed from biology, and just like in a biological brain, the ANN is built up of many individual *neurons*. Neurons can be thought of as providing the unit of computation in the ANN. The neuron works by taking multiple inputs and then combining them in a specified recipe to produce a single output, which can then act as one of the

inputs for another neuron or as part of the overall model’s output. The inputs to the neurons in a biological setting flow along *dendrites* and the outputs are channeled along *axons*.

But how do the inputs get transformed into the outputs? There are a few concepts we need to bring together to understand this process.

- **Weight:** Assigned to each connection between the neurons in the network is a numerical value that can be thought of as the “strength” of the connection. During the training of the neural network, the weights are one of the sets of values that are altered to minimize the loss. This is in line with the explanation of model training provided in *Chapter 3, From Model to Model Factory*.
- **Bias:** Every neuron in the network is given an another parameter that acts as an offset to the activation (defined below). This number is also updated during training and it gives the neural network more *degrees of freedom* to fit to the data. You can think of the bias as shifting the level at which the neuron will “fire” (or produce a certain output) and so having this as a variable value means there is more adaptability to the neuron.
- **Inputs:** These can be thought of as the raw data points that are fed to the neuron before we take into account the weights or the bias. If the neuron is being fed features based on the data, then the inputs are the feature values; if the neuron is being fed outputs from other neurons, then those are the values in that case.
- **Activation:** The neuron in an ANN receives multiple inputs; the activation is the linear combination of the inputs multiplied by the appropriate weights plus the bias term. This translates the multiple

pieces of incoming data into a single number that can then be used to determine what the neuron's output should be.

- **Activation function:** The activation is just a number, but the activation function is how we decide what that number means for the neuron. There is a variety of activation functions that are very popular in deep learning today but the important characteristic is that when this function acts on the activation value, it produces a number that is the output of the neuron or node.

These concepts are brought together diagrammatically in *Figure 7.1*. Deep learning models do not have a strict definition, but for our purposes, we can consider an ANN as deep as soon as it consists of three or more layers. This then means we must define some of the important characteristics of these layers, which we will do now:

- **Input layer:** This is the first layer of neurons that has as its input the raw data or prepared features created from the data.
- **Hidden layers:** These are the layers between the input and output layers, and can be thought of as where the main bulk of non-linear transformations of the data are performed. This is often simply because there are lots of hidden layers with lots of neurons! The way the neurons in the hidden layers are organized and connected are key parts of the *neural network architecture*.
- **Output layer:** The output layer is the one responsible for translating the outcome of the transformations that have been carried out in the neural network into a result that can be interpreted appropriately for the problem at hand. As an example, if we are using a neural network to classify an image, we need the final layer to output either a 1 or 0 for

the specified class or we could have it output a series of probabilities for different classes.

These concepts are useful background, but how do we start working with them in Python? The two most popular deep learning frameworks in the world are Tensorflow, released by Google Brain in 2015, and PyTorch, released by Meta AI in 2016. In this chapter, we will focus on examples using PyTorch, but many of the concepts apply equally well to TensorFlow with some modifications.

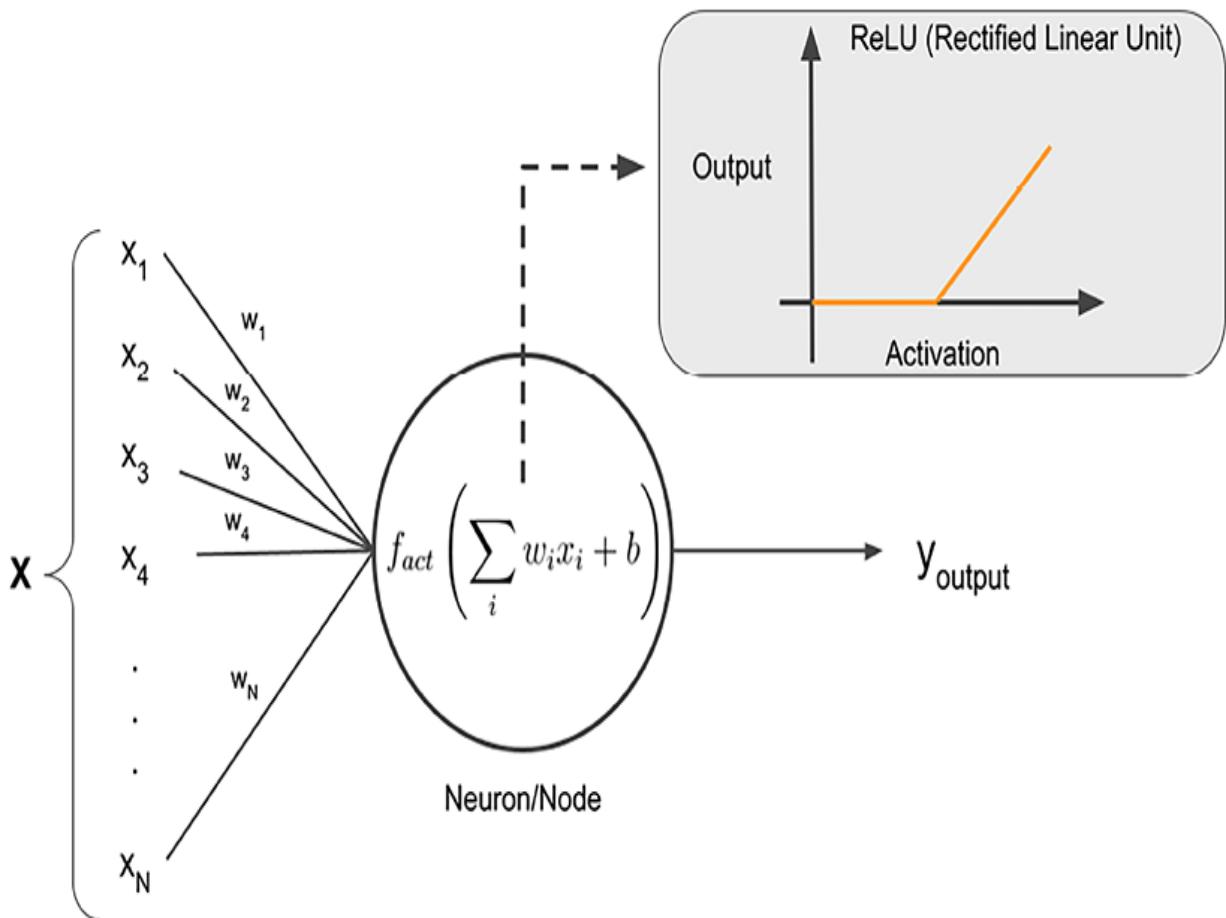


Figure 7.1: A schematic representation of a “neuron” in an ANN and how this takes input data, x , and transforms it into output, y .

Getting started with PyTorch

First, if you haven't already, install PyTorch. You can do this by following the PyTorch documentation at <https://pytorch.org/get-started/locally/>, for installing locally on Macbook, or use:

```
pip3 install torch
```

There are some important concepts and features of the PyTorch API that are useful to bear in mind when using PyTorch:

- `torch.Tensor`: Tensors are mathematical objects that can be represented by multi-dimensional arrays and are core components of any modern deep learning framework. The data we feed into the network should be cast as a tensor, for example:

```
inputs = torch.tensor(X_train, dtype=torch.float32)
labels = torch.tensor(y_train, dtype=torch.long)
```

- `torch.nn`: This is the main module used to define our neural network models. For example, we can use this to define a basic classification neural network containing three hidden layers, each with a **Rectified Linear Unit (ReLU)** activation function. When defining a model in PyTorch using this method, you should also write a method called `forward`, which defines how data is passed through the network during training. The following code shows how you can build a basic neural network inside a class that inherits from the

`torch.nn.Module` object. This network has four linear layers with ReLU activation functions and a simple forward-pass function:

```
import torch
import torch.nn as nn

class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.Sequential = nn.Sequential(
            nn.Linear(13, 64),
            nn.ReLU(),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 16),
            nn.ReLU(),
            nn.Linear(16, 3)
        )
    def forward(self, x):
        x = self.Sequential(x)
        return x
```

- **Loss functions:** In the `torch.nn` module, there is a series of loss functions that can be used for training the network. A popular choice is the cross-entropy loss, but there are many more to choose from in the documentation:

```
criterion = nn.CrossEntropyLoss()
```

- `torch.optim.Optimizer`: This is the base class for all optimizers in PyTorch. This allows for the implementation of most of the optimizers discussed in *Chapter 3, From Model to Model Factory*. When defining the optimizer in PyTorch, in most cases, you pass in the instantiated model's parameters and then the relevant parameters for the particular optimizer. For example, if we define an Adam optimizer with a learning rate of `0.001`, this is as simple as:

```
import torch.optim as optim
model = NeuralNetwork()
optimizer = torch.optim.Adam(
    model.parameters(),
    lr=0.001
)
```

- `torch.autograd`: Recall that training an ML model is really an optimization process that leverages a combination of linear algebra, calculus and some statistics. PyTorch performs model optimization using *automatic differentiation*, a method for converting the problem of finding partial derivatives of a function into the application of a series of primitives that is easy to compute but still results in calculating the differentiation to good precision. This is not to be confused with *finite differences or symbolic differentiation*. You call this implicitly by using a loss function and calling the `backward` method, which uses autograd to calculate the gradients for the weight updates in each epoch; this is then used in the optimizer by calling `optimizer.step()`. During a training run, it is important to reset any input tensors as tensors in PyTorch are mutable (operations change their data), and it is important to reset any gradients calculated in the

optimizer as well using `optimizer.zero_grad()`. Given this, an example training run with five hundred epochs would look like the following:

```
for epoch in range(500):
    running_loss = 0.0
    optimizer.zero_grad()

    inputs = torch.tensor(X_train, dtype=torch.float)
    labels = torch.tensor(y_train, dtype=torch.long)

    outputs = net(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()
```

- `torch.save` and `torch.load`: You can probably guess what these methods do from their names! But it is still important to show how to save and load your PyTorch models. When training deep learning models, it is also important to save the model periodically during the training process, as this often takes a long time. This is called “checkpointing” and means that you can pick up where you left off if anything goes wrong during the training runs. To save a PyTorch checkpoint, we can add syntax like the following to the training loop:

```
model_path = "path/to/model/my_model.pt"
torch.save({
            'epoch': epoch,
            'model_state_dict': model.state_()
```

```
'optimizer_state_dict': optimizer.state_dict(),
'loss': loss,
}, model_path)
```

- To then load in the model, you need to initialize another instance of your neural network class and of an optimizer object before reading in their states from the `checkpoint` object:

```
model = NeuralNetwork()
optimizer = torch.optim.Adam(model.parameters())
checkpoint = torch.load(model_path)
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
epoch = checkpoint['epoch']
loss = checkpoint['loss']
```

- `model.eval()` and `model.train()`: Once you have loaded in a PyTorch checkpoint, you need to set the model to the appropriate mode for the task you want to perform or there may be downstream issues. For example, if you want to perform testing and validation or you want to perform inference on new data with your model, then you need to call `model.eval()` before using it. This freezes any batch normalization or dropout layers you have included as they calculate statistics and perform updates during training that you do not want to be active during testing. Similarly, `model.train()` ensures these layers are ready to continue performing updates as expected during a training run.

It should be noted that there is a more extreme setting than `model.eval()` where you can entirely turn off any autograd functionality in your context by using the following syntax:

```
with torch.inference_mode():
```

This can give you added performance on inference but should only be used if you are certain that you do not need any gradient or tensor updates tracked or performed.

- **Evaluation:** If you wanted to test the model we have just trained in the example above you could calculate an accuracy using something like the syntax below, but any of the methods we have discussed in this book for model validation apply!

```
inputs = torch.tensor(X_test, dtype=torch.float)
labels = torch.tensor(y_test, dtype=torch.long)
outputs = net(inputs)
_, predicted = torch.max(outputs.data, 1)
correct = (predicted == labels).sum().item()
total = labels.size(0)
accuracy = correct / total
print('Accuracy on the test set: {:.2f}%')
```

And with that, you can now build, train, save, load and evaluate your first PyTorch model. We will now discuss how we take this further by considering some of the challenges of taking deep learning models into production.

Scaling and taking deep learning into production

Now we will move on to how to run deep learning models in a production system. To do this, we need to consider a few specific points that mark out DNNs from the other classical ML algorithms:

- **They are data-hungry:** DNNs often require relatively large amounts of data compared to other ML algorithms, due to the fact that they are performing an extremely complex multi-dimensional optimization, with the parameters of each neuron adding degrees of freedom. This means that for you to consider training a DNN from scratch, you have to do the leg work upfront to make sure you have enough data and that it is of a suitable variety to adequately train the model. The data requirements also typically mean that you need to be able to store a lot of data in memory as well, so this often has to be thought about ahead of time.
- **Training is more complex:** This point is related to the above but is subtly different. The very complex non-linear optimization problem we are solving means that during training, there are often many ways that the model can “get lost” and reach a sub-optimal local minimum. Techniques like the *checkpointing* example we described in the previous section are widespread in the deep learning community as you will often have to stop training at some step when the loss is not going in the right direction or stagnating, roll back, and try something different.
- **You have a new choice to make, the model architecture:** DNNs are also very different from classical ML algorithms because now you do not just have to worry about a few hyperparameters, but you also need to decide the architecture or shape of your neural network. This is often

a non-trivial exercise and can require detailed knowledge of neural networks. Even if you are working with a standard architecture like the Transformer architecture (see *Figure 7.2*), you should still have a solid grasp of what all the components are doing in order to effectively diagnose and resolve any issues. Techniques like automated architecture search as was discussed in *Chapter 3* in the section on *Learning about learning* can help speed up architecture design but sound foundational knowledge is still important.

- **Explainability is inherently harder:** A criticism that has been leveled at DNNs over the past few years is that their results can be very hard to explain. This is to be expected since the point is very much that DNN abstracts away a lot of the specifics of any problem into a more abstract approach. This can be fine in many scenarios but has now led to several high-profile cases of DNNs exhibiting undesired behavior like racial or gender bias, which can then be harder to explain and remediate. A challenge also arises in heavily regulated industries, like healthcare or finance, where your organization may have a legal duty to be able to evidence why a specific decision was made. If you used a DNN to help make this decision, this can often be quite challenging.

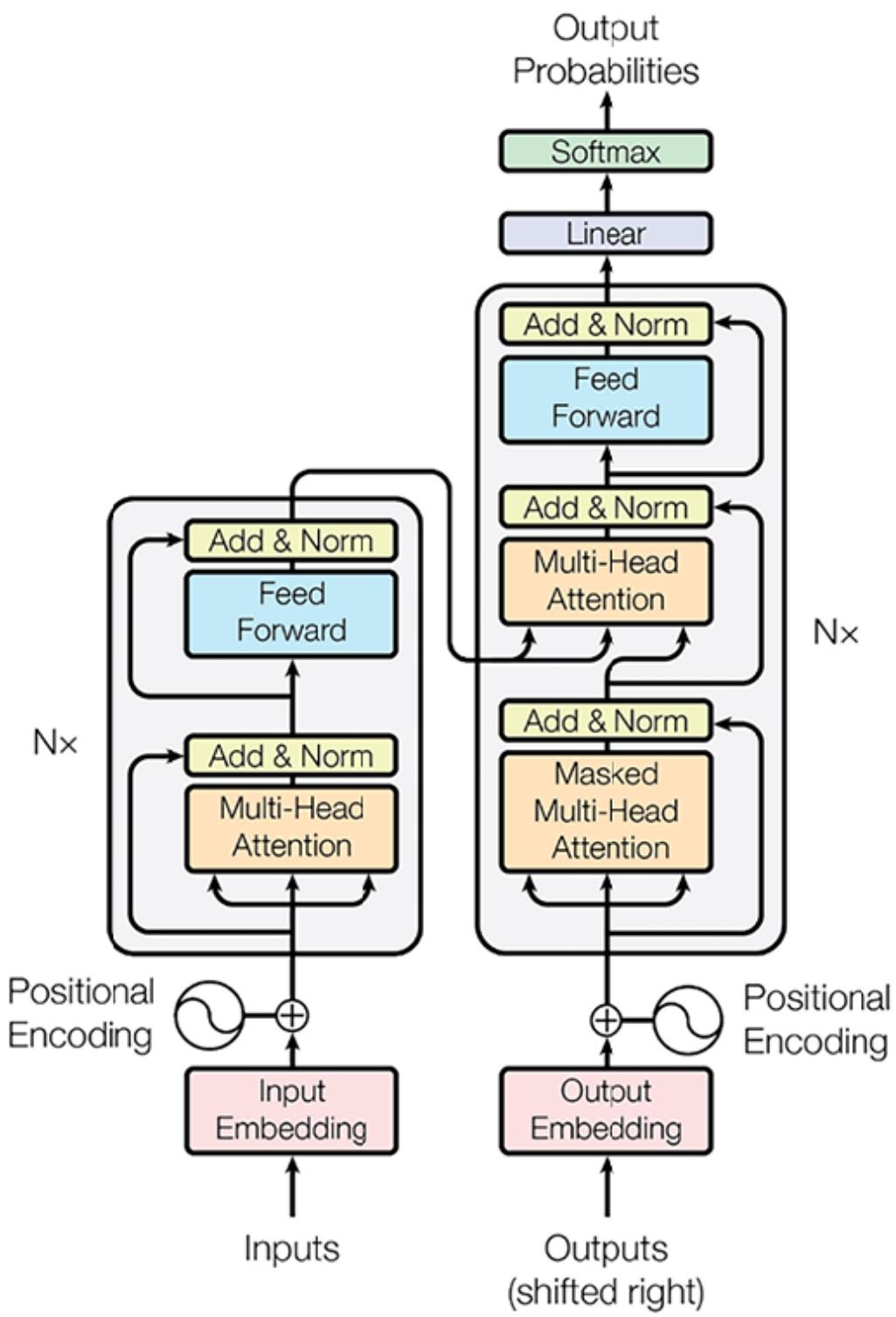


Figure 7.2: The Transformer architecture as originally published in the paper “Attention is all you need” by Google Brain, <https://arxiv.org/abs/1706.03762>.

Given all of this, what are some of the main things we should consider when using deep learning models for our ML-engineered systems? Well, one of the first things you can do is use existing pre-trained models, rather than train your own. This obviously comes with some risks around ensuring that the model and the data it was fed were of sufficient quality for your application, so always proceed with caution and do your due diligence.

In many cases, however, this approach is absolutely fine as we may be using a model that has been put through its paces in quite a public way and it may be known to be performant on the tasks we wish to use it for. Furthermore, we may have a use case where we are willing to accept the operational risk of importing and using this pre-existing model, contingent on our own testing. Let us assume we are in such an example now, and we want to build a basic pipeline to summarize some text conversations between clients and employees of a fictional organization. We can do this using an off-the-shelf transformer model, like that shown in *Figure 7.2*, from the Hugging Face **transformers** library.

All you need to get started is to know the name of the model you want to download from the Hugging Face model server; in this case, we will use the Pegasus text summarization model. Hugging Face provides a “**pipeline**” API to wrap around the model and make it easy to use:

```
from transformers import pipeline  
summarizer = pipeline("summarization", model= "c")
```

Performing our first deep learning model inference is then as easy as just passing in some inputs to this pipeline. So, for the fictional bot-human interaction described above, we can just pass in some example text and see

what this returns. Let's do this to summarize a fictional conversation between a customer and a chatbot, where the customer is trying to get more information on an order they have placed. The conversation is shown below:

```
text = "Customer: Hi, I am looking for some help
```

We will then feed this conversation into the summarizer `pipeline` object, and print the result:

```
summary = summarizer(text)
print(summary)
```

```
[{'summary_text': 'This is a live chat conversat']}
```

The result shows that the model has actually given a good summary of the nature of this interaction, highlighting how easy it is to start to do something that would probably have been very difficult or even impossible before the deep learning revolution.

We have just seen an example of using a pre-trained transformer model to perform some specific task, in this case text summarization, without any need for the model to be updated based on exposure to new data. In the next section, we will explore what to do when you want to update the model based on your own data.

Fine-tuning and transfer learning

In the previous section, we showed how easy it was to get started building solutions with existing deep learning models if ones could be found that were appropriate to your task. A good question to ask ourselves, however, is “What can I do if these models are not exactly right for my specific problem?” This is where the concepts of **fine-tuning** and **transfer learning** come in. Fine-tuning is when we take an existing deep learning model and then continue training the model on some new data. This means we are not starting from scratch and so can arrive at an optimized network far faster. Transfer learning is when we freeze most of the neural network’s state and retrain the last layer(s) with new data in order to perform some slightly different task or perform the same task in a way that is more appropriate to our problem. In both of these cases, this usually means that we can keep many of the powerful features of the original model, such as its feature representations, but start to adapt it for our specific use case.

To make this more concrete, we will now walk through an example of transfer learning in action. Fine-tuning can follow a similar process but just does not involve the adaptations to the neural network that we will implement. We will use the Hugging Face `datasets` and `evaluate` packages in this example, which will show how we can use a base **Bidirectional Encoder Representations from Transformers (BERT)** model and then use transfer learning to create a classifier that will estimate the star rating of reviews written in English on the Multilingual Amazon Reviews Corpus (<https://registry.opendata.aws/amazon-reviews-m1/>).

Figure 7.3 shows an example rating from this dataset:

```
1 dataset = fetch_dataset()  
2 import pprint  
3 pprint.pprint(dataset[100])
```

```
Found cached dataset amazon_reviews_multi (/Users/apmcm/.cache/huggingface/datasets/multilingual/amazon_reviews_multi/en/1.0.0/724e94f4b0c6c405ce7e476a6c5ef4f87db30799ad49f765094cf9770e0f)
```

```
{'language': 'en',  
 'product_category': 'sports',  
 'product_id': 'product_en_0610451',  
 'review_body': 'Two nights in the water tide to our dock in the lake..... I'd '  
 'say something liked this.',  
 'review_id': 'en_0143676',  
 'review_title': 'Let the picture tell you how go this is',  
 'reviewer_id': 'reviewer_en_0377453',  
 'stars': 1}
```

Figure 7.3: This shows an example review and star rating from the Multilingual Amazon Reviews Corpus.



Although we have used the BERT model in the following example, there are many variants that will work with the same example, such as DistilBERT or ALBERT, which are smaller models which aim to be quicker to train and to retain most of the performance of the original BERT model. You can play around with all of these, and may even find these models are faster to download due to their reduced size!

To start our transfer learning example:

1. First, we can use the `datasets` package to retrieve the dataset. We will use the concept of “configurations” and “splits” available for Hugging Face datasets, which specify specific subsets of the data and whether you want the train, test, or validate splits of the data. For this case, we want the English reviews and we will use the train split of the

data initially. *Figure 7.3* shows an example record from the dataset. The data is retrieved with the following syntax:

```
import datasets
from datasets import load_dataset

def fetch_dataset(dataset_name: str="amazon_reviews_multi",
                  configuration: str="en", size: int=1000000) -> datasets.arrow_dataset.Dataset:
    """
    Fetch dataset from HuggingFace datasets : https://huggingface.co/datasets/...
    """
    dataset = load_dataset(dataset_name, configuration, split="train")
    return dataset
```

2. The next step is to tokenize the dataset. To do this, we will use the **AutoTokenizer** that pairs with the BERT model we will use. Before we pull in that specific tokenizer, let's write a function that will use the selected tokenizer to transform the dataset. We will also define the logic to take the dataset and make it of the right form for use in later PyTorch processes. I have also added an option to downsample the data for testing:

```
import typing
from transformers import AutoTokenizer

def tokenize_dataset(tokenizer: AutoTokenizer,
                     dataset: datasets.arrow_dataset.Dataset,
                     sample=True) -> datasets.arrow_dataset.Dataset:
    """
    Tokenize dataset using AutoTokenizer
    """
    if sample:
        dataset = dataset.select(range(1000))
```

```
    Tokenize the HuggingFace dataset object &
    later Pytorch logic.
    ...
    tokenized_dataset = dataset.map(
        lambda x: tokenizer(x["review_body"],
                            truncation=True),
        batched=True
    )
    # Torch needs the target column to be named "label"
    tokenized_dataset = tokenized_dataset.rename_column("label", "target")

    # We can format the dataset for Torch using the PyTorch
    # Dataset API. This makes it very easy to create a DataLoader
    tokenized_dataset.set_format(
        type="torch", columns=["input_ids", "attention_mask"]
    )
    # Let's downsample to speed things up for this demo
    if sample==True:
        tokenized_dataset_small = tokenized_dataset.sample(1000,
                                                          shuffle=True)
        return tokenized_dataset_small
    else:
        return tokenized_dataset
```

3. Next, we need to create the PyTorch `dataloader` for feeding the data into the model:

```
from torch.utils.data import DataLoader

def create_dataloader(
```

```
        tokenized_dataset: datasets.arrow_dataset.Dataset,
        batch_size: int = 16,
        shuffle: bool = True
    ):
        dataloader = DataLoader(tokenized_dataset,
                               shuffle=shuffle,
                               batch_size=batch_size)
    return dataloader
```

4. Before we define the logic for training the model, it will be useful to write a helper function for defining the learning scheduler and the optimizer for the training run. This can then be called in our training function, which we will define in the next step. We will use the AdamW optimizer in this example:

```
from torch.optim import AdamW
from transformers import get_scheduler

def configure_scheduler_optimizer(
    model: typing.Any,
    dataloader: typing.Any,
    learning_rate: float,
    num_training_steps: int) -> tuple[typing.Any]:
    """
    Return a learning scheduler for use in the optimizer
    """
    optimizer = AdamW(model.parameters(), lr=learning_rate)
    lr_scheduler = get_scheduler(
        name="linear",
```

```
        optimizer=optimizer,
        num_warmup_steps=0,
        num_training_steps=num_training_steps
    )
    return lr_scheduler, optimizer
```

5. We can now define the model that we want to train using transfer learning. The `transformers` library from Hugging Face provides a very helpful wrapper to help you alter the classification head of a neural network based on a core BERT model. We instantiate this model and pass in the number of classes, which implicitly creates an update to the neural network architecture to give the logits for each of the classes upon running a prediction. When running inference, we will then take the class corresponding to the maximum of these logits as the inferred class. First, let's define the logic for training the model in a function:

```
import torch
from tqdm.auto import tqdm

def transfer_learn(
    model: typing.Any,
    dataloader: typing.Any,
    learning_rate: float = 5e-5,
    num_epochs: int = 3,
    progress_bar: bool = True )-> typing.Any
    device = torch.device("cuda") if torch.cuda.is_available()
                torch.device("cpu")
    model.to(device)

    num_training_steps = num_epochs * len(dataloader)
```

```
    lr_scheduler, optimizer = configure_sche
        model = model,
        dataloader = dataloader,
        learning_rate = learning_rate,
        num_training_steps = num_training_steps
    )

    if progress_bar:
        progress_bar = tqdm(range(num_training_steps))
    else:
        pass
    model.train()
    for epoch in range(num_epochs):
        for batch in dataloader:
            batch = {k: v.to(device) for k, v in batch.items()}
            outputs = model(**batch)
            loss = outputs.loss
            loss.backward()
            optimizer.step()
            lr_scheduler.step()
            optimizer.zero_grad()
            if progress_bar:
                progress_bar.update(1)
            else:
                pass
    return model
```

6. Finally, we can call all of these methods together to grab the tokenizer, pull in the dataset, transform it, define the model, configure the learning

scheduler and optimizer, and finally perform the transfer learning to create the final model:

```
tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
tokenized_dataset = tokenize_dataset(tokenizer,
                                      dataset)
dataloader = create_dataloader(tokenized_dataset)
model = AutoModelForSequenceClassification.from_pretrained(
    "bert-base-cased", num_labels=6)
transfer_learned_model = transfer_learn(
    model = model,
    dataloader=dataloader
)
```

7. We can then evaluate the performance of the model on the test split of the data using the Hugging Face `evaluate` package or any method we like. Note that in the example below, we call `model.eval()` so that the model is in evaluation mode as discussed previously:

```
import evaluate

device = torch.device("cuda") if torch.cuda.is_available() else
        torch.device("cpu")
metric = evaluate.load("accuracy")
model.eval()
eval_dataset = fetch_dataset(split="test")
tokenized_eval_dataset = tokenize_dataset(
    tokenizer=tokenizer, dataset=eval_dataset,
    eval_dataloader = create_dataloader(
        tokenized_dataset=tokenized_eval_dataset
```

```
for batch in eval_dataloader:
    batch = {k: v.to(device) for k, v in batch}
    with torch.no_grad():
        outputs = model(**batch)
    logits = outputs.logits
    predictions = torch.argmax(logits, dim=-1)
    metric.add_batch(predictions=predictions,
                      references=batch["labels"])
metric.compute()
```

This will return a dictionary with the value of the calculated metric like that shown below:

```
{'accuracy': 0.8}
```

And that is how you can use PyTorch and the Hugging Face `transformers` library to perform transfer learning.

The Hugging Face `transformers` library also now provides a very powerful Trainer API to help you perform fine-tuning in a more abstract way. If we take the same tokenizer and model from the previous examples, to use the Trainer API, we can just do the following:

1. When using the Trainer API, you need to define a `TrainingArguments` object, which can include hyperparameters and a few other flags. Let's just accept all of the default values but supply a path for checkpoints to be outputted to:

```
from transformers import TrainingArguments
training_args = TrainingArguments(output_dir=
```

2. We can then use the same `evaluate` package we used in the previous example to define a function for calculating any specified metrics, which we will pass into the main `trainer` object:

```
import numpy as np
import evaluate

metric = evaluate.load("accuracy")
def compute_metrics(eval_pred):
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)
    return metric.compute(predictions=predictions,
                          references=labels)
```

3. You then define the `trainer` object with all the relevant input objects:

```
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    compute_metrics=compute_metrics,
)
```

4. You train the model with these specified configurations and objects by calling

```
trainer.train().
```

And that is how you perform your own training on an existing model hosted on Hugging Face.

It is also useful to note that the Trainer API provides a really nice way to use a tool like **Optuna**, which we met in *Chapter 3, From Model to Model Factory*, in order to perform hyperparameter optimization. You can do this by specifying an Optuna trial search space:

```
def optuna_hp_space(trial):
    return {
        "learning_rate": trial.suggest_float("le
        lc
    }
```

And then defining a function for initializing the neural network during every state of the hyperparameter search:

```
def model_init():
    model = AutoModelForSequenceClassification.f
                    "bert-base-cased", num_labels=6)
    return model
```

You then just need to pass this into the `Trainer` object:

```
trainer = Trainer(
    model=None,
```

```
    args=training_args,  
    train_dataset=train_dataset,  
    eval_dataset=eval_dataset,  
    compute_metrics=compute_metrics,  
    tokenizer=tokenizer,  
    model_init=model_init,  
)
```

Finally, you can then run the hyperparameter search and retrieve the best run:

```
best_run = trainer.hyperparameter_search(  
    n_trials=20,  
    direction="maximize",  
    hp_space=optuna_hp_space  
)
```

And that concludes our example of transfer learning and fine-tuning of PyTorch deep learning models using the tools from Hugging Face. An important point to note is that both fine-tuning and transfer learning are still training processes and so can still be applied to the model factory methodology that was laid out in *Chapter 3, From Model to Model Factory*. For example, when we say “train” in the “train-run” process outlined in *Chapter 3*, this may now refer to the fine-tuning or transfer learning of a pre-trained deep learning model.

As we have covered extensively already, deep learning models can be very powerful tools for solving a variety of problems. One of the trends that has been explored aggressively in recent years by many groups and organizations is the question of what is possible as these models get larger

and larger. In the next section, we are going to start answering that question by exploring what happens when deep learning models get extremely large. It is time to enter the world of LLMs.

Living it large with LLMs

At the time of writing, GPT-4 has been released only a few months previously, in March 2023, by OpenAI. This model is potentially the largest ML model ever developed, with a reported one trillion parameters, although OpenAI has not confirmed the exact number. Since then, Microsoft and Google have announced advanced chat capabilities using similarly large models in their product suites and a raft of open-source packages and toolkits have been released. All of these solutions leverage some of the largest neural network models ever developed, LLMs. LLMs are part of an even wider class of models known as **foundation models**, which span not just text applications but video and audio as well. These models are roughly classified by the author as being too large for most organizations to consider training from scratch. This will mean organizations will either consume these models as third-party services or host and then fine-tune existing models. Solving this integration challenge in a safe and reliable way represents one of the main challenges in modern ML engineering. There is no time to lose, as new models and capabilities seem to be released every day; so let's go!

Understanding LLMs

The main focus of LLM-based systems is to create human-like responses to a wide range of text-based inputs. LLMs are based on transformer architectures, which we have already met. This enables these models to

process input in parallel, significantly reducing the amount of time for training on the same volume of data compared to other deep learning models.

The architecture of LLMs, as for any transformer, consists of a series of encoders and decoders that leverages self-attention and feed-forward neural networks.

At a high level, you can think of the encoders as being responsible for processing the input, transforming it into an appropriate numerical representation, and then feeding this into the decoders, from which the output can be generated. The magic of transformers comes from the use of **self-attention**, which is a mechanism for capturing the contextual relationships between words in a sentence. This results in **attention vectors** that represent this numerically, and when multiples of these are being calculated, it is called **multi-headed attention**. Both the encoder and decoder use self-attention mechanisms to capture the contextual dependencies of the input and output sequences.

One of the most popular transformer-based models used in LLMs is the BERT model. BERT was developed by Google and is a pre-trained model that can be fine-tuned for various natural language tasks.

Another popular architecture is the **Generative Pre-trained Transformer (GPT)**, created by OpenAI. The ChatGPT system, released by OpenAI in November 2022, apparently utilized a third-generation GPT model when it took the world by storm. At the time of writing in March 2023, these models are up to their fourth generation and are incredibly powerful. Although GPT-4 is still relatively new, it is already sparking a heated debate about the future of AI and whether or not we have reached **artificial general**

intelligence (AGI). The author does not believe we have, but what an exciting time to be in this space anyway!

The thing that makes LLMs infeasible to train anew in every new business context or organization is that they are trained on colossal datasets. GPT-3, which was released in 2020, was trained on almost 500 billion tokens of text. A token in this instance is a small fragment of a word used for the training and inferences process in LLMs, roughly around 4 characters in English. That is a lot of text! The costs for training these models are therefore concomitantly large and even inference can be hugely costly. This means that organizations whose sole focus is not producing these models will likely fail to see the economies of scale and the returns required to justify investing in them at this scale. This is before you even consider the need for specialized skill sets, optimized infrastructure, and the ability to grab all of that data. There are a lot of parallels with the advent of the public cloud several years ago, where organizations no longer had to invest in as much on-premises infrastructure or expertise and instead started to pay on a “what you use” basis. The same thing is now happening with the most sophisticated ML models. This is not to say that smaller, more domain-specific models have been ruled out. In fact, I think that this will remain one of the ways that organizations can leverage their own unique datasets to drive advantage over competitors and build out better products. The most successful teams will be those that combine this approach with the approach from the largest models in a robust way.

Scale is not the only important component though. ChatGPT and GPT-4 were not only trained on huge amounts of data but they were also then fine-tuned using a technique called **Reinforcement Learning with Human Feedback (RLHF)**. During this process, the model is presented with a prompt, such as a conversational question, and generates a series of potential

responses. The responses are then presented to a human evaluator who provides feedback on the quality of the response, usually by ranking them, which is then used to train a **reward model**. This model is then used to fine-tune the underlying language model through techniques like **Proximal Policy Optimization (PPO)**. The details of all of this are well beyond the scope of this book but hopefully, you are gaining an intuition for how this is not run-of-the-mill data science that any team can quickly scale up. And since this is the case, we have to learn how to work with these tools as more of a “black box” and consume them as third-party solutions. We will cover this in the next section.

Consuming LLMs via API

As discussed in the previous sections, the main change in our way of thinking as ML engineers who want to interact with LLMs, and foundation models in general, is that we can no longer assume we have access to the model artifact, the training data, or testing data. We have to instead treat the model as a third-party service that we should call out to for consumption. Luckily, there are many tools and techniques for implementing this.

The next example will show you how to build a pipeline that leverages LLMs by using the popular **LangChain** package. The name comes from the fact that to leverage the power of LLMs, we often have to chain together many interactions with them with calls to other systems and sources of information. LangChain also provides a wide variety of functionality that is useful when dealing with NLP and text-based applications more generally. For example, there are utilities for text splitting, working with vector databases, document loading and retrieval, and conversational state persistence, among others. This makes it a worthwhile package to check out

even in projects where you are not necessarily working with LLMs specifically.

First, we walk through a basic example of calling the OpenAI API:

1. Install the `langchain` and `openai` Python bindings:

```
pip install langchain  
pip install openai
```

2. We assume the user has set up an OpenAI account and has access to an API key. You can set this as an environment variable or use a secrets manager for storage, like the one that GitHub provides. We will assume the key is accessible as an environment variable:

```
import os  
openai_key = os.getenv('OPENAI_API_KEY')
```

3. Now, in our Python script or module, we can define the model we will call using the OpenAI API as accessed via the `langchain` wrapper. Here we will work with the `gpt-3.5-turbo` model, which is the most advanced of the GPT-3.5 chat models:

```
from langchain.chat_models import ChatOpenAI  
gpt = ChatOpenAI(model_name='''gpt-3.5-turbo
```

4. LangChain then facilitates the building up of pipelines using LLMs via prompt templates, which allow you to standardize how we will prompt and parse the response of the models:

```
template = '''Question: {question}
Answer: '''
prompt = PromptTemplate(
    template=template,
    input_variables=['question']
)
```

5. We can then create our first “chain,” which is the mechanism for pulling together related steps in `langchain`. This first chain is a simple one that takes a prompt template and the input to create an appropriate prompt to the LLM API, before returning an appropriately formatted response:

```
# user question
question = "Where does Andrew McMahon, author of 'Python for Data
Engineering with Python', work?"
# create prompt template > LLM chain
llm_chain = LLMChain(
    prompt=prompt,
    llm=gpt
)
```

6. You can then run this question and print the result to the terminal as a test:

```
print(llm_chain.run(question))
```

This returns:

```
As an AI language model, I do not have access
```

Given that I am an ML engineer employed by a large bank and am based in Glasgow, United Kingdom, you can see that even the most sophisticated LLMs will get things wrong. This is an example of what we term a *hallucination*, where an LLM gives an incorrect but plausible answer. We will return to the topic of when LLMs get things wrong in the section on Building the future with *LLMops*. This is still a good example of building a basic mechanism through which we can programmatically interact with LLMs in a standardized way.

LangChain also provides the ability to pull multiple prompts together using a method in the chain called `generate`:

```
questions = [  
    {'question': "Where does Andrew McMahon, aut",  
     'question': 'What is MLOps?'},  
    {'question': 'What is ML engineering?'},  
    {'question': 'What's your favorite flavor of j'}  
]  
print(llm_chain.generate(questions))
```

The response from this series of questions is rather verbose, but here is the first element of the returned object:

```
generations=[ [ChatGeneration(text='As an AI mode
```

Again, not *quite* right. You get the idea though! With some prompt engineering and better conversation design, this could quite easily be a lot better. I'll leave you to play around and have some fun with it.

This quick introduction to LangChain and LLMs only scratches the surface, but hopefully gives you enough to fold in calls to these models into your ML workflows.

Let's move on to discuss another important way that LLMs are becoming part of the ML engineering toolkit, as we explore software development using AI assistants.

Coding with LLMs

LLMs are not only useful for creating and analyzing natural language; they can also be applied to programming languages. This is the purpose of the OpenAI Codex family of models, which has been trained on millions of code repositories with the aim of being able to produce reasonable-looking and performing code when prompted. Since GitHub Copilot, an AI assistant for coding, was launched, the concept of an AI assistant helping you code has entered the mainstream. Many people have argued that these solutions provide massive productivity boosts and improved enjoyment when executing their own work. GitHub has published some of its own research suggesting that 60-75% of 2,000 developers asked reported less frustration and improved satisfaction when developing software. On a far smaller cohort of 95 developers, with 50 being in the control group, they also showed a speedup when developing an HTTP server in JavaScript with a given specification. I believe there should be much more work done on this topic before we declare that AI coding assistants are obviously making us all happier and faster, but the GitHub survey and test results definitely suggest

they are a useful tool to try out. These results are published at <https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/>. To this previous point, an interesting pre-print paper on the arXiv by researchers from Stanford University, [arXiv:2211.03622 \[cs.CR\]](#), seems to show that developers using an AI coding assistant based on the OpenAI [codex-davinci-002](#) model were more likely to introduce security vulnerabilities into their code and that users of the model would feel more confident in their work even though it contained these issues! It should be noted that the model they used is relatively old in terms of the family of LLMs that OpenAI offers now, so again more research is needed. This does raise the interesting possibility that AI coding assistants may provide a speed boost but also introduce more bugs. Time will tell. This area has also started to heat up with the introduction of powerful open-source contenders. A particular one to call out is StarCoder, which was developed through a collaboration between Hugging Face and ServiceNow

<https://huggingface.co/blog/starcoder>. The one thing that is certain is that these assistants are not going anywhere and they are only going to improve with time. In this section, we will start to explore the possibilities of working with these AI assistants in a variety of guises. Learning to work with AI is likely going to be a critical part of the ML engineering workflow in the near future, so let's get learning!

First, when would I want to use an AI coding assistant as an ML engineer? The consensus in the community, and in the GitHub research, looks to be that these assistants help with the development of boilerplate code on established languages, Python among them. They do not seem to be ideal for

when you want to do something particularly innovative or different; however, we will explore this also.

So, how do you actually work with an AI to help you code? At the time of writing, there seem to be two main methods (but given the pace of innovation, it could be that you are working with an AI through a brain-computer interface in no time; who knows?), each of which has its own advantages and disadvantages:

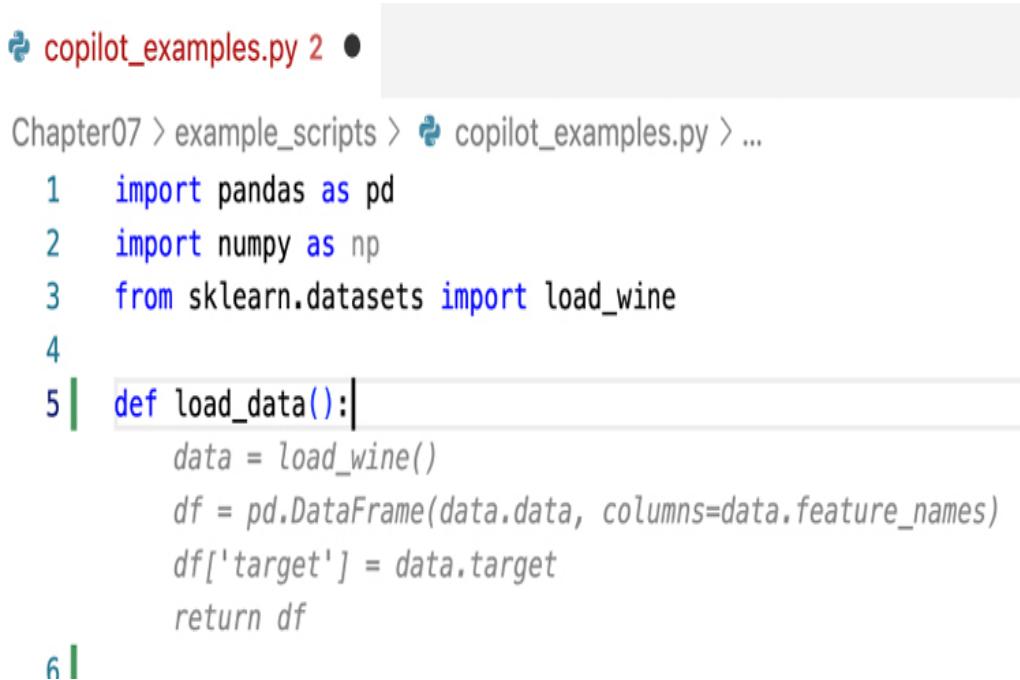
- **Direct editor or IDE integration:** In Copilot-supported code editors and IDEs, which at the time of writing include the PyCharm and VS Code environments we have used in this book, then you can enable Copilot to offer autocomplete suggestions for your code as you type it. You can also provide prompt information for the LLM model in your comments in the code. This mode of integration will likely always be there as long as developers are using these environments, but I can foresee a large number of AI assistant offerings in the future.
- **Chat interface:** If you are not using Copilot but instead another LLM, for example, OpenAI's GPT-4, then you will likely need to work in some sort of chat interface and copy and paste relevant information between your coding environment and the chat. This may seem a bit more clunky but is definitely far more versatile as it means you can easily switch between the models of your choice, or even combine multiples. You could actually build your own code to feed in your code with these models if you have the relevant access permissions and APIs to hit, but at that point, you are just redeveloping a tool like Copilot!

We will walk through an example showing both and highlighting how this may potentially help you in your future ML engineering projects.

If you navigate to the GitHub Copilot web page, you can sign up for an individual subscription for a monthly fee and they offer a free trial. Once you have done that, you can follow the setup instructions for your chosen code editor here:

<https://docs.github.com/en/copilot/getting-started-with-github-copilot>.

Once you have this setup, as I have done for VS Code, then you can start to use Copilot straight away. For example, I opened a new Python file and started typing some typical imports. When I started to write my first function, Copilot kicked in with a suggestion to complete the entire function, as shown in *Figure 7.4*.



The screenshot shows a Python script named `copilot_examples.py` in a VS Code editor. The file path is `Chapter07 > example_scripts > copilot_examples.py`. The code has been partially typed by the user:

```
1 import pandas as pd
2 import numpy as np
3 from sklearn.datasets import load_wine
4
5 def load_data():
6     data = load_wine()
7     df = pd.DataFrame(data.data, columns=data.feature_names)
8     df['target'] = data.target
9     return df
```

The line `def load_data():` is highlighted with a green vertical bar, indicating it is the current line being edited or completed. The code block is enclosed in a light gray box.

Figure 7.4: A suggested autocompletion from GitHub Copilot in VS Code.

As mentioned above, this is not the only way you can provide input to Copilot; you can also use comments to provide more information to the model. In *Figure 7.5*, we can see that providing some commentary in a

leading comment line helps define the logic we want to be contained in the function.

```
# Function to standardize numerical feature and one hot encode categorical features
def standardize(df):
    df = df.copy()
    for col in df.columns:
        if df[col].dtype == 'object':
            df[col] = df[col].astype('category')
            df[col] = df[col].cat.codes
        else:
            df[col] = (df[col] - df[col].mean()) / df[col].std()
    return df
```

Figure 7.5: By providing a leading comment, you can help Copilot suggest the logic you want for your code.

There are a few things that help get the best out of Copilot that are useful to bear in mind as you use it:

- **Be very modular:** The more modular you can make your code, the better. We've already discussed why this benefits maintenance and quicker development, but here it also helps the Codex model create more appropriate auto-completion suggestions. If your functions are going to be long, complex objects, then the likelihood is that the Copilot suggestion wouldn't be great.
- **Write clear comments:** This is always a good practice of course but it really helps Copilot understand what code you need. It can help to write longer comments at the top of the file describing what you want the solution to do and then write shorter but very precise comments before your functions. The example in *Figure 7.5* shows a comment that specifies the way in which I wanted the function to perform the feature

preparation, but if the comment simply said “*standardize features*,” the suggestion would likely not be as complete.

- **Write the interfaces and function signatures:** As in *Figure 7.5*, it helps if you start the piece of code off by providing the function signature and the types or the first line of the class definition if this was a class. This acts to prime the model to complete the rest of the code block.

Hopefully, this is enough to get you started on your journey working with AI to build your solutions. I think that as these tools become more ubiquitous, there are going to be many opportunities to use them to supercharge your development workflows.

Now that we know how to build some pipelines using LLMs and we know how to start leveraging them to aid our own development, we can turn to what I think is one of the most important topics in this field. I also think it is the one with the most unanswered questions and so it is a very exciting one to explore. This all relates to the question of the operational implications of leveraging LLMs, now being termed **LLMOps**.

Building the future with LLMOps

Given the rise in interest in LLMs recently, there has been no shortage of people expressing the desire to integrate these models into all sorts of software systems. For us as ML engineers, this should immediately trigger us to ask the question, “What will that mean operationally?” As discussed throughout this book, the marrying together of operations and development of ML systems is termed MLOps. Working with LLMs is likely to lead to its

own interesting challenges, however, and so a new term, LLMOps, has arisen to give this sub-field of MLOps some good marketing.

Is this really any different? I don't think it is *that* different, but should be viewed as a sub-field of MLOps with its own additional challenges. Some of the main challenges that I see in this area are:

- **Larger infrastructure, even for fine-tuning:** As discussed previously, these models are far too large for typical organizations or teams to consider training their own, so instead teams will have to leverage third-party models, be they open-source or proprietary, and fine-tune them. Fine-tuning models of this scale will still be very expensive and so there will be a higher premium on building very efficient data ingestion, preparation, and training pipelines.
- **Model management is different:** When you train your own models, as we showed several times in *Chapter 3, From Model to Model Factory*, effective ML engineering requires us to define good practices for versioning our models and storing metadata that provide lineage of the experiments and training runs we have gone through to produce these models. In a world where models are more often hosted externally, this is slightly harder to do, as we do not have access to the training data, to the core model artifacts, and probably not even to the detailed model architecture. Versioning metadata will then likely default to the publicly available metadata for the model, think along the lines of `gpt-4-v1.3` and similar-sounding names. That is not a lot of information to go on, and so you will likely have to think of ways to enrich this metadata, perhaps with your own example runs and test results in order to understand how that model behaved in certain scenarios. This then also links to the next point.

- **Rollbacks become more challenging:** If your model is hosted externally by a third party, you do not control the roadmap of that service. This means that if there is an issue with version 5 of a model and you want to roll back to version 4, that option might not be available to you. This is a different kind of “drift” from the model performance drift we’ve discussed at length in this book but it is going to become increasingly important. This will mean that you should have your own model, perhaps with nowhere near the same level of functionality or scale, ready as a last resort default to switch to in case of issues.
- **Model performance is more of a challenge:** As mentioned in the previous point, with foundation models being served as externally hosted services, you are no longer in as much control as you were. This then means that if you do detect any issues with the model you are consuming, be they drift or some other bugs, you are very limited in what you can do and you will need to consider that default rollback we just discussed.
- **Applying your own guardrails will be key:** LLMs hallucinate, they get things wrong, they can regurgitate training data, and they might even inadvertently offend the person interacting with them. All of this means that as these models are adopted by more organizations, there will be a growing need to develop methods for applying bespoke guardrails to systems utilizing them. As an example, if an LLM was being used to power a next-generation chatbot, you could envisage that between the LLM service and the chat interface, you could have a system layer that checked for abrupt sentiment changes and important keywords or data that should be obfuscated. This layer could utilize simpler ML models and a variety of other techniques. At its most

sophisticated, it could try and ensure that the chatbot did not lead to a violation of ethical or other norms established by the organization. If your organization has made the climate crisis an area of focus, you may want to screen the conversation in real time for information that goes against critical scientific findings in this area as an example.

Since the era of foundation models has only just begun, it is likely that more and more complex challenges will arise to keep us busy as ML engineers for a long time to come. To me, this is one of the most exciting challenges we face as a community, how we harness one of the most sophisticated and cutting-edge capabilities ever developed by the ML community in a way that still allows the software to run safely, efficiently, and robustly for users day in and day out. Are you ready to take on that challenge?

Let's dive into some of these topics in a bit more detail, first with a discussion of LLM validation.

Validating LLMs

The validation of generative AI models is inherently different from and seemingly more complex than the same for other ML models. The main reasons for this are that when you are *generating* content, you are often creating very complex data in your results that has never existed! If an LLM returns a paragraph of generated text when asked to help summarize and analyze some document, how do you determine if the answer is “good”? If you ask an LLM to reformat some data into a table, how can you build a suitable metric that captures if it has done this correctly? In a generative context, what do “model performance” and “drift” really mean and how do I calculate them? Other questions may be more use case dependent, for example, if you are building an information retrieval or Retrieval-

Augmented Generation (see *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*, <https://arxiv.org/pdf/2005.11401.pdf>) solution, how do you evaluate the truthfulness of the text generated by the LLM?

There are also important considerations around how we screen the LLM-generated outputs for any potential biased or toxic outputs that may cause harm or reputational damage to the organization running the model. The world of LLM validation is complex!

What can we do? Thankfully, this has not all happened in a vacuum and there have been several benchmarking tools and datasets released that can help us on our journey. Things are so young that there are not many worked examples of these tools yet, but we will discuss the key points so that you are aware of the landscape and can keep on top of how things are evolving. Let's list some of the higher-profile evaluation frameworks and datasets for LLMs:

- **OpenAI Evals:** This is a framework whereby OpenAI allows for the crowdsourced development of tests against proposed text completions generated by LLMs. The core concept at the heart of evals is the “Completion Function Protocol,” which is a mechanism for standardizing the testing of the strings returned when interacting with an LLM. The framework is available on GitHub at <https://github.com/openai/evals>.
- **Holistic Evaluation of Language Models (HELM):** This project, from Stanford University, styles itself as a “living benchmark” for LLM performance. It gives you a wide variety of datasets, models, and metrics and shows the performance across these different combinations. It is a very powerful resource that you can use to base your own test

scenarios on, or indeed just to use the information directly to understand the risks and potential benefits of using any specific LLM for your use case. The HELM benchmarks are available at <https://crfm.stanford.edu/helm/latest/>.

- **Guardrails AI:** This is a Python package that allows you to do validation on LLM outputs in the same style as `pydantic`, which is a very powerful idea! You can also use it to build control flows with the LLM for when issues arise like a response to a prompt not meeting your set criteria; in this case, you can use Guardrails AI to re-prompt the LLM in the hope of getting a different response. To use Guardrails AI, you specify a **Reliable AI Markup Language (RAIL)** file that defines the prompt format and expected behavior in an XML-like file. Guardrails AI is available on GitHub at <https://shreyar.github.io/guardrails/>.

There are several more of these frameworks being created all the time, but getting familiar with the core concepts and datasets out there will become increasingly important as more organizations want to take LLM-based systems from fun proofs-of-concept to production solutions. In the penultimate section of this chapter, we will briefly discuss some specific challenges I see around the management of “prompts” when building LLM applications.

PromptOps

When working with generative AI that takes text inputs, the data we input is often referred to as “prompts” to capture the conversational origin of working with these models and the concept that an input demands a response, the same way a prompt from a person would. For simplicity, we

will call any input data that we feed to an LLM a prompt, whether this is in a user interface or via an API call and irrespective of the nature of the content we provide to the LLM.

Prompts are often quite different beasts from the data we typically feed into an ML model. They can be effectively freeform, have a variety of lengths, and, in most cases, express the intent for how we want the model to act. In other ML modeling problems, we can certainly feed in unstructured textual data, but this intent piece is missing. This all leads to some important considerations for us as ML engineers working with these models.

First, the shaping of prompts is important. The term **prompt engineering** has become popular in the data community recently and refers to the fact that there is often a lot of thought that goes into designing the content and format of these prompts. This is something we need to bear in mind when designing our ML systems with these models. We should be asking questions like “Can I standardize the prompt formats for my application or use case?”, “Can I provide appropriate additional formatting or content on top of what a user or input system provides to get a better outcome?”, and similar questions. I will stick with calling this prompt engineering.

Secondly, prompts are not your typical ML input, and tracking and managing them is a new, interesting challenge. This challenge is compounded by the fact that the same prompt may give very different outputs for different models, or even with different versions of the same model. We should think carefully about tracking the lineage of our prompts and the outputs they generate. I term this challenge **prompt management**.

Finally, we have a challenge that is not necessarily unique to prompts but definitely becomes a more pertinent one if we allow users of a system to feed in their own prompts, for example in chat interfaces. In this case, we

need to apply some sort of screening and obfuscation rules to data coming in and coming out of the model to ensure that the model is not “jailbroken” in some way to evade any guardrails. We would also want to guard against adversarial attacks that may be designed to extract training data from these systems, thereby gaining personally identifiable or other critical information that we do not wish to be shared.

As you begin to explore this brave new world of LLMOps with the rest of the world, it will be important to keep these prompt-related challenges in mind. We will now conclude the chapter with a brief summary of what we have covered.

Summary

In this chapter, we focused on deep learning. In particular, we covered the key theoretical concepts behind deep learning, before moving on to discuss how to build and train your own neural networks. We walked through examples of using off-the-shelf models for inference and then adapting them to your specific use cases through fine-tuning and transfer learning. All of the examples shown were based on heavy use of the PyTorch deep learning framework and the Hugging Face APIs.

We then moved on to the topical question of the largest models ever built, LLMs, and what they mean for ML engineering. We explored a little of their important design principles and behaviors before showing how to interact with them in pipelines using the popular LangChain package and OpenAI APIs. We also explored the potential for using LLMs to help with improving software development productivity, and what this will mean for you as an ML engineer.

We finished the chapter with an exploration of the new topic of LLMOps, which is all about applying the principles of ML engineering and MLOps that we have been discussing throughout this book to LLMs. This covered the core components of LLMOps and also some new capabilities, frameworks, and datasets that can be used to validate your LLMs. We concluded with some pointers on managing your LLM prompts and how the concepts around experiment tracking we covered in *Chapter 3, From Model to Model Factory*, should be translated to apply in this case.

The next chapter will begin the final section of the book and will cover a detailed end-to-end example where we will build an ML microservice using Kubernetes. This will allow us to apply many of the skills we have learned through the book.

Join our community on Discord

Join our community's Discord space for discussion with the author and other readers:

<https://packt.link/mle>



8

Building an Example ML Microservice

This chapter will be all about bringing together some of what we have learned in the book so far with a realistic example. This will be based on one of the scenarios introduced in *Chapter 1, Introduction to ML Engineering*, where we were required to build a forecasting service for store item sales. We will discuss the scenario in a bit of detail and outline the key decisions that have to be made to make a solution a reality, before showing how we can employ the processes, tools, and techniques we have learned through out this book to solve key parts of the problem from an ML engineering perspective. By the end of this chapter, you should come away with a clear view of how to build your own ML microservices for solving a variety of business problems.

In this chapter, we will cover the following topics:

- Understanding the forecasting problem
- Designing our forecasting service
- Selecting the tools
- Training at scale
- Serving the models with FastAPI
- Containerizing and deploying to Kubernetes

Each topic will provide an opportunity for us to walk through the different decisions we have to make as engineers working on a complex ML delivery. This will provide us with a handy reference when we go out and do this in the real world!

With that, let's get started and build a forecasting microservice!

Technical requirements

The code examples in this chapter will be simpler to follow if you have the following installed and running on your machine:

- Postman or another API development tool
- A local Kubernetes cluster manager like minikube or kind
- The Kubernetes CLI tool, `kubectl`

There are several different `conda` environment `.yml` files contained in the `Chapter08` folder in the book's GitHub repo for the technical examples, as there are a few different sub-components. These are:

- `mlewp-chapter08-train`: This specifies the environment for running the training scripts.
- `mlewp-chapter08-serve`: This specifies the environment for the local FastAPI web service build.
- `mlewp-chapter08-register`: This gives the environment specification for running the MLflow tracking server.

In each case, create the Conda environment, as usual, with:

```
conda env create -f <ENVIRONMENT_NAME>.yml
```

The Kubernetes examples in this chapter also require some configuration of the cluster and the services we will deploy; these are given in the `Chapter08/forecast` folder under different `.yaml` files. If you are using kind, you can create a cluster with a simple configuration by running:

```
kind create cluster
```

Or you can use one of the configuration `.yaml` files provided in the repository:

```
kind create cluster --config cluster-config-ch0
```

Minikube does not provide an option to read in a cluster configuration `.yaml` like kind, so instead, you should simply run:

```
minikube start
```

to deploy your local cluster.

Understanding the forecasting problem

In *Chapter 1, Introduction to ML Engineering*, we considered the example of an ML team that has been tasked with providing forecasts of items at the level of individual stores in a retail business. The fictional business users had the following requirements:

- The forecasts should be rendered and accessible via a web-based dashboard.
- The user should be able to request updated forecasts if necessary.
- The forecasts should be carried out at the level of individual stores.
- Users will be interested in their own regions/stores in any one session and not be concerned with global trends.
- The number of requests for updated forecasts in any one session will be small.

Given these requirements, we can work with the business to create the following user stories, which we can put into a tool such as Jira, as explained in *Chapter 2, The Machine Learning Development Process*. Some examples of user stories covering these requirements would be the following:

- **User Story 1:** As a local logistics planner, I want to log in to a dashboard in the morning at 09:00 and be able to see forecasts of item demand at the store level for the next few days so that I can understand transport demand ahead of time.
- **User Story 2:** As a local logistics planner, I want to be able to request an update of my forecast if I see it is out of date. I want the new forecast to be returned in under 5 minutes so that I can make decisions on transport demand effectively.
- **User Story 3:** As a local logistics planner, I want to be able to filter for forecasts for specific stores so that I can understand what stores are driving demand and use this in decision-making.

These user stories are very important for the development of the solution as a whole. As we are focused on the ML engineering aspects of the problem, we can now dive into what these mean for building the solution.

For example, the desire to *be able to see forecasts of item demand at the store level* can be translated quite nicely into a few technical requirements for the ML part of the solution. This tells us that the target variable will be the number of items required on a particular day. It tells us that our ML model or models need to be able to work at the store level, so either we have one model per store or the concept of the store can be taken in as some sort of feature.

Similarly, the requirement that the user wants to *be able to request an update of my forecast if I see it is out of date ... I want the new forecast to be retrieved in under five minutes* places a clear latency requirement on training. We cannot build something that takes days to retrain, so this may suggest that one model built across all of the data may not be the best solution.

Finally, the request *I want to be able to filter for forecasts for specific stores* again supports the notion that whatever we build must utilize some sort of store identifier in the data but not necessarily as a feature for the algorithm. So, we may want to start thinking of application logic that will take a request for the forecast for a specific store, identified by this store ID, then the ML model and forecast are retrieved only for that store via some kind of lookup or retrieval that uses this ID in a filter.

Walking through this process, we can see how just a few lines of requirements have allowed us to start fleshing out how we will tackle the problem in practice. Some of these thoughts and others could be consolidated upon a little brainstorming among our team for the project in a table like that of *Table 8.1*:

User	Details	Technical

Story		Requirements
1	<p>As a local logistics planner, I want to log in to a dashboard in the morning at 09:00 and be able to see forecasts of item demand at the store level for the next few days so that I can understand transport demand ahead of time.</p>	<ul style="list-style-type: none"> • Target variable = item demand. • Forecast horizon – 1-7 days. • API access for a dashboard or other visualization solution.
2	<p>As a local logistics planner, I want to be able to request an update of my forecast if I see it is out of date. I want the new forecast to be returned in under 5 minutes so that I can make decisions on transport demand effectively.</p>	<ul style="list-style-type: none"> • Lightweight retraining. • Model per store.
3	<p>As a local logistics planner, I want to be able to filter for forecasts for specific stores so that I can understand what stores are driving demand and use this in decision-making.</p>	<ul style="list-style-type: none"> • Model per store.

Table 8.1: Translating user stories to technical requirements.

Now we will build on our understanding of the problem by starting to pull together a design for the ML piece of the solution.

Designing our forecasting service

The requirements in the *Understanding the forecasting problem* section are the definitions of the targets we need to hit, but they are not the method for getting there. Drawing on our understanding of design and architecture from *Chapter 5, Deployment Patterns and Tools*, we can start building out our design.

First, we should confirm what kind of design we should be working on. Since we need dynamic requests, it makes sense that we follow the microservice architecture discussed in *Chapter 5, Deployment Patterns and Tools*. This will allow us to build a service that has the sole focus of retrieving the right model from our model store and performing the requested inference. The prediction service should therefore have interfaces available between the dashboard and the model store.

Furthermore, since a user may want to work with a few different store combinations in any one session and maybe switch back and forth between the forecasts of these, we should provide a mechanism for doing so that is performant.

It is also clear from the scenario that we can quite easily have a very high volume of requests for predictions but a lower request for model updates. This means that separating out training and prediction will make sense and that we can follow the train-persist process outlined in *Chapter 3, From*

Model to Model Factory. This will mean that prediction will not be dependent on a full training run every time and that retrieval of models for prediction is relatively fast.

What we have also gathered from the requirements is that our training system doesn't necessarily need to be triggered by drift monitoring in this case, but by dynamic requests made by the user. This adds a bit of complexity as it means that our solution should not retrain for every request coming in but be able to determine whether retraining is worth it for a given request or whether the model is already up to date. For example, if four users log on and are looking at the same region/store/item combination and all request a retrain, it is pretty clear that we do not need to retrain our model four times! Instead, what should happen is that the training system registers a request, performs a retrain, and then safely ignores the other requests.

There are several ways to serve ML models, as we have discussed several times throughout this book. One very powerful and flexible way is to wrap the models, or the model serving logic, into a standalone service that is limited to only performing tasks required for the serving of the ML inference. This is the serving pattern that we will consider in this chapter and it is the classic “microservice” architecture, where different pieces of functionality are broken down into their own distinct and separated services. This builds resiliency and extensibility into your software systems so it is a great pattern to become comfortable with. This is also particularly amenable to the development of ML systems, as these have to consist of training, inference, and monitoring services, as outlined in *Chapter 3, From Model to Model Factory*. This chapter will walk through how to serve an ML model using a microservice architecture, using a few different

approaches with various pros and cons. You will then be able to adapt and build on these examples in your own future projects.

We can bring these design points together into a high-level design diagram, for example, in *Figure 8.1*:

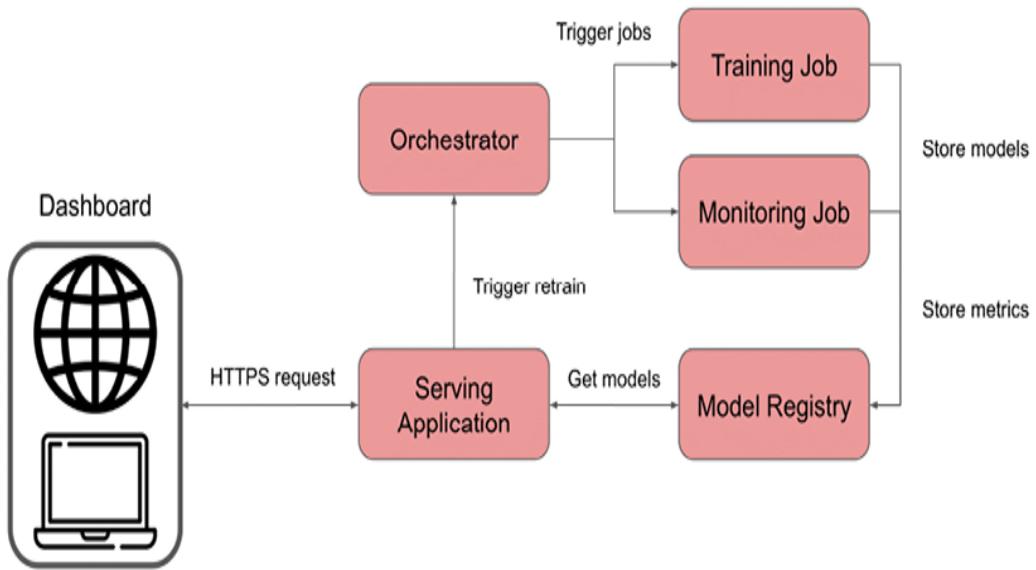


Figure 8.1: High-level design for the forecasting microservice.

The next section will focus on taking these high-level design considerations to a lower level of detail as we perform some tool selection ahead of development.

Selecting the tools

Now that we have a high-level design in mind and we have written down some clear technical requirements, we can begin to select the toolset we will use to implement our solution.

One of the most important considerations on this front will be what framework we use for modeling our data and building our forecasting

functionality. Given that the problem is a time-series modeling problem with a need for fast retraining and prediction, we can consider the pros and cons of a few options that may fit the bill before proceeding.

The results of this exercise are shown in *Table 8.2*:

Tool/Framework	Pros	Cons
Scikit-learn	<ul style="list-style-type: none">• Already understood by almost all data scientists.• Very easy-to-use syntax.• Lots of great community support.• Good feature engineering and pipelining support.	<ul style="list-style-type: none">• No native time-series modeling capabilities (but the popular <code>sktime</code> package does have these).• Will require more feature engineering to apply models to time-series data.
Prophet	<ul style="list-style-type: none">• Purely focused on forecasting.• Has inbuilt hyperparameter optimization capabilities.	<ul style="list-style-type: none">• Not as commonly used as scikit-learn (but still relatively popular).• Underlying methods are quite sophisticated – may lead to black box usage by data scientists.

	<ul style="list-style-type: none"> • Provides a lot of functionality out of the box. • Often gives accurate results on a wide variety of problems. • Provides confidence intervals out of the box. 	<ul style="list-style-type: none"> • Not inherently scalable.
Spark ML	<ul style="list-style-type: none"> • Natively scalable to large volumes. • Good feature engineering and pipelining support. 	<ul style="list-style-type: none"> • No native time-series modeling capabilities. • Algorithm options are relatively limited. • Can be harder to debug.

Table 8.2: The considered pros and cons of some different ML toolkits for solving this forecasting problem.

Based on the information in *Table 8.2*, it looks like the **Prophet** library would be a good choice and offer a nice balance between predictive power, the desired time-series capabilities, and experience among the developers and scientists on the team.

The data scientists could then use this information to build a proof-of-concept, with code much like that shown in *Chapter 1, Introduction to ML*

Engineering, in the *Example 2: Forecasting API* section, which applies Prophet to a standard retail dataset.

This covers the ML package we will use for modeling, but what about the other components? We need to build something that allows the frontend application to request actions be taken by the backend, so it is a good idea to consider some kind of web application framework. We also need to consider what happens when this backend application is hit by many requests, so it makes sense to build it with scale in mind. Another consideration is that we are tasked with training not one but several models in this use case, one for each retail store, and so we should try and parallelize the training as much as possible. The last pieces of the puzzle are going to be the use of a model management tool and the need for an orchestration layer in order to trigger training and monitoring jobs on a schedule or dynamically.

Putting all of this together, we can make some design decisions about the lower-level tooling required on top of using the Prophet library. We can summarize these in the following list:

1. **Prophet:** We met the Prophet forecasting library in *Chapter 1, Introduction to ML Engineering*. Here we will provide a deeper dive into that library and how it works before developing a training pipeline to create the types of forecasting models we saw for that retail use case in the first chapter.
2. **Kubernetes:** As discussed in *Chapter 6, Scaling Up*, this is a platform for orchestrating multiple containers across compute clusters and allows you to build highly scalable ML model-serving solutions. We will use this to host the main application.

3. **Ray Train:** We already met Ray in *Chapter 6, Scaling Up*. Here we will use Ray Train to train many different Prophet forecasting models in parallel, and we will also allow these jobs to be triggered upon a request to the main web service handling the incoming requests.
4. **MLflow:** We met MLflow in *Chapter 3, From Model to Model Factory*, and this will be used as our model registry.
5. **FastAPI:** For Python, the go-to backend web frameworks are typically Django, Flask, and FastAPI. We will use FastAPI to create the main backend routing application that will serve the forecasts and interact with the other components of the solution. FastAPI is a web framework designed to be simple to use and for building highly performant web applications and is currently being used by some high-profile organizations, including Uber, Microsoft, and Netflix (according to the FastAPI homepage).



There has been some recent discussion around the potential for memory leaks when using FastAPI, especially for longer-running services. This means that ensuring you have enough RAM on the machines running your FastAPI endpoints can be very important. In many cases, this does not seem to be a critical issue, but it is an active topic being discussed within the FastAPI community. For more on this, please see

<https://github.com/tiangolo/fastapi/discussions/9082>. Other frameworks, such as **Litestar**, <https://litestar.dev/>, do not seem to have the same issue, so feel free to play around with different web frameworks for the serving layer in the following example

and in your projects. FastAPI is still a very useful framework with lots of benefits so we will proceed with it in this chapter; it is just important to bear this point in mind.

In this chapter, we are going to focus on the components of this system that are relevant to serving the models at scale, as the scheduled train and retrain aspects will be covered in *Chapter 9, Building an Extract, Transform, Machine Learning Use Case*. The components we focus on can be thought to consist of our “serving layer,” although I will show you how to use Ray to train several forecasting models in parallel.

Now that we have made some tooling choices, let’s get building our ML microservice!

Training at scale

When we introduced Ray in *Chapter 6, Scaling Up*, we mentioned use cases where the data or processing time requirements were such that using a very scalable parallel computing framework made sense. What was not made explicit is that sometimes these requirements come from the fact that we actually want to train *many models*, not just one model on a large amount of data or one model more quickly. This is what we will do here.

The retail forecasting example we described in *Chapter 1, Introduction to ML Engineering* uses a data set with several different retail stores in it. Rather than creating one model that could have a store number or identifier as a feature, a better strategy would perhaps be to train a forecasting model for each individual store. This is likely to give better accuracy as the features of the data at the store level which may give some predictive power will not be averaged out by the model looking at a combination of all the

stores together. This is therefore the approach we will take, and this is where we can use Ray's parallelism to train multiple forecasting models simultaneously.

To use **Ray** to do this, we need to take the training code we had in *Chapter 1*, and adapt it slightly. First, we can bring together the functions we had for pre-processing the data and for training the forecasting models. Doing this means that we are creating one serial process that we can then distribute to run on the shard of the data corresponding to each store. The original functions for preprocessing and training the models were:

```
import ray
import ray.data
import pandas as pd
from prophet import Prophet

def prep_store_data(
    df: pd.DataFrame,
    store_id: int = 4,
    store_open: int = 1
) -> pd.DataFrame:
    df_store = df[
        (df['Store'] == store_id) &
        (df['Open'] == store_open)
    ].reset_index(drop=True)
    df_store['Date'] = pd.to_datetime(df_store[
        df_store.rename(columns= {'Date': 'ds', 'Sa
    return df_store.sort_values('ds', ascending

def train_predict(
    df: pd.DataFrame,
```

```
    train_fraction: float,
    seasonality: dict
) -> tuple[pd.DataFrame, pd.DataFrame, pd.DataFrame]
# grab split data
train_index = int(train_fraction*df.shape[0])
df_train = df.copy().iloc[0:train_index]
df_test = df.copy().iloc[train_index:]
#create Prophet model
model=Prophet(
    yearly_seasonality=seasonality['yearly'],
    weekly_seasonality=seasonality['weekly'],
    daily_seasonality=seasonality['daily'],
    interval_width = 0.95
)
# train and predict
model.fit(df_train)
predicted = model.predict(df_test)
return predicted, df_train, df_test, train_index
```

We can now combine these into a single function that will take a `pandas DataFrame`, preprocess that data, train a Prophet forecasting model and then return predictions on the test set, the training dataset, the test dataset and the size of the training set, here labelled by the `train_index` value. Since we wish to distribute the application of this function, we need to use the `@ray.remote` decorator that we introduced in *Chapter 6, Scaling Up*. We pass in the `num_returns=4` argument to the decorator to let Ray know that this function will return four values in a tuple.

```
@ray.remote(num_returns=4)
def prep_train_predict(
    df: pd.DataFrame,
    store_id: int,
    store_open: int=1,
    train_fraction: float=0.8,
    seasonality: dict={'yearly': True, 'weekly'}
) -> tuple[pd.DataFrame, pd.DataFrame, pd.DataFrame]:
    df = prep_store_data(df, store_id=store_id,
    return train_predict(df, train_fraction, se
```

Now that we have our remote function we just need to apply it. First, we assume that the dataset has been read into a `pandas` DataFrame in the same manner as in *Chapter 1, Introduction to ML Engineering*. The assumption here is that the dataset is small enough to fit in memory and doesn't require computationally intense transformations. This has the advantage of allowing us to use `pandas` relatively smart data ingestion logic, which allows for various formatting of the header row for example, as well as apply any filtering or transformation logic we want to before distribution using that now familiar `pandas` syntax. If the dataset was larger or the transformations more intense, we could have used the `ray.data.read_csv()` method from the Ray API to read the data in as a Ray Dataset. This reads the data into an Arrow data format, which has its own data manipulation syntax.

Now, we are ready to apply our distributed training and testing. First, we can retrieve all of the store identifiers from the dataset, as we are going to train a model for each one.

```
store_ids = df['Store'].unique()
```

Before we do anything else we will initialize the Ray cluster using the `ray.init()` command we met in *Chapter 6, Scaling Up*. This avoids performing the initialization when we first call the remote function, meaning we can get accurate timings of the actual processing if we perform any benchmarking. To aid performance, we can also use `ray.put()` to store the pandas DataFrame in the Ray object store. This stops us replicating this dataset every time we run a task. Putting an object in the store returns an id, which you can then use for function arguments just like the original object.

```
ray.init(num_cpus=4)
df_id = ray.put(df)
```

Now, we need to submit our Ray tasks to the cluster. Whenever you do this, a Ray object reference is returned that will allow you to retrieve the data for the process when we use `ray.get` to collect the results. The syntax I've used here may look a bit complicated, but we can break it down piece by piece. The core Python function `map`, just applies the list operation to all of the elements of the result of the `zip` syntax. The `zip(*iterable)` pattern allows us to unzip all of the elements in the list comprehension, so that we can have a list of prediction object references, training data object references, test data object references and finally the training index object references. Note the use of `df_id` for referencing the stored dataframe in the object store.

```
pred_obj_refs, train_obj_refs, test_obj_refs, t
    list,
    zip(*([prep_train_predict.remote(df_id, sto
)

```

We then need to get the actual results of these tasks, which we can do by using `ray.get()` as discussed.

```
ray_results = {
    'predictions': ray.get(pred_obj_refs),
    'train_data': ray.get(train_obj_refs),
    'test_data': ray.get(test_obj_refs),
    'train_indices': ray.get(train_index_obj_re
}

```

You can then access the values of these for each model with `ray_results['predictions'][<index>]` and so on.

In the Github repository, the file

`Chapter08/train/train_forecasters_ray.py` runs this syntax and an example for loop for training the Prophet models one by one in serial fashion for comparison. Using the `time` library for the measurements and running the experiment on my Macbook with four CPUs being utilized by the Ray cluster, I was able to train 1,115 Prophet models in just under 40 seconds using Ray, compared to around 3 minutes 50 seconds using the serial code. That's an almost six-fold speedup, without doing much optimization!



We did not cover the saving of the models and metadata into MLFlow, which you can do using the syntax we discussed in depth in *Chapter 3, From Model to Model Factory*. To avoid lots of communication overhead, it may be best to store the metadata temporarily as the result of the training process, like we have done in the dictionary storing the predictions and then write everything to MLFlow at the end. This means you do not slow down the Ray processes with communications to the MLFlow server. Note also that we could have optimized this parallel processing even further by using the Ray Dataset API discussed and altering the transformation logic to use Arrow syntax. A final option would also have been to use **Modin**, previously known as Pandas on Ray, which allows you to use `pandas` syntax whilst leveraging the parallelism of Ray.

Let's now start building out the serving layer for our solution, so that we can use these forecasting models to generate results for other systems and users.

Serving the models with FastAPI

The simplest and potentially most flexible approach to serving ML models in a microservice with Python is in wrapping the serving logic inside a lightweight web application. Flask has been a popular option among Python users for many years but now the FastAPI web framework has many

advantages, which means it should be seriously considered as a better alternative.

Some of the features of FastAPI that make it an excellent choice for a lightweight microservice are:

- **Data validation:** FastAPI uses and is based on the **Pydantic** library, which allows you to enforce type hints at runtime. This allows for the implementation of very easy-to-create data validation steps that make your system way more robust and helps avoid edge case behaviors.
- **Built-in async workflows:** FastAPI gives you asynchronous task management out of the box with `async` and `await` keywords, so you can build the logic you will need in many cases relatively seamlessly without resorting to extra libraries.
- **Open specifications:** FastAPI is based on several open source standards including the **OpenAPI REST API standard** and the **JSON Schema** declarative language, which helps create automatic data model documentation. These specs help keep the workings of FastAPI transparent and very easy to use.
- **Automatic documentation generation:** The last point mentioned this for data models, but FastAPI also auto-generates documentation for your entire service using SwaggerUI.
- **Performance:** Fast is in the name! FastAPI uses the **Asynchronous Server Gateway Interface (ASGI)** standard, whereas other frameworks like Flask use the **Web Server Gateway Interface (WSGI)**. The ASGI can process more requests per unit of time and does so more efficiently, as it can execute tasks without waiting for previous tasks to finish. The WSGI interface executes specified tasks sequentially and so takes longer to process requests.

So, the above are the reasons why it might be a good idea to use FastAPI to serve the forecasting models in this example, but how do we go about doing that? That is what we will now cover.

Any microservice has to have data come into it in some specified format; this is called the “request.” It will then return data, known as the “response.” The job of the microservice is to ingest the request, execute a series of tasks that the request either defines or gives input for, create the appropriate output, and then transform that into the specified request format. This may seem basic, but it is important to recap and gives us the starting point for designing our system. It is clear that we will have to take into account the following points in our design:

- 1. Request and response schemas:** Since we will be building a REST API, it is natural that we will specify the data model for the request and responses as JSON objects with associated schemas. The key when doing this is that the schemas are as simple as possible and that they contain all the information necessary for the client (the requesting service) and the server (the microservice) to perform the appropriate actions. Since we are building a forecasting service, the request object must provide enough information to allow the system to provide an appropriate forecast, which the upstream solution calling the service can present to users or perform further logic on. The response will have to contain the actual forecast data points or some pointer toward the location of the forecast.
- 2. Compute:** The creation of the response object, in this case, a forecast, requires computation, as discussed in *Chapter 1, Introduction to ML Engineering*.

A key consideration in designing ML microservices is the size of this compute resource and the appropriate tooling needed to execute it. As

an example, if you are running a computer vision model that requires a large GPU in order to perform inference, you cannot do this on the server running the web application backend if that is only a small machine running a CPU. Similarly, if the inference step requires the ingestion of a terabyte of data, this may require us to use a parallelization framework like Spark or Ray running on a dedicated cluster, which by definition will have to be running on different machines from the serving web application. If the compute requirements are small enough and fetching data from another location is not too intense, then you may be able to run the inference on the same machine hosting the web application.

3. **Model management:** This is an ML service, so, of course, there are models involved! This means, as discussed in detail in *Chapter 3, From Model to Model Factory*, we will need to implement a robust process for managing the appropriate model versions. The requirements for this example also mean that we have to be able to utilize many different models in a relatively dynamic fashion. This will require some careful consideration and the use of a model management tool like MLflow, which we met in *Chapter 3* as well. We also have to consider our strategies for updating and rolling back models; for example, will we use blue/green deployments or canary deployments, as discussed in *Chapter 5, Deployment Patterns and Tools*.
4. **Performance monitoring:** For any ML system, as we have discussed at length throughout the book, monitoring the performance of models will be critically important, as will taking appropriate action to update or roll back these models. If the truth data for any inference cannot be immediately given back to the service, then this will require its own

process for gathering together truth and inferences before performing the desired calculations on them.

These are some of the important points we will have to consider as we build our solution. In this chapter, we will focus on points 1 and 3, as *Chapter 9* will cover how to build training and monitoring systems that run in a batch setting. Now that we know some of the things we want to factor into our solution, let's get on and start building!

Response and request schemas

If the client is asking for a forecast for a specific store, as we are assuming in the requirements, then this means that the request should specify a few things. First, it should specify the store, using some kind of store identifier that will be kept in common between the data models of the ML microservice and the client application.

Second, the time range for the forecast should be provided in an appropriate format that can be easily interpreted and serviced by the application. The systems should also have logic in place to create appropriate forecast time windows if none are provided in the request, as it is perfectly reasonable to assume if a client is requesting “a forecast for store X,” then we can assume some default behavior that provides a forecast for some time period from now into the future will likely be useful to the client application.

The simplest request JSON schema that satisfies this is then something like:

```
{  
    "storeId": "4",  
    "beginDate": "2023-03-01T00:00:00Z",  
    "endDate": "2023-03-01T12:00:00Z",  
    "interval": "1H",  
    "storeType": "Grocery",  
    "forecastType": "Sales",  
    "forecastHorizon": "1W",  
    "confidenceLevel": "95%",  
    "storeName": "Marketplace",  
    "storeAddress": "123 Main Street",  
    "storeCity": "Anytown",  
    "storeState": "CA",  
    "storeZip": "90210",  
    "storeLat": 34.0522, "storeLon": -118.2437}
```

```
        "endDate": "2023-03-07T00:00:00Z"  
    }  
}
```

As this is a JSON object, all of the fields are strings, but they are populated with values that will be easily interpretable within our Python application. The Pydantic library will also help us to enforce data validation, which we will discuss later. Note that we should also allow for the client application to request multiple forecasts, so we should allow for this JSON to be extended to allow for lists of request objects:

```
[  
  {  
    "storeId": "2",  
    "beginDate": "2023-03-01T00:00:00Z",  
    "endDate": "2023-03-07T00:00:00Z"  
  },  
  {  
    "storeId": "4",  
    "beginDate": "2023-03-01T00:00:00Z",  
    "endDate": "2023-03-07T00:00:00Z"  
  }  
]
```

As mentioned, we would like to build our application logic so that the system would still work even if the client only made a request specifying the `store_id`, and then we infer the appropriate forecast horizon to be from now to some time in the future.

This means our application should work when the following is submitted as the JSON body in the API call:

```
[  
  {  
    "storeId": "4",  
  }  
]
```

To enforce these constraints on the request, we can use the Pydantic functionality where we inherit from the Pydantic `BaseModel` and create a data class defining the type requirements we have just made:

```
from pydantic import BaseModel  
  
class ForecastRequest(BaseModel):  
    store_id: str  
    begin_date: str | None = None  
    end_date: str | None = None
```

As you can see, we have enforced here that the `store_id` is a string, but we have allowed for the beginning and end dates for the forecast to be given as `None`. If the dates are not specified, we could make a reasonable assumption based on our business knowledge that a useful forecast time window would be from the datetime of the request to seven days from now. This could be something that is changed or even provided as a configuration variable in the application config. We will not deal with that particular aspect here to focus on the more the more exciting stuff, so this is left as fun exercise for the reader!

The forecasting model in our case will be based on the Prophet library, as discussed, and this requires an index that contains the datetimes for the

forecast to run over. To produce this based on the request, we can write a simple helper function:

```
import pandas as pd

def create_forecast_index(begin_date: str = None
    # Convert forecast begin date
    if begin_date == None:
        begin_date = datetime.datetime.now().re
    else:
        begin_date = datetime.datetime.strptime(
            '%Y-%m-%dT%H:%M:%SZ').repl

    # Convert forecast end date
    if end_date == None:
        end_date = begin_date + datetime.timede
    else:
        end_date = datetime.datetime.strptime(e
            '%Y-%m-%dT%H:%M:%SZ').replac
    return pd.date_range(start = begin_date, en
```

This logic then allows us to create the input to the forecasting model once it is retrieved from the model storage layer, in our case, MLflow.

The response object has to return the forecast in some data format, and it is always imperative that you return enough information for the client application to be able to conveniently associate the returned object with the response that triggered its creation. A simple schema that satisfies this would be something like:

```
[  
 {  
   "request": {  
     "store_id": "4",  
     "begin_date": "2023-03-01T00:00:00Z",  
     "end_date": "2023-03-07T00:00:00Z"  
   },  
   "forecast": [  
     {  
       "timestamp": "2023-03-01T00:00:  
       "value": 20716  
     },  
     {  
       "timestamp": "2023-03-02T00:00:  
       "value": 20816  
     },  
     {  
       "timestamp": "2023-03-03T00:00:  
       "value": 21228  
     },  
     {  
       "timestamp": "2023-03-04T00:00:  
       "value": 21829  
     },  
     {  
       "timestamp": "2023-03-05T00:00:  
       "value": 21686  
     },  
     {  
       "timestamp": "2023-03-06T00:00:  
       "value": 22696  
     }  
   ]  
 }]
```

```
        },
        {
            "timestamp": "2023-03-07T00:00:",
            "value": 21138
        }
    ]
}
```



We will allow for this to be extended as a list in the same way as the request JSON schema. We will work with these schemas for the rest of this chapter. Now, let's look at how we will manage the models in the application.

Managing models in your microservice

In *Chapter 3, From Model to Model Factory*, we discussed in detail how you can use MLflow as a model artifact and metadata storage layer in your ML systems. We will do the same here, so let's assume that you have an MLflow Tracking server already running and then we just need to define our logic for interacting with it. If you need a refresher, feel free to revisit *Chapter 3*.

We will need to write some logic that does the following:

1. Checks there are models available for use in production in the MLflow server.
2. Retrieves a version of the model satisfying any criteria we wish to set, for example, that the model was not trained more than a certain

number of days ago and that it has validation metrics within a chosen range.

3. Caches the model for use and reuse during the forecasting session if desired.
4. Does all of the above for multiple models if that is required from the response object.

For point 1, we will have to have models tagged as ready for production in the MLflow model registry and then we can use the `MLflowClient()` and `mlflow pyfunc` functionality we met in *Chapter 3, From Model to Model Factory*:

```
import mlflow
import mlflow.pyfunc
from mlflow.client import MlflowClient
import os

tracking_uri = os.getenv(["MLFLOW_TRACKING_URI"])
mlflow.set_tracking_uri(tracking_uri)
client = MlflowClient(tracking_uri=tracking_uri

def get_production_model(store_id:int):
    model_name = f"prophet-retail-forecaster-st
    model = mlflow.pyfunc.load_model(
        model_uri=f"models:/{{m
    )
    return model
```

For point 2, we can retrieve the metrics for a given model by using the MLflow functionality we will describe below. First, using the name of the model, you retrieve the model's metadata:

```
model_name = f"prophet-retail-forecaster-store-  
latest_versions_metadata = client.get_latest_ve  
name=model_name  
)
```

This will return a dataset like this:

```
[<ModelVersion: creation_timestamp=168137891371  
dev/Machine-Learning-Engineering-with-Python-Se
```

You can then use that data to retrieve the version via this object and then retrieve the model version metadata:

```
latest_model_version_metadata = client.get_mode  
name=model_name,  
version=latest_versions_metadata.version  
)
```

This contains metadata that looks something like:

```
<ModelVersion: creation_timestamp=1681377954142
```

The metrics information for this model version is associated with the `run_id`, so we need to get that:

```
latest_model_run_id = latest_model_version.meta
```

The value for the `run_id` would be something like:

```
'41f163b0a6af4b63852d9218bf07adb3'
```

You can then use this information to get the model metrics for the specific run, and then perform any logic you want on top of it. To retrieve the metric values, you can use the following syntax:

```
client.get_metric_history(run_id=latest_model_r
```

As an example, you could use logic like the one applied in *Chapter 2* in the *Continuous model performance testing* section and simply demand the root mean squared error is below some specified value before allowing it to be used in the forecasting service.

We may also want to allow for the service to trigger retraining if the model is out of tolerance in terms of age; this could act as another layer of model management on top of any training systems put in place.

If our training process is orchestrated by an Airflow DAG running on AWS MWAA, as we discussed in *Chapter 5, Deployment Patterns and Tools*, then the below code could be used to invoke the training pipeline:

```
import boto3
import http.client
import base64
import ast
# mwaa_env_name = 'YOUR_ENVIRONMENT_NAME'
# dag_name = 'YOUR_DAG_NAME'

def trigger_dag(mwaa_env_name: str, dag_name: str):
    client = boto3.client('mwaa')

    # get web token
    mwaa_cli_token = client.create_cli_token(
        Name=mwaa_env_name
    )

    conn = http.client.HTTPSConnection(
        mwaa_cli_token['WebServerHostname']
    )
    mwaa_cli_command = 'dags trigger'
    payload = mwaa_cli_command + " " + dag_name
    headers = {
        'Authorization': 'Bearer ' + mwaa_cli_token['AccessToken'],
        'Content-Type': 'text/plain'
    }
    conn.request("POST", "/aws_mwaa/cli", payload)
    res = conn.getresponse()
    data = res.read()
    dict_str = data.decode("UTF-8")
    mydata = ast.literal_eval(dict_str)
    return base64.b64decode(mydata['stdout']).decode("UTF-8")
```

The next sections will outline how these pieces can be brought together so that the FastAPI service can wrap around several pieces of this logic before discussing how we containerize and deploy the app.

Pulling it all together

We have successfully defined our request and response schemas, and we've written logic to pull the appropriate model from our model repository; now all that is left to do is to tie all this together and perform the actual inference with the model. There are a few steps to this, which we will break down now. The main file for the FastAPI backend is called `app.py` and contains a few different application routes. For the rest of the chapter, I will show you the necessary imports just before each relevant piece of code, but the actual file follows the PEP8 convention of imports at the top of the file.

First, we define our logger and we set up some global variables to act as a lightweight in-memory cache of the retrieved models and service handlers:

```
# Logging
import logging

log_format = "%(asctime)s - %(name)s - %(levelname
logging.basicConfig(format = log_format, level
handlers = {}
models = {}
MODEL_BASE_NAME = f"prophet-retail-forecaster-s
```

Using global variables to pass objects between application routes is only a good idea if you know that this app will run in isolation and not create race

conditions by receiving requests from multiple clients simultaneously. When this happens, multiple processes try and overwrite the variable. You can adapt this example to replace the use of global variables with the use of a cache like **Redis** or **Memcache** as an exercise!

We then have to instantiate a `FastAPI` app object and we can define any logic we want to run upon startup by using the start-up lifespan event method:

```
from fastapi import FastAPI
from registry.mlflow.handler import MLFlowHandl

app = FastAPI()
@app.on_event("startup")
async def startup():
    await get_service_handlers()
    logging.info("Updated global service handle")
async def get_service_handlers():
    mlflow_handler = MLFlowHandler()
    global handlers
    handlers['mlflow'] = mlflow_handler
    logging.info("Retreving mlflow handler {}".
    return handlers
```

As already mentioned, FastAPI is great for supporting async workflows, allowing for the compute resources to be used while awaiting for other tasks to complete. The instantiation of the service handlers could be a slower process so this can be useful to adopt here. When functions that use the `async` keyword are called, we need to use the `await` keyword,

which means that the rest of the function where the `async` function has been called can be suspended until a result is returned and the resources used for tasks elsewhere. Here we have only one handler to instantiate, which will handle connections to the MLflow Tracking server.

The `registry.mlflow.handler` module is one I have written containing the `MLFlowHandler` class, with methods we will use throughout the app. The following are the contents of that module:

```
import mlflow
from mlflow.client import MlflowClient
from mlflow.pyfunc import PyFuncModel
import os

class MLFlowHandler:
    def __init__(self) -> None:
        tracking_uri = os.getenv('MLFLOW_TRACKI
        self.client = MlflowClient(tracking_uri)
        mlflow.set_tracking_uri(tracking_uri)

    def check_mlflow_health(self) -> None:
        try:
            experiments = self.client.search_ex
            return 'Service returning experimen
        except:
            return 'Error calling MLFlow'

    def get_production_model(self, store_id: st
        model_name = f"prophet-retail-forecaste
        model = mlflow.pyfunc.load_model(
                    model_uri=f"models
```

```
        )  
    return model
```

As you can see, this handler has methods for checking the MLflow Tracking server is up and running and getting production models. You could also add methods for querying the MLflow API to gather the metrics data we mentioned before.

Returning to the main `app.py` file now, I've written a small health check endpoint to get the status of the service:

```
@app.get("/health/", status_code=200)  
async def healthcheck():  
    global handlers  
    logging.info("Got handlers in healthcheck.")  
    return {  
        "serviceStatus": "OK",  
        "modelTrackingHealth": handlers['mlflow']  
    }
```

Next comes a method to get the production model for a given retail store ID. This function checks if the model is already available in the `global` variable (acting as a simple cache) and if it isn't there, adds it. You could expand this method to include logic around the age of the model or any other metrics you wanted to use to decide whether or not to pull the model into the application:

```
async def get_model(store_id: str):
    global handlers
    global models
    model_name = MODEL_BASE_NAME + f"{store_id}"
    if model_name not in models:
        models[model_name] = handlers['mlflow']
                                .get_producti
    return models[model_name]
```

Finally, we have the forecasting endpoint, where the client can hit this application with the request objects we defined before and get the forecasts based on the Prophet models we retrieve from MLflow. Just like in the rest of the book, I've omitted longer comments for brevity:

```
@app.post("/forecast/", status_code=200)
async def return_forecast(forecast_request: List[ForecastRequest]):
    forecasts = []
    for item in forecast_request:
        model = await get_model(item.store_id)
        forecast_input = create_forecast_index(
            begin_date=item.begin_date,
            end_date=item.end_date
        )
        forecast_result = {}
        forecast_result['request'] = item.dict()
        model_prediction = model.predict(foreca
            .rename(columns={'ds': 'timestamp',
            model_prediction['value'] = model_predi
        forecast_result['forecast'] = model_pre
```

```
    forecasts.append(forecast_result)
return forecasts
```

You can then run the app locally with:

```
uvicorn app:app --host 127.0.0.1 --port 8000
```

And you can add the `-reload` flag if you want to develop the app while it is running. If you use Postman (or `curl` or any other tool of your choice) and query this endpoint with a request body as we described previously, see *Figure 8.2*, you will get an output like that shown in *Figure 8.3*.

Chapter 7 - Forecast Example / <http://127.0.0.1:8000/forecast> - local app

POST <http://127.0.0.1:8000/forecast>

Params Authorization Headers (8) **Body** • Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 [  
2 ... {  
3 ... "store_id": "4",  
4 ... "begin_date": "2023-03-01T00:00:00Z",  
5 ... "end_date": "2023-03-07T00:00:00Z"  
6 ... },  
7 ... {  
8 ... "store_id": "3",  
9 ... "begin_date": "2023-03-01T00:00:00Z",  
10 ... "end_date": "2023-03-07T00:00:00Z"  
11 ... },  
12 ... {  
13 ... "store_id": "10",  
14 ... "begin_date": "2023-03-01T00:00:00Z",  
15 ... "end_date": "2023-03-07T00:00:00Z"  
16 ... }  
17 ]  
18
```

Figure 8.2: A request to the ML microservice in the Postman app.

Body Cookies Headers (4) Test Results 

Pretty Raw Preview Visualize **JSON** 

```
1 []
2 {
3     "request": {
4         "store_id": "4",
5         "begin_date": "2023-03-01T00:00:00Z",
6         "end_date": "2023-03-07T00:00:00Z"
7     },
8     "forecast": [
9         {
10            "timestamp": "2023-03-01T00:00:00",
11            "value": 20716
12        },
13        {
14            "timestamp": "2023-03-02T00:00:00",
15            "value": 20816
16        },
17        {
18            "timestamp": "2023-03-03T00:00:00",
19            "value": 21228
20        },
21        {
22            "timestamp": "2023-03-04T00:00:00",
23            "value": 21829
24        },
25        {

```

Figure 8.3: The response from the ML microservice when querying with Postman.

And that's it, we have a relatively simple ML microservice that will return a Prophet model forecast for retail stores upon querying the endpoint! We will now move on to discussing how we can containerize this application and deploy it to a Kubernetes cluster for scalable serving.

Containerizing and deploying to Kubernetes

When we introduced Docker in *Chapter 5, Deployment Patterns and Tools*, we showed how you can use it to encapsulate your code and then run it across many different platforms consistently.

Here we will do this again, but with the idea in mind that we don't just want to run the application as a singleton on a different piece of infrastructure, we actually want to allow for many different replicas of the microservice to be running simultaneously with requests being routed effectively by a load balancer. This means that we can take what works and make it work at almost arbitrarily large scales.

We will do this by executing several steps:

1. Containerize the application using Docker.
2. Push this Docker container to Docker Hub to act as our container storage location (you could use another container management solution like AWS Elastic Container Registry or similar solutions on another cloud provider for this step).
3. Create a Kubernetes cluster. We will do this locally using minikube, but you can do this on a cloud provider using its managed Kubernetes service. On AWS, this is **Elastic Kubernetes Service (EKS)**.

4. Define a service and load balancer on the cluster that can scale. Here we will introduce the concept of manifests for programmatically defining service and deployment characteristics on Kubernetes clusters.
5. Deploy the service and test that it is working as expected.

Let us now go through these steps in the next section.

Containerizing the application

As introduced earlier in the book, if we want to use Docker, we need to give instructions for how to build the container and install any necessary dependencies in a Dockerfile. For this application, we can use one that is based on one of the available FastAPI container images, assuming we have a file called `requirements.txt` that contains all of our Python package dependencies:

```
FROM tiangolo/uvicorn-gunicorn-fastapi:latest
COPY ./requirements.txt requirements.txt
RUN pip install --no-cache-dir --upgrade -r req
COPY ./app /app
CMD ["uvicorn", "main:app", "--host", "0.0.0.0"]
```

We can then build this Docker container using the following command, where I have named the container `custom-forecast-service`:

```
docker build -t custom-forecast-service:latest
```

Once this has been successfully built, we need to push it to Docker Hub. You can do this by logging in to Docker Hub in the terminal and then pushing to your account by running:

```
docker login  
docker push <DOCKER_USERNAME>/custom-forecast-s
```

This means that other build processes or solutions can download and run your container.

Note that before you push to Docker Hub, you can test that the containerized application runs by executing a command like the following, where I have included a platform flag in order to run the container locally on my MacBook Pro:

```
docker run -d --platform linux/amd64 -p 8000:80
```

Now that we have built and shared the container, we can now work on scaling this up with a deployment to Kubernetes.

Scaling up with Kubernetes

Working with Kubernetes can be a steep learning curve for even the most seasoned developers, so we will only scratch the surface here and give you enough to get started on your own learning journey. This section will walk you through the steps you need to deploy your ML microservice onto a Kubernetes cluster running locally, as it takes the same steps to deploy to a remotely hosted cluster (with minor modifications). Operating Kubernetes

clusters seamlessly in production requires consideration of topics such as networking, cluster resource configuration and management, security policies, and much more. Studying all of these topics in detail would require an entire book in itself. In fact, an excellent resource to get up to speed on many of these details is *Kubernetes in Production Best Practices* by Aly Saleh and Murat Karsioğlu. In this chapter, we will instead focus on understanding the most important steps to get you up and running and developing ML microservices using Kubernetes.

First, let's get set up for Kubernetes development. I will use minikube here, as it has some handy utilities for setting up services that can be called via REST API calls. Previously in this book, I used kind (Kubernetes in Docker), and you can use this here; just be prepared to do some more work and use the documentation.

To get set up with minikube on your machine, follow the installation guide in the official docs for your platform at
<https://minikube.sigs.k8s.io/docs/start/>.

Once minikube is installed, you can spin up your first cluster using default configurations with:

```
minikube start
```

Once the cluster is up and running, you can deploy the `fast-api` service to the cluster with the following command in the terminal:

```
kubectl apply -f direct-kube-deploy.yaml
```

where `direct-kube-deploy.yaml` is a manifest that contains the following code:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fast-api-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: fast-api
  template:
    metadata:
      labels:
        app: fast-api
    spec:
      containers:
        - name: fast-api
          image: electricweegie/custom-forecast-s
          resources:
            limits:
              memory: "128Mi"
              cpu: "500m"
          ports:
            - containerPort: 8000
```

This manifest defines a Kubernetes Deployment that creates and manages two replicas of a Pod template containing a container named `fast-api`, which runs the Docker image we created and published previously,

`electricweegie/custom-forecast-service:latest`. It also defines resource limits for the containers running inside the Pods and ensures that the containers are listening on port `8000`.

Now that we have created a Deployment with the application in it, we need to expose this solution to incoming traffic, preferably through a load balancer so that incoming traffic can be efficiently routed to the different replicas of the application. To do this in minikube, you have to perform a few steps:

1. Network or host machine access is not provided by default to the Services running on the minikube cluster, so we have to create a route to expose the cluster IP address using the `tunnel` command:

```
minikube tunnel
```

2. Open a new terminal window. This allows the tunnel to keep running, and then you need to create a Kubernetes Service with type `LoadBalancer` that will access the `deployment` we have already set up:

```
kubectl expose deployment fast-api-deployment
```

3. You can then get the external IP for accessing the Service by running:

```
kubectl get svc
```

This should give an output that looks something like:

NAME	TYPE
fast-api-deployment	LoadBalancer

You will then be able to use the `EXTERNAL - IP` of the load balancer service to hit the API, so you can navigate to Postman, or your other API development tool, and use `http://<EXTERNAL - IP>:8080` as the root URL for the FastAPI service that you have now successfully built and deployed to Kubernetes!

Deployment strategies

As discussed in *Chapter 5, Deployment Patterns and Tools*, there are several different strategies you can use to deploy and update your ML services. There are two components to this: one is the deployment strategy for the models and the other is the deployment strategy for the hosting application or pipelines that serve the models. These can both be executed in tandem as well.

Here we will discuss how to take the application we just deployed to Kubernetes and update it using canary and blue/green deployment strategies. Once you know how to do this for the base application, performing a similar update strategy for the models can be added in by specifying in the canary or blue/green deployment a model version that has an appropriate tag. As an example, we could use the “staging” stage of the model registry in MLflow to give us our “blue” model, and then upon transition to “green,” ensure that we have moved this model to the “production” stage in the model registry using the syntax outlined earlier in the chapter and in *Chapter 3, From Model to Model Factory*.

As a canary deployment is a deployment of the new version of the application in a smaller subset of the production environment, we can create a new deployment manifest that enforces only one replica (this could be more on larger clusters) of the canary application is created and run. In this case, this only requires that you edit the previous manifest number of replicas to “1.”

To ensure that the canary deployment is accessible to the same load balancer, we have to utilize the concept of resource labels in Kubernetes. We can then deploy a load balancer that selects resources with the desired label. An example manifest for deploying such a load balancer is given below:

```
apiVersion: v1
kind: Service
metadata:
  name: fast-api-service
spec:
  selector:
    app: fast-api
  ports:
    - protocol: TCP
      port: 8000
      targetPort: 8000
  type: LoadBalancer
```

Or using the same minkube syntax as above:

```
kubectl expose deployment fast-api-deployment -
```

After deploying this load balancer and the canary deployment, you can then implement monitoring of the logs on the cluster or on your model in order to determine if the canary is successful and should get more traffic. In that case, you would just update the deployment manifest to contain more replicas.

Blue/green deployments will work in a very similar way; in each case, you just edit the Deployment manifest to label the application as blue or green. The core difference between blue/green and canary deployments though is that the switching of the traffic is a bit more abrupt, where here we can have the load balancer service switch production traffic to the green deployment with the following command, which uses the `kubectl` CLI to patch the definition of the selector in the service:

```
kubectl patch service fast-api-service -p '{"sp' < >
```

And that is how you perform canary and blue/green deployments in Kubernetes, and how you can use it to try different versions of the forecasting service; give it a try!

Summary

In this chapter, we walked through an example of how to take the tools and techniques from the first seven chapters of this book and apply them together to solve a realistic business problem. We discussed in detail how the need for a dynamically triggered forecasting algorithm can lead very quickly to a design that requires several small services to interact seamlessly. In particular, we created a design with components responsible

for handling events, training models, storing models, and performing predictions. We then walked through how we would choose our toolset to build to this design in a real-world scenario, by considering things such as appropriateness for the task at hand, as well as likely developer familiarity. Finally, we carefully defined the key pieces of code that would be required to build the solution to solve the problem repeatedly and robustly.

In the next, and final, chapter, we will build out an example of a batch ML process. We will name the pattern that this adheres to **Extract, Transform, Machine Learning**, and explore what key points should be covered in any project aiming to build this type of solution.

Join our community on Discord

Join our community's Discord space for discussion with the author and other readers:

<https://packt.link/mle>



9

Building an Extract, Transform, Machine Learning Use Case

Similar to *Chapter 8, Building an Example ML Microservice*, the aim of this chapter will be to try to crystallize a lot of the tools and techniques we have learned about throughout this book and apply them to a realistic scenario. This will be based on another use case introduced in *Chapter 1, Introduction to ML Engineering*, where we imagined the need to cluster taxi ride data on a scheduled basis. So that we can explore some of the other concepts introduced throughout the book, we will assume as well that for each taxi ride, there is also a series of textual data from a range of sources, such as traffic news sites and transcripts of calls between the taxi driver and the base, joined to the core ride information. We will then pass this data to a **Large Language Model (LLM)** for summarization. The result of this summarization can then be saved in the target data location alongside the basic ride date to provide important context for any downstream investigations or analysis of the taxi rides. We will also build on our previous knowledge of Apache Airflow, which we will use as the orchestration tool for our pipelines, by discussing some more advanced concepts to make your Airflow jobs more robust, maintainable, and scalable. We will explore this scenario so that we can outline the key

decisions we would make if building a solution in the real world, as well as discuss how to implement it by leveraging what has been covered in other chapters.

This use case will allow us to explore what is perhaps the most used pattern in **machine learning (ML)** solutions across the world—that of the batch inference process. Due to the nature of retrieving, transforming, and then performing ML on data, I have termed this **Extract, Transform, Machine Learning (ETML)**.

We will work through this example in the following sections:

- Understanding the batch processing problem
- Designing an ETML solution
- Selecting the tools
- Executing the build

All of these topics will help us understand the particular decisions and steps we need to take in order to build a successful ETML solution.

In the next section, we will revisit the high-level problem introduced in *Chapter 1, Introduction to ML Engineering*, and explore how to map the business requirements to technical solution requirements, given everything we have learned in the book so far.

Technical requirements

As in the other chapters, to create the environment to run the code examples in this chapter you can run:

```
conda env create -f mlewp-chapter09.yml
```

This will include installs of Airflow, PySpark, and some supporting packages. For the Airflow examples, we can just work locally, and assume that if you want to deploy to the cloud, you can follow the details given in *Chapter 5, Deployment Patterns and Tools*. If you have run the above `conda` command then you will have installed Airflow locally, along with PySpark and the Airflow PySpark connector package, so you can run Airflow as standalone with the following command in the terminal:

```
airflow standalone
```

This will then instantiate a local database and all relevant Airflow components. There will be a lot of output to the terminal, but near the end of the first phase of output, you should be able to spot details about the local server that is running, including a generated user ID and password. See *Figure 9.1* for an example.

```
webserver | [2023-05-21 21:55:46 +0100] [4488] [INFO] Starting gunicorn 20.1.0
webserver | [2023-05-21 21:55:46 +0100] [4488] [INFO] Listening at: http://0.0.0.0:8080 (4488)
webserver | [2023-05-21 21:55:46 +0100] [4488] [INFO] Using worker: sync
webserver | [2023-05-21 21:55:46 +0100] [4499] [INFO] Booting worker with pid: 4499
webserver | [2023-05-21 21:55:46 +0100] [4500] [INFO] Booting worker with pid: 4500
webserver | [2023-05-21 21:55:46 +0100] [4501] [INFO] Booting worker with pid: 4501
webserver | [2023-05-21 21:55:46 +0100] [4502] [INFO] Booting worker with pid: 4502
standalone |
standalone | Airflow is ready
standalone | Login with username: admin password: pqrXyuMhWyP5sMTP
standalone | Airflow Standalone is for development purposes only. Do not use this in production!
standalone |
```

Figure 9.1: Example of local login details created upon running Airflow in standalone mode. As the message says, do not use this mode for production!

If you navigate to the URL provided (in the second line of the screenshot you can see that the app is `Listening at` `http://0.0.0.0:8080`), you will see a page like that shown in *Figure*

9.2, where you can use the local username and password to log in (see *Figure 9.3*). When you log in to the standalone version of Airflow, you are presented with many examples of DAGs and jobs that you can base your own workloads on.

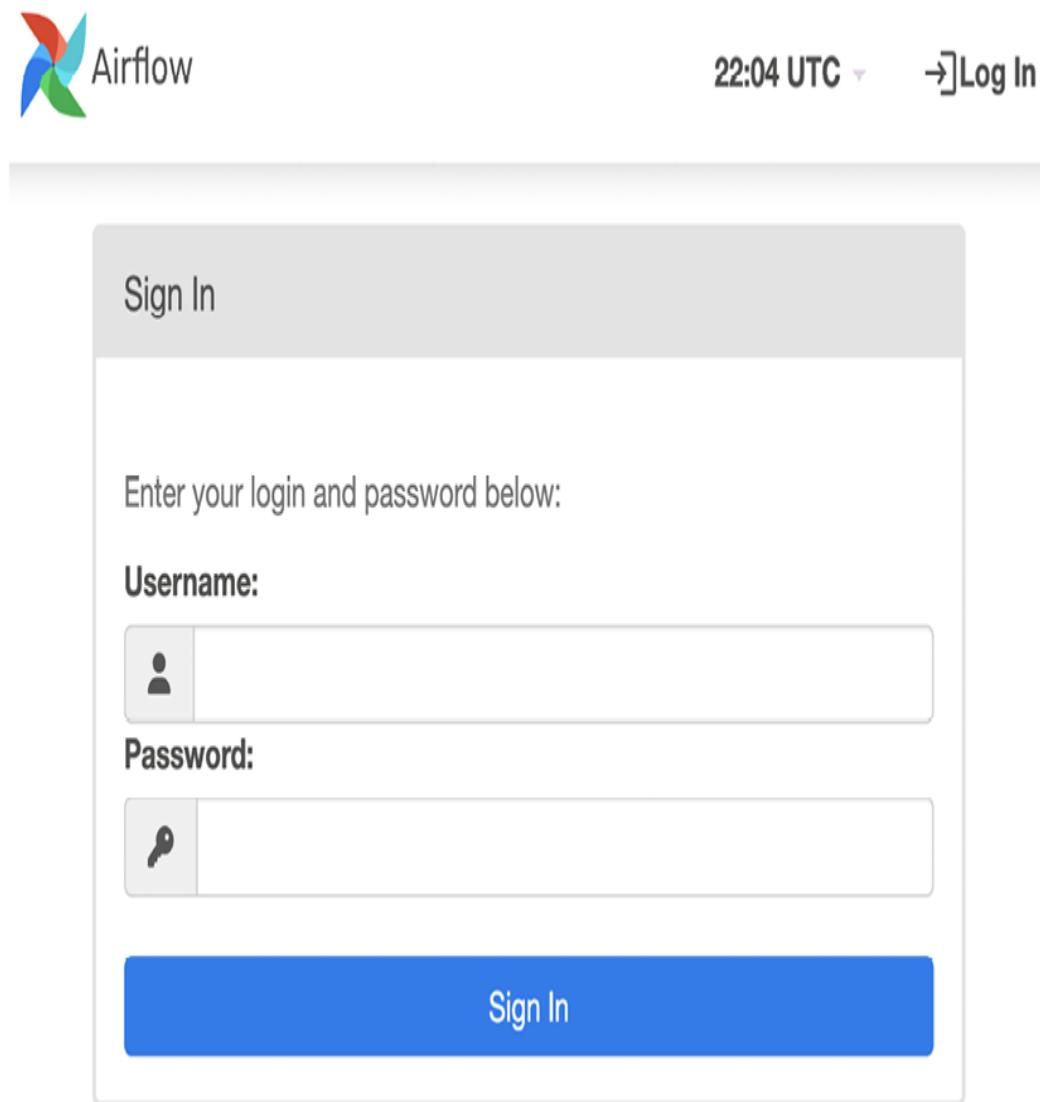


Figure 9.2: The login page of the standalone Airflow instance running on a local machine.

The screenshot shows the Airflow web interface's 'DAGs' page. At the top, there are navigation links for 'Airflow', 'DAGs', 'Datasets', 'Security', 'Browse', 'Admin', and 'Docs'. The top right corner shows the time '22:08 UTC' and a user icon labeled 'AU'. Below the header, the word 'DAGs' is displayed in a large, bold font. Underneath, there are several filters: 'All 45' (selected), 'Active 0', 'Paused 45', 'Filter DAGs by tag', 'Search DAGs', and 'Auto-refresh' with a refresh icon. The main content area is a table listing six example DAGs:

DAG	Owner	Runs	Schedule	Last Run	Next Run	Recent Tasks
dataset_consumes_1	airflow	00000	Dataset	On s3://dag1/output_1.txt		0000000
dataset_consumes_1_and_2	airflow	00000	Dataset	0 of 2 datasets updated		0000000
dataset_consumes_1_never_scheduled	airflow	00000	Dataset	0 of 2 datasets updated		0000000
dataset_consumes_unknown_never_scheduled	airflow	00000	Dataset	0 of 2 datasets updated		0000000
dataset_produces_1	airflow	00000	@daily	2023-05-20, 00:00:00		0000000
dataset_produces_2	airflow	00000	None			0000000

Figure 9.3: The landing page after logging in to the standalone Airflow instance. The page has been populated with a series of example DAGs.

Now that we have done some of the preliminary setup, let's move on to discussing the details of the problem we will try to solve before we build out our solution.

Understanding the batch processing problem

In *Chapter 1, Introduction to ML Engineering*, we saw the scenario of a taxi firm that wanted to analyze anomalous rides at the end of every day. The customer had the following requirements:

- Rides should be clustered based on ride distance and time, and anomalies/outliers identified.

- Speed (distance/time) was not to be used, as analysts would like to understand long-distance rides or those with a long duration.
- The analysis should be carried out on a daily schedule.
- The data for inference should be consumed from the company's data lake.
- The results should be made available for consumption by other company systems.

Based on the description in the introduction to this chapter, we can now add some extra requirements:

- The system's results should contain information on the rides classification as well as a summary of relevant textual data.
- Only anomalous rides need to have textual data summarized.

As we did in *Chapter 2, The Machine Learning Development Process*, and *Chapter 8, Building an Example ML Microservice*, we can now build out some user stories from these requirements, as follows:

- **User story 1:** As an operations analyst or data scientist, I want to be given clear labels of rides that are anomalous when considering their ride times in minutes and ride distances in miles so that I can perform further analysis and modeling on the volume of anomalous rides. The criteria for what counts as an anomaly should be determined by an appropriate ML algorithm, which defines an anomaly with respect to the other rides for the same day.
- **User story 2:** As an operations analyst or data scientist, I want to be provided with a summary of relevant textual data so that I can do further analysis and modeling on the reasons for some rides being anomalous.

- **User story 3:** As an internal application developer, I want all output data sent to a central location, preferably in the cloud, so that I can easily build dashboards and other applications with this data.
- **User story 4:** As an operations analyst or data scientist, I would like to receive a report every morning by 09.00. This report should clearly show which rides were anomalous or “normal” as defined by the selected ML algorithm. This will enable me to update my analyses and provide an update to the logistics managers.

User story 1 should be taken care of by our general clustering approach, especially since we are using the **Density-Based Spatial Clustering of Applications with Noise (DBSCAN)** algorithm, which provides a label of -1 for outliers.

User story 2 can be accommodated by leveraging the capabilities of LLMs that we discussed in *Chapter 7, Deep Learning, Generative AI, and LLMOps*. We can send the textual data we are given as part of the input batch to a GPT model with an appropriately formatted prompt; the prompt formatting can be done with LangChain or vanilla Python logic.

User story 3 means that we have to push the results to a location on the cloud that can then be picked up either by a data engineering pipeline or a web application pipeline. To make this as flexible as possible, we will push results to an assigned **Amazon Web Services (AWS) Simple Storage Service (S3)** bucket. We will initially export the data in the **JavaScript Object Notation (JSON)** format, which we have already met in several chapters, as this is a format that is often used in application development and can be read in by most data engineering tools.

The final user story, user story 4, gives us guidance on the scheduling we require for the system. In this case, the requirements mean we should run a

daily batch job.

Let's tabulate these thoughts in terms of some ML solution technical requirements, as shown in *Table 9.1*.

User Story	Details	Technical Requirements
1	As an operations analyst or data scientist, I want to be given clear labels of rides that have anomalously long ride times or distances so that I can perform further analysis and modeling on the volume of anomalous rides.	<ul style="list-style-type: none">• Algorithm type = anomaly detection/clustering/outlier detection.• Features = ride time and distance.
2	As an operations analyst or data scientist, I want to be provided with a summary of relevant textual data so that I can do further analysis and modeling on the reasons for some rides being anomalous.	<ul style="list-style-type: none">• Algorithm type = text summarization.• Potential models = transformers like BERT, and LLMs like GPT models.• Input requirements = formatted prompts.
3	As an internal application developer, I want all output data sent to a central location,	System output destination = S3 on AWS.

	preferably in the cloud, so that I can easily build dashboards and other applications with this data.	
4	As an operations analyst or data scientist, I would like to see the output data for the previous day's rides every morning so that I can update my analyses and provide an update to the logistics managers.	Batch frequency = daily.

Table 9.1: Translating user stories to technical requirements.

The process of taking some user stories and translating them into potential technical requirements is a very important skill for an ML engineer, and it can really help speed up the design and implementation of a potential solution. For the rest of the chapter, we will use the information in *Table 9.1*, but to help you practice this skill, can you think of some other potential user stories for the scenario given and what technology requirements these might translate to? Here are some thoughts to get you started:

- A data scientist in the firm may want to try and build models to predict customer satisfaction based on a variety of features of the rides, including times and



perhaps any of the traffic issues mentioned in the textual data. How often may they want this data? What data would they need? What would they do with it specifically?

- The developers of the mobile app in the firm may want to have forecasts of expected ride times based on traffic and weather conditions to render for users. How could they do this? Can the data come in batches, or should it be an event-driven solution?
- Senior management may want reports compiled of the firm's performance across several variables in order to make decisions. What sort of data may they want to see? What ML models would provide more insight? How often does the data need to be prepared, and what solutions could the results be shown in?

Now that we have done some of the initial work required to understand what we need the system to do and how it may do it, we can now move on to bringing these together into some initial designs.

Designing an ETML solution

The requirements clearly point us to a solution that takes in some data and augments it with ML inference, before outputting the data to a target location. Any design we come up with must encapsulate these steps. This is the description of any ETML solution, and this is one of the most used patterns in the ML world. In my opinion it will remain important for a long time to come as it is particularly suited to ML applications where:

- **Latency is not critical:** If you can afford to run on a schedule and there are no high-throughput or low-latency response time requirements, then running as an ETL batch is perfectly acceptable.
- **You need to batch the data for algorithmic reasons:** A great example of this is the clustering approach we will use here. There are ways to perform clustering in an online setting, where the model is continually updated as new data comes in, but some approaches are simpler if you have all the relevant data taken together in the relevant batch. Similar arguments can apply to deep learning models, which will require large batches of data to be processed on GPUs in parallel for maximum efficiency.
- **You do not have event-based or streaming mechanisms available:** Many organizations may still operate in batch mode simply because they have to! It can require investment to move to appropriate platforms that work in a different mode, and this may not have always been made available.
- **It is simpler:** Related to the previous point, getting event-based or streaming systems set up can take some learning for your team, whereas batching is relatively intuitive and easy to get started with.

Now, let's start discussing the design. The key elements our design has to cover were articulated in *Table 9.1*. We can then start to build out a design diagram that covers the most important aspects, including starting to pin down which technologies are used for which processes. *Figure 9.4* shows a simplified design diagram that starts to do this and shows how we can use an Airflow pipeline to pull data from an S3 bucket, storing our clustered data in S3 as an intermediate data storage step, before proceeding to summarize the text data using the LLM and exporting the final result to our final target S3 location.

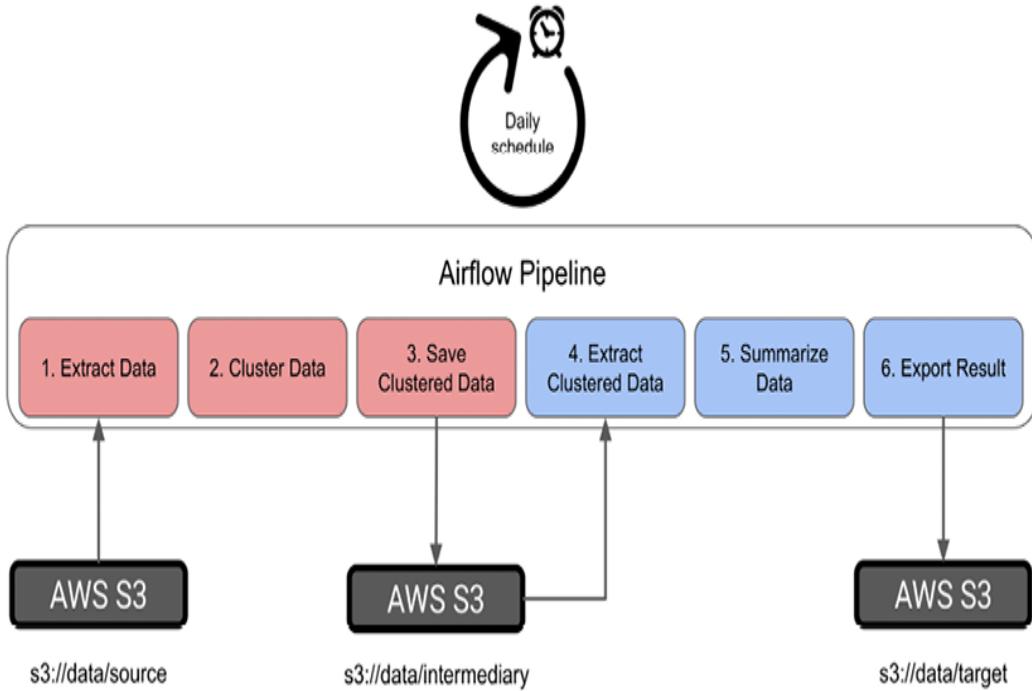


Figure 9.4: High-level design for the ETL clustering and summarization system. Steps 1–3 of the overall pipelines are the clustering steps and 4–6 are the summarization steps.

In the next section, we will look at some potential tools we can use to solve this problem, given everything we have learned in previous chapters.

Selecting the tools

For this example, and pretty much whenever we have an ETL problem, our main considerations boil down to a few simple things, namely the selection of the interfaces we need to build, the tools we need to perform the transformation and modeling at the scale we require, and how we orchestrate all of the pieces together. The next few sections will cover each of these in turn.

Interfaces and storage

When we execute the extract and load parts of ETML, we need to consider how to interface with the systems that store our data. It is important that whichever database or data technology we extract from, we use the appropriate tools to extract at whatever scale and pace we need. In this example, we can use S3 on AWS for our storage; our interfacing can be taken care of by the AWS `boto3` library and the AWS CLI. Note that we could have selected a few other approaches, some of which are listed in *Table 9.2* along with their pros and cons.

Potential Tools	Pros	Cons
The AWS CLI, S3, and <code>boto3</code>	<ul style="list-style-type: none"> • Relatively simple to use, and extensive documentation. • Connects to a wide variety of other AWS tools and services. 	<ul style="list-style-type: none"> • Not cloud-agnostic. • Not applicable to other environments or technologies.
SQL database and JDBC/ODBC connector	<ul style="list-style-type: none"> • Relatively tool-agnostic. • Cross-platform and cloud. • Optimized storage and querying possible. 	<ul style="list-style-type: none"> • Requires data modeling and database administration. • Not optimized for unstructured data.
Vendor-	<ul style="list-style-type: none"> • Often good 	<ul style="list-style-type: none"> • Requires data

supplied cloud data warehouse via their API	<p>documentation and examples to follow.</p> <ul style="list-style-type: none"> • Good optimizations in place. • Modern platforms have good connectivity to other well-used platforms. • Managed services available across multiple clouds. 	<p>modeling and database administration.</p> <ul style="list-style-type: none"> • Can sometimes support unstructured data but is not always easy to implement. • Can be expensive.
---	--	--

Table 9.2: Data storage and interface options for the ETL solution with some potential pros and cons.

Based on these options, it seems like using the AWS CLI, S3 and the `boto3` package will be the simplest mechanism that provides the most flexibility in this scenario. In the next section, we will consider the decisions we must make around the scalability of our modeling approach. This is very important when working with batches of data, which could be extremely large in some cases.

Scaling of models

In *Chapter 6, Scaling Up*, we discussed some of the mechanisms to scale up our analytics and ML workloads. We should ask ourselves whether any of these, or even other methods, apply to the use case at hand and use them accordingly. This works the other way too: if we are looking at relatively

small amounts of data, there is no need to provision a large amount of infrastructure, and there may be no need to spend time creating very optimized processing. Each case should be examined on its own merits and within its own context.

We have listed some of the options for the clustering part of the solution and their pros and cons in *Table 9.3*.

Potential Tools	Pros	Cons
Spark ML	Can scale to very large datasets.	Requires cluster management. Can have large overheads for smaller datasets or processing requirements. Relatively limited algorithm set.
Spark with pandas User-Defined Function (UDF)	Can scale to very large datasets. Can use any Python-based algorithm.	Might not make sense for some problems where parallelization is not easily applicable.
Scikit-learn	Well known by many data scientists. Can run on many different types of	Not very inherently scalable.

	<p>infrastructure.</p> <p>Small overheads for training and serving.</p>	
Ray AIR or Ray Serve	<p>Relatively easy-to-use API.</p> <p>Good integration with many ML libraries.</p>	<ul style="list-style-type: none"> • Requires cluster management with new types of clusters (Ray clusters). • Newer skillset for ML engineers.

Table 9.3: Options for the modeling part of the ETML solution, with their pros and cons and with a particular focus on scalability and ease of use.

Given these options, and assuming that the data volumes are not large for this example, we can comfortably stick with the Scikit-learn modeling approach, as this provides maximum flexibility and will likely be most easily usable by the data scientists in the team. It should be noted that the conversion of the Scikit-learn code to using a pandas UDF in Spark can be accomplished at a later date, with not too much work, if more scalable behavior is required.

As explained above, however, clustering is only one part of the “ML” in this ETML solution, the other being the text summarization part. Some potential options and pros and cons are shown in *Table 9.4*.

Potential Tools	Pros	Cons
GPT-X models	<ul style="list-style-type: none"> • Simple to use – we’ve 	<ul style="list-style-type: none"> • Can become

from OpenAI (or another vendor)	<p>already met this in <i>Chapter 7, Deep Learning, Generative AI, and LLM Ops.</i></p> <ul style="list-style-type: none"> Potentially the most performant models available. 	<p>expensive.</p> <ul style="list-style-type: none"> Not as in control of the model. Lack of visibility of data and model lineage.
Open-source LLMs	<ul style="list-style-type: none"> More transparent lineage of the data and model. More stable (you are in control of the model). 	<ul style="list-style-type: none"> Large infrastructure requirements. Requires very specialized skills if optimization is required. More operational management is required (LLM Ops).
Any other non-LLM deep learning model, e.g., a BERT variant	<ul style="list-style-type: none"> Can be easier to set up than some open-source LLMs. Extensively studied and documented. Easier to retrain and fine-tune (smaller models). 	<ul style="list-style-type: none"> More operational overhead than an API call (but less than hosting an LLM). Less performant.

	<ul style="list-style-type: none"> • Vanilla MLOps applicable. • May not require prompt engineering. 	
--	--	--

Table 9.4: Tooling options for the text summarization component of the EMTL solution.

Now that we have explored the tooling decisions we have to make around scalable ML models, we will move on to another important topic for ETML solutions—how we manage the scheduling of batch processing.

Scheduling of ETML pipelines

The kind of batch process that ETML corresponds to is often something that ties in quite nicely with daily batches, but given the other two points outlined previously, we may need to be careful about when we schedule our jobs—for example, a step in our pipeline may need to connect to a production database that does not have a read replica (a copy of the database specifically just for reading). If this were the case, then we may cause major performance issues for users of any solutions utilizing that database if we start hammering it with queries at 9 a.m. on a Monday morning. Similarly, if we run overnight and want to load into a system that is undergoing other batch upload processes, we may create resource contention, slowing down the process. There is no *one-size-fits-all* answer here; it is just important to consider your options. We look at the pros and cons of using some of the tools we have met throughout the book for the scheduling and job management of this problem in the following table:

Potential	Pros	Cons

Tools		
Apache Airflow	<ul style="list-style-type: none"> • Good scheduling management. • Relatively easy-to-use API. • Good documentation. • Cloud-hosted services are available, such as AWS Managed Workflows for Apache Airflow (MWAA). • Flexible for use across ML, data engineering, and other workloads. 	<ul style="list-style-type: none"> • Can take time to test pipelines. • Cloud services like MWAA can be expensive. • Airflow is relatively general-purpose (potentially also a pro), and does not have too much specific functionality for ML workloads.
ZenML	<ul style="list-style-type: none"> • Relatively easy-to-use API. • Good documentation. • Designed for ML engineers. • Multiple useful MLOps integrations. 	<ul style="list-style-type: none"> • Can take time to test pipelines. • Slightly steeper learning curve compared to Airflow.

	<ul style="list-style-type: none"> • A cloud option is available. 	
Kubeflow	<ul style="list-style-type: none"> • Relatively easy-to-use API. • Good documentation. • Simplifies the use of Kubernetes substantially if that is required. 	<ul style="list-style-type: none"> • Requires an AWS variant for use on AWS. • Slightly steeper learning curve than the others. • Can be harder to debug at times due to Kubernetes under the hood.

Table 9.5: Pros and cons of using Apache Airflow to manage our scheduling.

Given what we have in *Tables 9.3, 9.4, and 9.5*, all of the options considered have really strong pros and not too many cons. This means that there are many possible combinations of technology that will allow us to solve our problems. The requirements for our problem have stipulated that we have relatively small datasets that need to be processed in batches every day, first with some kind of clustering or anomaly detection algorithm before further analysis using an LLM. We can see that selecting `scikit-learn` for the modeling package, a GPT model from OpenAI called via the API, and Apache Airflow for orchestration can fit the bill. Again, this is not the only combination we could have gone with. It might be fun for you to take the example we work through in the rest of the chapter and try some of the other tools. Knowing multiple ways to do something is a key skill for an ML engineer that can help you adapt to many different situations.

The next section will discuss how we can proceed with the execution of the solution, given this information.

Executing the build

Execution of the build, in this case, will be very much about how we take the **proof-of-concept** code shown in *Chapter 1, Introduction to ML Engineering*, and then split this out into components that can be called by another scheduling tool such as Apache Airflow.

This will provide a showcase of how we can apply some of the ML engineering skills we learned throughout the book. In the next few sections, we will focus on how to build out an Airflow pipeline that leverages a series of different ML capabilities, creating a relatively complex solution in just a few lines of code.

Building an EML pipeline with advanced Airflow features

We already discussed Airflow in detail in *Chapter 5, Deployment Patterns and Tools*, but there we covered more of the details around how to deploy your DAGs on the cloud. Here we will focus on building in more advanced capabilities and control flows into your DAGs. We will work locally here on the understanding that when you want to deploy, you can use the process outlined in *Chapter 5*.

First we will look at some good DAG design practices. Many of these are direct applications of some of the good software engineering practices we discussed throughout the book; for a good review of some of these you can go back to *Chapter 4, Packaging Up*. Here we will emphasize how these can be applied to Airflow:

- **Embody separation of concerns for your tasks:** As discussed in *Chapter 4*, separation of concerns is all about ensuring that specific pieces of code or software perform specific functions with minimal overlap. This can also be phrased as “atomicity,” using the analogy of building up your solution with “atoms” of specific, focused functionality. At the level of Airflow DAGs we can embody this principle by ensuring our DAGs are built of tasks that have one clear job to do in each case. So, in this example we clearly have the “extract,” “transform,” “ML,” and “load” stages, for which it makes sense to have specific tasks in each case. Depending on the complexity of those tasks they may even be split further. This also helps us to create good control flows and error handling, as it is far easier to anticipate, test, and manage failure modes for smaller, more atomic pieces of code. We will see this in practice with the code examples in this section.
- **Use retries:** You can pass in several arguments to Airflow steps that help control how the task will operate under different circumstances. An important aspect of this is the concept of “retries,” which tells the task to, you guessed it, try the process again if there is a failure. This is a nice way to build in some resiliency to your system, as there could be all sorts of reasons for a temporary failure in a process, such as a drop in network connectivity if it includes a REST API call over HTTP. You can also introduce delays between retries and even exponential backoff, which is when your retries have increasing time delays between them. This can help if you hit an API with rate limits, for example, where the exponential backoff will mean the system is allowed to hit the endpoint again.

- **Mandate idempotency in your DAGs:** Idempotency is the quality of code that returns the same result when run multiple times on the same inputs. It can easily be assumed that most programs work this way but that is definitely not the case. We have made extensive use of Python objects in this book that contain internal states, for example, ML models in `scikit-learn` or neural networks in PyTorch. This means that it should not be taken for granted that idempotency is a feature of your solution. Idempotency is very useful, especially in EMTL pipelines, because it means if you need to perform retries then you know that this will not lead to unexpected side effects. You can enforce idempotency at the level of your DAG by enforcing it at the level of your tasks that make up the DAG. The challenge for an EMTL application is that we obviously have ML models, which I've just mentioned can be a challenge for this concept! So, some thought is required to make sure that retries and the ML steps of your pipeline can play nicely together.
- **Leverage the Airflow operators and provider package ecosystem:** Airflow comes with a large list of operators to perform all sorts of tasks, and there are several packages designed to help integrate with other tools called *provider packages*. The advice here is to **use them**. This speaks to the points discussed in *Chapter 4, Packaging Up*, about “not reinventing the wheel” and ensures that you can focus on building the appropriate logic you need for your workloads and system and not on creating boilerplate integrations. For example, we can use the Spark provider package. We can install it with:

```
pip install apache-airflow
pip install pyspark
```

```
pip install apache-airflow-providers-apache-
```

Then in a DAG we can submit a Spark application, for example one contained within a script called `spark-script.py`, for a run with:

```
from datetime import datetime
from airflow.models import DAG
from airflow.providers.apache.spark.operator
from airflow.providers.apache.spark.operator
from airflow.providers.apache.spark.operator

DAG_ID = "spark_example"
with DAG(
    dag_id=DAG_ID,
    schedule=None,
    start_date=datetime(2023, 5, 1),
    catchup=False,
    tags=["example"],
) as dag:
    submit_job = SparkSubmitOperator(
        application=\n            "${SPARK_HOME}/examples/src/main/\n            task_id="submit_job"
    )
```

- **Use `with DAG()` as `dag`:** In the example above you will see that we used this pattern. This context manager pattern is one of the three main ways you can define a DAG, the other two being using the DAG constructor and passing it to all tasks in your pipeline or using a

decorator to convert a function to a DAG. The use of a context manager means, as in any usage of it in Python, that any resources defined within the context are cleaned up correctly even if the code block exists with an exception. The mechanism that uses the constructor requires passing `dag=dag_name` into every task you define in the pipeline, which is quite laborious. Using the decorator is quite clean for basic DAGs but can become quite hard to read and maintain if you build more complex DAGs.

- **Remember to test (!):** After reading the rest of this book and becoming a confident ML engineer, I can hear you shouting at the page, “What about testing?!” , and you would be right to shout. Our code is only as good as the tests we can run on it. Luckily for us Airflow provides some out-of-the-box functionality that enables local testing and debugging of your DAGs. For debugging in your IDE or editor, if your DAG is called `dag`, like in the example above, all you need to do is add the below snippet to your DAG definition file to run the DAG in a local, serialized Python process within your chosen debugger. This does not run the scheduler; it just runs the DAG steps in a single process, which means it fails fast and gives quick feedback to the developer:

```
if __name__ == "__main__":
    dag.test()
```

We can also use `pytest` like we did in *Chapter 4, Packaging Up*, and elsewhere in the book.

Now that we have discussed some of the important concepts we can use, we will start to build up the Airflow DAG in some detail, and we will try and

do this in a way that shows you how to build some resiliency into the solution.

First, it is important to note that, for this example, we will actually perform the ETML process twice: once for the clustering component and once for the text summarization. Doing it this way means that we can use *intermediary storage* in between the steps, in this case, AWS S3 again, in order to introduce some resiliency into the system. This is so because if the second step fails, it doesn't mean the first step's processing is lost. The example we will walk through does this in a relatively straightforward way, but as always in this book, remember that the concepts can be extended and adapted to use tooling and processes of your choice, as long as the fundamentals remain solid.

Let's start building this DAG! It is a relatively short one that only contains two tasks; we will show the DAG first and then expand on the details:

```
from __future__ import annotations
import datetime
import pendulum
from airflow import DAG
from airflow.operators.python import PythonOperator
from utils.summarize import LLMSummarizer
from utils.cluster import Clusterer
import logging

logging.basicConfig(level=logging.INFO)

# Bucket name could be read in as an environment variable
bucket_name = "etml-data"
date = datetime.datetime.now().strftime("%Y%m%d")
```

```
file_name = f"taxi-rides-{date}.json"

with DAG(
    dag_id="etml_dag",
    start_date=pendulum.datetime(2021, 10, 1),
    schedule_interval="@daily",
    catchup=False,
) as dag:
    logging.info("DAG started . . .")
    logging.info("Extracting and clustering dat")
    extract_cluster_load_task = PythonOperator(
        task_id="extract_cluster_save",
        python_callable=Clusterer(bucket_name,
                                  cluster_and_1
                                  op_kwargs={"features": ["ride_dist", "r
    )

    logging.info("Extracting and summarizing da")
    extract_summarize_load_task = PythonOperato
        task_id="extract_summarize",
        python_callable=LLMSummarizer(bucket_na
    )

    extract_cluster_load_task >> extract_summar
```

As you can see from the code snippet, there are two tasks, and they follow sequentially after one another using the `>>` operator. Each task performs the following pieces of work:

- `extract_cluster_load_task`: This task will extract the input data from the appropriate S3 bucket and perform some clustering using

DBSCAN, before loading the original data joined to the model output to the intermediary storage location. For simplicity we will use the same bucket for intermediate storage, but this could be any location or solution that you have connectivity to.

- `extract_summarize_load_task`: Similarly, the first step here is to extract the data from S3 using the boto3 library. The next step is to take the data and then call the appropriate LLM to perform summarization on the text fields selected in the data, specifically those that contain information on the local news, weather, and traffic reports for the day of the batch run.

You will have noticed upon reading the DAG that the reason the DAG definition is so short is that we have abstracted away most of the logic into subsidiary modules, in line with the principles of keeping it simple, separating our concerns, and applying modularity. See *Chapter 4, Packaging Up*, for an in-depth discussion of these and other important concepts.

The first component that we use in the DAG is the functionality contained in the `Clusterer` class under `utils.cluster`. The full definition of this class is given below. I have omitted standard imports for brevity:

```
import boto3
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import DBSCAN
from utils.extractor import Extractor

model_params = {
    'eps': 0.3,
    'min_samples': 10,
```

```
}

class Clusterer:
    def __init__(
        self, bucket_name: str,
        file_name: str,
        model_params: dict = model_params
    ) -> None:
        self.model_params = model_params
        self.bucket_name = bucket_name
        self.file_name = file_name

    def cluster_and_label(self, features: list):
        extractor = Extractor(self.bucket_name,
        df = extractor.extract_data()
        df_features = df[features]
        df_features = StandardScaler().fit_tran
        db = DBSCAN(**self.model_params).fit(df)
        # Find labels from the clustering
        core_samples_mask = np.zeros_like(db.la
        core_samples_mask[db.core_sample_indexe
        labels = db.labels_
        # Add labels to the dataset and return.
        df['label'] = labels

        date = datetime.datetime.now().strftime
        boto3.client('s3').put_object(
            Body=df.to_json(orient='records'),
            Bucket=self.bucket_name,
            Key=f"clustered_data_{date}.json"
        )
```

Note that the constructor of the class contains a reference to a default set of `model_params`. These can be read in from a configuration file. I have included them here for visibility. The actual clustering and labeling method within the class is relatively simple; it just standardizes the incoming features, applies the DBSCAN clustering algorithm, and then exports the originally extracted dataset, which now includes the cluster labels. One important point to note is that the features used for clustering are supplied as a list so that this can be extended in the future if desired, or if richer data for clustering is available, simply by altering the `op_kwargs` argument supplied to the first task's `PythonOperator` object in the DAG.

After the first task, which uses the `Clusterer` class, runs successfully, a JSON is produced giving the source records and their cluster labels. Two random examples from the produced file are given in *Figure 9.5*.

```
{
    "ride_dist": 5.0956885357,
    "ride_time": 0.1182137494,
    "ride_speed": 43.105717902,
    "ride_id": 20230524134,
    "selection_idx": 1,
    "news": "It is expected to be a busy shopping day today as many retailers attempt are offering discounts to try to lure shoppers back into city centre stores after the COVID-19 pandemic and lockdown. Many are expected to make there way into Glasgow city centre today to take advantage of the discounts on offer. There is also an expected surge in activity on online shopping sites.",
    "weather": "The weather is expected to be sunny and dry over the next few days, with temperatures in the mid-teens looking set to entice people out and about.",
    "traffic": "Traffic is expected to be heavy on the M8 motorway near Glasgow today due to an influx of shoppers into the city centre.",
    "label": 0
},
{
    "ride_dist": 7.2791349999,
    "ride_time": 0.4467654754,
    "ride_speed": 16.2929666702,
    "ride_id": 20230524135,
    "selection_idx": 0,
    "news": "Reports are that there has been an accident on the M8 motorway near Glasgow due to icy conditions on the roads. No one has been seriously injured but the road is blocked and traffic is backed up for miles. The police are on the scene and are expected to aid the clearing of the scene in the next few hours.",
    "weather": "The forecast for the West of Scotland over the next few days is for cold and wet weather with icy conditions to remain. Drivers are advised to only make journeys where absolutely necessary, especially since the current cold spell has already led to a number of accidents across the region.",
    "traffic": "There is a traffic jam on the M8 motorway near Glasgow. Expect delays.",
    "label": -1
},
```

Figure 9.5: Two example records of the data produced after the clustering step in the ETL pipeline.

You may have noticed that, at the top of this example, another utility class is imported, the `Extractor` class from the `utils.extractor` module. This is simply a wrapper around some `boto3` functionality and is defined below:

```
import pandas as pd
import boto3

class Extractor:
    def __init__(self, bucket_name: str, file_n
                 self.bucket_name = bucket_name
                 self.file_name = file_name

    def extract_data(self) -> pd.DataFrame:
        s3 = boto3.client('s3')
        obj = s3.get_object(Bucket=self.bucket_
        df = pd.read_json(obj['Body'])
        return df
```

Now let's move on to the definition of the other class used in the DAG, the `LLMSummarizer` from the `utils.summarize` module:

```
openai.api_key = os.environ['OPENAI_API_KEY']

class LLMSummarizer:
    def __init__(self, bucket_name: str, file_n
                 self.bucket_name = bucket_name
                 self.file_name = file_name
```

```
def summarize(self) -> None:
    extractor = Extractor(self.bucket_name,
    df = extractor.extract_data()
    df['summary'] = ''

    df['prompt'] = df.apply(
        lambda x: self.format_prompt
            x['news'],
            x['weather'],
            x['traffic']
        ),
        axis=1
    )
    df.loc[df['label']==-1, 'summary'] = df
        'prompt'].apply(lambda x: self.g
date = datetime.datetime.now().strftime
boto3.client('s3').put_object(
    Body=df.to_json(orient='records'),
    Bucket=self.bucket_name,
    Key=f"clustered_summarized_{date}.j
)

def format_prompt(self, news: str, weather:
prompt = dedent(f'''
    The following information describes
    taxi journeys through a single day
    News: {news}
    Weather: {weather}
    Traffic: {traffic}
    Summarise the above information in
    ''')
```

```
        return prompt

def generate_summary(self, prompt: str) ->
    # Try chatgpt api and fall back if not
    try:
        response = openai.ChatCompletion.create(
            model = "gpt-3.5-turbo",
            temperature = 0.3,
            messages = [{"role": "user", "content": prompt}]
        )
        return response.choices[0].message['content']
    except:
        response = openai.Completion.create(
            model="text-davinci-003",
            prompt = prompt
        )
        return response['choices'][0]['text']
```

You can see that this class follows a similar design pattern to the one used in the `Clusterer` class, only now the method that we utilize has the role of prompting an OpenAI LLM of our choosing, using a standard template that we have hardcoded. Again, this prompt template can be extracted out into a configuration file that is packaged up with the solution, but is shown here for visibility. The prompt asks that the LLM summarizes the relevant pieces of contextual information supplied, concerning local news, weather, and traffic reports, so that we have a concise summary that can be used for downstream analysis or to render in a user interface. A final important point to note here is that the method to generate the summary, which wraps a call to OpenAI APIs, has a `try except` clause that will allow for a fallback

to a different model if the first model call experiences any issues. At the time of writing in May 2023, OpenAI APIs still show some brittleness when it comes to latency and rate limits, so steps like this allow you to build in more resilient workflows.

An example output from the application of the `LLMSummarizer` class when running the DAG is given in *Figure 9.6*.

```
{
    "ride_dist": 7.1756319208,
    "ride_time": 0.5520666183,
    "ride_speed": 12.9977645499,
    "ride_id": 20230524391,
    "selection_idx": 2,
    "news": "Economic conditions are slowly improving for the West of Scotland after a series of targeted investments in the area. Some high profile companies from across the globe have been lured to the Greater Glasgow Area after a targeted campaign by the Scottish Government to attract more investment in the area. The main pitch laid out to the investors has been centered around Scotland's excellent education system and it's very high quality of life for employees and customers.",
    "weather": "The forecast for the Greater Glasgow Area today remains overcast with a chance of showers and mild winds.",
    "traffic": "Traffic is expected to be normal today in the Greater Glasgow Area.",
    "label": -1,
    "summary": "Economic conditions in the West of Scotland are improving due to targeted investments and the attraction of high profile companies. The weather in the Greater Glasgow Area today will be overcast with a chance of showers and mild winds. Traffic is expected to be normal.",
    "prompt": "\nThe following information describes conditions relevant to taxi journeys through a single day in Glasgow, Scotland.\n\nNews: Economic conditions are slowly improving for the West of Scotland after a series of targeted investments in the area. Some high profile companies from across the globe have been lured to the Greater Glasgow Area after a targeted campaign by the Scottish Government to attract more investment in the area. The main pitch laid out to the investors has been centered around Scotland's excellent education system and it's very high quality of life for employees and customers.\nWeather: The forecast for the Greater Glasgow Area today remains overcast with a chance of showers and mild winds.\nTraffic: Traffic is expected to be normal today in the Greater Glasgow Area.\n\nSummarise the above information in 3 sentences or less.\n"
}
```

Figure 9.6: An example output from the `LLMSummarizer`; you can see that it takes in the news, weather, and traffic information and produces a concise summary that can help any downstream consumers of this data understand what it means for overall traffic conditions.

A potential area of optimization in this piece of code is around the prompt template used, as there is the potential for some nice prompt engineering to try and shape the output from the LLM to be more consistent. You could also use a tool like LangChain, which we met in *Chapter 7, Deep Learning, Generative AI, and LLM Ops*, to perform more complex prompting of the model. I leave this as a fun exercise for the reader.

Now that we have defined all the logic for our DAG and the components it uses, how will we actually configure this to run, even in standalone mode?

When we deployed a DAG to **MWAA**, the AWS-hosted and managed Airflow solution, in *Chapter 4*, you may recall that we had to send our DAG to a specified bucket that was then read in by the system.

For your own hosted or local Airflow instances the same point applies; this time we need to send the DAG to a `dags` folder, which is located in the `$AIRFLOW_HOME` folder. If you have not explicitly configured this for your installation of Airflow you will use the default, which is typically in a folder called `airflow` under your `home` directory. To find this and lots of other useful information you can execute the following command, which produces the output shown in *Figure 9.7*:

```
airflow info
```

> airflow info

Apache Airflow

version	2.6.1
executor	SequentialExecutor
task_logging_handler	airflow.utils.log.file_task_handler.FileTaskHandler
sql_alchemy_conn	sqlite:///Users/apmcm/airflow/airflow.db
dags_folder	/Users/apmcm/airflow/dags
plugins_folder	/Users/apmcm/airflow/plugins
base_log_folder	/Users/apmcm/airflow/logs
remote_base_log_folder	

System info

```
OS           Mac OS
architecture arm
uname       uname_result(system='Darwin', node='Andrews-MacBook-Pro.local',
                           release='21.4.0', version='Darwin Kernel Version 21.4.0: '
                           'Mon Oct 31 20:00:00 PDT 2022', machine='x86_64', domain='Darwin')
locale      ('en_US', 'UTF-8')
python_version 3.10.8 | packaged by conda-forge | (main, Nov 22 2022, 08:25:00)
python_location /opt/homebrew/Caskroom/miniforge/base/envs/mlewp-chapter08/t
```

Tools info

git	git version 2.39.1
ssh	OpenSSH_9.0p1, LibreSSL 3.3.6
kubectl	NOT AVAILABLE
gcloud	NOT AVAILABLE
cloud_sql_proxy	NOT AVAILABLE
mysql	NOT AVAILABLE
sqlite3	3.39.5 2022-10-14 20:58:05 554764a6e721fab307c63a4f98cd958cf
psql	NOT AVAILABLE

Paths info

Variable	Value
airflow_home	/Users/apmcm/airflow
system_path	/opt/homebrew/Caskroom/miniforge/base/envs/mlewp-chapter08/texes/App/usr/bin:/usr/bin:/bin:/usr/sbin:/sbin:/var/run/xd/codex.system/bootstrap/usr/appleinternal/bin
python_path	/opt/homebrew/Caskroom/miniforge/base/envs/mlewp-chapter08/trew/Caskroom/miniforge/base/envs/mlewp-chapter08/lib/python:/apmcm/airflow/plugins
airflow_on_path	True

Providers info

Provider's Info	
apache-airflow-providers-apache-spark	4.0.1
apache-airflow-providers-common-sql	1.4.0
apache-airflow-providers-ftp	3.3.1
apache-airflow-providers-http	4.3.0
apache-airflow-providers-imap	3.1.1
apache-airflow-providers-sqlite	3.3.2

Figure 9.7: The output from the airflow info command.

Once you have found the location of the `$AIRFLOW_HOME` folder, if there isn't a folder called `dags` already, create one. For simple, self-contained DAGs that do not use sub-modules, all you need to do to mock up deployment is to copy the DAG over to this folder, similar to how we sent our DAG to S3 in the example with MWAA in *Chapter 5*. Since we use multiple sub-modules in this example we can either decide to install them as a package, using the techniques developed in *Chapter 4, Packaging Up*, and make sure that it is available in the Airflow environment, or we can simply send the sub-modules into the same `dags` folder. For simplicity that is what we will do here, but please consult the official Airflow documentation for details on this.

Once we have copied over the code, if we access the Airflow UI, we should be able to see our DAG like in *Figure 9.8*. As long as the Airflow server is running, the DAG will run on the supplied schedule. You can also trigger manual runs in the UI for testing.

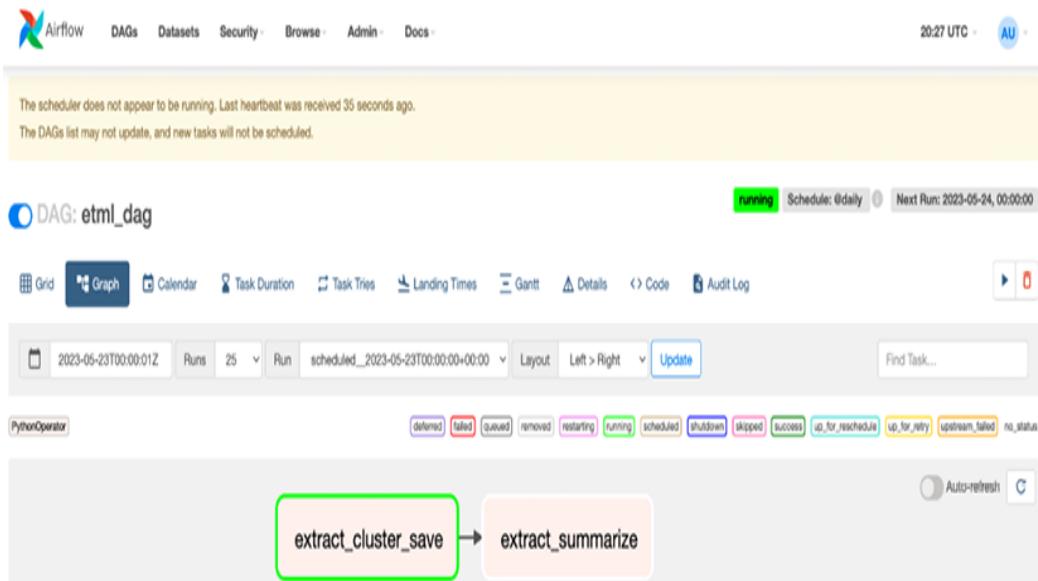


Figure 9.8: The ETML DAG in the Airflow UI.

Running the DAG will result in the intermediary and final output JSON files being created in the S3 bucket, as shown in *Figure 9.9*.

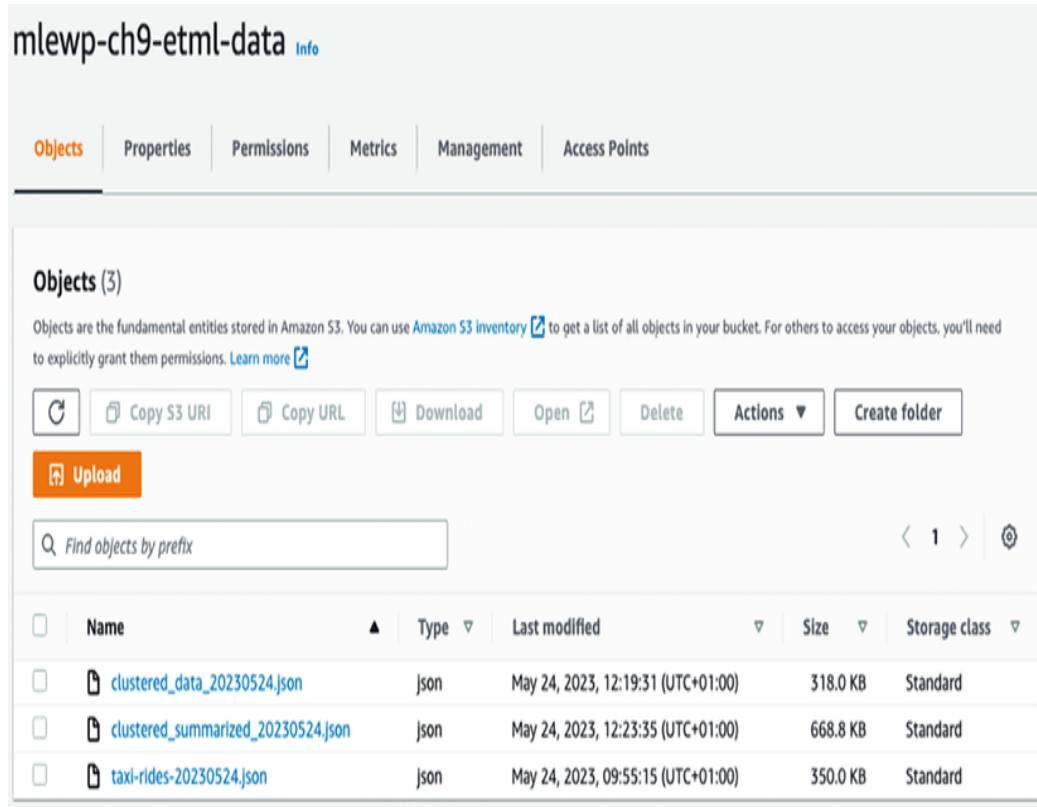


Figure 9.9: The successful run of the DAG creates the intermediate and final JSON files.

And with that, we have now built an ETML pipeline that takes in some taxi ride data, clusters this based on ride distance and time, and then performs text summarization on some contextual information using an LLM.

Summary

This chapter has covered how to apply a lot of the techniques learned in this book, in particular from *Chapter 2, The Machine Learning Development Process*, *Chapter 3, From Model to Model Factory*, *Chapter 4, Packaging Up*, and *Chapter 5, Deployment Patterns and Tools*, to a realistic

application scenario. The problem, in this case, concerned clustering taxi rides to find anomalous rides and then performing NLP on some contextual text data to try and help explain those anomalies automatically. This problem was tackled using the ETML pattern, which I offered up as a way to rationalize typical batch ML engineering solutions. This was explained in detail. A design for a potential solution, as well as a discussion of some of the tooling choices any ML engineering team would have to go through, was covered. Finally, a deep dive into some of the key pieces of work that would be required to make this solution production-ready was performed. In particular we showed how you can use good object-orientated programming techniques to wrap ML functionality spanning the Scikit-learn package, the AWS `boto3` library, and OpenAI APIs to use LLMs to create some complex functionality. We also explored in detail how to use more complex features of Airflow to orchestrate these pieces of functionality in ways that are resilient.

With that, you have not only completed this chapter but also the second edition of *Machine Learning Engineering with Python*, so congratulations! Throughout this book, we have covered a wide variety of topics related to ML engineering, from how to build your teams and what development processes could look like, all the way through to packaging, scaling, scheduling, deploying, testing, logging, and a whole bunch of stuff in between. We have explored AWS and managed cloud services, we have performed deep dives into open-source technologies that give you the ability to orchestrate, pipeline, and scale, and we have also explored the exciting new world of LLMs, generative AI, and LLMOps.

There are so many topics to cover in this fast-moving, ever-changing, and exhilarating world of ML engineering and MLOps that one book could never do the whole field justice. This second edition, however, has

attempted to improve upon the first by providing some more breadth and depth in areas I think will be important to help develop the next generation of ML engineering talent. It is my hope that you come away from reading this book not only feeling well equipped but also as excited as I am every day to go out and build the future. If you work in this space, or you are moving into it, then you are doing so at what I believe is a truly unique time in history. As ML systems are required to keep becoming more powerful, pervasive, and performant, I believe the demand for ML engineering skills is only going to grow. I hope this book has given you the tools you need to take advantage of that, and that you have enjoyed the ride as much as I have!

Join our community on Discord

Join our community's Discord space for discussion with the author and other readers:

<https://packt.link/mle>





packt.com

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

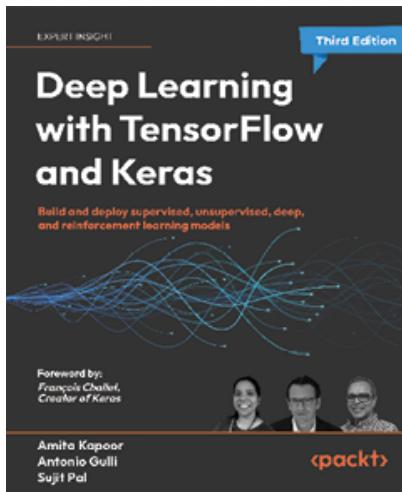
Why subscribe?

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

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

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Deep Learning with TensorFlow and Keras – Third Edition

Amita Kapoor

Antonio Gulli

Sujit Pal

ISBN: 9781803232911

- Learn how to use the popular GNNs with TensorFlow to carry out graph mining tasks

- Discover the world of transformers, from pretraining to fine-tuning to evaluating them
- Apply self-supervised learning to natural language processing, computer vision, and audio signal processing
- Combine probabilistic and deep learning models using TensorFlow Probability
- Train your models on the cloud and put TF to work in real environments
- Build machine learning and deep learning systems with TensorFlow 2.x and the Keras API

Mastering Kubernetes– Fourth Edition

Gigi Sayfan

ISBN: 9781804611395

- Learn how to govern Kubernetes using policy engines
- Learn what it takes to run Kubernetes in production and at scale
- Build and run stateful applications and complex microservices
- Master Kubernetes networking with services, Ingress objects, load balancers, and service meshes
- Achieve high availability for your Kubernetes clusters
- Improve Kubernetes observability with tools such as Prometheus, Grafana, and Jaeger
- Extend Kubernetes with the Kubernetes API, plugins, and webhooks

Packt is searching for authors like you

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

Share your thoughts

Now you've finished *Machine Learning Engineering with Python - Second Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here to go straight to the Amazon review page](#) for this book and share your feedback or leave a review on the site that you purchased it from.

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

Index

Symbols

`*args` [147](#)

`**kwargs` [147](#), [148](#)

A

abstraction [157](#)

activation function [322](#)

adaptive learning rate methods

AdaDelta [88](#)

AdaGrad [88](#)

Adam [88](#)

RMSprop [88](#)

advanced Airflow features

ETML pipeline, building with [405-417](#)

Agile

versus Waterfall [51](#)

Agile Manifesto

URL [49](#)

Airflow DAG [75](#)

Airflow role [242](#)

reference link [242](#)

Amazon CLI, configuring

reference link [215](#)

Amazon SageMaker [246](#)

Amazon Web Services (AWS) [39](#)

Apache Airflow [232-235](#)

CI/CD pipelines, setting up [244](#), [245](#)

on AWS [235-243](#)

used, for building pipelines [232](#)

using, pros and cons [403](#)

Apache Kafka [209](#)

Apache Spark [206](#)

Apache Storm [209](#)

APIs

large language models (LLMs), consuming via [342-345](#)

application

containerizing, with Docker [384](#), [385](#)

app.py [378](#)

Artifact store [247](#)

artificial general intelligence (AGI) [341](#)

artificial intelligence (AI) [319](#)

Artificial Neural Networks (ANNs) [320](#)

Asynchronous Server Gateway Interface (ASGI) [369](#)

attention vectors [341](#)

AutoKeras [126](#), [127](#)

Automated ML (AutoML) [83](#), [115](#), [124](#)

auto-sklearn [124-126](#)

AWS account

 setting up [40](#)

AWS CloudFormation [243](#)

AWS Command Line Interface (CLI) [215](#)

 reference link [215](#)

AWS Lambda [67](#), [294](#)

AWS Serverless Application Model (SAM) framework [294](#)

AWS Simple Storage Service (S3) [66](#), [206](#), [395](#)

AWS Solutions Library

 reference link [202](#)

Azure ML [246](#)

B

Bandit package

 reference link [181](#)

batch anomaly detection service

 example [15-21](#)

batch gradient descent [87](#)

batching [209](#), [210](#)

batch processing problem [394-397](#)

bias [322](#)

Bidirectional Encoder Representations from Transformers

(BERT) [331](#)

big data [206](#)

blue/green deployments [267](#), [268](#), [389](#)

versus canary deployments [389](#)

Bounded Contexts [205](#)

C

canary deployment [268](#), [388](#)

versus blue/green deployments [389](#)

checkpointing [326](#)

CI/CD example

building, with GitHub Actions [72-75](#)

CI/CD principles [235](#)

class [144](#), [153](#)

code

analyzing, for security issues [181-183](#)

packaging [157](#)

requirement to handle scenarios [196](#)

code version control [55](#)

coding styles

functional programming [154-156](#)

Object-Oriented Programming (OOP) [152-154](#)

selecting [151](#)

collections [146](#)

compute

scaling, for Ray [311](#), [312](#)

concept drift [98](#)

detecting [103-105](#)

Conda [52](#), [53](#)

constant learning rate approaches

batch gradient descent [87](#)

gradient descent [87](#)

mini-batch gradient descent [87](#)

stochastic gradient descent [87](#)

constructor [154](#)

container datatypes

reference link [147](#)

containerizing [211-214](#)

containers [146](#)

continuous integration/continuous deployment (CI/CD) [6](#), [33](#),
[67](#)

continuous model performance testing [75-78](#)

continuous model training [78-80](#)

**Covariance Matrix Adaptation Evolution Strategy (CMA-ES)
algorithm** [122](#)

CRISP-DM methodology [43](#), [44](#)

drawbacks [44](#)

versus ML engineering project [44](#)

Cross-Industry Standard Process for Data Mining (CRISP-DM)
[33](#)

Customer Resource Management (CRM) system [99](#)

D

Databricks [61](#), [67](#)

data disciplines

taxonomy, defining [4](#)

working, as effective team [8](#), [9](#)

data drift [98](#)

detecting [100-102](#)

data engineer [7](#)

data engineer, key areas of focus

access [8](#)

quality [7](#)

stability [7](#)

DataFrame APIs [273](#)

DataFrames [273](#)

data lakes [206](#), [207](#)

Data Mesh

selecting [205](#)

data pipeline, building with Python generators

reference link [146](#)

data scientist [4](#)

data scientist, key areas of focus

analysis [4](#)

modeling [4](#)

working with customer or user [5](#)

Datasets [273](#)

data warehouses [206](#)

deep learning [319-323](#)

obtaining, into production [327-331](#)

scaling, into production [327-331](#)

Deep Neural Networks (DNNs) [320](#)

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm [395](#)

deployment strategies [388](#), [389](#)

selecting [267](#), [268](#)

deploy stage [65](#)

CI/CD example, building with GitHub Actions [72-75](#)

continuous model performance testing [75-78](#)

continuous model training [78-80](#)

deployment options [66](#), [67](#)

DevOps [67-71](#)

MLOps [67-71](#)

develop stage [49](#)

code version control [55](#)

Git strategies [56-58](#)

model version control [61-65](#)

package management [51-53](#)

Poetry [53](#), [54](#)

software development methodology, selecting [49-51](#)

DevOps [67-71](#)

Directed Acyclic Graph (DAG) [136](#), [232](#), [274](#)

discovery phase [45](#)

user stories, using [45-47](#)

Django [215](#)

Docker [211](#)

installation link [213](#)

used, for containerizing application [384](#), [385](#)

Domain Name System (DNS) [230](#)

Domain-Specific Language (DSL) [262](#)

Don't Repeat Yourself (DRY) [161](#)

drift [83](#), [98](#)

diagnosing [106-108](#)

remediating [109](#)

drift, types [98](#)

concept drift [98](#)

data drift [98](#)

E

Elastic Compute Cloud (EC2) [66](#), [215](#)

Elastic Container Registry (ECR) [215](#)

pushing [216](#), [217](#)

Elastic Container Service (ECS) [215](#)

hosting [217](#)-[232](#)

Elastic Map Reduce (EMR) [282](#)

working [282](#)-[292](#)

engineering features, for machine learning

engineering categorical features [89](#)-[91](#)

engineering numerical features [91](#)-[94](#)

error handling [189](#)-[196](#)

ETML pipelines

building, with advanced Airflow features [405](#)-[417](#)

scheduling [403](#), [404](#)

ETML solution

designing [397](#)

event-based designs [208](#), [209](#)

event streaming [209](#)

pub/sub [209](#)

Evidently AI

URL [111](#)

Extract, Transform, Load (ETL) system [210](#)

Extract, Transform, Machine Learning (ETML) [210](#)

build, executing [404](#)

interfaces [399](#), [400](#)
models, scaling [400-403](#)
storage [399](#), [400](#)
tools, selecting [399](#)

F

FastAPI [363](#)

features [368](#)
using, to serve ML modes [368-370](#)

fine-tuning [331-340](#)

Flask [211](#), [215](#)

Forecasting API

example [22-26](#)

forecasting models

training [364-368](#)

forecasting problem [357](#)

forecasting service

designing [359](#), [360](#)

forward [324](#)

foundation models [340](#)

four-step methodology [44](#)

function [143](#), [144](#)

functional programming [154-156](#)

G

General Data Protection Regulation (GDPR) [181](#)

generative artificial intelligence (generative AI) [319](#)

Generative Pre-trained Transformer (GPT) [303](#), [341](#)

generators [145](#), [146](#)

Git [55](#)

Gitflow workflow [58-61](#)

GitHub [37](#)

GitHub Actions [244](#)

used, for building CI/CD example [72-75](#)

Git strategies [56](#), [57](#)

Google Cloud Platform (GCP) [39](#), [206](#)

Google Vertex AI [246](#)

gradient descent [87](#)

GraphFrames project [273](#)

Graphical User Interface (GUI) [61](#)

GraphX [273](#)

grid search [122](#)

Guardrails AI [351](#)

reference link [351](#)

H

Hadoop YARN [274](#)

hand cranking [115](#)

hidden layers [322](#)

higher-order functions [155](#)

high-level ML system design [15](#)

batch anomaly detection service example [15-21](#)

Forecasting API example [22-26](#)

streamed classification example [26-29](#)

Holistic Evaluation of Language Models (HELM) [351](#)

reference link [351](#)

Hyperopt [119-121](#)

hyperparameters

optimizing [116-118](#)

I

Identity and Access Management (IAM) roles [287](#)

immutability [154](#)

Infrastructure-as-a-Service (IaaS) [66](#)

Infrastructure-as-Code (IAC) [7](#)

input layer [322](#)

inputs [322](#)

Interactive Development Environment (IDE) [33](#)

J

JavaScript Object Notation (JSON) format [64, 395](#)

Java Virtual Machines (JVMs) [65, 273](#)

Jira [38](#)

joblib [64](#)

K

Kanban [50](#)

Keep It Simple, Stupid (KISS) [161](#)

Kubeflow [258-267](#), [302](#)

reference link [259](#)

KubeRay project

reference link [312](#)

Kubernetes [301](#), [312](#), [363](#)

architecture [302](#)

automated rollbacks [301](#)

containerizing to [384](#)

deploying to [384](#)

horizontal balancing [301](#)

load balancing [301](#)

self healing [301](#)

Kubernetes deployment

scaling up with [385-387](#)

L

label drift [102](#)

LangChain package [342](#)

large language models (LLMs) [320](#), [340-342](#), [391](#)

coding with [345-348](#)

consuming, via APIs [342-345](#)

validating [350](#), [351](#)

lazy evaluation [136](#)

library [142](#)

linter [72](#)

list comprehension [146](#)

Litestar

reference link [364](#)

LLMOps

building with [348](#)

challenges [349](#), [350](#)

logging [186-188](#)

reference link [187](#)

loss functions [86](#)

M

Machine Learning Engineering in Python (MEIP) [38](#)

machine learning (ML) [319](#)

engineering features [89](#)

Machine Learning Operations (MLOps) [6](#), [67-71](#)

Makefiles

used, for managing environment [166-170](#)

Managed Workflows for Apache Airflow (MWAA) [235](#), [414](#)

Mean Decrease in Impurity (MDI) [106](#)

merge conflict [57](#)

microservices [207](#), [208](#)

hosting, on AWS [214](#), [215](#)

Microsoft Azure [39](#), [206](#)

mini-batch gradient descent [87](#)

MIT license [37](#)

ML development process

deploy stage [65](#)

develop stage [49](#)

discovery phase [45](#)

play stage [47](#)

stages [41](#)

stages, outputs [42](#)

MLeap [65](#)

ML engineer [5](#)

ML engineering

in real world [9-12](#)

ML engineering project

versus CRISP-DM methodology [44](#)

ML engineer, key areas of focus

architecture [6](#)

productionization [6](#)

translation [5](#)

MLflow [61](#), [363](#)

ML model, retraining strategy

AutoML [124](#)

concept drift, detecting [103-105](#)

data drift, detecting [100-102](#)

drift, diagnosing [106-108](#)

drift, remediating [109](#)

hierarchies, of automation [114-116](#)

hyperparameters, optimizing [116-118](#)

limits, setting [105](#), [106](#)

tools, for monitoring [110-114](#)

training process, automating [114](#)

ML models [86](#)

adaptive learning rate methods [88](#)

constant learning rate approaches [87](#)

data, preparing [88](#), [89](#)

inference, performing [378-383](#)

losses, cutting [87](#), [88](#)

managing, in microservice [374-376](#)

retraining [98](#), [99](#)

request schemas [370-374](#)

response schemas [370-374](#)

serving, with FastAPI [368-370](#)

target, defining [86](#)

ML Operations Engineer [6](#)

ML Operations Engineer, key areas of focus

automation [7](#)

key MLOps capabilities, enabling [7](#)

platform engineering [7](#)

MLOps tools [110](#)

ML patterns

exploring [205](#)

ML pipelines

building [246](#)

Kubeflow [258-267](#)

ZenML, finding [246-258](#)

ML solution [12](#)

application layer [14](#)

compute layer [13](#)

storage layer [13](#)

ML systems

architecting [200-202](#)

building, with principles [202-205](#)

ML toolkits

cons [361](#)

pros [361](#)

model factory [85](#)

components [85](#)

model factory, with pipelines

building [131](#), [132](#)

scikit-learn pipelines [132](#)-[135](#)

spark ML pipelines [137](#), [138](#)

Spark ML pipelines [136](#)

Model Registry [128](#)

models

persisting [127](#)-[130](#)

model version control [61](#)-[64](#)

module [144](#)

monolithic [207](#)

multi-headed attention [341](#)

N

naming conventions [149](#)

neurons [320](#)

O

object [153](#)

object-oriented programming (OOP) [152](#)-[154](#), [204](#)

one-hot encoding [90](#)

on-premises deployment [66](#)

OpenAI Evals [351](#)

Open Neural Network Exchange (ONNX) [65](#)

Optuna [121-123, 339](#)

orchestrator [247](#)

ordinal encoding [90](#)

output layer [322](#)

output phase [94](#)

P

package [142-145](#)

building [164-166](#)

designing [159-164](#)

need for [157, 158](#)

package management [51-53](#)

packaging

use cases, selecting [158](#)

pandas UDFs [280](#)

parameters [116](#)

PEP 20

reference link [161](#)

pickle [64](#)

security [64](#)

versioning [64](#)

pip [52, 53](#)

pipelines [83, 131, 246](#)

building, with Airflow [232](#)

Platform-as-a-Service (PaaS) [67](#)

play stage [47](#), [48](#)

Poetry [53](#), [54](#), [170-176](#)

PEP-8 style guide

benefits, in Python projects [149](#)

prediction systems [85](#)

pre-trained classifier

applying, example [280](#), [281](#)

Principle of Least Effort [203](#)

Principle of Least Surprise [203](#)

prompt engineering [352](#)

prompt management [352](#)

PromptOps [352](#)

prompts [352](#)

Proof-of-Concept [404](#)

Prophet library [362](#)

using [363](#)

provider packages [406](#)

Proximal Policy Optimization (PPO) [342](#)

pull request [60](#)

PySpark

writing [151](#)

PyTest [177](#)

Python [14](#), [142](#), [211](#)

class [144](#)

function [143](#), [144](#)

module [144](#)

package [144](#), [145](#)

standards [149-151](#)

using, tips and tricks [145-149](#)

variable [143](#)

Python Enhancement Proposal 8 (PEP-8) [149](#)

classes [150](#)

function names [150](#)

modules [150](#)

packages [150](#)

reference link [149](#)

variables [150](#)

Python Packaging Authority (PyPA) [166](#)

reference link [166](#)

Python safety tool

reference link [183](#)

PyTorch

evaluation [327](#)

loss functions [324](#)

model.eval() [326](#)

model.train() [326](#)

reference link [323](#)

Torch.autograd [325](#)

torch.load [326](#)

Torch.nn [324](#)

Torch.optim.Optimizer [324](#)

Torch.save [326](#)

Torch.Tensor [323](#)

working with [323](#)

R

random search [122](#)

Ray [303](#), [306](#)

compute, scaling for [311](#), [312](#)

for ML [307](#)-[310](#)

scaling with [303](#)

serving layer, scaling with [312](#), [313](#)

Ray AI Runtime (AIR) [304](#)

Ray Batch Predictor [304](#)

Ray Core API [305](#)

Ray Data [304](#)

RayDP

reference link [316](#)

Ray RLib [304](#)

Ray Serve [312](#)

Ray Serving [304](#)

Ray Train [304](#), [363](#)

Ray Tune [304](#)

Rectified Linear Unit (ReLU) [324](#)

Reinforcement Learning with Human Feedback (RLHF) [342](#)

Reliable AI Markup Language (RAIL) [351](#)

request schemas [371-374](#)

response schemas [371-374](#)

Return on Investment (ROI) [12](#)

reward model [342](#)

RMSprop [88](#)

rolling deployment type [226](#)

S

scikit-learn pipelines [132-135](#)

 concepts [132](#)

Scrum [50](#)

security issues

 code, analyzing [181-183](#)

 dependencies, analyzing [183-186](#)

self-attention [341](#)

Separation of Concerns [202](#)

serverless infrastructure [293-301](#)

serverless tools, for ML solution

benefits [293](#)

Service-Level Agreements (SLAs) [105](#)

Services [220](#)

serving layer

scaling, with Ray [312](#), [313](#)

SHapley Additive exPlanation (SHAP) [107](#)

Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion (SOLID) [204](#), [205](#)

sklearn [232](#)

sklearn wine dataset [92](#)

software development methodology

selecting [49-51](#)

Software Principles

reference link [161](#)

solutions

securing [180](#), [181](#)

Spark [232](#), [272](#), [273](#)

scaling with [273](#), [274](#)

tips and tricks [274-276](#)

Spark-based computations

in cloud [282](#)

Spark cluster infrastructure [67](#)

SparkContext [274](#)

Spark MLlib [232](#), [273](#)

Spark ML pipelines [136-138](#)

Spark SQL [273](#)

standard practices [275](#)

Spark Structured Streaming [273](#)

Sprints [50](#)

SQLite database [128](#)

stacks [247](#)

stochastic gradient descent [87](#)

streamed classification

example [26-29](#)

structured APIs [273](#)

Synthetic Minority Over-Sampling Technique (SMOTE) [28](#)

systems

designing, at scale [313-316](#)

T

tasks [220](#)

TensorFlow [232](#)

Terraform [243](#)

testing [176-180](#)

text summarisation, EMTL solution

tooling options [402](#)

TF-IDF statistic [92](#)

TIOBE index

URL [15](#)

tools

selecting [361-364](#)

setting up [35-40](#)

training phase [94](#)

training system

designing [94, 95](#)

design options, training [95, 96](#)

train-persist [97, 98](#)

train-run [96, 97](#)

train-persist process [97, 98](#)

train-run process [96](#)

transfer learning [331-340](#)

Tree of Parzen Estimators (TPE) [119](#)

U

Unified Modeling Language (UML) [202](#)

Universal Approximation Theorems [321](#)

Untrained AutoEncoders (UAEs) [103](#)

use cases

selecting, for packaging [158](#)

User-Defined Function (UDF) [276](#)

building, example [276-279](#)

user stories [45, 357](#)

building, from requirements [395](#)
requirements [357](#), [358](#)
translating to technical requirements [396](#)

V

variable [143](#)
virtual private clouds (VPCs) [285](#)

W

Waterfall [49](#)
versus Agile [51](#)
Web Server Gateway Interface (WSGI) [369](#)
weights [321](#)
Wine dataset [104](#), [105](#)

Y

YARN [274](#)

Z

ZenML
finding [246](#)- [258](#)
reference link [267](#)

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

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

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

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

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781837631964>

2. Submit your proof of purchase

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