

Learn Amazon SageMaker

Second Edition

A guide to building, training, and deploying machine learning models for developers and data scientists



Julien Simon



Learn Amazon SageMaker

Second Edition

A guide to building, training, and deploying machine learning models for developers and data scientists

Julien Simon

Packt

BIRMINGHAM—MUMBAI

Learn Amazon SageMaker

Second Edition

Copyright © 2021 Packt Publishing

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

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

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

Publishing Product Manager: Ali Abidi

Senior Editor: David Sugarman

Content Development Editor: Joseph Sunil

Technical Editor: Devanshi Ayare

Copy Editor: Safis Editing

Project Coordinator: Aparna Nair

Proofreader: Safis Editing

Indexer: Pratik Shirodkar

Production Designer: Joshua Misquitta

First published: August 2020

Second published: November 2021

Production reference: 2191121

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80181-795-0

www.packt.com

Contributors

About the author

Julien Simon is a principal developer advocate for AI and **Machine Learning (ML)** at **Amazon Web Services (AWS)**. He focuses on helping developers and enterprises bring their ideas to life. He frequently speaks at conferences, blogs on the AWS Blog, as well as on Medium, and he also runs an AI/ML podcast.

Prior to joining AWS, Julien served as the CTO/VP of engineering in top-tier web start-ups over a period of 10 years, where he led large software and ops teams in charge of thousands of servers worldwide. In the process, he fought his way through a wide range of technical, business, and procurement issues, which helped him gain a deep understanding of physical infrastructure, its limitations, and how cloud computing can help.

About the reviewers

Antje Barth is a principal developer advocate for AI and ML at AWS, based in Düsseldorf, Germany. Antje is the co-author of the O'Reilly book, *Data Science on AWS*, the co-founder of the Düsseldorf chapter of Women in Big Data, and frequently speaks at AI and ML conferences and meetups around the world. She also chairs and curates content for O'Reilly AI Superstream events. Previously, Antje was an engineer at Cisco and MapR, focused on data center technologies, cloud computing, big data, and AI applications.

Brent Rabowsky is a principal data science consultant at AWS with over 10 years' experience in the field of ML. At AWS, he leverages his expertise to help AWS customers with their data science projects. Prior to AWS, he joined Amazon.com on an ML and algorithms team and previously worked on conversational AI agents for a government contractor and a research institute. He has also served as a technical reviewer of the books *Data Science on AWS*, by Chris Fregly and Antje Barth, published by O'Reilly, and *SageMaker Best Practices*, published by Packt.

Mia Champion is a HealthAI leader passionate about transformative technologies and strategic markets in the areas of life sciences, healthcare, ML/AI, and cloud computing. She has both a technical and entrepreneurial skillset that includes experience as a principal research scientist, cloud computing architect and developer, new business developer, and business strategist.

Table of Contents

Preface

Section 1: Introduction to Amazon SageMaker

1

Introducing Amazon SageMaker

Technical requirements	4	Setting up Amazon SageMaker Studio	15
Exploring the capabilities of Amazon SageMaker	4	Onboarding to Amazon SageMaker Studio	16
The main capabilities of Amazon SageMaker	5	Onboarding with the quick start procedure	16
The Amazon SageMaker API	7		
Setting up Amazon SageMaker on your local machine	10	Deploying one-click solutions and models with Amazon SageMaker JumpStart	21
Installing the SageMaker SDK with virtualenv	10	Deploying a solution	22
Installing the SageMaker SDK with Anaconda	12	Deploying a model	25
A word about AWS permissions	14	Fine-tuning a model	28
		Summary	31

2

Handling Data Preparation Techniques

Technical requirements	34	Labeling data with Amazon SageMaker Ground Truth	34
------------------------	----	--	----

Using workforces	35	Exporting a SageMaker Data Wrangler pipeline	62
Creating a private workforce	36		
Uploading data for labeling	39		
Creating a labeling job	39		
Labeling images	44		
Labeling text	46		
Transforming data with Amazon SageMaker Data Wrangler		Running batch jobs with Amazon SageMaker Processing	63
		Discovering the Amazon SageMaker Processing API	64
		Processing a dataset with scikit-learn	64
Loading a dataset in SageMaker Data Wrangler	49	Processing a dataset with your own code	72
Transforming a dataset in SageMaker Data Wrangler	50		
	57	Summary	73

Section 2: Building and Training Models

3

AutoML with Amazon SageMaker Autopilot

Technical requirements	78	Using the SageMaker Autopilot SDK	96
Discovering Amazon SageMaker Autopilot	78	Launching a job	97
Analyzing data	79	Monitoring a job	98
Feature engineering	80	Cleaning up	100
Model tuning	80	Diving deep on SageMaker Autopilot	100
Using Amazon SageMaker Autopilot in SageMaker Studio		The job artifacts	100
Launching a job	81	The data exploration notebook	102
Monitoring a job	86	The candidate generation notebook	103
Comparing jobs	89		
Deploying and invoking a model	94	Summary	107

4

Training Machine Learning Models

Technical requirements	110	Preparing data	116
Discovering the built-in algorithms in Amazon SageMaker	110	Configuring a training job	119
Supervised learning	110	Launching a training job	121
Unsupervised learning	111	Deploying a model	123
A word about scalability	112	Cleaning up	124
Training and deploying models with built-in algorithms	112	Working with more built-in algorithms	124
Understanding the end-to-end workflow	113	Regression with XGBoost	125
Using alternative workflows	114	Recommendation with Factorization Machines	127
Using fully managed infrastructure	114	Using Principal Component Analysis	135
Using the SageMaker SDK with built-in algorithms	116	Detecting anomalies with Random Cut Forest	137
		Summary	143

5

Training CV Models

Technical requirements	146	Working with RecordIO files	157
Discovering the CV built-in algorithms in Amazon SageMaker	146	Working with SageMaker Ground Truth files	163
Discovering the image classification algorithm	146	Using the built-in CV algorithms	165
Discovering the object detection algorithm	147	Training an image classification model	165
Discovering the semantic segmentation algorithm	148	Fine-tuning an image classification model	170
Training with CV algorithms	149	Training an object detection model	172
Preparing image datasets	150	Training a semantic segmentation model	175
Working with image files	150	Summary	181

6

Training Natural Language Processing Models

Technical requirements	184	Preparing data for word vectors with BlazingText	196
Discovering the NLP built-in algorithms in Amazon SageMaker	184	Preparing data for topic modeling with LDA and NTM	197
Discovering the BlazingText algorithm	185	Using datasets labeled with SageMaker Ground Truth	203
Discovering the LDA algorithm	185	Using the built-in algorithms for NLP	205
Discovering the NTM algorithm	186	Classifying text with BlazingText	205
Discovering the seq2sea algorithm	187	Computing word vectors with BlazingText	207
Training with NLP algorithms	188	Using BlazingText models with FastText	208
Preparing natural language datasets	188	Modeling topics with LDA	210
Preparing data for classification with BlazingText	189	Modeling topics with NTM	214
Preparing data for classification with BlazingText, version 2	193	Summary	218

7

Extending Machine Learning Services Using Built-In Frameworks

Technical requirements	220	Putting it all together	233
Discovering the built-in frameworks in Amazon SageMaker	220	Running your framework code on Amazon SageMaker	234
Running a first example with XGBoost	221	Using the built-in frameworks	238
Working with framework containers	225	Working with TensorFlow and Keras	239
Training and deploying locally	226	Working with PyTorch	242
Training with script mode	227	Working with Hugging Face	245
Understanding model deployment	229	Working with Apache Spark	253
Managing dependencies	231	Summary	260

8

Using Your Algorithms and Code

Technical requirements	262	Building a fully custom container for R	277
Understanding how SageMaker invokes your code	262	Coding with R and plumber	278
Customizing an existing framework container	265	Building a custom container	280
Setting up your build environment on EC2	266	Training and deploying a custom container on SageMaker	281
Building training and inference containers	266	Training and deploying with your own code on MLflow	282
Using the SageMaker Training Toolkit with scikit-learn	270	Installing MLflow	282
Building a fully custom container for scikit-learn	272	Training a model with MLflow	283
Training with a fully custom container	272	Building a SageMaker container with MLflow	285
Deploying a fully custom container	274	Building a fully custom container for SageMaker Processing	289
		Summary	291

Section 3: Diving Deeper into Training

9

Scaling Your Training Jobs

Technical requirements	296	Deciding when to scale	298
Understanding when and how to scale	296	Deciding how to scale	299
Understanding what scaling means	296	Scaling a BlazingText training job	300
Adapting training time to business requirements	297	Monitoring and profiling training jobs with Amazon SageMaker Debugger	304
Right-sizing training infrastructure	297	Viewing monitoring and profiling information in SageMaker Studio	304

Enabling profiling in SageMaker Debugger	306	Preparing the ImageNet dataset	317
Solving training challenges	309	Defining our training job	319
Streaming datasets with pipe mode	311	Training on ImageNet	320
Using pipe mode with built-in algorithms	312	Updating batch size	322
Using pipe mode with other algorithms and frameworks	313	Adding more instances	323
Simplifying data loading with MLIO	313	Summing things up	324
Training factorization machines with pipe mode	314	Training with the SageMaker data and model parallel libraries	324
Distributing training jobs	315	Training on TensorFlow with SageMaker DDP	325
Understanding data parallelism and model parallelism	315	Training on Hugging Face with SageMaker DDP	328
Distributing training for built-in algorithms	315	Training on Hugging Face with SageMaker DMP	329
Distributing training for built-in frameworks	316	Using other storage services	330
Distributing training for custom containers	316	Working with SageMaker and Amazon EFS	330
Scaling an image classification model on ImageNet	317	Working with SageMaker and Amazon FSx for Lustre	335
		Summary	338

10

Advanced Training Techniques

Technical requirements	340	Using managed spot training with object detection	344
Optimizing training costs with managed spot training	340	Using managed spot training and checkpointing with Keras	345
Comparing costs	340	Optimizing hyperparameters with automatic model tuning	349
Understanding Amazon EC2 Spot Instances	341	Understanding automatic model tuning	350
Understanding managed spot training	342		

Using automatic model tuning with object detection	351	Creating a feature group	371
Using automatic model tuning with Keras	354	Ingesting features	374
Using automatic model tuning for architecture search	359	Querying features to build a dataset	374
		Exploring other capabilities of SageMaker Feature Store	375
Exploring models with SageMaker Debugger	360	Detecting bias in datasets and explaining predictions with SageMaker Clarify	376
Debugging an XGBoost job	361		
Inspecting an XGBoost job	362	Configuring a bias analysis with SageMaker Clarify	376
Debugging and inspecting a Keras job	366	Running a bias analysis	378
		Analyzing bias metrics	379
Managing features and building datasets with SageMaker Feature Store	370	Running an explainability analysis	380
Engineering features with SageMaker Processing	370	Mitigating bias	382
		Summary	384

Section 4: Managing Models in Production

11

Deploying Machine Learning Models

Technical requirements	388	Examining and exporting Hugging Face models	394
Examining model artifacts and exporting models	389	Deploying models on real-time endpoints	396
Examining and exporting built-in models	389	Managing endpoints with the SageMaker SDK	396
Examining and exporting built-in CV models	391	Managing endpoints with the boto3 SDK	402
Examining and exporting XGBoost models	392	Deploying models on batch transformers	406
Examining and exporting scikit-learn models	393	Deploying models on inference pipelines	408
Examining and exporting TensorFlow models	394		

Monitoring prediction quality with Amazon SageMaker Model Monitor		Sending bad data	414
	409	Examining violation reports	415
Capturing data	410	Deploying models to container services	417
Creating a baseline	411	Training on SageMaker and deploying on Amazon Fargate	418
Setting up a monitoring schedule	413	Summary	425

12

Automating Machine Learning Workflows

Technical requirements	428	Implementing our first workflow	453
Automating with AWS CloudFormation	428	Adding parallel execution to a workflow	460
Writing a template	429	Adding a Lambda function to a workflow	461
Deploying a model to a real-time endpoint	432		
Modifying a stack with a change set	435	Building end-to-end workflows with Amazon SageMaker Pipelines	467
Adding a second production variant to the endpoint	438	Defining workflow parameters	468
Implementing canary deployment	440	Processing the dataset with SageMaker Processing	469
Implementing blue-green deployment	444	Ingesting the dataset in SageMaker Feature Store with SageMaker Processing	470
Automating with AWS CDK	446	Building a dataset with Amazon Athena and SageMaker Processing	471
Installing the CDK	446	Training a model	472
Creating a CDK application	446	Creating and registering a model in SageMaker Pipelines	473
Writing a CDK application	448	Creating a pipeline	474
Deploying a CDK application	450	Running a pipeline	475
Building end-to-end workflows with AWS Step Functions	452	Deploying a model from the model registry	477
Setting up permissions	452	Summary	479

13

Optimizing Prediction Cost and Performance

Technical requirements	482	Compiling and deploying an image classification model on SageMaker	498
Autoscaling an endpoint	482	Exploring models compiled with Neo	500
Deploying a multi-model endpoint	487	Deploying an image classification model on a Raspberry Pi	501
Understanding multi-model endpoints	487	Deploying models on AWS Inferentia	503
Building a multi-model endpoint with scikit-learn	487		
Deploying a model with Amazon Elastic Inference	492	Building a cost optimization checklist	504
Deploying a model with Amazon Elastic Inference	493	Optimizing costs for data preparation	504
Compiling models with Amazon SageMaker Neo	497	Optimizing costs for experimentation	505
Understanding Amazon SageMaker Neo	497	Optimizing costs for model training	506
		Optimizing costs for model deployment	508
		Summary	510

Other Books You May Enjoy

Index

Preface

Amazon SageMaker enables you to quickly build, train, and deploy machine learning models at scale without managing any infrastructure. It helps you focus on the machine learning problem at hand and deploy high-quality models by eliminating the heavy lifting typically involved in each step of the ML process. This second edition will help data scientists and ML developers to explore new features, such as SageMaker Data Wrangler, Pipelines, Clarify, Feature Store, and much more.

You'll start by learning how to use various capabilities of SageMaker as a single toolset to solve ML challenges and progress to cover features such as AutoML, built-in algorithms and frameworks, and writing your own code and algorithms to build ML models. The book will then show you how to integrate Amazon SageMaker with popular deep learning libraries, such as TensorFlow and PyTorch, to extend the capabilities of existing models. You'll see how automating your workflows can help you get to production faster with minimum effort and at a lower cost. Finally, you'll explore SageMaker Debugger and SageMaker Model Monitor to detect quality issues in training and production.

By the end of this Amazon book, you'll be able to use Amazon SageMaker on the full spectrum of ML workflows, from experimentation, training, and monitoring to scaling, deployment, and automation.

Who this book is for

This book is for software engineers, machine learning developers, data scientists, and AWS users who are new to using Amazon SageMaker and want to build high-quality machine learning models without worrying about infrastructure. Knowledge of AWS basics is required to grasp the concepts covered in this book more effectively. A solid understanding of machine learning concepts and the Python programming language will also be beneficial.

What this book covers

Chapter 1, Introducing Amazon SageMaker, provides an overview of Amazon SageMaker, what its capabilities are, and how it helps solve many pain points faced by machine learning projects today.

Chapter 2, Handling Data Preparation Techniques, discusses data preparation options. Although it isn't the core subject of the book, data preparation is a key topic in machine learning, and it should be covered at a high level.

Chapter 3, AutoML with Amazon SageMaker AutoPilot, shows how to build, train, and optimize machine learning models automatically with Amazon SageMaker AutoPilot.

Chapter 4, Training Machine Learning Models, shows how to build and train models using the collection of statistical machine learning algorithms built into Amazon SageMaker.

Chapter 5, Training Computer Vision Models, shows how to build and train models using the collection of computer vision algorithms built into Amazon SageMaker.

Chapter 6, Training Natural Language Processing Models, shows how to build and train models using the collection of natural language processing algorithms built into Amazon SageMaker.

Chapter 7, Extending Machine Learning Services Using Built-In Frameworks, shows how to build and train machine learning models using the collection of built-in open source frameworks in Amazon SageMaker.

Chapter 8, Using Your Algorithms and Code, shows how to build and train machine learning models using their own code on Amazon SageMaker, for example, R or custom Python.

Chapter 9, Scaling Your Training Jobs, shows how to distribute training jobs to many managed instances, using either built-in algorithms or built-in frameworks.

Chapter 10, Advanced Training Techniques, shows how to leverage advanced training in Amazon SageMaker.

Chapter 11, Deploying Machine Learning Models, shows how to deploy machine learning models in a variety of configurations.

Chapter 12, Automating Machine Learning Workflows, shows how to automate the deployment of machine learning models on Amazon SageMaker.

Chapter 13, Optimizing Cost and Performance, shows how to optimize model deployments, both from an infrastructure perspective and from a cost perspective.

To get the most out of this book

You will need a functional AWS account for running everything.

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

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Learn-Amazon-SageMaker-second-edition>. If there's an update to the code, it will be updated on the existing GitHub repository.

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

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781801817950_ColorImages.pdf.

Conventions used

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

Code in text: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "You can use the `describe-spot-price-history` API to collect this information programmatically."

A block of code is set as follows:

```
od = sagemaker.estimator.Estimator(  
    container,  
    role,  
    train_instance_count=2,  
    train_instance_type='ml.p3.2xlarge',  
    train_use_spot_instances=True,  
    train_max_run=3600,                      # 1 hours  
    train_max_wait=7200,                      # 2 hour  
    output_path=s3_output)
```

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

```
[<sagemaker.model_monitor.model_monitoring.MonitoringExecution  
at 0x7fdd1d55a6d8>,  
<sagemaker.model_monitor.model_monitoring.MonitoringExecution  
at 0x7fdd1d581630>,  
<sagemaker.model_monitor.model_monitoring.MonitoringExecution  
at 0x7fdce4b1c860>]
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "We can find more information about our monitoring job in the SageMaker console, in the **Processing jobs** section."

Tips or important notes

Appear like this.

Get in touch

Feedback from our readers is always welcome.

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

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

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

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

Share your thoughts

Once you've read *Learn Amazon Sagemaker, 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.

Section 1: Introduction to Amazon SageMaker

The objective of this section is to introduce you to the key concepts, help you download supporting data, and introduce you to example scenarios and use cases.

This section comprises the following chapters:

- *Chapter 1, Introducing Amazon SageMaker*
- *Chapter 2, Handling Data Preparation Techniques*

1

Introducing Amazon SageMaker

Machine learning (ML) practitioners use a large collection of tools in the course of their projects: open source libraries, deep learning frameworks, and more. In addition, they often have to write their own tools for automation and orchestration. Managing these tools and their underlying infrastructure is time-consuming and error-prone.

This is the very problem that Amazon SageMaker was designed to address (<https://aws.amazon.com/sagemaker/>). Amazon SageMaker is a fully managed service that helps you quickly build and deploy machine learning models. Whether you're just beginning with machine learning or you're an experienced practitioner, you'll find SageMaker features to improve the agility of your workflows, as well as the performance of your models. You'll be able to focus 100% on the machine learning problem at hand, without spending any time installing, managing, and scaling machine learning tools and infrastructure.

In this first chapter, we're going to learn what the main capabilities of SageMaker are, how they help solve pain points faced by machine learning practitioners, and how to set up SageMaker. This chapter will comprise the following topics:

- Exploring the capabilities of Amazon SageMaker
- Setting up Amazon SageMaker on your local machine
- Setting up Amazon SageMaker Studio
- Deploying one-click solutions and models with Amazon SageMaker JumpStart

Technical requirements

You will need an AWS account to run the examples included in this chapter. If you haven't got one already, please point your browser to <https://aws.amazon.com/getting-started/> to learn about AWS and its core concepts, and to create an AWS account. You should also familiarize yourself with the AWS Free Tier (<https://aws.amazon.com/free/>), which lets you use many AWS services for free within certain usage limits.

You will need to install and configure the AWS CLI for your account (<https://aws.amazon.com/cli/>).

You will need a working Python 3.x environment. Installing the Anaconda distribution (<https://www.anaconda.com/>) is not mandatory but is strongly encouraged as it includes many projects that we will need (Jupyter, pandas, numpy, and more).

Code examples included in the book are available on GitHub at <https://github.com/PacktPublishing/Learn-Amazon-SageMaker-second-edition>. You will need to install a Git client to access them (<https://git-scm.com/>).

Exploring the capabilities of Amazon SageMaker

Amazon SageMaker was launched at AWS re:Invent 2017. Since then, a lot of new features have been added: you can see the full (and ever-growing) list at <https://aws.amazon.com/about-aws/whats-new/machine-learning>.

In this section, you'll learn about the main capabilities of Amazon SageMaker and its purpose. Don't worry, we'll dive deep into each of them in later chapters. We will also talk about the SageMaker **Application Programming Interfaces (APIs)**, and the **Software Development Kits (SDKs)** that implement them.

The main capabilities of Amazon SageMaker

At the core of Amazon SageMaker is the ability to prepare, build, train, optimize, and deploy models on fully managed infrastructure at any scale. This lets you focus on studying and solving the machine learning problem at hand, instead of spending time and resources on building and managing infrastructure. Simply put, you can go from building to training to deploying more quickly. Let's zoom in on each step and highlight relevant SageMaker capabilities.

Preparing

Amazon SageMaker includes powerful tools to label and prepare datasets:

- **Amazon SageMaker Ground Truth:** Annotate datasets at any scale. Workflows for popular use cases are built in (image detection, entity extraction, and more), and you can implement your own. Annotation jobs can be distributed to workers that belong to private, third-party, or public workforces.
- **Amazon SageMaker Processing:** Run batch jobs for data processing (and other tasks such as model evaluation) using your own code written with scikit-learn or Spark.
- **Amazon SageMaker Data Wrangler:** Using a graphical interface, apply hundreds of built-in transforms (or your own) to tabular datasets, and export them in one click to a Jupyter notebook.
- **Amazon SageMaker Feature Store:** Store your engineered features offline in Amazon S3 to build datasets, or online to use them at prediction time.
- **Amazon SageMaker Clarify:** Using a variety of statistical metrics, analyze potential bias present in your datasets and models, and explain how your models predict.

Building

Amazon SageMaker provides you with two development environments:

- **Notebook instances:** Fully managed Amazon EC2 instances that come preinstalled with the most popular tools and libraries: Jupyter, Anaconda, and so on.
- **Amazon SageMaker Studio:** An end-to-end integrated development environment for machine learning projects, providing an intuitive graphical interface for many SageMaker capabilities. Studio is now the preferred way to run notebooks, and we recommend that you use it instead of notebook instances.

When it comes to experimenting with algorithms, you can choose from the following:

- A collection of 17 **built-in algorithms** for machine learning and deep learning, already implemented and optimized to run efficiently on AWS. No Machine learning code to write!
- A collection of built-in, open source frameworks (**TensorFlow**, **PyTorch**, **Apache MXNet**, **scikit-learn**, and more), where you simply bring your own code.
- Your own code running in your own container: custom Python, R, C++, Java, and so on.
- Algorithms and pre-trained models from AWS Marketplace for machine learning (<https://aws.amazon.com/marketplace/solutions/machine-learning>).
- Machine learning solutions and state-of-the-art models available in one click in **Amazon SageMaker JumpStart**.

In addition, **Amazon SageMaker Autopilot** uses AutoMachine learning to automatically build, train, and optimize models without the need to write a single line of Machine learning code.

Training

As mentioned earlier, Amazon SageMaker takes care of provisioning and managing your training infrastructure. You'll never spend any time managing servers, and you'll be able to focus on machine learning instead. On top of this, SageMaker brings advanced capabilities such as the following:

- **Managed storage** using either Amazon S3, Amazon EFS, or Amazon FSx for Lustre depending on your performance requirements.
- **Managed spot training**, using Amazon EC2 Spot instances for training in order to reduce costs by up to 80%.
- **Distributed training** automatically distributes large-scale training jobs on a cluster of managed instances, using advanced techniques such as data parallelism and model parallelism.
- **Pipe mode** streams infinitely large datasets from Amazon S3 to the training instances, saving the need to copy data around.
- **Automatic model tuning** runs hyperparameter optimization to deliver high-accuracy models more quickly.

- **Amazon SageMaker Experiments** easily tracks, organizes, and compares all your SageMaker jobs.
- **Amazon SageMaker Debugger** captures the internal model state during training, inspects it to observe how the model learns, detects unwanted conditions that hurt accuracy, and profiles the performance of your training job.

Deploying

Just as with training, Amazon SageMaker takes care of all your deployment infrastructure, and brings a slew of additional features:

- **Real-time endpoints** create an HTTPS API that serves predictions from your model. As you would expect, autoscaling is available.
- **Batch transform** uses a model to predict data in batch mode.
- **Amazon Elastic Inference** adds fractional GPU acceleration to CPU-based endpoints to find the best cost/performance ratio for your prediction infrastructure.
- **Amazon SageMaker Model Monitor** captures data sent to an endpoint and compares it with a baseline to identify and alert on data quality issues (missing features, data drift, and more).
- **Amazon SageMaker Neo** compiles models for a specific hardware architecture, including embedded platforms, and deploys an optimized version using a lightweight runtime.
- **Amazon SageMaker Edge Manager** helps you deploy and manage your models on edge devices.
- Last but not least, **Amazon SageMaker Pipelines** lets you build end-to-end automated pipelines to run and manage your data preparation, training, and deployment workloads.

The Amazon SageMaker API

Just like all other AWS services, Amazon SageMaker is driven by APIs that are implemented in the language SDKs supported by AWS (<https://aws.amazon.com/tools/>). In addition, a dedicated Python SDK, aka the SageMaker SDK is also available. Let's look at both, and discuss their respective benefits.

The AWS language SDKs

Language SDKs implement service-specific APIs for all AWS services: S3, EC2, and so on. Of course, they also include SageMaker APIs, which are documented here: <https://docs.aws.amazon.com/sagemaker/latest/dg/api-and-sdk-reference.html>.

When it comes to data science and machine learning, Python is the most popular language, so let's take a look at the SageMaker APIs available in `boto3`, the AWS SDK for the Python language (<https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/sagemaker.html>). These APIs are quite low-level and verbose: for example, `create_training_job()` has a lot of JSON parameters that don't look very obvious. You can see some of them in the next screenshot. You may think that this doesn't look very appealing for everyday Machine learning experimentation... and I would totally agree!

```
response = client.create_training_job(
    TrainingJobName='string',
    HyperParameters={
        'string': 'string'
    },
    AlgorithmSpecification={
        'TrainingImage': 'string',
        'AlgorithmName': 'string',
        'TrainingInputMode': 'Pipe'|'File',
        'MetricDefinitions': [
            {
                'Name': 'string',
                'Regex': 'string'
            },
        ],
        'EnableSageMakerMetricsTimeSeries': True|False
    },
    RoleArn='string',
    InputDataConfig=[
        {
            'ChannelName': 'string',
            'DataSource': {
                'S3DataSource': {
                    'S3DataType': 'ManifestFile'|'S3Prefix'|'AugmentedManifestFil',
                    'S3Uri': 'string',
                    'S3DataDistributionType': 'FullyReplicated'|'ShardedByS3Key',
                    'AttributeName': [
                        'string',
                    ]
                },
                'FileSystemDataSource': {
                    'FileSystemId': 'string',
                    'FileSystemAccessMode': 'rw'|'ro',
                    'FileSystemType': 'EFS'|'FSxLustre',
                    'DirectoryPath': 'string'
                }
            }
        }
    ]
)
```

Figure 1.1 – A (partial) view of the `create_training_job()` API in `boto3`

Indeed, these service-level APIs are not meant to be used for experimentation in notebooks. Their purpose is automation, through either bespoke scripts or Infrastructure as Code tools such as AWS CloudFormation (<https://aws.amazon.com/cloudformation>) and Terraform (<https://terraform.io>). Your DevOps team will use them to manage production, where they do need full control over each possible parameter.

So, what should you use for experimentation? You should use the Amazon SageMaker SDK.

The Amazon SageMaker SDK

The Amazon SageMaker SDK (<https://github.com/aws/sagemaker-python-sdk>) is a Python SDK specific to Amazon SageMaker. You can find its documentation at <https://sagemaker.readthedocs.io/en/stable/>.

Note

Every effort has been made to check the code examples in this book with the latest SageMaker SDK (v2.58.0 at the time of writing).

Here, the abstraction level is much higher: the SDK contains objects for models, estimators, models, predictors, and so on. We're definitely back in Machine learning territory.

For instance, this SDK makes it extremely easy and comfortable to fire up a training job (one line of code) and to deploy a model (one line of code). Infrastructure concerns are abstracted away, and we can focus on Machine learning instead. Here's an example. Don't worry about the details for now:

```
# Configure the training job
my_estimator = TensorFlow(
    entry_point='my_script.py',
    role=my_sagemaker_role,
    train_instance_type='machine learning.p3.2xlarge',
    instance_count=1,
    framework_version='2.1.0')
# Train the model
my_estimator.fit('s3://my_bucket/my_training_data/')
# Deploy the model to an HTTPS endpoint
my_predictor = my_estimator.deploy()
```

```
initial_instance_count=1,  
instance_type='machine learning.c5.2xlarge')
```

Now that we know a little more about Amazon SageMaker, let's see how we can set it up.

Setting up Amazon SageMaker on your local machine

A common misconception is that you can't use SageMaker outside of the AWS cloud. Obviously, it is a cloud-based service, and its most appealing capabilities require cloud infrastructure to run. However, many developers like to set up their development environment their own way, and SageMaker lets them do that: in this section, you will learn how to install the SageMaker SDK on your local machine or on a local server. In later chapters, you'll learn how to train and deploy models locally.

It's good practice to isolate Python environments in order to avoid dependency hell. Let's see how we can achieve this using two popular projects: `virtualenv` (<https://virtualenv.pypa.io>) and `Anaconda` (<https://www.anaconda.com/>).

Installing the SageMaker SDK with `virtualenv`

If you've never worked with `virtualenv` before, please read this tutorial before proceeding: <https://packaging.python.org/guides/installing-using-pip-and-virtual-environments/>:

1. First, let's create a new environment named `sagemaker` and activate it:

```
$ mkdir workdir  
$ cd workdir  
$ python3 -m venv sagemaker  
$ source sagemaker/bin/activate
```

2. Now, let's install `boto3`, the SageMaker SDK, and the `pandas` library (<https://pandas.pydata.org/>), which is also required:

```
$ pip3 install boto3 sagemaker pandas
```

3. Now, let's quickly check that we can import these SDKs into Python:

```
$ python3  
Python 3.9.5 (default, May 4 2021, 03:29:30)  
>>> import boto3
```

```
>>> import sagemaker
>>> print(boto3.__version__)
1.17.70
>>> print(sagemaker.__version__)
2.39.1
>>> exit()
```

The installation looks fine. Your own versions will certainly be newer and that's fine. Now, let's run a quick test with a local Jupyter server (<https://jupyter.org/>). If Jupyter isn't installed on your machine, you can find instructions at <https://jupyter.org/install>:

1. First, let's create a Jupyter kernel based on our virtual environment:

```
$ pip3 install jupyter ipykernel
$ python3 -m ipykernel install --user --name=sagemaker
```

2. Then, we can launch Jupyter:

```
$ jupyter notebook
```

3. Creating a new notebook, we can see that the sagemaker kernel is available, so let's select it in the **New** menu, as seen in the following screenshot:

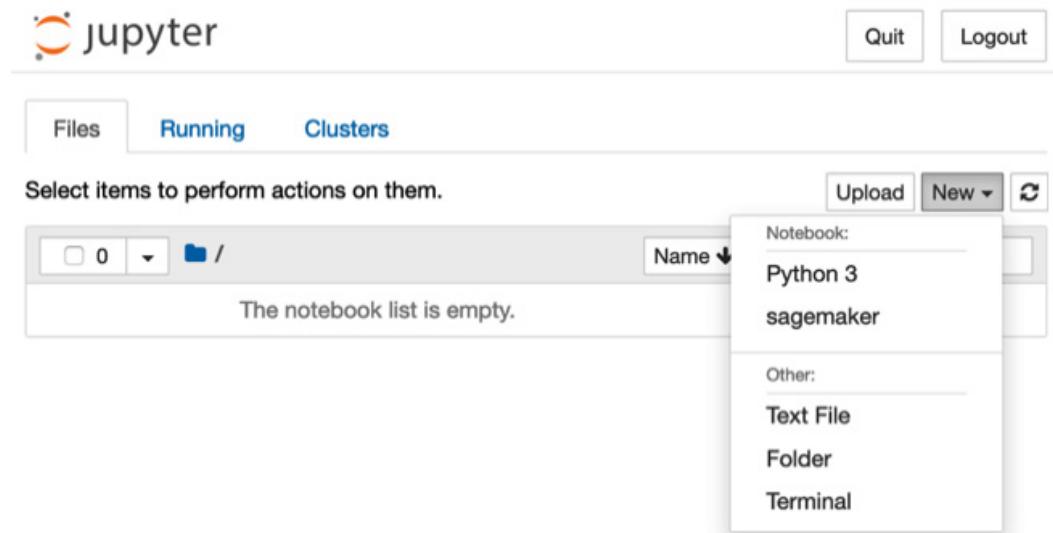
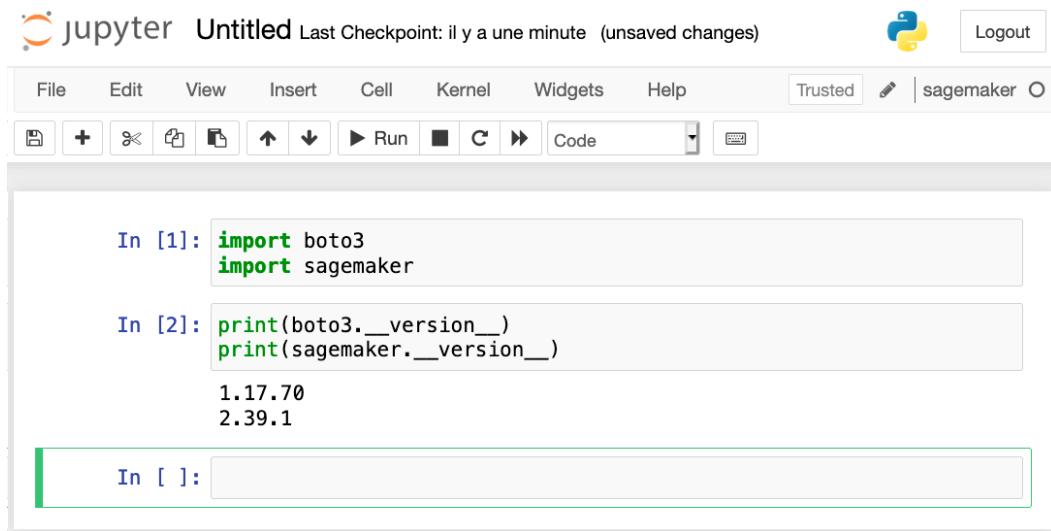


Figure 1.2 – Creating a new notebook

- Finally, we can check that the SDKs are available by importing them and printing their version, as shown in the following screenshot:



The screenshot shows a Jupyter Notebook interface. At the top, it says "jupyter Untitled Last Checkpoint: il y a une minute (unsaved changes)" with a Python logo icon and a "Logout" button. Below the header is a toolbar with icons for File, Edit, View, Insert, Cell, Kernel, Widgets, Help, Trusted, and sagemaker. The main area contains two code cells. The first cell (In [1]) contains the code "import boto3" and "import sagemaker". The second cell (In [2]) contains the code "print(boto3.__version__)" and "print(sagemaker.__version__)". The output of the second cell is "1.17.70" and "2.39.1". A new cell (In []) is currently selected for input.

Figure 1.3 – Checking the SDK version

This completes the installation with `virtualenv`. Don't forget to terminate Jupyter, and to deactivate your `virtualenv`:

```
$ deactivate
```

You can also install the SDK using Anaconda.

Installing the SageMaker SDK with Anaconda

Anaconda includes a package manager named `conda` that lets you create and manage isolated environments. If you've never worked with `conda` before, you should do the following:

- Install Anaconda: <https://docs.anaconda.com/anaconda/install/>.
- Read this tutorial: <https://docs.conda.io/projects/conda/en/latest/user-guide/getting-started.html#machine-learning>.

We will get started using the following steps:

- Let's create and activate a new `conda` environment named `conda-sagemaker`:

```
$ conda create -y -n conda-sagemaker
$ conda activate conda-sagemaker
```

- Then, we install pandas, boto3, and the SageMaker SDK. The latter has to be installed with pip as it's not available as a conda package:

```
$ conda install -y boto3 pandas
$ pip3 install sagemaker
```

- Now, let's add Jupyter and its dependencies to the environment, and create a new kernel:

```
$ conda install -y jupyter ipykernel
$ python3 -m ipykernel install --user --name conda-sagemaker
```

- Then, we can launch Jupyter:

```
$ jupyter notebook
```

Check that the conda-sagemaker kernel is present in the **New** menu, as is visible in the following screenshot:

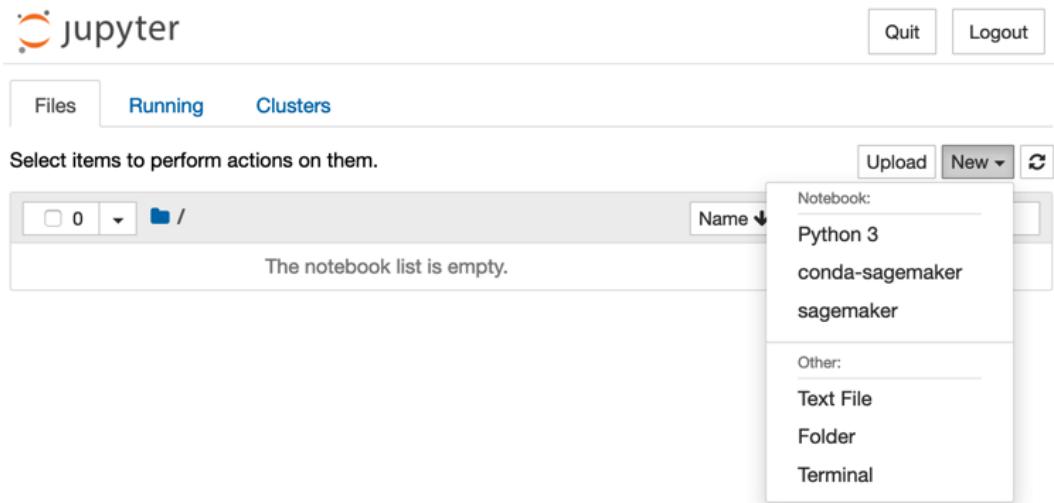


Figure 1.4 – Creating a new conda environment

- Just like in the previous section, we can create a notebook using this kernel and check that the SDKs are imported correctly.

This completes the installation with conda. Whether you'd rather use it instead of virtualenv is largely a matter of personal preference. You can definitely run all notebooks in this book and build your own projects with one or the other.

A word about AWS permissions

Amazon Identity and Access Management (IAM) enables you to manage access to AWS services and resources securely (<https://aws.amazon.com/iam>). Of course, this applies to Amazon SageMaker as well, and you need to make sure that your AWS user has sufficient permissions to invoke the SageMaker API.

IAM permissions

If you're not familiar with IAM at all, please read the following documentation:

https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html#machine_learning

You can run a quick test by using the AWS CLI on one of the SageMaker APIs, for example, `list-endpoints`. I'm using the `eu-west-1` region here, but feel free to use the region that is nearest to you:

```
$ aws sagemaker list-endpoints --region eu-west-1
{
    "Endpoints": []
}
```

If you get an error message complaining about insufficient permissions, you need to update the IAM role attached to your AWS user.

If you own the AWS account in question, you can easily do this yourself in the IAM console by adding the `AmazonSageMakerFullAccess` managed policy to your role. Note that this policy is extremely permissive: this is fine for a development account, but certainly not for a production account.

If you work with an account where you don't have administrative rights (such as a company-provided account), please contact your IT administrator to add SageMaker permissions to your AWS user.

For more information on SageMaker permissions, please refer to the documentation: https://docs.aws.amazon.com/sagemaker/latest/dg/security-iam.html#machine_learning.

Setting up Amazon SageMaker Studio

Experimentation is a key part of the Machine learning process. Developers and data scientists use a collection of open source tools and libraries for data exploration, data processing, and, of course, to evaluate candidate algorithms. Installing and maintaining these tools takes a fair amount of time, which would probably be better spent on studying the Machine learning problem itself!

Amazon SageMaker Studio brings you the machine learning tools you need from experimentation to production. At its core is an integrated development environment based on Jupyter that makes it instantly familiar.

In addition, SageMaker Studio is integrated with other SageMaker capabilities, such as SageMaker Experiments to track and compare all jobs, SageMaker Autopilot to automatically create machine learning models, and more. A lot of operations can be achieved in just a few clicks, without having to write any code.

SageMaker Studio also further simplifies infrastructure management. You won't have to create notebook instances: SageMaker Studio provides you with compute environments that are readily available to run your notebooks.

Note

This section requires basic knowledge of Amazon S3, Amazon VPC, and Amazon IAM. If you're not familiar with them at all, please read the following documentation:

<https://docs.aws.amazon.com/AmazonS3/latest/dev>Welcome.html#machine learning>

<https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html#machine learning>

<https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html#machine learning>

Now would also probably be a good time to take a look at (and bookmark) the SageMaker pricing page: <https://aws.amazon.com/sagemaker/pricing/>.

Onboarding to Amazon SageMaker Studio

You can access SageMaker Studio using any of these three options:

- **Use the quick start procedure:** This is the easiest option for individual accounts, and we'll walk through it in the following paragraphs.
- **Use AWS Single Sign-On (SSO):** If your company has an SSO application set up, this is probably the best option. You can learn more about SSO onboarding at <https://docs.aws.amazon.com/sagemaker/latest/dg/onboard-sso-users.html>. Please contact your IT administrator for details.
- **Use Amazon IAM:** If your company doesn't use SSO, this is probably the best option. You can learn more about SSO onboarding at <https://docs.aws.amazon.com/sagemaker/latest/dg/onboard-iam.html>. Again, please contact your IT administrator for details.

Onboarding with the quick start procedure

There are several steps to the quick start procedure:

1. First, open the AWS Console in one of the regions where Amazon SageMaker Studio is available, for example, <https://us-east-2.console.aws.amazon.com/sagemaker/>.
2. As shown in the following screenshot, the left-hand vertical panel has a link to **SageMaker Studio**:



Figure 1.5 – Opening SageMaker Studio

3. Clicking on this link opens the onboarding screen, and you can see its first section in the next screenshot:

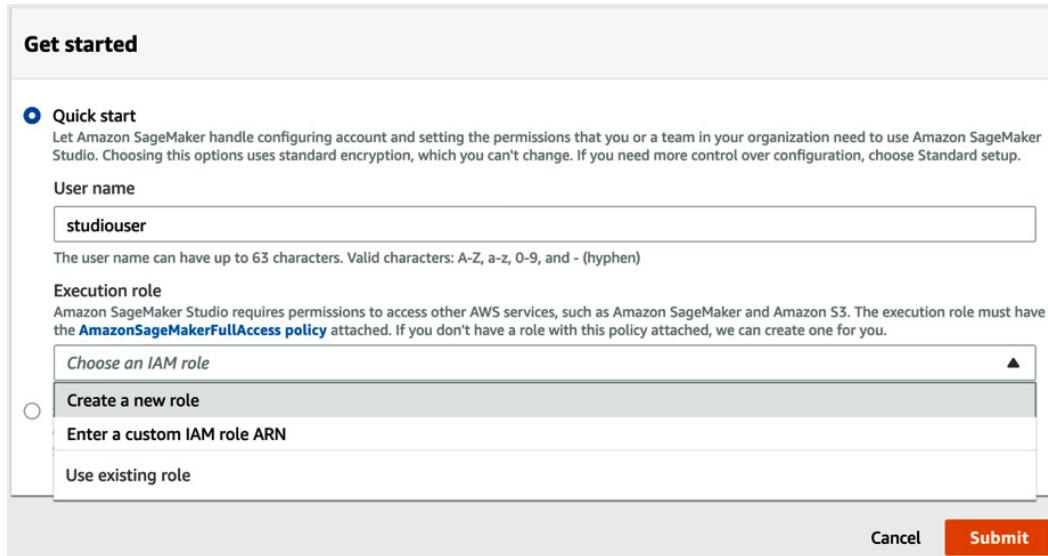


Figure 1.6 – Running Quick start

4. Let's select **Quick start**. Then, we enter the username we'd like to use to log in to SageMaker Studio, and we create a new IAM role as shown in the preceding screenshot. This opens the following screen:

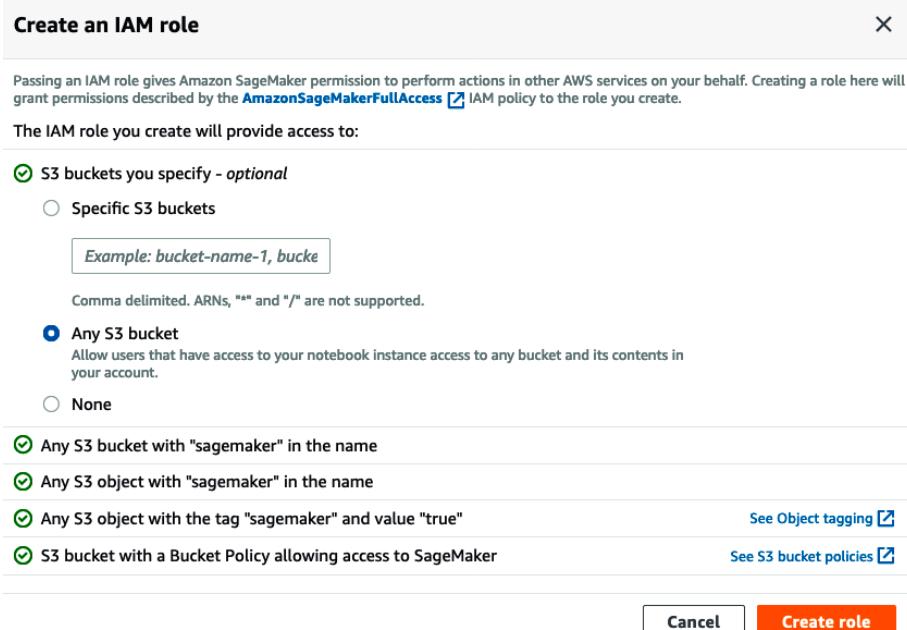


Figure 1.7 – Creating an IAM role

The only decision we have to make here is whether we want to allow our notebook instance to access specific Amazon S3 buckets. Let's select **Any S3 bucket** and click on **Create role**. This is the most flexible setting for development and testing, but we'd want to apply much stricter settings for production. Of course, we can edit this role later on in the IAM console, or create a new one.

5. Once we've clicked on **Create role**, we're back to the previous screen. Please make sure that project templates and JumpStart are enabled for this account. (this should be the default setting).
6. We just have to click on **Submit** to launch the onboarding procedure. Depending on your account setup, you may get an extra screen asking you to select a VPC and a subnet. I'd recommend selecting any subnet in your default VPC.
7. A few minutes later, SageMaker Studio is in service, as shown in the following screenshot. We could add extra users if we needed to, but for now, let's just click on **Open Studio**:

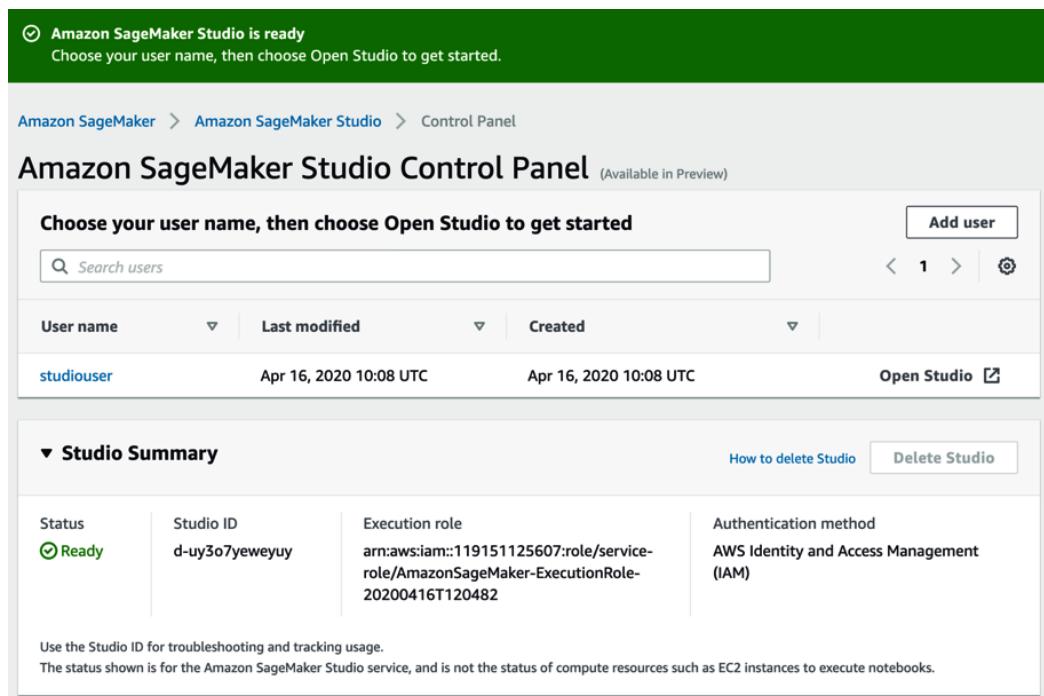


Figure 1.8 – Launching SageMaker Studio

Don't worry if this takes a few more minutes, as SageMaker Studio needs to complete the first-run setup of your environment. As shown in the following screenshot, once we open SageMaker Studio, we see the familiar JupyterLab layout:

Note

SageMaker Studio is a living thing. By the time you're reading this, some screens may have been updated. Also, you may notice small differences from one region to the next, as some features or instance types are not available there.

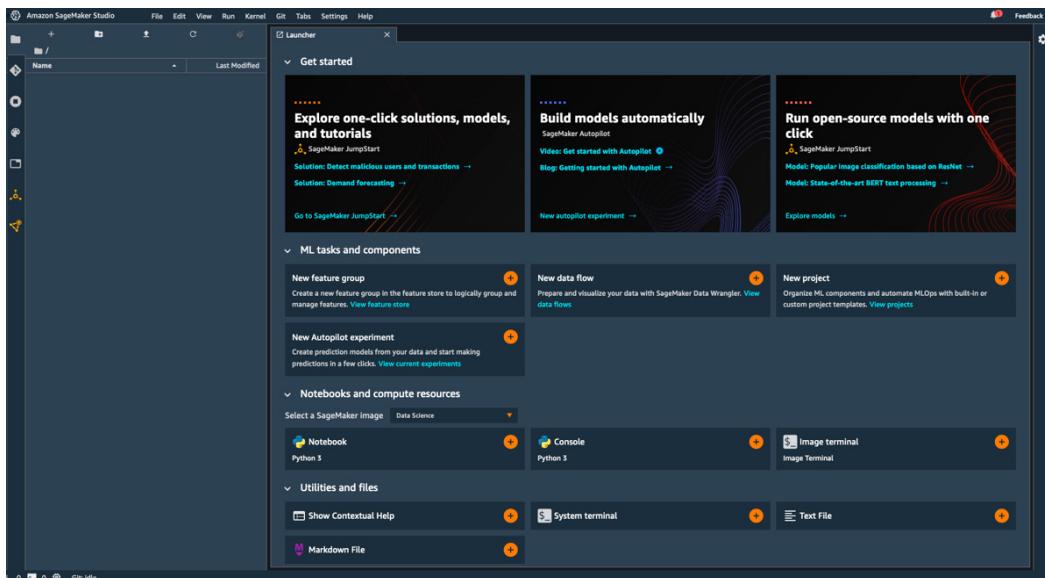


Figure 1.9 – SageMaker Studio welcome screen

8. We can immediately create our first notebook. In the **Launcher** tab, in the **Notebooks and compute resources** section, let's select **Data Science**, and click on **Notebook – Python 3**.

9. This opens a notebook, as is visible in the following screenshot. We first check that SDKs are readily available. As this is the first time we are launching the **Data Science** kernel, we need to wait for a couple of minutes.

```

Untitled.ipynb
[1]: import boto3
       import sagemaker

[2]: print(boto3.__version__)
       print(sagemaker.__version__)

1.17.58
2.38.0

```

Figure 1.10 – Checking the SDK version

10. As is visible in the following screenshot, we can easily list resources that are currently running in our Studio instance: an **machine learning.t3.medium** instance, the data science image supporting the kernel used in our notebook, and the notebook itself:

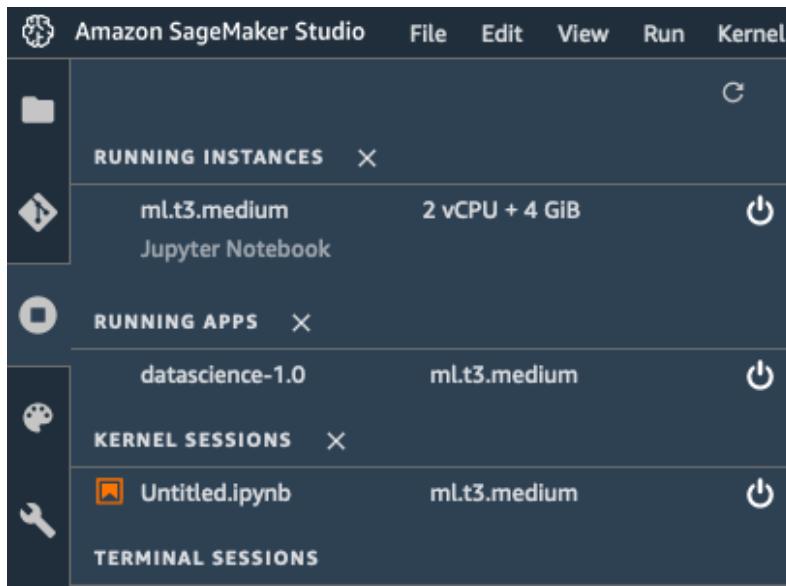


Figure 1.11 – Viewing Studio resources

11. To avoid unnecessary costs, we should shut these resources down when we're done working with them. For example, we can shut down the instance and all resources running on it, as you can see in the following screenshot. Don't do it now, we'll need the instance to run the next examples!

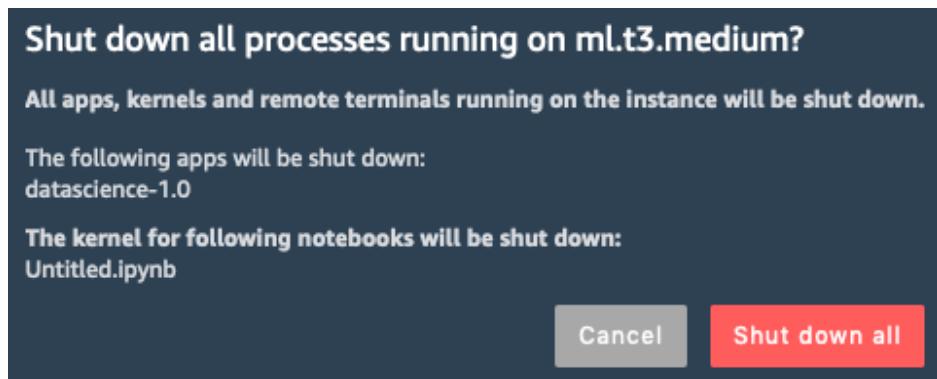


Figure 1.12 – Shutting down an instance

12. **Machine learning.t3.medium** is the default instance size that Studio uses. You can switch to other instance types by clicking on **2 vCPU + 4 GiB** at the top of your notebook. This lets you select a new instance size and launch it in Studio. After a few minutes, the instance is up and your notebook code has been migrated automatically. Don't forget to shut down the previous instance, as explained earlier.
13. When we're done working with SageMaker Studio, all we have to do is close the browser tab. If we want to resume working, we just have to go back to the SageMaker console and click on **Open Studio**.
14. If we wanted to shut down the Studio instance itself, we'd simply select **Shut Down** in the **File** menu. All files would still be preserved until we deleted Studio completely in the SageMaker console.

Now that we've completed the setup, I'm sure you're impatient to get started with machine learning. Let's start deploying some models!

Deploying one-click solutions and models with Amazon SageMaker JumpStart

If you're new to machine learning, you may find it difficult to get started with real-life projects. You've run all the toy examples, and you've read several blog posts on the state of the models for COMPUTER VISION OR NATURAL LANGUAGE PROCESSING. Now what? How can you start using these complex models on your own data to solve your own business problems?

Even if you're an experienced practitioner, building end-to-end machine learning solutions is not an easy task. Training and deploying models is just part of the equation: what about data preparation, automation, and so on?

Amazon SageMaker JumpStart was specifically built to help everyone get started more quickly with their machine learning projects. In literally one click, you can deploy the following:

- 16 end-to-end solutions for real-life business problems such as fraud detection in financial transactions, explaining credit decisions, predictive maintenance, and more
- Over 180 TensorFlow and PyTorch models pre-trained on a variety of computer vision and natural language processing tasks
- Additional learning resources, such as sample notebooks, blog posts, and video tutorials

Time to deploy a solution.

Deploying a solution

Let's begin:

1. Starting from the icon bar on the left, we open JumpStart. The following screenshot shows the opening screen:

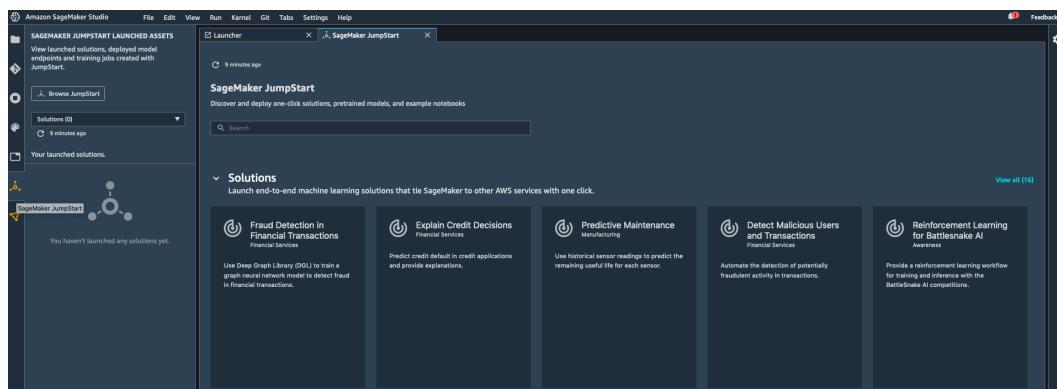


Figure 1.13 – Viewing solutions in JumpStart

2. Select **Fraud Detection in Financial Transactions**. As can be seen in the following screenshot, this is a fascinating example that uses graph data and graph neural networks to predict fraudulent activities based on interactions:

SOLUTION

Fraud Detection in Financial Transactions using Deep Graph Library

[Launch](#) [Browse JumpStart](#)

Launch Solution

Launch an end-to-end solution that will create supporting AWS infrastructure and tools to allow you to run specific ML workflows for different use cases.

Solution

Many online businesses lose billions annually to fraud, but machine learning based fraud detection models can help businesses predict what interactions or users are likely fraudulent and save them from incurring those costs.

In this project, we formulate the problem of fraud detection as a classification task on a heterogeneous interaction network. The machine learning model is a Graph Neural Network (GNN) that learns latent representations of users or transactions which can then be easily separated into fraud or legitimate.

This project shows how to use [Amazon SageMaker](#) and [Deep Graph Library \(DGL\)](#) to construct a heterogeneous graph from tabular data and train a GNN model to detect fraudulent transactions in the [IEEE-CS dataset](#).

See the [details page](#) to learn more about the techniques used, and the [online webinar](#) or [tutorial blog post](#) to see step by step explanations and instructions on how to use this solution.

Architecture

The project architecture deployed by the cloud formation template is shown here.

```

graph LR
    S3[S3 Bucket with data, models, results] --> DPLF[Data Preprocessing Lambda Function]
    DPLF --> ASPP[Amazon SageMaker Processing]
    ASPP --> ASN[Amazon SageMaker Notebook Manual orchestration]
    ASN --> S3
    
```

Figure 1.14 – Viewing solution details

3. Once we've read the solution details, all we have to do is click on the **Launch** button. This will run an AWS CloudFormation template in charge of building all the AWS resources required by the solution.

CloudFormation

If you're curious about CloudFormation, you may find this introduction useful:
Welcome.html

4. A few minutes later, the solution is ready, as can be seen in the following screenshot. We click on **Open Notebook** to open the first notebook.

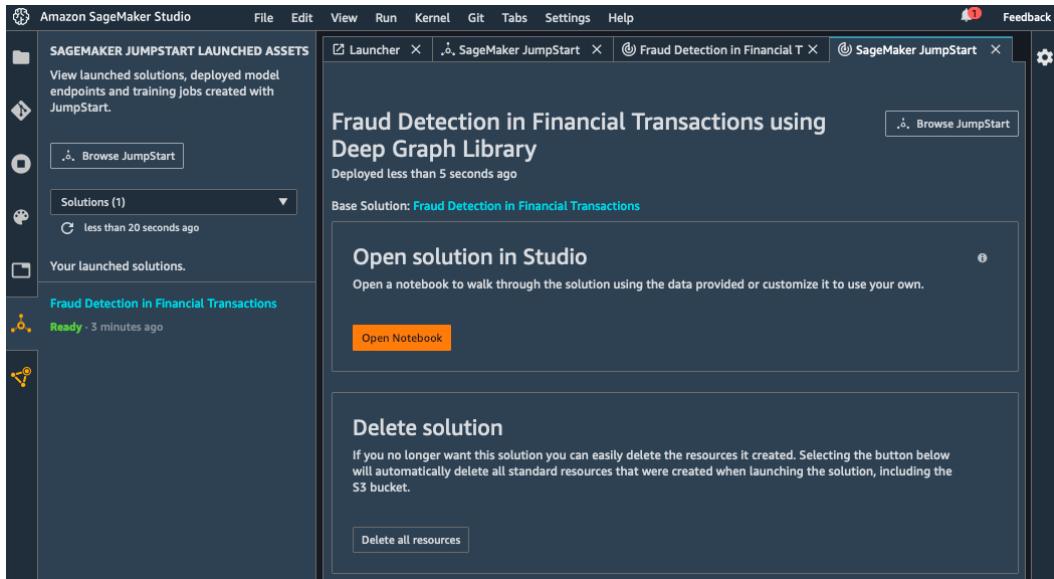


Figure 1.15 – Opening a solution

5. As you can see in the following screenshot, we can browse solution files in the left-hand pane: notebooks, training code, and so on:

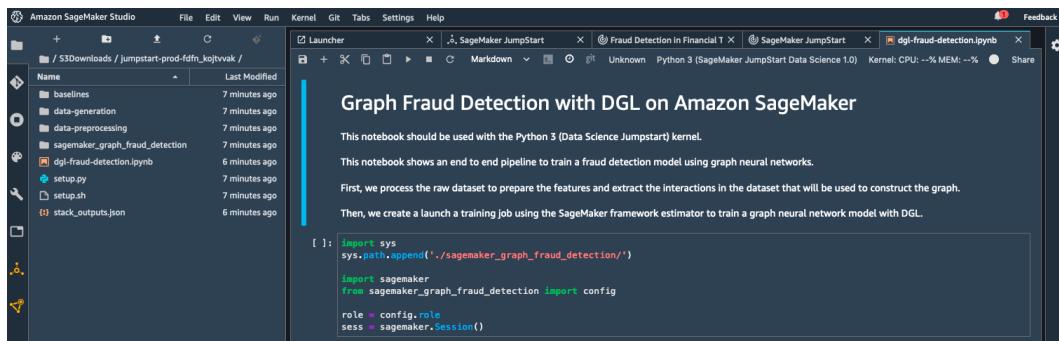


Figure 1.16 – Viewing solution files

6. From then on, you can start running and tweaking the notebook. If you're not familiar with the SageMaker SDK yet, don't worry about the details.
7. Once you're done, please go back to the solution page and click on **Delete all resources** to clean up and avoid unnecessary costs, as shown in the following screenshot:

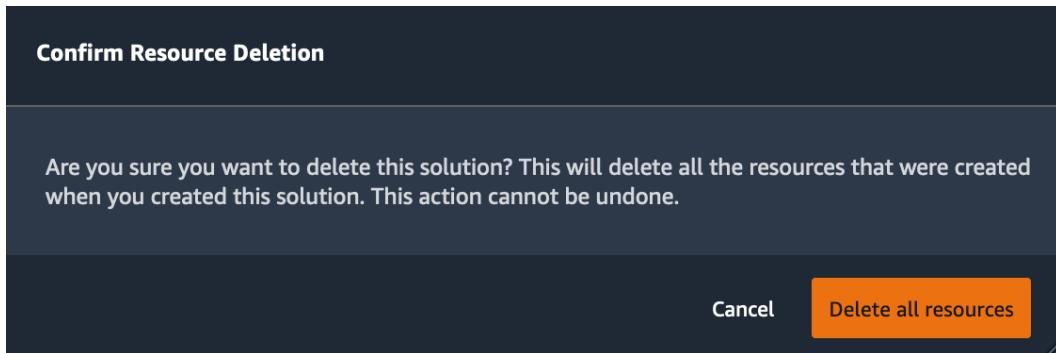


Figure 1.17 – Deleting a solution

As you can see, JumpStart solutions are a great way to explore how to solve business problems with machine learning and to start thinking about how you could do the same in your own business environment.

Now, let's see how we can deploy pre-trained models.

Deploying a model

JumpStart includes over 180 TensorFlow and PyTorch models pre-trained on a variety of computer vision and natural language processing tasks. Let's take a look at computer vision models:

1. Starting from the JumpStart main screen, we open **Vision models**, as can be seen in the following screenshot:

The screenshot shows the SageMaker JumpStart interface with the "Vision models" section expanded. It displays four pre-trained models:

- Inception V3**: Community Model - Vision. Task: Image Classification. Dataset: ImageNet. Fine-tunable: Yes. Source: TensorFlow Hub.
- ResNet 18**: Community Model - Vision. Task: Image Classification. Dataset: ImageNet. Fine-tunable: Yes. Source: PyTorch Hub.
- SSD EfficientDet D0**: Community Model - Vision. Task: Object Detection. Dataset: COCO 2017. Fine-tunable: No. Source: TensorFlow Hub.
- MobileNet V2**: Community Model - Vision. Task: Image Classification. Dataset: ImageNet. Fine-tunable: Yes. Source: TensorFlow Hub.

Figure 1.18 – Viewing computer vision models

2. Let's say that we're interested in trying out object detection models based on the **Single Shot Detector (SSD)** architecture. We click on the **SSD** model from the PyTorch Hub (the fourth one from the left).
3. This opens the model details page, telling us where the model comes from, what dataset it has been trained on, and which labels it can predict. We can also select which instance type to deploy the model. Sticking to the default, we click on **Deploy** to deploy the model on a real-time endpoint, as shown in the following screenshot:

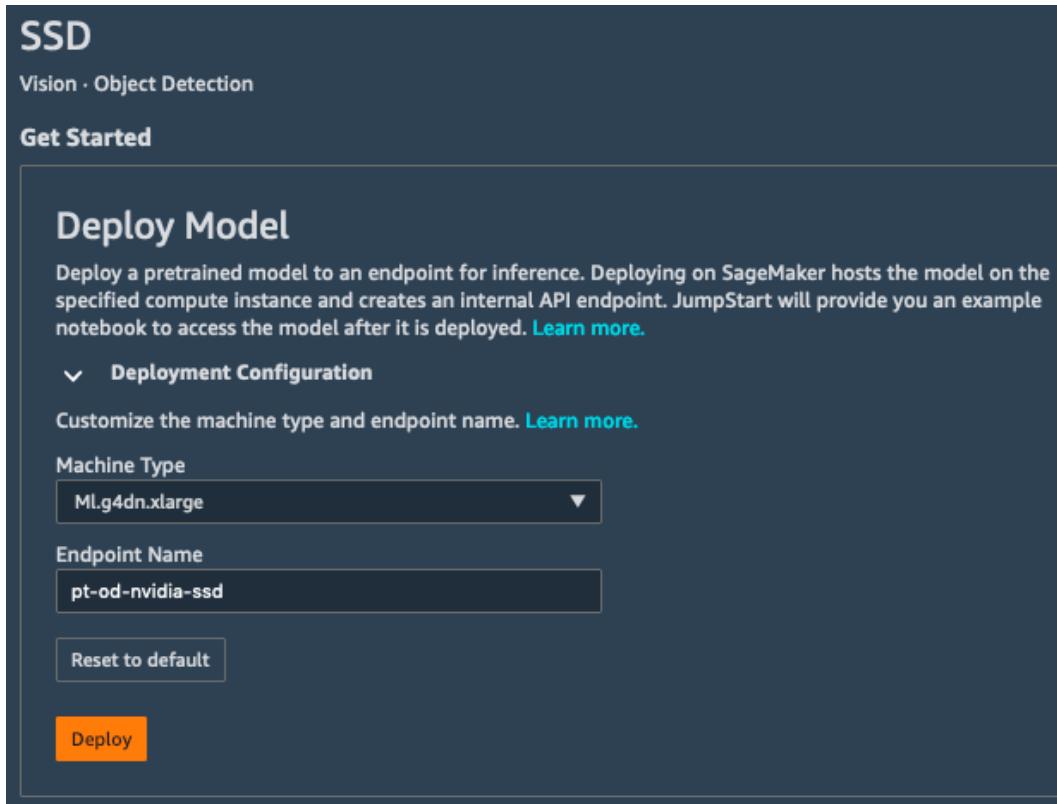


Figure 1.19 – Deploying a JumpStart model

4. A few minutes later, the model has been deployed. As can be seen in the following screenshot, we can see the endpoint status in the left-hand panel, and we simply click on **Open Notebook** to test it.

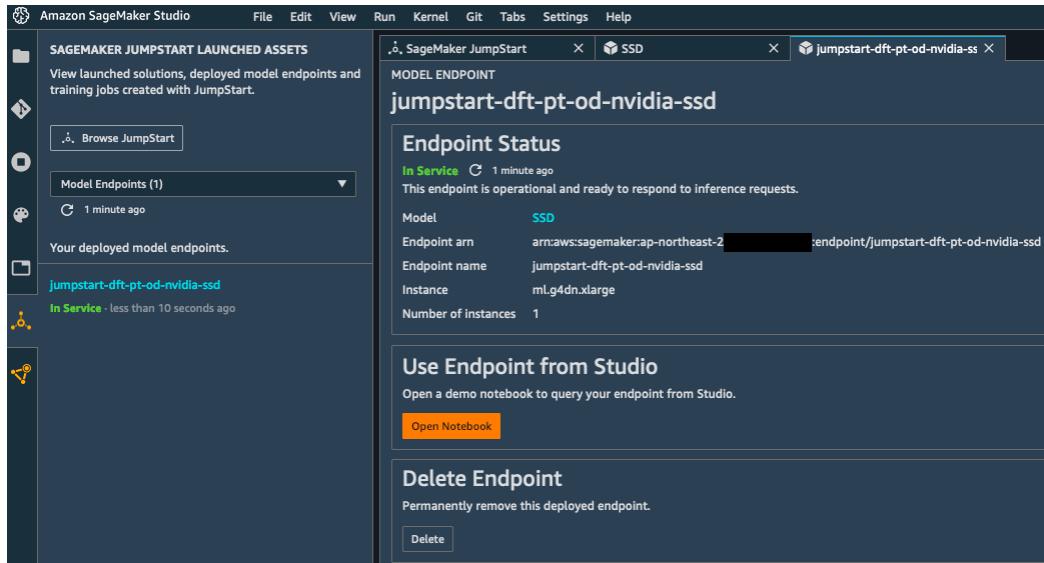


Figure 1.20 – Opening a JumpStart notebook

- Clicking through the notebook cells, we download a test image and we predict which objects it contains. Bounding boxes, classes, and probabilities are visible in the following screenshot:



Figure 1.21 – Detecting objects in a picture

6. When you're done, please make sure to delete the endpoint to avoid unnecessary charges: simply click on **Delete** in the endpoint details screen visible in *Figure 1.20*.

Not only does JumpStart make it extremely easy to experiment with state-of-the-art models, but it also provides you with code that you can readily use in your own projects: loading an image for prediction, predicting with an endpoint, plotting results, and so on.

As useful as pre-trained models are, we often need to fine-tune them on our own datasets. Let's see how we can do that with JumpStart.

Fine-tuning a model

Let's use an image classification model this time:

Note

A word of warning about fine-tuning text models: complex models such as BERT can take a very long time to fine-tune, sometimes several hours per epoch on a single GPU. In addition to the long waiting time, the cost won't be negligible, so I'd recommend avoiding these examples unless you have a real-life business project to work on.

1. We select the **Resnet 18** model (the second from the left in *Figure 1.18*).
2. On the model details page, we see that this model can be fine-tuned either on a default dataset available for testing (a TensorFlow dataset with five flower classes) or on our own dataset stored in S3. Scrolling down, we learn about the format that our dataset should have.
3. As visible in the following figure we stick to the default dataset. We also leave the deployment configuration and training parameters unchanged. Then, we click on **Train** to launch the fine-tuning job.

Fine-tune Model

Create a training job to fine-tune this pretrained model to fit your own data. Fine-tuning trains a pretrained model on a new dataset without training from scratch. It can produce accurate models with smaller datasets and less training time. [Learn more.](#)

▼ Data Source

Select the default dataset, or use your own data to fine-tune this model.

Default dataset Find S3 bucket Enter S3 bucket location

This option will fit the model to the default dataset. [Learn more.](#)

Default dataset:

> Deployment Configuration
 > Hyper-parameters

Train

Figure 1.22 – Fine-tuning a model

- After just a few minutes, fine-tuning is complete (which is why I picked this example!). We can see the output path in S3 where the fine-tuned model has been stored. Let's write down that path; we're going to need it in a minute.

smjs-d-pt-ic-resnet18-20210511-142657

Trained 34 minutes ago

Training Status

Complete 29 minutes ago

The training job that fine-tuned the pretrained model to create your own model is complete. From here you can see information about the model and deploy the model to an endpoint. You can also see this model in the AWS SageMaker console.

Parent model	ResNet 18
Training job name	smjs-d-pt-ic-resnet18-20210511-142657
Training job arn	arn:aws:sagemaker:ap-northeast-2:██████████:training-job/smjs-d-pt-ic-resnet18-20210511-142657
Training time	~5 minutes
Output path	s3://sagemaker-ap-northeast-2-██████████/smjs-d-pt-ic-resnet18-20210511-142657/output/model.tar.gz

> Instance Settings

Figure 1.23 – Viewing fine-tuning results

5. Then, we click on **Deploy** just like in the previous example. Once the model has been deployed, we open the sample notebook showing us how to predict with the initial pre-trained model.
6. This notebook uses images from the original dataset that the model was pre-trained on. No problem, let's adapt it! Even if we're not yet familiar with the SageMaker SDK, the notebook is simple enough that we can understand what's going on, and add a few cells to predict a flower image with our fine-tuned model.
7. First, we add a cell to copy the fine-tuned model artifact from S3, and we extract the list of classes and class indexes that JumpStart added:

```
%%sh
aws s3 cp s3://sagemaker-REGION_NAME-123456789012/smjs-d-
pt-ic-resnet18-20210511-142657/output/model.tar.gz .
tar xfz model.tar.gz
cat class_label_to_prediction_index.json
{"daisy": 0, "dandelion": 1, "roses": 2, "sunflowers": 3,
 "tulips": 4}
```

8. As expected, the fine-tuned model can predict five classes. Let's add a cell to download a sunflower image from Wikipedia:

```
%%sh
wget https://upload.wikimedia.org/wikipedia/commons/a/
a9/A_sunflower.jpg
```

9. Now, we load the image and invoke the endpoint:

```
import boto3
endpoint_name = 'jumpstart-ftd-pt-ic-resnet18'
client = boto3.client('runtime.sagemaker')
with open('A_sunflower.jpg', 'rb') as file:
    image = file.read()
response = client.invoke_endpoint(
    EndpointName=endpoint_name,
    ContentType='application/x-image',
    Body=image)
```

10. Finally, we print out the predictions. The highest probability is class #3 at 60.67%, confirming that our image contains a sunflower!

```
import json
model_predictions = json.loads(response['Body'].read())
print(model_predictions)
[0.30362239480018616, 0.06462913751602173,
0.007234351709485054, 0.6067869663238525,
0.017727158963680267]
```

11. When you're done testing, please make sure to delete the endpoint to avoid unnecessary charges.

This example illustrates how easy it is to fine-tune pre-trained models on your own datasets with SageMaker JumpStart and to use them to predict your own data. This is a great way to experiment with different models and find out which one could work best on the particular problem you're trying to solve.

This is the end of the first chapter, and it was already quite action-packed, wasn't it? It's now time to review what we've learned.

Summary

In this chapter, you discovered the main capabilities of Amazon SageMaker, and how they can help solve your machine learning pain points. By providing you with managed infrastructure and pre-installed tools, SageMaker lets you focus on the machine learning problem itself. Thus, you can go more quickly from experimenting with models to deploying them in production.

Then, you learned how to set up Amazon SageMaker on your local machine and in Amazon SageMaker Studio. The latter is a managed machine learning IDE where many other SageMaker capabilities are just a few clicks away.

Finally, you learned about Amazon SageMaker JumpStart, a collection of machine learning solutions and state-of-the-art models that you can deploy in one click, and start testing in minutes.

In the next chapter, we'll see how you can use Amazon SageMaker and other AWS services to prepare your datasets for training.

2

Handling Data Preparation Techniques

Data is the starting point of any machine learning project, and it takes lots of work to turn data into a dataset that can be used to train a model. That work typically involves annotating datasets, running bespoke scripts to preprocess them, and saving processed versions for later use. As you can guess, doing all this work manually, or building tools to automate it, is not an exciting prospect for machine learning teams.

In this chapter, you will learn about AWS services that help you build and process data. We'll first cover **Amazon SageMaker Ground Truth**, a capability of Amazon SageMaker that helps you quickly build accurate training datasets. Then, we'll introduce **Amazon SageMaker Data Wrangler**, a new way to transform your data interactively. Next, we'll talk about **Amazon SageMaker Processing**, another capability that helps you run your data processing workloads, such as feature engineering, data validation, model evaluation, and model interpretation. Finally, we'll quickly discuss other AWS services that may help with data analytics: **Amazon Elastic Map Reduce**, **AWS Glue**, and **Amazon Athena**.

This chapter consists of the following topics:

- Labeling data with Amazon SageMaker Ground Truth
- Transforming data with Amazon SageMaker Data Wrangler
- Running batch jobs with Amazon SageMaker Processing

Technical requirements

You will need an AWS account to run the examples included in this chapter. If you haven't got one already, please point your browser at <https://aws.amazon.com/getting-started/> to create one. You should also familiarize yourself with the AWS Free Tier, which lets you use many AWS services for free within certain usage limits.

You will need to install and to configure the AWS **Command Line Interface (CLI)** for your account (<https://aws.amazon.com/cli/>).

You will need a working Python 3.x environment. Installing the Anaconda distribution (<https://www.anaconda.com/>) is not mandatory, but strongly encouraged as it includes many projects that we will need (Jupyter, pandas, numpy, and more).

Code examples included in the book are available on GitHub at <https://github.com/PacktPublishing/Learn-Amazon-SageMaker-second-edition>. You will need to install a Git client to access them (<https://git-scm.com/>).

Labeling data with Amazon SageMaker Ground Truth

Added to Amazon SageMaker in late 2018, Amazon SageMaker Ground Truth helps you quickly build accurate training datasets. Machine learning practitioners can distribute labeling work to public and private workforces of human labelers. Labelers can be productive immediately, thanks to built-in workflows and graphical interfaces for common image, video, and text tasks. In addition, Ground Truth can enable automatic labeling, a technique that trains a machine learning model able to label data without additional human intervention.

In this section, you'll learn how to use Ground Truth to label images and text.

Using workforces

The first step in using Ground Truth is to create a workforce, a group of workers in charge of labeling data samples.

Let's head out to the SageMaker console: in the left-hand vertical menu, we click on **Ground Truth**, then on **Labeling workforces**. Three types of workforces are available: **Amazon Mechanical Turk**, **Vendor**, and **Private**. Let's discuss what they are, and when you should use them.

Amazon Mechanical Turk

Amazon Mechanical Turk (<https://www.mturk.com/>) makes it easy to break down large batch jobs into small work units that can be processed by a distributed workforce.

With Mechanical Turk, you can enroll tens or even hundreds of thousands of workers located across the globe. This is a great option when you need to label extremely large datasets. For example, think about a dataset for autonomous driving, made up of 1,000 hours of video: each frame would need to be processed in order to identify other vehicles, pedestrians, road signs, and more. If you wanted to annotate every single frame, you'd be looking at 1,000 hours x 3,600 seconds x 24 frames per second = **86.4 million images!** Clearly, you would have to scale out your labeling workforce to get the job done, and Mechanical Turk lets you do that.

Vendor workforce

As scalable as Mechanical Turk is, sometimes you need more control on who data is shared with, and on the quality of annotations, particularly if additional domain knowledge is required.

For this purpose, AWS has vetted a number of data labeling companies, which have integrated Ground Truth in their workflows. You can find the list of companies on **AWS Marketplace** (<https://aws.amazon.com/marketplace/>), under **Machine Learning | Data Labeling Services | Amazon SageMaker Ground Truth Services**.

Private workforce

Sometimes, data can't be processed by third parties. Maybe it's just too sensitive, or maybe it requires expert knowledge that only your company's employees have. In this case, you can create a private workforce made up of well-identified individuals that will access and label your data.

Creating a private workforce

Creating a private workforce is the quickest and simplest option. Let's see how it's done:

1. Starting from the **Labeling workforces** entry in the SageMaker console, we select the **Private** tab, as seen in the following screenshot. Then, we click on **Create private team**:

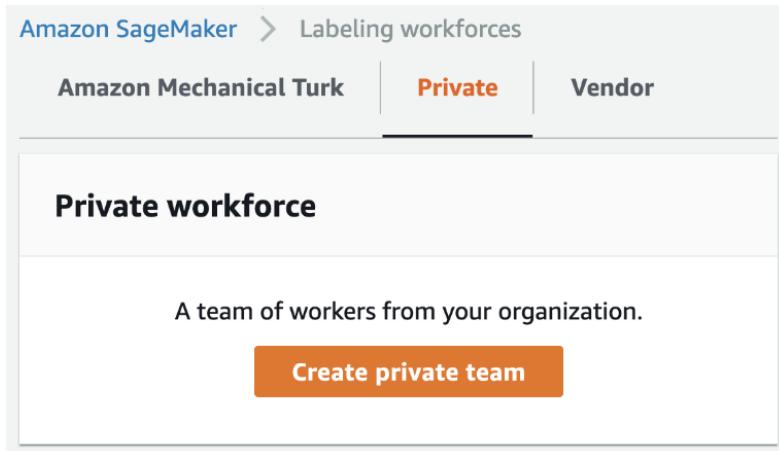


Figure 2.1 – Creating a private workforce

2. We give the team a name, then we have to decide whether we're going to invite workers by email, or whether we're going to import users that belong to an existing **Amazon Cognito** group.

Amazon Cognito (<https://aws.amazon.com/cognito/>) is a managed service that lets you build and manage user directories at any scale. Cognito supports both social identity providers (Google, Facebook, and Amazon), and enterprise identity providers (Microsoft Active Directory, SAML).

This makes a lot of sense in an enterprise context, but let's keep things simple and use email instead. Here, I will use some sample email addresses: please make sure to use your own, otherwise you won't be able to join the team!

3. Then, we need to enter an organization name, and more importantly a contact email that workers can use for questions and feedback on the labeling job. These conversations are extremely important in order to fine-tune labeling instructions, pinpoint problematic data samples, and more.
4. Optionally, we can set up notifications with **Amazon Simple Notification Service** (<https://aws.amazon.com/sns/>) to let workers know that they have work to do.

5. The screen should look like in the following screenshot. Then, we click on **Create private team**:

Team details

Team name
Give your work team a descriptive name. This name can't be changed later.
 Maximum of 63 alphanumeric characters. Can include hyphens, but not spaces. Must be unique within your account in an AWS Region.

Add workers Info
Add workers to your private work team by adding worker email addresses or importing workers from existing Amazon Cognito user groups.

Invite new workers by email Import workers from existing Amazon Cognito user groups

Email addresses Preview invitation

We send an invitation with instructions to each of the worker email addresses that you add here.

Use a comma between addresses. You can add up to 50 workers.

Organization name
We use this information to customize the email that we send to the workers.

Contact email
Workers use this address to report issues related to the task.

Enable Notifications

SNS topic - optional
Configuring SNS topic enables your work team to receive notifications on available work. [Learn more](#) ▾

[Cancel](#) **Create private team**

Figure 2.2 – Setting up a private workforce

6. A few seconds later, the team has been set up. Invitations have been sent to workers, requesting that they join the workforce by logging in to a specific URL. The invitation email looks like that shown in the following screenshot:

You're invited by My Awesome Private Team to work on a labeling project.

✉ no-reply@verificationemail.com
À [REDACTED]

You're invited to work on a labeling project.

You will need this user name and temporary password to log in the first time.

User name: [REDACTED]

Temporary password: [REDACTED]

Open the link below to log in:

<https://8rj5pzrz3j.labeling.eu-west-1.sagemaker.aws>

After you log in with your temporary password, you are required to create a new one. If you have any questions, please contact [REDACTED]

Figure 2.3 – Email invitation

7. Clicking on the link opens a login window. Once we've logged in and defined a new password, we're taken to a new screen showing available jobs, as in the following screenshot. As we haven't defined one yet, it's obviously empty:

The screenshot shows a web-based worker console interface. At the top, there is a header bar with the text "Hello, julien@[REDACTED]" on the left and a "Log out" button on the right. Below the header is a message box containing an information icon, the text "You're finished with the available tasks.", and a sub-instruction: "Refresh the page to see if there are new jobs. Select a job and choose Start working to work on new tasks." Further down, there is a section titled "Jobs" with a "Show instructions" button. A table below lists columns for "Description", "Customer ID", and "Creation time". The table currently has no data rows.

Figure 2.4 – Worker console

Let's keep our workers busy and create an image labeling job.

Uploading data for labeling

As you would expect, Amazon SageMaker Ground Truth uses Amazon S3 to store datasets:

1. Using the AWS CLI, we create an S3 bucket hosted in the same region we're running SageMaker in. Bucket names are globally unique, so please make sure to pick your own unique name when you create the bucket. Use the following code (feel free to use another AWS Region):

```
$ aws s3 mb s3://sagemaker-book --region eu-west-1
```

2. Then, we copy the cat images located in the `chapter2` folder of our GitHub repository as follows:

```
$ aws s3 cp --recursive cat/ s3://sagemaker-book/
chapter2/cat/
```

Now that we have some data waiting to be labeled, let's create a labeling job.

Creating a labeling job

As you would expect, we need to define the location of the data, what type of task we want to label it for, and what our instructions are:

1. In the left-hand vertical menu of the SageMaker console, we click on **Ground Truth**, then on **Labeling jobs**, then on the **Create labeling job** button.
2. First, we give the job a name, say '`my-cat-job`'. Then, we define the location of the data in S3. Ground Truth expects a **manifest file**: a manifest file is a **JSON** file that lets you filter which objects need to be labeled, and which ones should be left out. Once the job is complete, a new file, called the augmented manifest, will contain labeling information, and we'll be able to use this to feed data to training jobs.

3. Then, we define the location and the type of our input data, just like in the following screenshot:

Input data setup [Info](#)
Ground Truth needs a way to identify your input data. Use the automated setup to have Ground Truth automatically identify your dataset in S3. Use the manual setup if you have an input manifest file.

Automated data setup
Provide the S3 location of the dataset you want labeled and let Ground Truth automatically connect to and use this dataset for your job.

Manual data setup
Provide the S3 location of a file (an input manifest file) that identifies the data objects you want labeled

S3 location for input datasets [Info](#)
This is the location in S3 where your dataset objects are stored. Ground Truth will use all data objects in this location for your labeling job.

s3://sagemaker-book/chapter2/cat/

S3 location for output datasets [Info](#)
This is the location in S3 where your labeling job output data is stored.

Same location as input dataset

Specify a new location

Data type

Image

Supported formats are jpg, jpeg, and png

Figure 2.5 – Configuring input data

4. As is visible in the next screenshot, we select the IAM role that we created for SageMaker in the first chapter (your name will be different), and we then click on the **Complete data setup** button to validate this section:

IAM Role [Info](#)
Provide the ID or ARN for your own AWS KMS encryption key for Amazon SageMaker to access your S3 bucket. Choose a role or let us create a role with the [AmazonSageMakerFullAccess](#) IAM policy attached.

AmazonSageMaker-ExecutionRole-20200501T145026

Use this button to process and complete your input data setup.

Input data connection successful [View more details](#)

Figure 2.6 – Validating input data

Clicking on **View more details**, you can learn about what is happening under the hood. SageMaker Ground Truth crawls your data in S3 and creates a JSON file called the **manifest file**. You can go and download it from S3 if you're curious. This file points at your objects in S3 (images, text files, and so on).

5. Optionally, we could decide to work either with the full manifest, a random sample, or a filtered subset based on a **SQL** query. We could also provide an **Amazon KMS** key to encrypt the output of the job. Let's stick to the defaults here.
6. The **Task type** section asks us what kind of job we'd like to run. Please take a minute to explore the different task categories that are available (text, image, video, point cloud, and custom). As shown in the next screenshot, let's select the **Image** task category and the **Semantic segmentation** task, and then click **Next**:

Task type [Info](#)

Task category
Select the type of data being labeled to view available task templates for it or select 'Custom' to create your own.

Image ▾

Task selection
Select the task that a human worker will perform to label objects in your dataset.

Image Classification (Single Label)
Get workers to categorize images into individual classes. [Info](#)

Basketball
 Soccer



Image Classification (Multi-label)
Get workers to categorize images into one or more classes. [Info](#)

Human
 Vehicle
 Animal



Bounding box
Get workers to draw bounding boxes around specified objects in your images. [Info](#)



Semantic segmentation
Get workers to draw pixel level labels around specific objects and segments in your images. [Info](#)



Figure 2.7 – Selecting a task type

7. On the next screen, visible in the following screenshot, we first select our private team of workers:

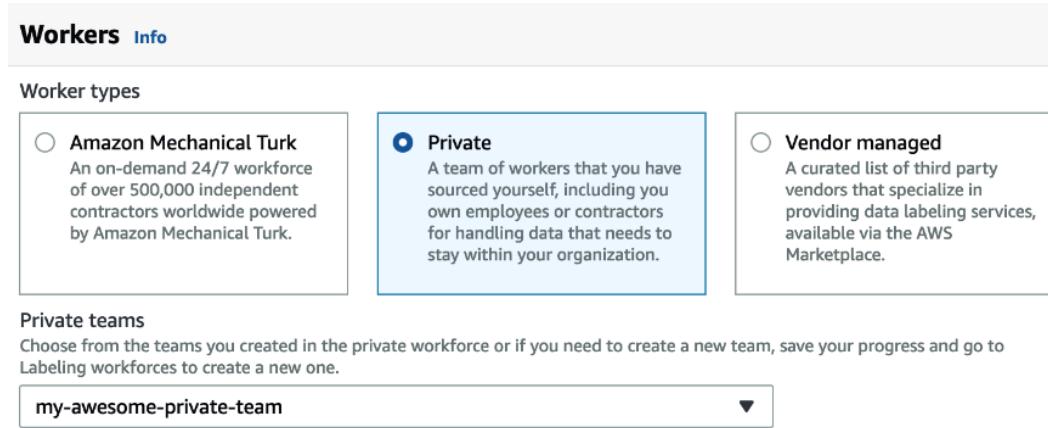


Figure 2.8 – Selecting a team type

8. If we had a lot of samples (say, tens of thousands or more), we should consider enabling **automated data labeling**, as this feature would reduce both the duration and the cost of the labeling job. Indeed, as workers would start labeling data samples, SageMaker Ground Truth would train a machine learning model on these samples. It would use them as a dataset for a supervised learning problem. With enough worker-labeled data, this model would pretty quickly be able to match and exceed human accuracy, at which point it would replace workers and label the rest of the dataset. If you'd like to know more about this feature, please read the documentation at <https://docs.aws.amazon.com/sagemaker/latest/dg/sms-automated-labeling.html>.
9. The last step in configuring our training job is to enter instructions for the workers. This is an important step, especially if your job is distributed to third-party workers. The better our instructions, the higher the quality of the annotations. Here, let's explain what the job is about, and enter a "cat" label for workers to apply. In a real-life scenario, you should add detailed instructions, provide sample images for good and bad examples, explain what your expectations are, and so on. The following screenshot shows what our instructions look like:

Semantic segmentation labeling tool

Provide labeling instructions with examples below for workers. Workers will be viewing these instructions when they perform your task. You can add up to 10 labels for workers to choose from. See guidelines for [creating high-quality instructions](#)

H₁ H₂ B I A

Good example
Enter description to explain a correctly done segmentation
 Add image here

Bad example
Enter description of an incorrectly done segmentation
 Add image here

The purpose of this job is to segment cats in the picture. If multiple cats are visible, please segment each cat individually. Only live cats should be segmented: ignore toy cats, cat pictures, etc.

Labels
Add up to 10 labels

Add label



▶ Additional instructions - optional

Cancel Previous Create

Figure 2.9 – Setting up instructions

- Once we're done with instructions, we click on **Create** to launch the labeling job. After a few minutes, the job is ready to be distributed to workers.

Labeling images

Logging in to the worker URL, we can see from the screen shown in the following screenshot that we have work to do:

The screenshot shows a web-based worker console. At the top, there is a header with "Hello, [REDACTED]" on the left and a "Log out" button on the right. Below the header is a "Show instructions" button with a toggle switch. The main area is titled "Jobs" and contains a table with three columns: "Description", "Customer ID", and "Creation time". A single row is highlighted in blue, representing a semantic segmentation task for cats. The "Description" column for this row contains the following text:

Semantic segmentation: The purpose of this job is to segment cats in the picture. If multiple cats are visible, please segment e

The "Customer ID" column shows the value 119151125607, and the "Creation time" column shows April 21, 2020 at 13:35:48 UTC.

Figure 2.10 – Worker console

We will use the following steps:

1. Clicking on **Start working** opens a new window, visible in the next picture. It displays instructions as well as a first image to work on:

Hello, [REDACTED]

Custom Task description: Draw pixel Task time: 0:43 of 60 Min Stop working Log out

Instructions	Labels
View full instructions View tool guide How to use the Auto-segment tool Good example Enter description to explain a correctly done segmentation  Add image here	Labels  cat 1
Bad example Enter description of an incorrectly done segmentation  Add image here	

The purpose of this job is to segment cats in the picture. If multiple cats are visible, please segment each cat individually. Only live cats should be segmented: ignore toy cats, cat pictures, etc.



Auto-segment Polygon Brush Eraser Dimmer Undo Redo Zoom in Zoom out

Nothing to label **Submit**

Treat the data in this task as confidential.

Figure 2.11 – Labeling images

- Using the graphical tools in the toolbar, and especially the auto-segment tool, we can very quickly produce high-quality annotations. Please take a few minutes to practice, and you'll be able to do the same in no time.

3. Once we're done with the three images, the job is complete, and we can visualize the labeled images under **Labeling jobs** in the SageMaker console. Your screen should look like the following screenshot:

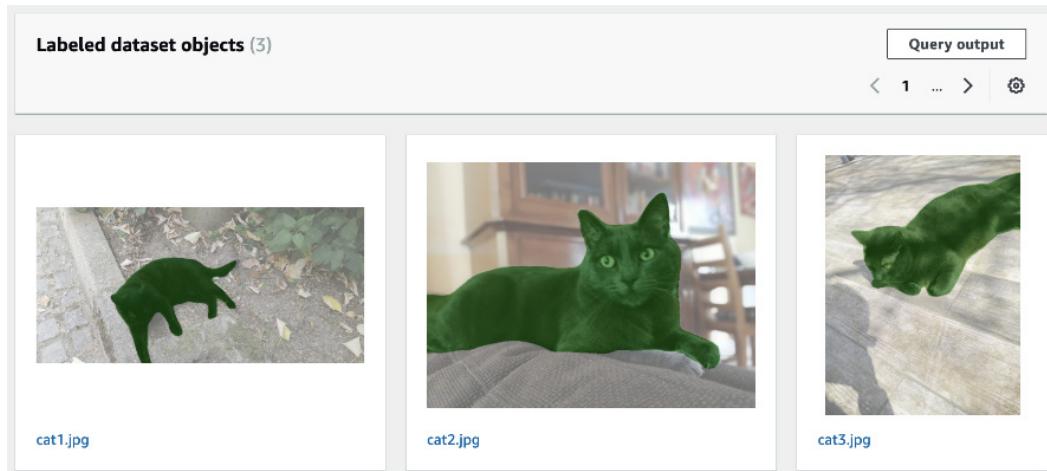


Figure 2.12 – Labeled images

More importantly, we can find labeling information in the S3 output location.

In particular, the **augmented manifest** (`output/my-cat-job/manifests/output/output.manifest`) contains annotation information on each data sample, such as the classes present in the image, and a link to the segmentation mask.

In *Chapter 5, Training Computer Vision Models*, we'll see how we can feed this information directly to the built-in computer vision algorithms implemented in Amazon SageMaker. Of course, we could also parse this information, and convert it for whatever framework we use to train our computer vision model.

As you can see, SageMaker Ground Truth makes it easy to label image datasets. You just need to upload your data to S3 and create a workforce. Ground Truth will then distribute the work automatically, and store the results in S3.

We just saw how to label images, but what about text tasks? Well, they're equally easy to set up and run. Let's go through an example.

Labeling text

This is a quick example of labeling text for named entity recognition. The dataset is made up of text fragments from one of my blog posts, where we'd like to label all AWS service names. These are available in our GitHub repository.

We will start labeling text using the following steps:

- First, let's upload text fragments to S3 with the following line of code:

```
$ aws s3 cp --recursive ner/ s3://sagemaker-book/
chapter2/ner/
```

- Just like in the previous example, we configure a text labeling job, set up input data, and select an IAM role, as shown in the following screenshot:

The screenshot shows the 'Job overview' configuration page for creating a text labeling job. The job name is 'my-ner-job'. Under 'Input data setup', 'Automated data setup' is selected, indicating the S3 location of the dataset. The 'S3 location for input datasets' field contains 's3://sagemaker-book/chapter2/ner/dataset-2020072'. Under 'S3 location for output datasets', 'Same location as input dataset' is selected. The 'Data type' is set to 'Text'. The 'IAM Role' is 'AmazonSageMaker-ExecutionRole-20200501T145026'. A success message at the bottom indicates 'Input data connection successful'.

Job overview

Job name
my-ner-job

Maximum of 63 alphanumeric characters. Can include hyphens (-), but not spaces. Must be unique within your account in an AWS Region.

I want to specify a label attribute name different from the labeling job name.

Label attribute name is the key where your labels are stored in the augmented manifest. Ground Truth uses the labeling job name as the default label attribute name.

Input data setup Info
Ground Truth needs a way to identify your input data. Use the automated setup to have Ground Truth automatically identify your dataset in S3. Use the manual setup if you have an input manifest file.

Automated data setup
Provide the S3 location of the dataset you want labeled and let Ground Truth automatically connect to and use this dataset for your job.

Manual data setup
Provide the S3 location of a file (an input manifest file) that identifies the data objects you want labeled

S3 location for input datasets Info
This is the location in S3 where your dataset objects are stored. Ground Truth will use all data objects in this location for your labeling job.

s3://sagemaker-book/chapter2/ner/dataset-2020072

S3 location for output datasets Info
This is the location in S3 where your labeling job output data is stored.

Same location as input dataset

Specify a new location

Data type

Text
Supported formats are txt and csv

IAM Role Info
Provide the ID or ARN for your own AWS KMS encryption key for Amazon SageMaker to access your S3 bucket. Choose a role or let us create a role with the [AmazonSageMakerFullAccess](#) IAM policy attached.

AmazonSageMaker-ExecutionRole-20200501T145026

Use this button to process and complete your input data setup.

Complete data setup

Input data connection successful [View more details](#)

Figure 2.13 – Creating a text labeling job

3. Then, we select **Text** as the category, and **Named entity recognition** as the task.
4. On the next screen, shown in the following screenshot, we simply select our private team again, add a label, and enter instructions:

The screenshot shows the 'Named entity recognition labeling tool' interface. At the top right is a 'Preview' button. Below it, a note says: 'Provide labeling instructions with examples below for workers. Workers will be viewing these instructions when they perform your task. You can add up to 30 labels for workers to choose from. See guidelines for [creating high-quality instructions](#)'.

On the left, there are styling tools (H1, H2, B, I, A, etc.) and sections for 'Enter description of the labels that workers have to choose from' and 'Add examples to help workers understand the label'. The main area contains a text example: 'Since 2006, Amazon Web Services has been striving to simplify IT infrastructure. Thanks to services like Amazon Elastic Compute Cloud (EC2), Amazon Simple Storage Service (S3), Amazon Relational Database Service (RDS), AWS CloudFormation and many more, millions of customers can build reliable, scalable, and secure platforms in any AWS region in minutes. Having spent 10 years procuring, deploying and managing more hardware than I care to remember, I'm still amazed every day by the pace of innovation that builders achieve with our services.' To the right, under 'Labels', there is a table where 'aws_servic' is listed with a short name 'Per' and a note 'Max 3 characters.'. An 'Add label' button is also present.

At the bottom, there is an optional section for 'Additional instructions - optional' and a footer with 'Cancel', 'Previous', and 'Create' buttons.

Figure 2.14 – Setting up instructions

5. Once the job is ready, we log in to the worker console and start labeling. You can see a labeled example in the following screenshot:

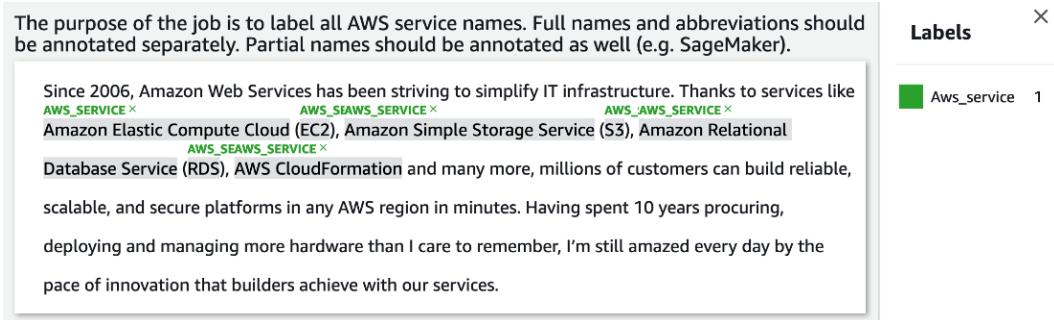


Figure 2.15 – Labeling text

6. We're done quickly, and we can find the labeling information in our S3 bucket. For each sample, we see a start offset, an end offset, and a label for each labeled entity.

Amazon SageMaker Ground Truth really makes it easy to label datasets at scale. It has many nice features including job chaining and custom workflows, which I encourage you to explore at <https://docs.aws.amazon.com/sagemaker/latest/dg/sms.html>.

Now that we know how to label datasets, let's see how we can easily transform data interactively with Amazon SageMaker Data Wrangler.

Transforming data with Amazon SageMaker Data Wrangler

Collecting and labeling data samples is only the first step in preparing a dataset. Indeed, it's very likely that you'll have to pre-process your dataset in order to do the following, for example:

- Convert it to the input format expected by the machine learning algorithm you're using.
- Rescale or normalize numerical features.
- Engineer higher-level features, for example, one-hot encoding.
- Clean and tokenize text for natural language processing applications

In the early stage of a machine learning project, it's not always obvious which transformations are required, or which ones are most efficient. Thus, practitioners often need to experiment with lots of different combinations, transforming data in many different ways, training models, and evaluating results.

In this section, we're going to learn about **Amazon SageMaker Data Wrangler**, a graphical interface integrated in SageMaker Studio that makes it very easy to transform data, and to export results to a variety of Jupyter notebooks.

Loading a dataset in SageMaker Data Wrangler

First, we need a dataset. We'll use the direct marketing dataset published by S. Moro, P. Cortez, and P. Rita in "A Data-Driven Approach to Predict the Success of Bank Telemarketing", *Decision Support Systems*, Elsevier, 62:22-31, June 2014.

This dataset describes a binary classification problem: will a customer accept a marketing offer, yes or no? It contains a little more than 41,000 customer samples, and labels are stored in the `y` column.

We will get started using the following steps:

1. Using the AWS command line, let's download the dataset, extract it, and copy it to the default SageMaker bucket for the region we're running in (it should have been created automatically). You can run this on your local machine or in a Jupyter terminal:

Note

In this example, I'm running SageMaker in the ap-northeast-2 region (Seoul). Replace accordingly.

```
$ aws s3 cp s3://sagemaker-sample-data-us-west-2.s3-us-west-2.amazonaws.com/autopilot/direct_marketing/bank-additional.zip .
$ unzip bank-additional.zip
$ aws s3 cp bank-additional/bank-additional-full.csv s3://sagemaker-ap-northeast-2-123456789012/direct-marketing/
```

2. In SageMaker Studio, we create a new Data Wrangler flow with **File | New | Data Wrangler Flow** to create. The following screenshot shows the Data Wrangler image being loaded:

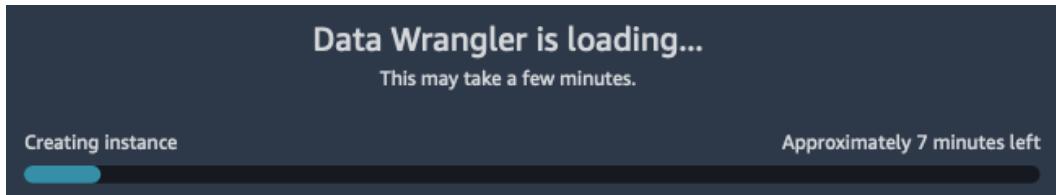


Figure 2.16 – Loading Data Wrangler

3. Once Data Wrangler is ready, the **Import** screen opens. We also see the Data Wrangler image in the left-hand pane, as shown in the next screenshot:

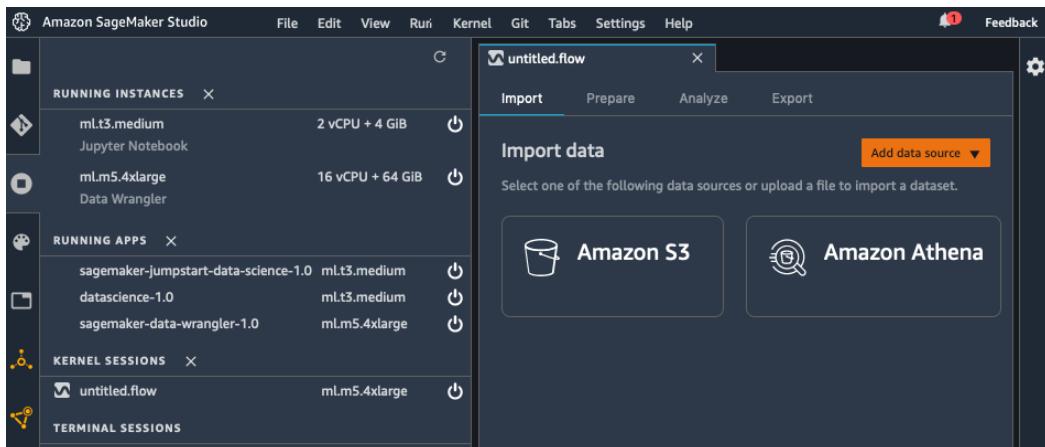


Figure 2.17 – Opening Data Wrangler

4. We can import data from S3, Athena or Redshift (by clicking on **Add data source**). Here, we click on S3.

5. As shown in the following screenshot, we can easily locate the dataset that we just uploaded. Let's click on it.

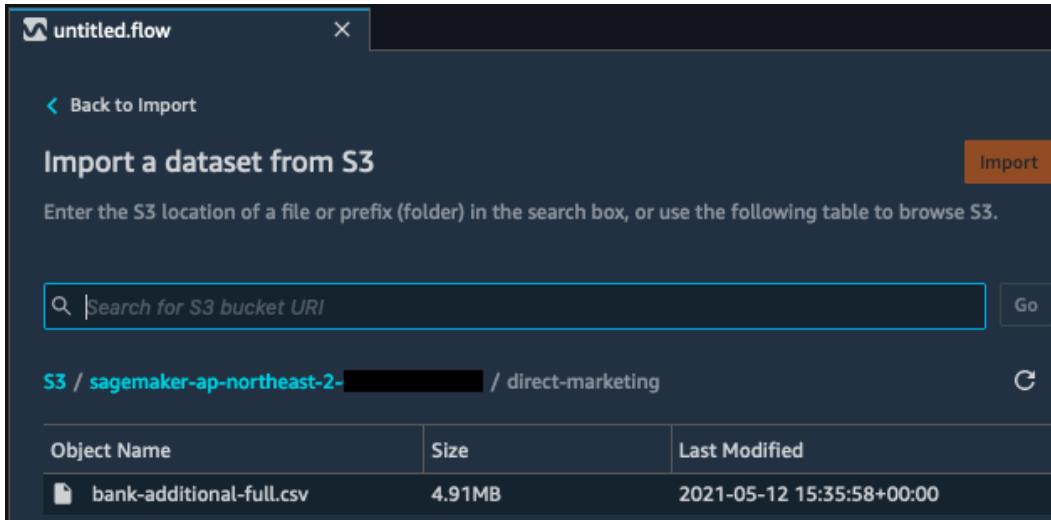


Figure 2.18 – Locating a dataset

6. This opens a preview of the dataset, as shown in the next screenshot:

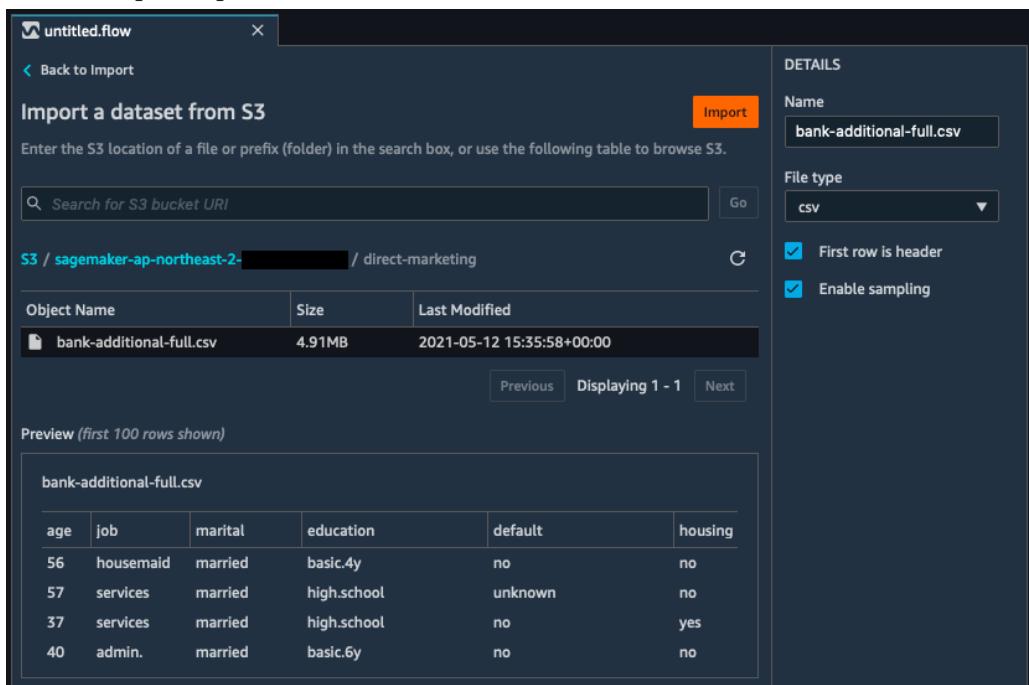


Figure 2.19 – Previewing a dataset

7. Let's just click on **Import**, which opens the **Prepare** view, as shown in the next screenshot:

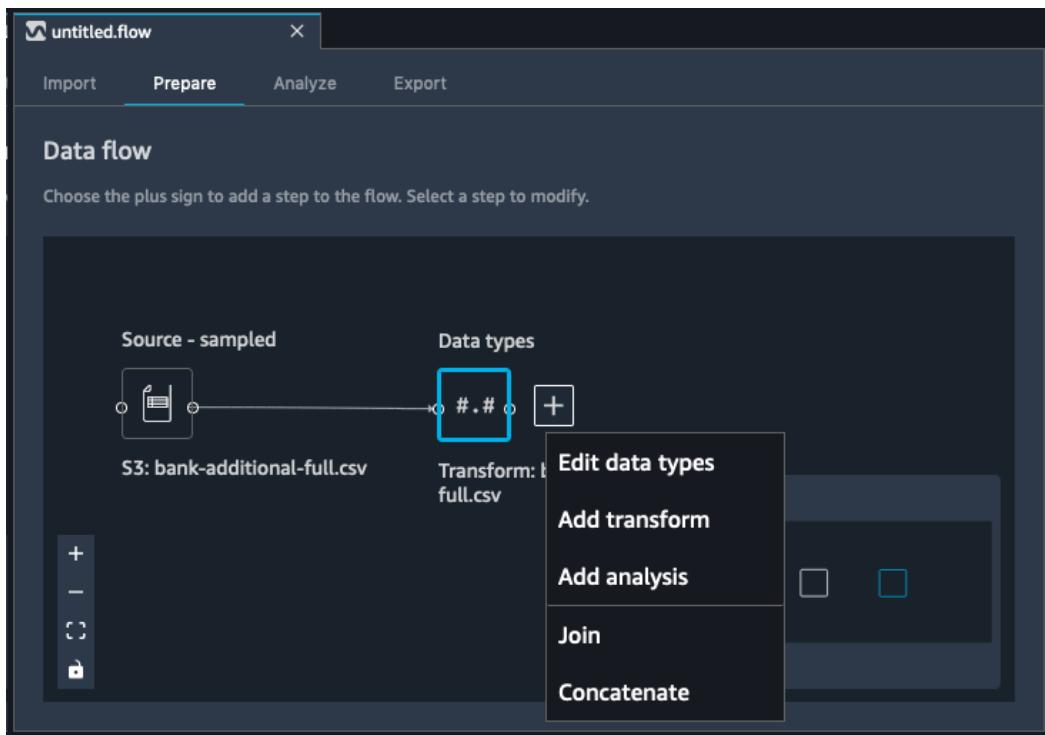


Figure 2.20 – Previewing a dataset

8. Clicking on the + icon, we could add more data sources, joining them or concatenating them to our dataset. We could also edit data types for all columns, should Data Wrangler have detected them incorrectly. Instead, let's select **Add analysis** to visualize properties of our dataset. This opens the **Analyze view**, visible in the next screenshot:

The screenshot shows the Data Wrangler interface with the 'Analyze' tab selected. The main area displays the 'Imported datasets / Transform: bank-additional-full.csv / Create Analysis' section. It includes a 'Histogram' icon with a note 'No Preview available' and links to 'Configure' built-in analyses or 'Code' for custom analysis. Below this is a 'Data table' section showing the first 8 rows of the dataset:

age	job	marital	education	default	housing
56	housemaid	married	basic.4y	no	no
57	services	married	high.school	unknown	no
37	services	married	high.school	no	yes
40	admin.	married	basic.6y	no	no
56	services	married	high.school	no	no
45	services	married	basic.9y	unknown	no
59	admin.	married	professional.course	no	no
41	blue-collar	married	unknown	unknown	no

The right side of the interface contains configuration options for the histogram analysis, including 'Analysis type' set to 'Histogram', a note about a 100,000 row limit, and fields for 'Analysis name' (set to 'Untitled'), 'X axis' (set to 'Select...'), 'Color by' (set to 'Select...'), and 'Facet by' (set to 'Select...'). Buttons for 'Clear', 'Preview', and 'Save' are at the bottom right.

Figure 2.21 – Visualizing a dataset

9. The next screenshot shows a scatter plot on duration vs. age. See how easy this is? You can experiment by selecting different columns, click on **Preview** to see results, and click on **Save** to create the analysis and save it for further use.

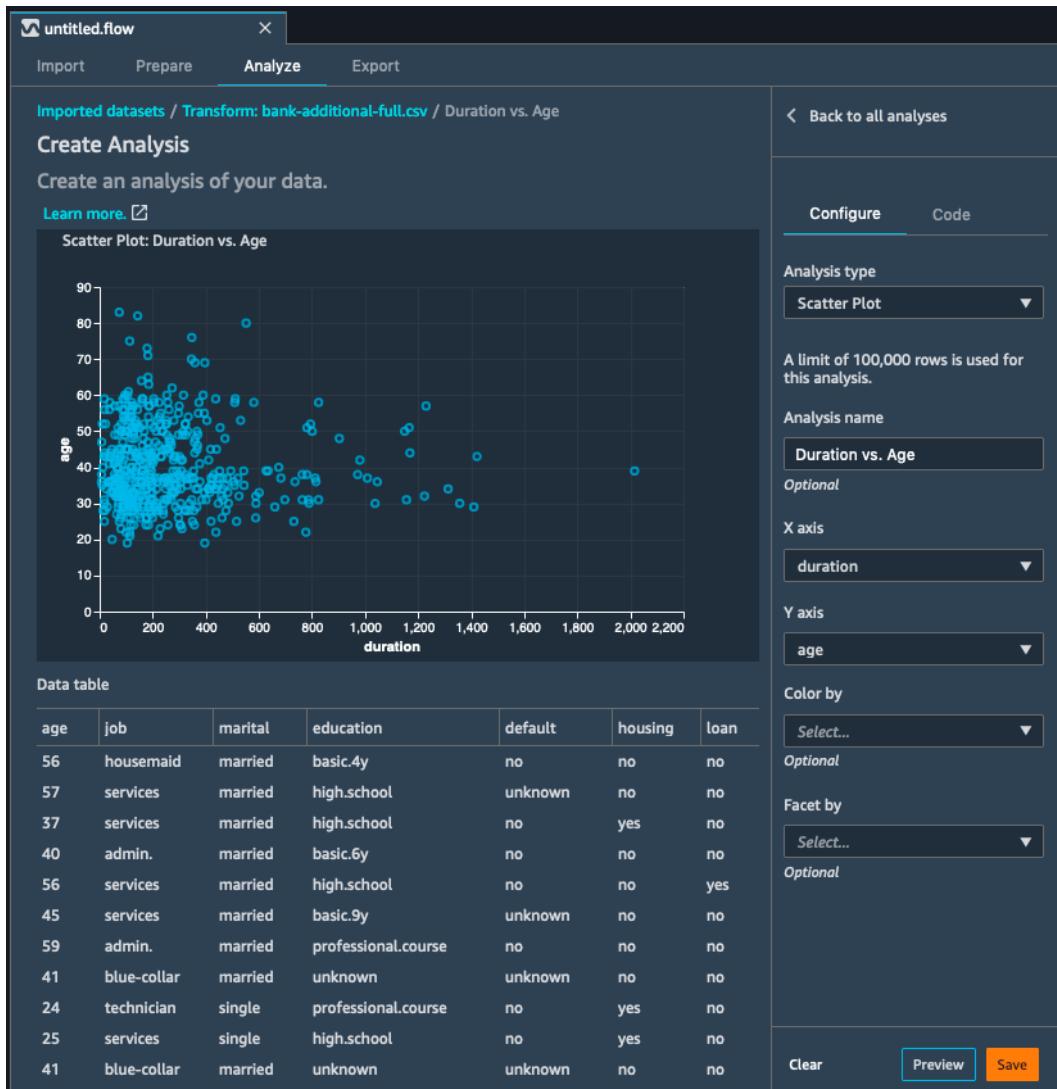


Figure 2.22 – Building a scatter plot

10. On top of histograms and scatter plots, we can also build **Table Summary**, **Bias Analysis**, and **Target Leakage** reports. Let's build the latter to find out if certain columns are either leaking into the prediction, or not helpful at all. You can see the report in the next screenshot:

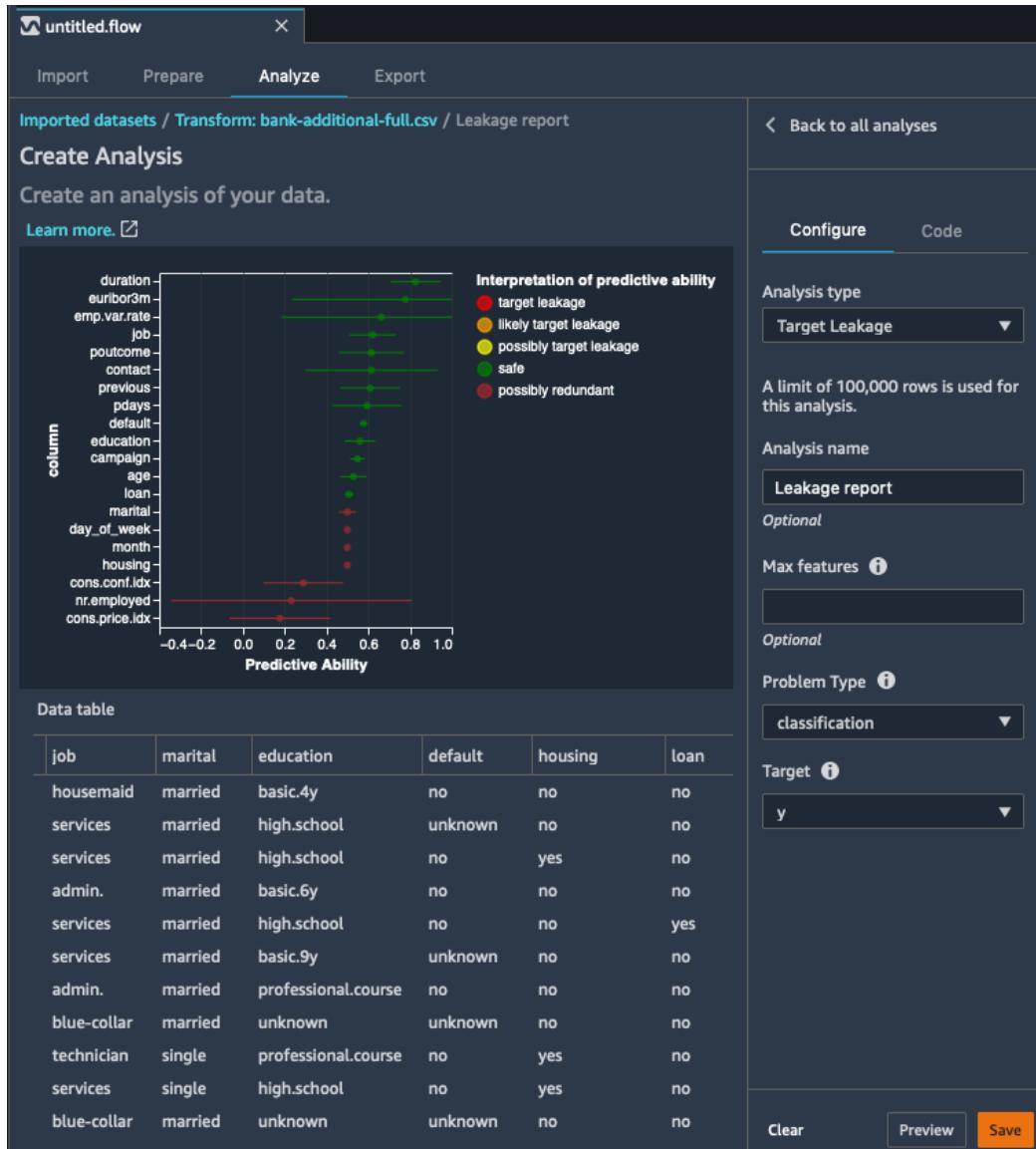


Figure 2.23 – Building a target leakage report

11. This report tells us that no column is leaking (all scores are lower than 1). Several columns are also not useful in predicting the target (some scores are 0.5 or lower): we should probably drop these columns during data processing.

We could also try the **Quick Model** report, which trains a model using a **Random Forest** algorithm implemented with Spark, right in SageMaker Studio. Unfortunately, an error message pops up, complaining about column names. Indeed, some column names include a dot, which is not allowed by Spark. No problem, we can easily fix this during data processing, and build the report later.

In fact, let's move on to transforming data with Data Wrangler.

Transforming a dataset in SageMaker Data Wrangler

Data Wrangler includes hundreds of built-in transforms, and we can also add our own.

1. Starting from the **Prepare** view visible in the next screenshot, we click on the + icon to add transforms.

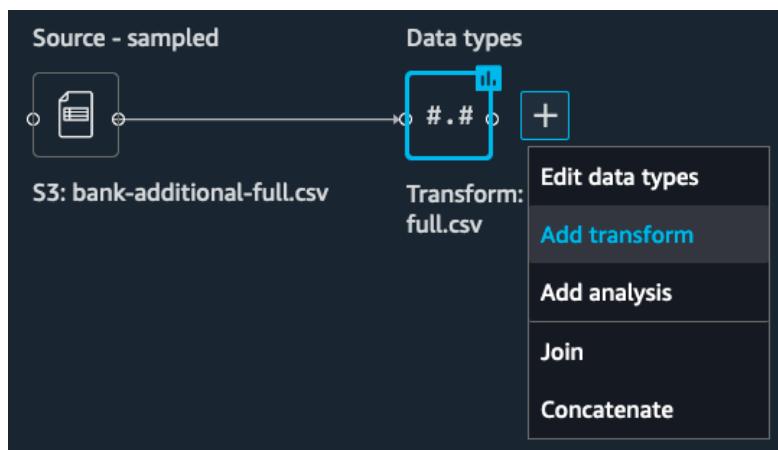


Figure 2.24 – Adding a transform

2. This opens the list of transforms, shown in the next screenshot. Take a minute to explore them.

3. Let's start by dropping the columns flagged as useless in the **Target Leakage** report: marital, day of week, month, housing, cons.conf.idx, nr.employed, cons.price.idx. We click on **Manage columns**, select the **Drop column** transform, and pick the marital column. Your screen should look like the following screenshot:

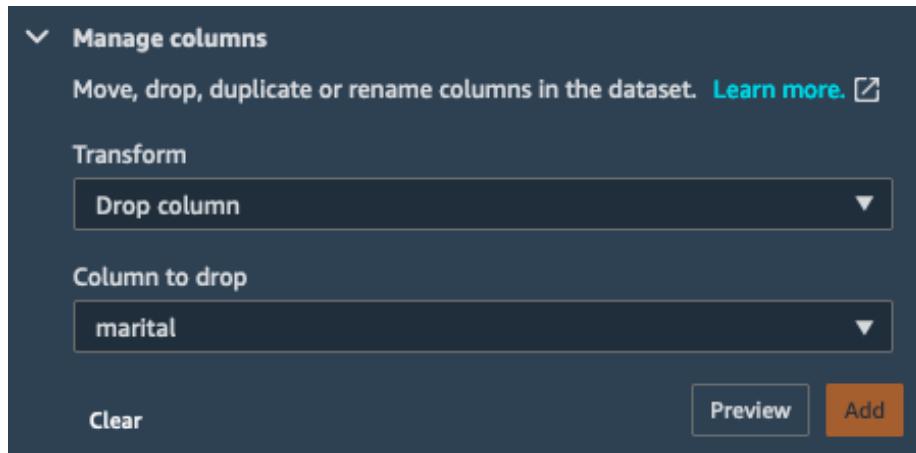


Figure 2.25 – Dropping a column

4. We can preview results and add the transform to our pipeline. We'll repeat the same operations for the other columns we want to drop.
5. Now, let's remove these annoying dots in column names, replacing them with underscores. The easiest way to do this is to use a **custom transform** in PySpark, as visible in the next screenshot. The dataset is available as a Pandas dataframe named df.

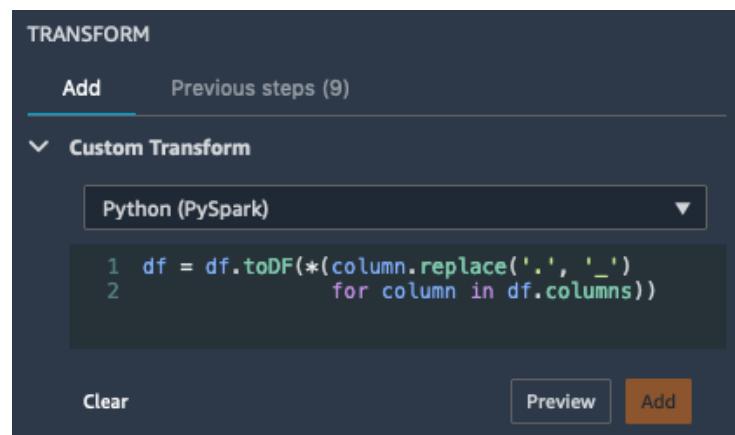


Figure 2.26 – Applying a custom transform

6. Jumping back to the **Analyze** view, and clicking on **Steps**, we can see the list of transforms that we've already applied, as shown in the next screenshot. We could also delete each transform by clicking on the icon to the right of it.

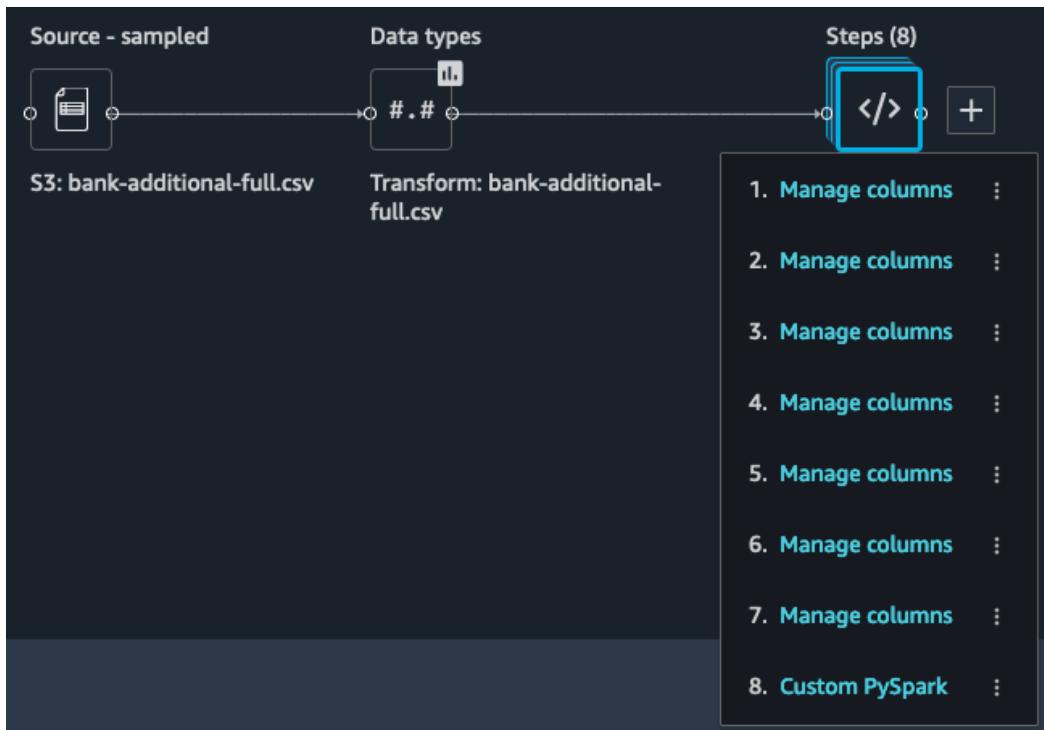


Figure 2.27 – Viewing a pipeline

7. Clicking on the + icon, we select **Add analysis** then we create a **Quick Model** on the y label, as shown in the next screenshot. The F1 score for this classification model is 0.881, and the most important features are duration, euribor3m, and pdays. By applying more transforms and building a quick model again, we can iteratively measure the positive impact (or the lack thereof) of our feature engineering steps.

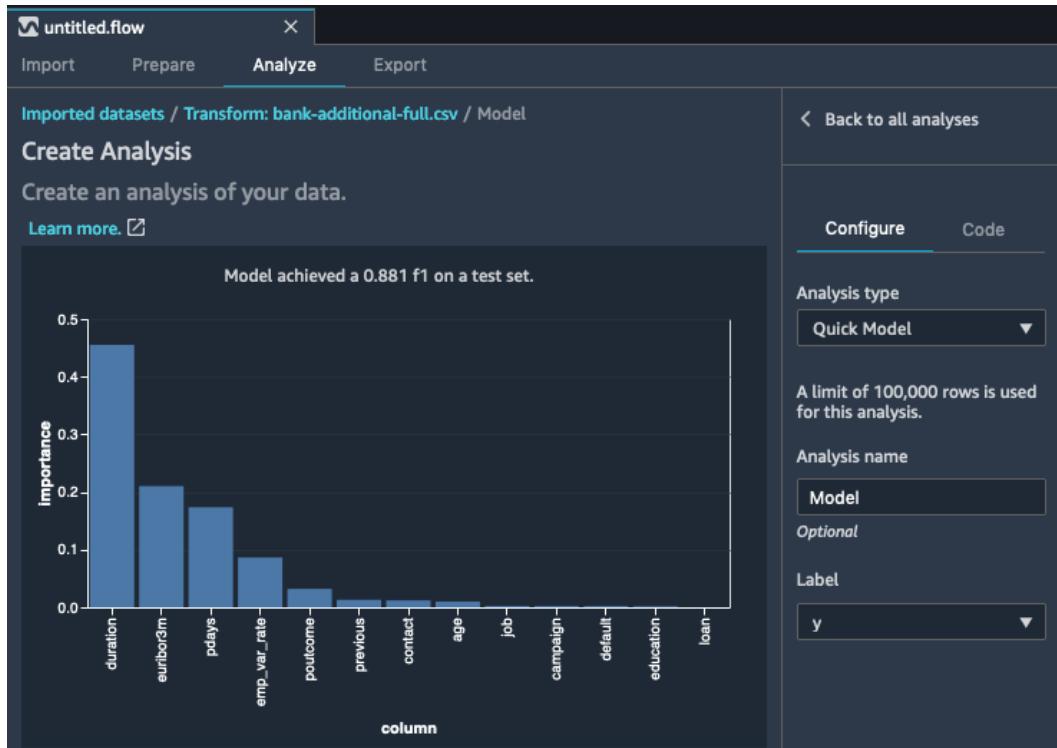


Figure 2.28 – Building a quick model

8. Coming back to the **Prepare** view, let's add a few more transforms. Our data set contains two categorical features: job and education. We decide to encode them to help algorithms understand that the different values are different dimensions to the problem. Starting with job, we apply the **Encode categorical** transform. As visible in the following screenshot, we see new columns for each job name. The original job column is automatically dropped.

Figure 2.29 – One-hot encoding a column

- The `job_admin.` column name contains a dot! We can remove it with the **Manage columns|Rename column** transform. Now, let's one-hot encode the `education` column... and remove the dots in column names. We could apply **Process numeric** transforms to scale and normalize numerical columns, but let's stop there for now. Feel free to explore and experiment!
- One last thing: Data Wrangler workflows are stored in `.flow` files, visible in the Jupyter file view. These are JSON files that you can (and should) store in your Git repositories, in order to reuse them later and share them with other team members.

Now that our pipeline is ready, let's see how we can export it to Python code. All it takes is a single click, and we won't have to write a single line of code.

Exporting a SageMaker Data Wrangler pipeline

Data Wrangler makes it easy to export a pipeline in four ways:

- Plain Python code that you can readily include in your machine learning project.
- A Jupyter notebook running a SageMaker Processing job, which will apply the pipeline to your dataset and save results in S3. The notebook also includes optional code to train a model.
- A Jupyter notebook storing the processed dataset in SageMaker Feature Store.
- A Jupyter notebook creating a SageMaker Pipelines workflow, with steps to process your dataset and train a model on it.

OK, let's go for it:

1. Starting from the **Export** view, we click on Steps and select the steps we'd like to export. Here, I selected them all, as shown in the next screenshot:

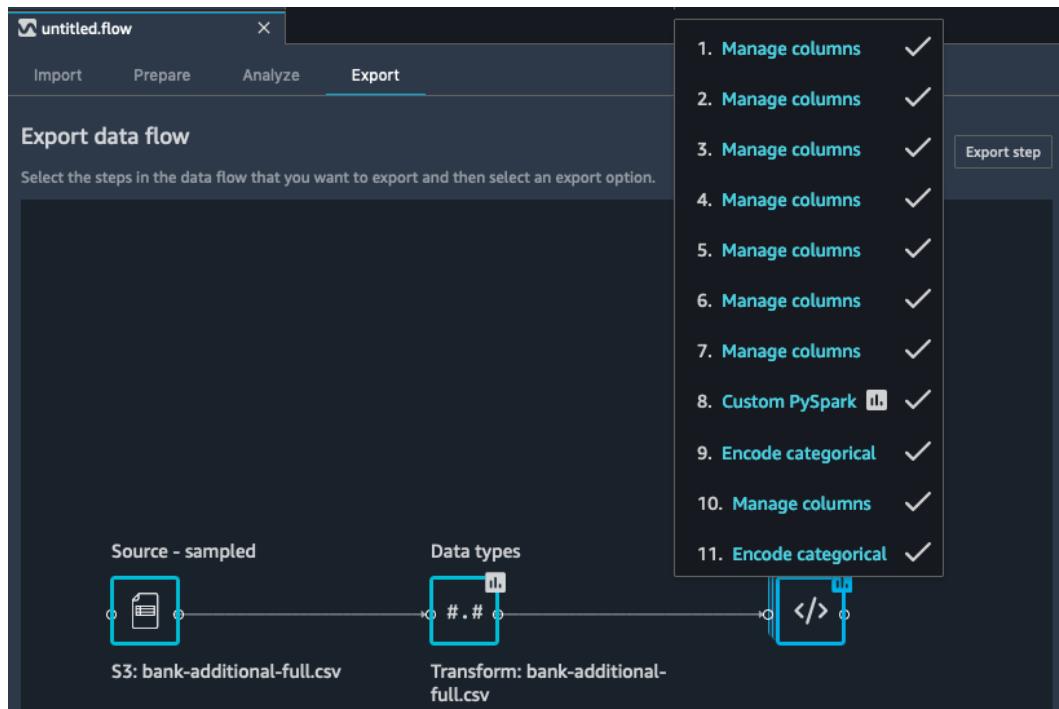


Figure 2.30 – Selecting steps to export

2. Then, we simply click on **Export step** and select one of the four options. Here, I go for **Save to S3** in order to run a SageMaker Processing job.

- This opens a new notebook. We'll discuss SageMaker Processing in the next section, but let's go ahead and run the job. Once the Job Status & S3 Output Location cell is complete, our dataset is available in S3, as visible in the next screenshot:

```
s3_job_results_path = f"s3://{bucket}/{s3_output_prefix}/{processing_job_name}"
print(f"Job results are saved to S3 path: {s3_job_results_path}")

job_result = sess.wait_for_processing_job(processing_job_name)
job_result

Job results are saved to S3 path: s3://sagemaker-ap-northeast-2-
-47ee047b/output/data-wrangler-flow-processing-13-08-54-36
.....! /export-flow-13-08-54-36
```

Figure 2.31 – Locating the processed dataset in S3

- Downloading and opening the CSV file stored at this location, we see that it contains the processed dataset, as shown in the next screenshot. In a typical machine learning workflow, we would then use this data directly to train a model.

```
age,default,loan,contact,duration,campaign,pdays,previous,poutcome,emp_var_rate,euribor3m,y,job_admin,job_blue-collar,job_technician,job_services,job_management,job_retired,job_entrepreneur,job_self-employed,job_housemaid,job_unemployed,job_student,job_unknown,education_university_degree,education_high_school,education_basic_9y,education_professional_course,education_basic_4y,education_basic_6y,education_unknown,education_illiterate
56,no,no,no,telephone,261,1,999,0,nonexistent,1,1,4,857,no,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
57,unknown,no,telephone,149,1,999,0,nonexistent,1,1,4,857,no,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0
37,no,no,telephone,226,1,999,0,nonexistent,1,1,4,857,no,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0
40,no,no,telephone,151,1,999,0,nonexistent,1,1,4,857,no,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0
56,no,yes,telephone,307,1,999,0,nonexistent,1,1,4,857,no,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0
45,unknown,no,telephone,198,1,999,0,nonexistent,1,1,4,857,no,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0
```

Figure 2.32 – Viewing the processed dataset

As you can see, SageMaker Data Wrangler makes it very easy (and even fun) to apply transforms to your datasets. Once you're done, you can immediately export them to Python code, without having to write a single line of code.

In the next section, we're going to learn about Amazon SageMaker Processing, a great way run batch jobs for data processing and other machine learning tasks.

Running batch jobs with Amazon SageMaker Processing

As discussed in the previous section, datasets usually need quite a bit of work to be ready for training. Once training is complete, you may also want to run additional jobs to post-process the predicted data and to evaluate your model on different datasets.

Once the experimentation phase is complete, it's good practice to start automating all these jobs, so that you can run them on demand with little effort.

Discovering the Amazon SageMaker Processing API

The Amazon SageMaker Processing API is part of the SageMaker SDK, which we installed in *Chapter 1, Introducing Amazon SageMaker*.

SageMaker Processing jobs run inside Docker containers:

- A built-in container for **scikit-learn** (<https://scikit-learn.org>)
- A built-in container for **PySpark** (<https://spark.apache.org/docs/latest/api/python/>), which supports distributed training
- Your own custom container

Logs are available in **Amazon CloudWatch Logs** in the /aws/sagemaker/ProcessingJobs log group.

Let's first see how we can use scikit-learn and SageMaker Processing to prepare a dataset for training.

Processing a dataset with scikit-learn

Here's the high-level process:

- Upload your unprocessed dataset to Amazon S3.
- Write a script with scikit-learn in order to load the dataset, process it, and save the processed features and labels.
- Run this script with SageMaker Processing on managed infrastructure.

Uploading the dataset to Amazon S3

We're going to reuse the direct marketing dataset introduced in the previous section, and apply our own transforms.

1. Creating a new Jupyter notebook, let's first download and extract the dataset:

```
%%sh
apt-get -y install unzip
wget -N https://sagemaker-sample-data-us-west-2.s3-us-
west-2.amazonaws.com/autopilot/direct_marketing/bank-
additional.zip
unzip -o bank-additional.zip
```

- Then, we load it with pandas:

```
import pandas as pd
data = pd.read_csv('./bank-additional/bank-additional-
full.csv')
print(data.shape)
(41188, 21)
```

- Now, let's display the first five lines:

```
data[:5]
```

This prints out the table visible in the following figure:

age	job	marital	education	default	housing	loan	contact	month	day_of_week	duration	campaign	pdays	previous	poutcome
0	56	housemaid	married	basic.4y	no	no	no	telephone	may	mon	261	1	999	0 nonexistent
1	57	services	married	high.school	unknown	no	no	telephone	may	mon	149	1	999	0 nonexistent
2	37	services	married	high.school	no	yes	no	telephone	may	mon	226	1	999	0 nonexistent
3	40	admin.	married	basic.6y	no	no	no	telephone	may	mon	151	1	999	0 nonexistent
4	56	services	married	high.school	no	no	yes	telephone	may	mon	307	1	999	0 nonexistent

Figure 2.33 – Viewing the dataset

Scrolling to the right, we can see a column named **y**, storing the labels.

- Now, let's upload the dataset to Amazon S3. We'll use a default bucket automatically created by SageMaker in the region we're running in. We'll just add a prefix to keep things nice and tidy:

```
import sagemaker
prefix = 'sagemaker/DEMO-smprocessing/input'
input_data = sagemaker.Session().upload_data(path='./
bank-additional/bank-additional-full.csv', key_-
prefix=prefix)
```

Writing a processing script with scikit-learn

As SageMaker Processing takes care of all infrastructure concerns, we can focus on the script itself. SageMaker Processing will also automatically copy the input dataset from S3 into the container, and the processed datasets from the container to S3.

Container paths are provided when we configure the job itself. Here's what we'll use:

- The input dataset: /opt/ml/processing/input
- The processed training set: /opt/ml/processing/train
- The processed test set: /opt/ml/processing/test

In our Jupyter environment, let's start writing a new Python file named `preprocessing.py`. As you would expect, this script will load the dataset, perform basic feature engineering, and save the processed dataset:

1. First, we read our single command-line parameter with the `argparse` library (<https://docs.python.org/3/library/argparse.html>): the ratio for the training and test datasets. The actual value will be passed to the script by the SageMaker Processing SDK:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument('--train-test-split-ratio',
                    type=float, default=0.3)
args, _ = parser.parse_known_args()
print('Received arguments {}'.format(args))
split_ratio = args.train_test_split_ratio
```

2. We load the input dataset using `pandas`. At startup, SageMaker Processing automatically copied it from S3 to a user-defined location inside the container, /`opt/ml/processing/input`:

```
import os
import pandas as pd
input_data_path = os.path.join('/opt/ml/processing/
input', 'bank-additional-full.csv')
df = pd.read_csv(input_data_path)
```

3. Then, we remove any line with missing values, as well as duplicate lines:

```
df.dropna(inplace=True)
df.drop_duplicates(inplace=True)
```

4. Then, we count negative and positive samples, and display the class ratio. This will tell us how unbalanced the dataset is:

```
one_class = df[df['y']=='yes']
one_class_count = one_class.shape[0]
zero_class = df[df['y']=='no']
zero_class_count = zero_class.shape[0]
zero_to_one_ratio = zero_class_count/one_class_count
print("Ratio: %.2f" % zero_to_one_ratio)
```

5. Looking at the dataset, we can see a column named pdays, telling us how long ago a customer has been contacted. Some lines have a 999 value, and that looks pretty suspicious: indeed, this is a placeholder value meaning that a customer has never been contacted. To help the model understand this assumption, let's add a new column stating it explicitly:

```
import numpy as np
df['no_previous_contact'] =
    np.where(df['pdays'] == 999, 1, 0)
```

6. In the job column, we can see three categories (student, retired, and unemployed) that should probably be grouped to indicate that these customers don't have a full-time job. Let's add another column:

```
df['not_working'] = np.where(np.in1d(df['job'],
    ['student', 'retired', 'unemployed']), 1, 0)
```

7. Now, let's split the dataset into training and test sets. Scikit-learn has a convenient API for this, and we set the split ratio according to a command-line argument passed to the script:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    df.drop('y', axis=1),
    df['y'],
    test_size=split_ratio, random_state=0)
```

8. The next step is to scale numerical features and to one-hot encode the categorical features. We'll use `StandardScaler` for the former, and `OneHotEncoder` for the latter:

```
from sklearn.compose import make_column_transformer
from sklearn.preprocessing import
StandardScaler,OneHotEncoder
preprocess = make_column_transformer(
    (StandardScaler(), ['age', 'duration', 'campaign',
'pdays', 'previous']),
    (OneHotEncoder(sparse=False), ['job', 'marital',
'education', 'default', 'housing', 'loan','contact',
'month', 'day_of_week', 'poutcome'])
)
```

9. Then, we process the training and test datasets:

```
train_features = preprocess.fit_transform(X_train)
test_features = preprocess.transform(X_test)
```

10. Finally, we save the processed datasets, separating the features and labels. They're saved to user-defined locations in the container, and SageMaker Processing will automatically copy the files to S3 before terminating the job:

```
train_features_output_path = os.path.join('/opt/ml/
processing/train', 'train_features.csv')
train_labels_output_path = os.path.join('/opt/ml/
processing/train', 'train_labels.csv')
test_features_output_path = os.path.join('/opt/ml/
processing/test', 'test_features.csv')
test_labels_output_path = os.path.join('/opt/ml/
processing/test', 'test_labels.csv')
pd.DataFrame(train_features).to_csv(train_features_
output_path, header=False, index=False)
pd.DataFrame(test_features).to_csv(test_features_output_
path, header=False, index=False)
y_train.to_csv(train_labels_output_path, header=False,
index=False)
y_test.to_csv(test_labels_output_path, header=False,
index=False)
```

That's it. As you can see, this code is vanilla scikit-learn, so it shouldn't be difficult to adapt your own scripts for SageMaker Processing. Now let's see how we can actually run this.

Running a processing script

Coming back to our Jupyter notebook, we use the `SKLearnProcessor` object from the SageMaker SDK to configure the processing job:

1. First, we define which version of scikit-learn we want to use, and what our infrastructure requirements are. Here, we go for an `ml.m5.xlarge` instance, an all-round good choice:

```
from sagemaker.sklearn.processing import SKLearnProcessor
sklearn_processor = SKLearnProcessor(
    framework_version='0.23-1',
    role=sagemaker.get_execution_role(),
    instance_type='ml.m5.xlarge',
    instance_count=1)
```

2. Then, we simply launch the job, passing the name of the script, the dataset input path in S3, the user-defined dataset paths inside the SageMaker Processing environment, and the command-line arguments:

```
from sagemaker.processing import ProcessingInput,
ProcessingOutput
sklearn_processor.run(
    code='preprocessing.py',
    inputs=[ProcessingInput(
        source=input_data,    # Our data in S3
        destination='/opt/ml/processing/input')
    ],
    outputs=[
        ProcessingOutput(
            source='/opt/ml/processing/train',
            output_name='train_data'),
        ProcessingOutput(
            source='/opt/ml/processing/test',
            output_name='test_data'
        )
    ],
```

```
    arguments=['--train-test-split-ratio', '0.2']
)
```

As the job starts, SageMaker automatically provisions a managed ml.m5.xlarge instance, pulls the appropriate container to it, and runs our script inside the container. Once the job is complete, the instance is terminated, and we only pay for the amount of time we used it. There is zero infrastructure management, and we'll never leave idle instances running for no reason.

3. After a few minutes, the job is complete, and we can see the output of the script as follows:

```
Received arguments Namespace(train_test_split_ratio=0.2)
Reading input data from /opt/ml/processing/input/bank-
additional-full.csv
Positive samples: 4639
Negative samples: 36537
Ratio: 7.88
Splitting data into train and test sets with ratio 0.2
Running preprocessing and feature engineering
transformations
Train data shape after preprocessing: (32940, 58)
Test data shape after preprocessing: (8236, 58)
Saving training features to /opt/ml/processing/train/
train_features.csv
Saving test features to /opt/ml/processing/test/test_
features.csv
Saving training labels to /opt/ml/processing/train/train_
labels.csv
Saving test labels to /opt/ml/processing/test/test_
labels.csv
```

The following screenshot shows the same log in **CloudWatch**:

The screenshot shows a CloudWatch Log Group for a SageMaker Processing job. The log entries are as follows:

- Received arguments Namespace(train_test_split_ratio=0.2)
- Reading input data from /opt/ml/processing/input/bank-additional-full.csv
- Positive samples: 4639
- Negative samples: 36537
- Ratio: 7.88
- Splitting data into train and test sets with ratio 0.2
- Running preprocessing and feature engineering transformations
- Train data shape after preprocessing: (32940, 58)
- Test data shape after preprocessing: (8236, 58)
- Saving training features to /opt/ml/processing/train/train_features.csv
- Saving test features to /opt/ml/processing/test/test_features.csv
- Saving training labels to /opt/ml/processing/train/train_labels.csv
- Saving test labels to /opt/ml/processing/test/test_labels.csv

Figure 2.34 – Viewing the log in CloudWatch Logs

- Finally, we can describe the job and see the location of the processed datasets:

```
preprocessing_job_description =
    sklearn_processor.jobs[-1].describe()
output_config = preprocessing_job_
description['ProcessingOutputConfig']
for output in output_config['Outputs']:
    print(output['S3Output']['S3Uri'])
```

This results in the following output:

```
s3://sagemaker-eu-west-1-123456789012/sagemaker-scikit-
learn-2020-04-22-10-09-43-146/output/train_data
s3://sagemaker-eu-west-1-123456789012/sagemaker-scikit-
learn-2020-04-22-10-09-43-146/output/test_data
```

In a terminal, we can use the AWS CLI to fetch the processed training set located at the preceding path, and take a look at the first sample and label:

```
$ aws s3 cp s3://sagemaker-eu-west-1-123456789012/
sagemaker-scikit-learn-2020-04-22-09-45-05-711/output/
train_data/train_features.csv .
$ aws s3 cp s3://sagemaker-eu-west-1-123456789012/
sagemaker-scikit-learn-2020-04-22-09-45-05-711/output/
train_data/train_labels.csv .
```

```
$ head -1 train_features.csv  
0.09604515376959515,-0.6572847857673993,-  
0.20595554104907898,0.19603112301129622,-  
0.35090125695736246,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,  
1.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,  
0.0,1.0,0.0,0.0,1.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,  
0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,  
1.0,0.0  
$ head -1 train_labels.csv  
no
```

Now that the dataset has been processed with our own code, we could use it to train a machine learning model. In real life, we would also automate these steps instead of running them manually inside a notebook.

Important Note

One last thing: here, our job writes output data to S3. SageMaker Processing also supports writing directly to an existing Feature Group in **SageMaker Feature Store** (which we'll introduce later in the book). API details are available at <https://sagemaker.readthedocs.io/en/stable/api/training/processing.html#sagemaker.ProcessingOutput>.

Processing a dataset with your own code

In the previous example, we used a built-in container to run our scikit-learn code. SageMaker Processing also makes it possible to use your own container. You can find an example at <https://docs.aws.amazon.com/sagemaker/latest/dg/build-your-own-processing-container.html>.

As you can see, SageMaker Processing makes it really easy to run data processing jobs. You can focus on writing and running your script, without having to worry about provisioning and managing infrastructure.

Summary

In this chapter, you learned how Amazon SageMaker Ground Truth helps you build highly accurate training datasets using image and text labeling workflows. We'll see in *Chapter 5, Training Computer Vision Models*, how to use image datasets labeled with Ground Truth.

Then, you learned about Amazon SageMaker Processing, a capability that helps you run your own data processing workloads on managed infrastructure: feature engineering, data validation, model evaluation, and so on.

Finally, we discussed three other AWS services (Amazon EMR, AWS Glue, and Amazon Athena), and how they could fit into your analytics and machine learning workflows.

In the next chapter, we'll start training models using the built-in machine learning models of Amazon SageMaker.

Section 2: Building and Training Models

In this section, you will understand how to build and train machine learning models with Amazon SageMaker. This part covers AutoML, built-in algorithms, built-in frameworks, and bring your own code. Using notebooks based on the SageMaker SDK, it will explain how to read training data, how to set up training jobs, how to define training parameters, and how to train on fully managed infrastructure.

This section comprises the following chapters:

- *Chapter 3, AutoML with Amazon SageMaker AutoPilot*
- *Chapter 4, Training Machine Learning Models*
- *Chapter 5, Training Computer Vision Models*
- *Chapter 6, Training Natural Language Processing Models*
- *Chapter 7, Extending Machine Learning Services Using Built-In Frameworks*
- *Chapter 8, Using Your Algorithms and Code*

3

AutoML with Amazon SageMaker Autopilot

In the previous chapter, you learned how Amazon SageMaker helps you build and prepare datasets. In a typical machine learning project, the next step would be to start experimenting with algorithms in order to find an early fit and get a sense of the predictive power you could expect from the model.

Whether you work with traditional machine learning or deep learning, three options are available when it comes to selecting an algorithm:

- Write your own, or customize an existing one. This only makes sense if you have strong skills in statistics and computer science, if you're quite sure that you can do better than well-tuned, off-the-shelf algorithms, and if you're given enough time to work on the project. Let's face it, these conditions are rarely met.
- Use a built-in algorithm implemented in one of your favorite libraries, such as **linear regression** or **XGBoost**. For deep learning problems, this includes pre-trained models available in **TensorFlow**, **PyTorch**, and so on. This option saves you the trouble of writing machine learning code. Instead, it lets you focus on feature engineering and model optimization.

- Use **AutoML**, a rising technique that lets you automatically build, train, and optimize machine learning models.

In this chapter, you will learn about **Amazon SageMaker Autopilot**, an AutoML capability part of Amazon SageMaker with built-in model explainability. We'll see how to use it in Amazon SageMaker Studio without writing a single line of code, and also how to use it with the Amazon SageMaker SDK:

- Discovering Amazon SageMaker Autopilot
- Using Amazon SageMaker Autopilot in SageMaker Studio
- Using Amazon SageMaker Autopilot with the SageMaker SDK
- Diving deep on Amazon SageMaker Autopilot

Technical requirements

You will need an AWS account to run the examples included in this chapter. If you haven't got one already, please point your browser at <https://aws.amazon.com/getting-started/> to create it. You should also familiarize yourself with the AWS Free Tier (<https://aws.amazon.com/free/>), which lets you use many AWS services for free within certain usage limits.

You will need to install and configure the AWS **Command-Line Interface (CLI)** for your account (<https://aws.amazon.com/cli/>).

You will need a working Python 3.x environment. Installing the Anaconda distribution (<https://www.anaconda.com/>) is not mandatory, but is strongly encouraged as it includes many projects that we will need (Jupyter, pandas, numpy, and more).

Code examples included in the book are available on GitHub at <https://github.com/PacktPublishing/Learn-Amazon-SageMaker-second-edition>. You will need to install a Git client to access them (<https://git-scm.com/>).

Discovering Amazon SageMaker Autopilot

Added to Amazon SageMaker in late 2019, **Amazon SageMaker Autopilot** is an AutoML capability that takes care of all the machine learning steps for you. You only need to upload a columnar dataset to an Amazon S3 bucket and define the column you want the model to learn (the **target attribute**). Then, you simply launch an Autopilot job, with either a few clicks in the SageMaker Studio GUI or a couple of lines of code with the SageMaker SDK.

The simplicity of SageMaker Autopilot doesn't come at the expense of transparency and control. You can see how your models are built, and you can keep experimenting to refine results. In that respect, SageMaker Autopilot should appeal to new and seasoned practitioners alike.

In this section, you'll learn about the different steps of a SageMaker Autopilot job and how they contribute to delivering high-quality models:

- Analyzing data
- Feature engineering
- Model tuning

Let's start by seeing how SageMaker Autopilot analyzes data.

Analyzing data

This step is responsible for understanding what type of machine learning problem we're trying to solve. SageMaker Autopilot currently supports **linear regression**, **binary classification**, and **multi-class classification**.

Note

A frequent question is "how much data is needed to build such models?" This is a surprisingly difficult question. The answer—if there is one—depends on many factors, such as the number of features and their quality. As a basic rule of thumb, some practitioners recommend having 10–100 times more samples than features. In any case, I'd advise you to collect no fewer than hundreds of samples (for each class, if you're building a classification model). Thousands or tens of thousands are better, especially if you have more features. For statistical machine learning, there is rarely a need for millions of samples, so start with what you have, analyze the results, and iterate before going on a data collection rampage!

By analyzing the distribution of the target attribute, SageMaker Autopilot can easily figure out which one is the right one. For instance, if the target attribute has only two values (say, "yes" and "no"), you're likely trying to build a binary classification model.

Then, SageMaker Autopilot computes statistics on the dataset and individual columns: the number of unique values, the mean, median, and so on. Machine learning practitioners very often do this in order to get an initial feel for the data, and it's nice to see it automated. In addition, SageMaker Autopilot generates a Jupyter notebook, the **data exploration notebook**, to present these statistics in a user-friendly way.

Once SageMaker Autopilot has analyzed the dataset, it builds **candidate pipelines** that will be used to train candidate models. A pipeline is a combination of the following:

- A data processing job, in charge of feature engineering. As you can guess, this job runs on **Amazon SageMaker Processing**, which we studied in *Chapter 2, Handling Data Preparation Techniques*.
- A training job, running on the processed dataset. Algorithms include the built-in Linear Learner in SageMaker, XGBoost, and multi-layer perceptrons.

Next, let's see how Autopilot can be used in feature engineering.

Feature engineering

This step is responsible for pre-processing the input dataset according to the pipelines defined during data analysis.

Candidate pipelines are fully documented in another autogenerated notebook – the **candidate generation notebook**. This notebook isn't just descriptive: you can actually run its cells, and manually reproduce the steps performed by SageMaker Autopilot. This level of transparency and control is extremely important as it lets you understand exactly how the model was built. Thus, you're able to verify that it performs the way it should, and you're able to explain it to your stakeholders. Also, you can use the notebook as a starting point for additional optimization and tweaking if you're so inclined.

Lastly, let's take a look at model tuning in Autopilot.

Model tuning

This step is responsible for training and tuning models according to the pipelines defined during data analysis. For each pipeline, SageMaker Autopilot will launch an **automatic model tuning** job (we'll cover this topic in detail in a later chapter). In a nutshell, each tuning job will use **hyperparameter optimization** to train a large number of increasingly accurate models on the processed dataset. As usual, all of this happens on managed infrastructure.

Once the model tuning is complete, you can view the model information and metrics in Amazon SageMaker Studio, build visualizations, and so on. You can do the same programmatically with the **Amazon SageMaker Experiments** SDK.

Finally, you can deploy your model of choice just like any other SageMaker model using either the SageMaker Studio GUI or the SageMaker SDK.

Now that we understand the different steps of an Autopilot job, let's run a job in SageMaker Studio.

Using Amazon SageMaker Autopilot in SageMaker Studio

We will build a model using only SageMaker Studio. We won't write a line of machine learning code, so get ready for zero-code AI.

In this section, you'll learn how to do the following:

- Launch a SageMaker Autopilot job in SageMaker Studio.
- Monitor the different steps of the job.
- Visualize models and compare their properties.

Launching a job

First, we need a dataset. We'll reuse the direct marketing dataset used in *Chapter 2, Handling Data Preparation Techniques*. This dataset describes a binary classification problem: will a customer accept a marketing offer, yes or no? It contains a little more than 41,000 labeled customer samples. Let's dive in:

1. Let's open SageMaker Studio. Create a new Python 3 notebook using the **Data Science** kernel, as shown in the following screenshot:



Figure 3.1 – Creating a notebook

2. Now, let's download and extract the dataset as follows:

```
%%sh
apt-get -y install unzip
wget -N https://sagemaker-sample-data-us-west-2.s3-us-
west-2.amazonaws.com/autopilot/direct_marketing/bank-
additional.zip
unzip -o bank-additional.zip
```

3. In *Chapter 2, Handling Data Preparation Techniques*, we ran a feature engineering script with Amazon SageMaker Processing. We will do no such thing here: we simply upload the dataset as is to S3, into the **default bucket** created by SageMaker:

```
import sagemaker
prefix = 'sagemaker/DEMO-autopilot/input'
sess = sagemaker.Session()
uri = sess.upload_data(path='./bank-additional/bank-
additional-full.csv', key_prefix=prefix)
print(uri)
```

The dataset will be available in S3 at the following location:

```
s3://sagemaker-us-east-2-123456789012/sagemaker/DEMO-
autopilot/input/bank-additional-full.csv
```

4. Now, we click on the **Components and registries** icon in the left-hand vertical icon bar, as can be seen in the following screenshot. This opens the **Experiments** tab, and we click on the **Create Autopilot Experiment** button to create a new Autopilot job.

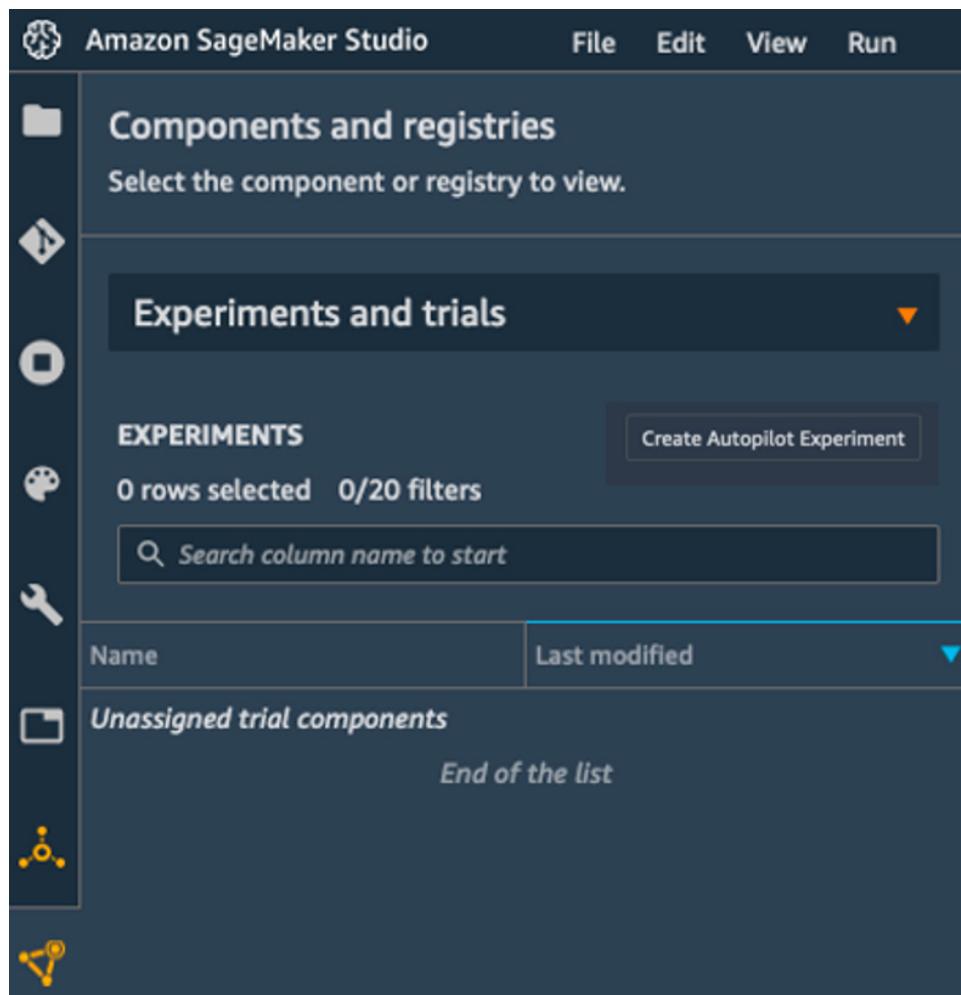


Figure 3.2 – Viewing experiments

5. The next screen is where we configure the job. Let's enter `my-first-autopilot-job` as the experiment name.

6. We set the location of the input dataset using the path returned in *step 3*. As can be seen in the following screenshot, we can either browse S3 buckets or enter the S3 location directly:

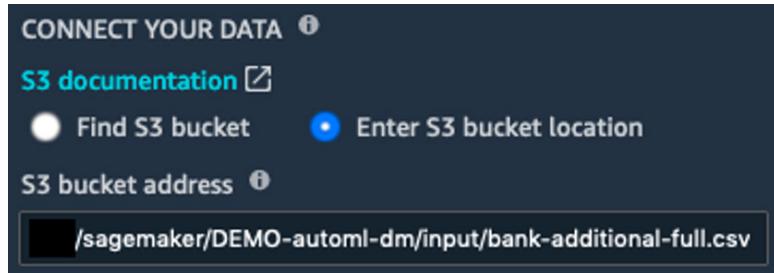


Figure 3.3 – Defining the input location

7. The next step is to define the name of the **target attribute**, as shown in the following screenshot. The column storing the "yes" or "no" label is called "y".

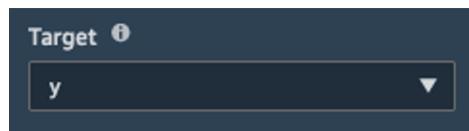


Figure 3.4 – Defining the target attribute

8. As shown in the following screenshot, we set the S3 output location where job artifacts will be copied to. I use `s3://sagemaker-us-east-2-123456789012/sagemaker/DEMO-autopilot/output/` here, and you should, of course, update it with your own region and account number.

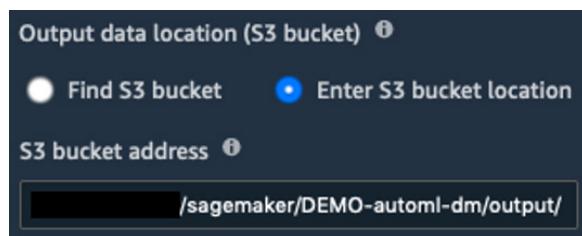


Figure 3.5 – Defining the output location

9. We set the type of job we want to train, as shown in the following screenshot. Here, we select **Auto** in order to let SageMaker Autopilot figure out the problem type. Alternatively, we could select **Binary classification**, and pick our metric: **Accuracy**, **AUC**, or **F1** (the default setting).

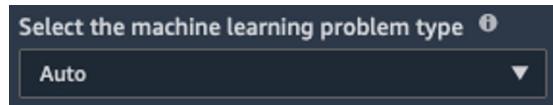


Figure 3.6 – Setting the problem type

10. Finally, we decide whether we want to run a full job, or simply generate notebooks. We'll go with the former, as shown in the following screenshot. The latter would be a good option if we wanted to train and tweak the parameters manually. We also decide not to deploy the best model automatically for now.

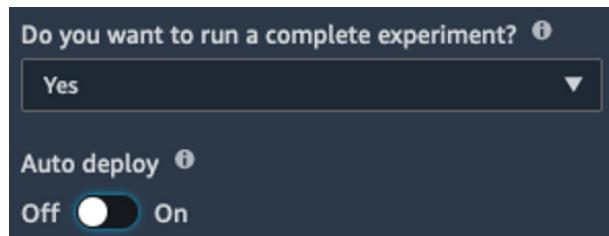


Figure 3.7 – Running a complete experiment

11. Optionally, in the **Advanced Settings** section, we would change the IAM role, set an encryption key for job artifacts, define the VPC where we'd like to launch job instances, and so on. Let's keep default values here.
12. The job setup is complete: all it took was this one screen. Then, we click on **Create Experiment**, and off it goes!

Monitoring a job

Once the job is launched, it goes through the three steps that we already discussed, which should take around 5 hours to complete. The new experiment is listed in the **Experiments** tab, and we can right-click **Describe AutoML Job** to describe its current status. This opens the following screen, where we can see the progress of the job:

- As expected, the job starts by analyzing data, as highlighted in the following screenshot:

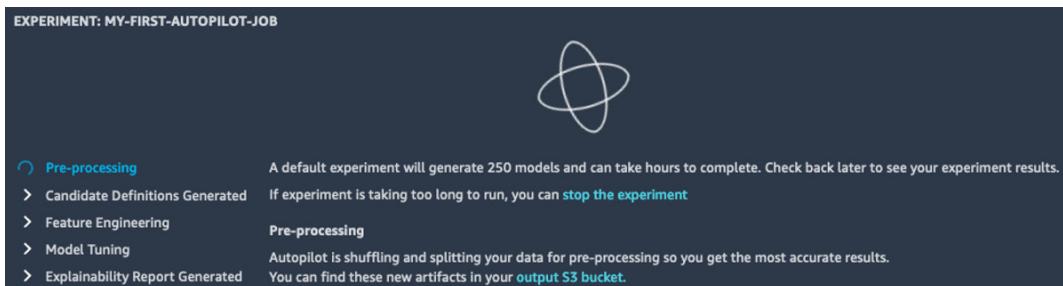


Figure 3.8 – Viewing job progress

- About 10 minutes later, data analysis is complete, and the job moves on to feature engineering, where the input dataset will be transformed according to the steps defined in the candidate pipelines. As shown in the following screenshot, we can also see new two buttons in the top-right corner, pointing at the **candidate generation** and **data exploration** notebooks: don't worry, we'll take a deeper look at both later in the chapter.

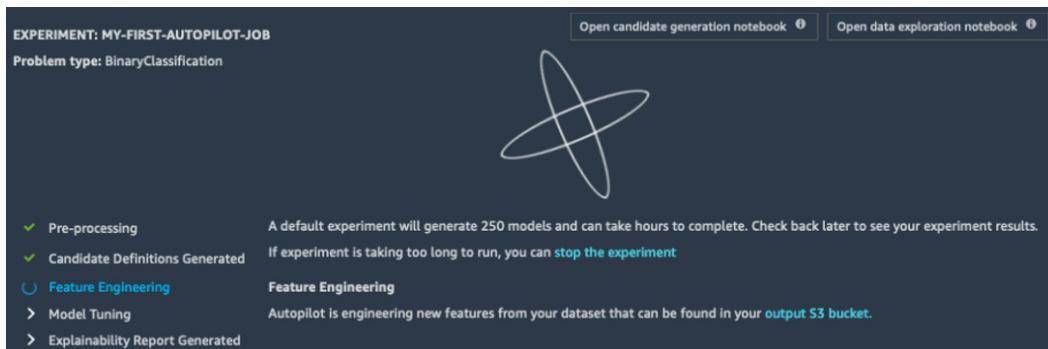


Figure 3.9 – Viewing job progress

- Once feature engineering is complete, the job then moves on to model tuning, where candidate models are trained and tuned. As can be seen in the following screenshot, the first training jobs quickly show up in the **Trials** tab. A "trial" is the name SageMaker uses for a collection of related jobs, such as processing jobs, batch transform jobs, and training jobs. We can see the **Objective**, that is to say, the metric that the job tried to optimize (in this case, it's the F1 score). We can sort jobs based on this metric, and the best tuning job so far is highlighted with a star.

The screenshot shows the SageMaker Studio interface for an experiment named "MY-FIRST-AUTOPILOT-JOB". The problem type is set to "BinaryClassification". The interface includes sections for experiment status, tuning parameters, and a table of completed trials.

EXPERIMENT: MY-FIRST-AUTOPILOT-JOB

Problem type: BinaryClassification

Open candidate generation notebook ⓘ | Open data exploration notebook ⓘ

Pre-processing: A default experiment will generate 250 models and can take hours to complete. Check back later to see your experiment results.

Candidate Definitions Generated: If experiment is taking too long to run, you can [stop the experiment](#).

Feature Engineering: Model Tuning

Model Tuning: Autopilot is tuning your models with Hyperparameter optimization.

Explainability Report Generated

Trials Job profile

TRIALS 0 rows selected Deploy model

Trial name	Status	Start time	Objective: F1
★ Best: tuning-job-1-446ba4a27c444dc895-010-3...	Completed	2 minutes ago	0.7540900111198425
tuning-job-1-446ba4a27c444dc895-006-997a6e0c	Completed	2 minutes ago	0.7465599775314331
tuning-job-1-446ba4a27c444dc895-007-9172f970	Completed	2 minutes ago	0.7462400197982788
tuning-job-1-446ba4a27c444dc895-008-8877f052	Completed	2 minutes ago	0.745710015296936
tuning-job-1-446ba4a27c444dc895-001-a85ffccdd	Completed	2 minutes ago	0.7264599800109863
tuning-job-1-446ba4a27c444dc895-002-54c2aa17	Completed	2 minutes ago	0.7264599800109863

Figure 3.10 – Viewing tuning jobs

4. Once the AutoPilot job is complete, your screen should look similar to the following screenshot. Here, the top model has reached an F1 score of 0.8031.

TRIALS 0 rows selected		Deploy model		
Trial name	Status	Start time	Objective: F1	
★ Best: tuning-job-1-446ba4a27c444dc895-236-e...	Completed	22 minutes ago	0.8031499981880188	188
tuning-job-1-446ba4a27c444dc895-173-9cdf03b6	Completed	38 minutes ago	0.8027499914169312	312
tuning-job-1-446ba4a27c444dc895-147-7b675689	Completed	43 minutes ago	0.8026400208473206	206
tuning-job-1-446ba4a27c444dc895-237-7e2a8ca4	Completed	22 minutes ago	0.8024799823760986	986
tuning-job-1-446ba4a27c444dc895-155-48310106	Completed	42 minutes ago	0.8022900223731995	1995

Figure 3.11 – Viewing results

5. If we select the best job and right-click **Open in model details**, we can see a model explainability graph showing us the most important features, as can be seen in the following screenshot. This graph is based on global SHapley Additive exPlanations (SHAP) (<https://github.com/slundberg/shap>) values computed automatically by AutoPilot.

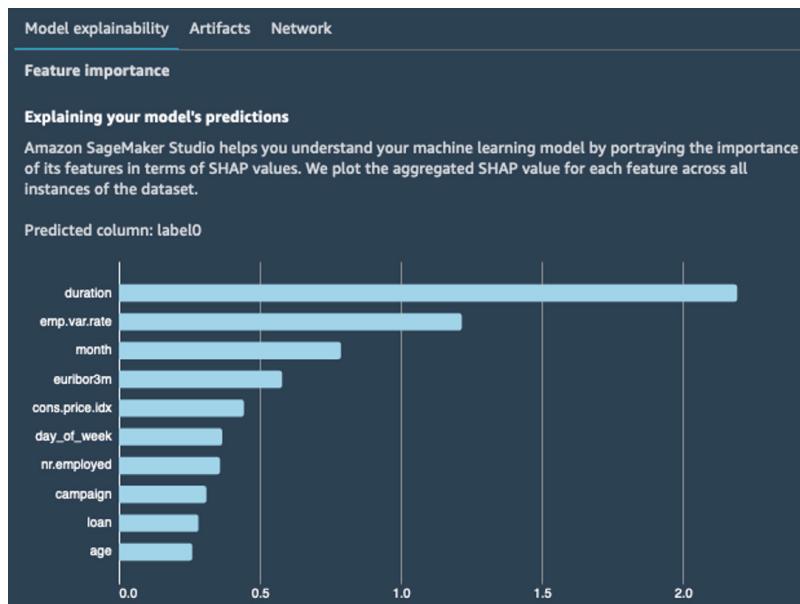


Figure 3.12 – Viewing the most important features

- In the **Artifacts** tab, we can also see a list of training artifacts and parameters involved in building the model: input data, training and validation splits, transformed datasets, feature engineering code, the algorithm (XGBoost in my case), and more.

At this point, we could simply deploy the best job, but instead, let's compare the top 10 ones using the visualization tools built into SageMaker Studio.

Comparing jobs

A single SageMaker Autopilot job trains 250 jobs by default. Over time, you may end up with tens of thousands of jobs, and you may wish to compare their properties. Let's see how:

- Going to the **Experiments** tab on the left, we locate our job and right-click **Open in trial component list**, as can be seen in the following screenshot:

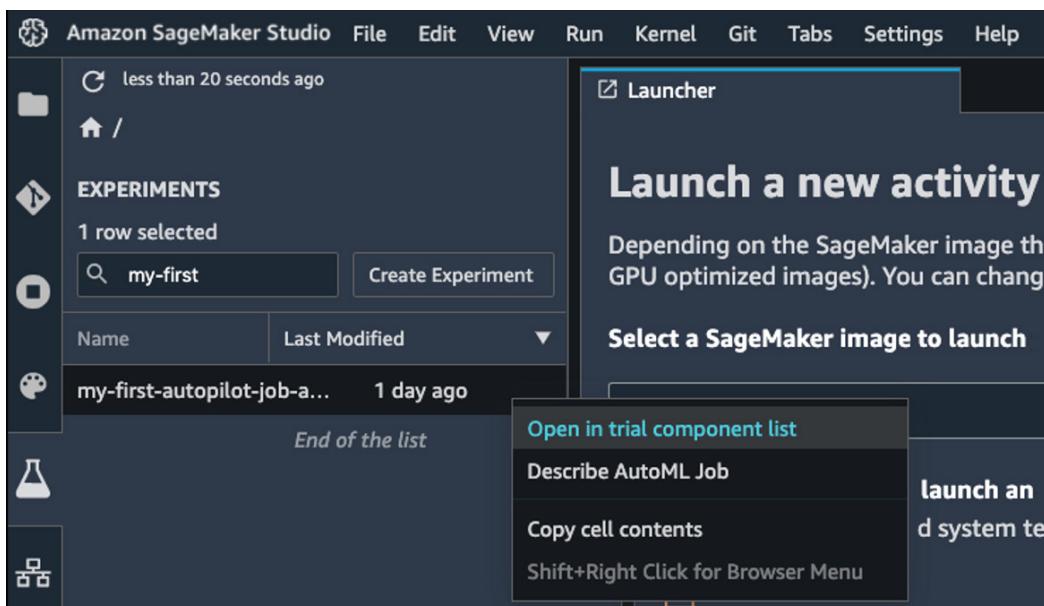


Figure 3.13 – Opening the list of trials

2. This opens **Trial Component List**, as shown in the following screenshot.

We open the **Table Properties** panel on the right by clicking on the icon representing a cog, and we untick everything except **Experiment name**, **Trial component name**, and **ObjectiveMetric**. In the main panel, we sort jobs by descending objective metrics by clicking on the arrow. We hold down the *Shift* key and click the top 10 jobs to select them, as shown in the following screenshot:

The screenshot shows the Amazon SageMaker Studio interface. On the left, there's a sidebar with 'Components and registries' and 'Experiments and trials'. Under 'Experiments', it says '1 row selected 0/20 filters'. A search bar is present. Below this is a table with columns: 'Experiment name', 'Trial component name', and 'ObjectiveMetric'. The table lists 10 rows of tuning jobs. To the right of the main area is the 'TABLE PROPERTIES' panel. It has a radio button for 'Auto refresh disabled' (which is selected). Below that is a 'Column group presets' section with a 'Toggle visibility for columns based on column type' button. Underneath are several checkboxes for different columns: 'Summary' (selected), 'Status', 'Experiment name' (selected), 'Trial name', 'Trial component name' (selected), 'Trial component type', 'Created', 'Last modified', 'Created by', 'Tags', 'Debugger status', 'Metrics' (selected), 'ObjectiveMetric' (selected), and 'train:f1' (unchecked).

Experiment name	Trial component name	ObjectiveMetric
my-first-autopilot-job...	tuning-job-1-446ba4a...	0.8031499981880188
my-first-autopilot-job...	tuning-job-1-446ba4a...	0.8027499914169312
my-first-autopilot-job...	tuning-job-1-446ba4a...	0.8026400208473206
my-first-autopilot-job...	tuning-job-1-446ba4a...	0.8024799823760986
my-first-autopilot-job...	tuning-job-1-446ba4a...	0.8022900223731995
my-first-autopilot-job...	tuning-job-1-446ba4a...	0.8016600012779236
my-first-autopilot-job...	tuning-job-1-446ba4a...	0.8014199733734131
my-first-autopilot-job...	tuning-job-1-446ba4a...	0.801219997901917
my-first-autopilot-job...	tuning-job-1-446ba4a...	0.8011699914932251
my-first-autopilot-job...	tuning-job-1-446ba4a...	0.8011400103569031
my-first-autopilot-job...	tuning-job-1-446ba4a...	0.801079988479613
my-first-autopilot-job...	tuning-job-1-446ba4a...	0.8007599711418152

Figure 3.14 – Comparing jobs

- Then, we click on the **Add chart** button. This opens a new view that can be seen in the following screenshot. Click inside the chart box at the bottom to open the **Chart properties** panel on the right.

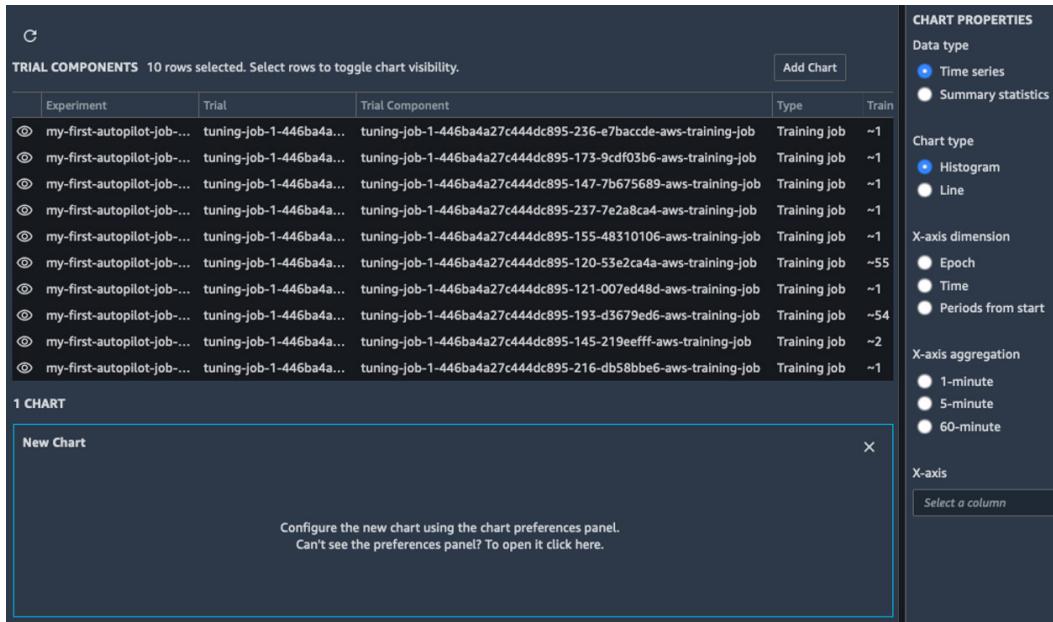


Figure 3.15 – Building a chart

As our training jobs are very short (about a minute), there won't be enough data for **Time series** charts, so let's select **Summary statistics** instead. We're going to build a **scatter plot**, putting the eta and lambda hyperparameters in perspective, as shown in the following screenshot. We also color data points with our trial names.

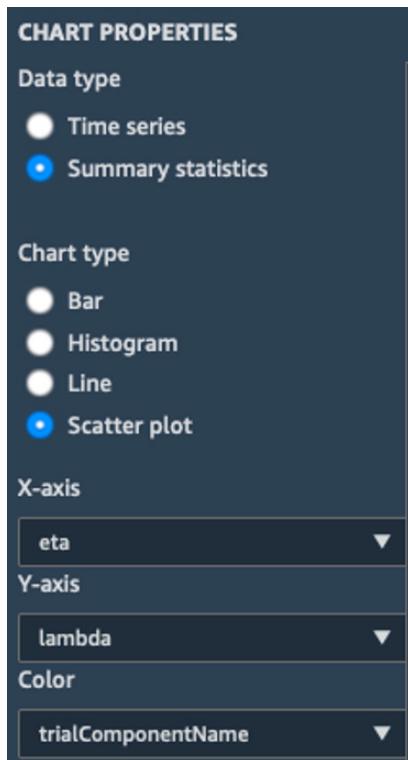


Figure 3.16 – Creating a chart

4. Zooming in on the following chart, we can quickly visualize our jobs and their respective parameters. We could build additional charts showing the impact of certain hyperparameters on accuracy. This would help us shortlist a few models for further testing. Maybe we would end up considering several of them for ensemble prediction.

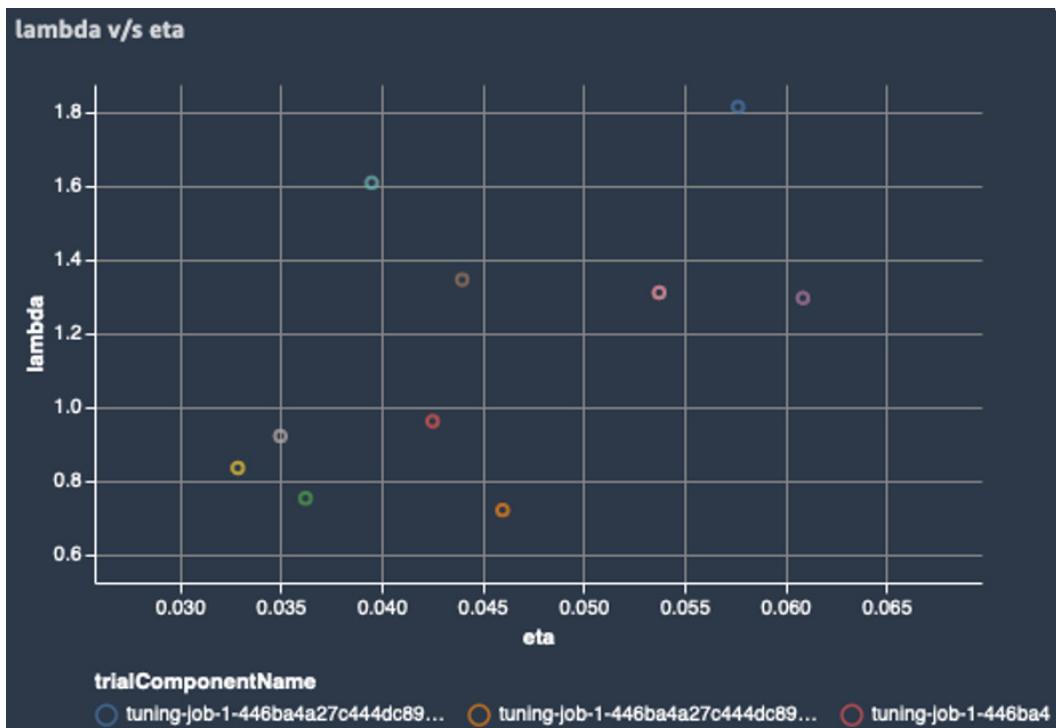


Figure 3.17 – Plotting hyperparameters

The next step is to deploy a model and start testing it.

Deploying and invoking a model

SageMaker Studio makes it extremely easy to deploy a model. Let's see how:

1. Going back to the **Experiments** tab, we right-click the name of our experiment and select **Describe AutoML Job**. This opens the list of training jobs. Making sure that they're sorted by descending objective, we select the best one (it's highlighted with a star), as shown in the screenshot that follows, and then we click on the **Deploy model** button:

EXPERIMENT: MY-FIRST-AUTOPILOT-JOB

Open candidate generation notebook ⓘ | Open data exploration notebook ⓘ

Problem type: BinaryClassification

Trials	Job profile		
TRIALS 1 row selected	Deploy model		
Trial name	Status	Start time	Objective: F1
★ Best: tuning-job-1-446ba4a27c... tuning-job-1-446ba4a27c444dc8... tuning-job-1-446ba4a27c444dc8... tuning-job-1-446ba4a27c444dc8... tuning-job-1-446ba4a27c444dc8...	Completed Completed Completed Completed Completed	58 minutes ago 1 hour ago 1 hour ago 58 minutes ago 1 hour ago	0.8031499981880188 0.8027499914169312 0.8026400208473206 0.8024799823760986 0.8022900223731995

Figure 3.18 – Deploying a model

2. Under **REALTIME DEPLOYMENT SETTINGS**, let's give the endpoint a name (`my-first-autopilot-endpoint`), leave all other settings as is, and click on **Deploy model**. As shown in the following screenshot, the model will be deployed on a real-time HTTPS endpoint backed by an `ml.m5.xlarge` instance:

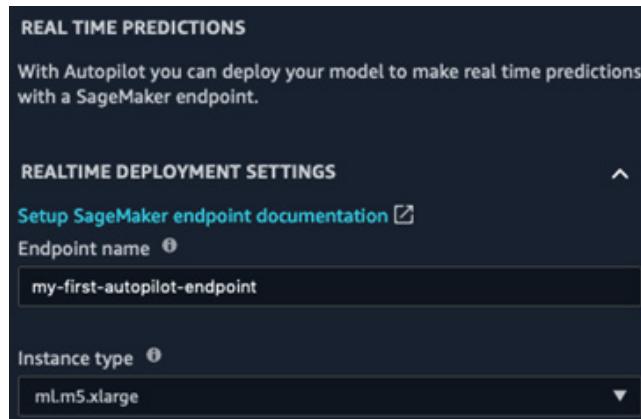


Figure 3.19 – Deploying a model

3. Heading to the **Endpoints** section in the left-hand vertical panel, we can see the endpoint being created. As shown in the following screenshot, it will initially be in the **Creating** state. After a few minutes, it's **In service**:

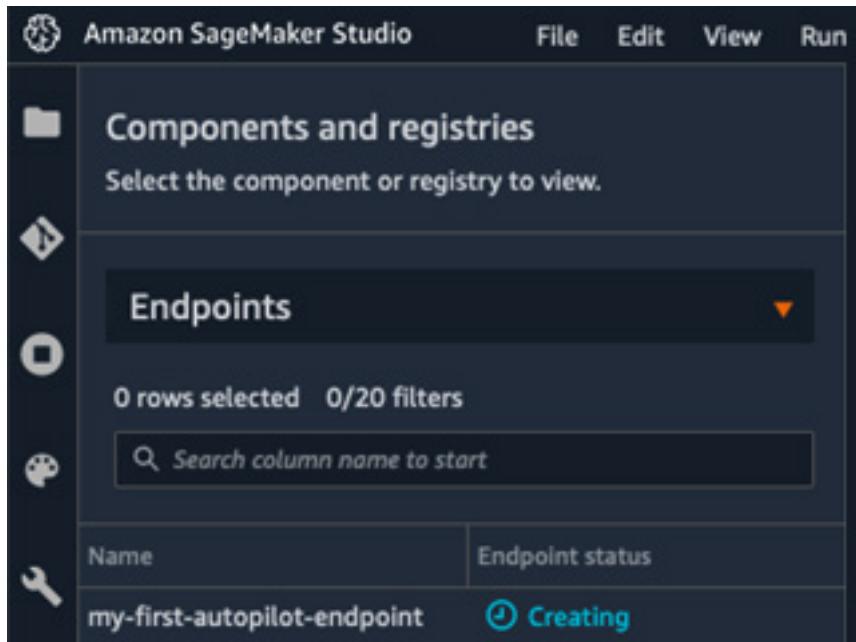


Figure 3.20 – Creating an endpoint

4. Moving to a Jupyter notebook (we can reuse the one we wrote to download the dataset), we define the name of the endpoint, and a sample to predict. Here, I'm using the first line of the dataset:

```
ep_name = 'my-first-autopilot-endpoint'  
sample = '56,housemaid,married,basic.4y,no,no,no,  
telephone,may,mon,261,1,999,0,nonexistent,1.1,93.994,  
-36.4,4.857,5191.0'
```

5. We create a boto3 client for the SageMaker runtime. This runtime contains a single API, `invoke_endpoint` (<https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/sagemaker-runtime.html>). This makes it efficient to embed in client applications that just need to invoke models:

```
import boto3  
sm_rt = boto3.Session().client('runtime.sagemaker')
```

6. We send the sample to the endpoint, also passing the input and output content types:

```
response = sm_rt.invoke_endpoint(EndpointName=ep_name,
                                  ContentType='text/csv',
                                  Accept='text/csv',
                                  Body=sample)
```

7. We decode the prediction and print it – this customer is not likely to accept the offer:

```
response = response['Body'].read().decode("utf-8")
print(response)
```

This sample is predicted as a "no":

```
no
```

8. When we're done testing the endpoint, we should delete it to avoid unnecessary charges. We can do this with the `delete_endpoint` API in `boto3` (https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/sagemaker.html#SageMaker.Client.delete_endpoint):

```
sm = boto3.Session().client('sagemaker')
sm.delete_endpoint(EndpointName=ep_name)
```

Congratulations! You've successfully built, trained, and deployed your first machine learning model on Amazon SageMaker. That was pretty simple, wasn't it? The only code we wrote was to download the dataset and to predict with our model.

Using **SageMaker Studio** is a great way to quickly experiment with a new dataset, and also to let fewer technical users build models on their own. Advanced users can also add their own custom images to SageMaker Studio, and they'll find more details at <https://docs.aws.amazon.com/sagemaker/latest/dg/studio-byoi.html>.

Now, let's see how we can use SageMaker Autopilot programmatically with the **SageMaker SDK**.

Using the SageMaker Autopilot SDK

The Amazon SageMaker SDK includes a simple API for SageMaker Autopilot. You can find its documentation at <https://sagemaker.readthedocs.io/en/stable/automl.html>.

In this section, you'll learn how to use this API to train a model on the same dataset as in the previous section.

Launching a job

The SageMaker SDK makes it extremely easy to launch an Autopilot job – just upload your data in S3, and call a single API! Let's see how:

1. First, we import the SageMaker SDK:

```
import sagemaker
sess = sagemaker.Session()
```

2. Then, we download the dataset:

```
%%sh
wget -N https://sagemaker-sample-data-us-west-2.s3-us-
west-2.amazonaws.com/autopilot/direct_marketing/bank-
additional.zip
unzip -o bank-additional.zip
```

3. Next, we upload the dataset to S3:

```
bucket = sess.default_bucket()
prefix = 'sagemaker/DEMO-automl-dm'
s3_input_data = sess.upload_data(path='./bank-additional/
bank-additional-full.csv', key_prefix=prefix+'input')
```

4. We then configure the AutoML job, which only takes one line of code. We define the **target attribute** (remember, that column is named "y"), and where to store training artifacts. Optionally, we can also set a maximum runtime for the job, a maximum runtime per job, or reduce the number of candidate models that will be tuned. Please note that restricting the job's duration too much is likely to impact its accuracy. For development purposes, this isn't a problem, so let's cap our job at one hour, or 250 tuning jobs (whichever limit it hits first):

```
from sagemaker.automl.automl import AutoML
auto_ml_job = AutoML(
    role = sagemaker.get_execution_role(),
    sagemaker_session = sess,
    target_attribute_name = 'y',
    output_path =
        's3://{{}}/{{}}/output'.format(bucket,prefix),
    max_runtime_per_training_job_in_seconds = 600,
    max_candidates = 250,
```

```
    total_job_runtime_in_seconds = 3600
)
```

5. Next, we launch the Autopilot job, passing it the location of the training set. We turn logs off (who wants to read hundreds of tuning logs?), and we set the call to non-blocking, as we'd like to query the job status in the next cells:

```
auto_ml_job.fit(inputs=s3_input_data, logs=False,
wait=False)
```

The job starts right away. Now let's see how we can monitor its status.

Monitoring a job

While the job is running, we can use the `describe_auto_ml_job()` API to monitor its progress:

1. For example, the following code will check the job's status every 60 seconds until the data analysis step completes:

```
from time import sleep
job = auto_ml_job.describe_auto_ml_job()
job_status = job['AutoMLJobStatus']
job_sec_status = job['AutoMLJobSecondaryStatus']
if job_status not in ('Stopped', 'Failed'):
    while job_status in ('InProgress') and job_sec_status in ('AnalyzingData'):
        sleep(60)
        job = auto_ml_job.describe_auto_ml_job()
        job_status = job['AutoMLJobStatus']
        job_sec_status =
            job['AutoMLJobSecondaryStatus']
        print (job_status, job_sec_status)
```

2. Once the data analysis is complete, the two autogenerated notebooks are available. We can find their location using the same API:

```
job = auto_ml_job.describe_auto_ml_job()
job_candidate_notebook = job['AutoMLJobArtifacts']
['CandidateDefinitionNotebookLocation']
job_data_notebook = job['AutoMLJobArtifacts']
['DataExplorationNotebookLocation']
```

```
print(job_candidate_notebook)
print(job_data_notebook)
```

This prints out the S3 paths for the two notebooks:

```
s3://sagemaker-us-east-2-123456789012/sagemaker/
DEMO-automl-dm/output/automl-2020-04-24-14-21-16-938/
sagemaker-automl-candidates/pr-1-a99cb56acb5945d695
c0e74afe8ffe3ddaebeafa94f394655ac973432d1/notebooks/
SageMakerAutopilotCandidateDefinitionNotebook.ipynb

s3://sagemaker-us-east-2-123456789012/sagemaker/
DEMO-automl-dm/output/automl-2020-04-24-14-21-16-938/
sagemaker-automl-candidates/pr-1-a99cb56acb5945d695
c0e74afe8ffe3ddaebeafa94f394655ac973432d1/notebooks/
SageMakerAutopilotDataExplorationNotebook.ipynb
```

3. Using the AWS CLI, we can copy the two notebooks locally. We'll take a look at them later in this chapter:

```
%%sh -s $job_candidate_notebook $job_data_notebook
aws s3 cp $1 .
aws s3 cp $2 .
```

4. While the feature engineering runs, we can wait for completion using the same code snippet as the preceding, looping while `job_sec_status` is equal to `FeatureEngineering`.
5. Once model tuning is complete, we can very easily find the best candidate:

```
job_best_candidate = auto_ml_job.best_candidate()
print(job_best_candidate['CandidateName'])
print(job_best_candidate['FinalAutoMLJobObjectiveMetric'])
```

This prints out the name of the best tuning job, along with its validation accuracy:

```
tuning-job-1-57d7f377bfe54b40b1-030-c4f27053
{'MetricName': 'validation:accuracy', 'Value':
0.9197599935531616}
```

Then, we can deploy and test the model using the SageMaker SDK. We've covered a lot of ground already, so let's save that for future chapters, where we'll revisit this example.

Cleaning up

SageMaker Autopilot creates many underlying artifacts, such as dataset splits, pre-processing scripts, pre-processed datasets, and models. If you'd like to clean up completely, the following code snippet will do that. Of course, you could also use the AWS CLI:

```
import boto3
job_outputs_prefix = '{}/output/{}'.format(prefix,
                                             job['AutoMLJobName'])
s3_bucket = boto3.resource('s3').Bucket(bucket)
s3_bucket.objects.filter(Prefix=job_outputs_prefix).delete()
```

Now that we know how to train models using both the SageMaker Studio GUI and the SageMaker SDK, let's take a look under the hood. Engineers like to understand how things really work, right?

Diving deep on SageMaker Autopilot

In this section, we're going to learn in detail how SageMaker Autopilot processes data and trains models. If this feels too advanced for now, you're welcome to skip this material. You can always revisit it later once you've gained more experience with the service.

First, let's look at the artifacts that SageMaker Autopilot produces.

The job artifacts

Listing our S3 bucket confirms the existence of many different artifacts:

```
$ aws s3 ls s3://sagemaker-us-east-2-123456789012/sagemaker/
DEMO-autopilot/output/my-first-autopilot-job/
```

We can see many new prefixes. Let's figure out what's what:

```
PRE data-processor-models/
PRE documentation/
PRE preprocessed-data/
PRE sagemaker-automl-candidates/
PRE transformed-data/
PRE tuning/
PRE validations/
```

- The preprocessed-data/tuning_data prefix contains the training and validation splits generated from the input dataset. Each split is broken down further into small CSV chunks.
- The sagemaker-automl-candidates prefix contains 10 data pre-processing scripts (dpp [0-9].py), one for each pipeline. It also contains the code to train them (trainer.py) on the input dataset, and the code to process the input dataset with each one of the 10 resulting models (sagemaker_serve.py). Last but not least, it contains the autogenerated notebooks.
- The data-processor-models prefix contains the 10 data processing models trained by the dpp scripts.
- The transformed-data prefix contains the 10 processed versions of the training and validation splits.
- The tuning prefix contains the actual models trained during the **Model Tuning** step.
- The documentation prefix contains the explainability report.

The following diagram summarizes the relationship between these artifacts:

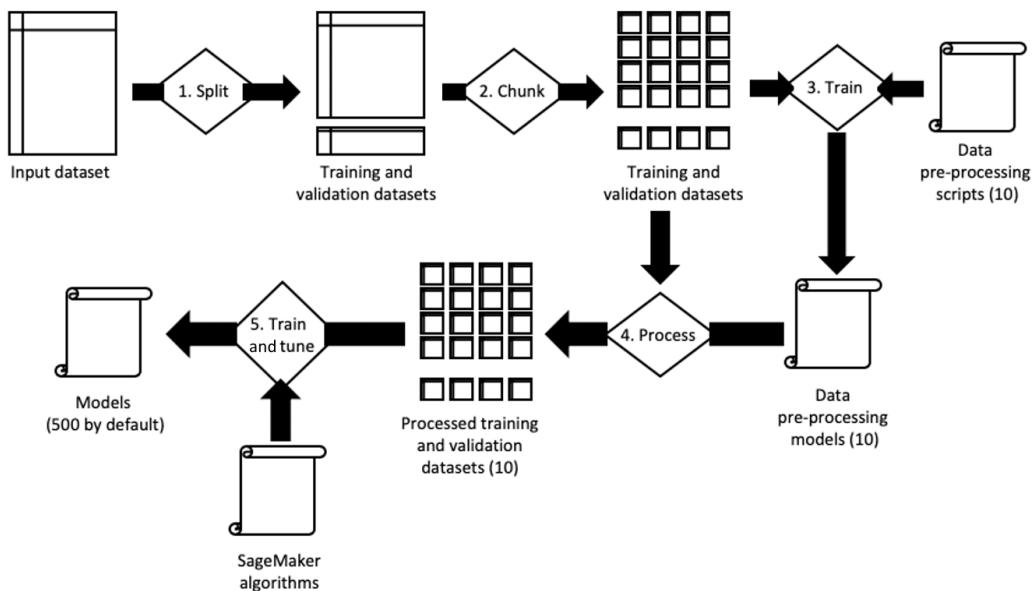


Figure 3.21 – Summing up the Autopilot process

In the next sections, we'll take a look at the two **autogenerated notebooks**, which are one of the most important features in SageMaker Autopilot.

The data exploration notebook

This notebook is available in Amazon S3 once the data analysis step is complete.

The first section, seen in the following screenshot, simply displays a sample of the dataset:

Dataset Sample																		
The following table is a random sample of 10 rows from the training dataset. For ease of presentation, we are only showing 20 of the 21 columns of the dataset.																		
💡 Suggested Action Items																		
<ul style="list-style-type: none"> Verify the input headers correctly align with the columns of the dataset sample. If they are incorrect, update the header names of your input dataset in Amazon Simple Storage Service (Amazon S3). 																		
0	45	blue-collar	married	high.school	no	no	no	telephone	jun	wed	...	4	999	0	nonexistent			
1	46	management	married	university.degree	no	yes	no	telephone	may	fri	...	5	999	0	nonexistent			
2	39	admin.	married	university.degree	no	no	no	cellular	may	tue	...	7	999	0	nonexistent			
3	25	student	single	high.school	no	yes	no	cellular	apr	wed	...	1	999	0	nonexistent			
4	38	services	divorced	high.school	no	no	no	cellular	jul	thu	...	1	999	0	nonexistent			
5	33	technician	married	university.degree	no	yes	no	telephone	may	thu	...	3	999	1	failure			
6	47	management	married	high.school	no	no	yes	cellular	apr	fri	...	1	999	1	failure			
7	48	technician	married	high.school	unknown	yes	no	telephone	may	fri	...	3	999	0	nonexistent			
8	33	services	married	university.degree	no	unknown	unknown	telephone	may	thu	...	1	999	0	nonexistent			
9	28	blue-collar	single	basic.9y	no	no	no	telephone	jun	tue	...	1	999	0	nonexistent			

Figure 3.22 – Viewing dataset statistics

Shown in the following screenshot, the second section focuses on column analysis: percentages of missing values, counts of unique values, and descriptive statistics. For instance, it appears that the pdays field has both a maximum value and a median of 999, which looks suspicious. As explained in the previous chapter, 999 is indeed a placeholder value, meaning that a customer has never been contacted before.

We found **10** of the **21** columns contained at least one numerical value. The table below shows the **10** columns which have the largest percentage of numerical values.

	% of Numerical Values	Mean	Median	Min	Max
<code>age</code>	100.0%	40.0241	38.0	17.0	98.0
<code>duration</code>	100.0%	258.285	180.0	0.0	4918.0
<code>campaign</code>	100.0%	2.56759	2.0	1.0	56.0
<code>pdays</code>	100.0%	962.475	999.0	0.0	999.0
<code>previous</code>	100.0%	0.172963	0.0	0.0	7.0
<code>emp.var.rate</code>	100.0%	0.0818855	1.1	-3.4	1.4
<code>cons.price.idx</code>	100.0%	93.5757	93.749	92.201	94.767
<code>cons.conf.idx</code>	100.0%	-40.5026	-41.8	-50.8	-26.9
<code>euribor3m</code>	100.0%	3.62129	4.857	0.634	5.045
<code>nr.employed</code>	100.0%	5167.04	5191.0	4963.6	5228.1

Figure 3.23 – Viewing dataset statistics

As you can see, this notebook saves us the trouble of computing these statistics ourselves, and we can use them to quickly check that the dataset is what we expect.

Now, let's look at the second notebook. As you will see, it's extremely insightful!

The candidate generation notebook

This notebook contains the definition of the 10 candidate pipelines, and how they're trained. This is a **runnable notebook**, and advanced practitioners can use it to replay the AutoML process, and keep refining their experiment. Please note that this is totally optional! It's perfectly OK to deploy the top model directly and start testing it.

Having said that, let's run one of the pipelines manually:

1. We open the notebook and save a read-write copy by clicking on the **Import notebook** link in the top-right corner.
2. Then, we run the cells in the **SageMaker Setup** section to import all the required artifacts and parameters.
3. Moving to the **Candidate Pipelines** section, we create a runner object that will launch jobs for selected candidate pipelines:

```
from sagemaker_automl import AutoMLInteractiveRunner,
AutoMLLocalCandidate
```

```
automl_interactive_runner =
AutoMLInteractiveRunner(AUTOML_LOCAL_RUN_CONFIG)
```

4. Then, we add the first pipeline (dpp0). The notebook tells us: "*This data transformation strategy first transforms 'numeric' features using RobustImputer (converts missing values to nan) and 'categorical' features using ThresholdOneHotEncoder. It merges all the generated features and applies RobustStandardScaler. The transformed data will be used to tune an XGBoost model*". We just need to run the following cell to add it:

```
automl_interactive_runner.select_candidate(
    {"data_transformer": {
        "name": "dpp0",
        ...
    }
}
```

If you're curious about the implementation of `RobustImputer` or `ThresholdOneHotEncoder`, hyperlinks take you to the appropriate source file in the `sagemaker_sklearn_extension` module (<https://github.com/aws/sagemaker-scikit-learn-extension/>).

This way, you can understand exactly how data has been processed. As these objects are based on scikit-learn objects, they should quickly look very familiar. For instance, we can see that `RobustImputer` is built on top of `sklearn.impute.SimpleImputer`, with added functionality. Likewise, `ThresholdOneHotEncoder` is an extension of `sklearn.preprocessing.OneHotEncoder`.

5. Taking a quick look at other pipelines, we see different processing strategies and algorithms. You should see the **Linear Learner** algorithm used in some pipelines. It's one of the **built-in algorithms** in SageMaker, and we'll cover it in the next chapter. You should also see the **mlp** algorithm, which is based on neural networks.
6. Scrolling down, we get to the **Selected Candidates** section, where we can indeed confirm that we have only selected the first pipeline:

```
automl_interactive_runner.display_candidates()
```

This is visible in the result here:

Candidate Name	Algorithm	Feature Transformer
dpp0-xgboost	xgboost	dpp0.py

Figure 3.24 – The results table

This also tells us that data will be processed by the `dpp0.py` script and that the model will be trained using the XGBoost algorithm.

- Clicking on the **dpp0** hyperlink opens the script. As expected, we see that it builds a scikit-learn transformer pipeline (not to be confused with the SageMaker pipeline composed of pre-processing and training jobs). Missing values are imputed in the numerical features, and the categorical features are one-hot encoded. Then, all features are scaled and the labels are encoded:

```
numeric_processors = Pipeline(
    steps=[('robustimputer',
            RobustImputer(strategy='constant', fill_
            values=nan))]
)
categorical_processors = Pipeline(
    steps=[('thresholdonehotencoder',
            ThresholdOneHotEncoder(threshold=301))]
)
column_transformer = ColumnTransformer(
    transformers=[
        ('numeric_processing', numeric_processors, numeric),
        ('categorical_processing', categorical_processors,
         categorical)])
)
return Pipeline(steps=[
    ('column_transformer', column_transformer),
    ('robuststandardscaler', RobustStandardScaler())])
)
```

- Back in the notebook, we launch this script in the **Run Data Transformation Steps** section:

```
automl_interactive_runner.fit_data_transformers(parallel_
jobs=7)
```

- This creates two sequential SageMaker jobs and their artifacts are stored in a new prefix created for the notebook run:

```
$ aws s3 ls s3://sagemaker-us-east-2-123456789012/
sagemaker/DEMO-autopilot/output/my-first-autopilot-job/
my-first-a-notebook-run-24-13-17-22/
```

The first job trains the dpp0 transformers on the input dataset.

The second job processes the input dataset with the resulting model. For the record, this job uses the SageMaker **Batch Transform** feature, which will be covered in a later chapter.

- Going back to SageMaker Studio, let's find out more about these two jobs. Starting from the **SageMaker components and registries** icon on the left, we select **Unassigned trial components**, and we see our two jobs there: `my-first-a-notebook-run-24-13-17-22-dpp0-train-24-13-38-38-aws-training-job` and `my-first-a-notebook-run-24-13-17-22-dpp0-transform-24-13-38-38-aws-transform-job`.
- Double-clicking a job name opens the **Open in trial details** window, as shown in the following screenshot. It tells us everything there is to know about the job: the parameters, location of artifacts, and more:

The screenshot shows the 'Trial stages' section of the SageMaker Studio interface. A specific trial stage is highlighted with a blue border:

Trial stages	Charts	Metrics	Parameters	Artifacts	AWS Settings	Debugger																				
my-first-a-notebook-run-24-13-17-22-dpp0-transform-24-13-38-38-aws-transform-job Created 1 day ago																										
<table border="1"> <thead> <tr> <th>Name</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>MAX_CONTENT_LENGTH</td> <td>20000000</td> </tr> <tr> <td>SAGEMAKER_DEFAULT_INVOCATIONS_ACCEPT</td> <td>text/csv</td> </tr> <tr> <td>SAGEMAKER_PROGRAM</td> <td>sagemaker_serve</td> </tr> <tr> <td>SAGEMAKER_SUBMIT_DIRECTORY</td> <td>/opt/ml/model/sagemaker_serve.py</td> </tr> <tr> <td>SageMaker.InstanceCount</td> <td>1</td> </tr> <tr> <td>SageMaker.InstanceType</td> <td>ml.m5.4xlarge</td> </tr> <tr> <td>SageMaker.ModelName</td> <td>my-first-a-notebook-run-24-13-17-22-dpp0-train-24-13-38-38</td> </tr> <tr> <td>SageMaker.ModelPrimary.DataSource</td> <td>s3://sagemaker-us-east-2-123456789012/sagemaker/DEMO-autopilot/o...</td> </tr> <tr> <td>SageMaker.ModelPrimary.Image</td> <td>257758044811.dkr.ecr.us-east-2.amazonaws.com/sagemaker-sklearn-aut...</td> </tr> </tbody> </table>							Name	Value	MAX_CONTENT_LENGTH	20000000	SAGEMAKER_DEFAULT_INVOCATIONS_ACCEPT	text/csv	SAGEMAKER_PROGRAM	sagemaker_serve	SAGEMAKER_SUBMIT_DIRECTORY	/opt/ml/model/sagemaker_serve.py	SageMaker.InstanceCount	1	SageMaker.InstanceType	ml.m5.4xlarge	SageMaker.ModelName	my-first-a-notebook-run-24-13-17-22-dpp0-train-24-13-38-38	SageMaker.ModelPrimary.DataSource	s3://sagemaker-us-east-2-123456789012/sagemaker/DEMO-autopilot/o...	SageMaker.ModelPrimary.Image	257758044811.dkr.ecr.us-east-2.amazonaws.com/sagemaker-sklearn-aut...
Name	Value																									
MAX_CONTENT_LENGTH	20000000																									
SAGEMAKER_DEFAULT_INVOCATIONS_ACCEPT	text/csv																									
SAGEMAKER_PROGRAM	sagemaker_serve																									
SAGEMAKER_SUBMIT_DIRECTORY	/opt/ml/model/sagemaker_serve.py																									
SageMaker.InstanceCount	1																									
SageMaker.InstanceType	ml.m5.4xlarge																									
SageMaker.ModelName	my-first-a-notebook-run-24-13-17-22-dpp0-train-24-13-38-38																									
SageMaker.ModelPrimary.DataSource	s3://sagemaker-us-east-2-123456789012/sagemaker/DEMO-autopilot/o...																									
SageMaker.ModelPrimary.Image	257758044811.dkr.ecr.us-east-2.amazonaws.com/sagemaker-sklearn-aut...																									

Figure 3.25 – Describing a trial

Once data processing is complete, the notebook proceeds with **automatic model tuning** and **model deployment**. We haven't yet discussed these topics, so let's stop there for now. I encourage you to go through the rest of the notebook once you're comfortable with them.

Summary

As you can see, Amazon SageMaker Autopilot makes it easy to build, train, and optimize machine learning models for beginners and advanced users alike.

In this chapter, you learned about the different steps of an Autopilot job, and what they mean from a machine learning perspective. You also learned how to use both the SageMaker Studio GUI and the SageMaker SDK to build a classification model with minimal coding. Then, we dived deep into the autogenerated notebooks, which give you full control and transparency over the modeling processing. In particular, you learned how to run the candidate generation notebook manually to replay all the steps involved.

In the next chapter, you will learn how to use the built-in algorithms in Amazon SageMaker to train models for a variety of machine learning problems.

4

Training Machine Learning Models

In the previous chapter, you learned how Amazon SageMaker Autopilot makes it easy to build, train, and optimize models automatically, without writing a line of machine learning code.

For problem types that are not supported by SageMaker Autopilot, the next best option is to use one of the algorithms already implemented in SageMaker and to train it on your dataset. These algorithms are referred to as **built-in algorithms**, and they cover many typical machine learning problems, from classification to time series to anomaly detection.

In this chapter, you will learn about built-in algorithms for supervised and unsupervised learning, what type of problems you can solve with them, and how to use them with the SageMaker SDK:

- Discovering the built-in algorithms in Amazon SageMaker
- Training and deploying models with built-in algorithms
- Using the SageMaker SDK with built-in algorithms
- Working with more built-in algorithms

Technical requirements

You will need an AWS account to run the examples included in this chapter. If you don't already have one, please point your browser to <https://aws.amazon.com/getting-started/> to create one. You should also familiarize yourself with the AWS Free Tier (<https://aws.amazon.com/free/>), which lets you use many AWS services for free within certain usage limits.

You will need to install and configure the AWS Command-Line Interface (CLI) for your account (<https://aws.amazon.com/cli/>).

You will need a working Python 3.x environment. Installing the Anaconda distribution (<https://www.anaconda.com/>) is not mandatory, but strongly encouraged as it includes many projects that we will need (Jupyter, pandas, numpy, and more).

Code examples included in the book are available on GitHub at <https://github.com/PacktPublishing/Learn-Amazon-SageMaker-second-edition>. You will need to install a Git client to access them (<https://git-scm.com/>).

Discovering the built-in algorithms in Amazon SageMaker

Built-in algorithms are machine learning algorithms implemented, and in some cases invented, by Amazon (<https://docs.aws.amazon.com/sagemaker/latest/dg/algos.html>). They let you quickly train and deploy your own models without writing a line of machine learning code. Indeed, since the training and prediction code is readily available, you don't have to worry about implementing it, and you can focus on the machine learning problem at hand. As usual with SageMaker, infrastructure is fully managed, saving you even more time.

In this section, you'll learn about the built-in algorithms for traditional machine learning problems. Algorithms for computer vision and natural language processing will be covered in the next two chapters.

Supervised learning

Supervised learning focuses on problems that require a labeled dataset, such as regression or classification:

- **Linear Learner** builds linear models to solve regression problems, as well as classification problems (binary or multi-class).

- **Factorization Machines** builds linear models to solve regression problems, as well as classification problems (binary or multi-class). Factorization machines are a generalization of linear models, and they're a good fit for high-dimension, sparse datasets, such as user-item interaction matrices in recommendation problems.
- **K-nearest neighbors (KNN)** builds non-parametric models for regression and classification problems.
- **XGBoost** builds models for regression, classification, and ranking problems. XGBoost is possibly the most widely used machine learning algorithm used today, and SageMaker uses the open source implementation available at <https://github.com/dmlc/xgboost>.
- **DeepAR** builds forecasting models for multivariate time series. DeepAR is an Amazon-invented algorithm based on **Recurrent Neural Networks**, and you can read more about it at <https://arxiv.org/abs/1704.04110>.
- **Object2Vec** learns low-dimension embeddings from general-purpose high-dimensional objects. Object2Vec is an algorithm invented by Amazon.
- **BlazingText** builds text classification models. This algorithm was invented by Amazon, and you can read more about it at <https://dl.acm.org/doi/10.1145/3146347.3146354>.

Unsupervised learning

Unsupervised learning doesn't require a labeled dataset, and includes problems such as clustering or anomaly detection:

- **K-means** builds clustering models. SageMaker uses a modified version of the web-scale k-means clustering algorithm (<https://www.eecs.tufts.edu/~dsculley/papers/fastkmeans.pdf>).
- **Principal Component Analysis (PCA)** builds dimensionality reduction models.
- **Random Cut Forest** builds anomaly detection models.
- **IP Insights** builds models to identify usage patterns for IPv4 addresses. This comes in handy for monitoring, cybersecurity, and so on.
- **BlazingText** computes word vectors, a very useful representation for natural language processing tasks.

We'll cover some of these algorithms in detail in the rest of this chapter.

A word about scalability

Before we dive into training and deploying models with the algorithms, you may wonder why you should use them instead of their counterparts in well-known libraries such as `scikit-learn` and R.

First, these algorithms have been implemented and tuned by Amazon teams, who are not exactly newcomers to machine learning! A lot of effort has been put into making sure that these algorithms run as fast as possible on AWS infrastructure, no matter what type of instance you use. In addition, many of these algorithms support **distributed training** out of the box, letting you split model training across a cluster of fully managed instances.

Thanks to this, benchmarks indicate that these algorithms are generally 10 times better than competing implementations. In many cases, they are also much more cost-effective. You can learn more about this at the following URLs:

- AWS Tel Aviv Summit 2018: *Speed Up Your Machine Learning Workflows with Built-In Algorithms*: <https://www.youtube.com/watch?v=IeIUr78OrE0>
- *Elastic Machine Learning Algorithms in Amazon*, Liberty et al., SIGMOD'20: SageMaker: <https://www.amazon.science/publications/elastic-machine-learning-algorithms-in-amazon-sagemaker>

Of course, these algorithms benefit from all the features present in SageMaker, as you will find out by the end of the book.

Training and deploying models with built-in algorithms

Amazon SageMaker lets you train and deploy models in many different configurations. Although it encourages best practices, it is a modular service that lets you do things your own way.

In this section, we'll first look at a typical end-to-end workflow, where we use SageMaker from data upload all the way to model deployment. Then, we'll discuss alternative workflows, and how you can cherry-pick the features that you need. Finally, we will take a look under the hood, and see what happens from an infrastructure perspective when we train and deploy.

Understanding the end-to-end workflow

Let's look at a typical SageMaker workflow. You'll see it again and again in our examples, as well as in the AWS notebooks available on GitHub (<https://github.com/awslabs/amazon-sagemaker-examples/>):

1. **Make your dataset available in Amazon S3:** In most examples, we'll download a dataset from the internet, or load a local copy. However, in real life, your raw dataset would probably already be in S3, and you would prepare it using one of the services discussed in *Chapter 2, Handling Data Preparation Techniques*: splitting it for training and validation, engineering features, and so on. In any case, the dataset must be in a format that the algorithm understands, such as CSV and protobuf (<https://developers.google.com/protocol-buffers>).
2. **Configure the training job:** This is where you select the algorithm that you want to train with, set hyperparameters, and define infrastructure requirements for the training job.
3. **Launch the training job:** This is where we pass the location of your dataset in S3. Training takes place on managed infrastructure, created and provisioned automatically according to your requirements. Once training is complete, the **model artifact** is saved in S3. The training infrastructure is terminated automatically, and you only pay for what you used.
4. **Deploy the model:** You can deploy a model either on a **real-time HTTPS endpoint** for live prediction or for **batch transform**. Again, you simply need to define infrastructure requirements.
5. **Predict data:** Either invoking a real-time endpoint or a batch transformer. As you would expect, infrastructure is managed here too. For production, you would also monitor the quality of data and predictions.
6. **Clean up!**: This involves taking the endpoint down, to avoid unnecessary charges.

Understanding this workflow is critical in being productive with Amazon SageMaker. Fortunately, the SageMaker SDK has simple APIs that closely match these steps, so you shouldn't be confused about which one to use, or when to use it.

Before we start looking at the SDK, let's consider alternative workflows that could make sense in your business and technical environments.

Using alternative workflows

Amazon SageMaker is a modular service that lets you work your way. Let's first consider a workflow where you would train on SageMaker and deploy on your own server, whatever the reasons may be.

Exporting a model

Steps 1-3 would be the same as in the previous example, and then you would do the following:

1. Download the training artifact from S3, which is materialized as a `model.tar.gz` file.
2. Extract the model stored in the artifact.
3. On your own server, load the model with the appropriate machine learning library:
 - **For XGBoost models:** Use one of the implementations available at <https://xgboost.ai/>.
 - **For BlazingText models:** Use the `fastText` implementation available at <https://fasttext.cc/>.
 - **For all other models:** Use **Apache MXNet** (<https://mxnet.apache.org/>).

Now, let's see how you could import an existing model and deploy it on SageMaker.

Importing a model

The steps are equally simple:

1. Package your model in a model artifact (`model.tar.gz`).
2. Upload the artifact to an S3 bucket.
3. Register the artifact as a SageMaker model.
4. Deploy the model and predict.

This is just a quick look. We'll run full examples for both workflows in *Chapter 11, Deploying Machine Learning Models*.

Using fully managed infrastructure

All SageMaker jobs run on managed infrastructure. Let's take a look under the hood and see what happens when we train and deploy models.

Packaging algorithms in Docker containers

All SageMaker algorithms must be packaged in **Docker** containers. Don't worry, you don't need to know much about Docker in order to use SageMaker. If you're not familiar with it, I would recommend going through this tutorial to understand key concepts and tools: <https://docs.docker.com/get-started/>. It's always good to know a little more than actually required!

As you would expect, built-in algorithms are pre-packaged, and containers are readily available for training and deployment. They are hosted in **Amazon Elastic Container Registry (ECR)**, AWS' Docker registry service (<https://aws.amazon.com/ecr/>). As ECR is a region-based service, you will find a collection of containers in each region where SageMaker is available.

You can find the list of built-in algorithm containers at <https://docs.aws.amazon.com/sagemaker/latest/dg/sagemaker-algo-docker-registry-paths.html>. For instance, the name of the container for the Linear Learner algorithm in the eu-west-1 region is `438346466558.dkr.ecr.eu-west-1.amazonaws.com/linear-learner:latest`. These containers can only be pulled to SageMaker managed instances, so you won't be able to run them on your local machine.

Now let's look at the underlying infrastructure.

Creating the training infrastructure

When you launch a training job, SageMaker fires up infrastructure according to your requirements (instance type and instance count).

Once a training instance is in service, it pulls the appropriate training container from ECR. Hyperparameters are applied to the algorithm, which also receives the location of your dataset. By default, the algorithm then copies the full dataset from S3 and starts training. If distributed training is configured, SageMaker automatically distributes dataset batches to the different instances in the cluster.

Once training is complete, the model is packaged in a model artifact saved in S3. Then, the training infrastructure is shut down automatically. Logs are available in **Amazon CloudWatch Logs**. Last but not least, you're only charged for the exact amount of training time.

Creating the prediction infrastructure

When you launch a deployment job, SageMaker once again creates infrastructure according to your requirements.

Let's focus on real-time endpoints for now, and not on batch transform.

Once an endpoint instance is in service, it pulls the appropriate prediction container from ECR and loads your model from S3. Then, the HTTPS endpoint is provisioned and is ready for prediction within minutes.

If you configured the endpoint with several instances, load balancing and high availability are set up automatically. If you configured **Auto Scaling**, this is applied as well.

As you would expect, an endpoint stays up until it's deleted explicitly, either in the AWS Console or with a SageMaker API call. In the meantime, you will be charged for the endpoint, so **please make sure to delete endpoints that you don't need!**

Now that we understand the big picture, let's start looking at the SageMaker SDK, and how we can use it to train and deploy models.

Using the SageMaker SDK with built-in algorithms

Being familiar with the SageMaker SDK is important to making the most of SageMaker. You can find its documentation at <https://sagemaker.readthedocs.io>.

Walking through a simple example is the best way to get started. In this section, we'll use the Linear Learner algorithm to train a regression model on the Boston Housing dataset (<https://www.kaggle.com/c/boston-housing>). We'll proceed very slowly, leaving no stone unturned. Once again, these concepts are essential, so please take your time, and make sure you understand every step fully.

Reminder

I recommend that you follow along and run the code available in the companion GitHub repository. Every effort has been made to check all code samples present in the text. However, for those of you who have an electronic version, copying and pasting may have unpredictable results: formatting issues, weird quotes, and so on.

Preparing data

Built-in algorithms expect the dataset to be in a certain format, such as **CSV**, **protobuf**, or **libsvm**. Supported formats are listed in the algorithm documentation. For instance, Linear Learner supports CSV and RecordIO-wrapped protobuf (https://docs.aws.amazon.com/sagemaker/latest/dg/linear-learner.html#ll-input_output).

Our input dataset is already in the repository in CSV format, so let's use that. The dataset preparation will be extremely simple, and we'll run it manually:

1. Using pandas, we load the CSV dataset with pandas:

```
import pandas as pd
dataset = pd.read_csv('housing.csv')
```

2. Then, we print the shape of the dataset:

```
print(dataset.shape)
```

It contains 506 samples and 13 columns:

```
(506, 13)
```

3. Now, we display the first 5 lines of the dataset:

```
dataset[:5]
```

This prints out the table visible in the following figure. For each house, we see 12 features, and a target attribute (medv) set to the median value of the house in thousands of dollars:

	crim	zn	indus	chas	nox	age	rm	dis	rad	tax	ptratio	bt	lstat	medv
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222.0	18.7	396.90	5.33	36.2

Figure 4.1 – Viewing the dataset

4. Reading the algorithm documentation (<https://docs.aws.amazon.com/sagemaker/latest/dg/cdf-training.html>), we see that *Amazon SageMaker requires that a CSV file doesn't have a header record and that the target variable is in the first column*. Accordingly, we move the medv column to the front of the dataframe:

```
dataset = pd.concat([dataset['medv'],
                     dataset.drop(['medv'], axis=1)],
                     axis=1)
```

5. A bit of `scikit-learn` magic helps split the dataframe up into two parts – 90% for training, and 10% for validation:

```
from sklearn.model_selection import train_test_split
training_dataset, validation_dataset =
    train_test_split(dataset, test_size=0.1)
```

6. We save these two splits to individual CSV files, without either an index or a header:

```
training_dataset.to_csv('training_dataset.csv',
                       index=False, header=False)
validation_dataset.to_csv('validation_dataset.csv',
                         index=False, header=False)
```

7. We now need to upload these two files to S3. We could use any bucket, and here we'll use the default bucket conveniently created by SageMaker in the region we're running in. We can find its name with the `sagemaker.Session.default_bucket()` API:

```
import sagemaker
sess = sagemaker.Session()
bucket = sess.default_bucket()
```

8. Finally, we use the `sagemaker.Session.upload_data()` API to upload the two CSV files to the default bucket. Here, the training and validation datasets are made of a single file each, but we could upload multiple files if needed. For this reason, **we must upload the datasets under different S3 prefixes**, so that their files won't be mixed up:

```
prefix = 'boston-housing'
training_data_path = sess.upload_data(
    path='training_dataset.csv',
    key_prefix=prefix + '/input/training')
validation_data_path = sess.upload_data(
    path='validation_dataset.csv',
    key_prefix=prefix + '/input/validation')
print(training_data_path)
print(validation_data_path)
```

The two S3 paths look like this. Of course, the account number in the default bucket name will be different:

```
s3://sagemaker-eu-west-1-123456789012/boston-housing/  
input/training/training_dataset.csv  
s3://sagemaker-eu-west-1-123456789012/boston-housing/  
input/validation/validation_dataset.csv
```

Now that data is ready in S3, we can configure the training job.

Configuring a training job

The `Estimator` object (`sagemaker.estimator.Estimator`) is the cornerstone of model training. It lets you select the appropriate algorithm, define your training infrastructure requirements, and more.

The SageMaker SDK also includes algorithm-specific estimators, such as `sagemaker.LinearLearner` or `sagemaker.PCA`. I generally find them less flexible than the generic estimator (no CSV support, for one thing), and I don't recommend using them. Using the `Estimator` object also lets you reuse your code across examples, as we will see in the next sections:

1. Earlier in this chapter, we learned that SageMaker algorithms are packaged in Docker containers. Using `boto3` and the `image_uris.retrieve()` API, we can easily find the name of the Linear Learner algorithm in the region we're running:

```
from sagemaker import get_execution_role  
from sagemaker.image_uris import retrieve  
region = sess.boto_session.region_name  
container = retrieve('linear-learner', region)
```

2. Now that we know the name of the container, we can configure our training job with the `Estimator` object. In addition to the container name, we also pass the IAM role that SageMaker instances will use, the instance type and instance count to use for training, as well as the output location for the model. `Estimator` will generate a training job automatically, and we could also set our own prefix with the `base_job_name` parameter:

```
from sagemaker.estimator import Estimator  
ll_estimator = Estimator(  
    container,
```

```
role=sagemaker.get_execution_role(),
instance_count=1,
instance_type='ml.m5.large',
output_path='s3://{{}}/{{}}/output'.format(bucket,
prefix))
```

SageMaker supports plenty of different instance types, with some differences across AWS regions. You can find the full list at <https://docs.aws.amazon.com/sagemaker/latest/dg/instance-types-az.html>.

Which one should we use here? Looking at the Linear Learner documentation (<https://docs.aws.amazon.com/sagemaker/latest/dg/linear-learner.html#ll-instances>), we see that *you can train the Linear Learner algorithm on single- or multi-machine CPU and GPU instances*. Here, we're working with a tiny dataset, so let's select the smallest training instance available in our region: `ml.m5.large`.

Checking the pricing page (<https://aws.amazon.com/sagemaker/pricing/>), we see that this instance costs \$0.128 per hour in the `eu-west-1` region (the one I'm using for this job).

3. Next, we have to set **hyperparameters**. This step is possibly one of the most obscure and most difficult parts of any machine learning project. Here's my tried and tested advice: read the algorithm documentation, stick to mandatory parameters only unless you really know what you're doing, and quickly check optional parameters for default values that could clash with your dataset. In *Chapter 10, Advanced Training Techniques*, we'll see how to solve hyperparameter selection with **Automatic Model Tuning**.

Let's look at the documentation and see which hyperparameters are mandatory (https://docs.aws.amazon.com/sagemaker/latest/dg/ll_hyperparameters.html). As it turns out, there is only one: `predictor_type`. It defines the type of problem that Linear Learner is training on (regression, binary classification, or multiclass classification).

Taking a deeper look, we see that the default value for `mini_batch_size` is 1000: this isn't going to work well with our 506-sample dataset, so let's set it to 32. We also learn that the `normalize_data` parameter is set to true by default, which makes it unnecessary to normalize data ourselves:

```
ll_estimator.set_hyperparameters(
    predictor_type='regressor',
    mini_batch_size=32)
```

- Now, let's define the data channels: a channel is a named source of data passed to a SageMaker estimator. All built-in algorithms need at least a training channel, and many also accept additional channels for validation and testing. Here, we have two channels, which both provide data in CSV format. The `TrainingInput()` API lets us define their location, their format, whether they are compressed, and so on:

```
from sagemaker import TrainingInput
training_data_channel = TrainingInput(
    s3_data=training_data_path,
    content_type='text/csv')
validation_data_channel = TrainingInput(
    s3_data=validation_data_path,
    content_type='text/csv')
```

By default, data served by a channel will be fully copied to each training instance, which is fine for small datasets. We'll study alternatives in *Chapter 10, Advanced Training Techniques*.

Everything is now ready for training, so let's launch our job.

Launching a training job

All it takes is one line of code:

- We simply pass a Python dictionary containing the two channels to the `fit()` API:

```
ll_estimator.fit(
    {'train': training_data_channel,
     'validation': validation_data_channel})
```

Immediately, the training job starts:

```
Starting - Starting the training job.
```

- As soon as the job is launched, it appears in the **SageMaker components and registries | Experiments and trials** panel. There, you can see all job metadata: the location of the dataset, hyperparameters, and more.

3. The training log is visible in the notebook, and it's also stored in Amazon CloudWatch Logs, under the /aws/sagemaker/TrainingJobs prefix. Here are the first few lines, showing the infrastructure being provisioned, as explained earlier, in the *Using fully managed infrastructure* section:

```
Starting - Starting the training job...
Starting - Launching requested ML instances.....
Starting - Preparing the instances for training...
Downloading - Downloading input data...
Training - Training image download completed.
```

4. At the end of the training log, we see information on the **mean square error (MSE)** and loss metrics:

```
#quality_metric: host=algo-1, validation mse
<loss>=13.7226685169
#quality_metric: host=algo-1, validation absolute_loss
<loss>=2.86944983987
```

5. Once training is complete, the model is copied automatically to S3, and SageMaker tells us how long the job took:

```
Uploading - Uploading generated training model
Completed - Training job completed
Training seconds: 49
Billable seconds: 49
```

We mentioned earlier that the cost of an ml.m5.large instance is \$0.128 per hour. As we trained for 49 seconds, this job cost us $(49/3600)*0.128 = \$0.00174$ – less than a fifth of a penny. Any time spent setting up infrastructure ourselves would have certainly cost more!

6. Looking at the output location in our S3 bucket, we see the model artifact:

```
%%bash -s "$l1_estimator.output_path"
aws s3 ls --recursive $1
```

You should see the model artifact `model.tar.gz`.

We'll see in *Chapter 11, Deploying Machine Learning Models*, what's inside that artifact, and how to deploy the model outside of SageMaker. For now, let's deploy it to a real-time endpoint.

Deploying a model

This is my favorite part of SageMaker; we only need one line of code to deploy a model to an **HTTPS endpoint**:

1. It's good practice to create identifiable and unique endpoint names. We could also let SageMaker create one for us during deployment:

```
from time import strftime, gmtime
timestamp = strftime('%d-%H-%M-%S', gmtime())
endpoint_name = 'linear-learner-demo-' + timestamp
print(endpoint_name)
```

Here, the endpoint name is `linear-learner-demo-29-08-37-25`.

2. We deploy the model using the `deploy()` API. As this is a test endpoint, we use the smallest endpoint instance available, `ml.t2.medium`. In the `eu-west-1` region, this will only cost us \$0.07 per hour:

```
ll_predictor = ll_estimator.deploy(
    endpoint_name=endpoint_name,
    initial_instance_count=1,
    instance_type='ml.t2.medium')
```

When the endpoint is created, we can see it in the **SageMaker components and registries | Endpoints** panel in SageMaker Studio.

3. A few minutes later, the endpoint is in service. We can use the `predict()` API to send it a CSV sample for prediction. We set serialization using built-in functions:

```
ll_predictor.serializer =
    sagemaker.serializers.CSVSerializer()
ll_predictor.deserializer =
    sagemaker.deserializers.CSVDeserializer()
test_sample = '0.00632,18.00,2.310,0,0.5380,6.5750,65.20,
4.0900,1,296.0,15.30,4.98'
response = ll_predictor.predict(test_sample)
print(response)
```

The prediction output tells us that this house should cost \$30,173:

```
[[30.17342185974121]]
```

We can also predict multiple samples at a time:

```
test_samples = [
    '0.00632,18.00,2.310,0,0.5380,6.5750,65.20,4.0900,1,296.0
    ,15.30,4.98',
    '0.02731,0.00,7.070,0,0.4690,6.4210,78.90,4.9671,2,242.0,
    17.80,9.14']
response = ll_predictor.predict(test_samples)
print(response)
```

Now the prediction output is as follows:

```
[['30.413358688354492'], ['24.884408950805664']]
```

When we're done working with the endpoint, **we shouldn't forget to delete it to avoid unnecessary charges.**

Cleaning up

Deleting an endpoint is as simple as calling the `delete_endpoint()` API:

```
ll_predictor.delete_endpoint()
```

At the risk of repeating myself, the topics covered in this section are extremely important, so please make sure you're completely familiar with them, as we'll constantly use them in the rest of the book. Please spend some time reading the service and SDK documentation as well:

- <https://docs.aws.amazon.com/sagemaker/latest/dg/algos.html>
- <https://sagemaker.readthedocs.io>

Now let's explore other built-in algorithms. You'll see that the workflow and the code are very similar!

Working with more built-in algorithms

In the rest of this chapter, we will run more examples with built-in algorithms, both in supervised and unsupervised mode. This will help you become very familiar with the SageMaker SDK and learn how to solve actual machine learning problems. The following list shows some of these algorithms:

- Classification with XGBoost
- Recommendation with Factorization Machines

- Dimensionality reduction with PCA
- Anomaly detection with Random Cut Forest

Regression with XGBoost

Let's train a model on the Boston Housing dataset with the **XGBoost** algorithm (<https://github.com/dmlc/xgboost>). As we will see in *Chapter 7, Extending Machine Learning Services Using Built-In Frameworks*, SageMaker also supports XGBoost scripts:

1. We reuse the dataset preparation steps from the previous examples.
2. We find the name of the XGBoost container. As several versions are supported, we select the latest one (1.3.1 at the time of writing):

```
from sagemaker import image_uris
region = sess.boto_session.region_name
container = image_uris.retrieve('xgboost', region,
                                version='latest')
```

3. We configure the `Estimator` function. The code is strictly identical to the code used with `LinearLearner`:

```
xgb_estimator = Estimator(
    container,
    role=sagemaker.get_execution_role(),
    instance_count=1,
    instance_type='ml.m5.large',
    output_path='s3://{{}}/{{}}/output'.format(bucket,
                                                prefix))
```

4. Taking a look at the hyperparameters (https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost_hyperparameters.html), we see that the only required one is `num_round`. As it's not obvious which value to set, we'll go for a large value, and we'll also define the `early_stopping_rounds` parameter in order to avoid overfitting. Of course, we need to set the objective for a regression problem:

```
xgb_estimator.set_hyperparameters(
    objective='reg:linear',
    num_round=200,
    early_stopping_rounds=10)
```

5. We define the training input, just like in the previous example:

```
from sagemaker import TrainingInput
training_data_channel = TrainingInput(
    s3_data=training_data_path,
    content_type='text/csv')
validation_data_channel = TrainingInput(
    s3_data=validation_data_path,
    content_type='text/csv')
```

6. We then launch the training job:

```
xgb_estimator.fit(
    {'train': training_data_channel,
     'validation': validation_data_channel})
```

7. The job only ran for 22 rounds, meaning that **early stopping** was triggered. Looking at the training log, we see that round #12 was actually the best one, with a **root mean square error (RMSE)** of 2.43126:

```
[12]#011train-rmse:1.25702#011validation-rmse:2.43126
<output removed>
[22]#011train-rmse:0.722193#011validation-rmse:2.43355
```

8. Deploying still takes one line of code:

```
from time import strftime, gmtime
timestamp = strftime('%d-%H-%M-%S', gmtime())
endpoint_name = 'xgb-demo'+'-' + timestamp
xgb_predictor = xgb_estimator.deploy(
    endpoint_name=endpoint_name,
    initial_instance_count=1,
    instance_type='ml.t2.medium')
```

9. Once the model is deployed, we use the `predict()` API again to send it a CSV sample:

```
test_sample = '0.00632,18.00,2.310,0,0.5380,6.5750,65.20,
4.0900,1,296.0,15.30,4.98'

xgb_predictor.serializer =
```

```
sagemaker.serializers.CSVSerializer()
xgb_predictor.deserializer =
    sagemaker.deserializers.CSVDeserializer()
response = xgb_predictor.predict(test_sample)
print(response)
```

The result tells us that this house should cost \$23,754:

```
[[23.73023223876953]]
```

10. Finally, we delete the endpoint when we're done:

```
xgb_predictor.delete_endpoint()
```

As you can see, the SageMaker workflow is pretty simple and makes it easy to experiment quickly with different algorithms without having to rewrite all your code.

Let's move on to the Factorization Machines algorithm. In the process, we will learn about the highly efficient RecordIO-wrapped protobuf format.

Recommendation with Factorization Machines

Factorization Machines is a generalization of linear models (<https://www.csie.ntu.edu.tw/~b97053/paper/Rendle2010FM.pdf>). It's well-suited for high-dimension sparse datasets, such as user-item interaction matrices for recommendation.

In this example, we're going to train a recommendation model based on the **MovieLens** dataset (<https://grouplens.org/datasets/movielens/>).

The dataset exists in several versions. To minimize training times, we'll use the 100k version. It contains 100,000 ratings (integer values from 1 to 5) assigned by 943 users to 1,682 movies. The dataset is already split for training and validation.

As you know by now, training and deploying with SageMaker is very simple. Most of the code will be identical to the two previous examples, which is great! This lets us focus on understanding and preparing data.

Understanding sparse datasets

Imagine building a matrix to store this dataset. It would have 943 lines (one per user) and 1,682 columns (one per movie). Cells would store the ratings. The following diagram shows a basic example:

	Movie 1	Movie 2	Movie 3	Movie 4	Movie 5	Movie 6
User 1		2		4	5	
User 2	3	1	4			4
User 3		2	3			
User 4		2			5	

Figure 4.2 – Sparse matrix

Hence, the matrix would have $943 \times 1,682 = 1,586,126$ cells. However, as only 100,000 ratings are present, 93.69% of cells would be empty. Storing our dataset this way would be extremely inefficient. It would needlessly consume RAM, storage, and network bandwidth to store and transfer lots of zero values!

In fact, things are much worse, as the algorithm expects the input dataset to look like in the following diagram:

User 1	User 2	User 3	User 4	Movie 1	Movie 2	Movie 3	Movie 4	Movie 5	Movie 6	Ratings
1					1					2
1								1		4
1								1		5
	1			1						3
	1				1					1
	1					1				4
	1								1	4
		1			1					2
		1				1				3
			1		1					2
			1					1		5

Figure 4.3 – Sparse matrix

Why do we need to store data this way? The answer is simple: Factorization Machines is a **supervised learning** algorithm, so we need to train it on labeled samples.

Looking at the preceding diagram, we see that each line represents a movie review. The matrix on the left stores its one-hot encoded features (users and movies), and the vector on the right stores its label. For instance, the last line tells us that user 4 has given movie 5 a "5" rating.

The size of this matrix is 100,000 lines by 2,625 columns (943 movies plus 1,682 movies). The total number of cells is 262,500,000, which are only 0.076% full (200,000 / 262,500,000). If we used a 32-bit value for each cell, we would need almost a gigabyte of memory to store this matrix. This is horribly inefficient but still manageable.

Just for fun, let's do the same exercise for the largest version of MovieLens, which has 25 million ratings, 62,000 movies, and 162,000 users. The matrix would have 25 million lines and 224,000 columns, for a total of 5,600,000,000,000 cells. Yes, that's 5.6 trillion cells, and although they would be 99.999% empty, we would still need over 20 terabytes of RAM to store them. Ouch. If that's not bad enough, consider recommendation models with millions of users and products: the numbers are mind-boggling!

Instead of using a plain matrix, we'll use a **sparse matrix**, a data structure specifically designed and optimized for sparse datasets. SciPy has exactly the object we need, named `lil_matrix` (https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.lil_matrix.html). This will help us to get rid of all these nasty zeros.

Understanding protobuf and RecordIO

So how will we pass this sparse matrix to the SageMaker algorithm? As you would expect, we're going to serialize the object and store it in S3. We're not going to use Python serialization, however. Instead, we're going to use **protobuf** (<https://developers.google.com/protocol-buffers/>), a popular and efficient serialization mechanism.

In addition, we're going to store the protobuf-encoded data in a record format called **RecordIO** (<https://mxnet.apache.org/api/faq/recordio/>). Our dataset will be stored as a sequence of records in a single file. This has the following benefits:

- A single file is easier to move around: who wants to deal with thousands of individual files that can get lost or corrupted?
- A sequential file is faster to read, which makes the training process more efficient.
- A sequence of records is easy to split for distributed training.

Don't worry if you're not familiar with protobuf and RecordIO. The SageMaker SDK includes utility functions that hide their complexity.

Building a Factorization Machines model on MovieLens

We will begin building the model using the following steps:

1. In a Jupyter notebook, we first download and extract the MovieLens dataset:

```
%%sh
wget http://files.grouplens.org/datasets/movielens/
ml-100k.zip
unzip ml-100k.zip
```

2. As the dataset is ordered by user ID, we shuffle it as a precaution. Then, we take a look at the first few lines:

```
%cd ml-100k
!shuf ua.base -o ua.base.shuffled
!head -5 ua.base.shuffled
```

We see four columns: the user ID, the movie ID, the rating, and a timestamp (which we'll ignore in our model):

378	43	3	880056609
919	558	5	875372988
90	285	5	891383687
249	245	2	879571999
416	64	5	893212929

3. We define sizing constants:

```
num_users = 943
num_movies = 1682
num_features = num_users+num_movies
num_ratings_train = 90570
num_ratings_test = 9430
```

4. Now, let's write a function to load a dataset into a sparse matrix. Based on the previous explanation, we go through the dataset line by line. In the X matrix, we set the appropriate user and movie columns to 1. We also store the rating in the Y vector:

```
import csv
import numpy as np
from scipy.sparse import lil_matrix
```

```

def loadDataset(filename, lines, columns):
    X = lil_matrix((lines, columns)).astype('float32')
    Y = []
    line=0
    with open(filename,'r') as f:
        samples=csv.reader(f,delimiter='\t')
        for userId,movieId,rating,timestamp in samples:
            X[line,int(userId)-1] = 1
            X[line,int(num_users)+int(movieId)-1] = 1
            Y.append(int(rating))
            line=line+1
    Y=np.array(Y).astype('float32')
    return X,Y

```

5. We then process the training and test datasets:

```

X_train, Y_train = loadDataset('ua.base.shuffled',
                               num_ratings_train,
                               num_features)
X_test, Y_test = loadDataset('ua.test',
                            num_ratings_test,
                            num_features)

```

6. We check that the shapes are what we expect:

```

print(X_train.shape)
print(Y_train.shape)
print(X_test.shape)
print(Y_test.shape)

```

This displays the dataset shapes:

```

(90570, 2625)
(90570,)
(9430, 2625)
(9430,)

```

- Now, let's write a function that converts a dataset to RecordIO-wrapped protobuf, and uploads it to an S3 bucket. We first create an in-memory binary stream with `io.BytesIO()`. Then, we use the lifesaving `write_spmatrix_to_sparse_tensor()` function to write the sample matrix and the label vector to that buffer in protobuf format. Finally, we use `boto3` to upload the buffer to S3:

```
import io, boto3
import sagemaker.amazon.common as smac
def writeDatasetToProtobuf(X, Y, bucket, prefix, key):
    buf = io.BytesIO()
    smac.write_spmatrix_to_sparse_tensor(buf, X, Y)
    buf.seek(0)
    obj = '{} / {}'.format(prefix, key)

    boto3.resource('s3').Bucket(bucket).Object(obj).
        upload_fileobj(buf)
    return 's3://{} / {}'.format(bucket, obj)
```

Had our data been stored in a `numpy` array instead of `lilmatrix`, we would have used the `write_numpy_to_dense_tensor()` function instead. It has the same effect.

8. We apply this function to both datasets, and we store their S3 paths:

9. Taking a look at the S3 bucket in a terminal, we see that the training dataset only takes 5.5 MB. The combination of sparse matrix, protobuf, and RecordIO has paid off:

```
$ aws s3 ls s3://sagemaker-eu-west-1-123456789012/
fm-movielens/train/train.protobuf
5796480 train.protobuf
```

10. What comes next is SageMaker business as usual. We find the name of the Factorization Machines container, configure the `Estimator` function, and set the hyperparameters:

```
from sagemaker.image_uris import retrieve
region = sess.boto_session.region_name

container=retrieve('factorization-machines', region)
fm=sagemaker.estimator.Estimator(
    container,
    role=sagemaker.get_execution_role(),
    instance_count=1,
    instance_type='ml.c5.xlarge',
    output_path=output_prefix)
fm.set_hyperparameters(
    feature_dim=num_features,
    predictor_type='regressor',
    num_factors=64,
    epochs=10)
```

Looking at the documentation (<https://docs.aws.amazon.com/sagemaker/latest/dg/fact-machines-hyperparameters.html>), we see that the required hyperparameters are `feature_dim`, `predictor_type`, and `num_factors`. The default setting for `epochs` is 1, which feels a little low, so we use 10 instead.

11. We then launch the training job. Did you notice that we didn't configure training inputs? We're simply passing the location of the two `protobuf` files. As `protobuf` is the default format for Factorization Machines (as well as other built-in algorithms), we can save a step:

```
fm.fit({'train': train_data, 'test': test_data})
```

12. Once the job is over, we deploy the model to a real-time endpoint:

```
endpoint_name = 'fm-movielens-100k'  
fm_predictor = fm.deploy(  
    endpoint_name=endpoint_name,  
    instance_type='ml.t2.medium',  
    initial_instance_count=1)
```

13. We'll now send samples to the endpoint in JSON format (<https://docs.aws.amazon.com/sagemaker/latest/dg/fact-machines.html#fm-inputoutput>). For this purpose, we write a custom serializer to convert input data to JSON. The default JSON deserializer will be used automatically since we set the content type to 'application/json':

```
import json  
from sagemaker.deserializers import JSONDeserializer  
from sagemaker.serializers import JSONSerializer  
  
class FMSerializer(JSONSerializer):  
    def serialize(self, data):  
        js = {'instances': []}  
        for row in data:  
            js['instances'].append({'features':  
                row.tolist()})  
        return json.dumps(js)  
fm_predictor.serializer = FMSerializer()  
fm_predictor.deserializer = JSONDeserializer()
```

14. We send the first three samples of the test set for prediction:

```
result = fm_predictor.predict(X_test[:3].toarray())  
print(result)
```

The prediction looks like this:

```
{'predictions': [{ 'score': 3.3772034645080566}, { 'score':  
3.4299235343933105}, { 'score': 3.6053106784820557}]}

---


```

15. Using this model, we could fill all the empty cells in the recommendation matrix. For each user, we would simply predict the score of all movies, and store, say, the top 50 movies. That information would be stored in a backend, and the corresponding metadata (title, genre, and so on) would be displayed to the user in a frontend application.
16. Finally, we delete the endpoint:

```
fm_predictor.delete_endpoint()
```

So far, we've only used supervised learning algorithms. In the next section, we'll move on to unsupervised learning with Principal Component Analysis.

Using Principal Component Analysis

Principal Component Analysis (PCA) is a dimension reductionality algorithm. It's often applied as a preliminary step before regression or classification. Let's use it on the protobuf dataset built in the Factorization Machines example. Its 2,625 columns are a good candidate for dimensionality reduction! We will use PCA by taking the following steps:

1. Starting from the processed dataset, we configure `Estimator` for PCA. By now, you should (almost) be able to do this with your eyes closed:

```
from sagemaker.image_uris import retrieve
region = sess.boto_session.region_name
container = retrieve('pca', region)
pca = sagemaker.estimator.Estimator(
    container=container,
    role=sagemaker.get_execution_role(),
    instance_count=1,
    instance_type='ml.c5.xlarge',
    output_path=output_prefix)
```

2. We then set the hyperparameters. The required ones are the initial number of features, the number of principal components to compute, and the batch size:

```
pca.set_hyperparameters(feature_dim=num_features,
                        num_components=64,
                        mini_batch_size=1024)
```

3. We train and deploy the model:

```
pca.fit({'train': train_data, 'test': test_data})  
  
pca_predictor = pca.deploy(  
    endpoint_name='pca-movielens-100k',  
    instance_type='ml.t2.medium',  
    initial_instance_count=1)
```

4. Then, we predict the first test sample, using the same serialization code as in the previous example:

```
import json  
  
from sagemaker.deserializers import JSONDeserializer  
from sagemaker.serializers import JSONSerializer  
  
class PCASerializer(JSONSerializer):  
    def serialize(self, data):  
        js = {'instances': []}  
        for row in data:  
            js['instances'].append({'features':  
                row.tolist()})  
        return json.dumps(js)  
  
pca_predictor.serializer = PCASerializer()  
pca_predictor.deserializer = JSONDeserializer()  
result = pca_predictor.predict(X_test[0].toarray())  
print(result)
```

This prints out the 64 principal components of the test sample. In real life, we typically would process the dataset with this model, save the results, and use them to train a regression model:

```
{'projections': [{ 'projection': [-0.008711372502148151,  
    0.0019895541481673717, 0.002355781616643071,  
    0.012406938709318638, -0.0069608548656105995,  
    -0.009556426666676998, <output removed>] } ]}
```

Don't forget to delete the endpoint when you're done. Then, let's run one more unsupervised learning example to conclude this chapter!

Detecting anomalies with Random Cut Forest

Random Cut Forest (RCF) is an unsupervised learning algorithm for anomaly detection (<https://proceedings.mlr.press/v48/guha16.pdf>). We're going to apply it to a subset of the household electric power consumption dataset (<https://archive.ics.uci.edu/ml/>), available in the GitHub repository for this book. Data is aggregated hourly over a period of a little less than a year (just under 8,000 values):

1. In a Jupyter notebook, we load the dataset with `pandas`, and we display the first few lines:

```
import pandas as pd
df = pd.read_csv('item-demand-time.csv', dtype = object,
names=['timestamp','value','client'])
df.head(3)
```

As shown in the following screenshot, the dataset has three columns – an hourly timestamp, the power consumption value (in kilowatt-hours), and the client ID:

	timestamp	value	client
0	2014-01-01 01:00:00	38.34991708126038	client_12
1	2014-01-01 02:00:00	33.5820895522388	client_12
2	2014-01-01 03:00:00	34.41127694859037	client_12

Figure 4.4 – Viewing the columns

2. Using `matplotlib`, we plot the dataset to get a quick idea of what it looks like:

```
import matplotlib
import matplotlib.pyplot as plt
df.value=pd.to_numeric(df.value)
df_plot=df.pivot(index='timestamp',columns='item',
                  values='value')
df_plot.plot(figsize=(40,10))
```

The plot is shown in the following diagram. We see three time series corresponding to three different clients:

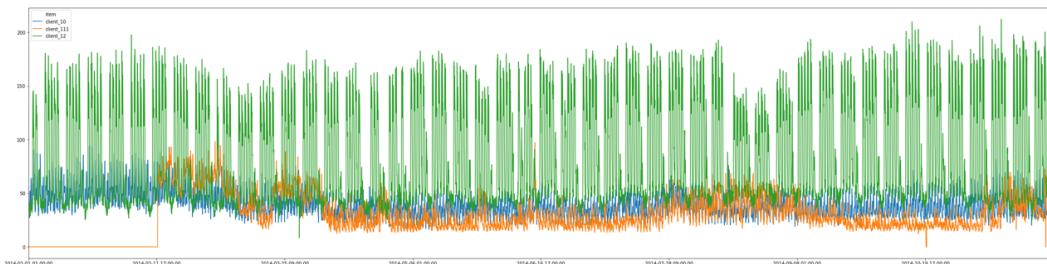


Figure 4.5 – Viewing the dataset

3. There are two issues with this dataset. First, it contains several time series: RCF can only train a model on a single series. Second, RCF requires **integer values**. Let's solve both problems with pandas – we only keep the "client_12" time series, we multiply its values by 100, and cast them to the integer type:

```
df = df[df['item']=='client_12']
df = df.drop(['item', 'timestamp'], axis=1)
df.value *= 100
df.value = df.value.astype('int32')
df.head()
```

The following diagram shows the first lines of the transformed dataset:

value	
0	3834
1	3358
2	3441

Figure 4.6 – The values of the first lines

4. We plot it again to check that it looks as expected. Note the large drop right after step 2000, highlighted by a box in the following screenshot. This is clearly an anomaly, and hopefully, our model will catch it:

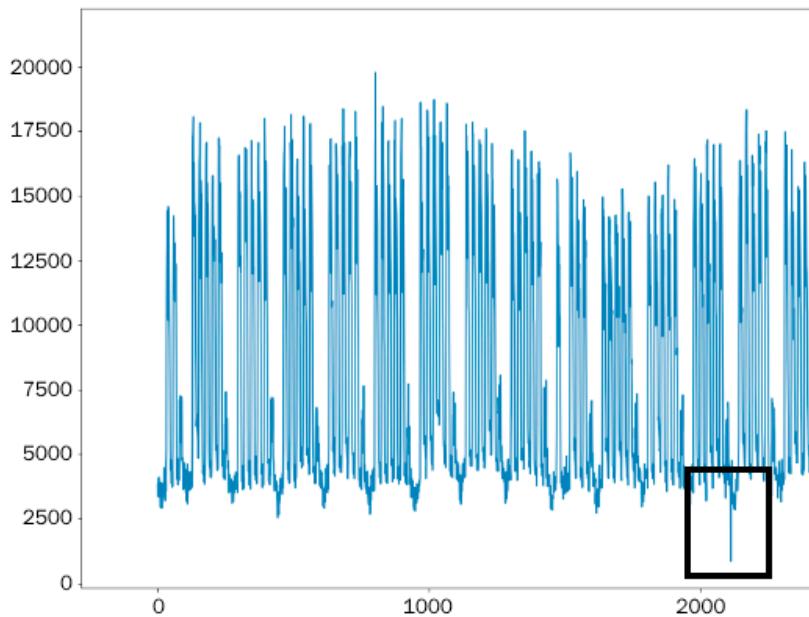


Figure 4.7 – Viewing a single time series

5. As in the previous examples, we save the dataset to a CSV file, which we upload to S3:

```
import sagemaker
sess = sagemaker.Session()
bucket = sess.default_bucket()
prefix = 'electricity'
df.to_csv('electricity.csv', index=False,
          header=False)
training_data_path = sess.upload_data(
                      path='electricity.csv',
                      key_prefix=prefix +
                      '/input/training')
```

6. Then, we define the **training channel**. There are a couple of quirks that we haven't met before. SageMaker generally doesn't have many of these, and reading the documentation goes a long way in pinpointing them (<https://docs.aws.amazon.com/sagemaker/latest/dg/randomcutforest.html>).

First, the **content type** must state that data is not labeled. The reason for this is that RCF can accept an optional test channel where anomalies are labeled (`label_size=1`). Even though the training channel never has labels, we still need to tell RCF. Second, the only **distribution policy** supported in RCF is `ShardedByS3Key`. This policy splits the dataset across the different instances in the training cluster, instead of sending them a full copy. We won't run distributed training here, but we need to set that policy nonetheless:

```
training_data_channel =
    sagemaker.TrainingInput(
        s3_data=training_data_path,
        content_type='text/csv;label_size=0',
        distribution='ShardedByS3Key')
rcf_data = {'train': training_data_channel}
```

7. The rest is business as usual: train and deploy! Once again, we reuse the code for the previous examples, and it's almost unchanged:

```
from sagemaker.estimator import Estimator
from sagemaker.image_uris import retrieve

region = sess.boto_session.region_name
container = retrieve('randomcutforest', region)
rcf_estimator = Estimator(container,
    role= sagemaker.get_execution_role(),
    instance_count=1,
    instance_type='ml.m5.large',
    output_path='s3://{{}}/{{}}/output'.format(bucket,
                                                prefix))

rcf_estimator.set_hyperparameters(feature_dim=1)
rcf_estimator.fit(rcf_data)
endpoint_name = 'rcf-demo'
rcf_predictor = rcf_estimator.deploy(
    endpoint_name=endpoint_name,
    initial_instance_count=1,
    instance_type='ml.t2.medium')
```

- After a few minutes, the model is deployed. We convert the input time series to a Python list, and we send it to the endpoint for prediction. We use CSV and JSON, respectively, for serialization and deserialization:

```
rcf_predictor.serializer =
    sagemaker.serializers.CSVSerializer()
rcf_predictor.deserializer =
    sagemaker.deserializers.JSONDeserializer()
values = df['value'].astype('str').tolist()
response = rcf_predictor.predict(values)
print(response)
```

The response contains the anomaly score for each value in the time series. It looks like this:

```
{'scores': [{ 'score': 1.0868037776}, { 'score':
1.5307718138}, { 'score': 1.4208102841} ...}
```

- We then convert this response to a Python list, and we then compute its mean and its standard deviation:

```
from statistics import mean, stdev
scores = []
for s in response['scores']:
    scores.append(s['score'])
score_mean = mean(scores)
score_std = stdev(scores)
```

- We plot a subset of the time series and the corresponding scores. Let's focus on the "[2000:2500]" interval, as this is where we saw a large drop. We also plot a line representing the mean plus three standard deviations (99.7% of the score distribution) – any score largely exceeding the line is likely to be an anomaly:

```
df[2000:2500].plot(figsize=(40,10))
plt.figure(figsize=(40,10))
plt.plot(scores[2000:2500])
plt.autoscale(tight=True)
plt.axhline(y=score_mean+3*score_std, color='red')
plt.show()
```

The drop is clearly visible in the following diagram:

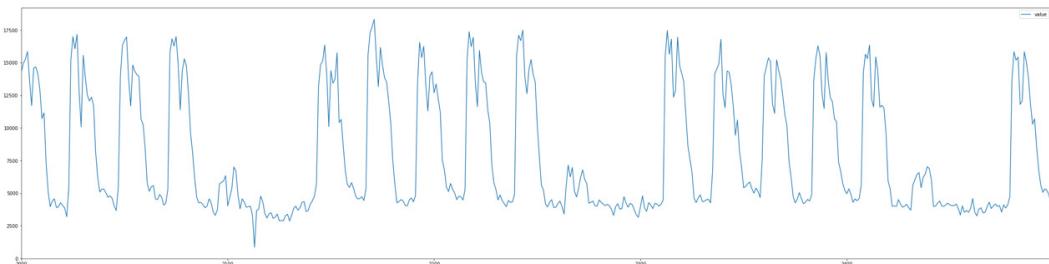


Figure 4.8 – Zooming in on an anomaly

As you can see on the following score plot, its score is sky high! Beyond a doubt, this value is an anomaly:

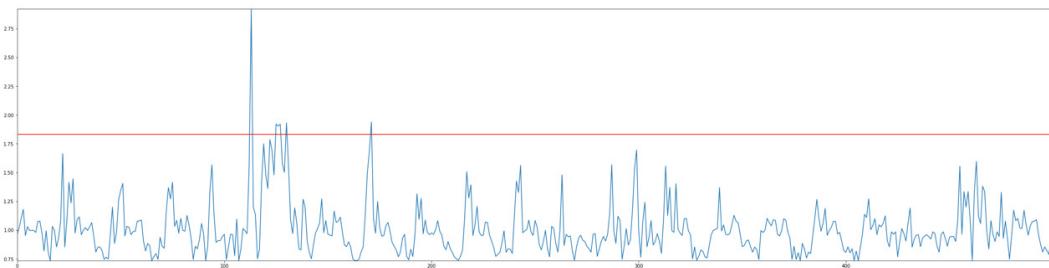


Figure 4.9 – Viewing anomaly scores

Exploring other intervals of the time series, we could certainly find more. Who said machine learning wasn't fun?

11. Finally, we delete the endpoint:

```
rcf_predictor.delete_endpoint()
```

Having gone through five complete examples, you should now be familiar with built-in algorithms, the SageMaker workflow, and the SDK. To fully master these topics, I would recommend experimenting with your datasets and running additional examples available at https://github.com/awslabs/amazon-sagemaker-examples/tree/master/introduction_to_amazon_algorithms.

Summary

As you can see, built-in algorithms are a great way to quickly train and deploy models without having to write any machine learning code.

In this chapter, you learned about the SageMaker workflow, and how to implement it with a handful of APIs from the SageMaker SDK, without ever worrying about infrastructure.

You learned how to work with data in CSV and RecordIO-wrapped protobuf format, the latter being the preferred format for large-scale training on bulky datasets. You also learned how to build models with important algorithms for supervised and unsupervised learning: Linear Learner, XGBoost, Factorization Machines, PCA, and Random Cut Forest.

In the next chapter, you will learn how to use additional built-in algorithms to build computer vision models.

5

Training CV Models

In the previous chapter, you learned how to use SageMaker's built-in algorithms for traditional machine learning problems, including classification, regression, and anomaly detection. We saw that these algorithms work well on tabular data, such as CSV files. However, they are not well suited for image datasets, and they generally perform very poorly on **CV** (**CV**) tasks.

For a few years now, CV has taken the world by storm, and not a month goes by without a new breakthrough in extracting patterns from images and videos. In this chapter, you will learn about three built-in algorithms designed specifically for CV tasks. We'll discuss the types of problems that you can solve with them. We'll also spend a lot of time explaining how to prepare image datasets, as this crucial topic is often inexplicably overlooked. Of course, we'll train and deploy models too.

This chapter covers the following topics:

- Discovering the CV built-in algorithms in Amazon SageMaker
- Preparing image datasets
- Using the built-in CV algorithms: **image classification**, **object detection**, and **semantic segmentation**

Technical requirements

You will need an AWS account to run the examples included in this chapter. If you haven't got one already, please point your browser at <https://aws.amazon.com/getting-started/> to create one. You should also familiarize yourself with the AWS Free Tier (<https://aws.amazon.com/free/>), which lets you use many AWS services for free within certain usage limits.

You will need to install and configure the AWS **Command Line Interface (CLI)** for your account (<https://aws.amazon.com/cli/>).

You will need a working Python 3.x environment. Installing the Anaconda distribution (<https://www.anaconda.com/>) is not mandatory, but strongly encouraged, as it includes many projects that we will need (Jupyter, pandas, numpy, and more).

The code examples included in the book are available on GitHub at <https://github.com/PacktPublishing/Learn-Amazon-SageMaker-second-edition>. You will need to install a Git client to access them (<https://git-scm.com/>).

Discovering the CV built-in algorithms in Amazon SageMaker

SageMaker includes three CV algorithms, based on proven deep learning networks. In this section, you'll learn about these algorithms, what kind of problem they can help you solve, and what their training scenarios are:

- **Image classification** assigns one or more labels to an image.
- **Object detection** detects and classifies objects in an image.
- **Semantic segmentation** assigns every pixel of an image to a specific class.

Discovering the image classification algorithm

Starting from an input image, the **image classification** algorithm predicts a probability for each class present in the training dataset. This algorithm is based on the **ResNet** convolutional neural network (<https://arxiv.org/abs/1512.03385>). Published in 2015, **ResNet** won the ILSVRC classification task that same year (<http://www.image-net.org/challenges/LSVRC/>). Since then, it has become a popular and versatile choice for image classification.

Many hyperparameters can be set, including the depth of the network, which can range from 18 to 200 layers. In general, the more layers the network has, the better it will learn, at the expense of increased training times.

Please note that the **image classification** algorithm supports both **single-label** and **multi-label** classification. We will focus on single-label classification in this chapter. Working with several labels is very similar, and you'll find a complete example at https://github.com/awslabs/amazon-sagemaker-examples/blob/master/introduction_to_amazon_algorithms/imageclassification_mscoco_multi_label/.

Discovering the object detection algorithm

Starting from an input image, the **object detection** algorithm predicts both the class and the location of each object in the image. Of course, the algorithm can only detect object classes present in the training dataset. The location of each object is defined by a set of four coordinates, called a **bounding box**.

This algorithm is based on the **Single Shot MultiBox Detector (SSD)** architecture (<https://arxiv.org/abs/1512.02325>). For classification, you can pick from two base networks: **VGG-16** (<https://arxiv.org/abs/1409.1556>) or **ResNet-50**.

The following output shows an example of object detection (source: <https://www.dressagechien.net/wp-content/uploads/2017/11/chien-et-velo.jpg>):

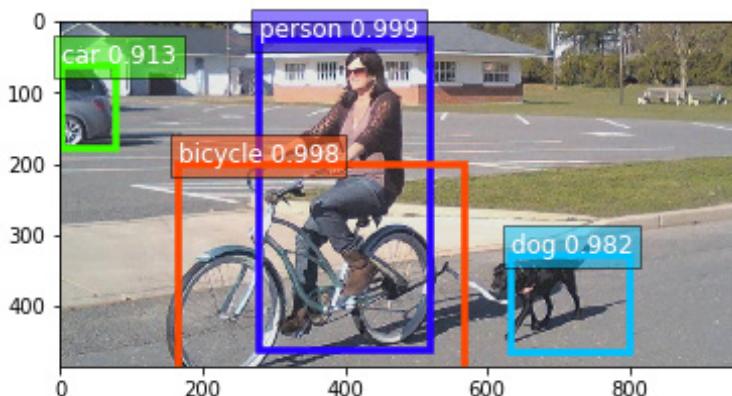


Figure 5.1 – Test image

Discovering the semantic segmentation algorithm

Starting from an input image, the **semantic segmentation** algorithm predicts the class of every pixel of the image. This is a much harder problem than image classification (which only considers the full image) or object detection (which only focuses on specific parts of the image). Using the probabilities contained in a prediction, it's possible to build **segmentation masks** that cover specific objects in the picture.

Three neural networks may be used for segmentation:

- **Fully Convolutional Networks (FCNs)**: <https://arxiv.org/abs/1411.4038>
- **Pyramid Scene Parsing (PSP)**: <https://arxiv.org/abs/1612.01105>
- **DeepLab v3**: <https://arxiv.org/abs/1706.05587>

The encoder network is **ResNet**, with either 50 or 101 layers.

The following output shows the result of segmenting the previous image. We see the segmentation masks, and each class is assigned a unique color; the background is black, and so on:

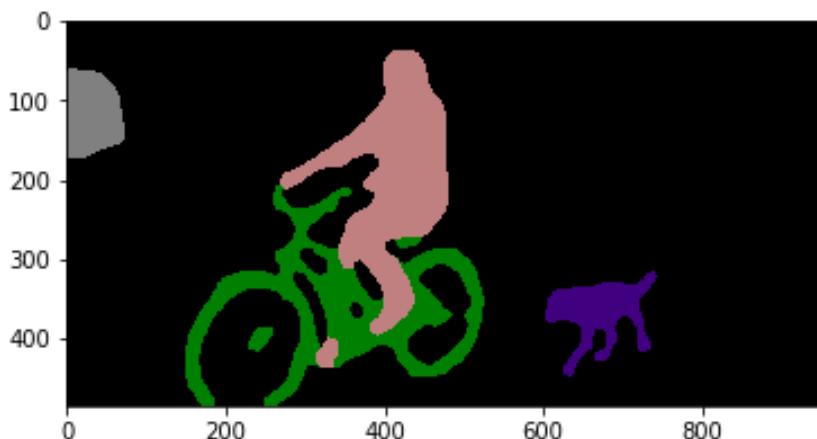


Figure 5.2 – Segmented test image

Now let's see how we can train these algorithms on our own data.

Training with CV algorithms

All three algorithms are based on **supervised learning**, so our starting point will be a labeled dataset. Of course, the nature of these labels will be different for each algorithm:

- Class labels for **image classification**
- Bounding boxes and class labels for **object detection**
- Segmentation masks and class labels for **semantic segmentation**

Annotating image datasets is a lot of work. If you need to build your own dataset, **Amazon SageMaker Ground Truth** can definitely help, and we studied it in *Chapter 2, Handling Data Preparation Techniques*. Later in this chapter, we'll show you how to use image datasets labeled with Ground Truth.

When it comes to packaging datasets, the use of **RecordIO** files is strongly recommended (<https://mxnet.apache.org/api/faq/recordio>). Packaging images in a small number of record-structured files makes it much easier to move datasets around and to split them for distributed training. Having said that, you can also train on individual image files if you prefer.

Once your dataset is ready in S3, you need to decide whether you'd like to train from scratch, or whether you'd like to start from a pretrained network.

Training from scratch is fine if you have plenty of data, and if you're convinced that there's value in building a specific model with it. However, this will take a lot of time, possibly hundreds of epochs, and hyperparameter selection will be absolutely critical in getting good results.

Using a pretrained network is generally a better option, even if you have lots of data. Thanks to **transfer learning**, you can start from a model trained on a huge collection of images (think millions) and fine-tune it on your data and classes. Training will be much shorter, and you will get models with higher accuracy rates quicker.

Given the complexity of the models and the size of datasets, training with CPU instances is simply not an option. We'll use GPU instances for all examples.

Last but not least, all three algorithms are based on **Apache MXNet**. This lets you export their models outside of SageMaker and deploy them anywhere you like.

In the next sections, we're going to zoom in on image datasets, and how to prepare them for training.

Preparing image datasets

Input formats are more complex for image datasets than for tabular datasets, and we need to get them exactly right. The CV algorithms in SageMaker support three input formats:

- Image files
- **RecordIO** files
- Augmented manifests built by **SageMaker Ground Truth**

In this section, you'll learn how to prepare datasets in these different formats. To the best of my knowledge, this topic has rarely been addressed in such detail. Get ready to learn a lot!

Working with image files

This is the simplest format, and it's supported by all three algorithms. Let's see how to use it with the **image classification** algorithm.

Converting an image classification dataset to image format

A dataset in image format has to be stored in S3. Image files don't need to be sorted in any way, and you simply could store all of them in the same bucket.

Images are described in a **list file**, a text file containing a line per image. For **image classification**, three columns are present: the unique identifier of the image, its class label, and its path. Here's an example:

```
1023 5 prefix/image2753.jpg
38   6 another_prefix/image72.jpg
983 2 yet_another_prefix/image863.jpg
```

The first line tells us that `image2753.jpg` belongs to class 5 and has been assigned ID 1023.

You need a list file for each channel, so you would need one for the training dataset, one for the validation dataset, and so on. You can either write bespoke code to generate them, or you can use a simple program that is part of **Apache MXNet**. This program is called `im2rec`, and it's available in Python and C++. We'll use the Python version.

Let's use the Dogs vs. Cats dataset available on **Kaggle** (<https://www.kaggle.com/c/dogs-vs-cats>). This dataset is 812 MB. Unsurprisingly, it contains two classes: dogs and cats. It's already split for training and testing (25,000 and 12,500 images, respectively). Here's how:

1. We create a **Kaggle** account, accept the rules of the Dogs vs. Cats competition, and install the kaggle CLI (<https://github.com/Kaggle/kaggle-api>).
2. On our local machine, we download and extract the training dataset (you can ignore the test set, which is only needed for the competition):

```
$ kaggle competitions download -c dogs-vs-cats
$ sudo yum -y install zip unzip
$ unzip dogs-vs-cats.zip
$ unzip train.zip
```

3. Dog and cat images are mixed up in the same folder. We create a subfolder for each class, and move the appropriate images there:

```
$ cd train
$ mkdir dog cat
$ find . -name 'dog.*' -exec mv {} dog \;
$ find . -name 'cat.*' -exec mv {} cat \;
```

4. We'll need validation images, so let's move 1,250 random dog images and 1,250 random cat images to specific directories. I'm using bash scripting here, but feel free to use any tool you like:

```
$ mkdir -p val/dog val/cat
$ ls dog | sort -R | tail -1250 | while read file;
do mv dog/$file val/dog; done
$ ls cat | sort -R | tail -1250 | while read file;
do mv cat/$file val/cat; done
```

5. We move the remaining 22,500 images to the training folder:

```
$ mkdir train
$ mv dog cat train
```

6. Our dataset now looks like this:

```
$ du -h
33M      ./val/dog
```

```
28M    ./val/cat  
60M    ./val  
289M   ./train/dog  
248M   ./train/cat  
537M   ./train  
597M   .
```

7. We download the im2rec tool from GitHub (<https://github.com/apache/incubator-mxnet/blob/master/tools/im2rec.py>). It requires dependencies, which we need to install (you may have to adapt the command to your own environment and flavor of Linux):

```
$ wget https://raw.githubusercontent.com/apache/  
incubator-mxnet/master/tools/im2rec.py  
  
$ sudo yum -y install python-devel python-pip opencv  
opencv-devel opencv-python  
  
$ pip3 install mxnet opencv-python
```

8. We run im2rec to build two list files, one for training data and one for validation data:

```
$ python3 im2rec.py --list --recursive dogscats-train  
train  
  
$ python3 im2rec.py --list --recursive dogscats-val val
```

This creates the dogscats-train.lst and dogscats-val.lst files. Their three columns are a unique image identifier, the class label (0 for cats, 1 for dogs), and the image path, as follows:

```
3197  0.000000  cat/cat.1625.jpg  
15084 1.000000  dog/dog.12322.jpg  
1479  0.000000  cat/cat.11328.jpg  
5262  0.000000  cat/cat.3484.jpg  
20714 1.000000  dog/dog.6140.jpg
```

9. We move the list files to specific directories. This is required because they will be passed to the Estimator as two new channels, train_lst and validation_lst:

```
$ mkdir train_lst val_lst  
$ mv dogscats-train.lst train_lst  
$ mv dogscats-val.lst val_lst
```

10. The dataset now looks like this:

```
$ du -h
33M      ./val/dog
28M      ./val/cat
60M      ./val
700K     ./train_lst
80K      ./val_lst
289M     ./train/dog
248M     ./train/cat
537M     ./train
597M     .
```

11. Finally, we sync this folder to the SageMaker default bucket for future use. Please make sure to only sync the four folders, and nothing else:

```
$ aws s3 sync .
s3://sagemaker-eu-west-1-123456789012/dogscats-images/
input/
```

Now, let's move on to using the image format with the object detection algorithms.

Converting detection datasets to image format

The general principle is identical. We need to build a file tree representing the four channels: `train`, `validation`, `train_annotation`, and `validation_annotation`.

The main difference lies in how labeling information is stored. Instead of list files, we need to build JSON files.

Here's an example of a fictitious picture in an object detection dataset. For each object in the picture, we define the coordinates of the top-left corner of its bounding box, its height, and its width. We also define the class identifier, which points to a category array that also stores class names:

```
{
  "file": "my-prefix/my-picture.jpg",
  "image_size": [{"width": 512, "height": 512, "depth": 3}],
  "annotations": [
    {
      "class_id": 1,
```

```
        "left": 67, "top": 121, "width": 61, "height": 128
    },
    {
        "class_id": 5,
        "left": 324, "top": 134, "width": 112, "height": 267
    }
],
"categories": [
    { "class_id": 1, "name": "pizza" },
    { "class_id": 5, "name": "beer" }
]
}
```

We would need to do this for every picture in the dataset, building a JSON file for the training set and one for the validation set.

Finally, let's see how to use the image format with the semantic segmentation algorithm.

Converting segmentation datasets to image format

Image format is the only format supported by the image segmentation algorithm.

This time, we need to build a file tree representing the four channels: `train`, `validation`, `train_annotation`, and `validation_annotation`. The first two channels contain the source images, and the last two contain the segmentation mask images.

File naming is critical in matching an image to its mask: the source image and the mask image must have the same name in their respective channels. Here's an example:

```
|__ train
|  |__ image001.png
|  |__ image007.png
|  |__ image042.png
|__ train_annotation
|  |__ image001.png
|  |__ image007.png
|  |__ image042.png
|__ validation
|  |__ image059.png
```

```
|   └── image062.png
|   └── image078.png
└── validation_annotation
    ├── image059.png
    ├── image062.png
    └── image078.png
```

You can see sample pictures in the following figure. The source image on the left would go to the `train` folder and the mask picture on the right would go to the `train_annotation` folder. They would have to have exactly the same name so that the algorithm could match them:



Figure 5.3 – Sample image from the Pascal VOC dataset

One clever feature of this format is how it matches class identifiers to mask colors. Mask images are PNG files with a 256-color palette. Each class in the dataset is assigned a specific entry in the color palette. These colors are the ones you see in masks for objects belonging to that class.

If your labeling tool or your existing dataset doesn't support this PNG feature, you can add your own color mapping file. Please refer to the AWS documentation for details: <https://docs.aws.amazon.com/sagemaker/latest/dg/semantic-segmentation.html>.

Now, let's prepare the **Pascal VOC** dataset. This dataset is frequently used to benchmark object detection and semantic segmentation model:

1. We first download and extract the 2012 version of the dataset. Again, I recommend using an AWS-hosted instance to speed up network transfers:

```
$ wget https://data.deeppai.org/PascalVOC2012.zip  
$ unzip PascalVOC2012.zip
```

2. We create a work directory where we'll build the four channels:

```
$ mkdir input  
$ cd input  
$ mkdir train validation train_annotation validation_annotation
```

3. Using the list of training files defined in the dataset, we copy the corresponding images to the `train` folder. I'm using bash scripting here; feel free to use your tool of choice:

```
$ for file in 'cat ../ImageSets/Segmentation/train.txt |  
xargs'; do cp ../JPEGImages/$file.jpg" train; done
```

4. We then do the same for validation images, training masks, and validation masks:

```
$ for file in 'cat ../ImageSets/Segmentation/val.txt |  
xargs'; do cp ../JPEGImages/$file.jpg" validation; done  
$ for file in 'cat ../ImageSets/Segmentation/train.txt  
| xargs'; do cp ../SegmentationClass/$file.png" train_  
annotation; done  
$ for file in 'cat ../ImageSets/Segmentation/val.  
txt | xargs'; do cp ../SegmentationClass/$file.png"  
validation_annotation; done
```

5. We check that we have the same number of images in the two training channels, and in the two validation channels:

```
$ for dir in train train_annotation validation  
validation_annotation; do find $dir -type f | wc -l; done
```

We see 1,464 training files and masks, and 1,449 validation files and masks. We're all set:

```
1464  
1464  
1449  
1449
```

6. The last step is to sync the file tree to S3 for later use. Again, please make sure to sync only the four folders:

```
$ aws s3 sync . s3://sagemaker-eu-west-1-123456789012/  
pascalvoc-segmentation/input/
```

We know how to prepare classification, detection, and segmentation datasets in image format. This is a critical step, and you have to get things exactly right.

Still, I'm sure that you found the steps in this section a little painful. So did I! Now imagine doing the same with millions of images. That doesn't sound very exciting, does it?

We need an easier way to prepare image datasets. Let's see how we can simplify dataset preparation with **RecordIO** files.

Working with RecordIO files

RecordIO files are easier to move around. It's much more efficient for an algorithm to read a large sequential file than to read lots of tiny files stored at random disk locations.

Converting an image classification dataset to RecordIO

Let's convert the Dogs vs. Cats dataset to RecordIO:

1. Starting from a freshly extracted copy of the dataset, we move the images to the appropriate class folder:

```
$ cd train  
$ mkdir dog cat  
$ find . -name 'dog.*' -exec mv {} dog \\;  
$ find . -name 'cat.*' -exec mv {} cat \\;
```

2. We run `im2rec` to generate list files for the training dataset (90%) and the validation dataset (10%). There's no need to split the dataset ourselves!

```
$ python3 im2rec.py --list --recursive --train-ratio 0.9  
dogscats .
```

3. We run `im2rec` once more to generate the RecordIO files:

```
$ python3 im2rec.py --num-thread 8 dogscats .
```

This creates four new files: two RecordIO files (`.rec`) containing the packed images, and two index files (`.idx`) containing the offsets of these images inside the record files:

```
$ ls dogscats*  
dogscats_train.idx dogscats_train.lst dogscats_train.rec  
dogscats_val.idx dogscats_val.lst dogscats_val.rec
```

4. Let's store the RecordIO files in S3, as we'll use them later:

```
$ aws s3 cp dogscats_train.rec s3://sagemaker-eu-  
west-1-123456789012/dogscats/input/train/  
$ aws s3 cp dogscats_val.rec s3://sagemaker-eu-  
west-1-123456789012/dogscats/input/validation/
```

This was much simpler, wasn't it? `im2rec` has additional options to resize images and more. It can also break the dataset into several chunks, a useful technique for **Pipe Mode** and **Distributed Training**. We'll study them in *Chapter 9, Scaling Your Training Jobs*.

Now, let's move on to using RecordIO files for object detection.

Converting an object detection dataset to RecordIO

The process is very similar. A major difference is the format of list files. Instead of dealing only with class labels, we also need to store bounding boxes.

Let's see what this means for the Pascal VOC dataset. The following image is taken from the dataset:



Figure 5.4 – Sample image from the Pascal VOC dataset

It contains three chairs. The labeling information is stored in an individual **XML** file, shown in slightly abbreviated form:

```
<annotation>
  <folder>VOC2007</folder>
  <filename>003988.jpg</filename>
  .
  .
  .
  <object>
    <name>chair</name>
    <pose>Unspecified</pose>
    <truncated>1</truncated>
    <difficult>0</difficult>
    <bndbox>
      <xmin>1</xmin>
      <ymin>222</ymin>
      <xmax>117</xmax>
      <ymax>336</ymax>
    </bndbox>
  </object>
  <object>
    <name>chair</name>
    <pose>Unspecified</pose>
```

```
<truncated>1</truncated>
<difficult>1</difficult>
<bndbox>
    <xmin>429</xmin>
    <ymin>216</ymin>
    <xmax>448</xmax>
    <ymax>297</ymax>
</bndbox>
</object>
<object>
    <name>chair</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>1</difficult>
    <bndbox>
        <xmin>281</xmin>
        <ymin>149</ymin>
        <xmax>317</xmax>
        <ymax>211</ymax>
    </bndbox>
</object>
</annotation>
```

Converting this to a list file entry should look like this:

9404	2	6	8.0000	0.0022	0.6607	0.2612	1.0000	0.0000	8.0000
0.9576	0.6429	1.0000	0.8839	1.0000	8.0000	0.6272	0.4435		
0.7076	0.6280	1.0000	VOC2007/JPEGImages/003988.jpg						

Let's decode each column:

- 9404 is a unique image identifier.
- 2 is the number of columns containing header information, including this one.
- 6 is the number of columns for labeling information. These six columns are the class identifier, the four bounding-box coordinates, and a flag telling us whether the object is difficult to see (we won't use it).

- The following is for the first object:
 - a) 8 is the class identifier. Here, 8 is the `chair` class.
 - b) 0.0022 0.6607 0.2612 1.0000 are the relative coordinates of the **bounding box** with respect to the height and width of the image.
 - c) 0 means that the object is not difficult.
- For the second object, we have the following:
 - a) 8 is the class identifier.
 - b) 0.9576 0.6429 1.0000 0.8839 are the coordinates of the second object.
 - c) 1 means that the object is difficult.
- The third object has the following:
 - a) 8 is the class identifier.
 - b) 0.6272 0.4435 0.7076 0.628 are the coordinates of the third object.
 - c) 1 means that the object is difficult.
- `VOC2007/JPEGImages/003988.jpg` is the path to the image.

So, how do we convert thousands of XML files into a couple of list files? Unless you enjoy writing parsers, this isn't a very exciting task.

Fortunately, our work has been cut out for us. Apache MXNet includes a Python script, `prepare_dataset.py`, that will handle this task. Let's see how it works:

1. For the next steps, I recommend using an Amazon Linux 2 EC2 instance with at least 10 GB of storage. Here are the setup steps:

```
$ sudo yum -y install git python3-devel python3-pip  
opencv opencv-devel opencv-python  
$ pip3 install mxnet opencv-python --user  
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/  
ec2-user/.local/lib/python3.7/site-packages/mxnet/  
$ sudo ldconfig
```

2. Download the 2007 and 2012 Pascal VOC datasets with `wget`, and extract them in the same directory:

```
$ mkdir pascalvoc  
$ cd pascalvoc  
$ wget https://data.deepai.org/PascalVOC2012.zip
```

```
$ wget https://data.deepai.org/PASCALVOC2007.zip  
$ unzip PascalVOC2012.zip  
$ unzip PASCALVOC2007.zip  
$ mv VOC2012 VOCtrainval_06-Nov-2007/VOCdevkit
```

3. Clone the Apache MXNet repository (<https://github.com/apache/incubator-mxnet/>):

```
$ git clone --single-branch --branch v1.4.x https://  
github.com/apache/incubator-mxnet
```

4. Run the `prepare_dataset.py` script to build our training dataset, merging the training and validation sets of the 2007 and 2012 versions:

```
$ cd VOCtrainval_06-Nov-2007  
$ python3 ../incubator-mxnet/example/ssd/tools/prepare_  
dataset.py --dataset pascal --year 2007,2012 --set  
trainval --root VOCdevkit --target VOCdevkit/train.lst  
$ mv VOCdevkit/train.* ..
```

5. Let's follow similar steps to generate our validation dataset, using the test set of the 2007 version:

```
$ cd ../../VOCtest_06-Nov-2007  
$ python3 ../incubator-mxnet/example/ssd/tools/prepare_  
dataset.py --dataset pascal --year 2007 --set test --root  
VOCdevkit --target VOCdevkit/val.lst  
$ mv VOCdevkit/val.* ..  
$ cd ..
```

6. In the top-level directory, we see the files generated by the script. Feel free to take a look at the list files; they should have the format presented previously:

```
train.idx train.lst train.rec  
val.idx val.lst val.rec
```

7. Let's store the RecordIO files in S3 as we'll use them later:

```
$ aws s3 cp train.rec s3://sagemaker-eu-  
west-1-123456789012/pascalvoc/input/train/  
$ aws s3 cp val.rec s3://sagemaker-eu-  
west-1-123456789012/pascalvoc/input/validation/
```

The `prepare_dataset.py` script has really made things simple here. It also supports the **COCO** dataset (<http://cocodataset.org>), and the workflow is extremely similar.

What about converting other public datasets? Well, your mileage may vary. You'll find more examples at https://gluon-cv.mxnet.io/build/examples_datasets/index.html.

RecordIO is definitely a step forward. Still, when working with custom datasets, it's very likely that you'll have to write your own list file generator. That's not a huge deal, but it's extra work.

Datasets labeled with **Amazon SageMaker Ground Truth** solve these problems altogether. Let's see how this works!

Working with SageMaker Ground Truth files

In *Chapter 2, Handling Data Preparation Techniques*, you learned about SageMaker Ground Truth workflows and their outcome, an **augmented manifest** file. This file is in **JSON Lines** format: each JSON object describes a specific annotation.

Here's an example from the semantic segmentation job we ran in *Chapter 2, Handling Data Preparation Techniques* (the story is the same for other task types). We see the paths to the source image and the segmentation mask, as well as color map information telling us how to match mask colors to classes:

```
{"source-ref": "s3://julien-sagemaker-book/chapter2/cat/cat1.jpg",
"my-cat-job-ref": "s3://julien-sagemaker-book/chapter2/cat/output/my-cat-job/annotations/consolidated-annotation/output/0_2020-04-21T13:48:00.091190.png",
"my-cat-job-ref-metadata": {
    "internal-color-map": {
        "0": { "class-name": "BACKGROUND", "hex-color": "#ffffffff",
               "confidence": 0.8054600000000001 },
        "1": { "class-name": "cat", "hex-color": "#2ca02c",
               "confidence": 0.8054600000000001 }
    },
    "type": "groundtruth/semantic-segmentation",
    "human-annotated": "yes",
    "creation-date": "2020-04-21T13:48:00.562419",
    "job-name": "labeling-job/my-cat-job" }}
```

The following images are the ones referenced in the preceding JSON document:

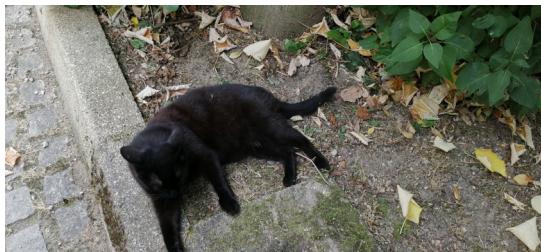


Figure 5.5 – Source image and segmented image

This is exactly what we would need to train our model. In fact, we can pass the augmented manifest to the SageMaker `Estimator` as is. No data processing is required whatsoever.

To use an **augmented manifest** pointing at labeled images in S3, we would simply pass its location and the name of the JSON attributes (highlighted in the previous example):

```
training_data_channel = sagemaker.s3_input(  
    s3_data=augmented_manifest_file_path,  
    s3_data_type='AugmentedManifestFile',  
    attribute_names=['source-ref', 'my-job-cat-ref'])
```

That's it! This is much simpler than anything we've seen before.

You can find more examples of using SageMaker Ground Truth at https://github.com/awslabs/amazon-sagemaker-examples/tree/master/ground_truth_labeling_jobs.

Now that we know how to prepare image datasets for training, let's put the CV algorithms to work.

Using the built-in CV algorithms

In this section, we're going to train and deploy models with all three algorithms using public image datasets. We will cover both training from scratch and transfer learning.

Training an image classification model

In this first example, let's use the image classification algorithm to build a model classifying the Dogs vs. Cats dataset that we prepared in a previous section. We'll first train using image format, and then using RecordIO format.

Training in image format

We will begin training using the following steps:

1. In a Jupyter notebook, we define the appropriate data paths:

```
import sagemaker
sess = sagemaker.Session()
bucket = sess.default_bucket()
prefix = 'dogscats-images'
s3_train_path =
    's3://{}//{}//input/train/'.format(bucket, prefix)
s3_val_path =
    's3://{}//{}//input/val/'.format(bucket, prefix)
s3_train_lst_path =
    's3://{}//{}//input/train_lst/'.format(bucket, prefix)
s3_val_lst_path =
    's3://{}//{}//input/val_lst/'.format(bucket, prefix)
s3_output = 's3://{}//{}//output/'.format(bucket, prefix)
```

2. We configure the Estimator for the image classification algorithm:

We use a GPU instance called `m1.p3.2xlarge`, which packs more than enough punch for this dataset (\$4.131/hour in eu-west-1).

What about hyperparameters (<https://docs.aws.amazon.com/sagemaker/latest/dg/IC-Hyperparameter.html>)? We set the number of classes (2) and the number of training samples (22,500). Since we're working with the image format, we need to resize images explicitly, setting the smallest dimension to 224 pixels. As we have enough data, we decide to train from scratch. In order to keep the training time low, we settle for an 18-layer **ResNet** model, and we train only for 10 epochs:

```
ic.set_hyperparameters(num_layers=18,
                      use_pretrained_model=0,
                      num_classes=2,
                      num_training_samples=22500,
                      resize=224,
                      mini_batch_size=128,
                      epochs=10)
```

3. We define the four channels, setting their content type to `application/x-image`:

```
from sagemaker import TrainingInput
train_data = TrainingInput (
    s3_train_path,
    content_type='application/x-image')
val_data = TrainingInput (
    s3_val_path,
    content_type='application/x-image')
train_lst_data = TrainingInput (
    s3_train_lst_path,
    content_type='application/x-image')
val_lst_data = TrainingInput (
    s3_val_lst_path,
    content_type='application/x-image')
s3_channels = {'train': train_data,
               'validation': val_data,
               'train_lst': train_lst_data,
               'validation_lst': val_lst_data}
```

4. We launch the training job as follows:

```
ic.fit(inputs=s3_channels)
```

In the training log, we see that data download takes about 3 minutes. Surprise, surprise: we also see that the algorithm builds RecordIO files before training. This step lasts about 1 minute:

```
Searching for .lst files in /opt/ml/input/data/train_lst.  
Creating record files for dogscats-train.lst  
Done creating record files...  
Searching for .lst files in /opt/ml/input/data/  
validation_lst.  
Creating record files for dogscats-val.lst  
Done creating record files...
```

5. As training starts, we see that an epoch takes approximately 22 seconds:

```
Epoch[0] Time cost=22.337  
Epoch[0] Validation-accuracy=0.605859
```

6. The job lasts 506 seconds in total (about 8 minutes), costing us $(506/3600) * \$4.131 = \0.58 . It reaches a validation accuracy of **91.2%** (hopefully, you see something similar). This is pretty good considering that we haven't even tweaked the hyperparameters yet.
7. We then deploy the model on a small CPU instance as follows:

```
ic_predictor = ic.deploy(initial_instance_count=1,  
                         instance_type='ml.t2.medium')
```

8. We download the following test image and send it for prediction in application/x-image format.



Figure 5.6 – Test picture

The simplest way to predict with built-in CV models is to use the `invoke_endpoint()` API in **boto3**. We'll use the following code to apply predictions to the image:

```
import boto3, json
import numpy as np
with open('test.jpg', 'rb') as f:
    payload = f.read()
    payload = bytearray(payload)

runtime = boto3.Session().client(
    service_name='runtime.sagemaker')
response = runtime.invoke_endpoint(
    EndpointName=ic_predictor.endpoint_name,
    ContentType='application/x-image',
    Body=payload)
```

```
result = response['Body'].read()
result = json.loads(result)
index = np.argmax(result)
print(result[index], index)
```

Printing out the probability and the class, our model indicates that this image is a dog with 99.997% confidence and that the image belongs to class 1:

```
0.9999721050262451 1
```

9. When we're done, we delete the endpoint as follows:

```
ic_predictor.delete_endpoint()
```

Now let's run the same training job with the dataset in RecordIO format.

Training in RecordIO format

The only difference is how we define the input channels. We only need two channels this time in order to serve the RecordIO files we uploaded to S3. Accordingly, the content type is set to `application/x-recordio`:

```
from sagemaker import TrainingInput
prefix = 'dogscats'
s3_train_path=
    's3://{{}}/{{}}/input/train/'.format(bucket, prefix)
s3_val_path=
    's3://{{}}/{{}}/input/validation/'.format(bucket, prefix)
train_data = TrainingInput(
    s3_train_path,
    content_type='application/x-recordio')
validation_data = TrainingInput(
    s3_val_path,
    content_type='application/x-recordio')
```

Training again, we see that data download takes 1 minute and that the file generation step has disappeared. Although it's difficult to draw any conclusion from a single run, using RecordIO datasets will generally save you time and money, even when training on a single instance.

The Dogs vs. Cats dataset has over 10,000 samples per class, which is more than enough to train from scratch. Now, let's try a dataset where that's not the case.

Fine-tuning an image classification model

Please consider the **Caltech-256** dataset, a popular public dataset of 15,240 images in 256 classes, plus a clutter class (http://www.vision.caltech.edu/Image_Datasets/Caltech256/). Browsing image categories, we see that all classes have a small number of samples. For instance, the "duck" class only has 60 images: it's doubtful that a deep learning algorithm, no matter how sophisticated, could extract the unique visual features of ducks with that little data.

In such cases, training from scratch is simply not an option. Instead, we will use a technique called **transfer learning**, where we start from a network that has already been trained on a very large and diverse image dataset. **ImageNet** (<http://www.image-net.org/>) is probably the most popular choice for pretraining, with 1,000 classes and millions of images.

The pretrained network has already learned how to extract patterns from complex images. Assuming that the images in our dataset are similar enough to those in the pretraining dataset, our model should be able to inherit that knowledge. Training for only a few more epochs on our dataset, we should be able to **fine-tune** the pretrained model on our data and classes.

Let's see how we can easily do this with SageMaker. In fact, we'll reuse the code for the previous example with minimal changes. Let's get into it:

1. We download the Caltech-256 in RecordIO format (if you'd like, you could download it in image format, and convert it to RecordIO: practice makes perfect!):

```
%%sh
wget http://data.mxnet.io/data/caltech-256/caltech-256-
60-train.rec
wget http://data.mxnet.io/data/caltech-256/caltech-256-
60-val.rec
```

2. We upload the dataset to S3:

```
import sagemaker
session = sagemaker.Session()
bucket = session.default_bucket()
prefix = 'caltech256/'
s3_train_path = session.upload_data(
    path='caltech-256-60-train.rec',
    bucket=bucket, key_prefix=prefix+'input/train')
s3_val_path = session.upload_data(
```

```
path='caltech-256-60-val.rec',
bucket=bucket, key_prefix=prefix+'input/validation')
```

3. We configure the `Estimator` function for the image classification algorithm. The code is strictly identical to *step 2* in the previous example.
4. We use **ResNet-50** this time, as it should be able to cope with the complexity of our images. Of course, we set `use_pretrained_network` to 1. The final fully connected layer of the pretrained network will be resized to the number of classes present in our dataset, and its weights will be assigned random values.

We set the correct number of classes (256+1) and training samples as follows:

```
ic.set_hyperparameters(num_layers=50,
                      use_pretrained_model=1,
                      num_classes=257,
                      num_training_samples=15240,
                      learning_rate=0.001,
                      epochs=5)
```

Since we're fine-tuning, we only train for 5 epochs, with a smaller learning rate of 0.001.

5. We configure channels and we launch the training job. The code is strictly identical to *step 4* in the previous example.
6. After 5 epochs and 272 seconds, we see the following metric in the training log:

```
Epoch [4] Validation-accuracy=0.8119
```

This is quite good for just a few minutes of training. Even with enough data, it would have taken much longer to get that result from scratch.

7. To deploy and test the model, we would reuse *steps 7-9* in the previous example.

As you can see, transfer learning is a very powerful technique. It can deliver excellent results, even when you have little data. You will also train for fewer epochs, saving time and money in the process.

Now, let's move on to the next algorithm, **object detection**.

Training an object detection model

In this example, we'll use the object detection algorithm to build a model on the Pascal VOC dataset that we prepared in a previous section:

1. We start by defining data paths:

```
import sagemaker
sess = sagemaker.Session()
bucket = sess.default_bucket()
prefix = 'pascalvoc'
s3_train_data = 's3://{}//{}//input/train'.format(bucket,
prefix)

s3_validation_data = 's3://{}//{}//input/validation'.
format(bucket, prefix)
s3_output_location = 's3://{}//{}//output'.format(bucket,
prefix)
```

2. We select the object detection algorithm:

```
from sagemaker.image_uris import retrieve
region = sess.boto_region_name
container = retrieve('object-detection', region)
```

3. We configure the Estimator:

```
od = sagemaker.estimator.Estimator(
    container,
    sagemaker.get_execution_role(),
    instance_count=1,
    instance_type='ml.p3.2xlarge',
    output_path=s3_output_location)
```

4. We set the required hyperparameters. We select a pretrained ResNet-50 network for the base network. We set the number of classes and training samples. We settle on 30 epochs, which should be enough to start seeing results:

```
od.set_hyperparameters(base_network='resnet-50',
use_pretrained_model=1,
num_classes=20,
```

```
    num_training_samples=16551,  
    epochs=30)
```

5. We then configure the two channels, and we launch the training job:

```
from sagemaker.session import TrainingInput  
train_data = TrainingInput (  
    s3_train_data,  
    content_type='application/x-recordio')  
validation_data = TrainingInput (  
    s3_validation_data,  
    content_type='application/x-recordio')  
data_channels = {'train': train_data,  
                 'validation': validation_data}  
od.fit(inputs=data_channels)
```

Selecting our job in **SageMaker components and registries | Experiments and trials**, we can see near-real-time metrics and charts. The next image shows the validation **mean average precision metric (mAP)**, a key metric for object detection models.

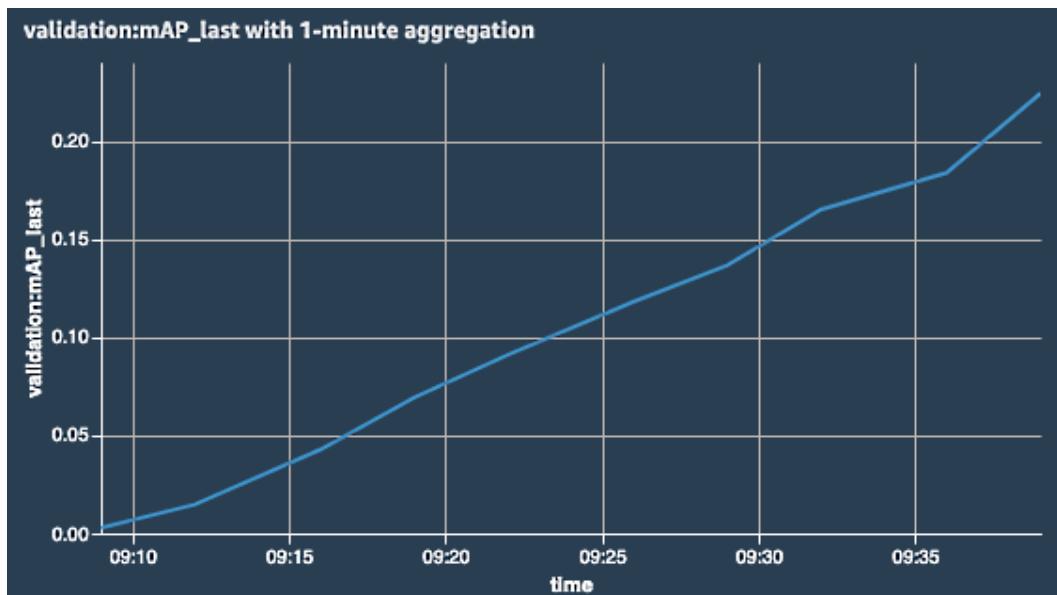


Figure 5.7 – Validation mAP

Please explore the other tabs (**Metrics**, **Parameters**, **Artifacts**, and so on). They contain everything you need to know about a particular job. Please note the **Stop training job** button in the top-right corner, which you can use to terminate a job at any time.

6. Training lasts for 1 hour and 40 minutes. This is a pretty heavy model! We get a **mean average precision metric (mAP)** of 0.5151. Production use would require more training, but we should be able to test the model already.
7. Given its complexity, we deploy the model to a larger CPU instance:

```
od_predictor = od.deploy(  
    initial_instance_count = 1,  
    instance_type= 'ml.c5.2xlarge')
```

8. We download a test image from Wikipedia and predict it with our model:

```
import boto3,json  
with open('test.jpg', 'rb') as image:  
    payload = image.read()  
    payload = bytearray(payload)  
  
runtime = boto3.Session().client(  
    service_name='runtime.sagemaker')  
response = runtime.invoke_endpoint(  
    EndpointName=od_predictor.endpoint_name,  
    ContentType='image/jpeg',  
    Body=payload)  
response = response['Body'].read()  
response = json.loads(response)
```

9. The response contains a list of predictions. Each individual prediction contains a class identifier, the confidence score, and the relative coordinates of the bounding box. Here are the first predictions in the response:

```
{'prediction':  
[[14.0, 0.7515302300453186, 0.39770469069480896,  
0.37605002522468567, 0.5998836755752563, 1.0],  
[14.0, 0.6490200161933899, 0.8020403385162354,  
0.2027685046195984, 0.9918708801269531,  
0.8575668931007385]
```

Using this information, we could plot the bounding boxes on the source image. For the sake of brevity, I will not include the code here, but you'll find it in the GitHub repository for this book. The following output shows the result:

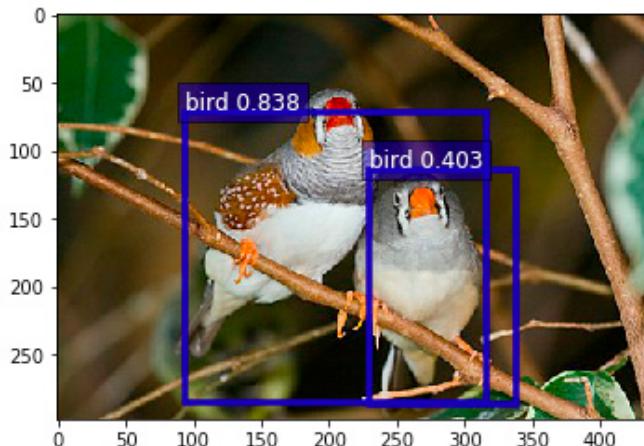


Figure 5.8 – Test image

- When we're done, we delete the endpoint as follows:

```
od_predictor.delete_endpoint()
```

This concludes our exploration of object detection. We have one more algorithm to go: **Semantic Segmentation**.

Training a semantic segmentation model

In this example, we'll use the semantic segmentation algorithm to build a model on the Pascal VOC dataset that we prepared in a previous section:

- As usual, we define the data paths, as follows:

```
import sagemaker
sess = sagemaker.Session()
bucket = sess.default_bucket()
prefix = 'pascalvoc-segmentation'
s3_train_data = 's3://{}//{}//input/train'.format(bucket,
prefix)
s3_validation_data = 's3://{}//{}//input/validation'.
format(bucket, prefix)
s3_train_annotation_data = 's3://{}//{}//input/train_
```

```
annotation'.format(bucket, prefix)
s3_validation_annotation_data = 's3://{}//{}//input/
validation_annotation'.format(bucket, prefix)
s3_output_location =
's3://{}//{}//output'.format(bucket, prefix)
```

2. We select the semantic segmentation algorithm, and we configure the `Estimator` function:

```
from sagemaker.image_uris import retrieve
container = retrieve('semantic-segmentation', region)
seg = sagemaker.estimator.Estimator(
    container,
    sagemaker.get_execution_role(),
    instance_count = 1,
    instance_type = 'ml.p3.2xlarge',
    output_path = s3_output_location)
```

3. We define the required hyperparameters. We select a pretrained ResNet-50 network for the base network and a pretrained **FCN** for detection. We set the number of classes and training samples. Again, we settle on 30 epochs, which should be enough to start seeing results:

```
seg.set_hyperparameters(backbone='resnet-50',
                       algorithm='fcn',
                       use_pretrained_model=True,
                       num_classes=21,
                       num_training_samples=1464,
                       epochs=30)
```

4. We configure the four channels, setting the content type to `image/jpeg` for source images, and `image/png` for mask images. Then, we launch the training job:

```
from sagemaker import TrainingInput
train_data = TrainingInput(
    s3_train_data,
    content_type='image/jpeg')
validation_data = TrainingInput(
    s3_validation_data,
    content_type='image/jpeg')
```

```

train_annotation = TrainingInput(
    s3_train_annotation_data,
    content_type='image/png')
validation_annotation = TrainingInput(
    s3_validation_annotation_data,
    content_type='image/png')
data_channels = {
    'train': train_data,
    'validation': validation_data,
    'train_annotation': train_annotation,
    'validation_annotation': validation_annotation
}
seg.fit(inputs=data_channels)

```

5. Training lasts about 32 minutes. We get a **mean intersection-over-union metric (mIOU)** of 0.4874, as shown in the following plot:

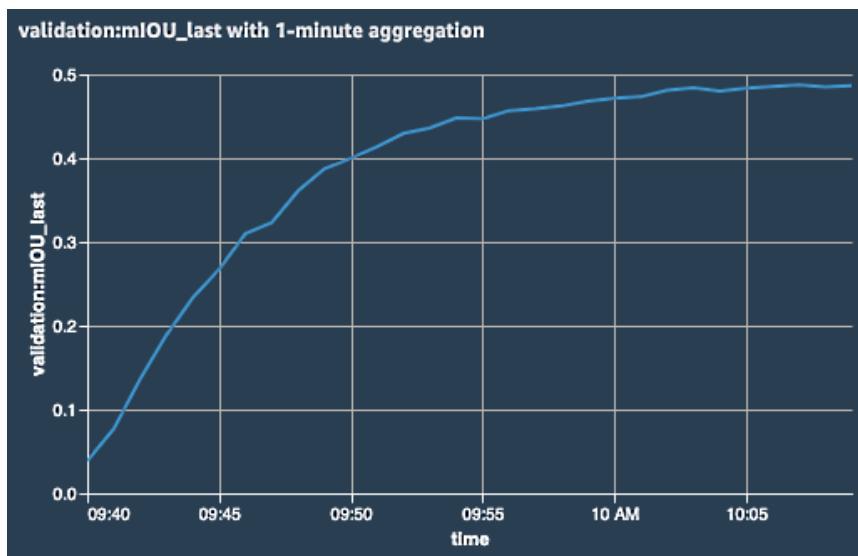


Figure 5.9 – Validation mIOU

6. We deploy the model to a CPU instance:

```

seg_predictor = seg.deploy(
    initial_instance_count=1,
    instance_type='ml.c5.2xlarge')

```

7. Once the endpoint is in service, we grab a test image, and we send it for prediction as a byte array with the appropriate content type:

```
!wget -O test.jpg https://bit.ly/3yhXB91
filename = 'test.jpg'
with open(filename, 'rb') as f:
    payload = f.read()
    payload = bytearray(payload)

runtime = boto3.Session().client(
    service_name='runtime.sagemaker')

response = runtime.invoke_endpoint(
    EndpointName=od_predictor.endpoint_name,
    ContentType='image/jpeg',
    Body=payload)
response = response['Body'].read()
response = json.loads(response)
```

8. Using the **Python Imaging Library (PIL)**, we process the response mask and display it:

```
import PIL
from PIL import Image
import numpy as np
import io
num_classes = 21
mask = np.array(Image.open(io.BytesIO(response)))
plt.imshow(mask, vmin=0, vmax=num_classes-1,
cmap='gray_r')
plt.show()
```

The following images show the source image and the predicted mask. This result is promising, and would improve with more training:

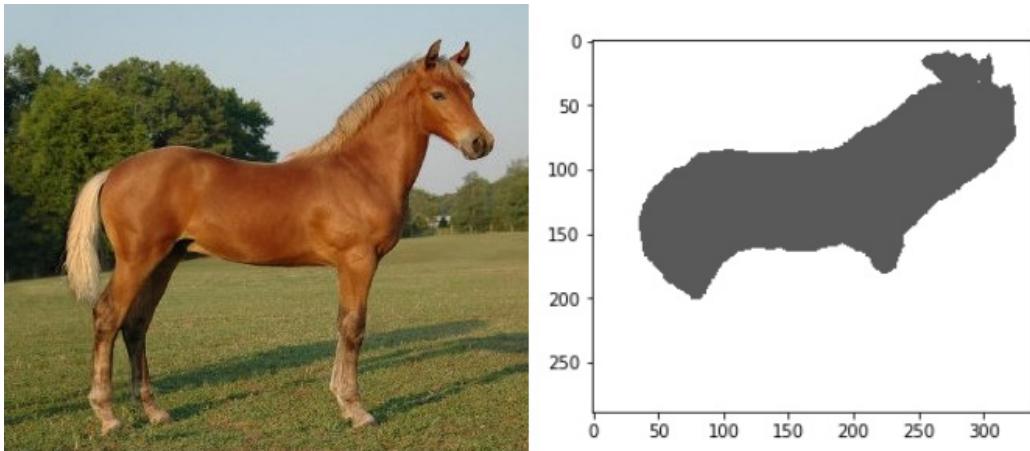


Figure 5.10 – Test image and segmented test image

- Predicting again with the `application/x-protobuf` accept type, we receive class probabilities for all pixels in the source image. The response is a protobuf buffer, which we save to a binary file:

```
response = runtime.invoke_endpoint(
    EndpointName=seg_predictor.endpoint_name,
    ContentType='image/jpeg',
    Accept='application/x-protobuf',
    Body=payload)
result = response['Body'].read()
seg_predictor.accept = 'application/x-protobuf'
response = seg_predictor.predict(img)
results_file = 'results.rec'
with open(results_file, 'wb') as f:
    f.write(response)
```

- The buffer contains two tensors: one with the shape of the probability tensor, and one with the actual probabilities. We load them using **Apache MXNet** and print their shape as follows:

```
from sagemaker.amazon.record_pb2 import Record
import mxnet as mx
rec = Record()
recordio = mx.recordio.MXRecordIO(results_file, 'r')
protobuf = rec.ParseFromString(recordio.read())
```

```
shape = list(rec.features["shape"].int32_tensor.values)
values = list(rec.features["target"].float32_tensor.
values)
print(shape.shape)
print(values.shape)
```

The output is as follows:

```
[1, 21, 289, 337]
2045253
```

This tells us that the `values` tensor describes one image of size 289x337, where each pixel is assigned 21 probabilities, one for each of the Pascal VOC classes. You can check that $289 \times 337 \times 21 = 2,045,253$.

11. Knowing that, we can now reshape the `values` tensor, retrieve the 21 probabilities for the (0,0) pixel, and print the class identifier with the highest probability:

```
mask = np.reshape(np.array(values), shape)
pixel_probs = mask[0, :, 0, 0]
print(pixel_probs)
print(np.argmax(pixel_probs))
```

Here is the output:

```
[9.68291104e-01 3.72813665e-04 8.14868137e-04
 1.22414716e-03
 4.57380433e-04 9.95167647e-04 4.29908326e-03
 7.52388616e-04
 1.46311778e-03 2.03254796e-03 9.60668200e-04
 1.51833100e-03
 9.39570891e-04 1.49350625e-03 1.51627266e-03
 3.63648031e-03
 2.17934581e-03 7.69103528e-04 3.05095245e-03
 2.10589729e-03
 1.12741732e-03]
0
```

The highest probability is at index 0: the predicted class for pixel (0,0) is class 0, the background class.

12. When we're done, we delete the endpoint as follows:

```
seg_predictor.delete_endpoint()
```

Summary

As you can see, these three algorithms make it easy to train CV models. Even with default hyperparameters, we get good results pretty quickly. Still, we start feeling the need to scale our training jobs. Don't worry once the relevant features have been covered in future chapters, we'll revisit some of our CV examples and we'll scale them radically!

In this chapter, you learned about image classification, object detection, and semantic segmentation algorithms. You also learned how to prepare datasets in Image, RecordIO, and SageMaker Ground Truth formats. Labeling and preparing data is a critical step that takes a lot of work, and we covered it in great detail. Finally, you learned how to use the SageMaker SDK to train and deploy models with the three algorithms, as well as how to interpret results.

In the next chapter, you will learn how to use built-in algorithms for natural language processing.

6

Training Natural Language Processing Models

In the previous chapter, you learned how to use SageMaker's built-in algorithms for **computer vision (CV)** to solve problems including image classification, object detection, and semantic segmentation.

Natural language processing (NLP) is another very promising field in ML. Indeed, NLP algorithms have proven very effective in modeling language and extracting context from unstructured text. Thanks to this, applications such as search and translation applications and chatbots are now commonplace.

In this chapter, you will learn about built-in algorithms designed specifically for NLP tasks and we'll discuss the types of problems that you can solve with them. As in the previous chapter, we'll also cover in great detail how to prepare real-life datasets such as Amazon customer reviews. Of course, we'll train and deploy models too. We will cover all of this under the following topics:

- Discovering the NLP built-in algorithms in Amazon SageMaker
- Preparing natural language datasets
- Using the built-in algorithms for NLP

Technical requirements

You will need an **Amazon Web Services (AWS)** account to run the examples included in this chapter. If you haven't got one already, please browse to <https://aws.amazon.com/getting-started/> to create it. You should also familiarize yourself with the AWS Free Tier (<https://aws.amazon.com/free/>), which lets you use many AWS services for free within certain usage limits.

You will need to install and configure the AWS **command-line interface (CLI)** tool for your account (<https://aws.amazon.com/cli/>).

You will need a working Python 3.x environment. Installing the Anaconda distribution (<https://www.anaconda.com/>) is not mandatory but strongly encouraged, as it includes many projects that we will need (Jupyter, pandas, numpy, and more).

The code examples included in the book are available on GitHub at <https://github.com/PacktPublishing/Learn-Amazon-SageMaker-second-edition>. You will need to install a Git client to access them (<https://git-scm.com/>).

Discovering the NLP built-in algorithms in Amazon SageMaker

SageMaker includes four NLP algorithms, enabling **supervised learning (SL)** and **unsupervised learning (UL)** scenarios. In this section, you'll learn about these algorithms, what kinds of problems they solve, and what their training scenarios are. Let's have a look at an overview of the algorithms we'll be discussing:

- **BlazingText** builds text classification models (SL) or computes word vectors (UL).
BlazingText is an Amazon-invented algorithm.

- **LDA** builds UL models that group a collection of text documents into topics. This technique is called **topic modeling**.
- **NTM** is another **topic modeling** algorithm based on neural networks, and it gives you more insight into how topics are built.
- **Sequence to Sequence (seq2seq)** builds **deep learning (DL)** models, predicting a sequence of output tokens from a sequence of input tokens.

Discovering the BlazingText algorithm

The BlazingText algorithm was invented by Amazon. You can read more about it at <https://dl.acm.org/doi/10.1145/3146347.3146354>. BlazingText is an evolution of **FastText**, a library for efficient text classification and representation learning developed by Facebook (<https://fasttext.cc>).

It lets you train text classification models, as well as computing **word vectors**. Also called **embeddings**, **word vectors** are the cornerstone of many NLP tasks, such as finding word similarities, word analogies, and so on. **Word2Vec** is one of the leading algorithms to compute these vectors (<https://arxiv.org/abs/1301.3781>), and it's the one BlazingText implements.

The main improvement of BlazingText is its ability to train on **graphics processing unit (GPU)** instances, where FastText only supports **central processing unit (CPU)** instances.

The speed gain is significant, and this is where its name comes from: "blazing" is faster than "fast"! If you're curious about benchmarks, you'll certainly enjoy this blog post: <https://aws.amazon.com/blogs/machine-learning/amazon-sagemaker-blazingtext-parallelizing-word2vec-on-multiple-cpus-or-gpus/>.

Finally, BlazingText is fully compatible with FastText. Models can be very easily exported and tested, as you will see later in the chapter.

Discovering the LDA algorithm

This UL algorithm uses a generative technique, named **topic modeling**, to identify topics present in a large collection of text documents. It was first applied to ML in 2003 (<http://jmlr.csail.mit.edu/papers/v3/blei03a.html>).

Please note that LDA is not a classification algorithm. You pass it the number of topics to build, not the list of topics you expect. To paraphrase Forrest Gump: "*Topic modeling is like a box of chocolates, you never know what you're gonna get.*"

LDA assumes that every text document in a collection was generated from several latent (meaning "hidden") topics. A topic is represented by a word probability distribution. For each word present in a collection of documents, this distribution gives the probability that the word appears in documents generated by this topic. For example, in a "finance" topic, the distribution would yield high probabilities for words such as "revenue," "quarter," or "earnings," and low probabilities for "ballista" or "platypus" (or so I should think).

Topic distributions are not considered independently. They are represented by a **Dirichlet distribution**, a multivariate generalization of univariate distributions (https://en.wikipedia.org/wiki/Dirichlet_distribution). This mathematical object gives the algorithm its name.

Given the number of words in the vocabulary and the number of latent topics, the purpose of the LDA algorithm is to build a model that is as close as possible to an ideal Dirichlet distribution. In other words, it will try to group words so that distributions are as well formed as possible and match the specified number of topics.

Training data needs to be carefully prepared. Each document needs to be converted into a **bag-of-words (BoW)** representation: each word is replaced by a pair of integers, representing a unique word **identifier (ID)** and the word count in the document. The resulting dataset can be saved either to **comma-separated values (CSV)** format or to **RecordIO-wrapped protobuf** format, a technique we already studied with **factorization machines** in *Chapter 4, Training Machine Learning Models*.

Once the model has been trained, we can score any document and get a score per topic. The expectation is that documents containing similar words should have similar scores, making it possible to identify their top topics.

Discovering the NTM algorithm

NTM is another algorithm for topic modeling. You can read more about it at <https://arxiv.org/abs/1511.06038>. The following blog post also sums up the key elements of the paper: <https://aws.amazon.com/blogs/machine-learning/amazon-sagemaker-neural-topic-model-now-supports-auxiliary-vocabulary-channel-new-topic-evaluation-metrics-and-training-subsampling/>.

As with LDA, documents need to be converted to a BoW representation, and the dataset can be saved to either CSV or RecordIO-wrapped protobuf format.

For training, NTM uses a completely different approach based on neural networks and—more precisely—on an encoder architecture (<https://en.wikipedia.org/wiki/Autoencoder>). In true DL fashion, the encoder trains on mini-batches of documents. It tries to learn their latent features by adjusting network parameters through backpropagation and optimization.

Unlike LDA, NTM can tell us which words are the most impactful in each topic. It also gives us two per-topic metrics, **word embedding topic coherence (WETC)** and **topic uniqueness (TU)**. These are outlined in more detail here:

- WETC tells us how semantically close the topic words are. This value is between 0 and 1; the higher, the better. It's computed using the **cosine similarity** (https://en.wikipedia.org/wiki/Cosine_similarity) of the corresponding word vectors in a pretrained **Global Vectors (GloVe)** model (another algorithm similar to Word2Vec).
- TU tells us how unique the topic is—that is to say, whether its words are found in other topics or not. Again, the value is between 0 and 1, and the higher the score, the more unique the topic is.

Once the model has been trained, we can score documents and get a score per topic.

Discovering the seq2seq algorithm

The seq2seq algorithm is based on **long short-term memory (LSTM)** neural networks (<https://arxiv.org/abs/1409.3215>). As its name implies, seq2seq can be trained to map one sequence of tokens to another. Its main application is machine translation, training on large bilingual corpora of text, such as the **Workshop on Statistical Machine Translation (WMT)** dataset (<http://www.statmt.org/wmt20/>).

In addition to the implementation available in SageMaker, AWS has also packaged the seq2seq algorithm into an open source project, **AWS Sockeye** (<https://github.com/awslabs/sockeye>), which also includes tools for dataset preparation.

I won't cover seq2seq in this chapter. It would take too many pages to get into the appropriate level of detail, and there's no point in just repeating what's already available in the Sockeye documentation.

You can find a seq2seq example in the notebook available at https://github.com/awslabs/amazon-sagemaker-examples/tree/master/introduction_to_amazon_algorithms/seq2seq_translation_en-de. Unfortunately, it uses the low-level boto3 **application programming interface (API)**, which we will cover in *Chapter 12, Automating Machine Learning Workflows*. Still, it's a valuable read, and you won't have too much trouble figuring things out.

Training with NLP algorithms

Just as for CV algorithms, training is the easy part, especially with the SageMaker **software development kit (SDK)**. By now, you should be familiar with the workflow and the APIs, and we'll keep using them in this chapter.

Preparing data for NLP algorithms is another story. First, real-life datasets are generally pretty bulky. In this chapter, we'll work with millions of samples and hundreds of millions of words. Of course, they need to be cleaned, processed, and converted to the format expected by the algorithm.

As we go through the chapter, we'll use the following techniques:

- Loading and cleaning data with the pandas library (<https://pandas.pydata.org>)
- Removing stop words and lemmatizing with the **Natural Language Toolkit (NLTK)** library (<https://www.nltk.org>)
- Tokenizing with the spaCy library (<https://spacy.io/>)
- Building vocabularies and generating BoW representations with the gensim library (<https://radimrehurek.com/gensim/>)
- Running data processing jobs with **Amazon SageMaker Processing**, which we studied in *Chapter 2, Handling Data Preparation Techniques*

Granted—this isn't an NLP book, and we won't go extremely far in processing data. Still, this will be quite fun, and hopefully an opportunity to learn about popular open source tools for NLP.

Preparing natural language datasets

For the CV algorithms in the previous chapter, data preparation focused on the technical format required for the dataset (**image** format, **RecordIO**, or **augmented manifest**). The images themselves weren't processed.

Things are quite different for NLP algorithms. The text needs to be heavily processed, converted, and saved in the right format. In most learning resources, these steps are abbreviated or even ignored. Data is already "automagically" ready for training, leaving the reader frustrated and sometimes dumbfounded on how to prepare their own datasets.

No such thing here! In this section, you'll learn how to prepare NLP datasets in different formats. Once again, get ready to learn a lot!

Let's start with preparing data for BlazingText.

Preparing data for classification with BlazingText

BlazingText expects labeled input data in the same format as FastText, outlined here:

- A plaintext file, with one sample per line.
- Each line has two fields, as follows:
 - a) A label in the form of `__label__LABELNAME__`
 - b) The text itself, formed into space-separated tokens (words and punctuation)

Let's get to work and prepare a customer review dataset for sentiment analysis (positive, neutral, or negative). We'll use the **Amazon Customer Reviews** dataset available at <https://s3.amazonaws.com/amazon-reviews-pds/readme.html>. That should be more than enough real-life data.

Before starting, please make sure that you have enough storage space. Here, I'm using a notebook instance with 10 **gigabytes (GB)** of storage. I've also picked a C5 instance type to run the processing steps faster. We'll proceed as follows:

1. Let's download the camera reviews by running the following code:

```
%%sh
aws s3 cp s3://amazon-reviews-pds/tsv/amazon_reviews_us_
Camera_v1_00.tsv.gz /tmp
```

2. We load the data with pandas, ignoring any line that causes an error. We also drop any line with missing values. The code is illustrated in the following snippet:

```
data = pd.read_csv(
    '/tmp/amazon_reviews_us_Camera_v1_00.tsv.gz',
    sep='\t', compression='gzip',
    error_bad_lines=False, dtype='str')
data.dropna(inplace=True)
```

3. We print the data shape and the column names, like this:

```
print(data.shape)
print(data.columns)
```

This gives us the following output:

```
(1800755, 15)
Index(['marketplace', 'customer_id', 'review_id', 'product_id',
       'product_parent', 'product_title', 'product_category',
       'star_rating', 'helpful_votes', 'total_votes', 'vine',
       'verified_purchase', 'review_headline', 'review_body',
       'review_date'], dtype='object')
```

4. 1.8 million lines! We keep 100,000, which is enough for our purpose. We also drop all columns except star_rating and review_body, as illustrated in the following code snippet:

```
data = data[:100000]
data = data[['star_rating', 'review_body']]
```

5. Based on star ratings, we add a new column named label, with labels in the proper format. You have to love how pandas makes this so simple. Then, we drop the star_rating column, as illustrated in the following code snippet:

```
data['label'] = data.star_rating.map({
    '1': '__label__negative__',
    '2': '__label__negative__',
    '3': '__label__neutral__',
    '4': '__label__positive__',
    '5': '__label__positive__'})
data = data.drop(['star_rating'], axis=1)
```

6. BlazingText expects labels at the beginning of each line, so we move the label column to the front, as follows:

```
data = data[['label', 'review_body']]
```

7. The data should now look like this:

	label	review_body
0	_label_positive_	ok
1	_label_positive_	Perfect, even sturdier than the original!
2	_label_negative_	If the words, "Cheap Chinese Junk"; com...
3	_label_positive_	Exactly what I wanted and expected. Perfect fo...
4	_label_positive_	I will look past the fact that they tricked me...

Figure 6.1 – Viewing the dataset

8. BlazingText expects space-separated tokens: each word and each punctuation sign must be space-separated from the next. Let's use the handy punkt tokenizer from the nltk library. Depending on the instance type you're using, this could take a couple of minutes. Here's the code you'll need:

```
!pip -q install nltk

import nltk
nltk.download('punkt')
data['review_body'] = data['review_body'].apply(nltk.
word_tokenize)
```

9. We join tokens into a single string, which we also convert to lowercase, as follows:

```
data['review_body'] =
    data.apply(lambda row: " ".join(row['review_body']))
        .lower(), axis=1)
```

10. The data should now look like this (notice that all tokens are correctly space-separated):

	label	review_body
0	_label_positive_	ok
1	_label_positive_	perfect , even sturdier than the original !
2	_label_negative_	if the words , & # 34 ; cheap chinese junk & #...
3	_label_positive_	exactly what i wanted and expected . perfect f...
4	_label_positive_	i will look past the fact that they tricked me...

Figure 6.2 – Viewing the tokenized dataset

11. Finally, we split the dataset for training (95%) and validation (5%), and we save both splits as plaintext files, as illustrated in the following code snippet:

```
from sklearn.model_selection import train_test_split
training, validation = train_test_split(data, test_size=0.05)
np.savetxt('/tmp/training.txt', training.values,
fmt='%s')
np.savetxt('/tmp/validation.txt', validation.values,
fmt='%s')
```

12. Opening one of the files, you should see plenty of lines similar to this one:

```
__label_neutral__ really works for me , especially on
the streets of europe . wished it was less expensive
though . the rain cover at the base really works . the
padding which comes in contact with your back though will
suffocate & make your back sweaty .
```

The data preparation wasn't too bad, was it? Still, tokenization ran for a minute or two. Now, imagine running it on millions of samples. Sure, you could fire up a larger environment in **SageMaker Studio**. You'd also pay more for as long as you're using it, which would probably be wasteful if only this one step required extra computing muscle. In addition, imagine having to run the same script on many other datasets. Do you want to do this manually again and again, waiting 20 minutes every time and hoping your notebook doesn't crash? Certainly not, I should say!

You already know the answer to both problems. It's **Amazon SageMaker Processing**, which we studied in *Chapter 2, Handling Data Preparation Techniques*. You should have the best of both worlds, using the smallest and least-expensive environment possible for experimentation, and running on-demand jobs when you need more resources. Day in, day out, you'll save money and get the job done faster.

Let's move this processing code to SageMaker Processing.

Preparing data for classification with BlazingText, version 2

We've covered this in detail in *Chapter 2, Handling Data Preparation Techniques*, so I'll go faster this time. We'll proceed as follows:

1. We upload the dataset to **Simple Storage Service (S3)**, as follows:

```
import sagemaker
session = sagemaker.Session()
prefix = 'amazon-reviews-camera'
input_data = session.upload_data(
    path='/tmp/amazon_reviews_us_Camera_v1_00.tsv.gz',
    key_prefix=prefix)
```

2. We define the processor by running the following code:

```
from sagemaker.sklearn.processing import SKLearnProcessor
sklearn_processor = SKLearnProcessor(
    framework_version='0.23-1',
    role= sagemaker.get_execution_role(),
    instance_type='ml.c5.2xlarge',
    instance_count=1)
```

3. We run the processing job, passing the processing script and its arguments, as follows:

```
from sagemaker.processing import ProcessingInput,
ProcessingOutput
sklearn_processor.run(
    code='preprocessing.py',
    inputs=[
        ProcessingInput(
            source=input_data,
            destination='/opt/ml/processing/input')
    ],
    outputs=[
        ProcessingOutput(
            output_name='train_data',
            source='/opt/ml/processing/train'),
```

```
        ProcessingOutput(
            output_name='validation_data',
            source='/opt/ml/processing/validation')
        ],
        arguments=[
            '--filename', 'amazon_reviews_us_Camera_v1_00.
tsv.gz',
            '--num-reviews', '100000',
            '--split-ratio', '0.05'
        ]
)
```

4. The abbreviated preprocessing script is shown in the following code snippet. The full version is in the GitHub repository for the book. We first install the `nltk` package, as follows:

```
import argparse, os, subprocess, sys
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
def install(package):
    subprocess.call([sys.executable, "-m", "pip",
                    "install", package])
if __name__=='__main__':
    install('nltk')
    import nltk
```

5. We read the command-line arguments, as follows:

```
parser = argparse.ArgumentParser()
parser.add_argument('--filename', type=str)
parser.add_argument('--num-reviews', type=int)
parser.add_argument('--split-ratio', type=float,
                    default=0.1)
args, _ = parser.parse_known_args()
filename = args.filename
num_reviews = args.num_reviews
split_ratio = args.split_ratio
```

6. We read the input dataset and process it, as follows:

```
input_data_path =  
    os.path.join('/opt/ml/processing/input', filename)  
data = pd.read_csv(input_data_path, sep='\t',  
                    compression='gzip', error_bad_lines=False,  
                    dtype='str')  
# Process data  
. . .
```

7. Finally, we split it for training and validation, and save it into two text files, as follows:

```
training, validation = train_test_split(  
    data,  
    test_size=split_ratio)  
training_output_path = os.path.join(  
    '/opt/ml/processing/train', 'training.txt')  
validation_output_path = os.path.join(  
    '/opt/ml/processing/validation', 'validation.txt')  
np.savetxt(training_output_path,  
           training.values, fmt='%s')  
np.savetxt(validation_output_path,  
           validation.values, fmt='%s')
```

As you can see, it doesn't take much to convert manual processing code into a SageMaker Processing job. You can actually reuse most of the code too, as it deals with generic topics such as command-line arguments, inputs, and outputs. The only trick is using `subprocess.call` to install dependencies inside the processing container.

Equipped with this script, you can now process data at scale as often as you want, without having to run and manage long-lasting notebooks.

Now, let's prepare data for the other BlazingText scenario: word vectors!

Preparing data for word vectors with BlazingText

BlazingText lets you compute word vectors easily and at scale. It expects input data in the following format:

- A plaintext file, with one sample per line.
- Each sample must have space-separated tokens (words and punctuations).

Let's process the same dataset as in the previous section, as follows:

1. We'll need the spaCy library, so let's install it along with its English language model, like this:

```
%%sh
pip -q install spacy
python -m spacy download en_core_web_sm
python -m spacy validate
```

2. We load the data with pandas, ignoring any line that causes an error. We also drop any line with missing values. We should have more than enough data anyway. Here's the code you'll need:

```
data = pd.read_csv(
    '/tmp/amazon_reviews_us_Camera_v1_00.tsv.gz',
    sep='\t', compression='gzip',
    error_bad_lines=False, dtype='str')
data.dropna(inplace=True)
```

3. We keep 100,000 lines, and we also drop all columns except `review_body`, as illustrated in the following code snippet:

```
data = data[:100000]
data = data[['review_body']]
```

We write a function to tokenize reviews with spaCy, and we apply it to the DataFrame. This step should be noticeably faster than nltk tokenization in the previous example, as spaCy is based on cython (<https://cython.org>). The code is illustrated in the following snippet:

```
import spacy
spacy_nlp = spacy.load('en_core_web_sm')
def tokenize(text):
    tokens = spacy_nlp.tokenizer(text)
```

```

tokens = [ t.text for t in tokens ]
return " ".join(tokens).lower()
data['review_body'] =
    data['review_body'].apply(tokenize)

```

The data should now look like this:

review_body	
0	ok
1	perfect , even sturdier than the original !
2	if the words , & # 34;cheap chinese junk" ;...
3	exactly what i wanted and expected . perfect f...
4	i will look past the fact that they tricked me...

Figure 6.3 – Viewing the tokenized dataset

- Finally, we save the reviews to a plaintext file, as follows:

```

import numpy as np
np.savetxt('/tmp/training.txt', data.values, fmt='%s')

```

- Opening this file, you should see one tokenized review per line, as illustrated in the following code snippet:

Ok
perfect , even sturdier than the original !

Here too, we should really be running these steps using SageMaker Processing. You'll find the corresponding notebook and preprocessing script in the GitHub repository for the book.

Now, let's prepare data for the LDA and NTM algorithms.

Preparing data for topic modeling with LDA and NTM

In this example, we will use the **Million News Headlines** dataset (<https://doi.org/10.7910/DVN/SYBGZL>), which is also available in the GitHub repository. As the name implies, it contains a million news headlines from the Australian news source *ABC*. Unlike product reviews, headlines are in very short sentences. Building a topic model should be an interesting challenge!

Tokenizing data

As you would expect, both algorithms require a tokenized dataset, so we'll proceed as follows:

1. We'll need the `nltk` and `gensim` libraries, so let's install them, as follows:

```
%%sh
pip -q install nltk gensim
```

2. Once we've downloaded the dataset, we load it entirely with `pandas`, like this:

```
num_lines = 1000000

data = pd.read_csv('abcnews-date-text.csv.gz',
                   compression='gzip', error_bad_
lines=False,
                   dtype='str', nrows=num_lines)
```

3. The data should look like this:

	publish_date	headline_text
0	20030219	aba decides against community broadcasting lic...
1	20030219	act fire witnesses must be aware of defamation
2	20030219	a g calls for infrastructure protection summit
3	20030219	air nz staff in aust strike for pay rise
4	20030219	air nz strike to affect australian travellers

Figure 6.4 – Viewing the tokenized dataset

4. It's sorted by date, and we shuffle it as a precaution. We then drop the `date` column by running the following code:

```
data = data.sample(frac=1)
data = data.drop(['publish_date'], axis=1)
```

5. We write a function to clean up and process the headlines. First, we get rid of all punctuation signs and digits. Using `nltk`, we also remove stop words—namely, words that are extremely common and don't add any context, such as "this," "any," and so on. In order to reduce the vocabulary size while keeping context, we could apply either **stemming** (<https://en.wikipedia.org/wiki/Stemming>) or **lemmatization** (<https://en.wikipedia.org/wiki/Lemmatisation>), two popular NLP techniques. Let's go with the latter here. Depending on your instance type, this could run for several minutes. Here's the code you'll need:

```
import string
import nltk
from nltk.corpus import stopwords
#from nltk.stem.snowball import SnowballStemmer
from nltk.stem import WordNetLemmatizer
nltk.download('stopwords')
stop_words = stopwords.words('english')
#stemmer = SnowballStemmer("english")
wnl = WordNetLemmatizer()
def process_text(text):
    for p in string.punctuation:
        text = text.replace(p, '')
    text = ''.join([c for c in text if not
                   c.isdigit()])
    text = text.lower().split()
    text = [w for w in text if not w in
            stop_words]
    #text = [stemmer.stem(w) for w in text]
    text = [wnl.lemmatize(w) for w in text]
    return text
data['headline_text'] =
    data['headline_text'].apply(process_text)
```

6. Once processed, the data should look like this:

headline_text	
0	[aba, decides, community, broadcasting, licence]
1	[act, fire, witness, must, aware, defamation]
2	[g, call, infrastructure, protection, summit]
3	[air, nz, staff, aust, strike, pay, rise]
4	[air, nz, strike, affect, australian, traveller]

Figure 6.5 – Viewing the lemmatized dataset

Now the reviews have been tokenized, we need to convert them to a BoW representation, replacing each word with a unique integer ID and its frequency count.

Converting data to a BoW representation

We will convert the reviews into a BoW representation using the following steps:

1. The `gensim` library has exactly what we need! We build a **dictionary**, a list of all words present in the document collection, using the following code:

```
from gensim import corpora
dictionary = corpora.Dictionary(data['headline_text'])
print(dictionary)
```

The dictionary looks like this:

```
Dictionary(83131 unique tokens: ['aba', 'broadcasting',
'community', 'decides', 'licence']...)
```

This number feels very high. If we have too many dimensions, training will be very long, and the algorithm may have trouble fitting the data; for example, NTM is based on a neural network architecture. The input layer will be sized based on the number of tokens, so we need to keep them reasonably low. It will speed up training and help the encoder learn a manageable number of latent features.

2. We could go back and clean the headlines some more. Instead, we use a `gensim` function that removes extreme words—outlier words that are either extremely rare or extremely frequent. Then, taking a bold bet, we decide to restrict the vocabulary to the top 512 remaining words. Yes—that's less than 1%. Here's the code to do this:

```
dictionary.filter_extremes(keep_n=512)
```

3. We write the vocabulary to a text file. Not only does this help us check what the top words are, but we'll also pass this file to the NTM algorithm as an extra **channel**. You'll see why this is important when we train the model. The code to do this is illustrated in the following snippet:

```
with open('vocab.txt', 'w') as f:
    for index in range(0,len(dictionary)) :
        f.write(dictionary.get(index) + '\n')
```

4. We use the dictionary to build a BoW for each headline. It's stored in a new column called `tokens`. When we're done, we drop the text review. The code is illustrated in the following snippet:

```
data['tokens'] = data.apply(lambda row: dictionary.
                           doc2bow(row['headline_text']), axis=1)
data = data.drop(['headline_text'], axis=1)
```

5. The data should now look like this:

	tokens
774398	[(0, 1), (1, 1), (2, 1), (3, 1), (4, 1)]
189893	[(5, 1), (6, 1), (7, 1)]
628809	[(8, 1), (9, 1)]
693184	[(10, 1)]
31959	[(11, 1), (12, 1), (13, 1), (14, 1), (15, 1)]

Figure 6.6 – Viewing the BoW dataset

As you can see, each word has been replaced with its unique ID and its frequency count in the review. For instance, the last line tells us that word #11 is present once, word #12 is present once, and so on.

The data processing is now complete. The last step is to save it to the appropriate input format.

Saving input data

NTM and LDA expect data in either a CSV format or a RecordIO-wrapped protobuf format. Just as with the **factorization matrix** example in *Chapter 4, Training Machine Learning Models*, the data we're working with is quite sparse. Any given review only contains a small number of words from the vocabulary. As CSV is a dense format, we would end up with a huge amount of zero-frequency words. Not a good idea!

Once again, we'll use `lil_matrix`, a **sparse matrix** object available in SciPy. It will have as many lines as we have reviews, and as many columns as we have words in the dictionary. We'll proceed as follows:

1. We create the sparse matrix, like this:

```
from scipy.sparse import lil_matrix
num_lines = data.shape[0]
num_columns = len(dictionary)
token_matrix = lil_matrix((num_lines, num_columns))
    .astype('float32')
```

2. We write a function to add a headline to the matrix. For each token, we simply write its frequency in the appropriate column, as follows:

```
def add_row_to_matrix(line, row):
    for token_id, token_count in row['tokens']:
        token_matrix[line, token_id] = token_count
    return
```

3. We then iterate over headlines and add them to the matrix. Quick note: we can't use row index values, as they might be larger than the number of lines. The code is illustrated in the following snippet:

```
line = 0
for _, row in data.iterrows():
    add_row_to_matrix(line, row)
    line+=1
```

4. The last step is to write this matrix into a memory buffer in protobuf format and upload it to S3 for future use, as follows:

```
import io, boto3
import sagemaker
import sagemaker.amazon.common as smac
buf = io.BytesIO()
smac.write_spmatrix_to_sparse_tensor(buf, token_matrix,
None)
buf.seek(0)
bucket = sagemaker.Session().default_bucket()
prefix = 'headlines-lda-ntm'
```

```

train_key = 'reviews.protobuf'
obj = '{} / {}'.format(prefix, train_key)
s3 = boto3.resource('s3')
s3.Bucket(bucket).Object(obj).upload_fileobj(buf)
s3_train_path = 's3://{} / {}'.format(bucket, obj)

```

- Building the (1000000, 512) matrix takes a few minutes. Once it's been uploaded to S3, we can see that it's only 42 **megabytes (MB)**. Lil' matrix indeed. The code is illustrated in the following snippet:

```

$ aws s3 ls s3://sagemaker-eu-west-1-123456789012/amazon-
reviews-ntm/training.protobuf
43884300 training.protobuf

```

This concludes the data preparation for LDA and NTM. Now, let's see how we can use text datasets prepared with **SageMaker Ground Truth**.

Using datasets labeled with SageMaker Ground Truth

As discussed in *Chapter 2, Handling Data Preparation Techniques*, SageMaker Ground Truth supports text classification tasks. We could definitely use its output to build a dataset for FastText or BlazingText.

First, I ran a quick text classification job on a few sentences, applying one of two labels: "aws_service" if the sentence mentions an AWS service, and "no_aws_service" if it doesn't.

Once the job is complete, I can fetch the **augmented manifest** from S3. It's in **JavaScript Object Notation Lines (JSON Lines)** format, and here's one of its entries:

```
{
  "source": "With great power come great responsibility. The second you create AWS resources, you're responsible for them: security of course, but also cost and scaling. This makes monitoring and alerting all the more important, which is why we built services like Amazon CloudWatch, AWS Config and AWS Systems Manager.", "my-text-classification-job": 0, "my-text-classification-job-metadata": { "confidence": 0.84, "job-name": "labeling-job/my-text-classification-job", "class-name": "aws_service", "human-annotated": "yes", "creation-date": "2020-05-11T12:44:50.620065", "type": "groundtruth/text-classification" }
}
```

Shall we write a bit of Python code to put this in BlazingText format? Of course! Here we go:

1. We load the augmented manifest directly from S3, as follows:

```
import pandas as pd
bucket = 'sagemaker-book'
prefix = 'chapter2/classif/output/my-text-classification-
job/manifests/output'
manifest = 's3://{}//{}//output.manifest'.format(bucket,
prefix)
data = pd.read_json(manifest, lines=True)
```

2. The data looks like this:

	my-text-classification-job	my-text-classification-job-metadata	source
0	0	{'confidence': 0.84, 'job-name': 'labeling-job...}	Since 2006, Amazon Web Services has been striv...
1	0	{'confidence': 0.84, 'job-name': 'labeling-job...}	With great power come great responsibility. Th...
2	1	{'confidence': 0.56, 'job-name': 'labeling-job...}	Still, customers told us that their operations...
3	0	{'confidence': 0.84, 'job-name': 'labeling-job...}	We got to work, and today we're very happy to ...

Figure 6.7 – Viewing the labeled dataset

3. The label is buried in the my-text-classification-job-metadata column. We extract it into a new column, as follows:

```
def get_label(metadata):
    return metadata['class-name']
data['label'] =
    data['my-text-classification-job-metadata'].apply(get_
label)
data = data[['label', 'source']]
```

4. The data now looks like that shown in the following screenshot. From then on, we can apply tokenization, and so on. That was easy, wasn't it?

	label	source
0	aws_service	Since 2006, Amazon Web Services has been striv...
1	aws_service	With great power come great responsibility. Th...
2	no_aws_service	Still, customers told us that their operations...
3	aws_service	We got to work, and today we're very happy to ...

Figure 6.8 – Viewing the processed dataset

Now, let's build NLP models!

Using the built-in algorithms for NLP

In this section, we're going to train and deploy models with BlazingText, LDA, and NTM. Of course, we'll use the datasets prepared in the previous section.

Classifying text with BlazingText

BlazingText makes it extremely easy to build a text classification model, especially if you have no NLP skills. Let's see how, as follows:

1. We upload the training and validation datasets to S3. Alternatively, we could use the output paths returned by a SageMaker Processing job. The code is illustrated in the following snippet:

```
import sagemaker
session = sagemaker.Session()
bucket = session.default_bucket()
prefix = 'amazon-reviews'
s3_train_path = session.upload_data(path='/tmp/training.txt',
                                     bucket=bucket,
                                     key_prefix=prefix+'/input/train')
s3_val_path = session.upload_data(
    path='/tmp/validation.txt',
    bucket=bucket,
    key_prefix=prefix+'/input/validation')
s3_output = 's3://{{}}/{{}}/output/'.format(bucket,
                                             prefix)
```

2. We configure the `Estimator` function for BlazingText, as follows:

```
from sagemaker.image_uris import retrieve
```

```
region_name = session.boto_session.region_name
container = retrieve('blazingtext', region)
bt = sagemaker.estimator.Estimator(container,
    sagemaker.get_execution_role(),
    instance_count=1,
    instance_type='ml.p3.2xlarge',
    output_path=s3_output)
```

3. We set a single hyperparameter, telling BlazingText to train in supervised mode, as follows:

```
bt.set_hyperparameters(mode='supervised')
```

4. We define channels, setting the content type to `text/plain`, and then we launch the training, as follows:

```
from sagemaker import TrainingInput
train_data = TrainingInput(
    s3_train_path, content_type='text/plain')
validation_data = TrainingInput(
    s3_val_path, content_type='text/plain')
s3_channels = {'train': train_data,
                'validation': validation_data}
bt.fit(inputs=s3_channels)
```

5. We get a validation accuracy of 88.4%, which is quite good in the absence of any hyperparameter tweaking. We then deploy the model to a small CPU instance, as follows:

```
bt_predictor = bt.deploy(initial_instance_count=1,
                         instance_type='ml.t2.medium')
```

6. Once the endpoint is up, we send three tokenized samples for prediction, asking for all three labels, as follows:

```
import json
sentences = ['This is a bad camera it doesnt work at all
, i want a refund . ' , 'The camera works , the pictures
are decent quality, nothing special to say about it . ' ,
'Very happy to have bought this , exactly what I needed .
']
```

```

payload = {"instances":sentences,
           "configuration":{"k": 3}}
bt_predictor.serializer =
    sagemaker.serializers.JSONSerializer()

response = bt_predictor.predict(json.dumps(payload))

```

7. Printing the response, we see that the three samples were correctly categorized, as illustrated here:

```

[{'prob': [0.9758228063583374, 0.023583529517054558,
0.0006236258195713162], 'label': ['__label__negative__',
 '__label__neutral__', '__label__positive__']},
 {'prob': [0.5177792906761169, 0.2864232063293457,
0.19582746922969818], 'label': ['__label__neutral__',
 '__label__positive__', '__label__negative__']},
 {'prob': [0.9997835755348206, 0.000205090589588508,
4.133415131946094e-05], 'label': ['__label__positive__',
 '__label__neutral__', '__label__negative__']}]

```

8. As usual, we delete the endpoint once we're done by running the following code:

```
bt_predictor.delete_endpoint()
```

Now, let's train BlazingText to compute word vectors.

Computing word vectors with BlazingText

The code is almost identical to the previous example, with only two differences. First, there is only one channel, containing training data. Second, we need to set BlazingText to UL mode.

BlazingText supports the training modes implemented in Word2Vec: **skipgram** and **continuous BoW (CBOW)**. It adds a third mode, **batch_skipgram**, for faster distributed training. It also supports **subword embeddings**, a technique that makes it possible to return a word vector for words that are misspelled or not part of the vocabulary.

Let's go for skipgram with subword embeddings. We leave the dimension of vectors unchanged (the default is 100). Here's the code you'll need:

```
bt.set_hyperparameters(mode='skipgram', subwords=True)
```

Unlike other algorithms, there is nothing to deploy here. The model artifact is in S3 and can be used for downstream NLP applications.

Speaking of which, BlazingText is compatible with FastText, so how about trying to load the models we just trained into FastText?

Using BlazingText models with FastText

First, we need to compile FastText, which is extremely simple. You can even do it on a notebook instance without having to install anything. Here's the code you'll need:

```
$ git clone https://github.com/facebookresearch/fastText.git  
$ cd fastText  
$ make
```

Let's first try our classification model.

Using a BlazingText classification model with FastText

We will try the model using the following steps:

1. We copy the model artifact from S3 and extract it as follows:

```
$ aws s3 cp s3://sagemaker-eu-west-1-123456789012/amazon-  
reviews/output/JOB_NAME/output/model.tar.gz .  
$ tar xvfz model.tar.gz
```

2. We load model.bin with FastText, as follows:

```
$ ./fasttext predict model.bin -
```

3. We predict samples and view their top class, as follows:

```
This is a bad camera it doesnt work at all , i want a  
refund .  
__label__negative__  
The camera works , the pictures are decent quality,  
nothing  
special to say about it .  
__label__neutral__  
Very happy to have bought this , exactly what I needed  
__label__positive__
```

We exit with *Ctrl + C*. Now, let's explore our vectors.

Using BlazingText word vectors with FastText

We will now use FastText with the vectors, as follows:

1. We copy the model artifact from S3 and we extract it, like this:

```
$ aws s3 cp s3://sagemaker-eu-west-1-123456789012/amazon-reviews-word2vec/output/JOB_NAME/output/model.tar.gz .
$ tar xvfz model.tar.gz
```

2. We can explore word similarities. For example, let's look for words that are closest to "telephoto". This could help us improve how we handle search queries or how we recommend similar products. Here's the code you'll need:

```
$ ./fasttext nn vectors.bin
Query word? Telephoto
telephotos 0.951023
75-300mm 0.79659
55-300mm 0.788019
18-300mm 0.782396
. . .
```

3. We can also look for analogies. For example, let's ask our model the following question: What's the Canon equivalent for the Nikon D3300 camera? The code is illustrated in the following snippet:

```
$ ./fasttext analogies vectors.bin
Query triplet (A - B + C)? nikon d3300 canon
xsi 0.748873
700d 0.744358
100d 0.735871
```

According to our model, you should consider the XSI and 700D cameras!

As you can see, word vectors are amazing and BlazingText makes it easy to compute them at any scale. Now, let's move on to topic modeling, another fascinating subject.

Modeling topics with LDA

In a previous section, we prepared a million news headlines, and we're now going to use them for topic modeling with LDA, as follows:

1. First, we define useful paths by running the following code:

```
import sagemaker
session = sagemaker.Session()
bucket = session.default_bucket()
prefix = 'reviews-lda-ntm'
train_key = 'reviews.protobuf'
obj = '{}/{}'.format(prefix, train_key)
s3_train_path = 's3://{}{}'.format(bucket, obj)
s3_output = 's3://{}{}/output/'.format(bucket, prefix)
```

2. We configure the `Estimator` function, like this:

```
from sagemaker.image_uris import retrieve
region_name = session.boto_session.region_name
container = retrieve('lda', region)

lda = sagemaker.estimator.Estimator(container,
                                      role=sagemaker.get_execution_role(),
                                      instance_count=1,
                                      instance_type='ml.c5.2xlarge',
                                      output_path=s3_output)
```

3. We set hyperparameters: how many topics we want to build (10), how many dimensions the problem has (the vocabulary size), and how many samples we're training on. Optionally, we can set a parameter named `alpha0`. According to the documentation: "*Small values are more likely to generate sparse topic mixtures and large values (greater than 1.0) produce more uniform mixtures.*" Let's set it to 0.1 and hope that the algorithm can indeed build well-identified topics. Here's the code you'll need:

```
lda.set_hyperparameters(num_topics=5,
                       feature_dim=len(dictionary),
                       mini_batch_size=num_lines,
                       alpha0=0.1)
```

4. We launch the training. As RecordIO is the default format expected by the algorithm, we don't need to define channels. The code is illustrated in the following snippet:

```
lda.fit(inputs={'train': s3_train_path})
```

5. Once training is complete, we deploy to a small CPU instance, as follows:

```
lda_predictor = lda.deploy(  
    initial_instance_count=1,  
    instance_type='ml.t2.medium')
```

6. Before we send samples for prediction, we need to process them just like we processed the training set. We write a function that takes care of this: building a sparse matrix, filling it with BoW, and saving to an in-memory protobuf buffer, as follows:

```
def process_samples(samples, dictionary):  
    num_lines = len(samples)  
    num_columns = len(dictionary)  
    sample_matrix = lil_matrix((num_lines,  
                                num_columns)).astype('float32')  
    for line in range(0, num_lines):  
        s = samples[line]  
        s = process_text(s)  
        s = dictionary.doc2bow(s)  
        for token_id, token_count in s:  
            sample_matrix[line, token_id] = token_count  
        line+=1  
    buf = io.BytesIO()  
    smac.write_spmatrix_to_sparse_tensor(  
        buf,  
        sample_matrix,  
        None)  
    buf.seek(0)  
    return buf
```

Please note that we need the dictionary here. This is why the corresponding SageMaker Processing job saved a pickled version of it, which we could later unpickle and use.

7. Then, we define a Python array containing five headlines, named `samples`. These are real headlines I copied from the ABC news website at <https://www.abc.net.au/news/>. The code is illustrated in the following snippet:

```
samples = [ "Major tariffs expected to end Australian  
barley trade to China", "Satellite imagery sparks more  
speculation on North Korean leader Kim Jong-un", "Fifty  
trains out of service as fault forces Adelaide passengers  
to 'pack like sardines", "Germany's Bundesliga plans its  
return from lockdown as football world watches", "All AFL  
players to face COVID-19 testing before training resumes"  
]
```

8. Let's process and predict them, as follows:

```
lda_predictor.serializer =  
    sagemaker.serializers.CSVSerializer()  
response = lda_predictor.predict(  
    process_samples(samples, dictionary))  
print(response)
```

9. The response contains a score vector for each review (extra decimals have been removed for brevity). Each vector reflects a mix of topics, with a score per topic. All scores add up to 1. The code is illustrated in the following snippet:

```
{'predictions': [  
    {'topic_mixture': [0,0.22,0.54,0.23,0,0,0,0,0,0]},  
    {'topic_mixture': [0.51,0.49,0,0,0,0,0,0,0,0]}, {'topic_  
    mixture': [0.38,0,0.22,0,0.40,0,0,0,0,0]}, {'topic_  
    mixture': [0.38,0.62,0,0,0,0,0,0,0,0]}, {'topic_mixture':  
    [0,0.75,0,0,0,0,0,0.25,0,0]}]}
```

10. This isn't easy to read. Let's print the top topic and its score, as follows:

```
import numpy as np  
vecs = [r['topic_mixture'] for r in  
response['predictions']]  
for v in vecs:  
    top_topic = np.argmax(v)  
    print("topic %s, %2.2f"%(top_topic,v[top_topic]))
```

This prints out the following result:

```
topic 2, 0.54
topic 0, 0.51
topic 4, 0.40
topic 1, 0.62
topic 1, 0.75
```

11. As usual, we delete the endpoint once we're done, as follows:

```
lda_predictor.delete_endpoint()
```

Interpreting LDA results is not easy, so let's be careful here. No wishful thinking!

- We see that each headline has a definite topic, which is good news. Apparently, LDA was able to identify solid topics, maybe thanks to the low `alpha0` value.
- The top topics for unrelated headlines are different, which is promising.
- The last two headlines are both about sports and their top topic is the same, which is another good sign.
- All five reviews scored zero on topics 5, 6, 8, and 9. This probably means that other topics have been built, and we would need to run more examples to discover them.

Is this a successful model? Probably. Can we be confident that topic 0 is about world affairs, topic 1 about sports, and topic 2 about sports? Not until we've predicted a few thousand more reviews and checked that related headlines are assigned to the same topic.

As mentioned at the beginning of the chapter, LDA is not a classification algorithm. It has a mind of its own and it may build totally unexpected topics. Maybe it will group headlines according to sentiment or city names. It all depends on the distribution of these words inside the document collection.

Wouldn't it be nice if we could see which words "weigh" more in a certain topic? That would certainly help us understand the topics a little better. Enter NTM!

Modeling topics with NTM

This example is very similar to the previous one. We'll just highlight the differences, and you'll find a full example in the GitHub repository for the book. Let's get into it, as follows:

1. We upload the **vocabulary file** to S3, like this:

```
s3_auxiliary_path =  
    session.upload_data(path='vocab.txt',  
    key_prefix=prefix + '/input/auxiliary')
```

2. We select the NTM algorithm, as follows:

```
from sagemaker.image_uris import retrieve  
region_name = session.boto_session.region_name  
container = retrieve('ntm', region)
```

3. Once we've configured the `Estimator` function, we set the hyperparameters, as follows:

```
ntm.set_hyperparameters(num_topics=10,  
    feature_dim=len(dictionary),  
    optimizer='adam',  
    mini_batch_size=256,  
    num_patience_epochs=10)
```

4. We launch training, passing the vocabulary file in the auxiliary channel, as follows:

```
ntm.fit(inputs={'train': s3_training_path,  
    'auxiliary': s3_auxiliary_path})
```

When training is complete, we see plenty of information in the training log. First, we see the average WETC and TU scores for the 10 topics, as follows:

```
(num_topics:10) [wetc 0.42, tu 0.86]
```

These are decent results. Topic unicity is high, and the semantic distance between topic words is average.

For each topic, we see its WETC and TU scores, as well as its top words—that is to say, the words that have the highest probability of appearing in documents associated with this topic.

Let's look at each one in detail and try to put names to topics.

Topic 0 is pretty obvious, I think. Almost all words are related to crime, so let's call it crime. You can see this topic here:

```
[0.51, 0.84] stabbing charged guilty pleads murder fatal man  
assault bail jailed alleged shooting arrested teen girl accused  
boy car found crash
```

The following topic 1 is a little fuzzier. How about legal? Have a look at it here:

```
[0.36, 0.85] seeker asylum climate live front hears change  
export carbon tax court wind challenge told accused rule legal  
face stand boat
```

Topic 2 is about accidents and fires. Let's call it disaster. You can see the topic here:

```
[0.39, 0.78] seeker crew hour asylum cause damage truck country  
firefighter blaze crash warning ta plane near highway accident  
one fire fatal
```

Topic 3 is obvious: sports. The TU score is the highest, showing that sports articles use a very specific vocabulary found nowhere else, as we can see here:

```
[0.54, 0.93] cup world v league one match win title final star  
live victory england day nrl miss beat team afl player
```

Topic 4 is a strange mix of weather information and natural resources. It has the lowest WETC and the lowest TU score too. Let's call it unknown1. Have a look at it here:

```
[0.35, 0.77] coast korea gold north east central pleads west  
south guilty queensland found qld rain beach cyclone northern  
nuclear crop mine
```

Topic 5 is about world affairs, it seems. Let's call it international. You can see the topic here:

```
[0.38, 0.88] iraq troop bomb trade korea nuclear kill soldier  
iraqi blast pm president china pakistan howard visit pacific u  
abc anti
```

Topic 6 feels like local news, as it contains abbreviations for Australian regions: qld is Queensland, ta is Tasmania, nsw is New South Wales, and so on. Let's call it local1. The topic is shown here:

```
[0.25, 0.88] news hour country rural national abc ta sport vic  
abuse sa nsw weather nt club qld award business
```

Topic 7 is a no-brainer: finance. It has the highest WETC score, showing that its words are closely related from a semantic point of view. Topic unicity is also very high, and we would probably see the same for domain-specific topics on medicine or engineering. Have a look at the topic here:

```
[0.62, 0.90] share dollar rise rate market fall profit price  
interest toll record export bank despite drop loss post high  
strong trade
```

Topic 8 is about politics, with a bit of crime thrown in. Some people would say that's actually the same thing. As we already have a crime topic, we'll name this one politics. Have a look at the topic here:

```
[0.41, 0.90] issue election vote league hunt interest poll  
parliament gun investigate opposition raid arrest police  
candidate victoria house northern crime rate
```

Topic 9 is another mixed bag. It's hard to say whether it's about farming or missing people! Let's go with unknown2. You can see the topic here:

```
[0.37, 0.84] missing search crop body found wind rain continues  
speaks john drought farm farmer smith pacific crew river find  
mark tourist
```

All things considered, that's a pretty good model: 8 clear topics out of 10.

Let's define our list of topics and run our sample headlines through the model after deploying it, as follows:

```
topics = ['crime', 'legal', 'disaster', 'sports', 'unknown1',  
         'international', 'local', 'finance', 'politics',  
         'unknown2']  
  
samples = [ "Major tariffs expected to end Australian barley  
trade to China", "US woman wanted over fatal crash asks for  
release after coronavirus halts extradition", "Fifty trains out  
of service as fault forces Adelaide passengers to 'pack like  
sardines", "Germany's Bundesliga plans its return from lockdown  
as football world watches", "All AFL players to face COVID-19  
testing before training resumes" ]
```

We use the following function to print the top three topics and their score:

```
import numpy as np
for r in response['predictions']:
    sorted_indexes = np.argsort(r['topic_weights']).tolist()
    sorted_indexes.reverse()
    top_topics = [topics[i] for i in sorted_indexes]
    top_weights = [r['topic_weights'][i]
                   for i in sorted_indexes]
    pairs = list(zip(top_topics, top_weights))
    print(pairs[:3])
```

Here's the output:

```
[('finance', 0.30), ('international', 0.22), ('sports', 0.09)]
[('unknown1', 0.19), ('legal', 0.15), ('politics', 0.14)]
[['crime', 0.32], ('legal', 0.18), ('international', 0.09)]
[['sports', 0.28], ('unknown1', 0.09), ('unknown2', 0.08)]
[['sports', 0.27], ('disaster', 0.12), ('crime', 0.11)]
```

Headlines 0, 2, 3, and 4 are right on target. That's not surprising given how strong these topics are.

Headline 1 scores very high on the topic we called `legal`. Maybe Adelaide passengers should sue the train company? Seriously, we would need to find other matching headlines to get a better sense of what the topic is really about.

As you can see, NTM makes it easier to understand what topics are about. We could improve the model by processing the vocabulary file, adding or removing specific words to influence topics, increasing the number of topics, fiddling with `alpha0`, and so on. My intuition tells me that we should really see a "weather" topic in there. Please experiment and see if you want to make it appear.

If you'd like to run another example, you'll find interesting techniques in this notebook:

[https://github.com/awslabs/amazon-sagemaker-examples/
blob/master/introduction_to_applying_machine_learning/
ntm_20newsgroups_topic_modeling/ntm_20newsgroups_topic_model.
ipynb](https://github.com/awslabs/amazon-sagemaker-examples/blob/master/introduction_to_applying_machine_learning/ntm_20newsgroups_topic_modeling/ntm_20newsgroups_topic_model.ipynb)

Summary

NLP is a very exciting topic. It's also a difficult one because of the complexity of language in general, and due to how much processing is required to build datasets. Having said that, the built-in algorithms in SageMaker will help you get good results out of the box. Training and deploying models are straightforward processes, which leaves you more time to explore, understand, and prepare data.

In this chapter, you learned about the BlazingText, LDA, and NTM algorithms. You also learned how to process datasets using popular open source tools such as `nltk`, `spaCy`, and `gensim`, and how to save them in the appropriate format. Finally, you learned how to use the SageMaker SDK to train and deploy models with all three algorithms, as well as how to interpret results. This concludes our exploration of built-in algorithms.

In the next chapter, you will learn how to use built-in ML frameworks such as **scikit-learn**, **TensorFlow**, **PyTorch**, and **Apache MXNet**.

7

Extending Machine Learning Services Using Built-In Frameworks

In the last three chapters, you learned how to use built-in algorithms to train and deploy models without having to write a line of machine learning code. However, these algorithms don't cover the full spectrum of machine learning problems. In a lot of cases, you'll need to write your own code. Thankfully, several open source frameworks make this reasonably easy.

In this chapter, you will learn how to train and deploy models with the most popular open source frameworks for machine learning and deep learning. We will cover the following topics:

- Discovering the built-in frameworks in Amazon SageMaker
- Running your framework code on Amazon SageMaker
- Using the built-in frameworks

Let's get started!

Technical requirements

You will need an AWS account to run the examples included in this chapter. If you haven't got one already, please point your browser to <https://aws.amazon.com/getting-started/> to create one. You should also familiarize yourself with the AWS Free Tier (<https://aws.amazon.com/free/>), which lets you use many AWS services for free within certain usage limits.

You will need to install and configure the AWS command-line interface for your account (<https://aws.amazon.com/cli/>).

You will need a working Python 3.x environment. Installing the Anaconda distribution (<https://www.anaconda.com/>) is not mandatory but strongly encouraged, as it includes many projects that we will need (Jupyter, pandas, numpy, and more).

You will need a working Docker installation. You can find installation instructions and the necessary documentation at <https://docs.docker.com>.

The code examples included in the book are available on GitHub at <https://github.com/PacktPublishing/Learn-Amazon-SageMaker-second-edition>. You will need to install a Git client to access them (<https://git-scm.com/>).

Discovering the built-in frameworks in Amazon SageMaker

SageMaker lets you train and deploy your models with the following machine learning and deep learning frameworks:

- **Scikit-learn**, undoubtedly the most widely used open source library for machine learning. If you're new to this topic, start here: <https://scikit-learn.org>.
- **XGBoost**, an extremely popular and versatile open source algorithm for regression, classification, and ranking problems (<https://xgboost.ai>). It's also available as a built-in algorithm, as presented in *Chapter 4, Training Machine Learning Models*. Using it in framework mode will give us more flexibility.
- **TensorFlow**, an extremely popular open source library for deep learning (<https://www.tensorflow.org>). SageMaker also supports the lovable **Keras** API (<https://keras.io>).
- **PyTorch**, another highly popular open source library for deep learning (<https://pytorch.org>). Researchers, in particular, enjoy its flexibility.

- **Apache MXNet**, an interesting challenger for deep learning. Natively implemented in C++, it's often faster and more scalable than its competitors. Its **Gluon** API provides rich toolkits for computer vision (<https://gluon-cv.mxnet.io>), **Natural Language Processing (NLP)** (<https://gluon-nlp.mxnet.io>), and time series data (<https://gluon-ts.mxnet.io>).
- **Chainer**, another worthy challenger for deep learning (<https://chainer.org>).
- **Hugging Face**, the most popular collection of state-of-the-art tools and models for NLP (<https://huggingface.co>).
- Frameworks for **reinforcement learning**, such as **Intel Coach**, **Ray RLlib**, and **Vowpal Wabbit**. I won't discuss this topic here as it could take up another book!
- **Spark**, thanks to a dedicated SDK that lets you train and deploy models directly from your Spark application using either **PySpark** or **Scala** (<https://github.com/aws/sagemaker-spark>).

You'll find plenty of examples of all of these at <https://github.com/awslabs/amazon-sagemaker-examples/tree/master/sagemaker-python-sdk>.

In this chapter, we'll focus on the most popular ones: XGBoost, scikit-learn, TensorFlow, PyTorch, and Spark.

The best way to get started is to run a first simple example. As you will see, the workflow is the same as for built-in algorithms. We'll highlight a few differences along the way, which we'll dive into later in this chapter.

Running a first example with XGBoost

In this example, we'll build a binary classification model with the XGBoost built-in framework. At the time of writing, the latest version supported by SageMaker is 1.3-1.

We'll use our own training script based on the `xgboost.XGBClassifier` object and the Direct Marketing dataset, which we used in *Chapter 3, AutoML with Amazon SageMaker Autopilot*:

1. First, we download and extract the dataset:

```
%%sh
wget -N https://sagemaker-sample-data-us-west-2.s3-us-
west-2.amazonaws.com/autopilot/direct_marketing/bank-
additional.zip
unzip -o bank-additional.zip
```

2. We import the SageMaker SDK and define an S3 prefix for the job:

```
import sagemaker
sess = sagemaker.Session()
bucket = sess.default_bucket()
prefix = 'xgboost-direct-marketing'
```

3. We load the dataset and apply very basic processing (as it's not our focus here). Simply one-hot encode the categorical features, move the labels to the first column (an XGBoost requirement), shuffle the dataset, split it for training and validation, and save the results in two separate CSV files:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
data = pd.read_csv('./bank-additional/bank-additional-full.csv')
data = pd.get_dummies(data)
data = data.drop(['y_no'], axis=1)
data = pd.concat([data['y_yes'],
                  data.drop(['y_yes'], axis=1)],
                 axis=1)
data = data.sample(frac=1, random_state=123)
train_data, val_data = train_test_split(
    data, test_size=0.05)
train_data.to_csv(
    'training.csv', index=False, header=False)
val_data.to_csv(
    'validation.csv', index=False, header=False)
```

4. We upload the two files to S3:

```
training = sess.upload_data(path='training.csv',
                            key_prefix=prefix + '/training')
validation = sess.upload_data(path='validation.csv',
                             key_prefix=prefix + "/validation")
output = 's3://{{}}/{{}}/output/'.format(bucket,prefix)
```

5. We define two inputs, with data in CSV format:

```
from sagemaker import TrainingInput
train_input = TrainingInput(
    training_path, content_type='text/csv')
val_input = TrainingInput(
    validation_path, content_type='text/csv')
```

6. Define an estimator for the training job. Of course, we could use the generic `Estimator` object and pass the name of the XGBoost container hosted in **Amazon ECR**. Instead, we use the `XGBoost` estimator, which automatically selects the right container:

```
from sagemaker.xgboost import XGBoost
xgb_estimator = XGBoost(
    role= sagemaker.get_execution_role(),
    entry_point='xgb-dm.py',
    instance_count=1,
    instance_type='ml.m5.large',
    framework_version='1.2-2',
    output_path=output,
    hyperparameters={
        'num_round': 100,
        'early_stopping_rounds': 10,
        'max-depth': 5,
        'eval-metric': 'auc'}
)
```

Several parameters are familiar here: the role, the infrastructure requirements, and the output path. What about the other ones? `entry_point` is the path of our training script (available in the GitHub repository for this book). `hyperparameters` is passed to the training script. We also have to select a `framework_version` value; this is the version of XGBoost that we want to use.

7. We train as usual:

```
xgb_estimator.fit({'train':training,
                    'validation':validation})
```

8. We also deploy as usual, creating a unique endpoint name:

```
from time import strftime, gmtime
xgb_endpoint_name =
    prefix+strftime("%Y-%m-%d-%H-%M-%S", gmtime())
xgb_predictor = xgb_estimator.deploy(
    endpoint_name=xgb_endpoint_name,
    initial_instance_count=1,
    instance_type='ml.t2.medium')
```

Then, we load a few samples from the validation set and send them for prediction in CSV format. The response contains a score between 0 and 1 for each sample:

```
payload = val_data[:10].drop(['y_yes'], axis=1)
payload = payload.to_csv(header=False,
                         index=False).rstrip('\n')
xgb_predictor.serializer =
    sagemaker.serializers.CSVSerializer()
xgb_predictor.deserializer =
    sagemaker.deserializers.CSVDeserializer()
response = xgb_predictor.predict(payload)
print(response)
```

This prints out the following probabilities:

```
[[[0.07206538], [0.02661967], [0.16043524],
  [4.026455e-05], [0.0002120432], [0.52123886],
  [0.50755614], [0.00015006188], [3.1439096e-05],
  [9.7614546e-05]]]
```

9. When we're done, we delete the endpoint:

```
xgb_predictor.delete_endpoint()
```

We used XGBoost here, but the workflow would be identical for another framework. This standard way of training and deploying makes it really easy to switch from built-in algorithms to frameworks, or from one framework to the next.

The points that we need to focus on here are as follows:

- **Framework containers:** What are they? Can we see how they're built? Can we customize them? Can we use them to train on our local machine?

- **Training:** How does a SageMaker training script differ from vanilla framework code? How does it receive hyperparameters? How should it read input data? Where should it save the model?
- **Deploying:** How is the model deployed? Should the script provide some code for this? What's the input format for prediction?
- **Managing dependencies:** Can we add additional source files besides the `entry_point` script? Can we add libraries for training and deployment?

All these questions will be answered now!

Working with framework containers

SageMaker contains a training and inference container for each built-in framework, and they are updated regularly to the latest versions. Different containers are also available for CPU and GPU instances. All these containers are collectively known as **Deep Learning Containers** (<https://aws.amazon.com/machine-learning/containers>).

As we saw in the previous example, they let you use your own code without having to maintain bespoke containers. In most cases, you won't need to look any further, and you can happily forget that these containers even exist. If this topic feels too advanced for now, feel free to skip it for now, and move on to the *Training and deploying locally* section.

If you're curious or have custom requirements, you'll be happy to learn that the code for these containers is open source:

- **Scikit-learn:** <https://github.com/aws/sagemaker-scikit-learn-container>
- **XGBoost:** <https://github.com/aws/sagemaker-xgboost-container>
- **TensorFlow, PyTorch, Apache MXNet, and Hugging Face:** <https://github.com/aws/deep-learning-containers>
- **Chainer:** <https://github.com/aws/sagemaker-chainer-container>

For starters, this lets you understand how these containers are built and how SageMaker trains and predicts with them. You could also do the following:

- Build and run them on your local machine for local experimentation.
- Build and run them on your favorite managed Docker service, such as **Amazon ECS**, **Amazon EKS**, or **Amazon Fargate** (<https://aws.amazon.com/containers>).

- Customize them, push them to Amazon ECR, and use them with the estimators present in the SageMaker SDK. We'll demonstrate this in *Chapter 8, Using Your Algorithms and Code*.

These containers have another nice property. You can use them with the SageMaker SDK to train and deploy models on your local machine. Let's see how this works.

Training and deploying locally

Local mode is the ability to train and deploy models with the SageMaker SDK without firing up on-demand managed infrastructure in AWS. You use your local machine instead. In this context, "local" means the machine running the notebook: it could be your laptop, a local server, or a small **notebook instance**.

Note

At the time of writing, local mode is not available in SageMaker Studio.

This is an excellent way to quickly experiment and iterate on a small dataset. You won't have to wait for instances to come up, and you won't have to pay for them either!

Let's revisit our previous XGBoost example, highlighting the changes required to use local mode:

1. Explicitly set the name of the IAM role. `get_execution_role()` does not work on your local machine (it does on a notebook instance):

```
#role = sagemaker.get_execution_role()  
role = 'arn:aws:iam::0123456789012:role/Sagemaker-  
fullaccess'
```

2. Load the training and validation datasets from local files. Store the model locally in `/tmp`:

```
training = 'file://training.csv'  
validation = 'file://validation.csv'  
output = 'file:///tmp'
```

3. In the XGBoost estimator, set `instance_type` to `local`. For local GPU training, we would use `local_gpu`.
4. In `xgb_estimator.deploy()`, set `instance_type` to `local`.

That's all it takes to train on your local machine using the same container you would use at scale on AWS. This container will be pulled once to your local machine and you'll be using it from then on. When you're ready to train at scale, just replace the `local` or `local_gpu` instance type with the appropriate AWS instance type and you're good to go.

Troubleshooting

If you see strange deployment errors, try restarting Docker (`sudo service docker restart`). I found that it doesn't like being interrupted during deployment, which it tends to do a lot when working inside Jupyter Notebooks!

Now, let's see what it takes to run our own code inside these containers. This feature is called **script mode**.

Training with script mode

Since your training code runs inside a SageMaker container, it needs to be able to do the following:

- Receive hyperparameters passed to the estimator.
- Read data available in input channels (training, validation, and more).
- Save the trained model in the right place.

Script mode is how SageMaker makes this possible. The name comes from the way your code is invoked in the container. Looking at the training log for our XGBoost job, we see this:

Invoking script with the following command:

```
/miniconda3/bin/python3 -m xgb-dm --early-stopping-rounds 10  
--eval-metric auc --max-depth 5
```

Our code is invoked like a plain Python script (hence the name script mode). We can see that hyperparameters are passed as command-line arguments, which answers the question of what we should use inside the script to read them: `argparse`.

Here's the corresponding code snippet in our script:

```
parser = argparse.ArgumentParser()
parser.add_argument('--max-depth', type=int, default=4)
parser.add_argument('--early-stopping-rounds', type=int,
                    default=10)
parser.add_argument('--eval-metric', type=str,
                    default='error')
```

What about the location of the input data and the saved model? If we look at the log a little more closely, we'll see this:

```
SM_CHANNEL_TRAIN=/opt/ml/input/data/train
SM_CHANNEL_VALIDATION=/opt/ml/input/data/validation
SM_MODEL_DIR=/opt/ml/model
```

These three environment variables define **local paths inside the container**, pointing to the respective locations for the training data, validation data, and the saved model. Does this mean we have to manually copy the datasets and the model from and to S3? No! SageMaker takes care of all this automatically for us. This is part of the support code present in the container.

Our script only needs to read these variables. I recommend using `argparse` again, as this will let us pass the paths to our script when we train outside of SageMaker (more on this soon).

Here's the corresponding code snippet in our script:

```
parser.add_argument('--model-dir', type=str,
                    default=os.environ['SM_MODEL_DIR'])
parser.add_argument('--training-dir', type=str,
                    default=os.environ['SM_CHANNEL_TRAIN'])
parser.add_argument('--validation', type=str,
                    default=os.environ['SM_CHANNEL_VALIDATION'])
```

Channel names

The `SM_CHANNEL_xxx` variables are named according to the channels passed to `fit()`. For instance, if your algorithm required a channel named `foobar`, you'd name it `foobar` in `fit()` and `SM_CHANNEL_FOOBAR` in your script. In your container, the data for that channel would automatically be available in `/opt/ml/input/data/foobar`.

To sum things up, in order to train framework code on SageMaker, we only need to do the following:

1. Use `argparse` to read hyperparameters passed as command-line arguments.
Chances are you're already doing this in your code anyway!
2. Read the `SM_CHANNEL_xxx` environment variables and load data from there.
3. Read the `SM_MODEL_DIR` environment variable and save the trained model there.

Now, let's talk about deploying models trained in script mode.

Understanding model deployment

In general, your script needs to include the following:

- A function to load the model
- A function to process input data before it's passed to the model
- A function to process predictions before they're returned to the caller

The amount of actual work required depends on the framework and the input format you use. Let's see what this means for TensorFlow, PyTorch, MXNet, XGBoost, and scikit-learn.

Deploying with TensorFlow

The TensorFlow inference container relies on the **TensorFlow Serving** model server for model deployment (<https://www.tensorflow.org/tfx/guide/serving>). For this reason, your training code must save the model in this format. Model loading and prediction are available automatically.

JSON is the default input format for prediction, and it also works for numpy arrays thanks to automatic serialization. JSON Lines and CSV are also supported. For other formats, you can implement your own preprocessing and postprocessing functions, `input_handler()` and `output_handler()`. You'll find more information at https://sagemaker.readthedocs.io/en/stable/using_tf.html#deploying-from-an-estimator.

You can also dive deeper into the TensorFlow inference container at <https://github.com/aws/deep-learning-containers/tree/master/tensorflow/inference>.

Deploying with PyTorch

The PyTorch inference container relies on the **TorchServe** model server (<https://pytorch.org/serve>). Models are loaded automatically. Prediction is automatically available if they implement the `__call__()` method. If not, you should provide a `predict_fn()` function in the inference script.

For prediction, numpy is the default input format. JSON Lines and CSV are also supported. For other formats, you can implement your own preprocessing and postprocessing functions. You'll find more information at https://sagemaker.readthedocs.io/en/stable/frameworks/pytorch/using_pytorch.html#serve-a-pytorch-model.

You can dive deeper into the PyTorch inference container at <https://github.com/aws/deep-learning-containers/tree/master/pytorch/inference>.

Deploying with Apache MXNet

The Apache MXNet inference container relies on **Multi-Model Server (MMS)** for model deployment (<https://github.com/awslabs/multi-model-server>). It uses the default MXNet model format.

Models based on the `Module` API do not require a model loading function. For prediction, they support data in JSON, CSV, or numpy format.

Gluon models do require a model loading function as parameters need to be explicitly initialized. Data can be sent in JSON or numpy format.

For other data formats, you can implement your own preprocessing, prediction, and postprocessing functions. You can find more information at https://sagemaker.readthedocs.io/en/stable/using_mxnet.html.

You can dive deeper into the MXNet inference container at <https://github.com/aws/deep-learning-containers/tree/master/mxnet/inference/docker>.

Deploying XGBoost and scikit-learn

Likewise, XGBoost and scikit-learn rely on <https://github.com/aws/sagemaker-xgboost-container> and <https://github.com/aws/sagemaker-scikit-learn-container>, respectively.

Your script needs to provide the following:

- A **mandatory** `model_fn()` function to load the model. Just like for training, the location of the model to load is passed in the `SM_MODEL_DIR` environment variable.

- Two optional functions to deserialize and serialize prediction data, named `input_fn()` and `output_fn()`. These functions are only required if you need another input format other than JSON, CSV, or numpy.
- An optional `predict_fn()` function passes deserialized data to the model and returns a prediction. This is only required if you need to preprocess data before predicting it, or to postprocess predictions.

For XGBoost and scikit-learn, the `model_fn()` function is extremely simple and quite generic. Here are a couple of examples that should work in most cases:

```
# Scikit-learn
def model_fn(model_dir):
    clf = joblib.load(os.path.join(model_dir,
                                   'model.joblib'))
    return clf

# XGBoost
def model_fn(model_dir):
    model = xgb.Booster()
    model.load_model(os.path.join(model_dir, 'xgb.model'))
    return model
```

SageMaker also lets you import and export models. You can upload an existing model to S3 and deploy it directly on SageMaker. Likewise, you can copy a trained model from S3 and deploy it elsewhere. We'll look at this in detail in *Chapter 11, Deploying Machine Learning Models*.

Now, let's talk about training and deployment dependencies.

Managing dependencies

In many cases, you'll need to add extra source files and libraries to the framework's containers. Let's see how we can easily do this.

Adding source files for training

By default, all estimators load the entry point script from the current directory. If you need additional source files for training, estimators let you pass a `source_dir` parameter, which points at the directory storing the extra files. Please note that the entry point script must be in the same directory.

In the following example, `myscript.py` and all additional source files must be placed in the `src` directory. SageMaker will automatically package the directory and copy it inside the training container:

```
sk = SKLearn(entry_point='myscript.py',
              source_dir='src',
              . . .
```

Adding libraries for training

You can use different techniques to add libraries that are required for training.

For libraries that can be installed with `pip`, the simplest technique is to add a `requirements.txt` file in the same folder as the entry point script. SageMaker will automatically install these libraries inside the container.

Alternatively, you can use `pip` to install libraries directly in the training script by issuing a `pip install` command. We used this in *Chapter 6, Training Natural Language Processing Models*, with LDA and NTM. This is useful when you don't want to or cannot modify the SageMaker code that launches the training job:

```
import subprocess, sys
def install(package):
    subprocess.call([sys.executable, "-m",
                    "pip", "install", package])
if __name__=='__main__':
    install('gensim')
    import gensim
. . .
```

For libraries that can't be installed with `pip`, you should use the `dependencies` parameter. It's available in all estimators, and it lets you list libraries to add to the training job. These libraries need to be present locally, in a virtual environment or a bespoke directory. SageMaker will package them and copy them inside the training container.

In the following example, `myscript.py` needs the `mylib` library. We install it in the `lib` local directory:

```
$ mkdir lib
$ pip install mylib -t lib
```

Then, we pass its location to the estimator:

```
sk = SKLearn(entry_point='myscript.py',  
             dependencies=['lib/mylib'],  
             . . .
```

The last technique is to install libraries in the Dockerfile for the container, rebuild the image, and push it to Amazon ECR. If you also need the libraries at prediction time (say, for preprocessing), this is the best option.

Adding libraries for deployment

If you need specific libraries to be available at prediction time, you can use a `requirements.txt` file for libraries that can be installed with `pip`.

For other libraries, the only option is to customize the framework container. You can pass its name to the estimator with the `image_uri` parameter:

```
sk = SKLearn(entry_point='myscript.py', image_uri=  
             '123456789012.dkr.ecr.eu-west-1.amazonaws.com/my-sklearn'  
             . . .
```

We covered a lot of technical topics in this section. Now, let's look at the big picture.

Putting it all together

The typical workflow when working with frameworks looks like this:

1. Implement script mode in your code; that is, read the necessary hyperparameters, input data, and output location.
2. If required, add a `model_fn()` function to load the model.
3. Test your training code locally, outside of any SageMaker container.
4. Configure the appropriate estimator (XGBoost, TensorFlow, and so on).
5. Train in local mode using the estimator, with either the built-in container or a container you've customized.
6. Deploy in local mode and test your model.
7. Switch to a managed instance type (say, `m1.m5.large`) for training and deployment.

This logical progression requires little work at each step. It minimizes friction, the risk of mistakes, and frustration. It also optimizes instance time and cost—no need to wait and pay for managed instances if your code crashes immediately because of a silly bug.

Now, let's put this knowledge to work. In the next section, we're going to run a simple scikit-learn example. The purpose is to make sure we understand the workflow we just discussed.

Running your framework code on Amazon SageMaker

We will start from a vanilla scikit-learn program that trains and saves a linear regression model on the Boston Housing dataset, which we used in *Chapter 4, Training Machine Learning Models*:

```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
import joblib
data = pd.read_csv('housing.csv')
labels = data[['medv']]
samples = data.drop(['medv'], axis=1)
X_train, X_test, y_train, y_test = train_test_split(
    samples, labels, test_size=0.1, random_state=123)
regr = LinearRegression(normalize=True)
regr.fit(X_train, y_train)
y_pred = regr.predict(X_test)
print('Mean squared error: %.2f'
      % mean_squared_error(y_test, y_pred))
print('Coefficient of determination: %.2f'
      % r2_score(y_test, y_pred))
joblib.dump(regr, 'model.joblib')
```

Let's update it so that it runs on SageMaker.

Implementing script mode

Now, we will use the framework to implement script mode, as follows:

1. First, read the hyperparameters as command-line arguments:

```
import argparse
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--normalize', type=bool,
                        default=False)
    parser.add_argument('--test-size', type=float,
                        default=0.1)
    parser.add_argument('--random-state', type=int,
                        default=123)
    args, _ = parser.parse_known_args()
    normalize = args.normalize
    test_size = args.test_size
    random_state = args.random_state
    data = pd.read_csv('housing.csv')
    labels = data[['medv']]
    samples = data.drop(['medv'], axis=1)
    X_train, X_test, y_train, y_test = train_test_split(
        samples, labels, test_size=test_size,
        random_state=random_state)
    . . .
```

2. Read the input and output paths as command-line arguments. We could decide to remove the splitting code and pass two input channels instead. Let's stick to one channel, that is, `training`:

```
import os
if __name__ == '__main__':
    . .
    parser.add_argument('--model-dir', type=str,
                        default=os.environ['SM_MODEL_DIR'])
    parser.add_argument('--training', type=str,
                        default=os.environ['SM_CHANNEL_TRAINING'])
```

```
    . . .
    model_dir = args.model_dir
    training_dir = args.training
    . . .
    filename = os.path.join(training_dir, 'housing.csv')
    data = pd.read_csv(filename)
    . . .
    model = os.path.join(model_dir, 'model.joblib')
    dump(regr, model)
```

3. As we're using scikit-learn, we need to add `model_fn()` to load the model at deployment time:

```
def model_fn(model_dir):
    model = joblib.load(os.path.join(model_dir,
                                     'model.joblib'))
    return model
```

With that, we're done. Time to test!

Testing locally

First, we test our script on our local machine in a Python 3 environment, outside of any SageMaker container. We just need to make sure that we have pandas and scikit-learn installed.

We set the environment variables to empty values as we will pass the paths on the command line:

```
$ source activate python3
$ export SM_CHANNEL_TRAINING=
$ export SM_MODEL_DIR=
$ python sklearn-boston-housing.py --normalize True -test-
    ration 0.1 --training . --model-dir .
Mean squared error: 41.82
Coefficient of determination: 0.63
```

Nice. Our code runs fine with command-line arguments. We can use this for local development and debugging, until we're ready to move it to SageMaker local mode.

Using local mode

We'll get started using the following steps:

1. Still on our local machine, we configure an SKLearn estimator in local mode, setting the role according to the setup we're using. Use local paths only:

```
role = 'arn:aws:iam::0123456789012:role/Sagemaker-  
fullaccess'  
  
sk = SKLearn(entry_point='sklearn-boston-housing.py',  
             role=role,  
             framework_version='0.23-1',  
             instance_count=1,  
             instance_type='local',  
             output_path=output_path,  
             hyperparameters={'normalize': True,  
                             'test-size': 0.1})  
  
sk.fit({'training':training_path})
```

2. As expected, we can see how our code is invoked in the training log. Of course, we get the same outcome:

```
/miniconda3/bin/python -m sklearn-boston-housing  
--normalize True --test-size 0.1  
  
...  
Mean squared error: 41.82  
Coefficient of determination: 0.63
```

3. We deploy locally and send some CSV samples for prediction:

```
sk_predictor = sk.deploy(initial_instance_count=1,  
                         instance_type='local')  
  
data = pd.read_csv('housing.csv')  
payload = data[:10].drop(['medv'], axis=1)  
payload = payload.to_csv(header=False, index=False)  
sk_predictor.serializer =  
    sagemaker.serializers.CSVSerializer()  
sk_predictor.deserializer =  
    sagemaker.deserializers.CSVDeserializer()  
response = sk_predictor.predict(payload)  
print(response)
```

By printing the response, we will see the predicted values:

```
[[29.801388899699845], ['24.990809475886074'],
 ['30.7379654455552'], ['28.786967125316544'],
 ['28.1421501991961'], ['25.301714533101716'],
 ['22.717977231840184'], ['19.302415613883348'],
 ['11.369520911229536'], ['18.785593532977657']]
```

With local mode, we can quickly iterate on our model. We're only limited by the compute and storage capabilities of the local machine. When that happens, we can easily move to managed infrastructure.

Using managed infrastructure

When it's time to train at scale and deploy in production, all we have to do is make sure the input data is in S3 and replace the "local" instance type with an actual instance type:

```
sess = sagemaker.Session()
bucket = sess.default_bucket()
prefix = 'sklearn-boston-housing'
training_path = sess.upload_data(path='housing.csv',
                                  key_prefix=prefix + "/training")
output_path = 's3://{}//{}//output/'.format(bucket,prefix)
sk = SKLearn(..., instance_type='ml.m5.large')
sk.fit({'training':training_path})
...
sk_predictor = sk.deploy(initial_instance_count=1,
                         instance_type='ml.t2.medium')
```

Since we're using the same container, we can be confident that training and deployment will work as expected. Again, I strongly recommend that you follow this logical progression: local work first, then SageMaker local mode, and finally, SageMaker managed infrastructure. It will help you focus on what needs to be done and when.

For the remainder of this chapter, we're going to run additional examples.

Using the built-in frameworks

We've covered XGBoost and scikit-learn already. Now, it's time to see how we can use deep learning frameworks. Let's start with TensorFlow and Keras.

Working with TensorFlow and Keras

In this example, we're going to use TensorFlow 2.4.1 to train a simple convolutional neural network on the Fashion-MNIST dataset (<https://github.com/zalandoresearch/fashion-mnist>).

Our code is split into two source files: one for the entry point script (`fashion.py`) and one for the model (`model.py`, based on Keras layers). For the sake of brevity, I will only discuss the SageMaker steps. You can find the full code in the GitHub repository for this book:

1. `fashion.py` starts by reading hyperparameters from the command line:

```
import tensorflow as tf
import numpy as np
import argparse, os
from model import FMNISTModel
parser = argparse.ArgumentParser()
parser.add_argument('--epochs', type=int, default=10)
parser.add_argument('--learning-rate', type=float,
                    default=0.01)
parser.add_argument('--batch-size', type=int,
                    default=128)
```

2. Next, we read the environment variables, that is, the input paths for the training set and the validation set, the output path for the model, and the number of GPUs available on the instance. It's the first time we're using the latter. It comes in handy to adjust the batch size for multi-GPU training as it's common practice to multiply the initial batch's size by the number of GPUs:

```
parser.add_argument('--training', type=str,
                    default=os.environ['SM_CHANNEL_TRAINING'])
parser.add_argument('--validation', type=str,
                    default=os.environ['SM_CHANNEL_VALIDATION'])
parser.add_argument('--model-dir', type=str,
                    default=os.environ['SM_MODEL_DIR'])
parser.add_argument('--gpu-count', type=int,
                    default=os.environ['SM_NUM_GPUS'])
```

3. Store the arguments in local variables. Then, load the dataset. Each channel provides us with a compressed numpy array for storing images and labels:

```
x_train = np.load(os.path.join(training_dir,  
                               'training.npz'))['image']  
y_train = np.load(os.path.join(training_dir,  
                               'training.npz'))['label']  
x_val = np.load(os.path.join(validation_dir,  
                               'validation.npz'))['image']  
y_val = np.load(os.path.join(validation_dir,  
                               'validation.npz'))['label']
```

4. Then, prepare the data for training by reshaping the image tensors, normalizing the pixel values, one-hot encoding the image labels, and creating the `tf.data.Dataset` objects that will feed data to the model.
5. Create the model, compile it, and fit it.
6. Once training is complete, save the model in TensorFlow Serving format at the appropriate output location. This step is important as this is the model server that SageMaker uses for TensorFlow models:

```
model.save(os.path.join(model_dir, '1'))
```

We train and deploy the model using the usual workflow:

1. In a notebook powered by a TensorFlow 2 kernel, we download the dataset and upload it to S3:

```
import os  
import numpy as np  
import keras  
from keras.datasets import fashion_mnist  
(x_train, y_train), (x_val, y_val) =  
    fashion_mnist.load_data()  
os.makedirs("./data", exist_ok = True)  
np.savez('./data/training', image=x_train,  
        label=y_train)  
np.savez('./data/validation', image=x_val,  
        label=y_val)  
prefix = 'tf2-fashion-mnist'
```

```
training_input_path = sess.upload_data(
    'data/training.npz',
    key_prefix=prefix+'/training')
validation_input_path = sess.upload_data(
    'data/validation.npz',
    key_prefix=prefix+'/validation')
```

2. We configure the TensorFlow estimator. We also set the `source_dir` parameter so that our model's file is also deployed in the container:

```
from sagemaker.tensorflow import TensorFlow
tf_estimator = TensorFlow(entry_point='fmnist.py',
    source_dir='.',
    role=sagemaker.get_execution_role(),
    instance_count=1,
    instance_type='ml.p3.2xlarge',
    framework_version='2.4.1',
    py_version='py37',
    hyperparameters={'epochs': 10})
```

3. Train and deploy as usual. We will go straight for managed infrastructure, but the same code will work fine on your local machine in local mode:

```
from time import strftime, gmtime
tf_estimator.fit(
    {'training': training_input_path,
     'validation': validation_input_path})
tf_endpoint_name = 'tf2-fmnist-'+strftime("%Y-%m-%d-%H-%M-%S", gmtime())
tf_predictor = tf_estimator.deploy(
    initial_instance_count=1,
    instance_type='ml.m5.large',
    endpoint_name=tf_endpoint_name)
```

4. The validation accuracy should be 91-92%. By loading and displaying a few sample images from the validation dataset, we can predict their labels. The numpy payload is automatically serialized to JSON, which is the default format for prediction data:

```
response = tf_predictor.predict(payload)
prediction = np.array(reponse['predictions'])
predicted_label = prediction.argmax(axis=1)
print('Predicted labels are:
{} '.format(predicted_label))
```

The output should look as follows:

Predicted labels are: [5 5 3 2 0]

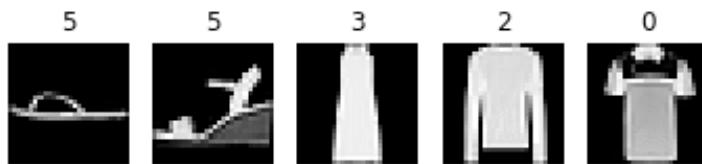


Figure 7.1 – Viewing predicted classes

5. When we're done, we delete the endpoint:

```
tf_predictor.delete_endpoint()
```

As you can see, the combination of script mode and built-in containers makes it easy to run TensorFlow on SageMaker. Once you get into the routine, you'll be surprised at how fast you can move your models from your laptop to AWS.

Now, let's take a look at PyTorch.

Working with PyTorch

PyTorch is extremely popular for computer vision, NLP, and more.

In this example, we're going to train a **Graph Neural Network (GNN)**. This category of networks works particularly well on graph-structured data, such as social networks, life sciences, and more. In fact, our PyTorch code will use the **Deep Graph Library (DGL)**, an open source library that makes it easier to build and train GNNs with TensorFlow, PyTorch, and Apache MXNet (<https://www.dgl.ai/>). DGL is already installed in these containers, so let's get to work directly.

We're going to work with the Zachary Karate Club dataset (<http://konect.cc/networks/ucidata-zachary/>). The following is the graph for this:

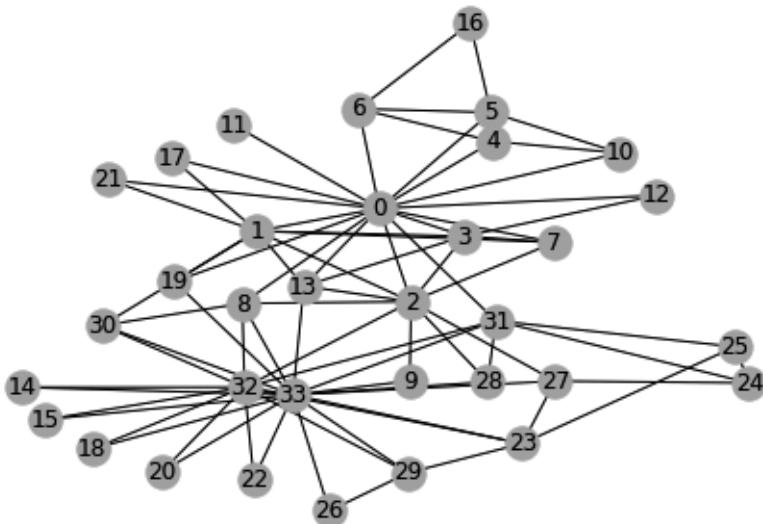


Figure 7.2 – The Zachary Karate Club dataset

Nodes 0 and 33 are teachers, while the other nodes are students. Edges represent ties between these people. As the story goes, the two teachers had an argument and the club needs to be split in two.

The purpose of the training job is to find the "best" split. This can be defined as a semi-supervision classification task. The first teacher (node 0) is assigned class 0, while the second teacher (node 33) is assigned class 1. All the other nodes are unlabeled, and their classes will be computed by a **graph convolutional network**. At the end of the last epoch, we'll retrieve the node classes and split the club accordingly.

The dataset is stored as a pickled Python list containing edges. Here are the first few edges:

```
[('0', '8'), ('1', '17'), ('24', '31'), . . .]
```

The SageMaker code is as simple as it gets. We upload the dataset to S3, create a PyTorch estimator, and train it:

```
import sagemaker
from sagemaker.pytorch import PyTorch
sess = sagemaker.Session()
prefix = 'dgl-karate-club'
```

```
training_input_path = sess.upload_data('edge_list.pickle',
key_prefix=prefix+'/training')
estimator = PyTorch(role=sagemaker.get_execution_role(),
entry_point='karate_club_sagemaker.py',
hyperparameters={'node_count': 34, 'epochs': 30},
framework_version='1.5.0',
py_version='py3',
instance_count=1,
instance_type='ml.m5.large')
estimator.fit({'training': training_input_path})
```

This hardly needs any explaining at all, does it?

Let's take a look at the abbreviated training script, where we're using script mode once again. The full version is available in the GitHub repository for this book:

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--epochs', type=int, default=30)
    parser.add_argument('--node_count', type=int)
    args, _ = parser.parse_known_args()
    epochs = args.epochs
    node_count = args.node_count
    training_dir = os.environ['SM_CHANNEL_TRAINING']
    model_dir = os.environ['SM_MODEL_DIR']
    with open(os.path.join(training_dir, 'edge_list.pickle'),
              'rb') as f:
        edge_list = pickle.load(f)
    # Build the graph and the model
    . .
    # Train the model
    . .
    # Print predicted classes
    last_epoch = all_preds[epochs-1].detach().numpy()
    predicted_class = np.argmax(last_epoch, axis=-1)
    print(predicted_class)
```

```
# Save the model
torch.save(net.state_dict(), os.path.join(model_dir,
'karate_club.pt'))
```

The following classes are predicted. Nodes 0 and 1 are class 0, node 2 is class 1, and so on:

```
[0 0 1 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 1 0 1 0 1 1 1 1 1 1 1 1 1 1]
```

By plotting them, we can see that the club has been cleanly split:

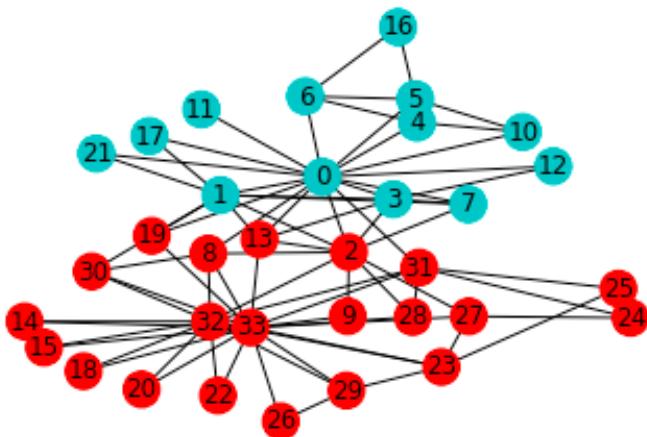


Figure 7.3 – Viewing predicted classes

Once again, the SageMaker code doesn't stand in your way. The workflow and APIs are consistent from one framework to the next, and you can focus on the machine learning problem itself. Now, let's do another example with Hugging Face, where we'll also see how to deploy a PyTorch model with the built-in PyTorch container.

Working with Hugging Face

Hugging Face (<https://huggingface.co>) has quickly become the most popular collection of open source models for NLP. At the time of writing, they host almost 10,000 state-of-the-art models (<https://huggingface.co/models>), pretrained on datasets (<https://huggingface.co/datasets>) in over 250 languages (<https://huggingface.co/languages>).

To make it easy to quickly build high-quality NLP applications, Hugging Face actively developed three open source libraries:

- **Transformers:** Train, fine-tune, and predict with Hugging Face models (<https://github.com/huggingface/transformers>).
- **Datasets:** Download and process Hugging Face datasets (<https://github.com/huggingface/datasets>).
- **Tokenizers:** Tokenize text for training and prediction with Hugging Face models (<https://github.com/huggingface/tokenizers>).

Hugging Face tutorial

If you are completely new to Hugging Face, please run through their tutorial first at <https://huggingface.co/transformers/quicktour.html>.

SageMaker added support for Hugging Face in March 2021, on both TensorFlow and PyTorch. As you would expect, you can use a HuggingFace estimator and built-in containers. Let's run an example where we build a sentiment analysis model for English language customer reviews. For this purpose, we'll fine-tune a **DistilBERT** model (<https://arxiv.org/abs/1910.01108>) implemented with PyTorch and pretrained on two large English language datasets (Wikipedia and the BookCorpus dataset).

Preparing the dataset

In this example, we'll use a Hugging Face dataset named `generated_reviews_enth` (https://huggingface.co/datasets/generated_reviews_enth). It includes an English review, its Thai translation, a flag indicating whether the translation is correct or not, and a star rating:

```
{'correct': 0, 'review_star': 4, 'translation': {'en': "I had a hard time finding a case for my new LG Lucid 2 but finally found this one on amazon. The colors are really pretty and it works just as well as, if not better than the otterbox. Hopefully there will be more available by next Xmas season. Overall, very cute case. I love cheetah's. :)", 'th': 'ฉันมีปัญหาในการหาเคสสำหรับ LG Lucid 2 ในเมืองฉัน แต่ในที่สุดก็พบเคสนี้ใน Amazon สีสวยงามและใช้งานได้ดีเช่นเดียวกับถ้าไม่ต้องว่านาก หวังว่าจะมีให้มากขึ้นในช่วงเทศกาลคริสต์มาสหน้า โดยรวมแล้วน่ารักมาก ๆ ฉันรักเสื้อชีต้าห์ :)"}}
```

This is the format that the DistilBERT tokenizer expects: a `labels` variable (0 for negative sentiment, 1 for positive) and a `text` variable with the English language review:

```
{'labels': 1,
 'text': "I had a hard time finding a case for my new LG Lucid
2 but finally found this one on amazon. The colors are really
pretty and it works just as well as, if not better than the
otterbox. Hopefully there will be more available by next Xmas
season. Overall, very cute case. I love cheetah's. :)"}
```

Let's get to work! I'll show you the individual steps, and you'll also find a **SageMaker Processing** version in the GitHub repository for this book:

1. We first install the `transformers` and `datasets` libraries:

```
!pip -q install "transformers>=4.4.2"
"datasets[s3]==1.5.0" --upgrade
```

2. We download the dataset, which is already split for training (141,369 instances) and validation (15,708 instances). All data is in JSON format:

```
from datasets import load_dataset
train_dataset, valid_dataset = load_dataset('generated_
reviews_enth',
split=['train', 'validation'])
```

3. In each review, we create a new variable named `labels`. We set it to 1 when `review_star` is equal to or higher than 4, and to 0 otherwise:

```
def map_stars_to_sentiment(row):
    return {
        'labels': 1 if row['review_star'] >= 4 else 0
    }
train_dataset =
    train_dataset.map(map_stars_to_sentiment)
valid_dataset =
    valid_dataset.map(map_stars_to_sentiment)
```

4. The reviews are nested JSON documents, making it difficult to remove variables we don't need. Let's flatten both datasets:

```
train_dataset = train_dataset.flatten()  
valid_dataset = valid_dataset.flatten()
```

5. We can now easily drop unwanted variables. We also rename the `translation.en` variable to `text`:

```
train_dataset = train_dataset.remove_columns(  
    ['correct', 'translation.th', 'review_star'])  
valid_dataset = valid_dataset.remove_columns(  
    ['correct', 'translation.th', 'review_star'])  
train_dataset = train_dataset.rename_column(  
    'translation.en', 'text')  
valid_dataset = valid_dataset.rename_column(  
    'translation.en', 'text')
```

The training and validation instances now have the format expected by the DistilBERT tokenizer. We already covered tokenization in *Chapter 6, Training Natural Language Processing Models*. A significant difference is that we use a tokenizer that was pretrained on the same English language corpus as the model:

1. We download the tokenizer for our pretrained model:

```
from transformers import AutoTokenizer  
tokenizer = AutoTokenizer.from_pretrained(  
    'distilbert-base-uncased')  
def tokenize(batch):  
    return tokenizer(batch['text'],  
        padding='max_length', truncation=True)
```

2. We tokenize both datasets. Words and punctuation are replaced with appropriate tokens. If needed, each sequence is padded or truncated to fit the input layer of the model (512 tokens):

```
train_dataset = train_dataset.map(tokenize,  
    batched=True, batch_size=len(train_dataset))  
valid_dataset = valid_dataset.map(tokenize,  
    batched=True, batch_size=len(valid_dataset))
```

3. We drop the `text` variable, as it's not needed anymore:

```
train_dataset = train_dataset.remove_columns(['text'])  
valid_dataset = valid_dataset.remove_columns(['text'])
```

4. Printing out an instance, we see the attention mask (all ones, meaning no token is masked in the input sequence), the inputs IDs (the sequence of tokens), and the label:

```
{"attention_mask": [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,  
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,  
,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,  
1,1,<zero padding>], "input_ids": [101,1045, 2018,1037,2  
524,2051,4531,1037,2553,2005,2026,2047,1048,2290,12776,3  
593,1016,2021,2633,2179,2023,2028,2006,9733,1012,1996,60  
87,2024,2428,3492,1998,2009,2573,2074,2004,2092,2004,1010  
,2065,2025,2488,2084,1996,22279,8758,1012,11504,2045,209  
7,2022,2062,2800,2011,2279,1060,9335,2161,1012,3452,1010  
,2200,10140,2553,1012,1045,2293,18178,12928,2232,1005,105  
5,1012,1024,1007,102,<zero padding>], "labels": 1}'
```

Data preparation is complete. Let's move on to training the model.

Fine-tuning a Hugging Face model

We're not going to train from scratch: it would take far too long, and we probably don't have enough data anyway. Instead, we're going to fine-tune the model. Starting from a model trained on a very large text corpus, we will just train it for one additional epoch on our own data, so that it picks up the particular patterns present in our data:

1. We start by uploading both datasets to S3. The `datasets` library provides a convenient API to do this:

```
import sagemaker  
from datasets.filesystems import S3FileSystem  
bucket = sagemaker.Session().default_bucket()  
s3_prefix = 'hugging-face/sentiment-analysis'  
train_input_path =  
    f's3://{bucket}/{s3_prefix}/training'  
valid_input_path =  
    f's3://{bucket}/{s3_prefix}/validation'  
s3 = S3FileSystem()  
train_dataset.save_to_disk(train_input_path, fs=s3)  
valid_dataset.save_to_disk(valid_input_path, fs=s3)
```

2. We define hyperparameters and configure a HuggingFace estimator. Note that we'll fine-tune the model for just one epoch:

```
hyperparameters={
    'epochs': 1,
    'train_batch_size': 32,
    'model_name':'distilbert-base-uncased'
}
from sagemaker.huggingface import HuggingFace
huggingface_estimator = HuggingFace(
    role=sagemaker.get_execution_role(),
    entry_point='train.py',
    hyperparameters=hyperparameters,
    transformers_version='4.4.2',
    pytorch_version='1.6.0',
    py_version='py36',
    instance_type='ml.p3.2xlarge',
    instance_count=1
)
```

For the sake of brevity, I won't discuss the training script (`train.py`), which is available in the GitHub repository for this book. There's nothing particular about it: we use the Trainer Hugging Face API, as well as script mode to interface it with SageMaker. As we only train for a single epoch, checkpointing is disabled (`save_strategy='no'`). This helps cuts down on training time (not saving checkpoints) and deployment time (the model artifact is smaller).

3. It's also worth noting that you can generate boilerplate code for your estimator on the Hugging Face website. As shown in the following screenshot, you can click on **Amazon SageMaker**, pick a task type, and copy and paste the generated code:



Figure 7.4 – Viewing our model on the Hugging Face website

4. We launch the training job as usual, and it lasts about 42 minutes:

```
huggingface_estimator.fit(  
    {'train': train_input_path,  
     'valid': valid_input_path})
```

Just like with other frameworks, we could call the `deploy()` API in order to deploy our model to a SageMaker endpoint. You can find an example at <https://aws.amazon.com/blogs/machine-learning/announcing-managed-inference-for-hugging-face-models-in-amazon-sagemaker/>.

Instead, let's see how we can deploy our model with the built-in PyTorch container and **TorchServe**. In fact, this deployment example can generalize to any PyTorch model that you'd like to serve with TorchServe.

I find this superb blog post by my colleague Todd Escalona extremely helpful in understanding how to serve PyTorch models with TorchServe: <https://aws.amazon.com/blogs/machine-learning/serving-pytorch-models-in-production-with-the-amazon-sagemaker-native-torchserve-integration/>.

Deploying a Hugging Face model

The only difference compared to previous examples is that we have to use the model artifact in S3 to create a `PyTorchModel` object, and to build a `Predictor` model that we can use `deploy()` and `predict()` on:

1. Starting from the model artifact, we define a `Predictor` object, and we create a `PyTorchModel` with it:

```
from sagemaker.pytorch import PyTorchModel  
from sagemaker.serializers import JSONSerializer  
from sagemaker.deserializers import JSONDeserializer  
model = PyTorchModel(  
    model_data=huggingface_estimator.model_data,  
    role=sagemaker.get_execution_role(),  
    entry_point='torchserve-predictor.py',  
    framework_version='1.6.0',  
    py_version='py36')
```

2. Zooming in on the inference script (`torchserve-predictor.py`), we write a model loading function to account for Hugging Face peculiarities that the PyTorch container can't handle by default:

```
def model_fn(model_dir):  
    config_path = '{}/config.json'.format(model_dir)  
    model_path = '{}/pytorch_model.bin'.format(model_dir)  
    config = AutoConfig.from_pretrained(config_path)  
    model = DistilBertForSequenceClassification  
        .from_pretrained(model_path, config=config)  
    return model
```

3. We also add a prediction function that returns a text label:

```
tokenizer = AutoTokenizer.from_pretrained(  
    'distilbert-base-uncased')  
CLASS_NAMES = ['negative', 'positive']  
def predict_fn(input_data, model):  
    inputs = tokenizer(input_data['text'],  
                      return_tensors='pt')  
    outputs = model(**inputs)  
    logits = outputs.logits  
    _, prediction = torch.max(logits, dim=1)  
    return CLASS_NAMES[prediction]
```

4. The inference script also includes basic `input_fn()` and `output_fn()` functions to check that data is in JSON format. You'll find the code in the GitHub repository for the book.
5. Coming back to our notebook, we deploy the model as usual:

```
predictor = model.deploy(  
    initial_instance_count=1,  
    instance_type='ml.m5.xlarge')
```

- Once the endpoint is up, we predict a text sample and print the result:

```
predictor.serializer = JSONSerializer()
predictor.deserializer = JSONDeserializer()
sample = {'text': 'This camera is really amazing!'}
prediction = predictor.predict(test_data)
print(prediction)
['positive']
```

- Finally, we delete the endpoint:

```
predictor.delete_endpoint()
```

As you can see, it's really easy to work with Hugging Face models. It's also a cost-effective way to build high-quality NLP models, as we typically fine-tune them for a very small number of epochs.

To close this chapter, let's look at how SageMaker and Apache Spark can work together.

Working with Apache Spark

In addition to the Python SageMaker SDK that we've been using so far, SageMaker also includes an SDK for Spark (<https://github.com/aws/sagemaker-spark>). This lets you run SageMaker jobs directly from a PySpark or Scala application running on a Spark cluster.

Combining Spark and SageMaker

First, you can decouple the **Extract-Transform-Load (ETL)** step and the machine learning step. Each usually has different infrastructure requirements (instance type, instance count, storage) that need to be the right size both technically and financially. Setting up your Spark cluster just right for ETL and using on-demand infrastructure in SageMaker for training and prediction is a powerful combination.

Second, although Spark's MLlib is an amazing library, you may need something else, such as custom algorithms in different languages or deep learning.

Finally, deploying models for prediction on Spark clusters may not be the best option. SageMaker endpoints should be considered instead, especially since they support the **MLeap** format (<https://combust.github.io/mleap-docs/>).

In the following example, we'll combine SageMaker and Spark to build a spam detection model. Data will be hosted in S3, with one text file for spam messages and one for non-spam ("ham") messages. We'll use Spark running on an Amazon EMR cluster to preprocess it. Then, we'll train and deploy a model with the XGBoost algorithm that's available in SageMaker. Finally, we'll predict data with it on our Spark cluster. For the sake of language diversity, we'll code with Scala this time.

First of all, we need to build a Spark cluster.

Creating a Spark cluster

We will create the cluster as follows:

1. Starting from the **Amazon EMR** console at <https://console.aws.amazon.com/elasticmapreduce>, we will create a cluster. First, click on **Create cluster**, then on **Go to advanced options**. This lets us select the list of EMR applications present on the cluster: starting from EMR 5.33.0, we install **JupyterHub**, **JupyterEnterpriseGateway**, **Zeppelin**, and **Spark**, as visible in the following screenshot:

Software Configuration

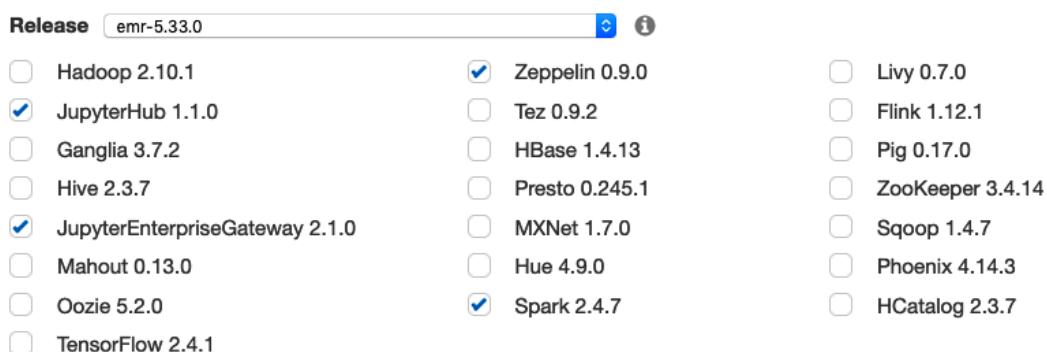


Figure 7.5 – Creating a Spark cluster

We then click on **Next** twice, name the cluster `sagemaker-cluster`, click on **Next** again, and then click on **Create cluster**. You can find additional details at <https://docs.aws.amazon.com/emr/>.

2. While the cluster is being created, we define our Git repository in the **Notebooks** entry in the left-hand side vertical menu. Then, we click on **Add repository**:

Add repository

Connect a GitHub or other Git-based repository with Amazon EMR notebooks. [Learn more](#) 

Repository name Names may only contain alphanumeric characters, hyphens (-), or underscores (_).

Git repository URL

Branch

Git credentials Amazon EMR saves your credentials using [Amazon Secrets Manager](#) 

Use an existing AWS secret
 Create a new secret
 Use a public repository without credentials

Figure 7.6 – Adding a Git repository

3. Then, we create a Jupyter notebook connected to the cluster. Starting from the **Notebooks** entry in the left-hand side vertical menu, as shown in the following screenshot, we give it a name and select both the EMR cluster and the repository we just created. Then, we click on **Create notebook**:

Name and configure your notebook

Name your notebook, choose a cluster or create one, and customize configuration options if desired. [Learn more](#)

Notebook name* Names may only contain alphanumeric characters, hyphens (-), or underscores (_).

Description

256 characters max.

Cluster* Choose an existing cluster [Choose](#) sagemaker-cluster [J-1FTR6V42NPLHE](#)

Create a cluster [?](#)

Security groups Use default security groups [?](#)

Choose security groups (vpc-bad436d3)

AWS service role* [?](#)

Notebook location* Choose an S3 location where files for this notebook are saved.

Use a location that EMR creates [?](#)
s3://aws-emr-resources-613904931467-ap-northeast-2/notebooks/

Choose an existing S3 location in ap-northeast-2

Git repository Link to a Git repository [Choose repository](#) book-repo [?](#)

Note: Make sure your cluster, service role and security groups have the required settings. [Learn more](#)

Figure 7.7 – Creating a Jupyter notebook

4. Once the cluster and the notebook are ready, we can click on **Open in Jupyter**, which takes us to the familiar Jupyter interface.

Everything is now ready. Let's write a spam classifier!

Building a spam classification model with Spark and SageMaker

In this example, we're going to use the combined benefits of Spark and SageMaker to train, deploy, and predict with a spam classification model, thanks to just a few lines of Scala code:

- First, we need to make sure that our dataset is available in S3. On our local machine, upload the two files to the default SageMaker bucket (feel free to use another bucket):

```
$ aws s3 cp ham s3://sagemaker-eu-west-1-123456789012
$ aws s3 cp spam s3://sagemaker-eu-west-1-123456789012
```

- Back in the Jupyter notebook, make sure it's running the Spark kernel. Then, import the necessary objects from Spark MLlib and the SageMaker SDK.
- Load the data from S3. Convert all the sentences into lowercase. Then, remove all punctuation and numbers and trim any whitespace:

```
val spam = sc.textFile(
  "s3://sagemaker-eu-west-1-123456789012/spam")
  .map(l => l.toLowerCase())
  .map(l => l.replaceAll("[^ a-z]", ""))
  .map(l => l.trim())
val ham = sc.textFile(
  "s3://sagemaker-eu-west-1-123456789012/ham")
  .map(l => l.toLowerCase())
  .map(l => l.replaceAll("[^ a-z]", ""))
  .map(l => l.trim())
```

- Then, split the messages into words and hash these words into 200 buckets. This technique is much less sophisticated than the word vectors we used in *Chapter 6, Training Natural Language Processing Models*, but it should do the trick:

```
val tf = new HashingTF(numFeatures = 200)
val spamFeatures = spam.map(
  m => tf.transform(m.split(" ")))
val hamFeatures = ham.map(
  m => tf.transform(m.split(" ")))
```

For example, the following message has one occurrence of a word from bucket 15, one from bucket 83, two words from bucket 96, and two from bucket 188:

```
Array((200,[15,83,96,188],[1.0,1.0,2.0,2.0]))
```

5. We assign a 1 label for spam messages and a 0 label for ham messages:

```
val positiveExamples = spamFeatures.map(  
    features => LabeledPoint(1, features))  
val negativeExamples = hamFeatures.map(  
    features => LabeledPoint(0, features))
```

6. Merge the messages and encode them in **LIBSVM** format, one of the formats supported by **XGBoost**:

```
val data = positiveExamples.union(negativeExamples)  
val data_libsvm =  
    MLUtils.convertVectorColumnsToML(data.toDF)
```

The samples now look similar to this:

```
Array([1.0, (200, [2, 41, 99, 146, 172, 181], [2.0, 1.0, 1.0, 1.0, 1.  
0])])
```

7. Split the data for training and validation:

```
val Array(trainingData, testData) =  
    data_libsvm.randomSplit(Array(0.8, 0.2))
```

8. Configure the XGBoost estimator available in the SageMaker SDK. Here, we're going to train and deploy in one single step:

```
val roleArn = "arn:aws:iam:YOUR_SAGEMAKER_ROLE"  
val xgboost_estimator = new XGBoostSageMakerEstimator(  
    trainingInstanceType="ml.m5.large",  
    trainingInstanceCount=1,  
    endpointInstanceType="ml.t2.medium",  
    endpointInitialInstanceCount=1,  
    sagemakerRole=IAMRole(roleArn))  
xgboost_estimator.setObjective("binary:logistic")  
xgboost_estimator.setNumRound(25)
```

9. Fire up a training job and a deployment job on the managed infrastructure, exactly like when we worked with built-in algorithms in *Chapter 4, Training Machine Learning Models*. The SageMaker SDK automatically passes the Spark DataFrame to the training job, so no work is required from our end:

```
val xgboost_model =  
    xgboost_estimator.fit(trainingData_libsvm)
```

As you would expect, these activities are visible in SageMaker Studio in the **Experiments** section.

10. When the deployment is complete, transform the test set and score the model. This automatically invokes the SageMaker endpoint. Once again, we don't need to worry about data movement:

```
val transformedData =  
    xgboost_model.transform(testData_libsvm)  
val accuracy = 1.0*transformedData.filter(  
    $"label" === $"prediction")  
.count/transformedData.count()
```

The accuracy should be around 97%, which is not too bad!

11. Once done, delete all SageMaker resources created by the job. This will delete the model, the endpoint, and the endpoint configuration (an object we haven't discussed yet):

```
val cleanup = new SageMakerResourceCleanup(  
    xgboost_model.sagemakerClient)  
cleanup.deleteResources(  
    xgboost_model.getCreatedResources)
```

12. Don't forget to terminate the notebook and the EMR cluster too. You can easily do this in the EMR console.

This example demonstrates how easy it is to combine the respective strengths of Spark and SageMaker. Another way to do this is to build MLlib pipelines with a mix of Spark and SageMaker stages. You'll find examples of this at <https://github.com/awslabs/amazon-sagemaker-examples/tree/master/sagemaker-spark>.

Summary

Open source frameworks such as scikit-learn and TensorFlow have made it simple to write machine learning and deep learning code. They've become immensely popular in the developer community and for good reason. However, managing training and deployment infrastructure still requires a lot of effort and skills that data scientists and machine learning engineers typically do not possess. SageMaker simplifies the whole process. You can go quickly from experimentation to production, without ever worrying about infrastructure.

In this chapter, you learned about the different frameworks available in SageMaker for machine learning and deep learning, as well as how to customize their containers. You also learned how to use script mode and local mode for fast iteration until you're ready to deploy in production. Finally, you ran several examples, including one that combines Apache Spark and SageMaker.

In the next chapter, you will learn how to use your own custom code on SageMaker, without having to rely on a built-in container.

8

Using Your Algorithms and Code

In the previous chapter, you learned how to train and deploy models with built-in frameworks such as **scikit-learn** and **TensorFlow**. Thanks to **script mode**, these frameworks make it easy to use your own code, without having to manage any training or inference containers.

In some cases, your business or technical environment could make it difficult or even impossible to use these containers. Maybe you need to be in full control of how containers are built. Maybe you'd like to implement your own prediction logic. Maybe you're working with a framework or language that's not natively supported by SageMaker.

In this chapter, you'll learn how to tailor training and inference containers to your own needs. You'll also learn how to train and deploy your own custom code, using either the SageMaker SDK directly or command-line open source tools.

We will cover the following topics in this chapter:

- Understanding how SageMaker invokes your code
- Customizing built-in framework containers
- Building custom training containers with the SageMaker Training Toolkit
- Building fully custom containers for training and inference with Python and R
- Training and deploying with your custom Python code on MLflow
- Building fully custom containers for SageMaker Processing

Technical requirements

You will need an AWS account to run the examples included in this chapter. If you haven't got one already, please point your browser at <https://aws.amazon.com/getting-started/> to create it. You should also familiarize yourself with the AWS Free Tier (<https://aws.amazon.com/free/>), which lets you use many AWS services for free within certain usage limits.

You will need to install and configure the AWS **Command-Line Interface (CLI)** for your account (<https://aws.amazon.com/cli/>).

You will need a working Python 3.x environment. Installing the Anaconda distribution (<https://www.anaconda.com/>) is not mandatory but strongly encouraged as it includes many projects that we will need (Jupyter, pandas, numpy, and more).

You will need a working Docker installation. You'll find installation instructions and documentation at <https://docs.docker.com>.

The code examples included in this book are available on GitHub at <https://github.com/PacktPublishing/Learn-Amazon-SageMaker-second-edition>. You will need to install a Git client to access them (<https://git-scm.com>).

Understanding how SageMaker invokes your code

When we worked with built-in algorithms and frameworks, we didn't pay much attention to how SageMaker actually invoked the training and deployment code. After all, that's what "built-in" means: grab what you need off the shelf and get to work.

Of course, things are different if we want to use our own custom code and containers. We need to understand how they interface with SageMaker so that we implement them exactly right.

In this section, we'll discuss this interface in detail. Let's start with the file layout.

Understanding the file layout inside a SageMaker container

To make our life simpler, SageMaker estimators automatically copy hyperparameters and input data inside training containers. Likewise, they automatically copy the trained model (and any checkpoints) from the container to S3. At deployment time, they do the reverse operation, copying the model from S3 into the container.

As you can imagine, this requires a file layout convention:

- Hyperparameters are stored as a JSON dictionary in `/opt/ml/input/config/hyperparameters.json`.
- Input channels are stored in `/opt/ml/input/data/CHANNEL_NAME`. We saw in the previous chapter that the channel names match the ones passed to the `fit()` API.
- The model should be saved in and loaded from `/opt/ml/model`.

Hence, we'll need to use these paths in our custom code. Now, let's see how the training and deployment code is invoked.

Understanding the options for custom training

In *Chapter 7, Extending Machine Learning Services Using Built-In Frameworks*, we studied script mode and how SageMaker uses it to invoke our training script. This feature is enabled by additional Python code present in the framework containers, namely, the SageMaker Training Toolkit (<https://github.com/aws/sagemaker-training-toolkit>).

In a nutshell, this training toolkit copies the entry point script, its hyperparameters, and its dependencies inside the container. It also copies data from the input channels inside the container. Then, it invokes the entry point script. Curious minds can read the code at `src/sagemaker_training/entry_point.py`.

When it comes to customizing your training code, you have the following options:

- Customize an existing framework container, adding only your extra dependencies and code. Script mode and the framework estimator will be available.
- Build a custom container based solely on the SageMaker Training Toolkit. Script mode and the generic `Estimator` module will be available, but you'll have to install everything else.
- Build a fully custom container. If you want to start from a blank page or don't want any extra code inside your container, this is the way to go. You'll train with the generic `Estimator` module, and script mode won't be available. Your training code will be invoked directly (more on this later).

Understanding the options for custom deployment

Framework containers include additional Python code for deployment. Here are the repositories for the most popular frameworks:

- **TensorFlow**: <https://github.com/aws/sagemaker-tensorflow-serving-container>. Models are served with **TensorFlow Serving** (<https://www.tensorflow.org/tfx/guide/serving>).
- **PyTorch**: <https://github.com/aws/sagemaker-pytorch-inference-toolkit>. Models are served with **TorchServe** (<https://pytorch.org/serve>).
- **Apache MXNet**: <https://github.com/aws/sagemaker-mxnet-inference-toolkit>. Models are served with the **Multi-Model Server** (<https://github.com/awslabs/multi-model-server>), integrated into the **SageMaker Inference Toolkit** (<https://github.com/aws/sagemaker-inference-toolkit>).
- **Scikit-learn**: <https://github.com/aws/sagemaker-scikit-learn-container>. Models are served with the Multi-Model Server.
- **XGBoost**: <https://github.com/aws/sagemaker-xgboost-container>. Models are served with the Multi-Model Server.

Just like for training, you have three options:

- Customize an existing framework container. Models will be served using the existing inference logic.
- Build a custom container based solely on the SageMaker Inference Toolkit. Models will be served by the Multi-Model Server.

- Build a fully custom container, doing away with any inference logic and implementing your own instead.

Whether you use a single container for training and deployment or two different containers is up to you. A lot of different factors come into play: who builds the containers, who runs them, and so on. Only you can decide what the best option for your particular setup is.

Now, let's run some examples!

Customizing an existing framework container

Of course, we could simply write a Dockerfile referencing one of the Deep Learning Containers images (https://github.com/aws/deep-learning-containers/blob/master/available_images.md) and add our own commands. See the following example:

```
FROM 763104351884.dkr.ecr.us-east-1.amazonaws.com/tensorflow-training:2.4.1-cpu-py37-ubuntu18.04  
... .
```

Instead, let's customize and rebuild the **PyTorch** training and inference containers on our local machine. The process is similar to other frameworks.

Build environment

Docker needs to be installed and running. To avoid throttling when pulling base images, I recommend that you create a **Docker Hub** account (<https://hub.docker.com>) and log in with `docker login` or **Docker Desktop**.

To avoid bizarre dependency issues (I'm looking at you, macOS), I also recommend that you build images on an **Amazon EC2** instance powered by **Amazon Linux 2**. You don't need a large one (`m5.large` should suffice), but please make sure to provision more storage than the default 8 GB. I recommend 64 GB. You also need to make sure that the **IAM** role for the instance allows you to push and pull EC2 images. If you're unsure how to create and connect to an EC2 instance, this tutorial will get you started: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EC2_GetStarted.html.

Setting up your build environment on EC2

We will get started using the following steps:

1. Once your EC2 instance is up, we connect to it with ssh. We first install Docker and add the ec2-user to the docker group. This will allow us to run Docker commands as a non-root user:

```
$ sudo yum -y install docker  
$ sudo usermod -a -G docker ec2-user
```

2. In order to apply this permission change, we log out and log in again.
3. We make sure that docker is running and we log in to Docker Hub:

```
$ service docker start  
$ docker login
```

4. We install git, Python 3, and pip:

```
$ sudo yum -y install git python3-devel python3-pip
```

Our EC2 instance is now ready, and we can move on to building containers.

Building training and inference containers

This can be done using the following steps:

1. We clone the deep-learning-containers repository, which centralizes all training and inference code for TensorFlow, PyTorch, Apache MXNet, and Hugging Face, and adds convenient scripts to build their containers:

```
$ git clone https://github.com/aws/deep-learning-  
containers.git  
$ cd deep-learning-containers
```

2. We set environment variables for our account ID, the region we're running in, and the name of a new repository we're going to create in Amazon ECR:

```
$ export ACCOUNT_ID=123456789012  
$ export REGION=eu-west-1  
$ export REPOSITORY_NAME=my-pt-dlc
```

3. We create the repository in Amazon ECR, and we log in. Please refer to the documentation for details (<https://docs.aws.amazon.com/ecr/index.html>):

```
$ aws ecr create-repository
--repository-name $REPOSITORY_NAME --region $REGION
$ aws ecr get-login-password --region $REGION | docker
login --username AWS --password-stdin $ACCOUNT_ID.dkr.
ecr.$REGION.amazonaws.com
```

4. We create a virtual environment, and we install the Python requirements:

```
$ python3 -m venv dlc
$ source dlc/bin/activate
$ pip install -r src/requirements.txt
```

5. Here, we'd like to build the training and inference containers for PyTorch 1.8, on both the CPU and GPU. We can find the corresponding Docker files in `pytorch/training/docker/1.8/py3/` and customize them to our needs. For example, we could pin Deep Graph Library to version 0.6.1:

```
&& conda install -c dgteam -y dgl==0.6.1 \
```

6. Once we've edited the Docker files, we take a look at the build configuration file for the latest PyTorch version (`pytorch/buildspec.yml`). We decide to customize image tags to make sure each image is clearly identifiable:

```
BuildCPUPTrainPy3DockerImage:
tag: !join [ *VERSION, "-", *DEVICE_TYPE, "-", *TAG_
PYTHON_VERSION, "-", *OS_VERSION, "-training" ]
BuildGPUPTTrainPy3DockerImage:
tag: !join [ *VERSION, "-", *DEVICE_TYPE, "-", *TAG_
PYTHON_VERSION, "-", *CUDA_VERSION, "-", *OS_VERSION,
"-training" ]
BuildCPUPInferencePy3DockerImage:
tag: !join [ *VERSION, "-", *DEVICE_TYPE, "-", *TAG_
PYTHON_VERSION, "-", *OS_VERSION, "-inference" ]
BuildGPUPTInferencePy3DockerImage:
tag: !join [ *VERSION, "-", *DEVICE_TYPE, "-", *TAG_
PYTHON_VERSION, "-", *CUDA_VERSION, "-", *OS_VERSION,
"-inference" ]
```

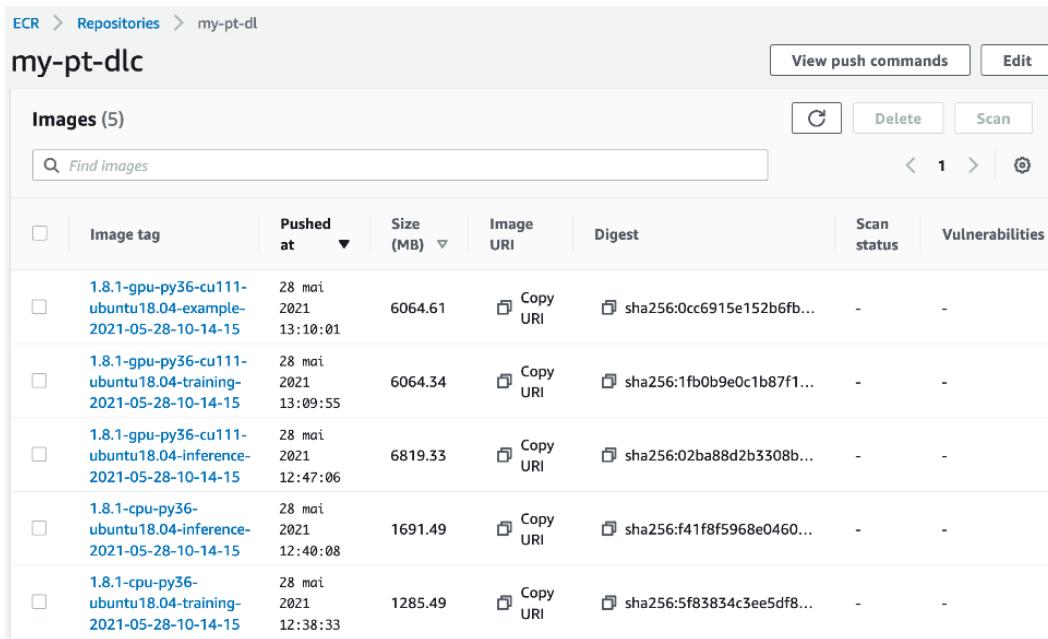
7. Finally, we run the setup script and launch the build process:

```
$ bash src/setup.sh pytorch  
$ python src/main.py --buildspec pytorch/buildspec.yml  
--framework pytorch --device_types cpu,gpu --image_types  
training,inference
```

8. After a little while, all four images are built (plus an example image), and we can see them in our local Docker:

```
$ docker images  
123456789012.dkr.ecr.eu-west-1.amazonaws.com/  
my-pt-dlc 1.8.1-gpu-py36-cu111-ubuntu18.04-  
example-2021-05-28-10-14-15  
123456789012.dkr.ecr.eu-west-1.amazonaws.com/  
my-pt-dlc 1.8.1-gpu-py36-cu111-ubuntu18.04-  
training-2021-05-28-10-14-15  
123456789012.dkr.ecr.eu-west-1.amazonaws.com/  
my-pt-dlc 1.8.1-gpu-py36-cu111-ubuntu18.04-  
inference-2021-05-28-10-14-15  
123456789012.dkr.ecr.eu-west-1.amazonaws.com/my-pt-dlc  
1.8.1-cpu-py36-ubuntu18.04-inference-2021-05-28-10-14-15  
123456789012.dkr.ecr.eu-west-1.amazonaws.com/my-pt-dlc  
1.8.1-cpu-py36-ubuntu18.04-training-2021-05-28-10-14-15
```

9. We can also see them in our ECR repository, as shown in the following screenshot:



The screenshot shows the AWS ECR interface for the repository 'my-pt-dlc'. At the top, there are buttons for 'View push commands' and 'Edit'. Below the header, there's a search bar with placeholder text 'Find images' and a table titled 'Images (5)'. The table has columns: 'Image tag', 'Pushed at', 'Size (MB)', 'Image URI', 'Digest', 'Scan status', and 'Vulnerabilities'. Each row contains a checkbox, the image tag, the date and time it was pushed, its size, a 'Copy URI' button, its SHA256 digest, and two empty fields for 'Scan status' and 'Vulnerabilities'. The five images listed are:

Image tag	Pushed at	Size (MB)	Image URI	Digest	Scan status	Vulnerabilities
1.8.1-gpu-py36-cu111-ubuntu18.04-example-2021-05-28-10-14-15	28 mai 2021 13:10:01	6064.61	Copy URI	sha256:0cc6915e152b6fb...	-	-
1.8.1-gpu-py36-cu111-ubuntu18.04-training-2021-05-28-10-14-15	28 mai 2021 13:09:55	6064.34	Copy URI	sha256:1fb0b9e0c1b87f1...	-	-
1.8.1-gpu-py36-cu111-ubuntu18.04-inference-2021-05-28-10-14-15	28 mai 2021 12:47:06	6819.33	Copy URI	sha256:02ba88d2b3308b...	-	-
1.8.1-cpu-py36-ubuntu18.04-inference-2021-05-28-10-14-15	28 mai 2021 12:40:08	1691.49	Copy URI	sha256:f41f8f5968e0460...	-	-
1.8.1-cpu-py36-ubuntu18.04-training-2021-05-28-10-14-15	28 mai 2021 12:38:33	1285.49	Copy URI	sha256:5f83834c3ee5df8...	-	-

Figure 8.1 – Viewing images in ECR

10. The images are now available with the SageMaker SDK. Let's train with our new CPU image. All we have to do is pass its name in the `image_uri` parameter of the PyTorch estimator. Please note that we can remove `py_version` and `framework_version`:

```
Estimator = PyTorch(
    image_uri='123456789012.dkr.ecr.eu-west-1.
amazonaws.com/my-pt-dlc:1.8.1-cpu-py36-ubuntu18.04-
training-2021-05-28-10-14-15',
    role=sagemaker.get_execution_role(),
    entry_point='karate_club_sagemaker.py',
    hyperparameters={'node_count': 34, 'epochs': 30},
    instance_count=1,
    instance_type='ml.m5.large')
```

As you can see, it's pretty easy to customize Deep Learning Containers. Now, let's go one level deeper and work only with the training toolkit.

Using the SageMaker Training Toolkit with scikit-learn

In this example, we're going to build a custom Python container with the SageMaker Training Toolkit. We'll use it to train a scikit-learn model on the Boston Housing dataset, using script mode and the SKLearn estimator.

We need three building blocks:

- The training script. Since script mode will be available, we can use exactly the same code as in the scikit-learn example from *Chapter 7, Extending Machine Learning Services Using Built-In Frameworks*.
- We need a Dockerfile and Docker commands to build our custom container.
- We also need an SKLearn estimator configured to use our custom container.

Let's take care of the container:

1. A Dockerfile can get quite complicated. No need for that here! We start from the official Python 3.7 image available on Docker Hub (https://hub.docker.com/_/python). We install scikit-learn, numpy, pandas, joblib, and the SageMaker Training Toolkit:

```
FROM python:3.7
RUN pip3 install --no-cache scikit-learn numpy pandas
joblib sagemaker-training
```

2. We build the image with the docker build command, tagging it as `sklearn-customer:sklearn`:

```
$ docker build -t sklearn-custom:sklearn -f Dockerfile .
```

Once the image is built, we find its identifier:

```
$ docker images
REPOSITORY          TAG           IMAGE ID
sklearn-custom      sklearn       bf412a511471
```

3. Using the AWS CLI, we create a repository in Amazon ECR to host this image, and we log in to the repository:

```
$ aws ecr create-repository --repository-name sklearn-custom --region eu-west-1  
$ aws ecr get-login-password --region eu-west-1 | docker login --username AWS --password-stdin 123456789012.dkr.ecr.eu-west-1.amazonaws.com/sklearn-custom:latest
```

4. Using the image identifier, we tag the image with the repository identifier:

```
$ docker tag bf412a511471 123456789012.dkr.ecr.eu-west-1.amazonaws.com/sklearn-custom:sklearn
```

5. We push the image to the repository:

```
$ docker push 123456789012.dkr.ecr.eu-west-1.amazonaws.com/sklearn-custom:sklearn
```

The image is now ready for training with a SageMaker estimator.

6. We define an SKLearn estimator, setting the `image_uri` parameter to the name of the container we just created:

```
sk = SKLearn(  
    role=sagemaker.get_execution_role(),  
    entry_point='sklearn-boston-housing.py',  
    image_name='123456789012.dkr.ecr.eu-west-1.amazonaws.com/sklearn-custom:sklearn',  
    instance_count=1,  
    instance_type='ml.m5.large',  
    output_path=output,  
    hyperparameters={  
        'normalize': True,  
        'test-size': 0.1  
    }  
)
```

7. We set the location of the training channel and launch the training as usual. In the training log, we see that our code is indeed invoked with script mode:

```
/usr/local/bin/python -m sklearn-boston-housing  
--normalize True --test-size 0.1
```

As you can see, it's easy to customize training containers. Thanks to the SageMaker Training Toolkit, you can work just as with a built-in framework container. We used scikit-learn here, and you can do the same with all other frameworks.

However, we cannot use this container for deployment, as it doesn't contain any model-serving code. We should add bespoke code to launch a web app, which is exactly what we're going to do in the next example.

Building a fully custom container for scikit-learn

In this example, we're going to build a fully custom container without any AWS code. We'll use it to train a scikit-learn model on the Boston Housing dataset, using a generic `Estimator` module. With the same container, we'll deploy the model thanks to a Flask web application.

We'll proceed in a logical way, first taking care of the training, and then updating the code to handle deployment.

Training with a fully custom container

Since we can't rely on script mode anymore, the training code needs to be modified. This is what it looks like, and you'll easily figure out what's happening here:

```
#!/usr/bin/env python
import pandas as pd
import joblib, os, json
if __name__ == '__main__':
    config_dir = '/opt/ml/input/config'
    training_dir = '/opt/ml/input/data/training'
    model_dir = '/opt/ml/model'
    with open(os.path.join(config_dir,
                           'hyperparameters.json')) as f:
        hp = json.load(f)
        normalize = hp['normalize']
        test_size = float(hp['test-size'])
        random_state = int(hp['random-state'])
        filename = os.path.join(training_dir, 'housing.csv')
        data = pd.read_csv(filename)
```

```
# Train model
...
joblib.dump(regr,
            os.path.join(model_dir, 'model.joblib'))
```

Using the standard file layout for SageMaker containers, we read hyperparameters from their JSON file. Then, we load the dataset, train the model, and save it at the correct location.

There's another very important difference, and we have to dive a bit into Docker to explain it. SageMaker will run the training container as `docker run <IMAGE_ID> train`, passing the `train` argument to the entry point of the container.

If your container has a predefined entry point, the `train` argument will be passed to it, say, `/usr/bin/python train`. If your container doesn't have a predefined entry point, `train` is the actual command that will be run.

To avoid annoying issues, I recommend that your training code ticks the following boxes:

- Name it `train`—no extension, just `train`.
- Make it executable.
- Make sure it's in the `PATH` value.
- The first line of the script should define the path to the interpreter, for example, `#!/usr/bin/env python`.

This should guarantee that your training code is invoked correctly whether your container has a predefined entry point or not.

We'll take care of this in the Dockerfile, starting from an official Python image. Note that we're not installing the SageMaker Training Toolkit any longer:

```
FROM python:3.7
RUN pip3 install --no-cache scikit-learn numpy pandas joblib
COPY sklearn-boston-housing-generic.py /usr/bin/train
RUN chmod 755 /usr/bin/train
```

The name of the script is correct. It's executable, and `/usr/bin` is in `PATH`.

We should be all set—let's create our custom container and launch a training job with it:

1. We build and push the image, using a different tag:

```
$ docker build -t sklearn-custom:estimator -f Dockerfile-generic .
$ docker tag <IMAGE_ID> 123456789012.dkr.ecr.eu-west-1.amazonaws.com/sklearn-custom:estimator
$ docker push 123456789012.dkr.ecr.eu-west-1.amazonaws.com/sklearn-custom:estimator
```

2. We update our notebook code to use the generic Estimator module:

```
from sagemaker.estimator import Estimator
sk = Estimator(
    role=sagemaker.get_execution_role(),
    image_name='123456789012.dkr.ecr.eu-west-1.amazonaws.com/sklearn-custom:estimator',
    instance_count=1,
    instance_type='ml.m5.large',
    output_path=output,
    hyperparameters={
        'normalize': True,
        'test-size': 0.1,
        'random-state': 123
    }
)
```

3. We train as usual.

Now let's add code to deploy this model.

Deploying a fully custom container

Flask is a highly popular web framework for Python (<https://palletsprojects.com/p/flask>). It's simple and well documented. We're going to use it to build a simple prediction API hosted in our container.

Just like for our training code, SageMaker requires that the deployment script is copied inside the container. The image will be run as `docker run <IMAGE_ID> serve`.

HTTP requests will be sent to port 8080. The container must provide a /ping URL for health checks and an /invocations URL for prediction requests. We'll use CSV as the input format.

Hence, your deployment code needs to tick the following boxes:

- Name it `serve`—no extension, just `serve`.
 - Make it executable.
 - Make sure it's in `PATH`.
 - Make sure port 8080 is exposed by the container.
 - Provide code to handle the `/ping` and `/invocations` URLs.

Here's the updated Dockerfile. We install Flask, copy the deployment code, and open port 8080:

```
FROM python:3.7
RUN pip3 install --no-cache scikit-learn numpy pandas joblib
RUN pip3 install --no-cache flask
COPY sklearn-boston-housing-generic.py /usr/bin/train
COPY sklearn-boston-housing-serve.py /usr/bin/serve
RUN chmod 755 /usr/bin/train /usr/bin/serve
EXPOSE 8080
```

This is how we could implement a simple prediction service with Flask:

1. We import the required modules. We load the model from /opt/ml/model and initialize the Flask application:

2. We implement the /ping URL for health checks, by simply returning HTTP code 200 (OK):

```
@app.route("/ping", methods=["GET"])
def ping():
    return Response(response="\n", status=200)
```

3. We implement the /invocations URL. If the content type is not text/csv, we return HTTP code 415 (Unsupported Media Type). If it is, we decode the request body and store it in a file-like memory buffer. Then, we read the CSV samples, predict them, and send the results:

```
@app.route("/invocations", methods=["POST"])
def predict():
    if flask.request.content_type == 'text/csv':
        data = flask.request.data.decode('utf-8')
        s = StringIO(data)
        data = pd.read_csv(s, header=None)
        response = model.predict(data)
        response = str(response)
    else:
        return flask.Response(
            response='CSV data only',
            status=415, mimetype='text/plain')
    return Response(response=response, status=200)
```

4. At startup, the script launches the Flask app on port 8080:

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8080)
```

That's not too difficult, even if you're not yet familiar with Flask.

5. We rebuild and push the image, and then we train again with the same estimator. No change is required here.
6. We deploy the model:

```
sk_predictor = sk.deploy(instance_type='ml.t2.medium',
                        initial_instance_count=1)
```

Reminder

If you see some weird behavior here (the endpoint not deploying, cryptic error messages, and so on), Docker is probably hosed. `sudo service docker restart` should fix most problems. Cleaning `tmp*` cruft in `/tmp` may also help.

7. We prepare a couple of test samples, set the content type to `text/csv`, and invoke the prediction API:

```
test_samples = ['0.00632, 18.00, 2.310, 0, 0.5380,
6.5750, 65.20, 4.0900, 1,296.0, 15.30, 396.90, 4.98',
'0.02731, 0.00, 7.070, 0, 0.4690, 6.4210, 78.90, 4.9671,
2,242.0, 17.80, 396.90, 9.14']
sk_predictor.serializer =
    sagemaker.serializers.CSVSerializer()
response = sk_predictor.predict(test_samples)
print(response)
```

You should see something similar to this. The API has been successfully invoked:

```
b'[ [29.801388899699845], [24.990809475886078] ]'
```

8. When we're done, we delete the endpoint:

```
sk_predictor.delete_endpoint()
```

In the next example, we're going to train and deploy a model using the R environment. This will give us an opportunity to step out of the Python world for a bit. As you will see, things are not really different.

Building a fully custom container for R

R is a popular language for data exploration and analysis. In this example, we're going to build a custom container to train and deploy a linear regression model on the Boston Housing dataset.

The overall process is similar to building a custom container for Python. Instead of using Flask to build our prediction API, we'll use plumber (<https://www.rplumber.io>).

Coding with R and plumber

Don't worry if you're not familiar with R. This is a really simple example, and I'm sure you'll be able to follow along:

1. We write a function to train our model. It loads the hyperparameters and the dataset from the conventional paths. It normalizes the dataset if we requested it:

```
# train_function.R
library("rjson")
train <- function() {
  hp <- fromJSON(file =
    '/opt/ml/input/config/hyperparameters.json')
  normalize <- hp$normalize
  data <- read.csv(file =
    '/opt/ml/input/data/training/housing.csv',
    header=T)
  if (normalize) {
    data <- as.data.frame(scale(data))
  }
}
```

It trains a linear regression model, taking all features into account to predict the median house price (the medv column). Finally, it saves the model in the right place:

```
model = lm(medv~., data)
saveRDS(model, '/opt/ml/model/model.rds')
}
```

2. We write a function to serve predictions. Using plumber annotations, we define a /ping URL for health checks and an /invocations URL for predictions:

```
# serve_function.R
#' @get /ping
function() {
  return('')
}

#' @post /invocations
function(req) {
  model <- readRDS('/opt/ml/model/model.rds')
```

```
conn <- textConnection(gsub('\\\\n', '\n',
                           req$postBody))
data <- read.csv(conn)
close(conn)
medv <- predict(model, data)
return(medv)
}
```

3. Putting these two pieces together, we write a main function that will serve as the entry point for our script. SageMaker will pass either a `train` or `serve` command-line argument, and we'll call the corresponding function in our code:

```
library('plumber')
source('train_function.R')
serve <- function() {
    app <- plumb('serve_function.R')
    app$run(host='0.0.0.0', port=8080)
}
args <- commandArgs()
if (any(grepl('train', args))) {
    train()
}
if (any(grepl('serve', args))) {
    serve()
}
```

This is all of the R code that we need. Now, let's take care of the container.

Building a custom container

We need to build a custom container storing the R runtime, as well as our script. The Dockerfile is as follows:

1. We start from an official R image in **Docker Hub** and add the dependencies we need (these are the ones I needed on my machine; your mileage may vary):

```
FROM r-base:latest
WORKDIR /opt/ml/
RUN apt-get update
RUN apt-get install -y libcurl4-openssl-dev libsodium-dev
RUN R -e "install.packages(c('rjson', 'plumber')) "
```

2. Then, we copy our code inside the container and define the main function as its explicit entry point:

```
COPY main.R train_function.R serve_function.R /opt/ml/
ENTRYPOINT ["/usr/bin/Rscript", "/opt/ml/main.R", "--no-save"]
```

3. We create a new repository in ECR. Then, we build the image (this could take a while and involve compilation steps) and push it:

```
$ aws ecr create-repository --repository-name r-custom
--region eu-west-1
$ aws ecr get-login-password --region eu-west-1 | docker
login --username AWS --password-stdin 123456789012.dkr.
ecr.eu-west-1.amazonaws.com/r-custom:latest
$ docker build -t r-custom:latest -f Dockerfile .
$ docker tag <IMAGE_ID> 123456789012.dkr.ecr.eu-west-1.
amazonaws.com/r-custom:latest
$ docker push 123456789012.dkr.ecr.eu-west-1.amazonaws.
com/r-custom:latest
```

We're all set, so let's train and deploy.

Training and deploying a custom container on SageMaker

Jumping to a Jupyter notebook, we use the SageMaker SDK to train and deploy our container:

1. We configure an Estimator module with our custom container:

```
r_estimator = Estimator(  
    role = sagemaker.get_execution_role(),  
    image_uri='123456789012.dkr.ecr.eu-west-1.amazonaws.  
com/r-custom:latest',  
    instance_count=1,  
    instance_type='ml.m5.large',  
    output_path=output,  
    hyperparameters={'normalize': False}  
)  
r_estimator.fit({'training':training})
```

2. Once the training job is complete, we deploy the model as usual:

```
r_predictor = r_estimator.deploy(  
    initial_instance_count=1,  
    instance_type='ml.t2.medium')
```

3. Finally, we read the full dataset (why not?) and send it to the endpoint:

```
import pandas as pd  
data = pd.read_csv('housing.csv')  
data.drop(['medv'], axis=1, inplace=True)  
data = data.to_csv(index=False)  
r_predictor.serializer =  
    sagemaker.serializers.CSVSerializer()  
response = r_predictor.predict(data)  
print(response)
```

The output should look like this:

```
b'[30.0337,25.0568,30.6082,28.6772,27.9288. . .
```

4. When we're done, we delete the endpoint:

```
r_predictor.delete_endpoint()
```

Whether you're using Python, R, or something else, it's reasonably easy to build and deploy your own custom container. Still, you need to build your own little web application, which is something you may neither know how to do nor enjoy doing. Wouldn't it be nice if we had a tool that took care of all of that pesky container and web stuff?

As a matter of fact, there is one: **MLflow**.

Training and deploying with your own code on MLflow

MLflow is an open source platform for machine learning (<https://mlflow.org>). It was initiated by Databricks (<https://databricks.com>), who also brought us **Spark**. MLflow has lots of features, including the ability to deploy Python-trained models on SageMaker.

This section is not intended to be an MLflow tutorial. You can find documentation and examples at <https://www.mlflow.org/docs/latest/index.html>.

Installing MLflow

On our local machine, let's set up a virtual environment for MLflow and install the required libraries. The following example was tested with MLflow 1.17:

1. We first initialize a new virtual environment named `mlflow-example`. Then, we activate it:

```
$ virtualenv mlflow-example  
$ source mlflow-example/bin/activate
```

2. We install MLflow and the libraries required by our training script:

```
$ pip install mlflow gunicorn pandas sklearn xgboost  
boto3
```

- Finally, we download the Direct Marketing dataset we already used with XGBoost in *Chapter 7, Extending Machine Learning Services Using Built-In Frameworks*:

```
$ wget -N https://sagemaker-sample-data-us-west-2.s3-us-west-2.amazonaws.com/autopilot/direct_marketing/bank-additional.zip  
$ unzip -o bank-additional.zip
```

The setup is complete. Let's train the model.

Training a model with MLflow

The training script sets the MLflow experiment for this run so that we may log metadata (hyperparameters, metrics, and so on). Then, it loads the dataset, trains an XGBoost classifier, and logs the model:

```
# train-xgboost.py  
import mlflow.xgboost  
import xgboost as xgb  
from load_dataset import load_dataset  
if __name__ == '__main__':  
    mlflow.set_experiment('dm-xgboost')  
    with mlflow.start_run(run_name='dm-xgboost-basic')  
        as run:  
            x_train, x_test, y_train, y_test = load_dataset(  
                'bank-additional/bank-additional-full.csv')  
            cls = xgb.XGBClassifier(  
                objective='binary:logistic',  
                eval_metric='auc')  
            cls.fit(x_train, y_train)  
            auc = cls.score(x_test, y_test)  
            mlflow.log_metric('auc', auc)  
            mlflow.xgboost.log_model(cls, 'dm-xgboost-model')  
            mlflow.end_run()
```

The `load_dataset()` function does what its name implies and logs several parameters:

```
# load_dataset.py  
import mlflow  
import pandas as pd
```

```

from sklearn.model_selection import train_test_split
def load_dataset(path, test_size=0.2, random_state=123):
    data = pd.read_csv(path)
    data = pd.get_dummies(data)
    data = data.drop(['y_no'], axis=1)
    x = data.drop(['y_yes'], axis=1)
    y = data['y_yes']
    mlflow.log_param("dataset_path", path)
    mlflow.log_param("dataset_shape", data.shape)
    mlflow.log_param("test_size", test_size)
    mlflow.log_param("random_state", random_state)
    mlflow.log_param("one_hot_encoding", True)
    return train_test_split(x, y, test_size=test_size,
                           random_state=random_state)

```

Let's train the model and visualize its results in the MLflow web application:

1. Inside the virtual environment we just created on our local machine, we run the training script just like any Python program:

```

$ python train-xgboost.py
INFO: 'dm-xgboost' does not exist. Creating a new
experiment
AUC 0.91442097596504

```

2. We launch the MLflow web application:

```
$ mlflow ui &
```

3. Pointing our browser at `http://localhost:5000`, we see information on our run, as shown in the following screenshot:

							Parameters >		Metrics	
	Start Time	Run Name	User	Source	Version	Models	dataset_path	dataset_shape	one_hot_encoding	auc
□	2021-05-26 21:	dm-xgboost-basic	julsimon	train-xgboost.py	-	xgboost	bank-additio...	(41188, 64)	True	0.914
Load more										

Figure 8.2 – Viewing our job in MLflow

The training was successful. Before we can deploy the model on SageMaker, we must build a SageMaker container. As it turns out, it's the simplest thing.

Building a SageMaker container with MLflow

All it takes is a single command on our local machine:

```
$ mlflow sagemaker build-and-push-container
```

MLflow will automatically build a Docker container compatible with SageMaker, with all required dependencies. Then, it creates a repository in Amazon ECR named `mlflow-pyfunc` and pushes the image to it. Obviously, this requires your AWS credentials to be properly set up. MLflow will use the default region configured by the AWS CLI.

Once this command completes, you should see the image in ECR, as shown in the following screenshot:

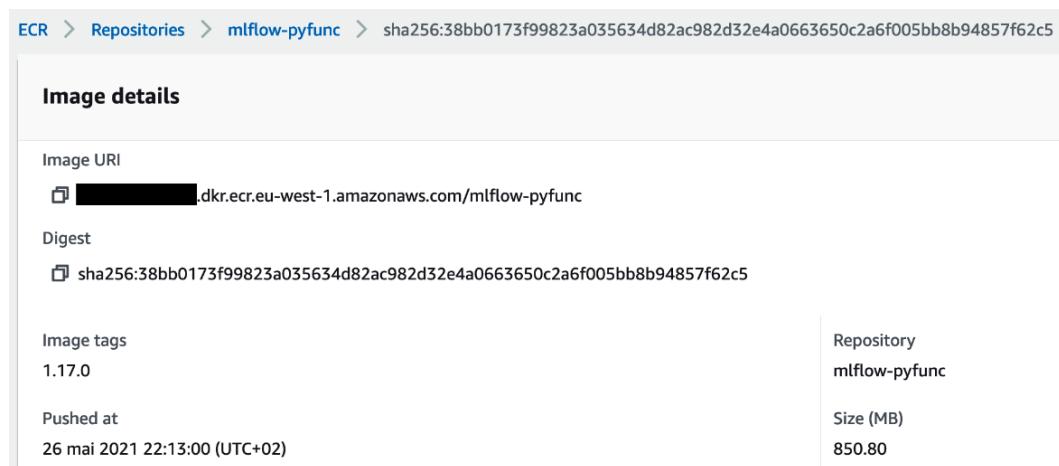


Figure 8.3 – Viewing our container in ECR

Our container is now ready for deployment.

Deploying a model locally with MLflow

We will deploy our model using the following steps:

1. We can deploy our model locally with a single command, passing its run identifier (visible in the MLflow URL for the run) and the HTTP port to use. This fires up a local web application based on gunicorn:

```
$ mlflow sagemaker run-local -p 8888 -m runs:/d08ab8383ee84f72a92164d3ca548693/dm-xgboost-model
```

You should see something similar to this:

```
[2021-05-26 20:21:23 +0000] [370] [INFO] Starting
gunicorn 20.1.0
[2021-05-26 20:21:23 +0000] [370] [INFO] Listening at:
http://127.0.0.1:8000 (370)
[2021-05-26 20:21:23 +0000] [370] [INFO] Using worker:
gevent
[2021-05-26 20:21:23 +0000] [381] [INFO] Booting worker
with pid: 381
```

2. Our prediction code is quite straightforward. We load CSV samples from the dataset, convert them into JSON format, and send them to the endpoint using the `requests` library, a popular Python library for HTTP (<https://requests.readthedocs.io>):

```
# predict-xgboost-local.py
import json
import requests
from load_dataset import load_dataset
port = 8888
if __name__ == '__main__':
    x_train, x_test, y_train, y_test = load_dataset(
        'bank-additional/bank-additional-full.csv')
    input_data = x_test[:10].to_json(orient='split')
    endpoint = 'http://localhost:{}/invocations'
        .format(port)
    headers = {'Content-type': 'application/json;
               format=pandas-split'}
    prediction = requests.post(
        endpoint,
        json=json.loads(input_data),
        headers=headers)
    print(prediction.text)
```

3. Running this code in another shell invokes the local model and prints out predictions:

```
$ source mlflow-example/bin/activate
$ python predict-xgboost-local.py
[0.00046298891538754106, 0.10499032586812973, . . .]
```

4. When we're done, we terminate the local server with *Ctrl + C*.

Now that we're confident that our model works locally, we can deploy it on SageMaker.

Deploying a model on SageMaker with MLflow

This is a one-liner again:

1. We need to pass an application name, the model path, and the name of the SageMaker role. You can use the same role you've used in previous chapters:

```
$ mlflow sagemaker deploy \
--region-name eu-west-1 \
-t ml.t2.medium \
-a mlflow-xgb-demo \
-m runs:/d08ab8383ee84f72a92164d3ca548693/dm-xgboost-
model \
-e arn:aws:iam::123456789012:role/Sagemaker-fullaccess
```

2. After a few minutes, the endpoint is in service. We invoke it with the following code. It loads the test dataset and sends the first 10 samples in JSON format to the endpoint named after our application:

```
# predict-xgboost.py
import boto3
from load_dataset import load_dataset
app_name = 'mlflow-xgb-demo'
region = 'eu-west-1'
if __name__ == '__main__':
    sm = boto3.client('sagemaker', region_name=region)
    smrt = boto3.client('runtime.sagemaker',
                        region_name=region)
    endpoint = sm.describe_endpoint(
        EndpointName=app_name)
```

```
print("Status: ", endpoint['EndpointStatus'])
x_train, x_test, y_train, y_test = load_dataset(
    'bank-additional/bank-additional-full.csv')
input_data = x_test[:10].to_json(orient="split")
prediction = smrt.invoke_endpoint(
    EndpointName=app_name,
    Body=input_data,
    ContentType='application/json',
    format='pandas-split')
prediction = prediction['Body']
.read().decode("ascii")
print(prediction)
```

Wait a minute! We are not using the SageMaker SDK. What's going on here?

In this example, we're dealing with an existing endpoint, not an endpoint that we created by fitting an estimator and deploying a predictor.

We could still rebuild a predictor using the SageMaker SDK, as we'll see in *Chapter 11, Deploying Machine Learning Models*. Instead, we use our good old friend `boto3`, the AWS SDK for Python. We first invoke the `describe_endpoint()` API to check that the endpoint is in service. Then, we use the `invoke_endpoint()` API to...invoke the endpoint! For now, we don't need to know more.

We run the prediction code on our local machine, and it produces the following output:

```
$ python3 predict-xgboost.py
Status: InService
[0.00046298891538754106, 0.10499032586812973,
0.016391035169363022, . . .]
```

3. When we're done, we delete the endpoint with the MLflow CLI. This cleans up all resources created for deployment:

```
$ mlflow sagemaker delete -a mlflow-xgb-demo -region-name
eu-west-1
```

The development experience with MLflow is pretty simple. It also has plenty of other features you may want to explore.

So far, we've run examples for training and prediction. There's another area of SageMaker that lets us use custom containers, **SageMaker Processing**, which we studied in *Chapter 2, Handling Data Preparation Techniques*. To close this chapter, let's build a custom Python container for SageMaker Processing.

Building a fully custom container for SageMaker Processing

We'll reuse the news headlines example from *Chapter 6, Training Natural Processing Models*:

1. We start with a Dockerfile based on a minimal Python image. We install dependencies, add our processing script, and define it as our entry point:

```
FROM python:3.7-slim
RUN pip3 install --no-cache gensim nltk sagemaker
RUN python3 -m nltk.downloader stopwords wordnet
ADD preprocessing-lda-ntm.py /
ENTRYPOINT ["python3", "/preprocessing-lda-ntm.py"]
```

2. We build the image and tag it as sm-processing-custom:latest:

```
$ docker build -t sm-processing-custom:latest -f
Dockerfile .
```

The resulting image is 497 MB. For comparison, it's 1.2 GB if we start from `python:3.7` instead of `python:3.7-slim`. This makes it faster to push and download.

3. Using the AWS CLI, we create a repository in Amazon ECR to host this image, and we log in to the repository:

```
$ aws ecr create-repository --repository-name
sm-processing-custom --region eu-west-1
$ aws ecr get-login-password | docker login --username
AWS --password-stdin 123456789012.dkr.ecr.eu-west-1.
amazonaws.com/sm-processing-custom:latest
```

4. Using the image identifier, we tag the image with the repository identifier:

```
$ docker tag <IMAGE_ID> 123456789012.dkr.ecr.eu-west-1.
amazonaws.com/sm-processing-custom:latest
```

5. We push the image to the repository:

```
$ docker push 123456789012.dkr.ecr.eu-west-1.amazonaws.
com/sm-processing-custom:latest
```

6. Moving to a Jupyter notebook, we configure a generic Processor object with our new container, which is the equivalent of the generic Estimator module we used for training. Accordingly, no framework_version parameter is required:

```
from sagemaker.processing import Processor
sklearn_processor = Processor(
    image_uri='123456789012.dkr.ecr.eu-west-1.amazonaws.
com/sm-processing-custom:latest',
    role=sagemaker.get_execution_role(),
    instance_type='ml.c5.2xlarge',
    instance_count=1)
```

7. Using the same ProcessingInput and ProcessingOutput objects, we run the processing job. As our processing code is now stored inside the container, we don't need to pass a code parameter as we did with SKLearnProcessor:

```
from sagemaker.processing import ProcessingInput,
ProcessingOutput
sklearn_processor.run(
    inputs=[
        ProcessingInput(
            source=input_data,
            destination='/opt/ml/processing/input')
    ],
    outputs=[
        ProcessingOutput(
            output_name='train_data',
            source='/opt/ml/processing/train/')
    ],
    arguments=[
        '--filename', 'abcnews-date-text.csv.gz'
    ]
)
```

8. Once the training job is complete, we can fetch its outputs in S3.

This concludes our exploration of custom containers in SageMaker. As you can see, you can pretty much run anything as long as it fits inside a Docker container.

Summary

Built-in frameworks are extremely useful, but sometimes you need something a little—or very—different. Whether starting from built-in containers or from scratch, SageMaker lets you build your training and deployment containers exactly the way you want them. Freedom for all!

In this chapter, you learned how to customize Python and R containers for data processing, training, and deployment. You saw how you could use them with the SageMaker SDK and its usual workflow. You also learned about MLflow, a nice open source tool that lets you train and deploy models using a CLI.

This concludes our extensive coverage of modeling options in SageMaker: built-in algorithms, built-in frameworks, and custom code. In the next chapter, you'll learn about SageMaker features that help you to scale your training jobs.

Section 3: Diving Deeper into Training

In this section, you will learn advanced training techniques relating to scaling, model optimization, model debugging, and cost optimization.

This section comprises the following chapters:

- *Chapter 9, Scaling Your Training Jobs*
- *Chapter 10, Advanced Training Techniques*

9

Scaling Your Training Jobs

In the four previous chapters, you learned how to train models with built-in algorithms, frameworks, or your own code.

In this chapter, you'll learn how to scale training jobs, allowing them to train on larger datasets while keeping training time and cost under control. We'll start by discussing when and how to take scaling decisions, thanks to monitoring information and simple guidelines. You'll also see how to collect profiling information with **Amazon SageMaker Debugger**, in order to understand how efficient your training jobs are. Then, we'll look at several key techniques for scaling: **pipe mode**, **distributed training**, **data parallelism**, and **model parallelism**. After that, we'll launch a large training job on the large **ImageNet** dataset and see how to scale it. Finally, we'll discuss storage alternatives to **S3** for large-scale training, namely **Amazon EFS** and **Amazon FSx for Lustre**.

We'll cover the following topics:

- Understanding when and how to scale
- Monitoring and profiling training jobs with Amazon SageMaker Debugger
- Streaming datasets with pipe mode
- Distributing training jobs
- Scaling an image classification model on ImageNet

- Training with data and model parallelism
- Using other storage services

Technical requirements

You will need an AWS account to run the examples included in this chapter. If you haven't got one already, please point your browser at <https://aws.amazon.com/getting-started/> to create it. You should also familiarize yourself with the AWS Free Tier (<https://aws.amazon.com/free/>), which lets you use many AWS services for free within certain usage limits.

You will need to install and configure the AWS **Command Line Interface (CLI)** for your account (<https://aws.amazon.com/cli/>).

You will need a working **Python 3.x** environment. Installing the **Anaconda** distribution (<https://www.anaconda.com/>) is not mandatory but strongly encouraged as it includes many projects that we will need (**Jupyter**, **pandas**, **numpy**, and more).

Code examples included in this book are available on GitHub at <https://github.com/PacktPublishing/Learn-Amazon-SageMaker-second-edition>. You will need to install a Git client to access them (<https://git-scm.com/>).

Understanding when and how to scale

Before we dive into scaling techniques, let's first discuss the monitoring information that we should consider when deciding whether we need to scale, and how we should do it.

Understanding what scaling means

The training log tells us how long the job lasted. In itself, this isn't really useful. How long is *too long*? This feels very subjective, doesn't it? Furthermore, even when training on the same dataset and infrastructure, changing a single hyperparameter can significantly impact training time. Batch size is one example of this, and there are many more.

When we're concerned about training time, I think we're really trying to answer three questions:

- Is the training time compatible with our business requirements?
- Are we making good use of the infrastructure we're paying for? Did we underprovision or overprovision?
- Could we train faster without spending more money?

Adapting training time to business requirements

Ask yourself this question—what would be the direct impact on your business if your training job ran twice as fast? In many cases, the honest answer should be *none*. There is no clear business metric that would be improved.

Sure, some companies run training jobs that last days, even weeks—think autonomous driving or life sciences. For them, any significant reduction in training time means that they get results much faster, analyze them, and launch the next iteration.

Some other companies want the freshest models possible, and they retrain every hour. Of course, training time needs to be kept under control to make the deadline.

In both types of companies, scaling is vital. For everyone else, things are not so clear. If your company trains a production model every week or every month, does it really matter whether training reaches the same level of accuracy 30 minutes sooner? Probably not.

Some people would certainly object that they need to train a lot of models all of the time. I'm afraid this is a fallacy. As SageMaker lets you create on-demand infrastructure whenever you need it, training activities will not be capacity-bound. This is the case when you work with physical infrastructure, but not with cloud infrastructure. Even if you need to train 1,000 **XGBoost** jobs every day, does it really matter whether each individual job takes 5 minutes instead of 6? Probably not.

Some would retort that "the faster you train, the less it costs." Again, this is a fallacy. The cost of a SageMaker training job is the training time in seconds multiplied by the cost of the instance type and by the number of instances. If you pick a larger instance type, training time will most probably decrease. Will it decrease enough to offset the increased instance cost? Maybe, maybe not. Some training workloads will make good use of the extra infrastructure, and some won't. The only way to know is to run tests and make data-driven decisions.

Right-sizing training infrastructure

SageMaker supports a long list of instance types, which looks like a very nice candy store (<https://aws.amazon.com/sagemaker/pricing/instance-types>). All you have to do is call an API to fire up an 8 GPU EC2 instance – more powerful than any server your company would have allowed you to buy. Caveat emptor – don't forget the "pricing" part of the URL!

Note

If the words "EC2 instance" don't mean much to you, I would definitely recommend reading a bit about **Amazon EC2** at <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html>.

Granted, cloud infrastructure doesn't require you to pay a lot of money upfront to buy and host servers. Still, the AWS bill will come at the end of the month. Hence, even using cost optimization techniques such as **Managed Spot Training** (which we'll discuss in the next chapter), it's critical that you right-size your training infrastructure.

My advice is always the same:

- Identify business requirements that depend on training time.
- Start with the smallest reasonable amount of infrastructure.
- Measure technical metrics and cost.
- If business requirements are met, did you overprovision? There are two possible answers:
 - a) **Yes:** Scale down and repeat.
 - b) **No:** You're done.
- If business requirements are not met, identify bottlenecks.
- Run some tests on scaling up (larger instance type) and scaling out (more instances).
- Measure technical metrics and costs.
- Implement the best solution for your business context.
- Repeat.

Of course, this process is as good as the people who take part in it. Be critical! "Too slow" is not a data point—it's an opinion.

Deciding when to scale

When it comes to monitoring information, you can rely on three sources: the training log, **Amazon CloudWatch** metrics, and the profiling capability in **Amazon SageMaker Debugger**.

Note

If "CloudWatch" doesn't mean much to you, I would definitely recommend reading a bit about it at <https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/>.

The training log shows you the total training time and the number of samples per second. As discussed in the previous section, total training time is not a very useful metric. Unless you have very strict deadlines, it's best to ignore it. The number of samples per second is more interesting. You can use it to compare your training job to benchmarks available in research papers or blog posts. If someone has managed to train the same model twice as fast on the same GPU, you should be able to do the same. When you get close to that number, you'll also know that there's not a lot of room for improvement and that other scaling techniques should be considered.

CloudWatch gives you coarse-grained infrastructure metrics with a 1-minute resolution. For simple training jobs, these metrics are all you need to check if your training makes efficient use of the underlying infrastructure and identify potential bottlenecks.

For more complex jobs (distributed training, custom code, and so on), SageMaker Debugger gives you fine-grained, near real-time infrastructure and Python metrics, with a resolution as low as 100 milliseconds. This information will let you drill down and identify complex performance and scaling problems.

Deciding how to scale

As mentioned earlier, you can either scale up (move to a bigger instance) or scale out (use several instances for distributed training). Let's look at the pros and cons.

Scaling up

Scaling up is simple. You just need to change the instance type. Monitoring stays the same, and there's only one training log to read. Last but not least, training on a single instance is predictable and very often delivers the best accuracy, as there's only one set of model parameters to learn and update.

On the downside, your algorithm may not be compute-intensive and parallel enough to benefit from the extra computing power. Extra vCPUs and GPUs are only useful if they're put to work. Your network and storage layers must also be fast enough to keep them busy at all times, which may require using alternatives to S3, generating some extra engineering work. Even if you don't hit any of these problems, there comes a point where there simply isn't a bigger instance you can use!

Scaling up with multi-GPU instances

As tempting as multi-GPU instances are, they create specific challenges. An **NVIDIA V100** GPU has 5,120 cores and 640 tensor cores. It takes a lot of CPU and I/O power to keep them 100% busy, and adding more GPUs on the same instance only increases that pressure. You may quickly get to a point where GPUs are stalled, wasting time and money on under-utilized infrastructure. Reducing network and storage latency helps, which is why monster instances such as `m1.g4dn.16xlarge` and `m1.p3dn.24xlarge` support 100-Gbit networking and ultra-fast SSD NVMe local storage. Still, that level of performance comes at a price, and you need to make sure it's really worth it.

You should keep in mind that bigger isn't always better. Inter-GPU communication, no matter how fast, introduces some overhead that could kill the performance of smaller training jobs. Here too, you should experiment and find the sweetest spot.

In my experience, getting great performance with multi-GPU instances takes some work. Unless the model is too large to fit on a single GPU or the algorithm doesn't support distributed training, I'd recommend trying first to scale out on single-GPU instances.

Scaling out

Scaling out lets you distribute large datasets to a cluster of training instances. Even if your training job doesn't scale linearly, you'll get a noticeable speedup compared to single-instance training. You can use plenty of smaller instances that only process a subset of your dataset, which helps to keep costs under control.

On the downside, datasets need to be prepared in a format that can be efficiently distributed across training clusters. As distributed training is pretty chatty, network I/O can also become a bottleneck. Still, the main problem is usually accuracy, which is often lower than for single-instance training, as each instance works with its own set of model parameters. This can be alleviated by asking training instances to synchronize their work periodically, but this is a costly operation that impacts training time.

If you think that scaling is harder than it seems, you're right. Let's try to put all of these notions into practice with a first simple example.

Scaling a **BlazingText** training job

In *Chapter 6, Training Natural Language Processing Models*, we used **BlazingText** and the Amazon reviews dataset to train a sentiment analysis model. At the time, we only trained it on 100,000 reviews. This time, we'll train it on the full dataset: 1.8 million reviews (151 million words).

Reusing our SageMaker Processing notebook, we process the full dataset on an `m1.c5.9xlarge` instance, store results in S3, and feed them to our training job. The size of the training set has grown to a respectable 720 MB.

To give BlazingText extra work, we apply the following hyperparameters to increase the complexity of the word vectors the job will learn:

```
bt.set_hyperparameters(mode='supervised', vector_dim=300, word_ngrams=3, epochs=50)
```

We train on a single `m1.c5.2xlarge` instance. It has 8 vCPU and 16 GB of RAM and uses **EBS** network storage (the `gp2` class, which is SSD-based).

The job runs for 2,109 seconds (a little more than 35 minutes), peaking at 4.84 million words per second. Let's take a look at the CloudWatch metrics:

1. Starting from the **Experiments and trials** panel in **SageMaker Studio**, we locate the training job and right-click on **Open in trial details**.
2. Then, we select the **AWS settings** tab. Scrolling down, we see a link named **View instance metrics**. Clicking on it takes us directly to the CloudWatch metrics for our training job.
3. Let's select **CPUUtilization** and **MemoryUtilization** in **All metrics** and visualize them as shown in the next screenshot:

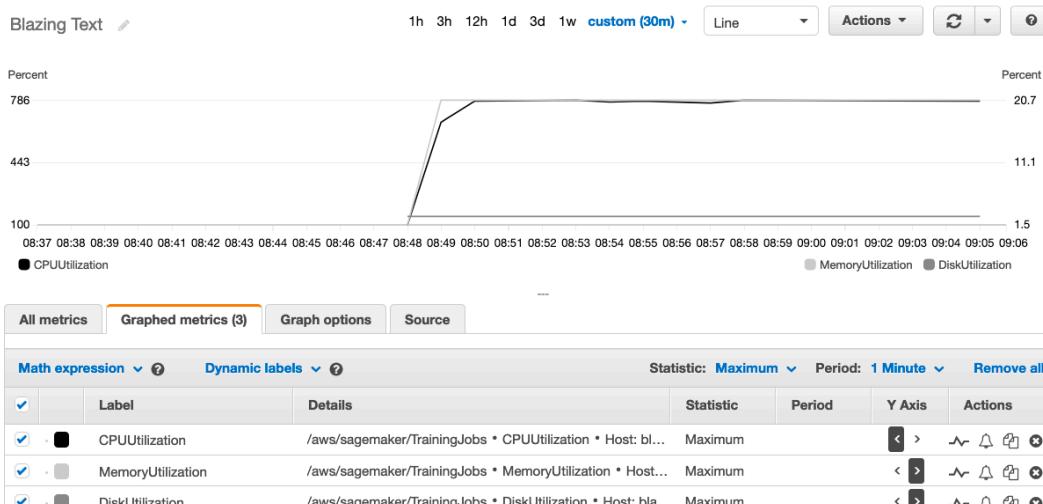


Figure 9.1 – Viewing CloudWatch metrics

On the right-hand Y-axis, memory utilization is stable at 20%, so we definitely don't need more RAM.

Still on the right-hand Y-axis, disk utilization is about 3% during the training, going up to 12% when the model is saved. We allocated way too much storage to this instance. By default, SageMaker instances get 30 GB of Amazon EBS storage, so how much money did we waste here? The EBS cost for SageMaker in `eu-west-1` is \$0.154 per GB-month, so 30 GB for 2,117 seconds costs $0.154 \times 30 \times (2109 / (24 \times 30 \times 3600)) = \0.00376 . That's a silly low amount, but if you train thousands of jobs per month, it will add up. Even if this saves us \$10 a year, we should save that! This can easily be done by setting the `volume_size` parameter in all estimators.

On the left-hand Y-axis, we see that the CPU utilization plateaus around 790%, very close to the maximum value of 800% (8 vCPUs at 100% usage). This job is obviously compute-bound.

So, what are our options? If BlazingText supported distributed training in supervised mode (it doesn't), we could have considered scaling out with smaller `m1.c5.xlarge` instances (4 vCPUs and 8 GB of RAM). That's more than enough RAM, and adding capacity in small chunks is good practice. This is what right-sizing is all about: not too much, not too little—it should be just right.

Anyway, our only choice here is to scale up. Looking at the list of available instances, we could try `m1.c5.4xlarge`. As BlazingText supports single-GPU acceleration, `m1.p3.2xlarge` (1 NVIDIA V100 GPU) is also an option.

Note

At the time of writing, the cost-effective `m1.g4dn.xlarge` is unfortunately not supported by BlazingText.

Let's try both and compare training times and costs.

Instance type	vCPUs	RAM	Time	Samples per second	Resource utilization	Compute cost (eu-west-1 prices)
ml.c5.2xlarge	8	16	2109	4.84M	CPU: 790% RAM: 20.7%	$(2109/3600) * \$0.461 = \0.270
ml.c5.4xlarge	16	32	1320 (-37%)	8.82M (+82%)	CPU: 1578% RAM: 9.5%	$(1320/3600) * \$0.922 = \0.338 (+25%)
ml.c5.9xlarge	36	72	1128 (-46%)	11.23M (+132%)	CPU : 3560% RAM : 3.9%	$(1128/3600)*\$2.074 = \0.649
ml.p3.2xlarge	8	61	747 (-64%)	31.2M (+544%)	CPU: 792% GPU: 98% RAM: 4.5% GPU RAM: 17%	$(747/3600) * \$4.131 = \0.857 (+217%)

The `ml.c5.4xlarge` instance provides a nice speedup for a moderate price increase. Interestingly, the job is still compute-bound, so I decided to try the even larger `ml.c5.9xlarge` instance (36 vCPUs) for good measure, but the speedup was large enough to offset the increased cost.

The GPU instance is almost 3x faster, as BlazingText has been optimized to utilize thousands of cores. It's also about 3x more expensive, which could be acceptable if minimizing training time was very important.

This simple example shows you that right-sizing your training infrastructure is not black magic. By following simple rules, looking at a few metrics, and using common sense, you can find the right instance size for your project.

Now, let's introduce the monitoring and profiling capability in Amazon SageMaker Debugger, which will give us even more information on the performance of our training jobs.

Monitoring and profiling training jobs with Amazon SageMaker Debugger

SageMaker Debugger includes a monitoring and profiling capability that lets us collect infrastructure and code performance information at much lower time resolution than CloudWatch (as often as every 100 milliseconds). It also allows us to configure and trigger built-in or custom rules that watch for unwanted conditions in our training jobs.

Profiling is very easy to use, and in fact, it's on by default! You may have noticed a line such as this one in your training log:

```
2021-06-14 08:45:30 Starting - Launching requested ML
instancesProfilerReport-1623660327: InProgress
```

This tells us that SageMaker is automatically running a profiling job, in parallel with our training job. The role of the profiling job is to collect data points that we can then display in SageMaker Studio, in order to visualize metrics and understand potential performance issues.

Viewing monitoring and profiling information in SageMaker Studio

Let's go back to the **Experiments and trials** view and locate the BlazingText training job we just ran on an `m1.p3.2xlarge` instance. We right-click on it and select **Open Debugger for insights** this time. This opens a new tab, visible in the next screenshot:

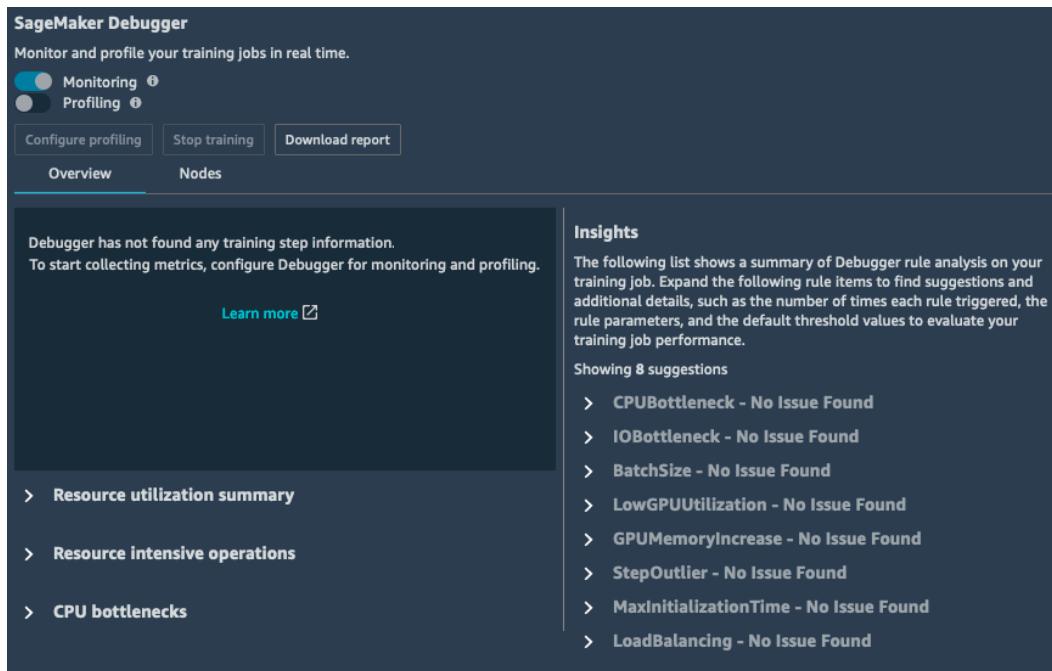


Figure 9.2 – Viewing monitoring and profiling information

At the top, we can see that monitoring is indeed on by default and that profiling isn't. Expanding the **Resource utilization summary** item in the **Overview** tab, we see a summary of infrastructure metrics, as shown in the next screenshot:

Resource utilization summary							
System usage statistics							
Node	Metric	Unit	Max	p99	p95	p50	Min
algo-1	Network		2544.6	139.98	0	0	0
algo-1	GPU		99	99	99	98	0
algo-1	CPU		100	99.5	99.25	98.18	0
algo-1	CPU memory		9.07	9.07	7.02	6.9	2.06
algo-1	GPU memory		37	37	36	35	0
algo-1	I/O		57.91	50.97	9.3	0	0

Figure 9.3 – Viewing utilization summary

Note

P50, p95, and p99 are percentiles. If you're not familiar with this concept, you can find more information at https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/cloudwatch_concepts.html#Percentiles.

Moving on to the **Nodes** tab, we see metrics graphed over time for each instance in the training cluster. Here, our job involved a single instance named algo-1. For example, you can see its GPU utilization in the next screenshot:

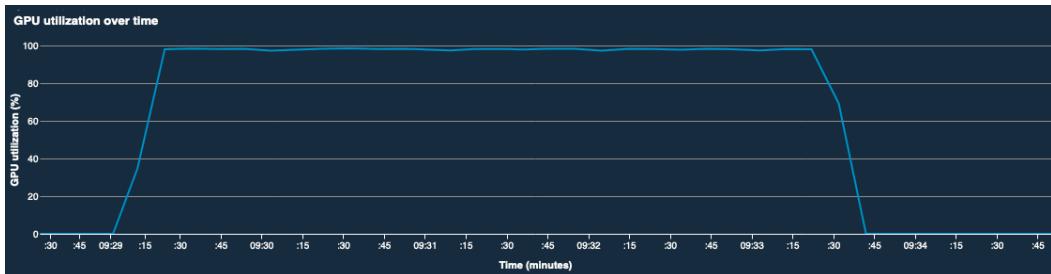


Figure 9.4 – Viewing GPU utilization over time

We also get a very nice view of system utilization over time, with one line per vCPU and GPU, as shown in the next screenshot:

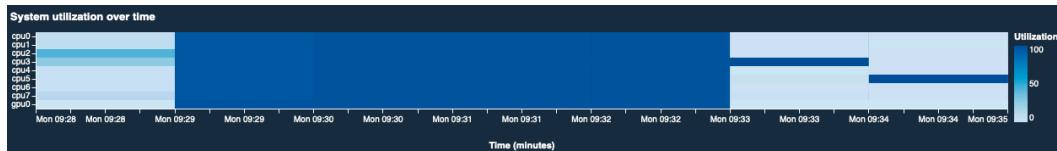


Figure 9.5 – Viewing system utilization over time

All this information is updated in near-real-time while your training job is running. Just launch a training job, open this view, and, after a few minutes, the graphs will show up and get updated.

Now, let's see how we can enable detailed profiling information in our training jobs.

Enabling profiling in SageMaker Debugger

Profiling collects framework metrics (**TensorFlow**, **PyTorch**, **Apache MXNet**, and **XGBoost**), data loader metrics, and Python metrics. For the latter, we can use **CProfile** or **Pyinstrument**.

Profiling can be configured in the estimator (which is the option we'll use). You can also enable it manually in SageMaker Studio on a running job (see the slider in *Figure 9.2*).

Let's reuse our TensorFlow/Keras example from *Chapter 6, Training Computer Vision Models*, and collect all profiling information every 100 milliseconds:

1. First, we create a `FrameworkProfile` object containing default settings for the profiling, data loading, and Python configurations. For each one of these, we could specify precise time ranges or step ranges for data collection:

```
from sagemaker.debugger import FrameworkProfile,
    DetailedProfilingConfig, DataloaderProfilingConfig,
    PythonProfilingConfig, PythonProfiler
framework_profile_params = FrameworkProfile(
    detailed_profiling_config=DetailedProfilingConfig(),
    dataloader_profiling_config=DataloaderProfilingConfig(),
    python_profiling_config=PythonProfilingConfig(
        python_profiler=PythonProfiler.PYINSTRUMENT)
)
```

2. Then, we create a `ProfilerConfig` object that sets framework parameters and the time interval for data collection:

```
from sagemaker.debugger import ProfilerConfig
profiler_config = ProfilerConfig(
    system_monitor_interval_millis=100,
    framework_profile_params=framework_profile_params)
```

3. Finally, we pass this configuration to our estimator, and train as usual:

```
tf_estimator = TensorFlow(
    entry_point='fmnist.py',
    ...
    profiler_config=profiler_config)
```

4. As the training job runs, profiling data is automatically collected and saved in a default location in S3 (you can define a custom path with the `s3_output_path` parameter in `ProfilingConfig`). We could also use the `smdebug` **SDK** (<https://github.com/awslabs/sagemaker-debugger>) to load and inspect profiling data.

5. Shortly after the training job completes, we see summary information in the **Overview** tab, as shown in the next screenshot:

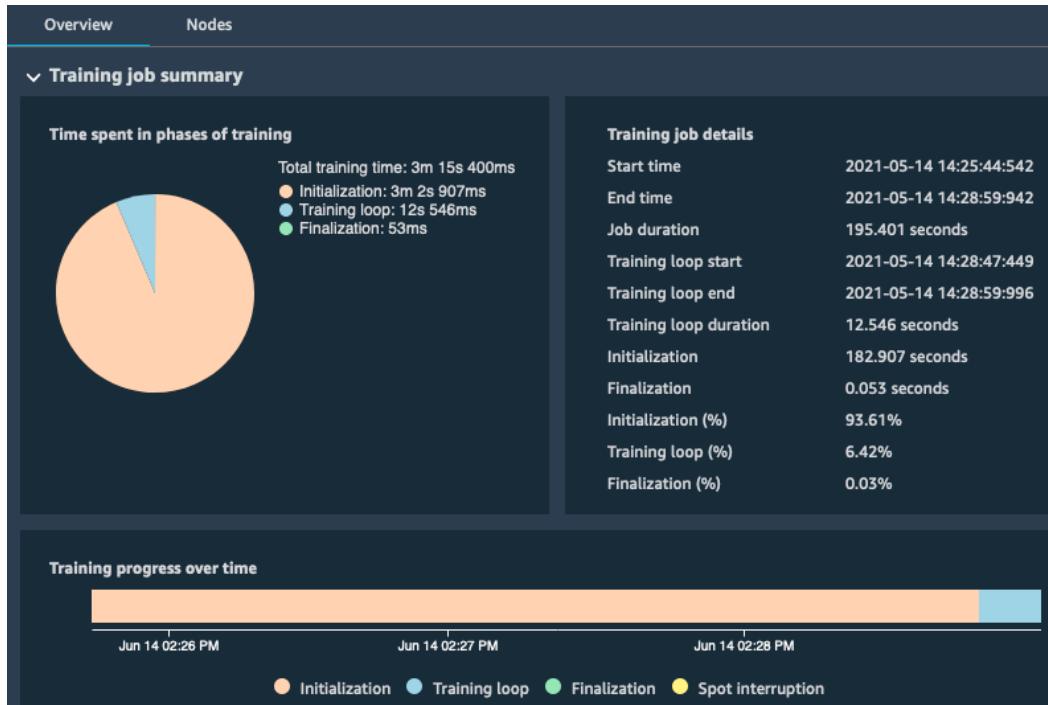


Figure 9.6 – Viewing profiling information

6. We can also download a detailed report in HTML format (see the button in *Figure 9.2*). For example, it tells us which are the most expensive GPU operators. Unsurprisingly, we see our `fmnist_model` function and the TensorFlow operator for 2D convolution, as visible in the next screenshot:

Overview: GPU operators

The following table shows a list of operators that your training job ran on GPU. The most expensive operator on GPU was "EagerKernelExecute 0" with 19 %

#	Percentage	Cumulative time in microse	GPU operator
0	19.67	31331.5	EagerKernelExecute 0
1	17.29	27538.75	fmnist_model
2	16.29	25949.5	gradient_tape
3	15.79	25148.5	Conv2D
4	15.76	25104.75	conv2d_1
5	8.56	13626	Conv2DBackpropInput
6	6.64	10574.5	Conv2DBackpropFilter

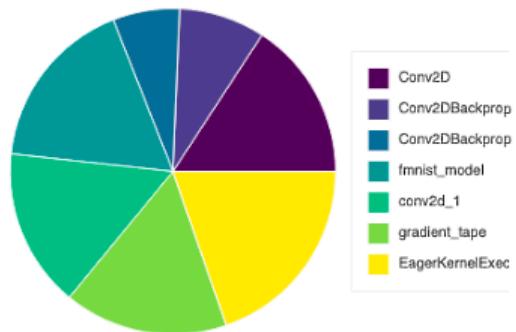


Figure 9.7 – Viewing the profiling report

The report also contains information on built-in rules that have been triggered during training, warning us about conditions such as low GPU usage, CPU bottlenecks, and more. These rules have default settings that can be customized if needed. We'll cover rules in more details in the next chapter when we'll discuss how to use SageMaker Debugger to debug training jobs.

For now, let's look at some common scaling issues for training jobs, and how we could address them. In the process, we'll mention several SageMaker features that will be covered in the rest of this chapter.

Solving training challenges

We will dive into the challenges, and their solutions, as follows:

I need lots of storage on training instances.

As discussed in the previous example, most SageMaker training instances use EBS volumes, and you can set their size in the estimator. The maximum size of an EBS volume is 16 TB, so you should have more than enough. If your algorithm needs lots of temporary storage for intermediate results, this is the way to go.

My dataset is very large, and it takes a long time to copy it to training instances.

Define "long"! If you're looking for a quick fix, you can use instance types with high network performance. For example, m1.g4dn and m1.p3dn instances support the **Elastic Fabric Adapter** (<https://aws.amazon.com/hpc/efa>), and can go all the way to 100 Gbit/s.

If that's not enough, and if you're training on a single instance, you should use pipe mode, which streams data from S3 instead of copying it.

If training is distributed, you can switch the **distribution policy** from `FullyReplicated` to `ShardedbyS3Key`, which will only distribute a fraction of the dataset to each instance. This can be combined with pipe mode for extra performance.

My dataset is very large, and it doesn't fit in RAM.

If you want to stick to a single instance, a quick way to solve the problem is to scale up. The `m1.r5d.24xlarge` and `m1.p3dn.24xlarge` instances have 768 GB of RAM! If distributed training is an option, then you should configure it and apply data parallelism.

CPU utilization is low.

Assuming you haven't overprovisioned, the most likely cause is I/O latency (network or storage). The CPU is stalled because it's waiting for data to be fetched from wherever it's stored.

The first thing you should review is the data format. As discussed in previous chapters, there's no escaping **RecordIO** or **TFRecord** files. If you're using other formats (CSV, individual images, and so on), you should start there before tweaking the infrastructure.

If data is copied from S3 to an EBS volume, you can try using an instance with more EBS bandwidth. Numbers are available at the following location:

<https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-optimized.html>

You can also switch to an instance type with local NVMe storage (`g4dn` and `p3dn`). If the problem persists, you should review the code that reads data and passes it to the training algorithm. It probably needs more parallelism.

If data is streamed from S3 with pipe mode, it's unlikely that you've hit the maximum transfer speed of 25 GB/s, but it's worth checking the instance metric in CloudWatch. If you're sure that nothing else could be the cause, you should move to other file storage services, such as **Amazon EFS** and **Amazon FSx for Lustre**.

GPU memory utilization is low.

The GPU doesn't receive enough data from the CPU. You need to increase batch size until memory utilization is close to 100%. If you increase it too much, you'll get an angry out of memory error message, such as this one:

```
/opt/brazil-pkg-cache/packages/MXNetECL/MXNetECL-v1.4.1.1457.0/
AL2012/generic-flavor/src/src/storage/.pooled_storage_
manager.h:151: cudaMalloc failed: out of memory
```

When working with a multi-GPU instance in a data-parallel configuration, you should multiply the batch size passed to the estimator by the number of GPUs present in an instance.

When increasing batch size, you have to factor in the number of training samples available. For example, the **Pascal** VOC dataset that we used for Semantic Segmentation in *Chapter 5, Training Computer Vision Models*, only has 1,464 samples, so it would probably not make sense to increase batch size above 64 or 128.

Finally, batch size has an important effect on job convergence. Very large batches may slow it down, so you may want to increase the learning rate accordingly.

Sometimes, you'll simply have to accept that GPU memory utilization is low!

GPU utilization is low.

Maybe your model is simply not large enough to keep the GPU really busy. You should try scaling down on a smaller GPU.

If you're working with a large model, the GPU is probably stalled because the CPU can't feed it fast enough. If you're in control of the data loading code, you should try to add more parallelism, such as additional threads for data loading and preprocessing. If you're not, you should try a larger instance type with more vCPUs. Hopefully, they can be put to good use by the data-loading code.

If there's enough parallelism in the data loading code, then slow I/O is likely to be responsible. You should look for a faster alternative (NVMe, EFS, or FSx for Lustre).

GPU utilization is high.

That's a good place to be! You're efficiently using the infrastructure that you're paying for. As discussed in the previous example, you can try scaling up (more vCPUs or more GPUs), or scaling out (more instances). Combining both can work for highly parallel workloads such as deep learning.

Now we know a little more about scaling jobs, let's learn about more SageMaker features, starting with pipe mode.

Streaming datasets with pipe mode

The default setting of estimators is to copy the dataset to training instances, which is known as **file mode**. Instead, **pipe mode** streams it directly from S3. The name of the feature comes from its use of **Unix named pipes** (also known as **FIFOs**): at the beginning of each epoch, one pipe is created per input channel.

Pipe mode removes the need to copy any data to training instances. Obviously, training jobs start quicker. They generally run faster too, as pipe mode is highly optimized. Another benefit is that you won't have to provision any storage for the dataset on training instances.

Cutting on training time and storage means that you'll save money. The larger the dataset, the more you'll save. You can find benchmarks at the following link:

<https://aws.amazon.com/blogs/machine-learning/accelerate-model-training-using-faster-pipe-mode-on-amazon-sagemaker/>

In practice, you can start experimenting with pipe mode for datasets in the hundreds of megabytes and beyond. In fact, this feature enables you to work with infinitely large datasets. As storage and RAM requirements are no longer coupled to the size of the dataset, there's no practical limit on the amount of data that your algorithm can crunch. Training on petabyte-scale datasets becomes possible.

Using pipe mode with built-in algorithms

The prime candidates for pipe mode are built-in algorithms, as most of them support it natively:

- **Linear Learner, k-Means, k-Nearest Neighbors, Principal Component Analysis, Random Cut Forest, and Neural Topic Modeling:** RecordIO-wrapped protobuf or CSV data
- **Factorization Machines, Latent Dirichlet Allocation:** RecordIO-wrapped protobuf data
- **BlazingText** (supervised mode): Augmented manifest
- **Image Classification or Object Detection:** RecordIO-wrapped protobuf data or augmented manifest
- **Semantic segmentation:** Augmented manifest.

You should already be familiar with **RecordIO-wrapped protobuf**. If not, please revisit *Chapters 4 and 5*, where we covered it in detail. With RecordIO, you can easily split the input dataset into multiple files (100 MB seems to be a sweet spot). This makes it possible to work with an unlimited amount of data, regardless of maximum file size, and it can increase I/O performance. The `im2rec` tool has an option to generate multiple list files (`--chunks`). If you have existing list files, you can of course split them yourself.

We looked at the **augmented manifest** format when we discussed datasets annotated by **SageMaker Ground Truth** in *Chapter 5, Training Computer Vision Models*. For computer vision algorithms, this **JSON Lines** file contains the location of images in S3 and their labeling information. You can learn more at the following link:

<https://docs.aws.amazon.com/sagemaker/latest/dg/augmented-manifest.html>

Using pipe mode with other algorithms and frameworks

TensorFlow supports pipe mode thanks to the `PipeModeDataset` class implemented by AWS. Here are some useful resources:

- <https://github.com/aws/sagemaker-tensorflow-extensions>
- https://github.com/awslabs/amazon-sagemaker-examples/tree/master/sagemaker-python-sdk/tensorflow_script_mode_pipe_mode
- <https://medium.com/@julsimon/making-amazon-sagemaker-and-tensorflow-work-for-you-893365184233>

For other frameworks and for your own custom code, it's still possible to implement pipe mode inside the training container. A Python example is available at the following link:

https://github.com/awslabs/amazon-sagemaker-examples/tree/master/advanced_functionality/pipe Bring_your_own

Simplifying data loading with MLIO

MLIO (<https://github.com/awslabs/ml-io>) is an AWS open source project that lets you load data stored in memory, on local storage, or in S3 with pipe mode. The data can then be converted into different popular formats.

Here are the high-level features:

- **Input formats:** CSV, Parquet, RecordIO-protobuf, JPEG, PNG
- **Conversion formats:** NumPy arrays, SciPy matrices, Pandas DataFrames, TensorFlow tensors, PyTorch tensors, Apache MXNet arrays, and Apache Arrow
- API available in Python and C++

Now, let's run some examples with pipe mode.

Training factorization machines with pipe mode

We're going to revisit the example we used in *Chapter 4, Training Machine Learning Models*, where we trained a recommendation model on the **MovieLens** dataset. At the time, we used a small version of the dataset, limited to 100,000 reviews. This time, we'll go for the largest version:

1. We download and extract the dataset:

```
%%sh
wget http://files.grouplens.org/datasets/movielens/
ml-25m.zip
unzip ml-25m.zip
```

2. This dataset includes 25,000,095 reviews, from 162,541 users, on 62,423 movies. Unlike the 100k version, movies are not numbered sequentially. The last movie ID is 209,171, which needlessly increases the number of features. The alternative would be to renumber movies, but let's not do that here:

```
num_users=162541
num_movies=62423
num_ratings=25000095
max_movieid=209171
num_features=num_users+max_movieid
```

3. Just like in *Chapter 4, Training Machine Learning Models* we load the dataset into a sparse matrix (`lil_matrix` from SciPy), split it for training and testing, and convert both datasets into RecordIO-wrapped protobuf. Given the size of the dataset, this could take 45 minutes on a small Studio instance. Then, we upload the datasets to S3.
4. Next, we configure the two input channels, and we set their input mode to pipe mode instead of file mode:

```
From sagemaker import TrainingInput
s3_train_data = TrainingInput (
    train_data,
    content_type='application/x-recordio-protobuf',
    input_mode='Pipe')
s3_test_data = TrainingInput (
    test_data,
    content_type='application/x-recordio-protobuf',
    input_mode='Pipe')
```

5. We then configure the estimator, and train as usual on an `m1.c5.xlarge` instance (4 vCPUs, 8 GB RAM, \$0.23 per hour in `eu-west-1`).

Looking at the training log, we see the following:

```
2021-06-14 15:02:08 Downloading - Downloading input data
2021-06-14 15:02:08 Training - Downloading the training
image...
```

As expected, no time was spent copying the dataset. The same step in file mode takes 66 seconds. Even with a modest 1.5 GB dataset, pipe mode already makes sense. As datasets get bigger, this advantage will only increase!

Now, let's move on to distributed training.

Distributing training jobs

Distributed training lets you scale training jobs by running them on a cluster of CPU or GPU instances. It can be used to solve two different problems: very large datasets, and very large models.

Understanding data parallelism and model parallelism

Some datasets are too large to be trained in a reasonable amount of time on a single CPU or GPU. Using a technique called *data parallelism*, we can distribute data across the training cluster. The full model is still loaded on each CPU/GPU, which only receive an equal share of the dataset, not the full dataset. In theory, this should speed up training linearly according to the number of CPU/GPUs involved, and as you can guess, the reality is often different.

Believe it or not, some state-of-the-art-deep learning models are too large to fit on a single GPU. Using a technique called *model parallelism*, we can split it, and distribute the layers across a cluster of GPUs. Hence, training batches will flow across several GPUs to be processed by all layers.

Now, let's see where we can use distributed training in SageMaker.

Distributing training for built-in algorithms

Data parallelism is available for almost all built-in algorithms (semantic segmentation and LDA are notable exceptions). As they are implemented with Apache MXNet, they automatically use its native distributed training mechanism.

Distributing training for built-in frameworks

TensorFlow, PyTorch, Apache MXNet, and **Hugging Face** have native data parallelism mechanisms, and they're supported on SageMaker. **Horovod** (<https://github.com/horovod/horovod>) is available too.

For TensorFlow, PyTorch, and Hugging Face, you can also use the newer **SageMaker Distributed Data Parallel Library** and **SageMaker Model Parallel Library**. Both will be covered later in this chapter.

Distributed training often requires framework-specific changes to your training code. You can find more information in the framework documentation (for example https://www.tensorflow.org/guide/distributed_training), and in sample notebooks hosted at <https://github.com/awslabs/amazon-sagemaker-examples>:

- **TensorFlow:**
 - a) sagemaker-python-sdk/tensorflow_script_mode_horovod
 - b) advanced_functionality/distributed_tensorflow_mask_rcnn
- **Keras:** sagemaker-python-sdk/keras_script_mode_pipe_mode_horovod
- **PyTorch:** sagemaker-python-sdk/pytorch_horovod_mnist

Each framework has its peculiarities, yet everything we discussed in the previous sections stands true. If you want to make the most of your infrastructure, you need to pay attention to batch size, synchronization, and so on. Experiment, monitor, analyze, and iterate!

Distributing training for custom containers

If you're training with your own custom container, you have to implement your own distributed training mechanism. Let's face it, this is going to be a lot of work. SageMaker only helps to provide the name of cluster instances and the name of the container network interface. They are available inside the container in the `/opt/ml/input/config/resourceconfig.json` file.

You can find more information at the following link:

<https://docs.aws.amazon.com/sagemaker/latest/dg/your-algorithms-training-algo-running-container.html>

It's time for a distributed training example!

Scaling an image classification model on ImageNet

In *Chapter 5, Training Computer Vision Models*, we trained the image classification algorithm on a small dataset with dog and cat images (25,000 training images). This time, let's go for something a little bigger.

We're going to train a ResNet-50 network from scratch on the **ImageNet** dataset – the reference dataset for many computer vision applications (<http://www.image-net.org>). The 2012 version contains 1,281,167 training images (140 GB) and 50,000 validation images (6.4 GB) from 1,000 classes.

If you want to experiment at a smaller scale, you can work with 5-10% of the dataset. Final accuracy won't be as good, but it doesn't matter for our purposes.

Preparing the ImageNet dataset

This requires a lot of storage – the dataset is 150 GB, so please make sure you have at least 500 GB available to store it in ZIP and processed formats. You're also going to need a lot of bandwidth and a lot of patience to download it. I used an EC2 instance running **Amazon Linux 2** in the `us-east-1` region, and my download took *five days*.

1. Visit the ImageNet website, register to download the dataset, and accept the conditions. You'll get a username and an access key allowing you to download the dataset.
2. One of the TensorFlow repositories includes a great script that will download the dataset and extract it. Using `nohup` is essential so that the process continues running even if your session is terminated:

```
$ git clone https://github.com/tensorflow/models.git
$ export IMAGENET_USERNAME=YOUR_USERNAME
$ export IMAGENET_ACCESS_KEY=YOUR_ACCESS_KEY
$ cd models/research/inception/inception/data
$ mv imagenet_2012_validation_synset_labels.txt synsets.txt
$ nohup bash download_imagenet.sh . synsets.txt >&
download.log &
```

3. Once this is over (again, downloading will take days), the `imagenet/train` directory contains the training dataset (one folder per class). The `imagenet/validation` directory contains 50,000 images in the same folder. We can use a simple script to organize it with one folder per class:

```
$ wget https://raw.githubusercontent.com/juliensimon/aws/master/mxnet/imagenet/build_validation_tree.sh  
$ chmod 755 build_validation_tree.sh  
$ cd imagenet/validation  
$ ../../build_validation_tree.sh  
$ cd ../../..
```

4. We're going to build RecordIO files with the `im2rec` tool present in the Apache MXNet repository. Let's install dependencies, and fetch `im2rec`:

```
$ sudo yum -y install python-devel python-pip opencv  
opencv-devel opencv-python  
$ pip3 install mxnet opencv-python -user  
$ wget https://raw.githubusercontent.com/apache/  
incubator-mxnet/master/tools/im2rec.py
```

5. In the `imagenet` directory, we run `im2rec` twice – once to build the list files, and once to build the RecordIO files. We create RecordIO files that are approximately 1 GB each (we'll see why that matters in a second). We also resize the smaller dimension of images to 224 so that the algorithm won't have to do it:

```
$ cd imagenet  
$ python3 ./im2rec.py --list --chunks 6 --recursive val  
validation  
$ python3 ./im2rec.py --num-thread 16 --resize 224 val_  
validation  
$ python3 ./im2rec.py --list --chunks 140 --recursive  
train train  
$ python3 ./im2rec.py --num-thread 16 --resize 224  
train_train
```

6. Finally, we sync the dataset to S3:

```
$ mkdir -p input/train input/validation  
$ mv train_*.rec input/train  
$ mv val_*.rec input/validation
```

```
$ aws s3 sync input s3://sagemaker-us-east-1-123456789012/imagenet-split/input/
```

The dataset is now ready for training.

Defining our training job

Now that the dataset is ready, we need to think about the configuration of our training job. Specifically, we need to come up with the following:

- An input configuration, defining the location and the properties of the dataset
- Infrastructure requirements to run the training job
- Hyperparameters to configure the algorithm

Let's look at each one of these items in detail.

Defining the input configuration

Given the size of the dataset, pipe mode sounds like a great idea. Out of curiosity, I tried training in file mode. Even with a 100 Gbit/s network interface, it took almost 25 minutes to copy the dataset from S3 to local storage. Pipe mode it is!

You may wonder why we took care of splitting the dataset into multiple files. Here's why:

- In general, multiple files create opportunities for more parallelism, making it easier to write fast data loading and processing code.
- We can shuffle the files at the beginning of each epoch, removing any potential bias caused by the order of samples.
- It makes it very easy to work with a fraction of the dataset.

Now that we've defined the input configuration, what about infrastructure requirements?

Defining infrastructure requirements

ImageNet is a large and complex dataset that requires a lot of training to reach good accuracy.

A quick test shows that a single m1 . p3 . 2xlarge instance with the batch size set to 128 will crunch through the dataset at about 335 images per second. As we have about 1,281,167 images, we can expect one epoch to last about 3,824 seconds (about 1 hour and 4 minutes).

Assuming that we need to train for 150 epochs to get decent accuracy, we're looking at a job that should last $(3,824/3,600)*150 = 158$ hours (about 6.5 days). This is probably not acceptable from a business perspective. For the record, at \$3.825 per instance per hour in `us-east-1`, that job would cost about \$573.

Let's try to speed up our job with `m1.p3dn.24xlarge` instances. Each one hosts eight NVIDIA V100s with 32 GB of GPU memory (twice the amount available on other p3 instances). They also have 96 **Intel Skylake** cores, 768 GB of RAM, and 1.8 TB of local NVMe storage. Although we're not going to use it here, the latter is a fantastic storage option for long-running, large-scale jobs. Last but not least, this instance type has 100 Gbit/s networking, a great feature for streaming data from S3 and for inter-instance communication.

Note

At \$35.894 per hour per instance in `us-east-1`, you may not want to try this at home or even at work without getting permission. Your service quotas probably don't let you run that much infrastructure anyway, and you would have to get in touch with AWS Support first.

In the next chapter, we're going to talk about *managed spot training* – a great way to slash training costs. We'll revisit the ImageNet example once we've covered this topic, so you definitely should refrain from training right now!

Training on ImageNet

Let's configure the training job:

1. We configure pipe mode on both input channels. The files of the training channel are shuffled for extra randomness:

```
prefix = 'imagenet-split'  
s3_train_path =  
's3://{{}}/{{}}/input/training/'.format(bucket, prefix)  
s3_val_path =  
's3://{{}}/{{}}/input/validation/'.format(bucket, prefix)  
s3_output =  
's3://{{}}/{{}}/output/'.format(bucket, prefix)  
from sagemaker import TrainingInput  
from sagemaker.session import ShuffleConfig  
train_data = TrainingInput(  
    s3_train_path
```

```
shuffle_config=ShuffleConfig(59),
content_type='application/x-recordio',
input_mode='Pipe')
validation_data = TrainingInput(
    s3_val_path,
    content_type='application/x-recordio',
    input_mode='Pipe')
s3_channels = {'train': train_data,
                'validation': validation_data}
```

2. To begin with, we configure the Estimator module with a single ml.p3dn.24xlarge instance:

```
from sagemaker import image_uris
region_name = boto3.Session().region_name
container = image_uris.retrieve(
    'image-classification', region)
ic = sagemaker.estimator.Estimator(
    container,
    role=sagemaker.get_execution_role(),
    instance_count=1,
    instance_type='ml.p3dn.24xlarge',
    output_path=s3_output)
```

3. We set hyperparameters, starting with a reasonable batch size of 1,024, and we launch training:

```
ic.set_hyperparameters(
    num_layers=50,
    use_pretrained_model=0,
    num_classes=1000,
    num_training_samples=1281167,
    mini_batch_size=1024,
    epochs=2,
    kv_store='dist_sync',
    top_k=3)
```

Updating batch size

Time per epochs is 727 seconds. For 150 epochs, this translates into 30.3 hours of training (1.25 days), and a cost of \$1,087. The good news is that we're going 5x faster. The bad news is that cost has gone up 2x. Let's start scaling this.

Looking at total GPU utilization in CloudWatch, we see that it doesn't exceed 300%. That is, 37.5% on each GPU. This probably means that our batch size is too low to keep the GPUs fully busy. Let's bump it to $(1,024/0.375)=2730$, rounded up to 2,736 to be divisible by 8:

Note

Depending on algorithm versions, **CUDA** versions, the number of instances involved, and so on, your mileage may vary. Reduce batch size a bit if you get out of memory errors.

```
ic.set_hyperparameters(  
    num_layers=50,  
    use_pretrained_model=0,  
    num_classes=1000,  
    num_training_samples=1281167,  
    mini_batch_size=2736,          # <-----  
    epochs=2,  
    kv_store='dist_sync',  
    top_k=3)
```

Training again, an epoch now lasts 758 seconds. It looks like maxing out GPU memory usage didn't make a big difference this time. Maybe it's offset by the cost of synchronizing gradients? Anyway, keeping GPU cores as busy as possible is good practice.

Adding more instances

Now, let's add a second instance to scale out the training job:

```
ic = sagemaker.estimator.Estimator(  
    container,  
    role,  
    instance_count=2,           # <-----  
    instance_type='ml.p3dn.24xlarge',  
    output_path=s3_output)
```

Time for epoch is now 378 seconds! For 150 epochs, this translates to 15.75 hours of training, and a cost of \$1,221. Compared to our initial job, this is 2x faster and 3x cheaper!

How about four instances? Let's see if we can keep scaling:

```
ic = sagemaker.estimator.Estimator(  
    container,  
    role,  
    instance_count=4,           # <-----  
    instance_type='ml.p3dn.24xlarge',  
    output_path=s3_output)
```

Time for epoch is now 198 seconds! For 150 epochs, this translates to 8.25 hours of training, and a cost of \$1,279. We sped up 2x again, with a marginal cost increase.

Now, shall we train eight instances? Of course! Who wouldn't want to train on 64 GPUs, 327K CUDA cores, and 2 TB (!) of GPU RAM:

```
ic = sagemaker.estimator.Estimator(  
    container,  
    role,  
    instance_count=8,           # <-----  
    instance_type='ml.p3dn.24xlarge',  
    output_path=s3_output)
```

Time for epoch is now 99 seconds. For 150 epochs, this translates into 4.12 hours of training, and a cost of \$1,277. We sped up 2x *again*, at no cost increase.

Summing things up

For 2x the initial cost, we've accelerated our training job 38x, thanks to pipe mode, distributed training, and state-of-the-art GPU instances.

Instance type	Instance count	Time per epoch (seconds)	Estimated time for 150 epochs (hours)	Total cost (us-east-1)
ml.p3.2xlarge	1	3824	158	\$573
ml.p3dn.24xlarge	1	727	30.3	\$1,087
ml.p3dn.24xlarge	2	378	15.75	\$1,221
ml.p3dn.24xlarge	4	198	8.25	\$1,279
ml.p3dn.24xlarge	8	99	4.12	\$1,277

Fig 9.8 Outcome of the training jobs

Not bad at all! Saving days on your training jobs helps you iterate faster, get to a high-quality model quicker, and get to production sooner. I'm pretty sure this would easily offset the extra cost. Still, in the next chapter, we'll see how we can slash training costs massively with managed spot training.

Now that we're familiar with distributed training, let's take a look at two new SageMaker libraries for data parallelism and model parallelism.

Training with the SageMaker data and model parallel libraries

These two libraries were introduced in late 2020, and significantly improve the performance of large-scale training jobs.

The **SageMaker Distributed Data Parallel (DDP)** library implements a very efficient distribution of computation on GPU clusters. It optimizes network communication by eliminating inter-GPU communication, maximizing the amount of time and resources they spend on training. You can learn more at the following link:

<https://aws.amazon.com/blogs/aws/managed-data-parallelism-in-amazon-sagemaker-simplifies-training-on-large-datasets/>

DDP is available for TensorFlow, PyTorch, and Hugging Face. The first two require minor modifications to the training code, but the last one doesn't. As DDP only makes sense for large, long-running training jobs, available instance sizes are `m1.p3.16xlarge`, `m1.p3dn24dnxlarge`, and `m1.p4d.24xlarge`.

The **SageMaker Distributed Model Parallel (DMP)** library solves a different problem. Some large deep learning models are simply too bulky to fit inside the memory of a single GPU. Others barely fit, forcing you to work with very small batch sizes, and slowing down your training jobs. DMP solves this problem by automatically partitioning models across a cluster of GPUs and orchestrating the flow of data through these different partitions. You can learn more at the following link:

<https://aws.amazon.com/blogs/aws/amazon-sagemaker-simplifies-training-deep-learning-models-with-billions-of-parameters/>

DMP is available for TensorFlow, PyTorch, and Hugging Face. Again, the first two require small modifications to the training code, and the last one doesn't, as the Hugging Face Trainer API fully supports DMP.

Let's give both a try by revisiting our TensorFlow and Hugging Face examples from *Chapter 7, Extending Machine Learning Services Using Built-In Frameworks*.

Training on TensorFlow with SageMaker DDP

Our initial code used the high-level Keras API: `compile()`, `fit()`, and so on. In order to implement DDP, we need to rewrite this code to use `tf.GradientTape()`, and to implement a custom training loop. It's not as difficult as it sounds, so let's get to work:

1. First, we need to import and initialize DDP:

```
import smdistributed.dataparallel.tensorflow as sdp  
sdp.init()
```

2. Then, we retrieve the list of GPUs present on an instance, and we assign them a local DDP rank, which is just an integer identifier. We also allow memory growth, a TensorFlow feature required by DDP:

```
gpus = tf.config.experimental.  
       list_physical_devices('GPU')  
if gpus:  
    tf.config.experimental.set_visible_devices(  
        gpus[sdp.local_rank()], 'GPU')
```

```
for gpu in gpus:  
    tf.config.experimental.set_memory_growth(  
        gpu, True)
```

3. As recommended by the documentation, we increase the batch size and the learning rate according to the number of GPUs present in the training cluster. This is very important for job accuracy:

```
batch_size = args.batch_size*sdp.size()  
lr          = args.learning_rate*sdp.size()
```

4. We then create a loss function and an optimizer. Labels have been one-hot encoded during preprocessing, so we use `CategoricalCrossentropy`, not `SparseCategoricalCrossentropy`. We also initialize model and optimizer variables on all GPUs:

```
loss = tf.losses.CategoricalCrossentropy()  
opt = tf.optimizers.Adam(lr)  
sdp.broadcast_variables(model.variables, root_rank=0)  
sdp.broadcast_variables(opt.variables(), root_rank=0)
```

5. Next, we need to write a `training_step()` function, and decorate it with `@tf.function` so that DDP recognizes it. As its name implies, this function is responsible for running a training step on each GPU in the training cluster: predict a batch, compute loss, compute gradients, and apply them. It's based on the `tf.GradientTape()` API, which we simply wrap with `sdp.DistributedGradientTape()`. At the end of each training step, we use `sdp.oob_allreduce()` to compute the average loss, using values coming from all GPUs:

```
@tf.function  
def training_step(images, labels):  
    with tf.GradientTape() as tape:  
        probs = model(images, training=True)  
        loss_value = loss(labels, probs)  
    tape = sdp.DistributedGradientTape(tape)  
    grads = tape.gradient(  
        loss_value, model.trainable_variables)  
    opt.apply_gradients(
```

```
    zip(grads, model.trainable_variables))
loss_value = sdp.oob_allreduce(loss_value)
return loss_value
```

6. Next, we write the training loop. There's nothing particular about it. To avoid log pollution, we only print out messages from the master GPU (rank 0):

```
steps = len(train)//batch_size

for e in range(epochs):
    if sdp.rank() == 0:
        print("Start epoch %d" % (e))
    for batch, (images, labels) in enumerate(train.take(steps)):
        loss_value = training_step(images, labels)
        if batch%10 == 0 and sdp.rank() == 0:
            print("Step #{} \tLoss: {:.6f}"
                  .format(batch, loss_value))
```

7. Finally, we save the model on GPU #0 only:

```
if sdp.rank() == 0:  
    model.save(os.path.join(model_dir, '1'))
```

8. Moving to our notebook, we configure this job with two `ml.p3.16xlarge` instances, and we enable data parallelism with an additional parameter in the estimator:

```
from sagemaker.tensorflow import TensorFlow
tf_estimator = TensorFlow(
    . . .
    instance_count=2,
    instance_type='ml.p3.16xlarge',
    hyperparameters={'epochs': 10,
                     'learning-rate': 0.0001, 'batch-size': 32},
    distribution={'smdistributed':
        {'dataparallel': {'enabled': True}}})
}
```

9. We train as usual, and we see steps going by in the training log:

```
[1,0]<stdout>:Step #0#011Loss: 2.306620
[1,0]<stdout>:Step #10#011Loss: 1.185689
[1,0]<stdout>:Step #20#011Loss: 0.909270
[1,0]<stdout>:Step #30#011Loss: 0.839223
[1,0]<stdout>:Step #40#011Loss: 0.772756
[1,0]<stdout>:Step #50#011Loss: 0.678521
. . .
```

As you can see, it's not really difficult to scale training jobs with SageMaker DDP, especially if your training code already uses low-level APIs. We used TensorFlow here, and the process for PyTorch is very similar.

Now, let's see how we can train large Hugging Face models with both libraries. Indeed, state-of-the-art NLP models are getting larger and more complex all the time, and they're good candidates for data parallelism and model parallelism.

Training on Hugging Face with SageMaker DDP

As the Hugging Face Trainer API fully supports DDP, we don't need to change anything in our training script. Woohoo. All it takes is an extra parameter in the estimator. Set the instance type and instance count, and you're good to go:

```
huggingface_estimator = HuggingFace(
    . . .
    distribution={'smdistributed':
        {'dataparallel': {'enabled': True}}}
    )
)
```

Training on Hugging Face with SageMaker DMP

Adding DMP is not difficult either. Our Hugging Face example uses a **DistilBERT** model that is about 250 MB. That's small enough to fit on a single GPU, but let's try to train with DMP anyway:

1. First, we need to configure **MPI** (<https://www.open-mpi.org>) settings, as it's used for GPU communication. You should set `processes_per_host` to a value lower or equal to the number of GPUs on a training instance. Here, I'll use an `ml.p3dn.24xlarge` instance with 8 NVIDIA V100 GPUs:

```
mpi_options = {
    'enabled' : True,
    'processes_per_host' : 8
}
```

2. Then, we configure DMP options. Here, I set the most important ones – the number of model partitions that we want (`partitions`), and how many times they should be replicated for increased parallelism (`microbatches`). In other words, our model will be split in four, each split will be duplicated, and these eight splits will each run on a different GPU. You can find more information on all parameters at the following link:

https://sagemaker.readthedocs.io/en/stable/api/training/smd_model_parallel_general.html

```
smp_options = {
    'enabled': True,
    'parameters': {
        'microbatches': 2,
        'partitions': 4
    }
}
```

3. Finally, we configure our estimator and train as usual:

```
huggingface_estimator = HuggingFace(
    ...
    instance_type='ml.p3dn.24xlarge',
    instance_count=1,
    distribution={'smdistributed':
```

```
{ 'modelparallel': smp_options,
  'mpi': mpi_options}
}
```

You can find additional examples here:

- TensorFlow and PyTorch
- https://github.com/aws/amazon-sagemaker-examples/tree/master/training/distributed_training
- Hugging Face: <https://github.com/huggingface/notebooks/tree/master/sagemaker>

To close this chapter, let's now look at storage options you should consider for very large-scale, high-performance training jobs.

Using other storage services

So far, we've used S3 to store training data. At a large scale, throughput and latency can become a bottleneck, making it necessary to consider other storage services:

- **Amazon Elastic File System (EFS):** <https://aws.amazon.com/efs>
- **Amazon FSx for Lustre:** <https://aws.amazon.com/fsx/lustre>.

Note

This section requires a little bit of AWS knowledge on VPCs, subnets, and security groups. If you're not familiar at all with these, I'd recommend reading the following:

https://docs.aws.amazon.com/vpc/latest/userguide/VPC_Subnets.html

https://docs.aws.amazon.com/vpc/latest/userguide/VPC_SecurityGroups.html

Working with SageMaker and Amazon EFS

EFS is a managed storage service compatible with **NFS v4**. It lets you create volumes that can be attached to EC2 instances and SageMaker instances. This is a convenient way to share data, and you can use it to scale I/O for large training jobs.

By default, files are stored in the **Standard** class. You can enable a life cycle policy that automatically moves files that haven't been accessed for a certain time to the **Infrequent Access**, which is slower but more cost-effective.

You can pick one of two throughput modes:

- **Bursting throughput:** Burst credits are accumulated over time, and burst capacity depends on the size of the filesystem: 100 MB/s, plus an extra 100 MB/s for each TB of storage.
- **Provisioned throughput:** You set the expected throughput, from 1 to 1,024 MB/s.

You can also pick one of two performance modes:

- **General purpose:** This is fine for most applications.
- **Max I/O:** This is the one to use if tens or hundreds of instances are accessing the volume. Throughput will be maximized at the expense of latency.

Let's create an 8 GB EFS volume. Then, we'll mount it on an EC2 instance to copy the **Pascal VOC** dataset that we previously prepared, and we'll train an object detection job. To keep costs reasonable, we won't scale the job, but the overall process would be exactly the same at any scale.

Provisioning an EFS volume

The EFS console makes it extremely simple to create a volume. You can find detailed instructions at <https://docs.aws.amazon.com/efs/latest/ug/getting-started.html>:

1. We set the volume name to `sagemaker-demo`.
2. We select our default VPC, and use **Regional** availability.

3. We create the volume. Once it's ready, you should see something similar to the following screenshot:

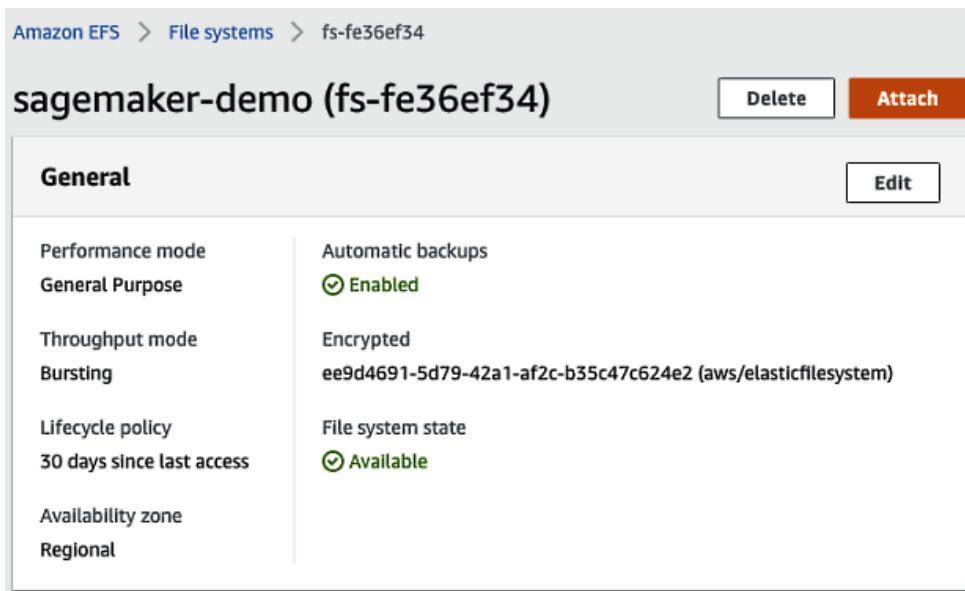


Figure 9.9– Creating an EFS volume

The EFS volume is ready to receive data. We're now going to create a new EC2 instance, mount the EFS volume, and copy the dataset.

Creating an EC2 instance

As EFS volumes live inside a VPC, they can only be accessed by instances located in the same VPC. These instances must also have a *security group* that allows inbound NFS traffic:

1. In the VPC console (<https://console.aws.amazon.com/vpc/#vpcs:sort=VpcId>), we write down the ID of our default VPC. For me, it's `vpc-def884bb`.
2. Still in the VPC console, we move to the **Subnets** section (<https://console.aws.amazon.com/vpc/#subnets:sort=SubnetId>). We write down the subnet IDs and the availability zone for all subnets hosted in the default VPC.

For me, they look like what's shown in the next screenshot:

	Name	Subnet ID	State	VPC	Availability Zone
<input type="checkbox"/>	subnet-63715206	available	vpc-def884bb DEFAULT ...	eu-west-1a	
<input type="checkbox"/>	subnet-cbf5bdb	available	vpc-def884bb DEFAULT ...	eu-west-1b	
<input type="checkbox"/>	subnet-59395b00	available	vpc-def884bb DEFAULT ...	eu-west-1c	

Figure 9.10 – Viewing subnets for the default VPC

3. Moving to the EC2 console, we create an EC2 instance. We select the Amazon Linux 2 image and a t2.micro instance size.
4. Next, we set **Network** to the default VPC, and **Subnet** to the subnet hosted in the eu-west-1a **Availability Zone**. We also assign it the security group we just created, **IAM role** to a role with appropriate S3 permissions, and **File Systems** to the EFS filesystem that we just created. We also make sure to tick the box that automatically creates and attaches the required security groups.
5. In the next screens, we leave storage and tags as they are, and we attach a security group that allows incoming ssh. Finally, we launch instance creation.

Accessing an EFS volume

Once the instance is ready, we can ssh to it:

1. We see that the EFS volume has been automatically mounted:

```
[ec2-user]$ mount | grep efs
127.0.0.1:/ on /mnt/efs/fs1 type nfs4
```

2. We move to that location, and sync our PascalVOC dataset from S3. As the filesystem is mounted as root, we need to use sudo.

```
[ec2-user] cd /mnt/efs/fs1
[ec2-user] sudo aws s3 sync s3://sagemaker-ap-northeast-2-123456789012/pascalvoc/input input
```

Job done. We can log out and shut down or terminate the instance, as we won't need it anymore.

Now, let's train with this dataset.

Training an object detection model with EFS

The training process is identical, except for the location of the input data:

- Instead of using the `TrainingInput` object to define input channels, we use the `FileSystemInput` object, passing the identifier of our EFS volume and the absolute data path inside the volume:

```
from sagemaker.inputs import FileSystemInput
efs_train_data = FileSystemInput(
    file_system_id='fs-fe36ef34',
    file_system_type='EFS',
    directory_path='/input/train')
efs_validation_data = FileSystemInput(
    file_system_id='fs-fe36ef34',
    file_system_type='EFS',
    directory_path='/input/validation')
data_channels = {'train': efs_train_data,
                 'validation': efs_validation_data}
```

- We configure the `Estimator` module, passing the list of subnets for the VPC hosting the EFS volume. SageMaker will launch training instances there so that they may mount the EFS volume. We also need to pass a security group allowing NFS traffic. We can reuse the one that was automatically created for our EC2 instance (not the one allowing ssh access) – it's visible in the **Security** tab in the instance details, as shown in the next screenshot:

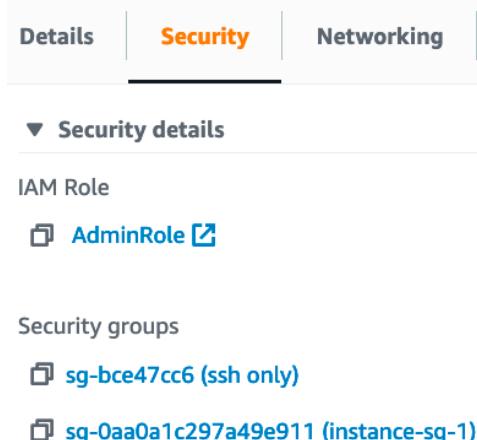


Figure 9.11 – Viewing security groups

The subnet and security group IDs are passed to the Estimator module like so:

```
from sagemaker import image_uris
container = image_uris.retrieve('object-detection',
                                region)
od = sagemaker.estimator.Estimator(
    container,
    role=sagemaker.get_execution_role(),
    instance_count=1,
    instance_type='ml.p3.2xlarge',
    output_path=s3_output_location,
    subnets=['subnet-63715206', 'subnet-cbf5bdbc',
             'subnet-59395b00'],
    security_group_ids=['sg-0aa0a1c297a49e911']
)
```

3. For testing purposes, we only train for one epoch. Business as usual, although, this time, data is loaded from our EFS volume.

Once training is complete, you may delete the EFS volume in the EFS console to avoid unnecessary costs.

Now, let's see how we can use another storage service – Amazon FSx for Lustre.

Working with SageMaker and Amazon FSx for Lustre

Very large-scale workloads require high throughput and low latency storage – two qualities that Amazon FSx for Lustre possesses. As the name implies, this service is based on the Lustre filesystem (<http://lustre.org>), a popular open source choice for **HPC** applications.

The smallest filesystem you can create is 1.2 TB (like I said, "very large-scale"). We can pick one of two deployment options for FSx filesystems:

- **Persistent:** This should be used for long-term storage that requires high availability.
- **Scratch:** Data is not replicated, and it won't persist if a file server fails. In exchange, we get high burst throughput, making this a good choice for spiky, short-term jobs.

Optionally, a filesystem can be backed by an S3 bucket. Objects are automatically copied from S3 to FSx when they're first accessed.

Just like for EFS, a filesystem lives inside a VPC, and we'll need a security group allowing inbound Lustre traffic (ports 988 and 1,021-2,023). You can create this in the EC2 console, and it should be similar to the following screenshot:

Inbound rules	Outbound rules	Tags	
Inbound rules (3)			
Type	Protocol	Port range	Source
Custom TCP	TCP	988	0.0.0.0/0
Custom TCP	TCP	988	::/0
Custom TCP	TCP	1021 - 1023	0.0.0.0/0

Figure 9.12 – Creating a security group for FSx for Lustre

Let's create the filesystem:

1. In the FSx console, we create a filesystem named `sagemaker-demo`, and we select the **Scratch** deployment type.
2. We set storage capacity to 1.2 TB.
3. In the **Network & security** section, we choose to host in the `eu-west-1a` subnet of the default VPC, and we assign it to the security group we just created.
4. In the **Data repository integration** section, we set the import bucket (`s3://sagemaker-eu-west-1-123456789012`) and the prefix (`pascalvoc`).
5. On the next screen, we review our choices, as shown in the following screenshot, and we create the filesystem.

After a few minutes, the filesystem is in service, as shown in the following screenshot:

File systems (1)						
File system name	File system ID	File system type	Status	Deployment type	Storage type	Storage capacity
<code>sagemaker-demo</code>	<code>fs-043055d89920277e4</code>	Lustre	Available	Scratch 2	SSD	1,200 GiB

Figure 9.13 – Creating an FSx volume

As the filesystem is backed by an S3 bucket, we don't need to populate it. We can proceed directly to training.

Training an object detection model with FSx for Lustre

Now, we will train the model using FSx as follows:

1. Similar to what we just did with EFS, we define input channels with `FileSystemInput`. One difference is that the directory path must start with the name of the filesystem mount point. You can find it as **Mount name** in the FSx console:

```
from sagemaker.inputs import FileSystemInput
fsx_train_data = FileSystemInput(
    file_system_id='fs-07914cf5a60649dc8',
    file_system_type='FSxLustre',
    directory_path='/bmgbtbmv/pascalvoc/input/train')
fsx_validation_data = FileSystemInput(
    file_system_id='fs-07914cf5a60649dc8',
    file_system_type='FSxLustre',
    directory_path='/bmgbtbmv/pascalvoc/input/validation')
data_channels = {'train': fsx_train_data,
                 'validation': fsx_validation_data }
```

2. All other steps are identical. Don't forget to update the name of the security group passed to the `Estimator` module.
3. When we're done training, we delete the FSx filesystem in the console.

This concludes our exploration of storage options for SageMaker. Summing things up, here are my recommendations:

- First, you should use RecordIO or TFRecord data as much as possible. They're convenient to move around, faster to train on, and they work with both file mode and pipe mode.
- For development and small-scale production, file mode is completely fine. Your primary focus should always be your machine learning problem, not useless optimization. Even at a small scale, EFS can be an interesting option for collaboration, as it makes it easy to share datasets and notebooks.

- If you train with built-in algorithms, pipe mode is a no-brainer, and you should use it at every opportunity. If you train with frameworks or your own code, implementing pipe mode will take some work, and is probably not worth the engineering effort unless you're working at a significant scale (hundreds of gigabytes or more).
- If you have large, distributed workloads with tens of instances or more, EFS in Performance Mode is worth trying. Don't go near the mind-blowing FSx for Lustre unless you have insane workloads.

Summary

In this chapter, you learned how and when to scale training jobs. You saw that it definitely takes some careful analysis and experimentation to find the best setup: scaling up versus scaling out, CPU versus GPU versus multi-GPU, and so on. This should help you to make the right decisions for your own workloads and avoid costly mistakes.

You also learned how to achieve significant speedup with techniques such as distributed training, data parallelism, model parallelism, RecordIO, and pipe mode. Finally, you learned how to set Amazon EFS and Amazon FSx for Lustre for large-scale training jobs.

In the next chapter, we'll cover advanced features for hyperparameter optimization, cost optimization, model debugging, and more.

10

Advanced Training Techniques

In the previous chapter, you learned when and how to scale training jobs using features such as **Pipe mode** and **distributed training**, as well as alternatives to **S3** for dataset storage.

In this chapter, we'll conclude our exploration of training techniques. In the first part of the chapter, you'll learn how to slash down your training costs with **managed spot training**, how to squeeze every drop of accuracy from your models with **automatic model tuning**, and how to crack models open with **SageMaker Debugger**.

In the second part of the chapter, we'll introduce two new SageMaker capabilities that help you build more efficient workflows and higher quality models: **SageMaker Feature Store** and **SageMaker Clarify**.

This chapter covers the following topics:

- Optimizing training costs with managed spot training
- Optimizing hyperparameters with automatic model tuning
- Exploring models with SageMaker Debugger
- Managing features and building datasets with SageMaker Feature Store
- Detecting bias and explaining predictions with SageMaker Clarify

Technical requirements

You will need an AWS account to run the examples included in this chapter. If you haven't got one already, please point your browser at <https://aws.amazon.com/getting-started/> to create it. You should also familiarize yourself with the AWS free tier (<https://aws.amazon.com/free/>), which lets you use many AWS services for free within certain usage limits.

You will need to install and configure the AWS **Command-Line Interface (CLI)** for your account (<https://aws.amazon.com/cli/>).

You will need a working **Python 3.x** environment. Installing the **Anaconda** distribution (<https://www.anaconda.com/>) is not mandatory but strongly encouraged as it includes many projects that we will need (**Jupyter**, **pandas**, **numpy**, and more).

Code examples included in this book are available on GitHub at <https://github.com/PacktPublishing/Learn-Amazon-SageMaker-second-edition>. You will need to install a Git client to access them (<https://git-scm.com/>).

Optimizing training costs with managed spot training

In the previous chapter, we trained the **image classification** algorithm on the **ImageNet** dataset. The job ran for a little less than 4 hours. At about \$290 per hour, this job cost us roughly \$1,160. That's a lot of money... but is it really?

Comparing costs

Before you throw your arms up the air yelling "*What is he thinking?*", please consider how much it would cost your organization to own and run this training cluster:

1. A back-of-the-envelope calculation for capital expenditure (servers, storage, GPUs, 100 Gbit/s networking equipment) says at least \$1.5M. As far as operational expenditure is concerned, hosting costs won't be cheap, as each equivalent server will require 4-5 kW of power. That's enough to fill one rack at your typical hosting company, so even if high-density racks are available, you'll need several. Add bandwidth, cross connects, and so on, and my gut feeling says it would cost about \$15K per month (much more in certain parts of the world).
2. We would need to add hardware support contracts (say, 10% per year, so \$150K). Depreciating this cluster over 5 years, total monthly costs would be $(\$1.5M + 60 * \$15K + 5 * \$150K) / 60 = \$52.5K$. Let's round it to \$55K to account for labor costs for server maintenance and so on.

Using conservative estimates, this spend is equivalent to 190 hours of training with the large \$290-an-hour cluster we've used for our ImageNet example. As we will see later in this chapter, managed spot training routinely delivers savings of 70%. So, now the spend would be equivalent to about 633 hours of ImageNet training per month.

This amounts to 87% usage ($633/720$) month in, month out, and it's very unlikely you'd keep your training cluster that busy. Add downtime, accelerated depreciation caused by hardware innovation, hardware insurance costs, the opportunity cost of not investing \$1.5M in other ventures, and so on, and the business case for physical infrastructure gets worse by the minute.

Financials matter, but the worst thing is that you'd only have one cluster. What if a potential business opportunity required another one? Would you spend another \$1.5M? If not, would you have to time-share the existing cluster? Of course, only you could decide what's best for your organization. Just make sure that you look at the big picture.

Now, let's see how you can easily enjoy that 70% cost reduction.

Understanding Amazon EC2 Spot Instances

At any given time, **Amazon EC2** has more capacity than needed. This allows customers to add on-demand capacity to their platforms whenever they need to. On-demand instances may be created explicitly using an API call, or automatically if **Auto Scaling** is configured. Once a customer has acquired an on-demand instance, they will keep it until they decide to release it, either explicitly or automatically.

Spot Instances are a simple way to tap into this unused capacity and to enjoy very significant discounts (50-70% are typical). You can request them in the same way, and they behave the same too. The only difference is that should AWS need the capacity to build on-demand instances, your Spot Instance may be reclaimed. It will receive an interruption notification two minutes before being forcefully terminated.

This isn't as bad as it sounds. Depending on regions and instance families, Spot Instances may not be reclaimed very often, and customers routinely keep them for days, if not more. In addition, you can architecture your application for this requirement, for example, by running stateless workloads on Spot Instances and relying on managed services for data storage. The cost benefit is too good to pass!

Going to the **Spot Requests** section in the EC2 console, you can view the price history per instance type in each region. For example, the following screenshot shows the spot price of p3dn.24xlarge for the last three months, where the spot price has been 60-70% cheaper than the on-demand price:

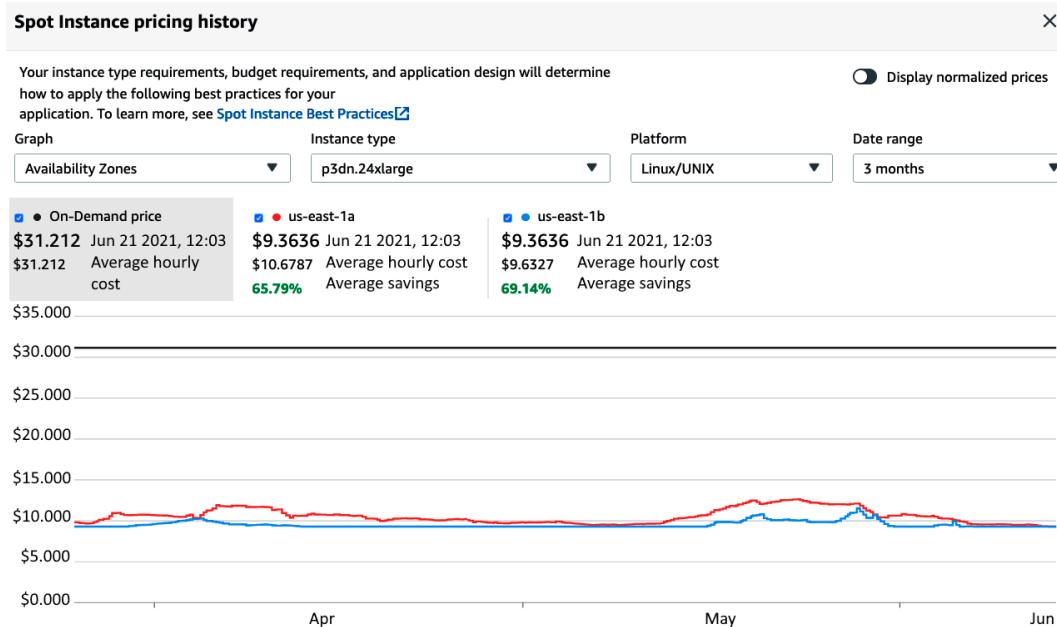


Figure 10.1 – Viewing the spot price of p3dn.24xlarge

These are EC2 prices, but the same discount rates apply to SageMaker prices. Discounts vary across instance types, regions, and even availability zones. You can use the `describe-spot-price-history` API to collect this information programmatically and use it in your workflows:

<https://docs.aws.amazon.com/cli/latest/reference/ec2/describe-spot-price-history.html>

Now, let's see what this means for SageMaker.

Understanding managed spot training

Training with Spot Instances is available in all SageMaker configurations: single-instance training, distributed training, built-in algorithms, frameworks, and your own algorithms.

Setting a couple of estimator parameters is all it takes. You don't need to worry about handling notifications and interruptions. SageMaker automatically does it for you.

If a training job is interrupted, SageMaker regains adequate spot capacity and relaunches the training job. If the algorithm uses checkpointing, training resumes from the latest checkpoint. If not, the job restarts from the beginning.

How much work is required to implement checkpointing depends on the algorithm you're using:

- The three built-in algorithms for computer vision and **XGBoost** support checkpointing.
- All other built-in algorithms don't. You can still train them with Spot Instances. However, the maximum running time is limited to 60 minutes to minimize potential waste. If your training job takes longer than 60 minutes, you should try scaling it. If that's not enough, you'll have to use on-demand instances.
- The **deep learning containers** for **TensorFlow**, **PyTorch**, **Apache MXNet**, and **Hugging Face** come with built-in checkpointing, and you don't need to modify your training script.
- If you use other frameworks or your own custom code, you need to implement checkpointing.

During training, checkpoints are saved inside the training container. The default path is `/opt/ml/checkpoints`, and you can customize it with an estimator parameter. SageMaker also automatically persists these checkpoints to a user-defined S3 path. If your training job is interrupted and relaunched, checkpoints are automatically copied inside the container. Your code can check for their presence and load the appropriate one to resume training.

Note

Please note that checkpointing is available even when you train with on-demand instances. This may come in handy if you'd like to store checkpoints in S3 for further inspection or for incremental training. The only restriction is that checkpointing is not available with **Local mode**.

Last but not least, checkpointing does slow down jobs, especially for large models. However, this is a small price to pay to avoid restarting long-running jobs from scratch.

Now, let's add managed spot training to the **object detection** job we ran in *Chapter 5, Training Computer Vision Models*.

Using managed spot training with object detection

Switching from on-demand training to managed spot training is very simple. We just have to set the maximum duration of the training job, including any time spent waiting for Spot Instances to be available.

We set a maximum running time of 2 hours, plus 8 hours for any spot delay. If either one of these bounds is exceeded, the job will be terminated automatically. This is helpful in killing runaway jobs that last much longer than expected or jobs that are stuck waiting for spot instances:

```
od = sagemaker.estimator.Estimator(  
    container,  
    role,  
    instance_count=2,  
    instance_type='ml.p3.2xlarge',  
    use_spot_instances=True,  
    max_run=7200,                      # 2 hour  
    max_wait=36000,                     # +8 hours  
    output_path=s3_output_location)
```

We train with the same configuration as before: Pipe mode and `dist_sync` mode. As the first epoch completes, the training log tells us that checkpointing is active. A new checkpoint is saved automatically each time the validation metric improves:

```
Updating the best model with validation-  
mAP=1.615789635726003e-05  
Saved checkpoint to "/opt/ml/model/model_algo_1-0000.params"
```

Once the training job is complete, the training log tells us how much we saved:

```
Training seconds: 7794  
Billable seconds: 2338  
Managed Spot Training savings: 70.0%
```

Not only is this job 70% cheaper than its on-demand counterpart, but it's also less than half the price of our original single-instance job. This means that we could use more instances and accelerate our training job for the same budget. Indeed, managed spot training lets you optimize the duration of a job and its cost. Instead of complex capacity planning, you can set a training budget that fits your business requirements, and then grab as much infrastructure as possible.

Let's try another example where we implement checkpointing in **Keras**.

Using managed spot training and checkpointing with Keras

In this example, we'll build a simple CNN to classify the **Fashion-MNIST** dataset. We've already worked with it in *Chapter 7, Extending Machine Learning Services with Built-in Frameworks*, and we'll use **Script mode** again. This time, we build our model using the old-style Sequential API in TensorFlow 2.1.

Checkpointing with Keras

Let's first look at the Keras script itself. For the sake of brevity, only important steps are presented here. You can find the full code in the GitHub repository for this book:

1. Using Script mode, we store dataset paths and hyperparameters.
2. Then, we load the dataset and normalize pixel values to the [0,1] range. We also one-hot encode class labels.
3. We build a Sequential model: two convolution blocks (Conv2D / BatchNormalization / ReLU / MaxPooling2D / Dropout), then two fully connected blocks (Dense / BatchNormalization / ReLU / Dropout), and finally, a softmax output layer for the 10 classes in the dataset.
4. We compile the model using the **categorical cross-entropy** loss function and the **Adam** optimizer:

```
model.compile(  
    loss=tf.keras.losses.categorical_crossentropy,  
    optimizer=tf.keras.optimizers.Adam(),  
    metrics=['accuracy'])
```

5. We define a Keras callback to checkpoint the model each time validation accuracy improves:

```
from tensorflow.keras.callbacks import ModelCheckpoint  
chk_dir = '/opt/ml/checkpoints'  
chk_name = 'fmnist-cnn-{epoch:04d}'  
checkpointer = ModelCheckpoint(  
    filepath=os.path.join(chk_dir, chk_name),  
    monitor='val_accuracy')
```

6. We train the model, adding the callback we just created:

```
model.fit(x=x_train, y=y_train,  
          validation_data=(x_val, y_val),  
          batch_size=batch_size, epochs=epochs,  
          callbacks=[checkpointer],  
          verbose=1)
```

7. When training is complete, we save the model in the **TensorFlow Serving** format, which is required to deploy on SageMaker:

```
from tensorflow.keras.models import save_model  
save_model(model, os.path.join(model_dir, '1'),  
           save_format='tf')
```

Now, let's look at our training notebook.

Training with managed spot training and checkpointing

We use the same workflow as before:

1. We download the Fashion-MNIST dataset and save it to a local directory. We upload the dataset to S3, and we define the S3 location where SageMaker should copy the checkpoints.
2. We configure a TensorFlow estimator, enabling managed spot training and passing the S3 output location for checkpoints. This time, we use an `ml.g4dn.xlarge` instance. This very cost-effective GPU instance (\$0.822 in `eu-west-1`) is more than enough for a small model:

```
from sagemaker.tensorflow import TensorFlow  
tf_estimator = TensorFlow(  
    entry_point='fmnist-1.py',  
    role=sagemaker.get_execution_role(),  
    instance_count=1,  
    instance_type='ml.g4dn.xlarge',  
    framework_version='2.1.0',  
    py_version='py3',  
    hyperparameters={'epochs': 20},  
    output_path=output_path,  
    use_spot_instances=True,  
    max_run=3600,
```

```
max_wait=7200,
checkpoint_s3_uri=chk_path)
```

3. We launch training as usual, and the job hits 93.11% accuracy. Training lasts 289 seconds, and we're only billed for 87 seconds, thanks to a 69.9% discount. The total cost is 1.98 cents! Who said GPU training had to be costly?
4. In the training log, we see that a checkpoint is created every time validation accuracy improves:

```
INFO:tensorflow:Assets written to /opt/ml/checkpoints/
fmnist-cnn-0001/assets
```

While the job is running, we also see that checkpoints are copied to S3:

```
$ aws s3 ls s3://sagemaker-eu-west-1-123456789012/keras2
fashion-mnist/checkpoints/
PRE fmnist-cnn-0001/
PRE fmnist-cnn-0002/
PRE fmnist-cnn-0003/
PRE fmnist-cnn-0006/
.
.
```

If our spot job gets interrupted, SageMaker will copy checkpoints inside the container so that we can use them to resume training. This requires some logic in our Keras script to load the latest checkpoint. Let's see how to do this.

Resuming training from a checkpoint

This is a pretty simple process—look for checkpoints, and resume training from the latest one:

1. We list the checkpoint directory:

```
import glob
checkpoints = sorted(
    glob.glob(os.path.join(chk_dir, 'fmnist-cnn-*')))
```

2. If checkpoints are present, we find the most recent and its epoch number. Then, we load the model:

```
from tensorflow.keras.models import load_model
if checkpoints :
    last_checkpoint = checkpoints[-1]
```

```
last_epoch = int(last_checkpoint.split('-')[-1])
model = load_model(last_checkpoint)
print('Loaded checkpoint for epoch ', last_epoch)
```

3. If no checkpoint is present, we build the model as usual:

```
else:
    last_epoch = 0
    model = Sequential()
    . . .
```

4. We compile the model, and we launch training, passing the number of the last epoch:

```
model.fit(x=x_train, y=y_train,
           validation_data=(x_val, y_val),
           batch_size=batch_size,
           epochs=epochs,
           initial_epoch=last_epoch,
           callbacks=[checkpointer],
           verbose=1)
```

How can we test this? There is no way to intentionally cause a spot interruption.

Here's the trick: start a new training job with existing checkpoints in the `checkpoint_s3_uri` path, and increase the number of epochs. This will simulate resuming an interrupted job.

Setting the number of epochs to 25 and keeping the checkpoints in `s3://sagemaker-eu-west-1-123456789012/keras2`

`fashion-mnist/checkpoints`, we launch the training job again.

In the training log, we see that the latest checkpoint is loaded and that training resumes at epoch 21:

```
Loaded checkpoint for epoch 20
. . .
Epoch 21/25
```

We also see that new checkpoints are created as validation accuracy improves, and they're copied to S3:

```
INFO:tensorflow:Assets written to: /opt/ml/checkpoints/fmnist-cnn-0021/assets
```

As you can see, it's not difficult to set up checkpointing in SageMaker, and you should be able to do the same for other frameworks. Thanks to this, you can enjoy the deep discount provided by managed spot training without the risk of losing any work if an interruption occurs. Of course, you can use checkpointing on its own to inspect intermediate training results, or for incremental training.

In the next section, we're going to introduce another important feature: automatic model tuning.

Optimizing hyperparameters with automatic model tuning

Hyperparameters have a huge influence on the training outcome. Just like in **chaos theory**, tiny variations of a single hyperparameter can cause wild swings in accuracy. In most cases, the "why?" evades us, leaving us perplexed about what to try next.

Over the years, several techniques have been devised to try to solve the problem of selecting optimal hyperparameters:

1. **Manual search:** This means using our best judgment and experience to select the "best" hyperparameters. Let's face it: this doesn't really work, especially with deep learning and its horde of training and network architecture parameters.
2. **Grid search:** This entails systematically exploring the hyperparameter space, zooming in on hot spots, and repeating the process. This is much better than a manual search. However, this usually requires training hundreds of jobs. Even with scalable infrastructure, the time and dollar budgets can be significant.
3. **Random search:** This refers to selecting hyperparameters at random. Unintuitive as it sounds, James Bergstra and Yoshua Bengio (of Turing Award fame) proved in 2012 that this technique delivers better models than a grid search with the same compute budget
4. <http://www.jmlr.org/papers/v13/bergstra12a.html>

5. **Hyperparameter optimization** (HPO): This means using optimization techniques to select hyperparameters, such as **Bayesian optimization** and **Gaussian process regression**. With the same compute budget, HPO typically delivers results with 10x fewer training epochs than other techniques.

Understanding automatic model tuning

SageMaker includes an **automatic model tuning** capability that lets you easily explore hyperparameter ranges and quickly optimize any training metric with a limited number of jobs.

Model tuning supports both random search and HPO. The former is an interesting baseline that helps you to check whether the latter is indeed overperforming. You can find a very detailed comparison in this excellent blog post:

<https://aws.amazon.com/blogs/machine-learning/amazon-sagemaker-automatic-model-tuning-now-supports-random-search-and-hyperparameter-scaling/>

Model tuning is completely agnostic to the algorithm you're using. It works with built-in algorithms, and the documentation lists the hyperparameters that can be tuned. It also works with all frameworks and custom containers, and hyperparameters are passed in the same way.

For each hyperparameter that we want to optimize, we have to define the following:

- A name
- A type (parameters can either be an integer, continuous, or categorical)
- A range of values to explore
- A scaling type (linear, logarithmic, or reverse logarithmic, or auto)—this lets us control how a specific parameter range will be explored

We also define the metric we want to optimize for. It can be any numerical value as long as it's visible in the training log and you can pass a regular expression to extract it.

Then, we launch the tuning jobs, passing all of these parameters as well as the number of training jobs to run and their degree of parallelism. With Bayesian optimization, you'll get the best results with sequential jobs (no parallelism), as optimization can be applied after each job. Having said that, running a small number of jobs in parallel is acceptable. Random search has no restrictions on parallelism as jobs are completely unrelated.

Calling the `deploy()` API on the tuner object deploys the best model. If tuning is still in progress, it will deploy the best model so far, which can be useful for early testing.

Let's run the first example with a built-in algorithm and learn about the model tuning API.

Using automatic model tuning with object detection

We're going to optimize our object detection job. Looking at the documentation, we can see the list of tunable hyperparameters:

<https://docs.aws.amazon.com/sagemaker/latest/dg/object-detection-tuning.html>

Let's try to optimize the learning rate, momentum, and weight decay:

1. We set up the input channels using Pipe mode. There's no change here.
2. We also configure the estimator as usual, setting up managed spot training to minimize costs. We'll train on a single instance for maximum accuracy:

```
od = sagemaker.estimator.Estimator(  
    container,  
    role,  
    instance_count=1,  
    instance_type='ml.p3.2xlarge',  
    output_path=s3_output_location,  
    use_spot_instances=True,  
    max_run=7200,  
    max_wait=36000,  
    volume_size=1)
```

3. We use the same hyperparameters as before:

```
od.set_hyperparameters(base_network='resnet-50',  
                      use_pretrained_model=1,  
                      num_classes=20,  
                      epochs=30,  
                      num_training_samples=16551,  
                      mini_batch_size=90)
```

4. We define the three extra hyperparameters we want to tune. We explicitly set logarithmic scaling for the learning rate, to make sure that different orders of magnitude are explored:

```
from sagemaker.tuner import ContinuousParameter,  
hyperparameter_ranges = {  
    'learning_rate': ContinuousParameter(0.001, 0.1,  
                                         scaling_type='Logarithmic'),  
    'momentum': ContinuousParameter(0.8, 0.999),  
    'weight_decay': ContinuousParameter(0.0001, 0.001)  
}
```

5. We set the metric to optimize for:

```
objective_metric_name = 'validation:mAP'  
objective_type = 'Maximize'
```

6. We put everything together, using the `HyperparameterTuner` object. We decide to run 30 jobs, with two jobs in parallel. We also enable early stopping to weed out low performing jobs, saving us time and money:

```
from sagemaker.tuner import HyperparameterTuner  
tuner = HyperparameterTuner(od,  
                           objective_metric_name,  
                           hyperparameter_ranges,  
                           objective_type=objective_type,  
                           max_jobs=30,  
                           max_parallel_jobs=2,  
                           early_stopping_type='Auto')
```

7. We launch training on the tuner object (not on the estimator) without waiting for it to complete:

```
tuner.fit(inputs=data_channels, wait=False)
```

8. At the moment, **SageMaker Studio** doesn't provide a convenient view of tuning jobs. Instead, we can track progress in the **Hyperparameter tuning jobs** section of the SageMaker console, as shown in the following screenshot:

Training job status counter

Completed 17 In Progress 0 Stopped 13 Failed 0 (Retryable: 0, Non-retryable: 0)

Training jobs

Sorting by objective metric value will display only jobs that have metric values.

Name	Status	Objective metric value	Creation time	Training Duration
object-detection-200603-1732-030-0c8822af	Stopped	0.4082950949668884	Jun 04, 2020 06:05 UTC	1 hour(s), 12 minute(s)
object-detection-200603-1732-029-ca57f3e5	Completed	0.629538893699646	Jun 04, 2020 05:55 UTC	1 hour(s), 14 minute(s)
object-detection-200603-1732-028-7dd1a914	Stopped	0.1219444984197617	Jun 04, 2020 05:40 UTC	13 minute(s)
object-detection-200603-1732-027-1d3304a3	Completed	0.545764684677124	Jun 04, 2020 04:50 UTC	1 hour(s), 13 minute(s)

Figure 10.2 – Viewing tuning jobs in the SageMaker console

The job runs for 17 hours (wall time). 22 jobs completed and 8 stopped early. The total training time is 30 hours and 15 minutes. Applying the 70% spot discount, the total cost is $25.25 * \$4.131 * 0.3 = \37.48 .

How well did this tuning job do? With default hyperparameters, our standalone training job reached a **mAP** accuracy of 0 . 2453. Our tuning job hits 0 . 6337, as shown in the following screenshot:

Best training job summary

This training job is the best training job for only this hyperparameter tuning job.

Name	Status	Objective metric	Value
object-detection-210702-0916-024-a7253d1f	Completed	validation:mAP	0.6337428689002991

Figure 10.3 – Tuning job results

The graph for validation mAP is shown in the next image. It tells me that we could probably train a little longer and get extra accuracy:

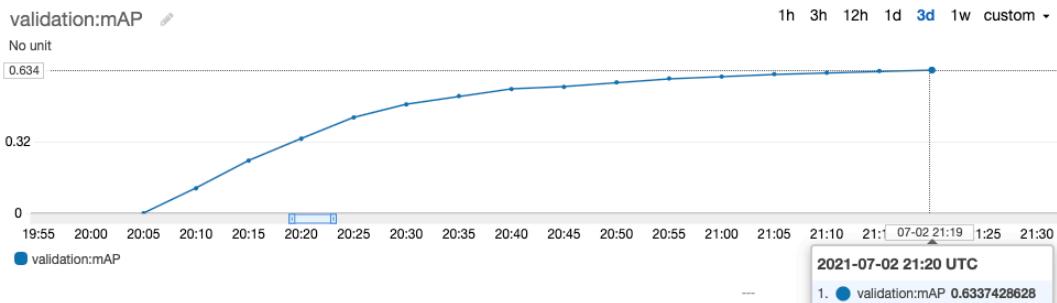


Figure 10.4 – Viewing the mAP metric

One idea would be to launch a single training job with the best hyperparameters and let it run for more epochs. We could also resume the tuning job using **warm start** and continue exploring the hyperparameter range. We also call `deploy()` on the tuner object and test our model just like any SageMaker model.

As you can see, automatic model tuning is extremely powerful. By running a small number of jobs, we improved our metric by 158%! The cost is negligible compared to the time you would spend experimenting with other techniques.

In fact, running the same tuning job using the random strategy delivers a top accuracy of 0.52. We would certainly need to run many more training jobs to even hope hitting 0.6315.

Let's now try to optimize the Keras example we used earlier in this chapter.

Using automatic model tuning with Keras

Automatic model tuning can easily be used any algorithm on SageMaker, which of course includes all frameworks. Let's see how this works with Keras.

Earlier in this chapter, we trained our Keras CNN on the Fashion MNIST dataset for 20 epochs and reached a validation accuracy of 93.11%. Let's see if we can improve it with automatic model tuning. In the process, we'll also learn how to optimize for any metric present in the training log, not just metrics that are predefined in SageMaker.

Optimizing on a custom metric

Modifying our training script, we install the `keras-metrics` package (<https://github.com/netrak/keras-metrics>) and add the **precision**, **recall**, and **f1 score** metrics to the training log:

```
import subprocess, sys
def install(package):
    subprocess.call([sys.executable, "-m", "pip",
                    "install", package])
install('keras-metrics')
import keras_metrics
...
model.compile(
    loss=tf.keras.losses.categorical_crossentropy,
    optimizer=tf.keras.optimizers.Adam(),
    metrics=['accuracy',
              keras_metrics.precision(),
              keras_metrics.recall(),
              keras_metrics.f1_score()])
```

After 20 epochs, the metrics now look like this:

```
loss: 0.0869 - accuracy: 0.9678 - precision: 0.9072 - recall:
0.8908 - f1_score: 0.8989 - val_loss: 0.2301 - val_accuracy:
0.9310 - val_precision: 0.9078 - val_recall: 0.8915 - val_f1_
score: 0.8996
```

If we wanted to optimize on the f1 score, we would define the tuner metrics like this:

```
objective_metric_name = 'val_f1'
objective_type = 'Maximize'
metric_definitions = [
    {'Name': 'val_f1',
     'Regex': 'val_f1_score: ([0-9\\.]+)'}
]
```

That's all it takes. As long as a metric is printed in the training log, you can use it to tune models.

Optimizing our Keras model

Now, let's run our tuning job:

1. We define the metrics for HyperparameterTuner like so, optimizing for accuracy and also displaying the f1 score:

```
objective_metric_name = 'val_acc'  
objective_type = 'Maximize'  
metric_definitions = [  
    {'Name': 'val_f1',  
     'Regex': 'val_f1_score: ([0-9\\.]+)'},  
    {'Name': 'val_acc',  
     'Regex': 'val_accuracy: ([0-9\\.]+)'}  
]
```

2. We define the parameter ranges to explore:

```
from sagemaker.tuner import ContinuousParameter,  
IntegerParameter  
hyperparameter_ranges = {  
    'learning_rate': ContinuousParameter(0.001, 0.2,  
                                         scaling_type='Logarithmic'),  
    'batch-size': IntegerParameter(32, 512)  
}
```

3. We use the same estimator (20 epochs with spot instances) and we define the tuner:

```
tuner = HyperparameterTuner(  
    tf_estimator,  
    objective_metric_name,  
    hyperparameter_ranges,  
    metric_definitions=metric_definitions,  
    objective_type=objective_type,  
    max_jobs=20,  
    max_parallel_jobs=2,  
    early_stopping_type='Auto')
```

- We launch the tuning job. While it's running, we can use the **SageMaker SDK** to display the list of training jobs and their properties:

```
from sagemaker.analytics import
HyperparameterTuningJobAnalytics
exp = HyperparameterTuningJobAnalytics(
    hyperparameter_tuning_job_name=
    tuner.latest_tuning_job.name)
jobs = exp.dataframe()
jobs.sort_values('FinalObjectiveValue', ascending=0)
```

This prints out the table visible in the next screenshot:

batch-size	learning_rate	TrainingJobName	TrainingJobStatus	FinalObjectiveValue	TrainingStartTime	TrainingEndTime	TrainingElapsedTimeSeconds
2 32.0	0.110305	tensorflow-training-210702-1206-004-ct0fe06c	Completed	0.9344	2021-07-02 12:16:07+00:00	2021-07-02 12:21:17+00:00	310.0
5 165.0	0.045242	tensorflow-training-210702-1206-001-8eb0922b	Completed	0.9315	2021-07-02 12:09:12+00:00	2021-07-02 12:13:10+00:00	238.0
4 297.0	0.013010	tensorflow-training-210702-1206-002-5570de46	Stopped	0.9229	2021-07-02 12:09:31+00:00	2021-07-02 12:12:09+00:00	158.0
3 404.0	0.136788	tensorflow-training-210702-1206-003-5abd31e5	Completed	0.9228	2021-07-02 12:14:36+00:00	2021-07-02 12:18:09+00:00	213.0
0 32.0	0.076613	tensorflow-training-210702-1206-006-2632fa65	InProgress	NaN	NaT	NaT	NaN
1 126.0	0.054494	tensorflow-training-210702-1206-005-9807cb1	InProgress	NaN	2021-07-02 12:21:24+00:00	NaT	NaN

Figure 10.5 – Viewing information on a tuning job

The tuning job runs for 2 hours and 8 minutes (wall time). Top validation accuracy is 93.46% – a decent improvement over our baseline.

We could certainly do better by training longer. However, the longer we train for, the more overfitting becomes a concern. We can alleviate it with early stopping, which can be implemented with a Keras callback. However, we should make sure that the job reports the metric for the best epoch, not for the last epoch. How can we display this in the training log? With another callback!

Adding callbacks for early stopping

Adding a Keras callback for early stopping is very simple:

- We add a built-in callback for early stopping, based on validation accuracy:

```
from tensorflow.keras.callbacks import EarlyStopping
early_stopping = EarlyStopping(
    monitor='val_accuracy',
    min_delta=0.0001,
```

```
    patience=10,  
    verbose=1,  
    mode='auto')
```

2. We add a custom callback to store validation accuracy at the end of each epoch, and to display the best one at the end of training:

```
from tensorflow.keras.callbacks import Callback  
  
class LogBestMetric(Callback):  
    def on_train_begin(self, logs={}):  
        self.val_accuracy = []  
  
    def on_train_end(self, logs={}):  
        print("Best val_accuracy:",  
              max(self.val_accuracy))  
  
    def on_epoch_end(self, batch, logs={}):  
        self.val_accuracy.append(  
            logs.get('val_accuracy'))  
        best_val_metric = LogBestMetric()
```

3. We add these two callbacks to the training API:

```
model.fit(. . .  
          callbacks=[checkpointer, early_stopping,  
                     best_val_metric])
```

Testing with a few individual jobs, the last lines of the training log now look like this:

```
Epoch 00048: early stopping  
Best val_accuracy: 0.9259
```

4. In the notebook, we update our metric definition in order to extract the best validation accuracy:

```
objective_metric_name = 'val_acc'  
objective_type = 'Maximize'  
metric_definitions = [  
    {'Name': 'val_acc',
```

```
'Regex': 'Best val_accuracy: ([0-9\\.]+)'}
```

```
]
```

Training for 60 epochs this time (about 3 hours wall time), top validation accuracy is now at 93.78%. It looks like this is as good as it gets by tweaking the learning rate and the batch size.

Using automatic model tuning for architecture search

Our neural network has plenty more hyperparameters: number of convolution filters, dropout, and so on. Let's try to optimize these as well:

1. We modify our training script to add command-line parameters for the following network parameters, which are used by Keras layers in our model:

```
parser.add_argument(
    '--filters1', type=int, default=64)
parser.add_argument(
    '--filters2', type=int, default=64)
parser.add_argument(
    '--dropout-conv', type=float, default=0.2)
parser.add_argument(
    '--dropout-fc', type=float, default=0.2)
```

As you certainly guessed, the parameters let us set values for the number of convolution filters in each layer, the dropout value for convolution layers, and the dropout value for fully connected layers.

2. Accordingly, in the notebook, we define these hyperparameters and their ranges. For the learning rate and the batch size, we use narrow ranges centered on the optimal values discovered by the previous tuning job:

```
from sagemaker.tuner import ContinuousParameter,
                                IntegerParameter
hyperparameter_ranges = {
    'learning-rate': ContinuousParameter(0.01, 0.14),
    'batch-size': IntegerParameter(130, 160),
    'filters1': IntegerParameter(16, 256),
    'filters2': IntegerParameter(16, 256),
    'dropout-conv': ContinuousParameter(0.001, 0.5,
                                         scaling_type='Logarithmic'),
    'dropout-fc': ContinuousParameter(0.001, 0.5,
```

```
        scaling_type='Logarithmic')  
    }
```

3. We launch the tuning job, running 50 jobs two at a time for 100 epochs.

The tuning job runs for about 12 hours, for a total cost of about \$15. Top validation accuracy hits 94.09%. Compared to our baseline, automatic model tuning has improved the accuracy of our model by almost 1 percentage point – a very significant gain. If this model is used to predict 1 million samples a day, this translates to over 10,000 additional accurate predictions!

In total, we've spent less about \$50 on tuning our Keras model. Whatever business metric would be improved by the extra accuracy, it's fair to say that this spend would be recouped in no time. As many customers have told me, automatic model tuning pays for itself, and then some.

This concludes our exploration of automatic model tuning, one of my favorite features in SageMaker. You can find more examples at https://github.com/awslabs/amazon-sagemaker-examples/tree/master/hyperparameter_tuning.

Now, let's learn about SageMaker Debugger, and how it can help us to understand what's happening inside our models.

Exploring models with SageMaker Debugger

SageMaker Debugger lets you configure *debugging rules* for your training job. These rules will inspect its internal state and check for specific unwanted conditions that could be developing during training. SageMaker Debugger includes a long list of built-in rules (<https://docs.aws.amazon.com/sagemaker/latest/dg/debugger-built-in-rules.html>), and you can add your own written in Python.

In addition, you can save and inspect the model state (gradients, weights, and so on) as well as the training state (metrics, optimizer parameters, and so on). At each training step, the **tensors** storing these values may be saved in near-real-time in an S3 bucket, making it possible to visualize them while the model is training.

Of course, you can select the tensor **collections** that you'd like to save, how often, and so on. Depending on the framework you use, different collections are available. You can find more information at <https://github.com/awslabs/sagemaker-debugger/blob/master/docs/api.md>. Last but not least, you can save either raw tensor data or tensor reductions to limit the amount of data involved. Reductions include min, max, median, and more.

If you are working with the built-in containers for supported versions of TensorFlow, PyTorch, Apache MXNet, or the built-in XGBoost algorithm, you can use SageMaker Debugger out of the box, without changing a line of code in your script. Yes, you read that right. All you have to do is add extra parameters to the estimator, as we will in the next examples.

With other versions, or with your own containers, minimal modifications are required. You can find the latest information and examples at <https://github.com/aws-labs/sagemaker-debugger>.

Debugging rules and saving tensors can be configured on the same training job. For clarity, we'll run two separate examples. First, let's use the XGBoost and Boston Housing example from *Chapter 4, Training Machine Learning Models*.

Debugging an XGBoost job

First, we will configure several built-in rules, train our model, and check the status of all rules:

1. Taking a look at the list of built-in rules, we decide to use `overtraining` and `overfit`. Each rule has extra parameters that we could tweak. We stick to defaults, and we configure the Estimator accordingly:

```
from sagemaker.debugger import rule_configs, Rule
xgb_estimator = Estimator(container,
                           role=sagemaker.get_execution_role(),
                           instance_count=1,
                           instance_type='ml.m5.large',
                           output_path='s3://{}//{}//output'.format(bucket, prefix),
                           rules=[  
    Rule.sagemaker(rule_configs.overtraining()),  
    Rule.sagemaker(rule_configs.overfit())  
]  
)
```

2. We set hyperparameters and launch training without waiting for the training job to complete. The training log won't be visible in the notebook, but it will still be available in **CloudWatch Logs**:

```
xgb_estimator.set_hyperparameters(  
    objective='reg:linear', num_round=100)  
xgb_estimator.fit(xgb_data, wait=False)
```

3. In addition to the training job, one debugging job per rule is running under the hood, and we can check their statuses:

```
description = xgb_estimator.latest_training_job.rule_job_
summary()

for rule in description:
    rule.pop('LastModifiedTime')
    rule.pop('RuleEvaluationJobArn')
    print(rule)
```

This tells us that the debugger jobs are running:

```
{'RuleConfigurationName': 'Overtraining',
 'RuleEvaluationStatus': 'InProgress'}
{'RuleConfigurationName': 'Overfit',
 'RuleEvaluationStatus': 'InProgress'}
```

4. Running the same cell once the training job is complete, we see that no rule was triggered:

```
{'RuleConfigurationName': 'Overtraining',
 'RuleEvaluationStatus': 'NoIssuesFound'}
{'RuleConfigurationName': 'Overfit',
 'RuleEvaluationStatus': 'NoIssuesFound'}
```

Had a rule been triggered, we would get an error message, and the training job would be stopped. Inspecting tensors stored in S3 would help us understand what went wrong.

Inspecting an XGBoost job

Let's configure a new training job that saves all tensor collections available for XGBoost:

1. We configure the `Estimator`, passing a `DebuggerHookConfig` object. We save three tensor collections at each training step: metrics, feature importance, and average **SHAP** (<https://github.com/slundberg/shap>) values. These help us understand how each feature in a data sample contributes to increasing or decreasing the predicted value.

For larger models and datasets, this could generate a lot of data, which would take a long time to load and analyze. We would either increase the save interval or save tensor reductions instead of full tensors:

```
from sagemaker.debugger import DebuggerHookConfig,
CollectionConfig
save_interval = '1'
xgb_estimator = Estimator(container,
role=role,
instance_count=1,
instance_type='ml.m5.large',
output_path='s3://{{}}/{{}}/output'.format(bucket,
prefix),  
  
debugger_hook_config=DebuggerHookConfig(
s3_output_path=
's3://{{}}/{{}}/debug'.format(bucket,prefix),
collection_configs=[  
    CollectionConfig(name='metrics',
parameters={"save_interval":
save_interval}),  
    CollectionConfig(name='average_shap',
parameters={"save_interval":
save_interval}),  
    CollectionConfig(name='feature_importance',
parameters={"save_interval": save_interval})
]  
)  
)
```

- Once the training job has started, we can create a trial and load data that has already been saved. As this job is very short, we see all data within a minute or so:

```
from smdebug.trials import create_trial
s3_output_path = xgb_estimator.latest_job_debugger_
artifacts_path()
trial = create_trial(s3_output_path)
```

3. We can list the name of all tensors that were saved:

```
trial.tensor_names()  
['average_shap/f0', 'average_shap/f1', 'average_shap/f10',  
...  
'feature_importance/cover/f0', 'feature_importance/cover/  
f1', ...  
'train-rmse', 'validation-rmse']
```

4. We can also list the name of all tensors in a given collection:

```
trial.tensor_names(collection="metrics")  
['train-rmse', 'validation-rmse']
```

5. For each tensor, we can access training steps and values. Let's plot feature information from the `average_shap` and `feature_importance` collections:

```
def plot_features(tensor_prefix):  
    num_features = len(dataset.columns)-1  
    for i in range(0,num_features):  
        feature = tensor_prefix+'/'+str(i)  
        steps = trial.tensor(feature).steps()  
        v = [trial.tensor(feature).value(s) for s in steps]  
        plt.plot(steps, v, label=dataset.columns[i+1])  
        plt.autoscale()  
        plt.title(tensor_prefix)  
        plt.legend(loc='upper left')  
        plt.show()
```

6. We build the `average_shap` plot:

```
plot_features('average_shap')
```

7. You can see it in the following screenshot – **dis**, **crim**, and **nox** have the largest average values:

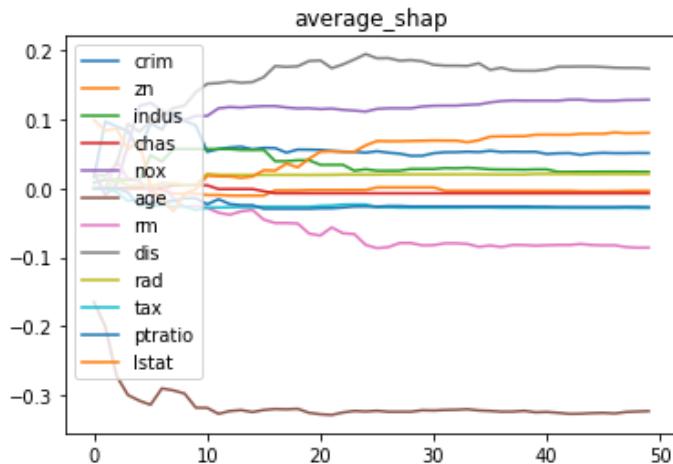


Figure 10.6 – Plotting average SHAP values over time

8. We build the `feature_importance/weight` plot:

```
plot_features('feature_importance/weight')
```

You can see it in the following screenshot – **crim**, **age**, and **dis** have the largest weights:

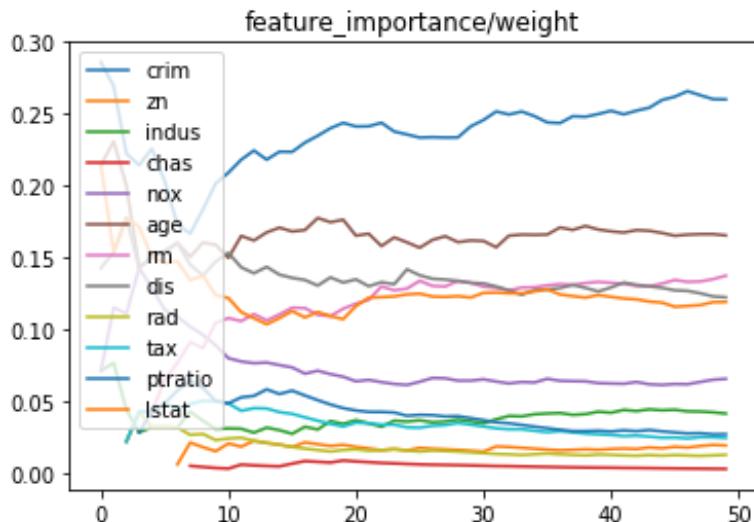


Figure 10.7 – Plotting feature weights over time

Now, let's use SageMaker Debugger on our Keras and Fashion-MNIST example.

Debugging and inspecting a Keras job

We can inspect and debug a Keras job using the following steps:

1. The default behavior in TensorFlow 2.x is eager mode, where gradients are not available. Hence, we disable eager mode in our script, which is the only modification required:

```
tf.compat.v1.disable_eager_execution()
```

2. We start from the same estimator. The dataset has 70,000 samples (60,000 for training, plus 10,000 for validation). With 30 epochs and a batch size of 128, our training job will have about 16,400 steps ($70,000 * 30 / 128$). Saving tensors at each step feels like overkill. Let's save them every 100 steps instead:

```
from sagemaker.tensorflow import TensorFlow
from sagemaker.debugger import rule_configs, Rule,
    DebuggerHookConfig, CollectionConfig
save_interval = '100'
tf_estimator = TensorFlow(entry_point='fmnist-5.py',
    role=role,
    instance_count=1,
    instance_type='ml.p3.2xlarge',
    framework_version='2.1.0',
    py_version='py3',
    hyperparameters={'epochs': 30},
    output_path=output_path,
    use_spot_instances=True,
    max_run=3600,
    max_wait=7200,
```

3. Looking at the built-in rules available for TensorFlow, we decide to set up `poor_weight_initialization`, `dead_relu`, and `check_input_images`. We need to specify the index of channel information in the input tensor. It's 4 for TensorFlow (batch size, height, width, and channels):

```
rules=[
    Rule.sagemaker(
        rule_configs.poor_weight_INITIALIZATION()),
    Rule.sagemaker(
        rule_configs.dead_relu()),
```

```
    Rule.sagemaker(
        rule_configs.check_input_images(),
        rule_parameters={"channel": '3'})
    ] ,
```

4. Looking at the collections available for TensorFlow, we decide to save metrics, losses, outputs, weights, and gradients:

```
    debugger_hook_config=DebuggerHookConfig(
        s3_output_path='s3://{}//{}//debug'
            .format(bucket, prefix),
        collection_configs=[
            CollectionConfig(name='metrics',
                parameters={"save_interval":
                    save_interval}),
            CollectionConfig(name='losses',
                parameters={"save_interval":
                    save_interval}),
            CollectionConfig(name='outputs',
                parameters={"save_interval":
                    save_interval}),
            CollectionConfig(name='weights',
                parameters={"save_interval":
                    save_interval}),
            CollectionConfig(name='gradients',
                parameters={"save_interval":
                    save_interval}))
        ],
    )
)
```

5. As training starts, we see the rules being launched in the training log:

```
***** Debugger Rule Status *****
*
* PoorWeightInitialization: InProgress
* DeadRelu: InProgress
* CheckInputImages: InProgress
```

```
*  
*****
```

6. When training is complete, we check the status of the debugging rules:

```
description = tf_estimator.latest_training_job.rule_job_summary()  
  
for rule in description:  
    rule.pop('LastModifiedTime')  
    rule.pop('RuleEvaluationJobArn')  
    print(rule)  
  
    {'RuleConfigurationName': 'PoorWeightInitialization',  
     'RuleEvaluationStatus': 'NoIssuesFound'}  
  
    {'RuleConfigurationName': 'DeadRelu',  
     'RuleEvaluationStatus': 'NoIssuesFound'}  
  
    {'RuleConfigurationName': 'CheckInputImages',  
     'RuleEvaluationStatus': 'NoIssuesFound'}
```

7. We create a trial using the same tensors saved in S3:

```
from smdebug.trials import create_trial  
s3_output_path = tf_estimator.latest_job_debugger_artifacts_path()  
trial = create_trial(s3_output_path)
```

8. Let's inspect the filters in the first convolution layer:

```
w = trial.tensor('conv2d/weights/conv2d/kernel:0')  
g = trial.tensor(  
    'training/Adam/gradients/gradients/conv2d/Conv2D_grad/  
    Conv2DBackpropFilter:0')  
print(w.value(0).shape)  
print(g.value(0).shape)  
(3, 3, 1, 64)  
(3, 3, 1, 64)
```

As defined in our training script, the first convolution layer has 64 filters. Each one is 3x3 pixels, with a single channel (2D). Accordingly, gradients have the same shape.

9. We write a function to plot filter weights and gradients over time, and we plot weights in the last filter of the first convolution layer:

```
plot_conv_filter('conv2d/weights/conv2d/kernel:0', 63)
```

You can see the graph in the following screenshot:

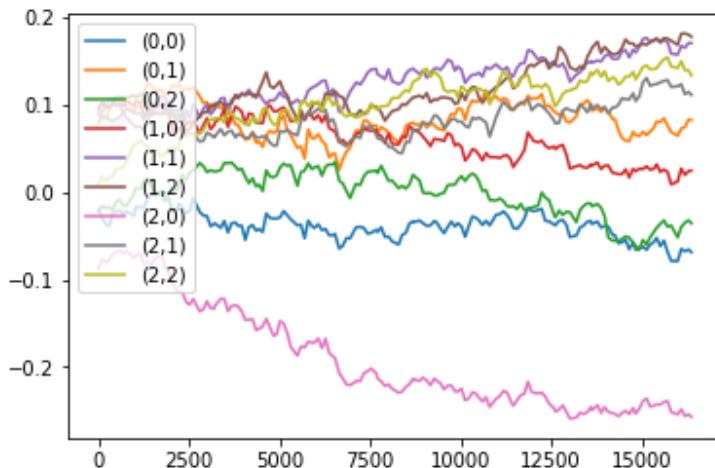


Figure 10.8 – Plotting the weights of a convolution filter over time

As you can see, SageMaker Debugger makes it really easy to inspect training jobs. If you work with the built-in containers that support it, you don't need to modify your code. All configuration takes place in the estimator.

You can find additional examples at <https://github.com/awslabs/amazon-sagemaker-examples>, including some advanced use cases such as real-time visualization and model pruning.

This concludes the first part of the chapter, where we learned how to optimize the cost of training jobs with managed spot training, their accuracy with automatic model tuning, and how to inspect their internal state with SageMaker Debugger.

In the second part, we're going to dive into two advanced capabilities that will help us build better training workflows – SageMaker Feature Store and SageMaker Clarify.

Managing features and building datasets with SageMaker Feature Store

Until now, we've engineered our training and validation features in a notebook or in a SageMaker Processing script, before storing them as S3 objects. Then, we used these objects as-is to train and evaluate models. This is a perfectly reasonable workflow. However, the following questions may arise as your machine learning workflows grow and mature:

- How can we apply a well-defined schema to our features?
- How can we select a subset of our features to build different datasets?
- How can we store and manage different feature versions?
- How can we discover and reuse feature engineering by other teams?
- How can we access engineered features at prediction time?

SageMaker Feature Store is designed to answer these questions. Let's add it to the classification training workflow we built with BlazingText and Amazon Reviews in *Chapter 6, Training Natural Language Processing Models*.

Engineering features with SageMaker Processing

We can reuse our previous SageMaker Processing job almost as-is. The only difference is the output format of the engineered data. In the original job, we saved it as a plain text file according to the input format expected by BlazingText. This format is inconvenient for SageMaker Feature Store, as we need easy access to each column. CSV doesn't work either as reviews contain commas, so we decide to use TSV instead:

1. Accordingly, we add a few lines to our processing script:

```
fs_output_dir = '/opt/ml/processing/output/fs/'  
os.makedirs(fs_output_dir, exist_ok=True)  
fs_output_path = os.path.join(fs_output_dir, 'fs_data.  
tsv')  
data.to_csv(fs_output_path, index=False, header=True,  
sep='\t')
```

2. Running our SageMaker Processing job as before, we now see two outputs: a plain text output for BlazingText (in case we wanted to train directly on the full dataset) and a TSV output that we'll ingest in SageMaker Feature Store:

```
s3://sagemaker-us-east-1-123456789012/sagemaker-scikit-  
learn-2021-07-07-54-15-145/output/bt_data
```

```
s3://sagemaker-us-east-1-123456789012/sagemaker-scikit-
learn-2021-07-05-07-54-15-145/output/fs_data
```

- Let's load the TSV file in a pandas dataframe and display the first few rows:

```
fs_training_output_path = 's3://sagemaker-
us-east-1-123456789012/sagemaker-scikit-
learn-2021-07-05-07-54-15-145/output/fs_data/fs_data.tsv'
data = pd.read_csv(fs_training_output_path, sep='\t',
                    error_bad_lines=False, dtype='str')
data.head()
```

This prints out the table visible in the next image:

	review_id	product_id	star_rating	review_body	label
0	R1NBG94582SJ2	B00I01JQJM	5	five stars ok	_label_positive_
1	R273DCA6Y0H9V7	B00TC00ZAA	5	love it !!! perfect , even sturdier than the...	_label_positive_
2	RQVOXO7WUOKF6	B00B7733E0	2	another motion detect fail if the words , & # ...	_label_negative_
3	R1KWKSF21PO6HO	B006ZN4U34	5	exactly what i wanted and expected . exactly w...	_label_positive_
4	R38H3UO1J190GI	B00HUEBGMU	5	good mic at a good price ... not canon though ...	_label_positive_

Figure 10.9 – Viewing the first rows

Now, let's create a feature group where we'll ingest this data.

Creating a feature group

A **feature group** is a resource that stores a collection of related features. Feature groups are organized in rows, which have a unique identifier and a timestamp. Each row contains key-value pairs, where each pair represents a feature name and a feature value.

- First, let's define the name of our feature group:

```
from sagemaker.feature_store.feature_group import
FeatureGroup
feature_group_name = 'amazon-reviews-feature-group-' +
strftime('%d-%H-%M-%S', gmtime())
feature_group = FeatureGroup(
    name=feature_group_name,
    sagemaker_session=feature_store_session)
```

2. Next, we set the name of the feature that contains a unique identifier – `review_id` works perfectly here, and you could use any unique value present in your data source, such as a primary key:

```
record_identifier_feature_name = 'review_id'
```

3. Then, we add a timestamp column to all rows in our pandas dataframe. If your data source already contains a timestamp, you can reuse that value, either in the `float64` format or in the **UNIX** date/time format:

```
event_time_feature_name = 'event_time'  
current_time_sec = int(round(time.time()))  
data = data.assign(event_time=current_time_sec)
```

Our dataframe now looks like the following picture:

	review_id	product_id	star_rating	review_body	label	event_time
0	R1NBG94582SJ2	B00I01JQJM	5	five stars ok	_label_positive_	1625473181
1	R273DCA6Y0H9V7	B00TC00ZAA	5	love it !!! perfect , even sturdier than the...	_label_positive_	1625473181
2	RQVOXO7WUOK6	B00B7733E0	2	another motion detect fail if the words , & # ...	_label_negative_	1625473181
3	R1KWKSF21PO6HO	B006ZN4U34	5	exactly what i wanted and expected . exactly w...	_label_positive_	1625473181
4	R38H3UO1J190GI	B00HUEBGMU	5	good mic at a good price ... not canon though ...	_label_positive_	1625473181

Figure 10.10 – Viewing timestamps

4. The next step is to define a schema for the feature group. We can either provide it explicitly in a JSON document or let SageMaker pick it up from the pandas dataframe. We use the second option:

```
data['review_id'] = data['review_id']  
    .astype('str').astype('string')  
data['product_id'] = data['product_id']  
    .astype('str').astype('string')  
data['review_body'] = data['review_body']  
    .astype('str').astype('string')  
data['label'] = data['label']  
    .astype('str').astype('string')  
data['star_rating'] = data['star_rating']  
    .astype('int64')  
data['event_time'] = data['event_time']  
    .astype('float64')
```

We then load feature definitions:

```
feature_group.load_feature_definitions(
    data_frame=data)
```

- Finally, we create the feature group, passing the S3 location where features will be stored. This is where we'll query them to build datasets. We enable the online store, which will give us low-latency access to features at prediction time. We also add a description and tags which make it easier to discover the feature group:

```
feature_group.create(
    role_arn=role,
    s3_uri='s3://{}{}'.format(default_bucket, prefix),
    enable_online_store=True,
    record_identifier_name=
        record_identifier_feature_name,
    event_time_feature_name=
        event_time_feature_name,
    description="1.8M+ tokenized camera reviews from the
                Amazon Customer Reviews dataset",
    tags=[
        { 'Key': 'Dataset',
          'Value': 'amazon customer reviews' },
        { 'Key': 'Subset',
          'Value': 'cameras' },
        { 'Key': 'Owner',
          'Value': 'Julien Simon' }
    ]
)
```

After a few seconds, the feature group is ready and visible in SageMaker Studio, under **Components and registries / Feature Store**, as shown in the following screenshot:

FEATURE STORE							
<input type="text"/> Search column name to start Create feature group							
Feature group name	Short description	Tags	Record Identifier	Store type	Status	Created	Offline store status
amazon-reviews-feature-group-05-09-03-39	1.8M+ tokenized camera reviews f...	Owner: Julien Simon +2 review_id	review_id	Online/Offline	Created	3 hours ago	Active End of the list

Figure 10.11 – Viewing a feature group

Now, let's ingest data.

Ingesting features

SageMaker Feature Store lets us ingest data in three ways:

- Call the `PutRecord()` API to ingest a single record.
- Call the `ingest()` API to upload the contents of a pandas dataframe.
- If we used **SageMaker Data Wrangler** for feature engineering, use an auto-generated notebook to create a feature group and ingest data.

We use the second option here, which is as simple as the following:

```
feature_group.ingest(data_frame=data, max_workers=10,  
                      wait=True)
```

Once ingestion is complete, features are stored at the S3 location we specified, as well as in a dedicated low-latency backend. Let's use the former to build a dataset.

Querying features to build a dataset

When we create the feature group, SageMaker automatically adds a new table for it in the **AWS Glue Data Catalog**. This makes it easy to use **Amazon Athena** to query data and build datasets on demand.

Let's say that we'd like to build a dataset that contains best-selling cameras with at least 1,000 reviews:

1. First, we write an SQL query that computes the average rating for each camera, counts how many reviews each camera received, only keeps cameras with at least 1,000 reviews, and orders cameras by descending average rating:

```
query_string =  
    'SELECT label,review_body FROM  
    "' + feature_group_table + '"'  
    + ' INNER JOIN (  
        SELECT product_id FROM (  
            SELECT product_id, avg(star_rating) as  
                avg_rating, count(*) as review_count  
            FROM "' + feature_group_table+ '" + '  
                GROUP BY product_id)  
        WHERE review_count > 1000) tmp
```

```
ON '''+feature_group_table+'''  
+ '.product_id=tmp.product_id;'
```

2. Then, we use Athena to query our feature group, store selected rows in a pandas dataframe, and display the first few rows:

```
dataset = pd.DataFrame()  
  
feature_group_query.run(query_string=query_string,  
output_location='s3://'+default_bucket+'/query_results/')  
feature_group_query.wait()  
dataset = feature_group_query.  
as_dataframe()  
dataset.head()
```

This prints out the table visible in the next image:

	label	review_body
0	__label__negative__	poor software pros : 1 . works out of the bo...
1	__label__positive__	fujifilm instax mini instant film happy with t...
2	__label__positive__	these are awesome . product came quickly and w...
3	__label__positive__	awesome clarity- awesome camera i was blown aw...
4	__label__positive__	great camera i was amazed at the quality of th...

Figure 10.12 – Viewing query results

From then on, it's business as usual. We can save this dataframe to a CSV file and use it to train models. You'll find an end-to-end example in the GitHub repository.

Exploring other capabilities of SageMaker Feature Store

Over time, we could store different versions of the same feature – that is, several records with the same identifier but with different timestamps. This would allow us to retrieve earlier versions of a dataset – "time traveling" in our data with a simple SQL query.

Last but not least, features are also available in the online store. We can retrieve individual records with the `GetRecord()` API and use features at prediction time whenever needed.

Again, you'll find code samples for both capabilities in the GitHub repository.

To close this chapter, let's look at Amazon SageMaker Clarify, a capability that helps us build higher quality models by detecting potential bias present in datasets and models.

Detecting bias in datasets and explaining predictions with SageMaker Clarify

A **machine learning** (ML) model is only as good as the dataset it was built from. If a dataset is inaccurate or unfair in representing the reality it's supposed to capture, a corresponding model is very likely to learn this biased representation and perpetuate it in its predictions. As ML practitioners, we need to be aware of these problems, understand how they impact predictions, and limit that impact whenever possible.

In this example, we'll work with the **Adult Data Set**, available at the **UCI Machine Learning Repository** (<http://archive.ics.uci.edu/ml>, Dua, D. and Graff, C., 2019). This dataset describes a binary classification task, where we try to predict if an individual earns less or more than \$50,000 per year. Here, we'd like to check whether this dataset introduces gender bias or not. In other words, does it help us build models that predict equally well for men and women?

Note

The dataset you'll find in the GitHub repository has been slightly processed. The label column has been moved to the front as per XGBoost requirements. Categorical variables have been one-hot encoded.

Configuring a bias analysis with SageMaker Clarify

SageMaker Clarify computes pre-training and post-training metrics that help us understand how a model predicts.

Post-training metrics obviously require a trained model, so we first train a binary classification model with XGBoost. It's nothing we haven't seen many times already, and you'll find the code in the GitHub repository. This model hits a validation AuC of 92.75%.

Once training is complete, we can proceed with the bias analysis:

1. Bias analyses run as SageMaker Processing jobs. Accordingly, we create a `SageMakerClarifyProcessor` object with our infrastructure requirements. As the job is small-scale, we use a single instance. For larger jobs, we could use an increased instance count, and the analysis would automatically run on **Spark**:

```
from sagemaker import clarify
```

```
clarify_processor = clarify.SageMakerClarifyProcessor(  
    role=role,  
    instance_count=1,  
    instance_type='ml.m5.large',  
    sagemaker_session=session)
```

2. Then, we create a DataConfig object describing the dataset to analyze:

```
bias_report_output_path = 's3://{{}/{}}/clarify-bias'.  
format(bucket, prefix)  
data_config = clarify.DataConfig(  
    s3_data_input_path=train_uri,  
    s3_output_path=bias_report_output_path,  
    label='Label',  
    headers=train_data.columns.to_list(),  
    dataset_type='text/csv')
```

3. Likewise, we create a ModelConfig object describing the model to analyze:

```
model_config = clarify.ModelConfig(  
    model_name=xgb_predictor.endpoint_name,  
    instance_type='ml.t2.medium',  
    instance_count=1,  
    accept_type='text/csv')
```

4. Finally, we create a BiasConfig object describing the metrics to compute. The label_values_or_threshold defines the label value for the positive outcome (1, indicating a revenue higher than \$50K). The facet_name defines the feature on which we'd like to run the analysis (Sex_), and facet_values_or_threshold defines the feature value for the potentially disadvantaged group (1, indicating women).

```
bias_config = clarify.BiasConfig(  
    label_values_or_threshold=[1],  
    facet_name='Sex_',  
    facet_values_or_threshold=[1])
```

We're now ready to run the analysis.

Running a bias analysis

Putting everything together, we launch the analysis with the following:

```
clarify_processor.run_bias(
    data_config=data_config,
    model_config=model_config,
    bias_config=bias_config)
```

Once the analysis is complete, the results are visible in SageMaker Studio. A report is also generated and stored in S3 in HTML, PDF, and notebook format.

In **Experiments and trials**, we locate our SageMaker Clarify job, and we right-click on **Open trial details**. Selecting **Bias report**, we see bias metrics, as shown in the next screenshot:

The screenshot shows the SageMaker Studio interface with the 'Bias report' tab selected. The top navigation bar includes 'Trial components: clarify-bias-2021-07-06-12-12-54-225-aws-process...', 'Charts', 'Metrics', 'Parameters', 'Artifacts', 'AWS settings', 'Debugger', 'Model explainability', and 'Bias report'. Below the navigation bar, it says 'The computed bias metrics are below:' and 'Predicted column: Label'. It also shows 'Predicted value or threshold: 1'. The main area displays a table of bias metrics with columns 'Bias metric', 'Bias value', and 'Description'.

Bias metric	Bias value	Description
Conditional Demographic Disparity in Labels (CDDL)	Unable to display	The metric examines whether, in the training data, the disadvantaged class has a bigger proportion of the rejected outcomes than the advantaged class.
Class Imbalance (CI)	0,35	Detects if the advantaged group is represented in the dataset at a substantially higher rate than the disadvantaged group.
Difference in Positive Proportions in Labels (DPL)	0,2	Detects if one class has a significantly higher proportion of desirable (or, alternatively, undesirable) outcomes than the other.
Jensen-Shannon Divergence (JS)	0,03	JS measures how much the label distributions of different classes diverge from each other. If the average label distribution is uniform, JS is zero.
Kullback-Liebler Divergence (KL)	0,14	In a binary case, a relative entropy measure of how much the label distribution of an advantaged class diverges from the label distribution of a disadvantaged class.
Kolmogorov-Smirnov Distance (KS)	0,2	This metric is equal to the maximum divergence in a label across the classes of a data set. It complements the JS metric.
L-p Norm (LP)	0,28	This measure of distance in label distributions is the normed direct distance between the distributions. We take the L1 norm for binary classification.
Total Variation Distance (TVD)	0,2	This measure of distance in label distributions is half the Hamming distance between the probability distributions.
Accuracy Difference (AD)	-0,09	This metric examines whether the classification by the model is more accurate for one class than the other.
Conditional Demographic Disparity in Predicted Labels (CDPDL)	Unable to display	The metric examines whether the model predicted a bigger proportion of rejected outcomes for the disadvantaged class than for the advantaged class.
Difference in Acceptance Rates (DAR)	-0,028	The difference in the rates of positive predicted outcomes across the advantaged and disadvantaged classes is measured.
Difference in Conditional Acceptance (DCA)	-0,11	This metric compares the actual labels to the predicted labels from the model and assesses whether this is the case.
Difference in Conditional Outcomes (DCR)	0,034	This metric compares the actual labels to the predicted labels from the model and assesses whether this is the case.
Disparate (Adverse) Impact (DI)	0,34	This metric examines whether the model predicts outcomes differently for each class. This is the ratio version of the DPL metric.
Difference in Positive Proportions in Predicted Labels (DPDL)	0,18	This metric examines whether the model predicts outcomes differently for each class. This is also known as p-value.
Difference in Rejection Rates (DRR)	0,085	The difference in the rates of negative predicted outcomes (rejections) across the advantaged and disadvantaged classes is measured.
Counterfactuals: Fliptest (FT)	-0,012	The fliptest is an approach that looks at each member of the disadvantaged class and assesses whether similar outcomes are predicted.
Recall Difference (RD)	0,036	Checks whether there is a difference in recall of the model across the attributes of interest. Recall is how often the model correctly identifies the target class.
Treatment Equality (TE)	0,84	This is defined as the difference in the ratio of false negatives to false positives for the advantaged vs. disadvantaged classes.

Figure 10.13 – Viewing bias metrics

Analyzing bias metrics

If you'd like to learn more about bias metrics, what they mean, and how they're computed, I highly recommend these resources:

- <https://pages.awscloud.com/rs/112-TZM-766/images/Fairness.Measures.for.Machine.Learning.in.Finance.pdf>
- <https://pages.awscloud.com/rs/112-TZM-766/images/Amazon.AI.Fairness.and.Explainability.Whitepaper.pdf>
- <https://github.com/aws/amazon-sagemaker-clarify>

Let's look at two pre-training metrics, **Class Imbalance (CI)** and **Difference in Positive Proportions in Labels (DPL)**, and one post-training metric, **Difference in Positive Proportions in Predicted Labels (DPPL)**.

A non-zero value of CI indicates that the dataset is imbalanced. Here, the difference between the men fraction and the women fraction is 0.35. Indeed, the men group is about two-thirds of the dataset, the women group is about one-third. This isn't a very severe imbalance, but we should also look at the proportion of positive labels for each class.

The DPL measures if each class has the same proportion of positive labels. In other words, does the dataset contain the same ratio of men and women earning \$50K? The DPL is non-zero (0.20), which tells us that men have a higher ratio of \$50K earners.

The DPPL is a post-training metric similar to the DPL. Its value (0.18) shows that the model unfortunately picked up the bias present in the dataset, only lightly reducing it. Indeed, the model predicts a more favorable outcome for men (over-predicting \$50K earners) and a less favorable outcome for women (under-predicting 50K earners).

That's clearly a problem. Although the model has a rather nice validation AuC (92.75%), it doesn't predict both classes equally well.

Before we dive into the data and try to mitigate this issue, let's run an explainability analysis.

Running an explainability analysis

SageMaker Clarify can compute local and global SHAP (<https://github.com/slundberg/shap>) values. They help us understand feature importance, and how individual feature values contribute to a positive or negative outcome.

Bias analyses run as SageMaker Processing jobs, and the process is similar:

1. We create a DataConfig object describing the dataset to analyze:

```
explainability_output_path = 's3://{}//{}//clarify-  
explainability.format(bucket, prefix)  
data_config = clarify.DataConfig(  
    s3_data_input_path=train_uri,  
    s3_output_path= explainability_output_path,  
    label='Label',  
    headers=train_data.columns.to_list(),  
    dataset_type='text/csv')
```

2. We create a SHAPConfig object describing how we'd like to compute SHAP values – that is, which baseline to use (I use the test set where I removed labels), how many samples to use (twice the number of features plus 2048, a common default), and how to aggregate values:

```
shap_config = clarify.SHAPConfig(  
    baseline=test_no_labels_uri,  
    num_samples=2*86+2048,  
    agg_method='mean_abs',  
    save_local_shap_values=True  
)
```

3. Finally, we run the analysis:

```
clarify_processor.run_explainability(  
    data_config=explainability_data_config,  
    model_config=model_config,  
    explainability_config=shap_config  
)
```

Results are available in SageMaker Studio, under **Experiments and trials / Open trial details / Model explainability**. As shown in the next image, the `Sex` feature is by far the most important, which confirms the bias analysis. Ethical considerations aside, this doesn't seem to make a lot of sense from a business perspective. Features such as education or capital gain should be more important.

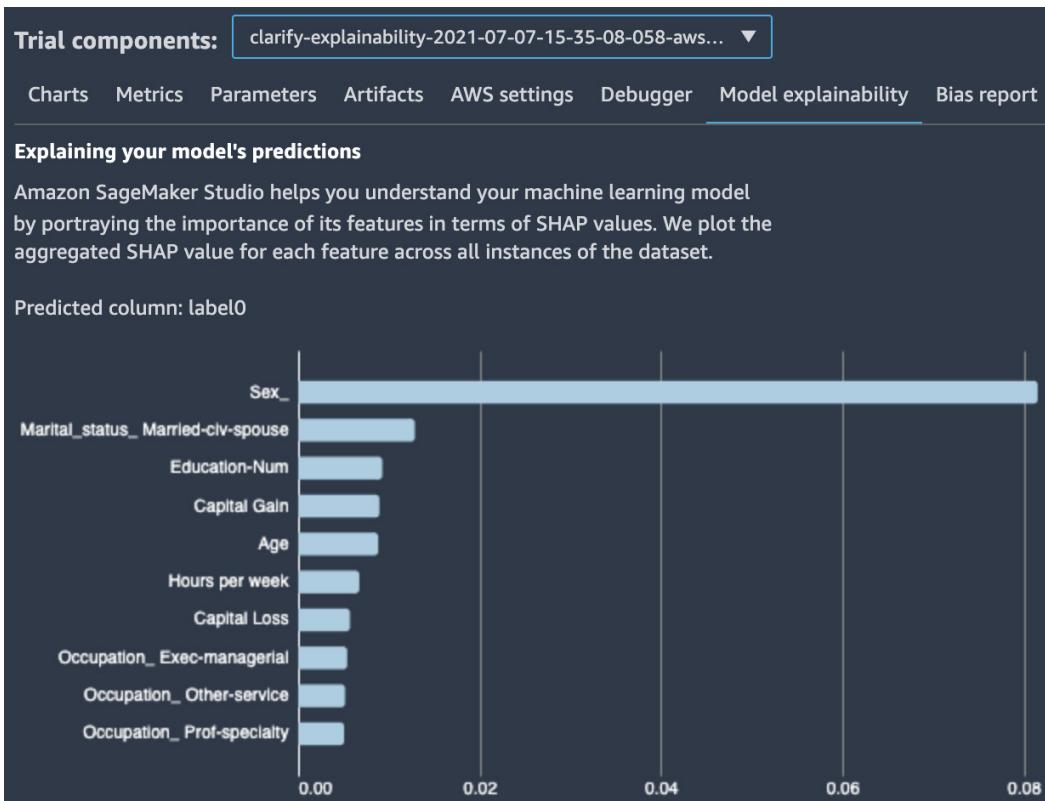


Figure 10.14 – Viewing feature importance

Local SHAP values have also been computed and stored in S3. We could use them to understand how feature values impact the prediction of each individual sample.

Now, let's see how we can try to mitigate the bias we detected in our dataset.

Mitigating bias

This dataset combines two problems. First, it contains more men than women. Second, the men group has a higher proportion of positive outcomes. The combination of these two problems leads to a situation where the dataset contains a disproportionately low number of women who earn more than \$50K. This makes it harder for the model to learn in a fair way, and it tends to favor the majority class.

Bias mitigation techniques include the following:

- Undersampling the majority class by removing majority samples to rebalance the dataset
- Oversampling the minority class by adding more samples through duplication of existing ones
- Adding synthetic samples to the minority class by generating new samples that have statistical properties similar to existing samples

Note

Altering data shouldn't be done lightly, especially in organizations operating in regulated industries. This can have serious business, compliance, and legal consequences. Please make sure to get approval before doing this in production.

Let's try a combined approach based on the **imbalanced-learn** open source library (<https://imbalanced-learn.org>). First, we'll add synthetic samples to the minority class with the **Synthetic Minority Oversampling Technique (SMOTE)** algorithm, in order to match the ratio of \$50K earners present in the majority samples. Then, we'll undersample the majority class to match the number of samples of the minority class. The result will be a perfectly balanced dataset, where both classes have the same size and the same ratio of \$50K earners. Let's get started:

1. First, we need to compute the ratios for both classes:

```
female_male_not_50k_count = train_data['Sex_'].where(
    train_data['Label']==0).value_counts()
female_male_50k_count = train_data['Sex_'].where(
    train_data['Label']==1).value_counts()
ratios = female_male_50k_count /
    female_male_not_50k_count
print(ratios)
```

This gives us the following result, showing that the majority class (class 0) has a much larger ratio of \$50k earners:

0.0	0.457002
1.0	0.128281

2. Then, we generate synthetic minority samples:

```
from imblearn.over_sampling import SMOTE
female_instances = train_data[train_data['Sex_']==1]
female_X = female_instances.drop(['Label'], axis=1)
female_Y = female_instances['Label']
oversample = SMOTE(sampling_strategy=ratios[0])
balanced_female_X, balanced_female_Y = oversample.fit_resample(female_X, female_Y)
balanced_female=pd.concat([balanced_female_X, balanced_female_Y], axis=1)
```

3. Next, we rebuild the dataset with the original majority class and the rebalanced minority class:

```
male_instances = train_data[train_data['Sex_']==0]
balanced_train_data=pd.concat(
    [male_instances, balanced_female], axis=0)
```

4. Finally, we undersample the original majority class to rebalance ratios:

```
from imblearn.under_sampling import RandomUnderSampler
X = balanced_train_data.drop(['Sex_'], axis=1)
Y = balanced_train_data['Sex_']
undersample = RandomUnderSampler(
    sampling_strategy='not minority')
X,Y = undersample.fit_resample(X, Y)
balanced_train_data=pd.concat([X, Y], axis=1)
```

5. We count both classes and compute their ratios again:

```
female_male_count= balanced_train_data['Sex_']
.fvalue_counts()
female_male_50k_count = balanced_train_data['Sex_']
.where(balanced_train_data['Label']==1)
.value_counts()
```

```
ratios = female_male_50k_count/female_male_count
print(female_male_count)
print(female_male_50k_count)
print(ratios)
```

This displays the following results:

1.0	0.313620
0.0	0.312039

Training with this rebalanced dataset, and using the same test set, we get a validation AUC of 92.95%, versus 92.75% for the original model. Running a new bias analysis, CI is zero, and the DPL and DPPL are close to zero.

Not only have we built a model that predicts more fairly, but it's also a little bit more accurate. For once, it looks like we got the best of both worlds!

Summary

This chapter concludes our exploration of training techniques. You learned about managed spot training, a simple way to slash training costs by 70% or more. You also saw how checkpointing helps to resume jobs that have been interrupted. Then, you learned about automatic model tuning, a great way to extract more accuracy from your models by exploring hyperparameter ranges. You learned about SageMaker Debugger, an advanced capability that automatically inspects training jobs for unwanted conditions and saves tensor collections to S3 for inspection and visualization. Finally, we discovered two capabilities that help you build higher quality workflows and models, SageMaker Feature Store and SageMaker Clarify.

In the next chapter, we'll study model deployment in detail.

Section 4: Managing Models in Production

In this section, you will learn how to deploy machine learning models in a variety of configurations, both with the SDK and with several automation tools. Finally, you will learn how to find the best cost/performance ratio for their prediction infrastructure.

This section comprises the following chapters:

- *Chapter 11, Deploying Machine Learning Models*
- *Chapter 12, Automating Machine Learning Workflows*
- *Chapter 13, Optimizing Cost and Performance*

11

Deploying Machine Learning Models

In previous chapters, we've deployed models in the simplest way possible: by configuring an estimator, calling the `fit()` **application programming interface (API)** to train the model, and calling the `deploy()` API to create a real-time endpoint. This is the simplest scenario for development and testing, but it's not the only one.

Models can be imported. For example, you could take an existing model that you trained on your local machine, import it into SageMaker, and deploy it as if you had trained it on SageMaker.

In addition, models can be deployed in different configurations, as follows:

- A single model on a real-time endpoint, which is what we've done so far, as well as several model variants in the same endpoint.
- A sequence of up to five models, called an **inference pipeline**.
- An arbitrary number of related models that are loaded on demand on the same endpoint, known as a **multi-model endpoint**. We'll examine this configuration in *Chapter 13, Optimizing Cost and Performance*.
- A single model or an inference pipeline that predicts data in batch mode through a feature known as **batch transform**.

Of course, models can also be exported. You can grab a training artifact in **Simple Storage Service (S3)**, extract the model, and deploy it anywhere you like.

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

- Examining model artifacts and exporting models
- Deploying models on real-time endpoints
- Deploying models on batch transformers
- Deploying models on inference pipelines
- Monitoring prediction quality with Amazon SageMaker Model Monitor
- Deploying models on container services
- Let's get started!

Technical requirements

You will need an **Amazon Web Services (AWS)** account to run the examples included in this chapter. If you haven't got one already, please browse to <https://aws.amazon.com/getting-started/> to create one. You should also familiarize yourself with the AWS Free Tier (<https://aws.amazon.com/free/>), which lets you use many AWS services for free within certain usage limits.

You will need to install and configure the AWS **Command Line Interface (CLI)** for your account (<https://aws.amazon.com/cli/>).

You will need a working Python 3.x environment. Installing the Anaconda distribution (<https://www.anaconda.com/>) is not mandatory but strongly encouraged as it includes many projects that we will need (Jupyter, pandas, numpy, and more).

The code examples included in this book are available on GitHub at <https://github.com/PacktPublishing/Learn-Amazon-SageMaker-second-edition>. You will need to install a Git client to access them (<https://git-scm.com/>).

Examining model artifacts and exporting models

A model artifact contains one or several files that are produced by a training job and that are required for model deployment. The number and nature of these files depend on the algorithm that was trained. As we've seen many times, the model artifact is stored as a `model.tar.gz` file, at the S3 output location defined in the estimator.

Let's look at different examples, where we reuse artifacts from the jobs we previously trained.

Examining and exporting built-in models

Almost all built-in algorithms are implemented with **Apache MXNet**, and their artifacts reflect this. For more information on MXNet, please visit <https://mxnet.apache.org/>.

Let's see how we can load these models directly. Another option would be to use **Multi Model Server (MMS)** (<https://github.com/awslabs/multi-model-server>), but we'll proceed as follows:

1. Let's start from the artifact for the **Linear Learner** model we trained in *Chapter 4, Training Machine Learning Models*, as illustrated in the following code snippet:

```
$ tar xvfz model.tar.gz
x model_algo-1
$ unzip model_algo-1
archive: model_algo-1
extracting: additional-params.json
extracting: manifest.json
extracting: mx-mod-symbol.json
extracting: mx-mod-0000.params
```

2. We load the symbol file, which contains a **JavaScript Object Notation (JSON)** definition of the model, as follows:

```
import json
sym_json = json.load(open('mx-mod-symbol.json'))
sym_json_string = json.dumps(sym_json)
```

3. We use this JSON definition to instantiate a new Gluon model. We also define the name of its input symbol (`data`), as follows:

```
import mxnet as mx
from mxnet import gluon
net = gluon.nn.SymbolBlock(
    outputs=mx.sym.load_json(sym_json_string),
    inputs=mx.sym.var('data'))
```

4. Now, we can easily plot the model, like this:

```
mx.viz.plot_network(
    net(mx.sym.var('data'))[0],
    node_attrs={'shape': 'oval', 'fixedsize': 'false'})
```

This creates the following output:

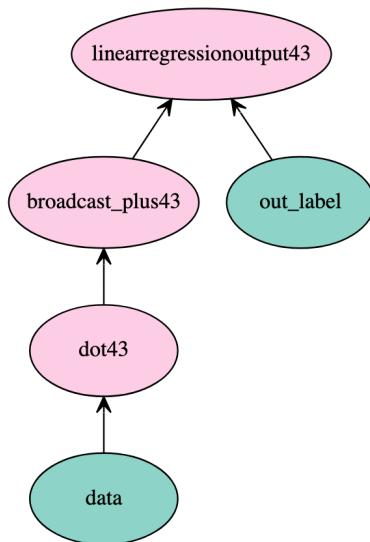


Figure 11.1 – Linear Learner model

5. Then, we load the model parameters learned during training, as follows:

```
net.load_parameters('mx-mod-0000.params',
                    allow_missing=True)
net.collect_params().initialize()
```

6. We define a test sample stored in an MXNet NDArray (<https://mxnet.apache.org/versions/1.6/api/python/docs/api/ndarray/index.html>), as follows:

```
test_sample = mx.nd.array(
[0.00632, 18.00, 2.310, 0, 0.5380, 6.5750, 65.20, 4.0900, 1, 296.0
, 15.30, 4.98])
```

7. Finally, we forward it through the model and read the output, as follows:

```
response = net(test_sample)
print(response)
```

The predicted price of this house is **US Dollars (USD)** 30,173, as illustrated here:

```
array([[30.173424]], dtype=float32)
```

This technique should work with all MXNet-based algorithms. Now, let's take a look at the built-in algorithms for **Computer Vision (CV)**.

Examining and exporting built-in CV models

The three built-in algorithms for CV are also based on Apache MXNet. The process is exactly the same, as outlined here:

1. The following is the artifact for the **image classification** model we trained on dogs and cats in *Chapter 5, Training Computer Vision Models*:

```
$ tar xvfz model.tar.gz
x image-classification-0010.params
x model-shapes.json
x image-classification-symbol.json
```

2. Load the model and its parameters, as follows:

```
import mxnet, json
from mxnet import gluon
sym_json = json.load(
    open('image-classification-symbol.json'))
```



```
sym_json_string = json.dumps(sym_json)
net = gluon.nn.SymbolBlock(
    outputs=mx.sym.load_json(sym_json_string),
```

```
inputs=mx.sym.var('data'))  
net.load_parameters(  
    'image-classification-0010.params',  
    allow_missing=True)  
net.collect_params().initialize()
```

3. The input shape is a 300x300 color image with three channels (**red, green, and blue**, or **RGB**). Accordingly, we create a fake image using random values. We forward it through the model and read the results, as follows:

```
test_sample = mx.ndarray.random.normal(  
    shape=(1, 3, 300, 300))  
response = net(test_sample)  
print(response)
```

Funnily enough, this random image is classified as a cat, as defined in the following code snippet:

```
array([[0.99126923, 0.00873081]], dtype=float32)
```

Reusing **Object Detection** is more complicated as the training network needs to be modified for prediction. You can find an example at <https://github.com/aws-samples/amazon-sagemaker-aws-greengrass-custom-object-detection-model/>.

Now, let's look at **Extreme Gradient Boosting (XGBoost)** artifacts.

Examining and exporting XGBoost models

An XGBoost artifact contains a single file—the model itself. However, the format of the model depends on how you're using XGBoost.

With the built-in algorithm, the model is a pickled file that stores a `Booster` object. Once the artifact has been extracted, we simply unpickle the model and load it, as follows:

```
$ tar xvfz model.tar.gz  
x xgboost-model  
$ python  
>>> import pickle
```

```
>>> model = pickle.load(open('xgboost-model', 'rb'))
>>> type(model)
<class 'xgboost.core.Booster'>
```

With the built-in framework, the model is just a saved model. Once the artifact has been extracted, we load the model directly, as follows:

```
$ tar xvfz model.tar.gz
x xgb.model
$ python
>>> import xgboost as xgb
>>> bst = xgb.Booster({'nthread': 4})
>>> model = bst.load_model('xgb.model')
>>> type(bst)
<class 'xgboost.core.Booster'>
```

Now, let's look at **scikit-learn** artifacts.

Examining and exporting scikit-learn models

Scikit-learn models are saved and loaded with `joblib` (<https://joblib.readthedocs.io>), as illustrated in the following code snippet. This library contains a set of tools that provide lightweight pipelining, but we'll only use it to save models:

```
$ tar xvfz model.tar.gz
x model.joblib
$ python
>>> import joblib
>>> model = joblib.load('model.joblib')
>>> type(model)
<class 'sklearn.linear_model._base.LinearRegression'>
```

Finally, let's look at **TensorFlow** artifacts.

Examining and exporting TensorFlow models

TensorFlow and **Keras** models are saved in **TensorFlow Serving** format, as illustrated in the following code snippet:

```
$ mkdir /tmp/models
$ tar xvfz model.tar.gz -C /tmp/models
x 1/
x 1/saved_model.pb
x 1/assets/
x 1/variables/
x 1/variables/variables.index
x 1/variables/variables.data-00000-of-00002
x 1/variables/variables.data-00001-of-00002
```

The easiest way to serve such a model is to run the **Docker** image for TensorFlow Serving, as illustrated in the following code snippet. You can find more details at https://www.tensorflow.org/tfx/serving/serving_basic:

```
$ docker run -t --rm -p 8501:8501
-v "/tmp/models:/models/fmnist"
-e MODEL_NAME=fmnist
tensorflow/serving
```

Let's look at a final example where we export a Hugging Face model.

Examining and exporting Hugging Face models

Hugging Face models can be trained on either TensorFlow or PyTorch. Let's reuse our Hugging Face example from *Chapter 7, Extending Machine Learning Services with Built-in Frameworks*, where we trained a sentiment analysis model with PyTorch, and proceed as follows:

1. We copy the model artifact from S3 and extract it, like this:

```
$ tar xvfz model.tar.gz
training_args.bin
config.json
pytorch_model.bin
```

2. In a Jupyter notebook, we use the Hugging Face API to load the model configuration. We then build the model using a `DistilBertForSequenceClassification` object, which corresponds to the model that we trained on SageMaker. Here's the code to accomplish this:

```
from transformers import AutoConfig,  
DistilBertForSequenceClassification  
config = AutoConfig.from_pretrained(  
    './model/config.json')  
model = DistilBertForSequenceClassification  
    .from_pretrained('./model/pytorch_model.bin',  
        config=config)
```

3. Next, we fetch the tokenizer associated with the model, as follows:

```
from transformers import AutoTokenizer  
tokenizer = AutoTokenizer.from_pretrained(  
    'distilbert-base-uncased')
```

4. We write a short function that will apply `softmax` to the activation values returned by the output layer of the model, as follows:

```
import torch  
def probs(logits):  
    softmax = torch.nn.Softmax(dim=1)  
    pred = softmax(logits).detach().numpy()  
    return pred
```

5. Finally, we define a sample and predict it with our model, as follows:

```
inputs = tokenizer("The Phantom Menace was a really bad  
movie. What a waste of my life.", return_tensors='pt')  
  
outputs = model(**inputs)  
print(probs(outputs.logits))
```

As expected, the sentiment is strongly negative, as we can see here:

```
[[0.22012234 0.7798777 ]]
```

This concludes the section on exporting models from SageMaker. As you can see, it's really not difficult at all.

Now, let's learn how to deploy models on real-time endpoints.

Deploying models on real-time endpoints

SageMaker endpoints serve real-time predictions using models hosted on fully managed infrastructure. They can be created and managed with either the SageMaker **software development kit (SDK)** or with an AWS SDK such as `boto3`.

You can find information on your endpoints in SageMaker Studio, under **SageMaker resources/Endpoints**.

Now, let's look at the SageMaker SDK in greater detail.

Managing endpoints with the SageMaker SDK

The SageMaker SDK lets you work with endpoints in several ways, as outlined here:

- Configuring an estimator, training it with `fit()`, deploying an endpoint with `deploy()`, and invoking it with `predict()`
- Importing and deploying a model
- Invoking an existing endpoint
- Updating an existing endpoint

We've used the first scenario in many examples so far. Let's look at the other ones.

Importing and deploying an XGBoost model

This is useful when you want to import a model that wasn't trained on SageMaker, or when you want to redeploy a SageMaker model. In the previous section, we saw what model artifacts look like, and how we should use them to package models. We'll now proceed as follows:

1. Starting from an XGBoost model that we trained and saved locally with `save_model()`, we first create a model artifact by running the following code:

```
$ tar cvfz model-xgb.tar.gz xgboost-model
```

2. In a Jupyter notebook, we upload the model artifact to our default bucket, like this:

```
import sagemaker  
sess = sagemaker.Session()  
prefix = 'export-xgboost'
```

```
model_path = sess.upload_data(  
    path='model-xgb.tar.gz',  
    key_prefix=prefix)
```

3. Then, we create an XGBoostModel object, passing the location of the artifact and an inference script (more on this in a second). We also select a framework version, and it should match the one we use to train the model. The code is illustrated in the following snippet:

```
from sagemaker.xgboost.model import XGBoostModel  
xgb_model = XGBoostModel(  
    model_data=model_path,  
    entry_point='xgb-script.py',  
    framework_version='1.3-1',  
    role=sagemaker.get_execution_role())
```

4. The inference script is very simple. It only needs to contain a model-loading function, as explained when we discussed deploying framework models in *Chapter 7, Extending Machine Learning Services with Built-in Frameworks*. The code is illustrated in the following snippet:

```
import os  
import xgboost as xgb  
def model_fn(model_dir):  
    model = xgb.Booster()  
    model.load_model(  
        os.path.join(model_dir, 'xgboost-model'))  
    return model
```

5. Back in the notebook, we then deploy and predict as usual, as follows:

```
xgb_predictor = xgb_model.deploy(...)  
xgb_predictor.predict(...)
```

Now, let's do the same with a TensorFlow model.

Importing and deploying a TensorFlow model

The process is very similar, as we will see next:

1. We first use `tar` to package a TensorFlow model that we trained and saved in TensorFlow Serving format. Our artifact should look like this (please don't forget to create the top-level directory!):

```
$ tar tvfz model.tar.gz
1/
1/saved_model.pb
1/assets/
1/variables/
1/variables/variables.index
1/variables/variables.data-00000-of-00002
1/variables/variables.data-00001-of-00002
```

2. Then, we upload the artifact to S3, as follows:

```
import sagemaker
sess = sagemaker.Session()
prefix = 'byo-tf'

model_path = sess.upload_data(
    path='model.tar.gz',
    key_prefix=prefix)
```

3. Next, we create a SageMaker model from the artifact. By default, we don't have to provide an inference script. We would pass if we needed custom preprocessing and postprocessing handlers for feature engineering, exotic serialization, and so on. You'll find more information at https://sagemaker.readthedocs.io/en/stable/frameworks/tensorflow/using_tf.html#deploying-from-an-estimator. The code is illustrated in the following snippet:

```
from sagemaker.tensorflow.model import TensorFlowModel
tf_model = TensorFlowModel(
    model_data=model_path,
    framework_version='2.3.1',
    role=sagemaker.get_execution_role())
```

4. We then deploy and predict as usual, thanks to the **Deep Learning Container (DLC)** for TensorFlow.

Now, let's do a final example, where we import and deploy a Hugging Face model with the DLC for PyTorch and an inference script for model loading and custom processing.

Importing and deploying a Hugging Face model with PyTorch

Let's reuse our Hugging Face example, and first focus on the inference script. It contains four functions: model loading, preprocessing, prediction, and postprocessing. We'll proceed as follows:

1. The model-loading function uses the same code that we used when we exported the model. The only difference is that we load the file from `model_dir`, which is passed by SageMaker to the PyTorch container. We also load the tokenizer once. The code is illustrated in the following snippet:

```
tokenizer = AutoTokenizer.from_pretrained(
    'distilbert-base-uncased')
def model_fn(model_dir):
    config_path='{} / config.json'.format(model_dir)
    model_path='{} / pytorch_model.bin'.format(model_dir)
    config=AutoConfig.from_pretrained(config_path)
    model= DistilBertForSequenceClassification
        .from_pretrained(model_path, config=config)
    return model
```

2. The preprocessing and postprocessing functions are simple. They only check for the correct content and accept types. You can see these in the following code snippet:

```
def input_fn(serialized_input_data,
            content_type=JSON_CONTENT_TYPE):
    if content_type == JSON_CONTENT_TYPE:
        input_data = json.loads(serialized_input_data)
        return input_data
    else:
        raise Exception('Unsupported input type: '
                        + content_type)
def output_fn(prediction_output,
              accept=JSON_CONTENT_TYPE):
```

```
    if accept == JSON_CONTENT_TYPE:
        return json.dumps(prediction_output), accept
    else:
        raise Exception('Unsupported output type: '
                        + accept)
```

- Finally, the prediction function tokenizes input data, predicts it, and returns the name of the most probable class, as follows:

```
CLASS_NAMES = ['negative', 'positive']
def predict_fn(input_data, model):
    inputs = tokenizer(input_data['text'],
                       return_tensors='pt')
    outputs = model(**inputs)
    logits = outputs.logits
    _, prediction = torch.max(logits, dim=1)
    return CLASS_NAMES[prediction]
```

Now our inference script is ready, let's move to a notebook, import the model, and deploy it, as follows:

- We create a PyTorchModel object, passing the location of the model artifact in S3 and the location of our inference script, as follows:

```
from sagemaker.pytorch import PyTorchModel
model = PyTorchModel(
    model_data=model_data_uri,
    role=sagemaker.get_execution_role(),
    entry_point='torchserve-predictor.py',
    source_dir='src',
    framework_version='1.6.0',
    py_version='py36')
```

- We deploy with `model.deploy()`. Then, we create two samples and send them to our endpoint, as follows:

```
positive_data = {'text': "This is a very nice camera, I'm
super happy with it."}
negative_data = {'text': "Terrible purchase, I want my
money back!"}
```

```
prediction = predictor.predict(positive_data)
print(prediction)
prediction = predictor.predict(negative_data)
print(prediction)
```

As expected, the outputs are positive and negative.

This concludes the section on importing and deploying models. Now, let's learn how to invoke an endpoint that has already been deployed.

Invoking an existing endpoint

This is useful when you want to work with a live endpoint but don't have access to the predictor. All we need to know is the endpoint's name. Proceed as follows:

1. Build a `TensorFlowPredictor` predictor for the endpoint we deployed in a previous example. Again, the object is framework-specific. The code is illustrated in the following snippet:

```
from sagemaker.tensorflow.model import
TensorFlowPredictor
another_predictor = TensorFlowPredictor(
    endpoint_name=tf_endpoint_name,
    serializer=sagemaker.serializers.JSONSerializer()
)
```

2. Then, predict it as usual, as follows:

```
another_predictor.predict(...)
```

Now, let's learn how to update endpoints.

Updating an existing endpoint

The `update_endpoint()` API lets you update the configuration of an endpoint in a non-disruptive fashion. The endpoint stays in service, and you can keep predicting with it.

Let's try this on our TensorFlow endpoint, as follows:

1. We set the instance count to 2 and update the endpoint, as follows:

```
another_predictor.update_endpoint(
    initial_instance_count=2,
    instance_type='ml.t2.medium')
```

2. The endpoint is immediately updated, as shown in the following screenshot.

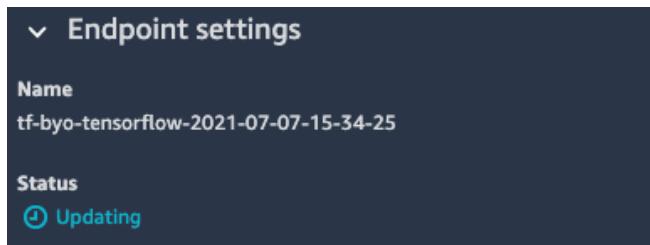


Figure 11.2 – Endpoint being updated

3. Once the update is complete, the endpoint is now backed by two instances, as shown in the following screenshot:

Endpoint runtime settings						
	Current weight	Desired weight	Instance type	Elastic inference	Current instance count	Desired instance count
	1	1	mL.t2.medium	-	2	2

Figure 11.3 – Endpoint backed by two instances

As you can see, it's very easy to import, deploy, redeploy, and update models with the SageMaker SDK. However, some operations require that we work with lower-level APIs. They're available in the AWS language SDKs, and we'll use our good friend `boto3` to demonstrate them.

Managing endpoints with the `boto3` SDK

`boto3` is the AWS SDK for Python (<https://aws.amazon.com/sdk-for-python/>). It includes APIs for all AWS services (unless they don't have APIs!). The SageMaker API is available at <https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/sagemaker.html>.

`boto3` APIs are service-level APIs, and they give us full control over all service operations. Let's see how they can help us deploy and manage endpoints in ways that the SageMaker SDK doesn't allow.

Deploying endpoints with the `boto3` SDK

Deploying an endpoint with `boto3` is a four-step operation, outlined as follows:

1. Create one or more models with the `create_model()` API. Alternatively, we could use existing models that have been trained or imported with the SageMaker SDK. For the sake of brevity, we'll do this here.

2. Define one or more **production variants**, listing the infrastructure requirements for each model.
3. Create an **endpoint configuration** with the `create_endpoint_config()` API, passing the production variants defined previously and assigning each one a weight.
4. Create an endpoint with the `create_endpoint()` API.

Let's put these APIs to work and deploy an endpoint running two variants of the XGBoost model we trained on the Boston Housing dataset, as follows:

1. We define two variants; both are backed by a single instance. However, they will receive nine-tenths and one-tenth of incoming requests, respectively—that is to say, "variant weight/sum of weights". We could use this setup if we wanted to introduce a new model in production and make sure it worked fine before sending it traffic. The code is illustrated in the following snippet:

```
production_variants = [  
    { 'VariantName': 'variant-1',  
      'ModelName': model_name_1,  
      'InitialInstanceCount': 1,  
      'InitialVariantWeight': 9,  
      'InstanceType': 'ml.t2.medium' },  
    { 'VariantName': 'variant-2',  
      'ModelName': model_name_2,  
      'InitialInstanceCount': 1,  
      'InitialVariantWeight': 1,  
      'InstanceType': 'ml.t2.medium' }]
```

2. We create an endpoint configuration by passing our two variants and setting optional tags, as follows:

```
import boto3  
sm = boto3.client('sagemaker')  
endpoint_config_name = 'xgboost-two-models-epc'  
response = sm.create_endpoint_config(  
    EndpointConfigName=endpoint_config_name,  
    ProductionVariants=production_variants,  
    Tags=[{ 'Key': 'Name',  
           'Value': endpoint_config_name},  
          { 'Key': 'Algorithm', 'Value': 'xgboost' }])
```

We can list all endpoint configurations with `list_endpoint_configs()` and describe a particular one with the `describe_endpoint_config()` boto3 APIs.

3. We create an endpoint based on this configuration:

```
endpoint_name = 'xgboost-two-models-ep'
response = sm.create_endpoint(
    EndpointName=endpoint_name,
    EndpointConfigName=endpoint_config_name,
    Tags=[{'Key': 'Name', 'Value': endpoint_name},
          {'Key': 'Algorithm', 'Value': 'xgboost'},
          {'Key': 'Environment',
           'Value': 'development'}])
```

We can list all the endpoints with `list_endpoints()` and describe a particular one with the `describe_endpoint()` boto3 APIs.

4. Creating a boto3 waiter is a handy way to wait for the endpoint to be in service. You can see one being created here:

```
waiter = sm.get_waiter('endpoint_in_service')
waiter.wait(EndpointName=endpoint_name)
```

5. After a few minutes, the endpoint is in service. As shown in the following screenshot, it now uses two production variants:

Endpoint runtime settings					
Variant name	Current weight	Desired weight	Instance type	Elastic inference	Current instance count
variant-1	9	9	ml.t2.medium	-	1
variant-2	1	1	ml.t2.medium	-	1

Figure 11.4 – Viewing production variants

6. Then, we invoke the endpoint, as shown in the following code snippet. By default, prediction requests are forwarded to variants according to their weights:

```
smrt = boto3.Session().client(
    service_name='runtime.sagemaker')
response = smrt.invoke_endpoint(
    EndpointName=endpoint_name,
    ContentType='text/csv',
    Body=test_sample)
```

7. We can also select the variant that receives the prediction request. This is useful for A/B testing, where we need to stick users to a given model. The following code snippet shows you how to do this:

```
variants = ['variant-1', 'variant-2']
for v in variants:
    response = smrt.invoke_endpoint(
        EndpointName=endpoint_name,
        ContentType='text/csv',
        Body=test_sample,
        TargetVariant=v)
    print(response['Body'].read())
```

This results in the following output:

```
b'[0.0013231043703854084]'
b'[0.001262241625227034]'
```

8. We can also update weights—for example, give equal weights to both variants so that they receive the same share of incoming traffic—as follows:

```
response = sm.update_endpoint_weights_and_capacities(
    EndpointName=endpoint_name,
    DesiredWeightsAndCapacities=[
        { 'VariantName': 'variant-1',
          'DesiredWeight': 5},
        { 'VariantName': 'variant-2',
          'DesiredWeight': 5}])
```

9. We can remove one variant entirely and send all traffic to the remaining one. Here too, the endpoint stays in service the whole time, and no traffic is lost. The code is illustrated in the following snippet:

```
production_variants_2 = [
    {'VariantName': 'variant-2',
     'ModelName': model_name_2,
     'InitialInstanceCount': 1,
     'InitialVariantWeight': 1,
     'InstanceType': 'ml.t2.medium'}]
endpoint_config_name_2 = 'xgboost-one-model-epc'
response = sm.create_endpoint_config(
```

```
EndpointConfigName=endpoint_config_name_2,  
ProductionVariants=production_variants_2,  
Tags=[{'Key': 'Name',  
       'Value': endpoint_config_name_2},  
      {'Key': 'Algorithm', 'Value': 'xgboost'}])  
response = sm.update_endpoint(  
    EndpointName=endpoint_name,  
    EndpointConfigName=endpoint_config_name_2)
```

10. Finally, we clean up by deleting the endpoint and the two endpoint configurations, as follows:

```
sm.delete_endpoint(EndpointName=endpoint_name)  
sm.delete_endpoint_config(  
    EndpointConfigName=endpoint_config_name)  
sm.delete_endpoint_config(  
    EndpointConfigName=endpoint_config_name_2)
```

As you can see, the boto3 API is more verbose, but it also gives us the flexibility we need for **machine learning (ML)** operations. In the next chapter, we'll learn how to automate these.

Deploying models on batch transformers

Some use cases don't require a real-time endpoint. For example, you may want to predict **10 gigabytes (GB)** of data once a week in one go, get the results, and feed them to a downstream application. Batch transformers are a very simple way to get this done.

In this example, we will use the scikit-learn script that we trained on the Boston Housing dataset in *Chapter 7, Extending Machine Learning Services with Built-in Frameworks*. Let's get started, as follows:

1. Configure the estimator as usual, by running the following code:

```
from sagemaker.sklearn import SKLearn  
sk = SKLearn(entry_point='sklearn-boston-housing.py',  
             role=sagemaker.get_execution_role(),  
             instance_count=1,  
             instance_type='ml.m5.large',  
             output_path=output,
```

```
hyperparameters=
    {'normalize': True, 'test-size': 0.1})
sk.fit({'training':training})
```

- Let's predict the training set in batch mode. We remove the target value, save the dataset to a **comma-separated values (CSV)** file, and upload it to S3, as follows:

```
import pandas as pd
data = pd.read_csv('housing.csv')
data.drop(['medv'], axis=1, inplace=True)
data.to_csv('data.csv', header=False, index=False)
batch_input = sess.upload_data(
    path='data.csv',
    key_prefix=prefix + '/batch')
```

- Create a transformer object and launch batch processing, as follows:

```
sk_transformer = sk.transformer(
    instance_count=1,
    instance_type='ml.m5.large')
sk_transformer.transform(
    batch_input,
    content_type='text/csv',
    wait=True, logs=True)
```

- In the training log, we can see that SageMaker creates a temporary endpoint and uses it to predict data. For large-scale jobs, we could optimize throughput by mini-batching samples for prediction (using the `strategy` parameter), increase the level of prediction concurrency (`max_concurrent_transforms`), and increase the maximum payload size (`max_payload`).
- Once the job is complete, predictions are available in S3, as indicated here:

```
print(sk_transformer.output_path)
s3://sagemaker-us-east-1-123456789012/sagemaker-scikit-
learn-2020-06-12-08-28-30-978
```

- Using the AWS CLI, we can easily retrieve these predictions by running the following code:

```
%%bash -s "$sk_transformer.output_path"
aws s3 cp $1/data.csv.out .
```

```
head -1 data.csv.out
[[29.73828574177013], [24.920634119498292], ...]
```

7. Just as for training, the infrastructure used by the transformer is shut down as soon as the job completes, so there's nothing to clean up.

In the next section, we will look at inference pipelines and how to use them to deploy a sequence of related models.

Deploying models on inference pipelines

Real-life ML scenarios often involve more than one model; for example, you may need to run preprocessing steps on incoming data or reduce its dimensionality with the **Principal Component Analysis (PCA)** algorithm.

Of course, you could deploy each model to a dedicated endpoint. However, orchestration code would be required to pass prediction requests to each model in sequence. Multiplying endpoints would also introduce additional costs.

Instead, **inference pipelines** let you deploy up to five models on the same endpoint or for batch transform and automatically handle the prediction sequence.

Let's say that we wanted to run PCA and then Linear Learner. Building the inference pipeline would look like this:

1. Train the PCA model on the input dataset.
2. Process the training and validation sets with PCA and store the results in S3. batch transform is a good way to do this.
3. Train the Linear Learner model using the datasets processed by PCA as input.
4. Use the `create_model()` API to create an inference pipeline, as follows:

```
response = sagemaker.create_model(
    ModelName='pca-linearlearner-pipeline',
    Containers=[
        {
            'Image': pca_container,
            'ModelDataURL': pca_model_artifact,
            ...
        },
        {
            'Image': ll_container,
```

```
'ModelDataUrl': ll_model_artifact,  
    . . .  
}  
,  
ExecutionRoleArn=role  
)
```

5. Create an endpoint configuration and an endpoint in the usual way. We could also use the pipeline with a batch transformer.

You can find a complete example that uses scikit-learn and Linear Learner at https://github.com/awslabs/amazon-sagemaker-examples/tree/master/sagemaker-python-sdk/scikit_learn_inference_pipeline.

Spark is a very popular choice for data processing, and SageMaker lets you deploy Spark models with the **SparkML Serving** built-in container (<https://github.com/aws/sagemaker-sparkml-serving-container>), which uses the **mleap** library (<https://github.com/combust/mleap>). Of course, these models can be part of an **inference pipeline**. You can find several examples at https://github.com/awslabs/amazon-sagemaker-examples/tree/master/advanced_functionality.

This concludes our discussion on model deployment. In the next section, we'll introduce a SageMaker capability that helps us detect data issues that impact prediction quality: **SageMaker Model Monitor**.

Monitoring prediction quality with Amazon SageMaker Model Monitor

SageMaker Model Monitor has two main features, outlined here:

- Capturing data sent to an endpoint, as well as predictions returned by the endpoint. This is useful for further analysis, or to replay real-life traffic during the development and testing of new models.
- Comparing incoming traffic to a baseline built from the training set, as well as sending alerts about data quality issues, such as missing features, mistyped features, and differences in statistical properties (also known as "data drift").

We'll use the **Linear Learner** example from *Chapter 4, Training Machine Learning Models*, where we trained a model on the Boston Housing dataset. First, we'll add data capture to the endpoint. Then, we'll build a **baseline** and set up a **monitoring schedule** to periodically compare the incoming data to that baseline.

Capturing data

We can set up the data-capture process when we deploy an endpoint. We can also enable it on an existing endpoint with the `update_endpoint()` API that we just used with production variants.

At the time of writing, there are certain caveats that you should be aware of, as outlined here:

- You can only send **one sample at a time** if you want to perform model monitoring. Mini-batch predictions will be captured, but they will cause the monitoring job to fail.
- Likewise, data samples and predictions must be **flat, tabular data**. Structured data (such as lists of lists and nested JSON) will be captured, but the model-monitoring job will fail to process it. Optionally, you can add a preprocessing script and a postprocessing script to flatten it. You can find more information at <https://docs.aws.amazon.com/sagemaker/latest/dg/model-monitor-pre-and-post-processing.html>.
- The content type and the accept type must be **identical**. You can use either CSV or JSON, but you can't mix them.
- You cannot delete an endpoint if it has a monitoring schedule attached to it. You have to **delete the monitoring schedule first**, then delete the endpoint.

Knowing that, let's capture some data! Here we go:

1. Training takes place as usual. You can find the code in the GitHub repository.
2. We create a data-capture configuration for 100% of the prediction requests and responses, storing everything in S3, as follows:

```
from sagemaker.model_monitor.data_capture_config import  
DataCaptureConfig  
  
capture_path = 's3://{{}}/{{}}/capture/'.format(bucket,  
prefix)  
  
ll_predictor = ll.deploy(  
    initial_instance_count=1,  
    instance_type='ml.t2.medium',
```

```
data_capture_config = DataCaptureConfig(  
    enable_capture = True,  
    sampling_percentage = 100,  
    capture_options = ['REQUEST', 'RESPONSE'],  
    destination_s3_uri = capture_path))
```

- Once the endpoint is in service, we send data for prediction. Within a minute or two, we see captured data in S3 and then copy it locally, as follows:

```
%%bash -s "$capture_path"  
aws s3 ls --recursive $1  
aws s3 cp --recursive $1 .
```

- Opening one of the files, we see samples and predictions, as follows:

```
{"captureData": {"endpointInput": {"observedContentType": "text/csv", "mode": "INPUT", "data": "0.00632,18.00,2.310,0,0.5380,6.5750,65.20,4.0900,1,296.0,15.30,4.98", "encoding": "CSV"}, "endpointOutput": {"observedContentType": "text/csv; charset=utf-8", "mode": "OUTPUT", "data": "30.4133586884", "encoding": "CSV"}}, "eventMetadata": {"eventId": "8f45e35c-fa44-40d2-8ed3-1bcab3a596f3", "inferenceTime": "2020-07-30T13:36:30Z"}, "eventVersion": "0"}
```

If this were live data, we could use it to test new models later on in order to compare their performance to existing models.

Now, let's learn how to create a baseline from the training set.

Creating a baseline

SageMaker Model Monitor includes a built-in container we can use to build the baseline, and we can use it directly with the `DefaultModelMonitor` object. You can also bring your own container, in which case you would use the `ModelMonitor` object instead. Let's get started, as follows:

- A baseline can only be built on CSV datasets and JSON datasets. Our dataset is space-separated and needs to be converted into a CSV file, as follows. We can then upload it to S3:

```
data.to_csv('housing.csv', sep=' ', index=False)  
training = sess.upload_data(
```

```
    path='housing.csv',
    key_prefix=prefix + "/baseline")
```

Note

There is a small caveat here: the baselining job is a Spark job running in **SageMaker Processing**. Hence, column names need to be Spark-compliant, or your job will fail in cryptic ways. In particular, dots are not allowed in column names. We don't have that problem here, but please keep this in mind.

2. Define the infrastructure requirements, the location of the training set, and its format, as follows:

```
from sagemaker.model_monitor import DefaultModelMonitor
from sagemaker.model_monitor.dataset_format import
DatasetFormat
ll_monitor = DefaultModelMonitor(role=role,
    instance_count=1, instance_type='ml.m5.large')
ll_monitor.suggest_baseline(baseline_dataset=training,
    dataset_format=DatasetFormat.csv(header=True))
```

3. As you can guess, this is running as a SageMaker Processing job, and you can find its log in **CloudWatch Logs** under the /aws/sagemaker/ProcessingJobs prefix.

Two JSON artifacts are available at its output location: `statistics.json` and `constraints.json`. We can view their content with pandas by running the following code:

```
baseline = ll_monitor.latest_baselining_job
constraints = pd.io.json.json_normalize(
    baseline.suggested_constraints()
    .body_dict["features"])
schema = pd.io.json.json_normalize(
    baseline.baseline_statistics().body_dict["features"])
```

4. As shown in the following screenshot, the constraints file gives us the inferred type of each feature, its completeness in the dataset, and whether it contains negative values or not:

	name	inferred_type	completeness	num_constraints.is_non_negative
0	crim	Fractional	1.0	True
1	zn	Fractional	1.0	True
2	indus	Fractional	1.0	True
3	chas	Integral	1.0	True
4	nox	Fractional	1.0	True
5	age	Fractional	1.0	True

Figure 11.5 – Viewing the inferred schema

5. The statistics file adds basic statistics, as shown in the following screenshot:

	name	inferred_type	numerical_statistics.common.num_present	numerical_statistics.common.num_missing	numerical_statistics.mean	numerical_statistics.sum	numerical_statistics.std_dev
0	crim	Fractional	506	0	3.613524	1828.44292	8.593041
1	zn	Fractional	506	0	11.363636	5750.00000	23.299396
2	indus	Fractional	506	0	11.136779	5635.21000	6.853571
3	chas	Integral	506	0	0.069170	35.00000	0.253743
4	nox	Fractional	506	0	0.554695	280.67570	0.115763
5	age	Fractional	506	0	6.284634	3180.02500	0.701923

Figure 11.6 – Viewing data statistics

It also includes distribution information based on KLL sketches (<https://arxiv.org/abs/1603.05346v2>), a compact way to define quantiles.

Once a baseline has been created, we can set up a monitoring schedule in order to compare incoming traffic to the baseline.

Setting up a monitoring schedule

We simply pass the name of the endpoint, the statistics, the constraints, and the frequency at which the analysis should run. We will go for hourly, which is the shortest frequency allowed. The code is illustrated in the following snippet:

```
from sagemaker.model_monitor import CronExpressionGenerator
ll_monitor.create_monitoring_schedule(
    monitor_schedule_name='ll-housing-schedule',
    endpoint_input=ll_predictor.endpoint,
    statistics=ll_monitor.baseline_statistics(),
```

```
constraints=ll_monitor.suggested_constraints(),
schedule_cron_expression=CronExpressionGenerator.hourly())
```

Here, the analysis will be performed by a built-in container. Optionally, we could provide our own container with bespoke analysis code. You can find more information at <https://docs.aws.amazon.com/sagemaker/latest/dg/model-monitor-byoc-containers.html>.

Now, let's send some nasty data to the endpoint and see if SageMaker Model Monitor picks it up.

Sending bad data

Unfortunately, a model may receive incorrect data at times. Maybe it's been corrupted at the source, maybe the application in charge of invoking the endpoint is buggy, and so on. Let's simulate this and see how much impact this has on the quality of the prediction, as follows:

1. Starting from a valid sample, we get a correct prediction, as illustrated here:

```
test_sample = '0.00632,18.00,2.310,0,0.5380,6.5750,65.20,
4.0900,1,296.0,15.30,4.98'
ll_predictor.serializer =
    sagemaker.serializers.CSVSerializer()
ll_predictor.deserializer =
    sagemaker.deserializers.CSVDeserializer()
response = ll_predictor.predict(test_sample)
print(response)
```

The price of this house is USD 30,173:

```
[[30.1734218597]]
```

2. Now, let's multiply the first feature by 10,000, as shown in the following code snippet. Scaling and unit errors are quite frequent in application code:

```
bad_sample_1 = '632.0,18.00,2.310,0,0.5380,6.5750,65.20,4
.0900,1,296.0,15.30,4.98'
response = ll_predictor.predict(bad_sample_1)
print(response)
```

Ouch! The price is negative, as we can see here. Clearly, this is a bogus prediction:

```
[[-35.7245635986]]
```

- Let's try negating the last feature by running the following code:

```
bad_sample_2 = '0.00632,18.00,2.310,0,0.5380,6.5750,65.2
0,
4.0900,1,296.0,15.30,-4.98'
response = ll_predictor.predict(bad_sample_2)
print(response)
```

The prediction is much higher than what it should be, as we can see in the following snippet. This is a sneakier issue, which means it is harder to detect and could have serious business consequences:

```
[34.4245414734]]
```

You should try experimenting with bad data and see which features are the most brittle. All this traffic will be captured by SageMaker Model Monitor. Once the monitoring job has run, you should see entries in its **violation report**.

Examining violation reports

Previously, we created an hourly monitoring job. Don't worry if it takes a little more than 1 hour to see results; job execution is load-balanced by the backend, and short delays are likely:

- We can find more information about our monitoring job in the SageMaker console, in the **Processing jobs** section. We can also call the `describe_schedule()` API and list executions with the `list_executions()` API, as follows:

```
ll_executions = ll_monitor.list_executions()
print(ll_executions)
```

Here, we can see three executions:

```
[<sagemaker.model_monitor.model_monitoring.
MonitoringExecution at 0x7fd1d55a6d8>,
<sagemaker.model_monitor.model_monitoring.
MonitoringExecution at 0x7fd1d581630>,
<sagemaker.model_monitor.model_monitoring.
MonitoringExecution at 0x7fdce4b1c860>]
```

2. The violations report is stored as a JSON file in S3. We can read it and display it with pandas, as follows:

```
violations = ll_monitor.latest_monitoring_constraint_
violations()

violations = pd.io.json.json_normalize(
    violations.body_dict["violations"])

violations
```

This prints out the violations that were detected by the last monitoring job, as shown in the following screenshot:

	feature_name	constraint_check_type	description
0	tax	data_type_check	Data type match requirement is not met. Expect...
1	nox	data_type_check	Data type match requirement is not met. Expect...
2	rad	data_type_check	Data type match requirement is not met. Expect...
3	chas	data_type_check	Data type match requirement is not met. Expect...

Figure 11.7 – Viewing violations

3. Of course, we can also fetch the file in S3 and display its contents, as follows:

```
%%bash -s "$report_path"
echo $1
aws s3 ls --recursive $1
aws s3 cp --recursive $1 .
```

Here's a sample entry, warning us that the model received a fractional value for the chas feature, although it's defined as an integer in the schema:

```
{
    "feature_name" : "chas",
    "constraint_check_type" : "data_type_check",
    "description" : "Data type match requirement is not
met.

        Expected data type: Integral, Expected match:
100.0%.
        Observed: Only 0.0% of data is Integral."
}
```

We could also emit these violations to CloudWatch metrics and trigger alarms to notify developers of potential data-quality issues. You can find more information at <https://docs.aws.amazon.com/sagemaker/latest/dg/model-monitor-interpreting-cloudwatch.html>.

4. When you're done, don't forget to delete the monitoring schedule and the endpoint itself, as follows:

```
response = ll_monitor.delete_monitoring_schedule()  
ll_predictor.delete_endpoint()
```

As you can see, SageMaker Model Monitor helps you capture both incoming data and predictions, a useful feature for model testing. In addition, you can also perform data-quality analysis using a built-in container or your own.

In the next section, we're going to move away from endpoints and learn how to deploy models to container services.

Deploying models to container services

Previously, we saw how to fetch a model artifact in S3 and how to extract the actual model from it. Knowing this, it's pretty easy to deploy it on a container service, such as **Amazon Elastic Container Service (ECS)**, **Amazon Elastic Kubernetes Service (EKS)**, or **Amazon Fargate**.

Maybe it's company policy to deploy everything in containers, maybe you just like them, or maybe both! Whatever the reason is, you can definitely do it. There's nothing specific to SageMaker here, and the AWS documentation for these services will tell you everything you need to know.

A sample high-level process could look like this:

1. Train a model on SageMaker.
2. When training is complete, grab the artifact and extract the model.
3. Push the model to a Git repository.
4. Write a task definition (for ECS and Fargate) or a pod definition (for EKS). It could use one of the built-in containers or your own. Then, it could run a model server or your own code to clone the model from your Git repository, load it, and serve predictions.
5. Using this definition, run a container on your cluster.

Let's apply this to Amazon Fargate.

Training on SageMaker and deploying on Amazon Fargate

Amazon Fargate lets you run containers on fully managed infrastructure (<https://aws.amazon.com/fargate>). There's no need to create and manage clusters, which makes it ideal for users who don't want to get involved with infrastructure details. However, please note that, at the time of writing, Fargate doesn't support **graphics processing unit (GPU)** containers.

Preparing a model

We prepare the model using the following steps:

1. First, we train a TensorFlow model on Fashion-MNIST. Business as usual.
2. We find the location of the model artifact in S3 and set it as an environment variable, as follows:

```
%env model_data {tf_estimator.model_data}
```

3. We download the artifact from S3 and extract it to a local directory, like this:

```
%%sh
aws s3 cp ${model_data} .
mkdir test-models
tar xvfz model.tar.gz -C test-models
```

4. We open a terminal and commit the model to a public Git repository, as illustrated in the following code snippet. I'm using one of mine here (<https://gitlab.com/juliensimon/test-models>); you should replace it with yours:

```
<initialize git repository>
$ cd test-models
$ git add model
$ git commit -m "New model"
$ git push
```

Configuring Fargate

Now that the model is available in a repository, we need to configure Fargate. We'll use the command line this time. You could do the same with boto3 or any other language SDK. We'll proceed as follows:

1. `ecs-cli` is a convenient CLI tool used to manage clusters. Let's install it by running the following code:

```
%%sh
sudo curl -o /usr/local/bin/ecs-cli https://amazon-ecs-
cli.s3.amazonaws.com/ecs-cli-linux-amd64-latest
sudo chmod 755 /usr/local/bin/ecs-cli
```

2. We use it to "create" a Fargate cluster. In practice, this isn't creating any infrastructure; we're only defining a cluster name that we'll use to run tasks. Please make sure that your **Identity and Access Management (IAM)** role includes the required permission for `ecs:CreateCluster`. If not, please add it before continuing. The code is illustrated in the following snippet:

```
%%sh
aws ecs create-cluster --cluster-name fargate-demo
ecs-cli configure --cluster fargate-demo --region
eu-west-1
```

3. We create a log group in **CloudWatch** where our container will write its output. We only need to do this once. Here's the code to accomplish this:

```
%%sh
aws logs create-log-group --log-group-name awslogs-tf-ecs
```

- We will need a **security group** for our task that opens the two inbound TensorFlow Serving ports (8500 for Google remote procedure call (**gRPC**); 8501 for the **REpresentational State Transfer (REST)** API). If you don't have one already, you can easily create one in the **Elastic Compute Cloud (EC2)** console. Here, I created one in my default **virtual private cloud (VPC)**. It looks like this:

The screenshot shows the AWS EC2 Security Groups page. At the top, there's a breadcrumb navigation: EC2 > Security Groups > sg-0504d9aef33f34caf - ssh-tfserving. Below the breadcrumb, the security group name is displayed as "sg-0504d9aef33f34caf - ssh-tfserving". There are two buttons at the top right: "Delete security group" and "Copy to new security group".

Details

Security group name ssh-tfserving	Security group ID sg-0504d9aef33f34caf	Description SSH + Tensorflow Serving	VPC ID vpc-def884bb
Owner [REDACTED]	Inbound rules count 2 Permission entries	Outbound rules count 1 Permission entry	

Below the details, there are three tabs: "Inbound rules" (which is selected), "Outbound rules", and "Tags".

Inbound rules

Type	Protocol	Port range	Source	Description - optional
Custom TCP	TCP	8500 - 8501	0.0.0.0/0	-
Custom TCP	TCP	8500 - 8501	::/0	-

Figure 11.8 – Viewing the security group

Defining a task

Now, we need to write a **JSON** file containing a **task definition**: the container image to use, its entry point, and its system and network properties. Let's get started, as follows:

- First, we define the amount of **central processing unit (CPU)** and memory that the task is allowed to consume. Unlike ECS and EKS, Fargate only allows a limited set of values, available at <https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task-cpu-memory-error.html>. We will go for 4 **virtual CPUs (vCPUs)** and 8 GB of **random-access memory (RAM)**, as illustrated in the following code snippet:

```
{
  "requiresCompatibilities": ["FARGATE"],
  "family": "inference-fargate-tf-230",
  "memory": "8192",
  "cpu": "4096",
```

2. Next, we define a container that will load our model and run predictions. We will use the DLC for TensorFlow 2.3.0. You can find a full list at https://github.com/aws/deep-learning-containers/blob/master/available_images.md. The code is illustrated in the following snippet:

```
"containerDefinitions": [ {  
    "name": "dlc-tf-inference",  
    "image": "763104351884.dkr.ecr.us-east-1.amazonaws.  
com/tensorflow-inference:2.3.2-cpu-py37-ubuntu18.04",  
    "essential": true,  
}
```

3. Its entry point creates a directory, clones the repository where we pushed the model, and launches TensorFlow Serving, as follows:

```
"command": [  
    "mkdir -p /test && cd /test && git clone https://  
gitlab.com/juliensimon/test-models.git && tensorflow_  
model_server --port=8500  
    --rest_api_port=8501 --model_name=1  
    --model_base_path=/test/test-models/model"  
],  
"entryPoint": ["sh", "-c"],
```

4. Accordingly, we map the two TensorFlow Serving ports, like this:

```
"portMappings": [  
    {  
        "hostPort": 8500,  
        "protocol": "tcp",  
        "containerPort": 8500  
    },  
    {  
        "hostPort": 8501,  
        "protocol": "tcp",  
        "containerPort": 8501  
    }  
,
```

5. We define the log configuration that's pointing at the CloudWatch log group we created earlier, as follows:

```
"logConfiguration": {  
    "logDriver": "awslogs",  
    "options": {  
        "awslogs-group": "awslogs-tf-ecs",  
        "awslogs-region": "eu-west-1",  
        "awslogs-stream-prefix": "inference"  
    }  
},  
}] ,
```

6. We set the networking mode for the container, as illustrated in the following code snippet. awsvpc is the most flexible option, and it will allow our container to be publicly accessible, as explained at <https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task-networking.html>. It will create an **elastic network interface** in the subnet of our choice:

```
"networkMode": "awsvpc"
```

7. Finally, we define an IAM role for the task. If this is the first time you're working with ECS, you should create this role in the IAM console. You can find instructions for this at https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task_execution_IAM_role.html. The code is illustrated in the following snippet:

```
"executionRoleArn":  
"arn:aws:iam::123456789012:role/ecsTaskExecutionRole"  
}
```

Running a task

We're now ready to run our task using the security group we created earlier and one of the subnets in our default VPC. Let's get started, as follows:

1. We launch the task with the `run-task` API, passing the family name of the task definition (not the filename!). Please pay attention to the version number as well as it will automatically increase every time you register a new version of the task definition, so make sure you're using the latest one. The code is illustrated in the following snippet:

```
%%sh
aws ecs run-task
  --cluster fargate-demo
  --task-definition inference-fargate-tf-230:1
  --count 1
  --launch-type FARGATE
  --network-configuration
    "awsvpcConfiguration={subnets=[${SUBNET_ID}],
      securityGroups=[${SECURITY_GROUP_ID}],
      assignPublicIp=ENABLED}"
```

2. A few seconds later, we can see our prediction container running (showing the task **identifier (ID)**, state, ports, and task definition), as follows:

```
%%sh
ecs-cli ps --desired-status RUNNING
a9c9a3a8-8b7c-4dbb-9ec4-d20686ba5aec/dlc-tf-inference
  RUNNING
  52.49.238.243:8500->8500/tcp,
  52.49.238.243:8501->8501/tcp
  inference-fargate-tf230:1
```

3. Using the public **Internet Protocol (IP)** address of the container, we build a TensorFlow Serving prediction request with 10 sample images and send it to our container, as follows:

```
import random, json, requests
inference_task_ip = '52.49.238.243'
inference_url = 'http://' +
  inference_task_ip +
```

```
        ':8501/v1/models/1:predict'
indices = random.sample(range(x_val.shape[0] - 1), 10)
images = x_val[indices]/255
labels = y_val[indices]

data = images.reshape(num_samples, 28, 28, 1)
data = json.dumps(
    {"signature_name": "serving_default",
     "instances": data.tolist()})
headers = {"content-type": "application/json"}
json_response = requests.post(
    inference_url,
    data=data,
    headers=headers)
predictions = json.loads(
    json_response.text)['predictions']
predictions = np.array(predictions).argmax(axis=1)

print("Labels      : ", labels)
print("Predictions: ", predictions)
Labels      : [9 8 8 8 0 8 9 7 1 1]
Predictions: [9 8 8 8 0 8 9 7 1 1]
```

4. When we're done, we stop the task using the task **Amazon Resource Name (ARN)** returned by the run-task API and delete the cluster, as illustrated in the following code snippet. Of course, you can also use the ECS console:

```
%%sh
aws ecs stop-task --cluster fargate-demo \
                  --task $TASK_ARN
ecs-cli down --force --cluster fargate-demo
```

The processes for ECS and EKS are extremely similar. You can find simple examples at <https://gitlab.com/juliensimon/dlcontainers>. They should be a good starting point if you wish to build your own workflow.

Kubernetes fans can also use **SageMaker Operators for Kubernetes** and use native tools such as `kubectl` to train and deploy models. A detailed tutorial is available at <https://sagemaker.readthedocs.io/en/stable/workflows/kubernetes/index.html>.

Summary

In this chapter, you learned about model artifacts, what they contain, and how to use them to export models outside of SageMaker. You also learned how to import and deploy existing models, as well as how to manage endpoints in detail, both with the SageMaker SDK and the `boto3` SDK.

Then, we discussed alternative deployment scenarios with SageMaker, using either batch transform or inference pipelines, as well as outside of SageMaker with container services.

Finally, you learned how to use SageMaker Model Monitor to capture endpoint data and monitor data quality.

In the next chapter, we'll discuss automating ML workflows with three different AWS services: **AWS CloudFormation**, the **AWS Cloud Development Kit (AWS CDK)**, and **Amazon SageMaker Pipelines**.

12

Automating Machine Learning Workflows

In the previous chapter, you learned how to deploy machine learning models in different configurations, using both the **SageMaker SDK** and the `boto3` SDK. We used their APIs in **Jupyter Notebooks** – the preferred way to experiment and iterate quickly.

However, running notebooks for production tasks is not a good idea. Even if your code has been carefully tested, what about monitoring, logging, creating other AWS resources, handling errors, rolling back, and so on? Doing all of this right would require a lot of extra work and code, opening the possibility for more bugs. A more industrial approach is required.

In this chapter, you'll first learn how to provision SageMaker resources with **AWS CloudFormation** and **AWS Cloud Development Kit (CDK)** – two AWS services purposely built to bring repeatability, predictability, and robustness. You'll see how you can preview changes before applying them, in order to avoid uncontrolled and potentially destructive operations.

Then, you'll learn how to automate end-to-end machine learning workflows with two other services – **AWS Step Functions** and **Amazon SageMaker Pipelines**. You'll see how to build workflows with simple APIs, and how to visualize results in **SageMaker Studio**.

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

- Automating with AWS CloudFormation
- Automating with AWS CDK
- Building end-to-end workflows with AWS Step Functions
- Building end-to-end workflows with Amazon SageMaker Pipelines

Technical requirements

You will need an AWS account to run the examples included in this chapter. If you haven't got one already, please point your browser at <https://aws.amazon.com/getting-started/> to create one. You should also familiarize yourself with the AWS free tier (<https://aws.amazon.com/free/>), which lets you use many AWS services for free within certain usage limits.

You will need to install and configure the **AWS Command Line Interface (CLI)** for your account (<https://aws.amazon.com/cli/>).

You will need a working **Python 3.x** environment. Installing the **Anaconda** distribution (<https://www.anaconda.com/>) is not mandatory, but strongly encouraged, as it includes many projects that we will need (Jupyter, pandas, numpy, and more).

Code examples included in this book are available on GitHub at <https://github.com/PacktPublishing/Learn-Amazon-SageMaker-second-edition>. You will need to install a Git client to access them (<https://git-scm.com/>).

Automating with AWS CloudFormation

AWS CloudFormation has long been the preferred way to automate infrastructure builds and operations on AWS (<https://aws.amazon.com/cloudformation>). You could certainly write a book on the topic, but we'll stick to the basics in this section.

The first step in using CloudFormation is to write a template – that is, a **JSON** or **YAML** text file describing the **resources** that you want to build, such as an **EC2** instance or an **S3** bucket. Resources are available for almost all AWS services, and SageMaker is no exception. If we look at https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/AWS_SageMaker.html, we see that we can create SageMaker Studio applications, deploy endpoints, and more.

A template can (and should) include parameters and outputs. The former help make templates as generic as possible. The latter provide information that can be used by downstream applications, such as endpoint URLs or bucket names.

Once you've written your template file, you pass it to CloudFormation to create a **stack** – that is, a collection of AWS resources. CloudFormation will parse the template and create all resources automatically. Dependencies are also managed automatically, and resources will be created in the correct order. If a stack can't be created correctly, CloudFormation will roll it back, deleting resources that have been built so far.

A stack can be updated by applying a newer template revision. CloudFormation will analyze changes, and will create, delete, update, or replace resources accordingly. Thanks to **change sets**, you can verify changes before they are performed, and then decide whether to proceed or not.

Of course, a stack can be deleted, and CloudFormation will automatically tear down all its resources, which is a great way to clean up your builds without leaving any cruft behind.

Let's run a first example, where we deploy a model to a real-time endpoint.

Writing a template

This stack will be equivalent to calling the `boto3` API we studied in *Chapter 11, Deploying Machine Learning Models*: `create_model()`, `create_endpoint_configuration()`, and `create_endpoint()`. Accordingly, we'll define three CloudFormation resources (a model, an endpoint configuration, and an endpoint) and their parameters:

1. Creating a new YAML file named `endpoint-one-model.yml`, we first define the input parameters for the stack in the `Parameters` section. Each parameter has a name, a description, and a type. Optionally, we can provide default values:

```
AWSTemplateFormatVersion: 2010-09-09
```

```
Parameters:
```

```
  ModelName:
```

```
    Description: Model name
```

```
Type: String
ModelDataUrl:
  Description: Location of model artifact
  Type: String
ContainerImage:
  Description: Container used to deploy the model
  Type: String
InstanceType:
  Description: Instance type
  Type: String
  Default: ml.m5.large
InstanceCount:
  Description: Instance count
  Type: String
  Default: 1
RoleArn:
  Description: Execution Role ARN
  Type: String
```

2. In the Resources section, we define a model resource, using the `Ref` built-in function to reference the appropriate input parameters:

```
Resources:
  Model:
    Type: "AWS::SageMaker::Model"
    Properties:
      Containers:
        -
          Image: !Ref ContainerImage
          ModelDataUrl: !Ref ModelDataUrl
          ExecutionRoleArn: !Ref RoleArn
          ModelName: !Ref ModelName
```

3. We then define an endpoint configuration resource. We use the `GetAtt` built-in function to get the name of the model resource. Of course, this requires that the model resource already exists, and CloudFormation will make sure that resources are created in the right order:

```
EndpointConfig:  
  Type: "AWS::SageMaker::EndpointConfig"  
  Properties:  
    ProductionVariants:  
      -  
        ModelName: !GetAtt Model.ModelName  
        VariantName: variant-1  
        InitialInstanceCount: !Ref InstanceCount  
        InstanceType: !Ref InstanceType  
        InitialVariantWeight: 1.0
```

4. Finally, we define an endpoint resource. Likewise, we use `GetAtt` to find the name of the endpoint configuration:

```
Endpoint:  
  Type: "AWS::SageMaker::Endpoint"  
  Properties:  
    EndpointConfigName: !GetAtt  
      EndpointConfig.EndpointConfigName
```

5. In the `Outputs` section, we return the CloudFormation identifier of the endpoint, as well as its name:

```
Outputs:  
  EndpointId:  
    Value: !Ref Endpoint  
  EndpointName:  
    Value: !GetAtt Endpoint.EndpointName
```

Now that the template is complete (`endpoint-one-model.yml`), we can create a stack.

Note

Please make sure that your IAM role has permission to invoke CloudFormation APIs. If not, please add the `AWSCloudFormationFullAccess` managed policy to the role.

Deploying a model to a real-time endpoint

Let's use the `boto3` API to create a stack deploying a **TensorFlow** model. We'll reuse a model trained with **Keras** on **Fashion MNIST**:

Note

As our template is completely region-independent, you can use any region that you want. Just make sure that you have trained a model there, and that you're using the appropriate container image.

1. We'll need `boto3` clients for SageMaker and CloudFormation:

```
import boto3
sm = boto3.client('sagemaker')
cf = boto3.client('cloudformation')
```

2. We describe the training job to find the location of its artifact, and its execution role:

```
training_job =
    'tensorflow-training-2021-05-28-14-25-57-394'

job = sm.describe_training_job(
    TrainingJobName=training_job)
model_data_url =
    job['ModelArtifacts']['S3ModelArtifacts']

role_arn = job['RoleArn']
```

3. We set the container to use for deployment. In some cases, this is unnecessary, as the same container is used for training and deployment. For **TensorFlow** and other frameworks, SageMaker uses two different containers. You can find more information at https://github.com/aws/deep-learning-containers/blob/master/available_images.md:

```
container_image = '763104351884.dkr.ecr.us-east-1.  
amazonaws.com/tensorflow-inference:2.1.0-cpu-py36-  
ubuntu18.04'
```

4. Then, we read our template, create a new stack, and pass the required parameters:

```
import time  
  
timestamp = time.strftime("%Y-%m-%d-%H-%M-%S", time.  
gmtime())  
  
stack_name='endpoint-one-model-' + timestamp  
  
with open('endpoint-one-model.yml', 'r') as f:  
    response = cf.create_stack(  
        StackName=stack_name,  
        TemplateBody=f.read(),  
        Parameters=[  
            { "ParameterKey": "ModelName",  
              "ParameterValue": training_job+  
                '-' + timestamp },  
            { "ParameterKey": "ContainerImage",  
              "ParameterValue": container_image },  
            { "ParameterKey": "ModelDataUrl",  
              "ParameterValue": model_data_url },  
            { "ParameterKey": "RoleArn",  
              "ParameterValue": role_arn }  
        ]  
    )
```

5. Jumping to the CloudFormation console, we see that the stack is being created, as shown in the following screenshot. Notice that resources are created in the right order: model, endpoint configuration, and endpoint:

endpoint-one-model-2021-07-21-12-46-07					
Stack info	Events	Resources	Outputs	Parameters	
Events (9)					
<input type="text"/> Search events					
Timestamp	Logical ID	Status	Status reason		
2021-07-21 14:46:21 UTC+0200	Endpoint	ⓘ CREATE_IN_PROGRESS	Resource creation Initiated		
2021-07-21 14:46:20 UTC+0200	Endpoint	ⓘ CREATE_IN_PROGRESS	-		
2021-07-21 14:46:18 UTC+0200	EndpointConfig	✔ CREATE_COMPLETE	-		
2021-07-21 14:46:18 UTC+0200	EndpointConfig	ⓘ CREATE_IN_PROGRESS	Resource creation Initiated		

Figure 12.1 – Viewing stack creation

As we would expect, we also see the endpoint in SageMaker Studio, as shown in the following screenshot:

Endpoint: Endpoint-MTaOls4Vexpt		AWS settings			
Data quality	Model quality	Model explainability	Model bias	Monitoring job history	AWS settings
Endpoint settings					
Name	Endpoint-MTaOls4Vexpt	Project	-		
Status	ⓘ Creating	ARN	arn:aws:sagemaker:eu-west-1: XXXXXXXXXX		
Creation time	3 minutes ago	URL	https://runtime.sagemaker.eu-west-1.amazonaws.com/invocations	Learn more about the API	
Last updated	3 minutes ago				

Figure 12.2 – Viewing endpoint creation

- Once the stack creation is complete, we can use its output to find the name of the endpoint:

```
response = cf.describe_stacks(StackName=stack_name)
print(response['Stacks'][0]['StackStatus'])
for o in response['Stacks'][0]['Outputs']:
    if o['OutputKey']=='EndpointName':
        endpoint_name = o['OutputValue']
print(endpoint_name)
```

This prints out the stack status and the endpoint name autogenerated by CloudFormation:

CREATE_COMPLETE

Endpoint-MTaOIs4Vexpt

- We can test the endpoint as usual. Then, we can delete the stack and its resources:

```
cf.delete_stack(StackName=stack_name)
```

However, let's not delete the stack right away. Instead, we're going to update it using a change set.

Modifying a stack with a change set

Here, we're going to update the number of instances backing the endpoint:

- We create a new change set using the same template and parameters, except InstanceCount, which we set to 2:

```
response = cf.create_change_set(
    StackName=stack_name,
    ChangeSetName='add-instance',
    UsePreviousTemplate=True,
    Parameters=[
        { "ParameterKey": "InstanceCount",
          "ParameterValue": "2" },
        { "ParameterKey": "ModelName",
          "UsePreviousValue": True },
        { "ParameterKey": "ContainerImage",
          "UsePreviousValue": True },
        { "ParameterKey": "ModelDataUrl",
```

```

        "UsePreviousValue": True } ,
    { "ParameterKey": "RoleArn",
      "UsePreviousValue": True }
]
)

```

2. We see details on the change set in the CloudFormation console, as shown in the next screenshot. We could also use the `describe_change_set()` API:

The screenshot shows the AWS CloudFormation console with the 'Changes' tab selected. The interface includes tabs for 'Changes', 'Input', 'Template', and 'JSON changes'. Below the tabs, a search bar is present. The main area displays a table titled 'Changes (2)' with the following data:

Action	Logical ID	Physical ID	Resource type	Replacement
Modify	EndpointConfig	arn:aws:sagemaker:eu-west-1:████████:endpoint-config/endpointconfig-bhrvtb9gvx5p	AWS::SageMaker::EndpointConfig	Conditional
Modify	Endpoint	arn:aws:sagemaker:eu-west-1:████████:endpoint/endpoint-mtaois4vexpt	AWS::SageMaker::Endpoint	Conditional

Figure 12.3 – Viewing a change set

This tells us that the endpoint configuration and the endpoint need to be modified, and possibly replaced. As we already know from *Chapter 11, Deploying Machine Learning Models*, a new endpoint will be created and applied in a non-disruptive fashion to the existing endpoint.

Note

When working with CloudFormation, it's critical that you understand the **replacement policy** for your resources. Details are available in the documentation for each resource type.

3. By clicking on the **Execute** button, we execute the change set. We could also use the `execute_change_set()` API. As expected, the endpoint is immediately updated, as shown in the following screenshot:

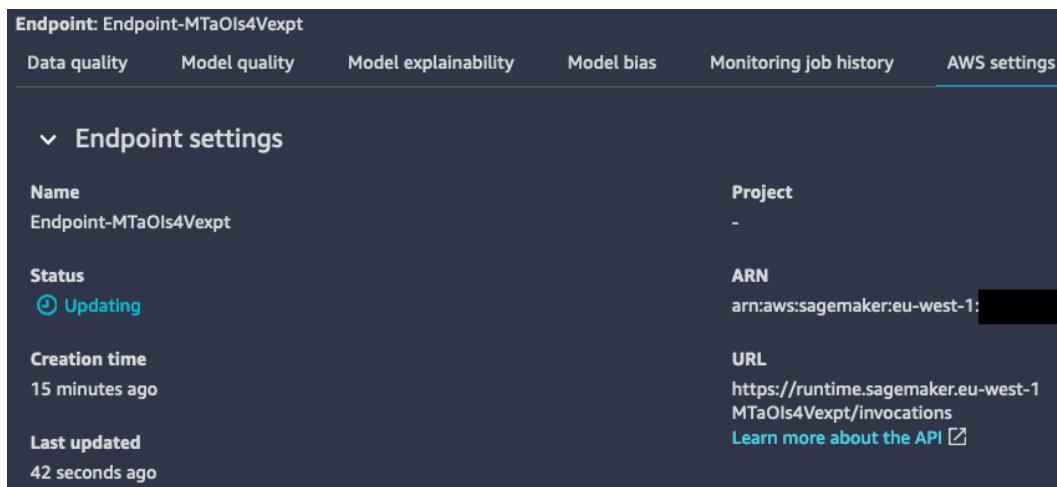


Figure 12.4 – Updating the endpoint

- Once the update is complete, we see the sequence of events in the CloudFormation console, as shown in the next screenshot. A new endpoint configuration has been created and applied to the endpoint. The previous endpoint configuration has been deleted:

Events (21)			
<input type="text"/> Search events			
Timestamp	Logical ID	Status	Status reason
2021-07-21 15:06:41 UTC+0200	endpoint-one-model-2021-07-21-12-46-07	✓ UPDATE_COMPLETE	-
2021-07-21 15:06:41 UTC+0200	EndpointConfig	✓ DELETE_COMPLETE	-
2021-07-21 15:06:40 UTC+0200	EndpointConfig	ℹ️ DELETE_IN_PROGRESS	-
2021-07-21 15:06:39 UTC+0200	endpoint-one-model-2021-07-21-12-46-07	ℹ️ UPDATE_COMPLETE_CLEARED_BY_ANUP_IN_PROGRESS	-
2021-07-21 15:06:36 UTC+0200	Endpoint	✓ UPDATE_COMPLETE	-
2021-07-21 15:00:27 UTC+0200	Endpoint	ℹ️ UPDATE_IN_PROGRESS	-
2021-07-21 15:00:25 UTC+0200	EndpointConfig	✓ UPDATE_COMPLETE	-
2021-07-21 15:00:25 UTC+0200	EndpointConfig	ℹ️ UPDATE_IN_PROGRESS	Resource creation Initiated

Figure 12.5 – Updating the stack

5. We can check that the endpoint is now backed by two instances:

```
r = sm.describe_endpoint(EndpointName=endpoint_name)
print r(['ProductionVariants'][0]
        ['CurrentInstanceCount'])
```

This prints out the number of instances backing the Production Variant:

```
2
```

Let's keep working with change sets and add a second production variant to the endpoint.

Adding a second production variant to the endpoint

Our initial template only defined a single production variant. We'll update it and add another one (`endpoint-two-models.yml`):

1. In the Parameters section, we add entries for a second model:

```
ModelName2:
  Description: Second model name
  Type: String
ModelDataUrl2:
  Description: Location of second model artifact
  Type: String
VariantWeight2:
  Description: Weight of second model
  Type: String
  Default: 0.0
```

2. We do the same in the Resources section:

```
Model2:
  Type: "AWS::SageMaker::Model"
  Properties:
    Containers:
      -
        Image: !Ref ContainerImage
        ModelDataUrl: !Ref ModelDataUrl2
    ExecutionRoleArn: !Ref RoleArn
    ModelName: !Ref ModelName2
```

3. Moving back to our notebook, we get information on another training job. We then create a change set, reading the updated template and passing all required parameters:

```
training_job_2 = 'tensorflow-
training-2020-06-08-07-32-18-734'
job_2=sm.describe_training_job(
    TrainingJobName=training_job_2)
model_data_url_2=
    job_2['ModelArtifacts']['S3ModelArtifacts']
with open('endpoint-two-models.yml', 'r') as f:
    response = cf.create_change_set(
        StackName=stack_name,
        ChangeSetName='add-model',
        TemplateBody=f.read(),
        Parameters=[
            { "ParameterKey": "ModelName",
              "UsePreviousValue": True },
            { "ParameterKey": "ModelDataUrl",
              "UsePreviousValue": True },
            { "ParameterKey": "ContainerImage",
              "UsePreviousValue": True },
            { "ParameterKey": "RoleArn",
              "UsePreviousValue": True },
            { "ParameterKey": "ModelName2",
              "ParameterValue": training_job_2+'-
                '+timestamp},
            { "ParameterKey": "ModelDataUrl2",
              "ParameterValue": model_data_url_2 }
        ]
    )
```

4. Looking at the CloudFormation console, we see the changes caused by the change set. Create a new model and modify the endpoint configuration and the endpoint:

Changes (3)				
Action	Logical ID	Physical ID	Resource type	Replacement
Modify	EndpointConfig	arn:aws:sagemaker:eu-west-1:████████:endpoint-config/endpointconfig-p1d5qlpvnaw	AWS::SageMaker::EndpointConfig	Conditional
Modify	Endpoint	arn:aws:sagemaker:eu-west-1:████████:endpoint/endpoint-mtaois4vexpt	AWS::SageMaker::Endpoint	Conditional
Add	Model2	-	AWS::SageMaker::Model	-

Figure 12.6 – Viewing the change set

5. We execute the change set. Once it's complete, we see that the endpoint now supports two production variants. Note that the instance count is back to its initial value, as we defined it as 1 in the updated template:

▼ Endpoint runtime settings					
Variant name	Current weight	Desired weight	Instance type	Elastic inference	Current instance count
variant-1	1	1	ml.m5.large	-	1
variant-2	0	0	ml.m5.large	-	1

Figure 12.7 – Viewing production variants

The new production variant has a weight of 0, so it won't be used for prediction. Let's see how we can gradually introduce it using **canary deployment**.

Implementing canary deployment

Canary deployment is a popular technique for gradual application deployment (<https://martinfowler.com/bliki/CanaryRelease.html>), and it can also be used for machine learning models.

Simply put, we'll use a series of stack updates to gradually increase the weight of the second production variant in 10% increments, until it completely replaces the first production variant. We'll also create a CloudWatch alarm monitoring the latency of the second production variant – if the alarm is triggered, the change set will be rolled back:

1. We create a CloudWatch alarm monitoring the 60-second average latency of the second production variant. We set the threshold at 500 milliseconds:

```
cw = boto3.client('cloudwatch')
alarm_name = 'My_endpoint_latency'
response = cw.put_metric_alarm(
    AlarmName=alarm_name,
    ComparisonOperator='GreaterThanOrEqualToThreshold',
    EvaluationPeriods=1,
    MetricName='ModelLatency',
    Namespace='AWS/SageMaker',
    Period=60,
    Statistic='Average',
    Threshold=500000.0,
    AlarmDescription=
        '1-minute average latency exceeds 500ms',
    Dimensions=[
        { 'Name': 'EndpointName',
          'Value': endpoint_name },
        { 'Name': 'VariantName',
          'Value': 'variant-2' }
    ],
    Unit='Microseconds'
)
```

2. We find the ARN of the alarm:

```
response = cw.describe_alarms(AlarmNames=[alarm_name])
for a in response['MetricAlarms']:
    if a['AlarmName'] == alarm_name:
        alarm_arn = a['AlarmArn']
```

3. Then, we loop over weights and update the stack. Change sets are unnecessary here, as we know exactly what's going to happen from a resource perspective. We set our CloudWatch alarm as a **rollback trigger**, giving it five minutes to go off after each update before moving on to the next:

```
for w in list(range(10,110,10)):
    response = cf.update_stack(
        StackName=stack_name,
        UsePreviousTemplate=True,
        Parameters=[
            { "ParameterKey": "ModelName",
              "UsePreviousValue": True },
            { "ParameterKey": "ModelDataUrl",
              "UsePreviousValue": True },
            { "ParameterKey": "ContainerImage",
              "UsePreviousValue": True },
            { "ParameterKey": "RoleArn",
              "UsePreviousValue": True },
            { "ParameterKey": "ModelName2",
              "UsePreviousValue": True },
            { "ParameterKey": "ModelDataUrl2",
              "UsePreviousValue": True },
            { "ParameterKey": "VariantWeight",
              "ParameterValue": str(100-w) },
            { "ParameterKey": "VariantWeight2",
              "ParameterValue": str(w) }
        ],
        RollbackConfiguration={
            'RollbackTriggers': [
                { 'Arn': alarm_arn, :
                  'AWS::CloudWatch::Alarm' }]
```

```

        ] ,
        'MonitoringTimeInMinutes': 5
    }
)
waiter = cf.get_waiter('stack_update_complete')
waiter.wait(StackName=stack_name)
print("Sending %d% of traffic to new model" % w)

```

That's all it takes. Pretty cool, don't you think?

This cell will run for a couple of hours, so don't stop it. In another notebook, the next step is to start sending some traffic to the endpoint. For the sake of brevity, I won't include the code, which is identical to the one we used in *Chapter 7, Extending Machine Learning Services with Built-in Frameworks*. You'll find the notebook in the GitHub repository for this book (`Chapter12/cloudformation/Predict Fashion MNIST images.ipynb`).

Now, all we have to do is sit back, have a cup of tea, and enjoy the fact that our model is being deployed safely and automatically. As endpoint updates are seamless, client applications won't notice a thing.

After a couple of hours, deployment is complete. The next screenshot shows invocations for both variants over time. As we can see, traffic was gradually shifted from the first variant to the second one:



Figure 12.8 – Monitoring canary deployment

Latency stayed well under our 500-millisecond limit, and the alarm wasn't triggered, as shown in the next screenshot:

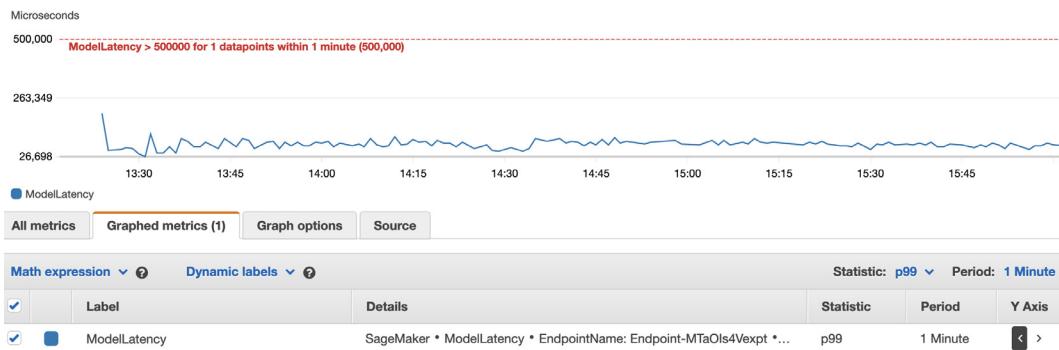


Figure 12.9 – Viewing the CloudWatch alarm

This example can serve as a starting point for your own deployments. For example, you could add an alarm monitoring 4xx or 5xx HTTP errors. You could also monitor a business metric directly impacted by prediction latency and accuracy, such as click-through rate, conversion rate, and so on. A useful thing to add would be an alarm notification (email, SMS, or even a Lambda function) in order to trigger downstream actions, should model deployment fail. The possibilities are endless!

When you're done, *don't forget to delete the stack*, either in the CloudFormation console or with the `delete_stack()` API. This will automatically clean up all AWS resources created by the stack.

Blue-green deployment is another popular technique. Let's see how we can implement it on SageMaker.

Implementing blue-green deployment

Blue-green deployment requires two production environments (<https://martinfowler.com/bliki/BlueGreenDeployment.html>):

- The live production environment (blue) running version n
- A copy of this environment (green) running version n+1

Let's look at two possible scenarios, which we could implement using the same APIs we've used for canary deployment.

Implementing blue-green deployment with a single endpoint

Starting from an existing endpoint running the current version of the model, we would carry out the following steps:

1. Create a new endpoint configuration with two production variants: one for the current model and one for the new model. Initial weights would be set to 1 and 0 respectively.
2. Apply it to the endpoint.
3. Run tests on the new production variant, selecting it explicitly with the `TargetVariant` parameter in `invoke_endpoint()`.
4. When tests are satisfactory, update weights to 0 and 1. This will seamlessly switch traffic to the new model. If anything goes wrong, revert the weights to 1 and 0.
5. When the deployment is complete, update the endpoint to delete the first production variant.

This is a simple and robust solution. However, updating an endpoint takes several minutes, making the whole process not as quick as one may want. Let's see how we can fix this problem by using two endpoints.

Implementing blue-green deployment with two endpoints

Starting from an existing endpoint running the current version of the model, we would implement the following steps:

1. Create a second endpoint running the new version of the model.
2. Run tests on this new endpoint.
3. When the tests are satisfactory, switch all traffic to the new endpoint. This could be achieved in different ways; for example, updating a parameter in your business application, or updating a private DNS entry. If anything goes wrong, revert to the previous setting.
4. When the deployment is complete, delete the old endpoint.

This setup is a little more complex, but it lets you switch instantly from one model version to the next, both for deployments and rollbacks.

CloudFormation is a fantastic tool for automation, and any time spent learning it will pay dividends. Yet some AWS users prefer writing code to writing templates, which is why we introduced the CDK.

Automating with AWS CDK

AWS CDK is a multi-language SDK that lets you write code to define AWS infrastructure (<https://github.com/aws/aws-cdk>). Using the CDK CLI, you can then provision this infrastructure, using CloudFormation under the hood.

Installing the CDK

The CDK is natively implemented with **Node.js**, so please make sure that the `npm` tool is installed on your machine (<https://www.npmjs.com/get-npm>).

Installing the CDK is then as simple as this:

```
$ npm i -g aws-cdk  
$ cdk --version  
1.114.0 (build 7e41b6b)
```

Let's create a CDK application and deploy an endpoint.

Creating a CDK application

We'll deploy the same model that we deployed with CloudFormation. I'll use Python, and you could also use **JavaScript**, **TypeScript**, **Java**, and **.NET**. API documentation is available at <https://docs.aws.amazon.com/cdk/api/latest/python/>:

1. First, we create a Python application named `endpoint`:

```
$ mkdir cdk  
$ cd cdk  
$ cdk init --language python --app endpoint
```

2. This automatically creates a virtual environment, which we need to activate:

```
$ source .venv/bin/activate
```

3. This also creates a default `app.py` file for our CDK code, a `cdk.json` file for application configuration, and a `requirements.txt` file to install dependencies. Instead, we'll use the files present in the GitHub repository:

4. In the `requirements.txt` file, we install the CDK package for S3 and SageMaker. Each service requires a different package. For example, we would add `aws_cdk.aws_s3` for S3:

```
-e .  
aws_cdk.aws_s3  
aws_cdk.aws_sagemaker
```

5. We then install requirements as usual:

```
$ pip install -r requirements.txt
```

6. In the `cdk.json` file, we store the application context. Namely, key-value pairs that can be read by the application for configuration (<https://docs.aws.amazon.com/cdk/latest/guide/context.html>):

```
{  
    "app": "python3 app.py",  
    "context": {  
        "role_arn": "arn:aws:iam::123456789012:role/  
Sagemaker-fullaccess"  
        "model_name": "tf2-fmnist",  
        "epc_name": "tf2-fmnist-epc",  
        "ep_name": "tf2-fmnist-ep",  
        "image": "763104351884.dkr.ecr.us-east-1.amazonaws.  
com/tensorflow-inference:2.1-cpu",  
        "model_data_url": "s3://sagemaker-us-  
east-1-123456789012/keras2-fashion-mnist/output/  
tensorflow-training-2020-06-08-07-46-04-367/output/model.  
tar.gz"  
        "instance_type": "ml.t2.xlarge",  
        "instance_count": 1  
    }  
}
```

This is the preferred way to pass values to your application. You should manage this file with version control in order to keep track of how stacks were built.

7. We can view the context of our application with the `cdk context` command:

#	Key	Value
1	ep_name	"tf2-fmnist-ep"
2	epc_name	"tf2-fmnist-epc"
3	image	"763104351884.dkr.ecr.eu-west-1.amazonaws.com/tensorflow-inference:2.1-cpu"
4	instance_count	1
5	instance_type	"ml.t2.xlarge"
6	model_data_url	"s3://sagemaker-eu-west-1-[REDACTED]/tensorflow-training-2021-05-28-14-25-57-394/output/model.tar.gz"
7	model_name	"tf2-fmnist"
8	role_arn	"arn:aws:iam:[REDACTED]:role/Sagemaker-fullaccess"

Figure 12.10 – Viewing CDK context

Now, we need to write the actual application.

Writing a CDK application

All code goes in the `app.py` file, which we implement in the following steps:

1. We import the required packages:

```
import time
from aws_cdk import (
    aws_sagemaker as sagemaker,
    core
)
```

2. We extend the `core.Stack` class to create our own stack:

```
class SagemakerEndpoint(core.Stack):
    def __init__(self, app: core.App, id: str, **kwargs) -> None:
        timestamp =
            '-' + time.strftime(
                "%Y-%m-%d-%H-%M-%S", time.gmtime())
        super().__init__(app, id, **kwargs)
```

3. We add a `CfnModel` object, reading the appropriate context values:

```
model = sagemaker.CfnModel(
    scope = self,
```

```
        id="my_model",
        execution_role_arn=
            self.node.try_get_context("role_arn"),
        containers=[{
            "image":
                self.node.try_get_context("image"),
            "modelDataUrl":
                self.node.try_get_context("model_data_url")
        }],
        model_name= self.node.try_get_context(
            "model_name")+timestamp
    )
```

4. We add a CfnEndpointConfig object, using the built-in `get_att()` function to associate it to the model. This creates a dependency that CloudFormation will use to build resources in the right order:

```
epc = sagemaker.CfnEndpointConfig(
    scope=self,
    id="my_epc",
    production_variants=[{
        "modelName": core.Fn.get_att(
            model.logical_id,
            'ModelName'
        ).to_string(),
        "variantName": "variant-1",
        "initialVariantWeight": 1.0,
        "initialInstanceCount": 1,
        "instanceType":
            self.node.try_get_context(
                "instance_type")
    }],
    endpoint_config_name=
        self.node.try_get_context("epc_name")
        +timestamp
)
```

5. We add a CfnEndpoint object, using the built-in `get_att()` function to associate it to the endpoint configuration:

```
ep = sagemaker.CfnEndpoint(  
    scope=self,  
    id="my_ep",  
    endpoint_config_name=  
        core.Fn.get_att(  
            epc.logical_id,  
            'EndpointConfigName'  
        ).to_string(),  
    endpoint_name=  
        self.node.try_get_context("ep_name")  
        +timestamp  
)
```

6. Finally, we instantiate the application:

```
app = core.App()  
SagemakerEndpoint(  
    app,  
    "SagemakerEndpoint",  
    env={'region': 'eu-west-1'}  
)  
app.synth()
```

Our code is complete!

Deploying a CDK application

We can now deploy the endpoint:

1. We can list the available stacks:

```
$ cdk list  
SagemakerEndpointEU
```

2. We can also see the actual CloudFormation template. It should be extremely similar to the template we wrote in the previous section:

```
$ cdk synth SagemakerEndpointEU
```

- Deploying the stack is equally simple, as shown in the next screenshot:

```
(.venv) (base) ➔ cdk git:(master) ✘ cdk deploy SagemakerEndpointEU
SagemakerEndpointEU: deploying...
SagemakerEndpointEU: creating CloudFormation changeset...
[██████████ .....] (3/5)

18:25:37 | CREATE_IN_PROGRESS | AWS::CloudFormation::Stack | SagemakerEndpointEU
18:25:51 | CREATE_IN_PROGRESS | AWS::SageMaker::Endpoint | my_ep
```

Figure 12.11 – Deploying an endpoint

- Looking at CloudFormation, we see that the stack is created using a change set. A few minutes later, the endpoint is in service.
- Editing app.py, we set the initial instance count to 2. We then ask CDK to deploy the stack, but without executing the change set, as shown in the next screenshot:

```
(.venv) (base) ➔ cdk git:(master) ✘ cdk deploy --no-execute SagemakerEndpointEU
SagemakerEndpointEU: deploying...
SagemakerEndpointEU: creating CloudFormation changeset...
Changeset arn:aws:cloudformation:eu-west-1:613904931467:changeSet/cdk-deploy-change-set/f755ef57-3570-48f4-8d1a-4c2f63da74b1 created and waiting in review for manual execution (-no-execute)

✓ SagemakerEndpointEU
```

Figure 12.12 – Creating a change set

- If we're happy with the change set, we can execute it in the CloudFormation console, or run the previous command again without --no-execute. As expected, and as shown in the next screenshot, the endpoint is updated:

```
(.venv) (base) ➔ cdk git:(master) ✘ cdk deploy SagemakerEndpointEU
SagemakerEndpointEU: deploying...
SagemakerEndpointEU: creating CloudFormation changeset...
[██████████ .....] (2/4)

18:35:29 | UPDATE_IN_PROGRESS | AWS::CloudFormation::Stack | SagemakerEndpointEU
18:35:50 | UPDATE_IN_PROGRESS | AWS::SageMaker::Endpoint | my_ep
```

Figure 12.13 – Updating the endpoint

- When we're done, we can destroy the stack:

```
$ cdk destroy SagemakerEndpointEU
```

As you can see, the CDK is an interesting alternative to writing templates directly, while still benefiting from the rigor and the robustness of CloudFormation.

One thing we haven't done yet is to automate an end-to-end workflow, from training to deployment. Let's do this with AWS Step Functions.

Building end-to-end workflows with AWS Step Functions

AWS Step Functions let you define and run workflows based on **state machines** (<https://aws.amazon.com/step-functions/>). A state machine is a combination of steps, which can be sequential, parallel, or conditional. Each step receives an input from its predecessor, performs an operation, and passes the output to its successor. Step Functions are integrated with many AWS services, such as Amazon SageMaker, **AWS Lambda**, container services, **Amazon DynamoDB**, **Amazon EMR**, **AWS Glue**, and more.

State machines can be defined using JSON and the **Amazon States Language**, and you can visualize them in the service console. State machine execution is fully managed, so you don't need to provision any infrastructure to run.

When it comes to SageMaker, Step Functions has a dedicated Python SDK, oddly named the **Data Science SDK** (<https://github.com/aws/aws-step-functions-data-science-sdk-python>).

Let's run an example where we automate training and deployment for a **scikit-learn** model trained on the **Boston Housing** dataset.

Setting up permissions

First, please make sure that the IAM role for your user or for your notebook instance has permission to invoke Step Functions APIs. If not, please add the **AWSStepFunctionsFullAccess** managed policy to the role.

Then, we need to create a service role for Step Functions, allowing it to invoke AWS APIs on our behalf:

1. Starting from the IAM console (<https://console.aws.amazon.com/iam/home#/roles>), we click on **Create role**.
2. We select **AWS service** and **Step Functions**.
3. We click through the next screens until we can enter the role name. Let's call it **StepFunctionsWorkflowExecutionRole**, and click on **Create role**.
4. Selecting this role, we click on its **Permission** tab, then on **Add inline policy**.
5. Selecting the **JSON** tab, we replace the empty policy with the content of the `Chapter12/step_functions/service-role-policy.json` file, and we click on **Review policy**.

6. We name the policy `StepFunctionsWorkflowExecutionPolicy` and click on **Create policy**.
7. We write down the ARN on the role, and we close the IAM console.

The setup is now complete. Now, let's create a workflow.

Implementing our first workflow

In this workflow, we'll go through the following step sequence: train the model, create it, use it for a batch transform, create an endpoint configuration, and deploy the model to an endpoint:

1. We upload the training set to S3, as well as a test set where we removed the target attribute. We'll use the latter for a batch transform:

```
import sagemaker
import pandas as pd
sess = sagemaker.Session()
bucket = sess.default_bucket()
prefix = 'sklearn-boston-housing-stepfunc'
training_data = sess.upload_data(
    path='housing.csv',
    key_prefix=prefix + "/training")
data = pd.read_csv('housing.csv')
data.drop(['medv'], axis=1, inplace=True)
data.to_csv('test.csv', index=False, header=False)
batch_data = sess.upload_data(
    path='test.csv',
    key_prefix=prefix + "/batch")
```

2. We configure our estimator as usual:

```
from sagemaker.sklearn import SKLearn
output = 's3://{{}}/{{}}/output/'.format(bucket,prefix)
sk = SKLearn(
    entry_point='sklearn-boston-housing.py',
    role=sagemaker.get_execution_role(),
    framework_version='0.23-1',
    train_instance_count=1,
    train_instance_type='ml.m5.large',
```

```
        output_path=output,
        hyperparameters={
            'normalize': True,
            'test-size': 0.1
        }
    )
```

3. We also define the transformer that we'll use for batch transform:

```
sk_transformer = sk.transformer(
    instance_count=1,
    instance_type='ml.m5.large')
```

4. We import the Step Functions objects required by the workflow. You can find the API documentation at <https://aws-step-functions-data-science-sdk.readthedocs.io/en/latest/>:

```
import stepfunctions
from stepfunctions import steps
from stepfunctions.steps import TrainingStep, ModelStep,
TransformStep
from stepfunctions.inputs import ExecutionInput
from stepfunctions.workflow import Workflow
```

5. We define the input of the workflow. We'll pass it a training job name, a model name, and an endpoint name:

```
execution_input = ExecutionInput(schema={
    'JobName': str,
    'ModelName': str,
    'EndpointName': str
})
```

6. The first step of the workflow is the training step. We pass it the estimator, the location of the dataset in S3, and a training job name:

```
from sagemaker.inputs import TrainingInput
training_step = TrainingStep(
    'Train Scikit-Learn on the Boston Housing dataset',
    estimator=sk,
    data={'training': TrainingInput(
```

```
        training_data, content_type='text/csv')},
    job_name=execution_input['JobName']
)
```

7. The next step is the model creation step. We pass it the location of the model trained in the previous step, and a model name:

```
model_step = ModelStep(
    'Create the model in SageMaker',
    model=training_step.get_expected_model(),
    model_name=execution_input['ModelName']
)
```

8. The next step is running a batch transform on the test dataset. We pass the transformer object, the location of the test dataset in S3, and its content type:

```
transform_step = TransformStep(
    'Transform the dataset in batch mode',
    transformer=sk_transformer,
    job_name=execution_input['JobName'],
    model_name=execution_input['ModelName'],
    data=batch_data,
    content_type='text/csv'
)
```

9. The next step is creating the endpoint configuration:

```
endpoint_config_step = EndpointConfigStep(
    "Create an endpoint configuration for the model",
    endpoint_config_name=execution_input['ModelName'],
    model_name=execution_input['ModelName'],
    initial_instance_count=1,
    instance_type='ml.m5.large'
)
```

10. The last step is creating the endpoint:

```
endpoint_step = EndpointStep(
    "Create an endpoint hosting the model",
    endpoint_name=execution_input['EndpointName'],
```

```
    endpoint_config_name=execution_input['ModelName']
)
```

11. Now that all steps have been defined, we chain them in sequential order:

```
workflow_definition = Chain([
    training_step,
    model_step,
    transform_step,
    endpoint_config_step,
    endpoint_step
])
```

12. We now build our workflow, using the workflow definition and the input definition:

```
import time
timestamp = time.strftime("%Y-%m-%d-%H-%M-%S", time.
                           gmttime())
workflow_execution_role = "arn:aws:iam::0123456789012:r
ole/
StepFunctionsWorkflowExecutionRole"
workflow = Workflow(
    name='sklearn-boston-housing-workflow1-{}'
        .format(timestamp),
    definition=workflow_definition,
    role=workflow_execution_role,
    execution_input=execution_input
)
```

13. We can visualize the state machine, an easy way to check that we built it as expected, as shown in the next screenshot:

```
workflow.render_graph(portrait=True)
```

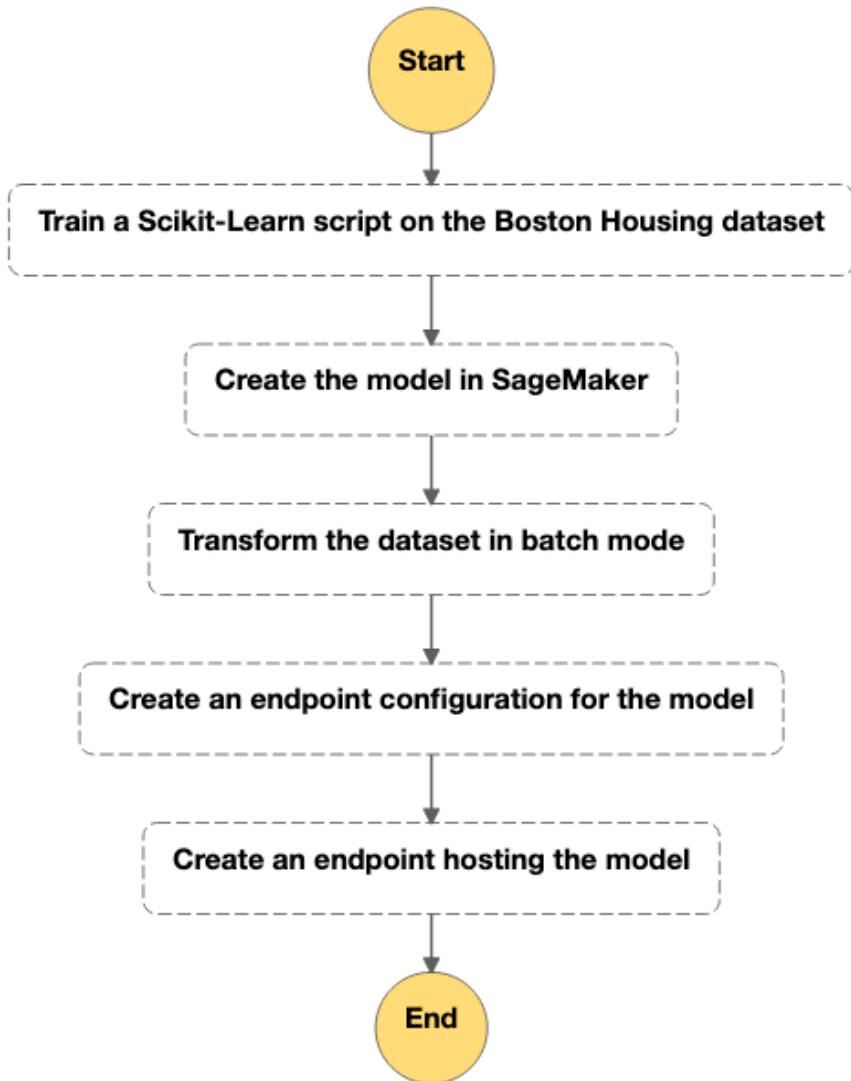


Figure 12.14 – Viewing the state machine

14. We create the workflow:

```
workflow.create()
```

15. It's visible in the Step Functions console, as shown in the following screenshot.
 We can see both its graphical representation and its JSON definition based on the Amazon States Language. We could edit the workflow as well if needed:

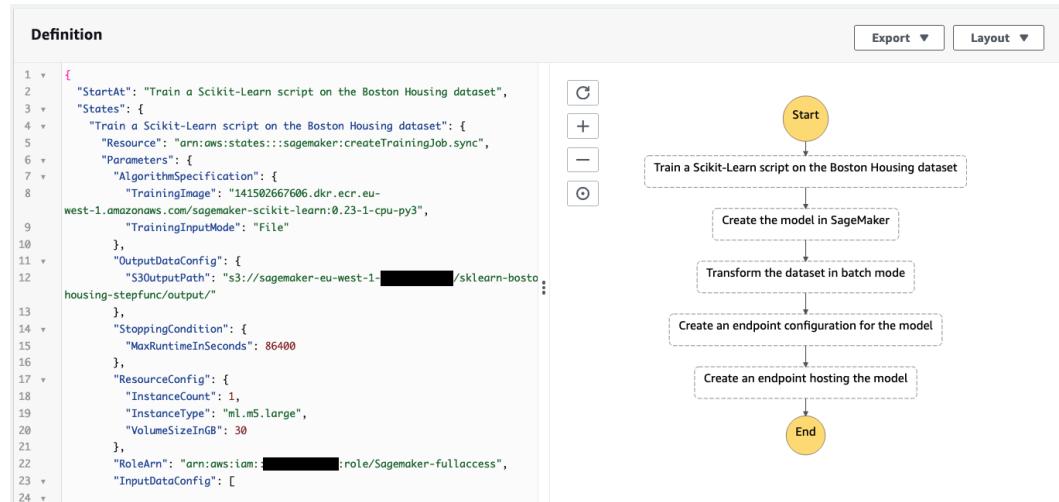


Figure 12.15 – Viewing the state machine in the console

16. We run the workflow:

```

execution = workflow.execute(
    inputs={
        'JobName': 'sklearn-boston-housing-{}'
                    .format(timestamp),
        'modelName': 'sklearn-boston-housing-{}'
                    .format(timestamp),
        'endpointName': 'sklearn-boston-housing-{}'
                      .format(timestamp)
    }
)
  
```

17. We can track its progress with `render_progress()` and the `list_events()` API. We can also see it in the console, as shown in the next screenshot. Note that we also see the input and output of each step, which is a great way to troubleshoot problems:

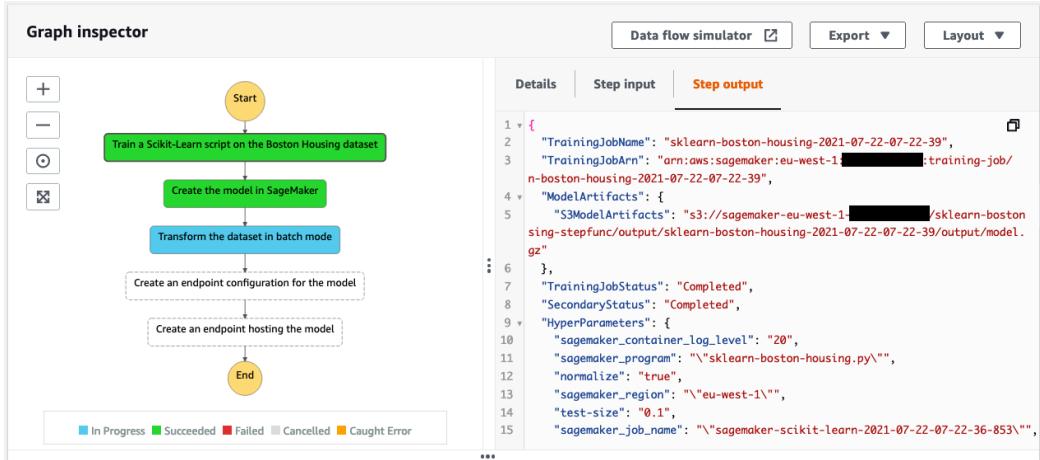


Figure 12.16 – Running the state machine

18. When the workflow is complete, you can test the endpoint as usual. *Don't forget to delete it in the SageMaker console when you're done.*

This example shows how simple it is to build a SageMaker workflow with this SDK. Still, we could improve it by running batch transform and endpoint creation in parallel.

Adding parallel execution to a workflow

The next screenshot shows the workflow we're going to build. The steps themselves are exactly the same. We're only going to modify the way they're chained:

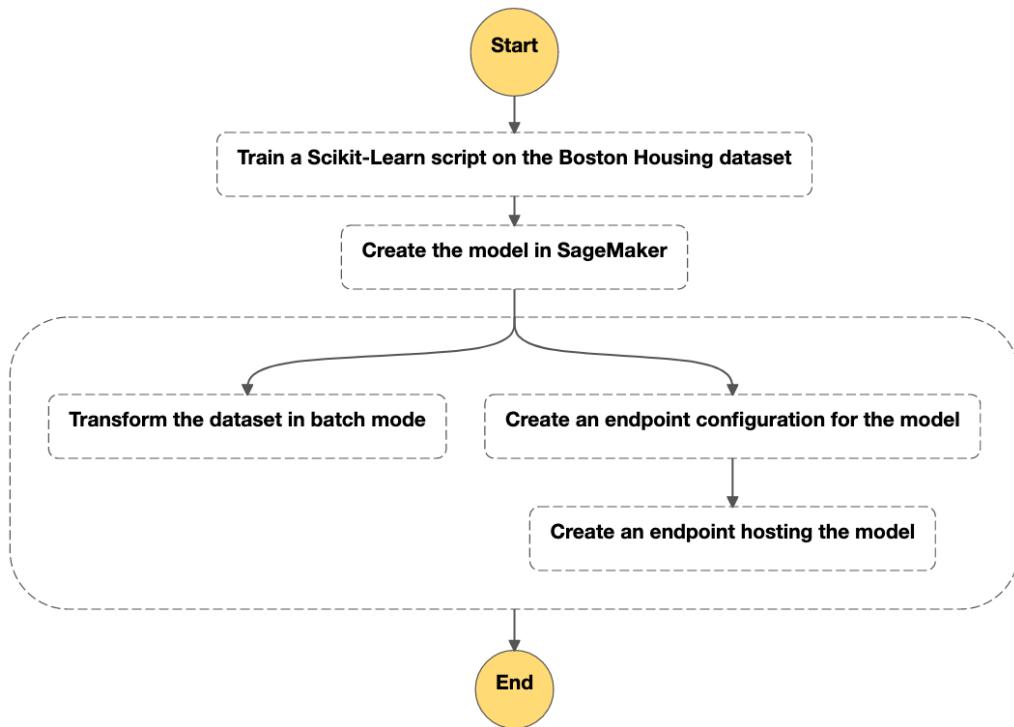


Figure 12.17 – Viewing the parallel state machine

We will get started using the following steps:

1. Our workflow has two branches – one for batch transform and one for the endpoint:

```
batch_branch = Chain([
    transform_step
])
endpoint_branch = Chain([
    endpoint_config_step,
    endpoint_step
])
```

2. We create a `Parallel` step in order to allow parallel execution of these two branches:

```
parallel_step = Parallel('Parallel execution')
parallel_step.add_branch(batch_branch)
parallel_step.add_branch(endpoint_branch)
```

3. We put everything together:

```
workflow_definition = Chain([
    training_step,
    model_step,
    parallel_step
])
```

That's it! We can now create and run this workflow just like in the previous example.

Looking at the Step Functions console, we see that the workflow does run the two branches in parallel. There is a minor problem, however. The endpoint creation step is shown as complete, although the endpoint is still being created. You can see in the SageMaker console that the endpoint is listed as `Creating`. This could cause a problem if a client application tried to invoke the endpoint right after the workflow completes.

Let's improve this by adding an extra step, waiting for the endpoint to be in service. We can easily do this with a Lambda function, allowing us to run our own code anywhere in a workflow.

Adding a Lambda function to a workflow

If you've never looked at **AWS Lambda** (<https://aws.amazon.com/lambda>), you're missing out! Lambda is at the core of serverless architectures, where you can write and deploy short functions running on fully managed infrastructure. These functions can be triggered by all sorts of AWS events, and they can also be invoked on demand.

Setting up permissions

Creating a Lambda function is simple. The only prerequisite is to create an **execution role** – that is, an IAM role that gives the function permission to invoke other AWS services. Here, we only need permission for the `DescribeEndpoint` API, as well as permission to create a log in CloudWatch. Let's use the `boto3` API for this. You can find more information at <https://docs.aws.amazon.com/lambda/latest/dg/lambda-permissions.html>:

1. We first define a **trust policy** for the role, allowing it to be assumed by the Lambda service:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {"Effect": "Allow",  
         "Principal": {  
             "Service": "lambda.amazonaws.com"  
         },  
         "Action": "sts:AssumeRole"  
     ]  
}
```

2. We create a role and attach the trust policy to it:

```
iam = boto3.client('iam')  
with open('trust-policy.json') as f:  
    policy = f.read()  
    role_name = 'lambda-role-sagemaker-describe-endpoint'  
response = iam.create_role(  
    RoleName=role_name,  
    AssumeRolePolicyDocument=policy,  
    Description='Allow function to invoke all SageMaker  
APIs'  
)  
role_arn = response['Role']['Arn']
```

3. We define a policy listing the APIs that are allowed:

```
{  
    "Version": "2012-10-17",
```

```
"Statement": [
    {
        "Effect": "Allow",
        "Action": "sagemaker:DescribeEndpoint",
        "Resource": "*"
    },
    {
        "Effect": "Allow",
        "Action": [
            "logs:CreateLogGroup",
            "logs:CreateLogStream",
            "logs:PutLogEvents"
        ],
        "Resource": "*"
    }
]
```

4. We create the policy and add it to the role:

```
with open('policy.json') as f:
    policy = f.read()
policy_name = 'Sagemaker-describe-endpoint'
response = iam.create_policy(
    PolicyName=policy_name,
    PolicyDocument=policy,
    Description='Allow the DescribeEndpoint API'
)
policy_arn = response['Policy']['Arn']
response = iam.attach_role_policy(
    RoleName=role_name,
    PolicyArn=policy_arn
)
```

The IAM setup is now complete.

Writing a Lambda function

We can now write a short Lambda function. It receives a JSON event as input, which stores the ARN of the endpoint being created by the EndpointStep step. It simply extracts the endpoint name from the ARN, creates a boto3 waiter, and waits until the endpoint is in service. The following screenshot shows the code in the Lambda console:

```

import boto3, time
def lambda_handler(event, context):
    print(event)
    endpoint_name = event['EndpointArn'].split('/')[-1]
    sm = boto3.client('sagemaker')
    waiter = sm.get_waiter('endpoint_in_service')
    waiter.wait(EndpointName=endpoint_name)
    return {
        'statusCode': 200,
        'body': endpoint_name
    }

```

Figure 12.18 – Our Lambda function

Let's deploy this function:

1. We create a deployment package for the Lambda function and upload it to S3:

```
$ zip -9 lambda.zip lambda.py
$ aws s3 cp lambda.zip s3://my-bucket
```

2. We create the function with a timeout of 15 minutes, the longest possible runtime for a Lambda function. Endpoints are typically deployed in less than 10 minutes, so this should be more than enough:

```

lambda_client = boto3.client('lambda')
response = lambda_client.create_function(
    FunctionName='sagemaker-wait-for-endpoint',
    Role=role_arn,
    Runtime='python3.6',
    Handler='lambda.lambda_handler',

```

```
Code={  
    'S3Bucket': bucket_name,  
    'S3Key': 'lambda.zip'  
},  
Description='Wait for endpoint to be in service',  
Timeout=900,  
MemorySize=128  
)
```

3. Now that the Lambda function has been created, we can easily add it to the existing workflow. We define a `LambdaStep` and add it to the endpoint branch. Its payload is the endpoint ARN, extracted from the output of the `EndpointStep`:

```
lambda_step = LambdaStep(  
    'Wait for endpoint to be in service',  
    parameters={  
        'FunctionName': 'sagemaker-wait-for-endpoint',  
        'Payload': { "EndpointArn.$": "$.EndpointArn" }  
    },  
    timeout_seconds=900  
)  
endpoint_branch = steps.Chain([  
    endpoint_config_step,  
    endpoint_step,  
    lambda_step  
])
```

4. Running the workflow again, we see in the following screenshot that this new step receives the endpoint ARN as input and waits for the endpoint to be in service:

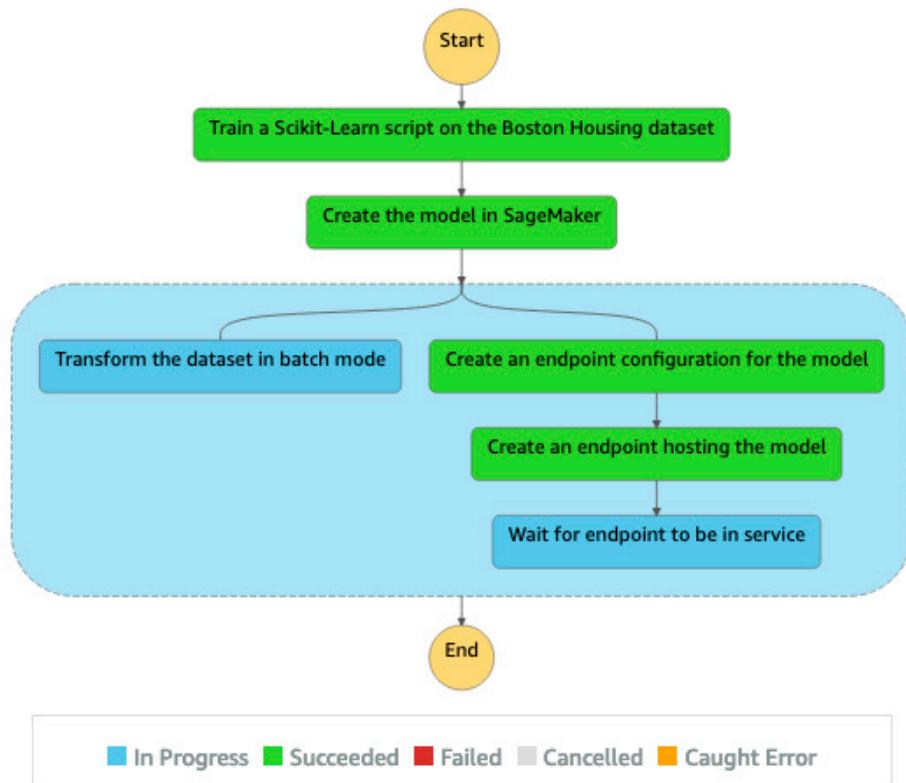


Figure 12.19 – Running the state machine with Lambda

There are many other ways you can use Lambda functions with SageMaker. You can extract training metrics, predict test sets on an endpoint, and more. The possibilities are endless.

Now, let's automate end-to-end workflows with Amazon SageMaker Pipelines.

Building end-to-end workflows with Amazon SageMaker Pipelines

Amazon SageMaker Pipelines lets us create and run end-to-end machine learning **workflows** based on SageMaker steps for training, tuning, batch transform, and processing scripts, using SageMaker APIs SDK that are very similar to the ones we used in Step Functions.

Compared to Step Functions, SageMaker Pipelines adds the following features:

- The ability to write, run, visualize and manage your workflows directly in SageMaker Studio, without having to jump to the AWS console.
- A **model registry**, which makes it easier to manage model versions, deploy only approved versions, and track **lineage**.
- **MLOps templates** – a collection of CloudFormation templates published via **AWS Service Catalog** that help you automate the deployment of your models. Built-in templates are provided, and you can add your own. You (or your Ops team) can learn more at <https://docs.aws.amazon.com/sagemaker/latest/dg/sagemaker-projects.html>.

Note

One thing that SageMaker Pipelines lacks is integration with other AWS services. At the time of writing, SageMaker Pipelines only supports **SQS**, whereas Step Functions supports many compute and big data services. With SageMaker Pipelines, the assumption is either that your training data has already been processed, or that you'll process it with SageMaker Processing steps.

Now that we know what SageMaker Pipelines is, let's run a complete example based on the Amazon Reviews dataset and the BlazingText algorithm we used in *Chapter 6, Training Natural Language Processing Models*, and *Chapter 10, Advanced Training Techniques*, putting together many of the services we learned about so far. Our pipeline will contain the following steps:

- A processing step, where we prepare the dataset with **SageMaker Processing**.
- An ingestion step, where we load the processed data set in **SageMaker Feature Store**.
- A dataset building step, where we use **Amazon Athena** to query the offline store and save a dataset to S3.

- A training step, where we train a BlazingText model on the dataset.
- A model creation step, where we save the trained model as a SageMaker model.
- A model registration step, where we add the model to the SageMaker Pipelines model registry.

In real life, you should not initially worry about automation. You should first experiment with Jupyter Notebooks and iterate on all these steps. Then, as the project matures, you should start automating each step, eventually assembling them as a pipeline.

My recommendation is to first automate each processing step, with individual SageMaker Processing jobs. Not only will this come in handy in the development phase, but it will also create a simple and step-by-step path to full automation. Indeed, once steps run fine with SageMaker Processing, it takes little effort to combine them with SageMaker Pipelines. In fact, you can use the exact same Python script. You'll only have to write code with the Pipelines SDK. As you'll see in a minute, it's very similar to the Processing SDK.

This is the approach I've followed with the following example. In the GitHub repository, you'll find SageMaker Processing notebooks for the data processing, ingestion, and dataset building steps, as well as another notebook for the end-to-end workflow. Here, we'll focus on the latter. Let's get started!

Defining workflow parameters

Just like CloudFormation templates, you can (and should) define parameters in your workflows. This makes them easier to reuse in other projects. Parameters can be strings, integers, and floats, with an optional default value.

1. We create parameters for the AWS region and for the instances we'd like to use for processing and training:

```
from sagemaker.workflow.parameters import
ParameterInteger, ParameterString
region = ParameterString(
    name='Region',
    default_value='eu-west-1')
processing_instance_type = ParameterString(
    name='ProcessingInstanceType',
    default_value='ml.m5.4xlarge')
processing_instance_count = ParameterInteger(
    name='ProcessingInstanceCount',
    default_value=1)
```

```

    training_instance_type = ParameterString(
        name='TrainingInstanceType',
        default_value='ml.p3.2xlarge')
    training_instance_count = ParameterInteger(
        name='TrainingInstanceCount',
        default_value=1)

```

2. We also create parameters for the location of input data, the model name, and the model status to set in the model registry (more on this later).

```

    input_data = ParameterString(name='InputData')
    model_name = ParameterString(name='ModelName')
    model_approval_status = ParameterString(
        name='ModelApprovalStatus',
        default_value='PendingManualApproval')

```

Now, let's define the data processing step.

Processing the dataset with SageMaker Processing

We reuse the processing script we wrote in *Chapter 6* (`preprocessing.py`).

1. We create a `SKLearnProcessor` object with the parameters we just defined:

```

from sagemaker.sklearn.processing import SKLearnProcessor
sklearn_processor = SKLearnProcessor(
    framework_version='0.23-1',
    role=role,
    instance_type=processing_instance_type,
    instance_count=processing_instance_count)

```

2. We then define the data processing step. Remember that it creates two outputs: one in BlazingText format, and one for ingestion in SageMaker Feature Store. As mentioned earlier, the SageMaker Pipelines syntax is extremely close to the SageMaker Processing syntax (inputs, outputs, and arguments).

```

from sagemaker.workflow.steps import ProcessingStep
from sagemaker.processing import ProcessingInput,
    ProcessingOutput
step_process = ProcessingStep(
    name='process-customer-reviews'

```

```
processor=sklearn_processor,
inputs=[
    ProcessingInput(source=input_data,
                    destination="/opt/ml/processing/input")],
outputs=[
    ProcessingOutput(output_name='bt_data',
                     source='/opt/ml/processing/output/bt'),
    ProcessingOutput(output_name='fs_data',
                     source='/opt/ml/processing/output/fs')],
code='preprocessing.py',
job_arguments=[
    '--filename',
    'amazon_reviews_us_Camera_v1_00.tsv.gz',
    '--library',
    'spacy']
)
```

Now, let's define the ingestion step.

Ingesting the dataset in SageMaker Feature Store with SageMaker Processing

We reuse the processing script we wrote in *Chapter 10* (`ingesting.py`).

1. We first define a name for the feature group:

```
feature_group_name = 'amazon-reviews-feature-group-' +
strftime('%d-%H-%M-%S', gmtime())
```

2. We then define a processing step, setting the data input to the output of the first processing job. To illustrate step chaining, we define an output pointing to a file saved by the script, which contains the name of the feature group:

```
step_ingest = ProcessingStep(
    name='ingest-customer-reviews',
    processor=sklearn_processor,
    inputs=[
        ProcessingInput(
            source=
```

```
step_process.properties.ProcessingOutputConfig
    .Outputs['fs_data'].S3Output.S3Uri,
    destination="/opt/ml/processing/input")],
outputs = [
    ProcessingOutput(
        output_name='feature_group_name',
        source='/opt/ml/processing/output/')),
    code='ingesting.py',
    job_arguments=[
        '--region', region,
        '--bucket', bucket,
        '--role', role,
        '--feature-group-name', feature_group_name,
        '--max-workers', '32']
)
```

Now, let's take care of the dataset building step.

Building a dataset with Amazon Athena and SageMaker Processing

We reuse the processing script we wrote in *Chapter 10* (*querying.py*).

We set the input to the output of the ingestion step, in order to retrieve the name of the feature group. We also define two outputs for the training and validation datasets:

```
step_build_dataset = ProcessingStep(
    name='build-dataset',
    processor=sklearn_processor,
    inputs=[

        ProcessingInput(
            source=
                step_ingest.properties.ProcessingOutputConfig
                .Outputs['feature_group_name'].S3Output.S3Uri,
            destination='/opt/ml/processing/input')],

    outputs=[

        ProcessingOutput(
            output_name='training',
```

```
        source='/opt/ml/processing/output/training') ,  
    ProcessingOutput(  
        output_name='validation',  
        source='/opt/ml/processing/output/validation')] ,  
    code='querying.py',  
    job_arguments=[  
        '--region', region,  
        '--bucket', bucket,]  
)
```

Now, let's move on to the training step.

Training a model

No surprises here:

1. We define an `Estimator` module for this job:

```
container = image_uris.retrieve(  
    'blazingtext',  
    str(region)) # region is a ParameterString  
prefix = 'blazing-text-amazon-reviews'  
s3_output = 's3://{}//{}//output/'.format(bucket, prefix)  
bt = Estimator(container,  
                role,  
                instance_count=training_instance_count,  
                instance_type=training_instance_type,  
                output_path=s3_output)  
bt.set_hyperparameters(mode='supervised')
```

2. We then define the training step, passing the training and validation datasets as inputs:

```
from sagemaker.workflow.steps import TrainingStep  
from sagemaker.inputs import TrainingInput  
step_train = TrainingStep(  
    name='train-blazing-text',  
    estimator=bt,  
    inputs={
```

```
'train': TrainingInput(s3_data=
    step_build_dataset.properties.ProcessingOutputConfig
    .Outputs['training'].S3Output.S3Uri,
    content_type='text/plain'),
    'validation': TrainingInput(s3_data=
    step_build_dataset.properties.ProcessingOutputConfig
    .Outputs['validation'].S3Output.S3Uri,
    content_type='text/plain')
}
)
```

Now, let's take care of the model creation and model registration steps (the last ones in the pipeline).

Creating and registering a model in SageMaker Pipelines

Once the model has been trained, we need to create it as a SageMaker model and register it in the model registry.

1. We create the model, passing the location of the training container and of the model artifact:

```
from sagemaker.model import Model
from sagemaker.workflow.steps import CreateModelStep
model = Model(
    image_uri=container,
    model_data=step_train.properties
        .ModelArtifacts.S3ModelArtifacts,
    sagemaker_session=session,
    name=model_name,    # workflow parameter
    role=role)
step_create_model = CreateModelStep(
    name='create-model',
    model=model,
    inputs=None)
```

2. We then register the model in the model registry, passing the list of allowed instance types for deployment, as well as the approval status. We associate it to a model package group that will hold this model, as well as further versions we train in the future:

```
from sagemaker.workflow.step_collections import
RegisterModel
step_register = RegisterModel(
    name='register-model',
    estimator=bt,
    model_data=step_train.properties.ModelArtifacts
        .S3ModelArtifacts,
    content_types=['text/plain'],
    response_types=['application/json'],
    inference_instances=['ml.t2.medium'],
    transform_instances=['ml.m5.xlarge'],
    model_package_group_name='blazing-text-on-amazon-
customer-reviews-package',
    approval_status=model_approval_status
)
```

All the steps are now defined, so let's assemble them in a pipeline.

Creating a pipeline

We simply put together all the steps and their parameters. Then, we create the pipeline (or update it if it existed previously):

```
from sagemaker.workflow.pipeline import Pipeline
pipeline_name = 'blazing-text-amazon-customer-reviews'
pipeline = Pipeline(
    name=pipeline_name,
    parameters=[region, processing_instance_type, processing_
instance_count, training_instance_type, training_instance_
count, model_approval_status, input_data, model_name],
    steps=[step_process, step_ingest, step_build_dataset, step_
train, step_create_model, step_register])
pipeline.upsert(role_arn=role)
```

We're all set. Let's run our pipeline!

Running a pipeline

It takes a single line of code to fire up a pipeline execution:

1. We assign values to the data location and model name parameters (the other ones have default values):

```
execution = pipeline.start(  
    parameters=dict(  
        InputData=input_data_uri,  
        ModelName='blazing-text-amazon-reviews'  
    )
```

2. In SageMaker Studio, we go **SageMaker resources / Pipelines**, and we see the pipeline executing, as shown in the next screenshot:

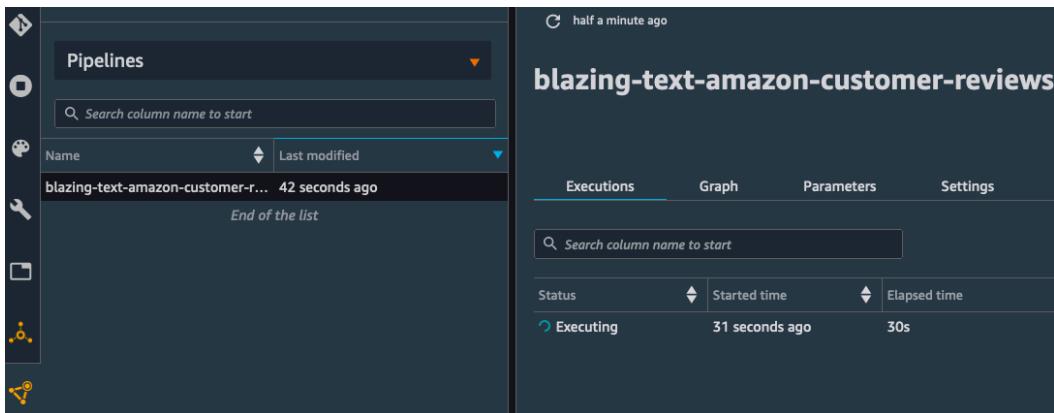


Figure 12.20 – Executing a pipeline

After an hour and a half, the pipeline is complete, as shown in the next screenshot:

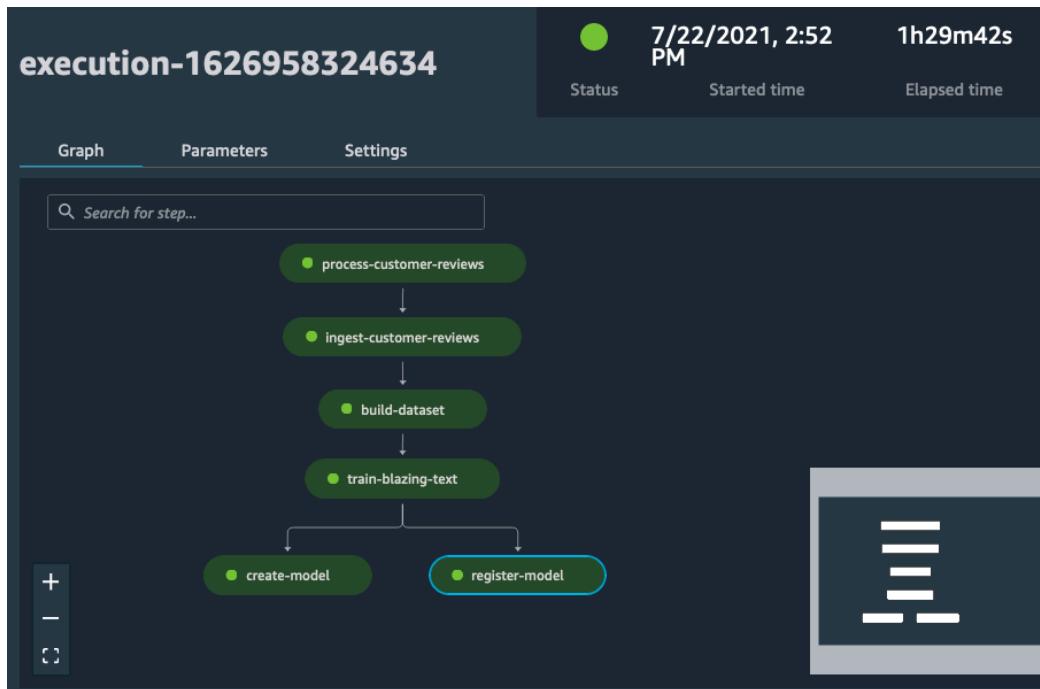


Figure 12.21 – Visualizing a pipeline

- Finally, for each step of the pipeline, we can see the lineage of all artifacts:

```
from sagemaker.lineage.visualizer import  
LineageTableVisualizer  
viz = LineageTableVisualizer(session)  
for execution_step in reversed(execution.list_steps()):  
    print(execution_step)  
    display(viz.show(  
        pipeline_execution_step=execution_step))
```

For example, the output for the training step is shown in the next image. We see exactly which datasets and which container were used to train the model:

('StepName': 'train-blazing-text', 'StartTime': datetime.datetime(2021, 7, 22, 14, 15, 51, 179000, tzinfo=tzlocal()), 'EndTime': datetime.datetime(2021, 7, 22, 14, 21, 44, 405000, tzinfo=tzlocal()), 'StepStatus': 'Succeeded', 'Metadata': {'TrainingJob': {'Arn': 'arn:aws:sagemaker:eu-west-1:613904931467:training-job/pipelines-rend1ydhfvz-train-blazing-text-r7h0n0amxb'}}}					
	Name/Source	Direction	Type	Association Type	Lineage Type
0	s3://...021-07-22-12-52-03-988/output/validation	Input	DataSet	ContributedTo	artifact
1	s3://...-2021-07-22-12-52-03-988/output/training	Input	DataSet	ContributedTo	artifact
2	68538...cr.eu-west-1.amazonaws.com/blazingtext:1	Input	Image	ContributedTo	artifact
3	s3://...zing-text-r7H0N0amXb/output/model.tar.gz	Output	Model	Produced	artifact

Figure 12.22 – Viewing the lineage for the training step

Let's see how we can deploy this model.

Deploying a model from the model registry

Going to **SageMaker resources / Model registry**, we also see that the model has been registered in the model registry, as shown in the next screenshot. If we train further versions of the model, they will also appear here:

Name	Created
blazing-text-on-amazon-customer-reviews-package	1 hour ago

Versions	Settings
1	None

Figure 12.23 – Viewing a model in the model registry

As its status is Pending, it can't be deployed for now. We need to change it to Approved in order to allow deployment. This is a safe way to guarantee that only good models are deployed, once all appropriate tests have been performed.

We right-click on the model and select **Update model version status**. We then set the model status to Approved. We also note the model ARN, which is visible in the **Settings** tab.

Now, we can deploy and test the model:

1. Back in our Jupyter Notebook, we create a ModelPackage object pointing at the model version we'd like to deploy:

```
from sagemaker import ModelPackage
model_package_arn = 'arn:aws:sagemaker:eu-west-1:123456789012:model-package/blazing-text-on-amazon-customer-reviews-package/1'
model = sagemaker.ModelPackage(
    role = role,
    model_package_arn = model_package_arn)
```

2. We call `deploy()` as usual:

```
model.deploy(
    initial_instance_count = 1,
    instance_type = 'ml.t2.medium',
    endpoint_name='blazing-text-on-amazon-reviews')
```

3. We create a Predictor and send a test sample for prediction:

```
from sagemaker.predictor import Predictor
bt_predictor = Predictor(
    endpoint_name='blazing-text-on-amazon-reviews',
    serializer=
        sagemaker.serializers.JSONSerializer(),
    deserializer=
        sagemaker.deserializers.JSONDeserializer())
instances = [' I really love this camera , it takes
amazing pictures . ']
payload = {'instances': instances,
           'configuration': {'k': 3}}
response = bt_predictor.predict(payload)
print(response)
```

This prints out the probabilities for all three classes:

```
[{'label': ['__label_positive__', '__label_neutral__',
'__label_negative__'],
'prob': [0.9999945163726807, 2.51355941145448e-05,
1.0307396223652177e-05}],
```

-
4. Once we're done, we can delete the endpoint.

Note

For a full clean-up, you should also delete the pipeline, the feature store, and the model package group. You'll find a clean-up notebook in the GitHub repository.

As you can see, SageMaker Pipelines provides you with robust and powerful tools to build, run, and track end-to-end machine learning workflows. These tools are nicely integrated in SageMaker Studio, which should help you to be more productive and get high-quality models in production quicker

Summary

In this chapter, you first learned how to deploy and update endpoints with AWS CloudFormation. You also saw how it can be used to implement canary deployment and blue-green deployment.

Then, you learned about the AWS CDK, an SDK specifically built to easily generate and deploy CloudFormation templates using a variety of programming languages.

Finally, you built complete end-to-end machine learning workflows with AWS Step Functions and Amazon SageMaker Pipelines.

In the next and final chapter, you'll learn about additional SageMaker capabilities that help you optimize the cost and performance of predictions.

13

Optimizing Prediction Cost and Performance

In the previous chapter, you learned how to automate training and deployment workflows.

In this final chapter, we'll focus on optimizing cost and performance for prediction infrastructure, which typically accounts for 90% of the machine learning spend by AWS customers. This number may come as a surprise, until we realize that a model built by a single training job may end on multiple endpoints running 24/7 on a large scale.

Hence, great care must be taken to optimize your prediction infrastructure to ensure that you get the most bang for your buck!

This chapter features the following topics:

- Autoscaling an endpoint
- Deploying a multi-model endpoint
- Deploying a model with Amazon Elastic Inference
- Compiling models with Amazon SageMaker Neo

Technical requirements

You will need an AWS account to run the examples included in this chapter. If you haven't got one already, please point your browser at <https://aws.amazon.com/getting-started/> to create it. You should also familiarize yourself with the AWS Free Tier (<https://aws.amazon.com/free/>), which lets you use many AWS services for free within certain usage limits.

You will need to install and configure the AWS **Command Line Interface (CLI)** for your account (<https://aws.amazon.com/cli/>).

You will need a working Python 3.x environment. Installing the Anaconda distribution (<https://www.anaconda.com/>) is not mandatory but strongly encouraged, as it includes many projects that we will need (Jupyter, pandas, numpy, and more).

Code examples included in the book are available on GitHub at <https://github.com/PacktPublishing/Learn-Amazon-SageMaker-second-edition>. You will need to install a Git client to access them (<https://git-scm.com/>).

Autoscaling an endpoint

Autoscaling has long been the most important technique in adjusting infrastructure size for incoming traffic, and it's available for SageMaker endpoints. However, it's based on **Application Auto Scaling** and not on **EC2 Auto Scaling** (<https://docs.aws.amazon.com/autoscaling/application/userguide/what-is-application-auto-scaling.html>), although the concepts are extremely similar.

Let's set up autoscaling for the **XGBoost** model we trained on the Boston Housing dataset:

1. We first create an **endpoint configuration**, and we use it to build the endpoint. Here, we use the m5 instance family; t2 and t3 are not recommended for autoscaling as their burstable behavior makes it harder to measure their real load:

```
model_name = 'sagemaker-xgboost-2020-06-09-08-33-24-782'
endpoint_config_name = 'xgboost-one-model-epc'
endpoint_name = 'xgboost-one-model-ep'
production_variants = [
    'VariantName': 'variant-1',
    'ModelName': model_name,
    'InitialInstanceCount': 2,
    'InitialVariantWeight': 1,
    'InstanceType': 'ml.m5.large'}]
```

```
sm.create_endpoint_config(
    EndpointConfigName=endpoint_config_name,
    ProductionVariants=production_variants)
sm.create_endpoint(
    EndpointName=endpoint_name,
    EndpointConfigName=endpoint_config_name)
```

- Once the endpoint is in service, we define the target value that we want to scale on, namely the number of instances backing the endpoint:

```
app = boto3.client('application-autoscaling')
app.register_scalable_target(
    ServiceNamespace='sagemaker',
    ResourceId=
        'endpoint/xgboost-one-model-ep/variant/variant-1',
    ScalableDimension=
        'sagemaker:variant:DesiredInstanceCount',
    MinCapacity=2,
    MaxCapacity=10)
```

- Then, we apply a scaling policy for this target value:

```
policy_name = 'xgboost-scaling-policy'
app.put_scaling_policy(
    PolicyName=policy_name,
    ServiceNamespace='sagemaker',
    ResourceId=
        'endpoint/xgboost-one-model-ep/variant/variant-1',
    ScalableDimension=
        'sagemaker:variant:DesiredInstanceCount',
    PolicyType='TargetTrackingScaling',
```

4. We use the only built-in metric available in SageMaker, `SageMakerVariantInvocationsPerInstance`. We could also define a custom metric if we wanted to. We set the metric threshold at 1,000 invocations per minute. This is a bit of an arbitrary value. In real life, we would run a load test on a single instance and monitor model latency in order to find the actual value that ought to trigger autoscaling. You can find more information at <https://docs.aws.amazon.com/sagemaker/latest/dg/endpoint-scaling-loadtest.html>. We also define a 60-second cooldown for scaling in and out, a good practice for smoothing out transient traffic drops and peaks:

```
TargetTrackingScalingPolicyConfiguration={  
    'TargetValue': 1000.0,  
    'PredefinedMetricSpecification': {  
        'PredefinedMetricType':  
            'SageMakerVariantInvocationsPerInstance'  
    },  
    'ScaleInCooldown': 60,  
    'ScaleOutCooldown': 60  
}  
)
```

5. As shown in the following screenshot, autoscaling is now configured on the endpoint:

Endpoint runtime settings										Update weights	Update instance count	Configure auto scaling
	Variant name	Current weight	Desired weight	Instance type	Elastic Inference	Current instance count	Desired instance count	Instance min - max	Automatic scaling			
<input checked="" type="radio"/>	variant-1	1	1	ml.m5.large	-	2	2	2 - 10	Yes			

Figure 13.1 – Viewing autoscaling

6. Using an infinite loop, we send some traffic to the endpoint:

```
test_sample = '0.00632, 18.00, 2.310, 0, 0.5380, 6.5750,
65.20, 4.0900, 1, 296.0, 15.30, 396.90, 4.98'
smrt=boto3.Session().client(service_name='runtime.sagemaker')
while True:
    smrt.invoke_endpoint(EndpointName=endpoint_name,
                         ContentType='text/csv',
                         Body=test_sample)
```

7. Looking at the **CloudWatch** metrics for the endpoints, as shown in the following screenshot, we see that invocations per instance exceed the threshold we defined: 1.42k versus 1k:

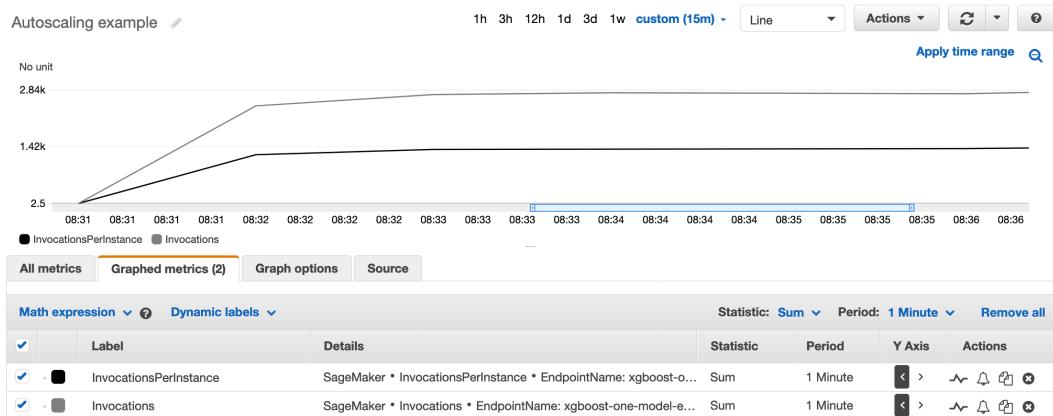


Figure 13.2 – Viewing CloudWatch metrics

8. Autoscaling quickly kicks in and decides to add another instance, as visible in the following screenshot. If the load was even higher, it could decide to add several instances at once:

Endpoint runtime settings									Update weights	Update instance count	Configure auto scaling
	Variant name	Current weight	Desired weight	Instance type	Elastic Inference	Current instance count	Desired instance count	Instance min - max	Automatic scaling		
<input type="radio"/>	variant-1	1	1	m1.m5.large	-	2	3	n/a	n/a		

Figure 13.3 – Viewing autoscaling

9. A few minutes later, the extra instance is in service, and invocations per instance are now below the threshold (935 versus 1,000):

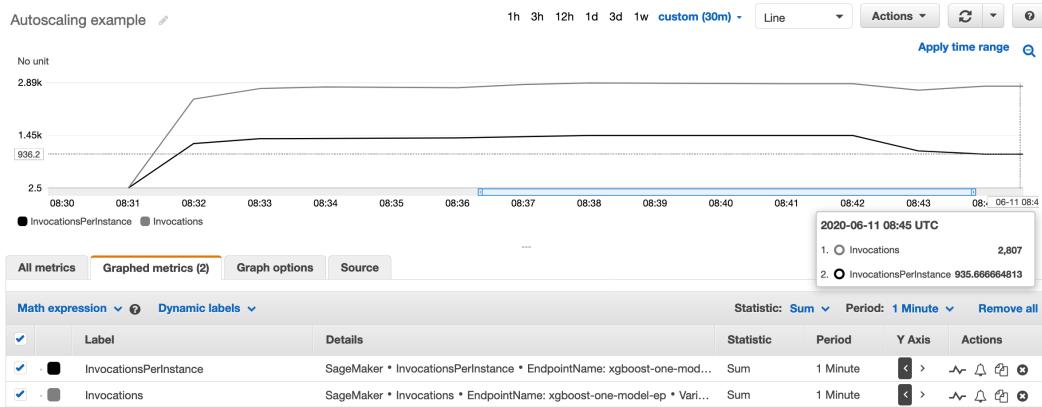


Figure 13.4 – Viewing CloudWatch metrics

A similar process takes place when traffic decreases.

10. Once we're finished, we delete everything:

```
app.delete_scaling_policy(
    PolicyName=policy_name,
    ServiceNamespace='sagemaker',
    ScalableDimension='sagemaker:variant
        :DesiredInstanceCount',
    ResourceId='endpoint/xgboost-one-model-ep/variant/
        variant-1')
    sm.delete_endpoint(EndpointName=endpoint_name)
    sm.delete_endpoint_config(
        EndpointConfigName=endpoint_config_name)
```

Setting up autoscaling is easy. It helps you automatically adapt your prediction infrastructure and the associated costs to changing business conditions.

Now, let's study another technique that you'll find extremely useful when you're dealing with a very large number of models: **multi-model endpoints**.

Deploying a multi-model endpoint

Multi-model endpoints are useful when you're dealing with a large number of models where it wouldn't make sense to deploy to individual endpoints. For example, imagine a SaaS company building a regression model for each one of their 10,000 customers. Surely, they wouldn't want to manage (and pay for) 10,000 endpoints!

Understanding multi-model endpoints

A multi-model endpoint can serve CPU-based predictions from an arbitrary number of models stored in S3 (GPUs are not supported at the time of writing). The path of the model artifact to use is passed in each prediction request. Models are loaded and unloaded dynamically, according to usage and the amount of memory available on the endpoint. Models can also be added to, or removed from, the endpoint by simply copying or deleting artifacts in S3.

In order to serve multiple models, your inference container must implement a specific set of APIs that the endpoint will invoke: LOAD MODEL, LIST MODEL, GET MODEL, UNLOAD MODEL, and INVOKE MODEL. You can find the details at <https://docs.aws.amazon.com/sagemaker/latest/dg/mms-container-apis.html>.

At the time of writing, the latest built-in containers for **scikit-learn**, **TensorFlow**, **Apache MXNet**, and **PyTorch** natively support these APIs. The **XGBoost**, **kNN**, **Linear Learner**, and **Random Cut Forest** built-in algorithms also support them.

For other algorithms and frameworks, your best option is to build a custom container that includes the **SageMaker Inference Toolkit**, as it already implements the required APIs (<https://github.com/aws/sagemaker-inference-toolkit>).

This toolkit is based on the multi-model server (<https://github.com/awslabs/multi-model-server>), which you could also use directly from the CLI to serve predictions from multiple models. You can find more information at <https://docs.aws.amazon.com/sagemaker/latest/dg/build-multi-model-build-container.html>.

Building a multi-model endpoint with scikit-learn

Let's build a multi-model endpoint with **scikit-learn**, hosting models trained on the Boston Housing dataset. This is only supported on scikit-learn 0.23-1 and above:

1. We upload the dataset to S3:

```
import sagemaker, boto3
sess = sagemaker.Session()
```

```
bucket = sess.default_bucket()
prefix = 'sklearn-boston-housing-mme'
training = sess.upload_data(path='housing.csv',
                             key_prefix=prefix +
                             '/training')
output = 's3://{}//{}/output/'.format(bucket,prefix)
```

2. We train three models with a different test size, storing their names in a dictionary. Here, we use the latest version of scikit-learn, the first one to support multi-model endpoints:

```
from sagemaker.sklearn import SKLearn
jobs = {}
for test_size in [0.2, 0.1, 0.05]:
    sk = SKLearn(entry_point=
                  'sklearn-boston-housing.py',
                  role=sagemaker.get_execution_role(),
                  framework_version='0.23-1',
                  instance_count=1,
                  instance_type='ml.m5.large',
                  output_path=output,
                  hyperparameters={ 'normalize': True,
                                    'test-size': test_size })
    sk.fit({'training':training}, wait=False)
    jobs[sk.latest_training_job.name] = {}
    jobs[sk.latest_training_job.name]['test-size'] =
        test_size
```

3. We find the S3 URI of the model artifact along with its prefix:

```
import boto3
sm = boto3.client('sagemaker')
for j in jobs.keys():
    job = sm.describe_training_job(TrainingJobName=j)
    jobs[j]['artifact'] =
        job['ModelArtifacts']['S3ModelArtifacts']
    jobs[j]['key'] = '/'.join(
        job['ModelArtifacts']['S3ModelArtifacts']
        .split '/') [3:]
```

4. We delete any previous model stored in S3:

```
%%sh -s "$bucket" "$prefix"
aws s3 rm --recursive s3://$1/$2/models
```

5. We copy the three model artifacts to this location:

```
s3 = boto3.client('s3')
for j in jobs.keys():
    copy_source = { 'Bucket': bucket,
                    'Key': jobs[j]['key'] }
    s3.copy_object(CopySource=copy_source,
                    Bucket=bucket,
                    Key=prefix+'/models/'+j+'.tar.gz')
response = s3.list_objects(Bucket=bucket,
                            Prefix=prefix+'/models/')
for o in response['Contents']:
    print(o['Key'])
```

This lists the model artifacts:

```
sklearn-boston-housing-mme/models/sagemaker-scikit-
learn-2021-09-01-07-52-22-679
sklearn-boston-housing-mme/models/sagemaker-scikit-
learn-2021-09-01-07-52-26-399
sklearn-boston-housing-mme/models/sagemaker-scikit-
learn-2021-09-01-08-05-33-229
```

6. We define the name of the script and the S3 location where we'll upload the code archive. Here, I'm passing the training script, which includes a `model_fn()` function to load the model. This is the only function that will be used to serve predictions:

```
script = 'sklearn-boston-housing.py'  
script_archive = 's3://{}//{}//source/source.tar.gz'.  
format(bucket, prefix)
```

7. We create the code archive and we upload it to S3:

```
%%sh -s "$script" "$script_archive"  
tar cvfz source.tar.gz $1  
aws s3 cp source.tar.gz $2
```

8. We create the multi-model endpoint with the `create_model()` API and we set the `Mode` parameter accordingly:

```
import time  
model_name = prefix+'-'+time.strftime("%Y-%m-%d-%H-%M-%S", time.gmtime())  
response = sm.create_model(  
    ModelName = model_name,  
    ExecutionRoleArn = role,  
    Containers = [  
        'Image': sk.image_uri,  
        'ModelDataUrl': 's3://{}//{}//models/'.format(bucket,  
                                                       prefix),  
        'Mode': 'MultiModel',  
        'Environment': {  
            'SAGEMAKER_PROGRAM' : script,  
            'SAGEMAKER_SUBMIT_DIRECTORY' : script_archive  
        }  
    ]  
)
```

9. We create the endpoint configuration as usual:

```
epc_name = prefix+'-epc'+time.strftime("%Y-%m-%d-%H-%M-%S", time.gmtime())
response = sm.create_endpoint_config(
    EndpointConfigName = epc_name,
    ProductionVariants=[{
        'InstanceType': 'ml.m5.large',
        'InitialInstanceCount': 1,
        'InitialVariantWeight': 1,
        'ModelName': model_name,
        'VariantName': 'variant-1'}]
)
```

10. We create the endpoint as usual:

```
ep_name = prefix+'-ep'+time.strftime("%Y-%m-%d-%H-%M-%S",
time.gmtime())
response = sm.create_endpoint(
    EndpointName=ep_name,
    EndpointConfigName=epc_name)
```

11. Once the endpoint is in service, we load samples from the dataset and convert them to a numpy array:

```
import pandas as pd
import numpy as np
from io import BytesIO
data = pd.read_csv('housing.csv')
payload = data[:10].drop(['medv'], axis=1)
buffer = BytesIO()
np.save(buffer, payload.values)
```

12. We predict these samples with all three models, passing the name of the model to use for each prediction request, such as **sagemaker-scikit-learn-2021-09-01-08-05-33-229**:

```
smrt = boto3.client('runtime.sagemaker')
for j in jobs.keys():
    model_name=j+'.tar.gz'
```

```
response = smrt.invoke_endpoint(  
    EndpointName=ep_name,  
    TargetModel=model_name,  
    Body=buffer.getvalue(),  
    ContentType='application/x-npy')  
print(response['Body'].read())
```

13. We could train more models, copy their artifacts to the same S3 location, and use them directly without recreating the endpoint. We could also delete those models we don't need.
14. Once we're finished, we delete the endpoint:

```
sm.delete_endpoint(EndpointName=ep_name)  
sm.delete_endpoint_config(EndpointConfigName=epc_name)
```

As you can see, multi-model endpoints are a great way to serve as many models as you'd like from a single endpoint, and setting them up isn't difficult.

In the next section, we're going to study another cost optimization technique that can help you save a lot of money on GPU prediction: **Amazon Elastic Inference**.

Deploying a model with Amazon Elastic Inference

When deploying a model, you have to decide whether it should run on a CPU instance or on a GPU instance. In some cases, there isn't much of a debate. For example, some algorithms simply don't benefit from GPU acceleration, so they should be deployed to CPU instances. At the other end of the spectrum, complex deep learning models for computer vision or natural language processing run best on GPUs.

In many cases, the situation is not that clear-cut. First, you should know what the maximum predicted latency is for your application. If you're predicting a click-through rate for a real-time ad tech application, every millisecond counts; if you're predicting customer churn in a back-office application, not so much.

In addition, even models that could benefit from GPU acceleration may not be large and complex enough to fully utilize the thousands of cores available on a modern GPU. In such scenarios, you're stuck between a rock and a hard place: deploying on CPU would be a little slow for your needs, and deploying on GPU wouldn't be cost-effective.

This is the problem that Amazon Elastic Inference aims to solve (<https://aws.amazon.com/machine-learning/elastic-inference/>). It lets you attach fractional GPU acceleration to any EC2 instance, including notebook instances and endpoint instances. **Accelerators** come in three different sizes (medium, large, and extra large), which let you find the best cost-performance ratio for your application.

Elastic Inference is available for **TensorFlow**, **PyTorch**, and **Apache MXNet**. You can use it in your own code running on EC2 instances, thanks to AWS extensions available in the **Deep Learning AMI**. You can also use it with **Deep Learning Containers**. More information is available at <https://docs.aws.amazon.com/elastic-inference/latest/developerguide/working-with-ei.html>.

Of course, **Elastic Inference** is available on SageMaker. You can attach an accelerator to a **Notebook Instance** at creation time and work with the built-in **conda** environments. You can also attach an accelerator to an endpoint, and we'll show you how to do this in the next example.

Deploying a model with Amazon Elastic Inference

Let's reuse the **Image Classification** model we trained on dog and cat images in *Chapter 5, Training Computer Vision Models*. This is based on an 18-layer **ResNet** model, which is pretty small as far as convolution neural networks are concerned:

1. Once the model has been trained, we deploy it as usual on two endpoints: one backed by an `m1.c5.large` instance and another one backed by an `m1.g4dn.xlarge` instance, the most cost-effective GPU instance available on SageMaker:

```
import time
endpoint_name = 'c5-'+time.strftime("%Y-%m-%d-%H-%M-%S",
time.gmtime())
c5_predictor = ic.deploy(initial_instance_count=1,
instance_type='ml.c5.large',
endpoint_name=endpoint_name,
wait=False)
endpoint_name = 'g4-'+time.strftime("%Y-%m-%d-%H-%M-%S",
time.gmtime())
g4_predictor = ic.deploy(
initial_instance_count=1,
instance_type='ml.g4dn.xlarge',
endpoint_name=endpoint_name,
wait=False)
```

2. We then download a test image, predict it 1,000 times, and measure the total time it takes:

```
with open(file_name, 'rb') as f:  
    payload = f.read()  
    payload = bytearray(payload)  
def predict_images(predictor, iterations=1000):  
    total = 0  
    for i in range(0, iterations):  
        tick = time.time()  
        response = runtime.invoke_endpoint(  
            EndpointName=predictor.endpoint_name,  
            ContentType='application/x-image',  
            Body=payload)  
        tock = time.time()  
        total += tock-tick  
    return total/iterations  
predict_images(c5_predictor)  
predict_images(g4_predictor)
```

3. The results are shown in the next table (us-east-1 prices):

Instance type	Acceleration	Cost	Time per image	Images/s
ml.c5.large	none	\$0.102	74 ms	13.5
ml.g4dn.xlarge	1 NVIDIA T4 GPU	\$0.736	42 ms	23.6

Unsurprisingly, the GPU instance is about twice as fast. Yet, the CPU instance is more cost-effective, as it's over four times less expensive. Putting it another way, you could run your endpoint with four CPU instances instead of one GPU instance and get more throughput for the same cost. This shows why it's so important to understand the latency requirement of your application. "Fast" and "slow" are very relative concepts!

4. We then deploy the same model on three more endpoints backed by `ml.c5.large` instances, accelerated by a medium, large, and extra-large **Elastic Inference accelerator**. All it takes is an extra parameter in the `deploy()` API. Here's the code for the medium endpoint:

```
endpoint_name = 'c5-medium-'  
+time.strftime("%Y-%m-%d-%H-%M-%S", time.gmtime())
```

```
c5_medium_predictor = ic.deploy(
    initial_instance_count=1,
    instance_type='ml.c5.large',
    accelerator_type='ml.eia2.medium',
    endpoint_name=endpoint_name,
    wait=False)
predict_images(c5_medium_predictor)
```

You can see the results in the following table:

Instance type	Acceleration	Cost	Time per image	Images/s
ml.c5.large + ml.eia2.medium	1 FP-32 TFLOPS 8 FP-16 TFLOPS	\$0.270	70.3 ms	14.3
ml.c5.large + ml.eia2.large	2 FP-32 TFLOPS 16 FP-16 TFLOPS	\$0.438	68.3 ms	14.7
ml.c5.large + ml.eia2.xlarge	4 FP-32 TFLOPS 32 FP-16 TFLOPS	\$0.680	63 ms	15.9

We get up to 20% speed-up compared to the naked CPU endpoint, and the cost is lower than if we used a GPU instance. Let's keep tweaking:

5. Attentive readers will have noticed that the previous tables include teraFLOP values for both 32-bit and 16-bit floating-point values. Indeed, either one of these data types may be used to store model parameters. Looking at the documentation for the image classification algorithm, we see that we can actually select a data type with the `precision_dtype` parameter and that the default value is `float32`. This begs the question: would the results differ if we trained our model in `float16` mode? There's only one way to know, isn't there?

```
ic.set_hyperparameters(
    num_layers=18,
    use_pretrained_model=0,
    num_classes=2,
    num_training_samples=22500,
    mini_batch_size=128,
```

```
precision_dtype='float16',  
epochs=10)
```

6. Training again, we hit the same model accuracy as in `float32` mode. Deploying benchmarking again, we get the following results:

Instance type	Acceleration	Cost	Time	Images/s
ml.c5.large	None	\$0.102	75.9 ms	13.3
ml.g4dn.xlarge	1 NVIDIA T4 GPU	\$0.736	43.7 ms	22.9
ml.c5.large + ml.eia2.medium	1 FP-32 TFLOPS 8 FP-16 TFLOPS	\$0.270	70.1 ms	14.3
ml.c5.large + ml.eia2.large	2 FP-32 TFLOPS 16 FP-16 TFLOPS	\$0.438	62 ms	16.4
ml.c5.large + ml.eia2.xlarge	4 FP-32 TFLOPS 32 FP-16 TFLOPS	\$0.680	59.3 ms	16.9

No meaningful difference is visible on the naked instances. Predicting with an **FP-16** model on the large and extra-large accelerators helps us speed up predictions by about 10% compared to the **FP-32** model. Pretty good! This performance level is definitely a nice upgrade compared to a naked CPU instance, and it's cost-effective compared to a GPU instance.

In fact, switching a single endpoint instance from `ml.g4dn.xlarge` to `ml.c5.large+ml.eia2.large` would save you $(\$0.736 - \$0.438) \times 24 \times 30 = \214 dollars per month. That's serious money!

As you can see, Amazon Elastic Inference is extremely easy to use, and it gives you additional deployment options. Once you've defined the prediction latency requirement for your application, you can quickly experiment and find the best cost-performance ratio.

Now, let's talk about another SageMaker capability that lets you compile models for a specific hardware architecture: **Amazon Neo**.

Compiling models with Amazon SageMaker Neo

Embedded software developers have long learned how to write highly optimized code that both runs fast and uses hardware resources frugally. In theory, the same techniques could also be applied to optimize machine learning predictions. In practice, this is a daunting task given the complexity of machine learning libraries and models.

This is the problem that Amazon SageMaker Neo aims to solve.

Understanding Amazon SageMaker Neo

Amazon Neo has two components: a model compiler that optimizes models for the underlying hardware, and a small runtime named **Deep Learning Runtime (DLR)**, used to load optimized models and run predictions (<https://aws.amazon.com/sagemaker/neo>).

Amazon SageMaker Neo can compile models trained with the following:

- **Two built-in algorithms:** XGBoost and Image Classification.
- **Built-in frameworks:** TensorFlow, PyTorch, and Apache MXNet, as well as models in **ONNX** format. Many operators are supported, and you can find the full list at <https://aws.amazon.com/releasenotes/sagemaker-neo-supported-frameworks-and-operators>.

Training takes place as usual, using your estimator of choice. Then, using the `compile_model()` API, we can easily compile the model for one of these hardware targets:

- Amazon EC2 instances of the following families: c4, c5, m4, m5, p2, p3, and `inf1` (which we'll discuss later in this chapter), as well as Lambda
- AI-powered cameras: AWS DeepLens and Acer aiSage
- NVIDIA Jetson platforms: TX1, TX2, Nano, and Xavier
- Raspberry Pi
- System-on-chip platforms from Rockchip, Qualcomm, Ambarella, and more

Model compilation performs both architecture optimizations (such as fusing layers) and code optimizations (replacing machine learning operators with hardware-optimized versions). The resulting artifact is stored in S3 and contains both the original model and its optimized form.

The DLR is then used to load the model and predict with it. Of course, it can be used in a standalone fashion, such as on a Raspberry Pi. You can find installation instructions at <https://neo-ai-dlr.readthedocs.io>. As the DLR is open source (<https://github.com/neo-ai/neo-ai-dlr>), you can also build it from source and – why not? – customize it for your own hardware platform!

When it comes to using the DLR with SageMaker, things are much simpler. SageMaker provides built-in containers with Neo support, and these are the ones you should use to deploy models compiled with Neo (as already mentioned, the training container remains unchanged). You can find a list of Neo-enabled containers at <https://docs.aws.amazon.com/sagemaker/latest/dg/neo-deployment-hosting-services-cli.html>.

Last, but not least, one of the benefits of the DLR is its small size. For example, the Python package for p2 and p3 instances is only 5.4 MB in size, orders of magnitude smaller than your typical deep learning library and its dependencies. This is obviously critical for embedded environments, and it's also welcome on SageMaker as containers will be smaller too.

Let's reuse our image classification example and see whether Neo can speed it up.

Compiling and deploying an image classification model on SageMaker

In order to give Neo a little more work, we train a 50-layer ResNet this time. Then, we'll compile it, deploy it to an endpoint, and compare it with the vanilla model:

1. Setting `num_layers` to 50, we train the model for 30 epochs. Then, we deploy it to an `m1.c5.4xlarge` instance as usual:

```
ic_predictor = ic.deploy(initial_instance_count=1,
                           instance_type='m1.c5.4xlarge',
                           endpoint_name=ic_endpoint_name)
```

2. We compile the model with Neo, targeting the EC2 c5 instance family. We also define the input shape of the model: one image, three channels (red, green, blue), and 224 x 224 pixels (the default value for the image classification algorithm). As built-in algorithms are implemented with Apache MXNet, we set the framework accordingly:

```
output_path = 's3://{{}}/{{}}/output-neo/'
               .format(bucket, prefix)
```

```
ic_neo_model = ic.compile_model(  
    target_instance_family='ml_c5',  
    input_shape={'data':[1, 3, 224, 224]},  
    role=role,  
    framework='mxnet',  
    framework_version='1.5.1',  
    output_path=output_path)
```

3. We then deploy the compiled model as usual, explicitly setting the prediction container to the Neo-enabled version of image classification:

```
ic_neo_model.image = get_image_uri(  
    session.boto_region_name,  
    'image-classification-neo',  
    repo_version='latest')  
ic_neo_predictor = ic_neo_model.deploy(  
    endpoint_name=ic_neo_endpoint_name,  
    initial_instance_count=1,  
    instance_type='ml.c5.4xlarge')
```

4. Downloading a test image, and using the same benchmarking function that we used for Amazon Elastic Inference, we measure the time required to predict 1,000 images:

```
predict_images(ic_predictor)  
predict_images(ic_neo_predictor)
```

Prediction with the vanilla model takes 87 seconds. Prediction with the Neo-optimized model takes 28.5 seconds, three times faster! That compilation step sure paid off. You'll also be happy to learn that compiling Neo models is free of charge, so there's really no reason not to try it.

Let's take a look at these compiled models.

Exploring models compiled with Neo

Looking at the output location passed to the `compile_model()` API, we see the model artifact generated by Neo:

```
$ aws s3 ls s3://sagemaker-eu-west-1-123456789012/dogscats/
  output-neo/
    model-ml_c5.tar.gz
```

Copying it locally and extracting it, we see that it contains both the original model and its compiled version:

```
$ aws s3 cp s3://sagemaker-eu-west-1-123456789012/dogscats/
  output-neo/model-ml_c5.tar.gz .
$ tar xvfz model-ml_c5.tar.gz
compiled.meta
model-shapes.json
compiled.params
compiled_model.json
compiled.so
```

In particular, the `compiled.so` file is a native file containing hardware-optimized versions of the model operators:

```
$ file compiled.so
compiled.so: ELF 64-bit LSB shared object, x86-64
$ nm compiled.so | grep conv | head -3
0000000000005880 T fused_nn_contrib_conv2d_NCHWc
00000000000347a0 T fused_nn_contrib_conv2d_NCHWc_1
0000000000032630 T fused_nn_contrib_conv2d_NCHWc_2
```

We could look at the assembly code for these, but something tells me that most of you wouldn't particularly enjoy it. Joking aside, this is completely unnecessary. All we need to know is how to compile and deploy models with Neo.

Now, how about we deploy our model on a **Raspberry Pi**?

Deploying an image classification model on a Raspberry Pi

The Raspberry Pi is a fantastic device, and despite its limited compute and memory capabilities, it's well capable of predicting images with complex deep learning models. Here, I'm using a Raspberry Pi 3 Model B, with a 1.2 GHz quad-core ARM processor and 1 GB of memory. That's definitely not much, yet it could run a vanilla Apache MXNet model.

Inexplicably, there is no pre-packaged version of MXNet for Raspberry Pi, and building it from source is a painstakingly long and unpredictable process. (I'm looking at you, OOM errors!) Fortunately, thanks to the DLR, we can do away with all of it!

1. In our SageMaker notebook, we compile the model for the Raspberry Pi:

```
output_path = 's3://{} / {} / output-neo /'
              .format(bucket, prefix)
ic_neo_model = ic.compile_model(
    target_instance_family='rasp3b',
    input_shape={'data': [1, 3, 224, 224]},
    role=role,
    framework='mxnet',
    framework_version='1.5.1',
    output_path=output_path)
```

2. On our local machine, we fetch the compiled model artifact from S3 and copy it to the Raspberry Pi:

```
$ aws s3 cp s3://sagemaker-eu-west-1-123456789012/
  dogscats/output-neo/model-rasp3b.tar.gz .
$ scp model-rasp3b.tar.gz pi@raspberrypi:~
```

3. Moving to the Raspberry Pi, we extract the compiled model to the resnet50 directory:

```
$ mkdir resnet50
$ tar xvfz model-rasp3b.tar.gz -C resnet50
```

4. Installing the DLR is very easy. We locate the appropriate package at <https://github.com/neo-ai/neo-ai-dlr/releases>, download it, and use pip to install it:

```
$ wget https://neo-ai-dlr-release.s3-us-west-2.amazonaws.com/v1.9.0/rasp3b/dlr-1.9.0-py3-none-any.whl  
$ pip3 install dlr-1.9.0-py3-none-any.whl
```

5. We first write a function that loads an image from a file, resizes it to 224 x 224 pixels, and shapes it as a (1, 3, 224, 224) numpy array, the correct input shape of our model:

```
import numpy as np  
from PIL import Image  
  
def process_image(filename):  
    image = Image.open(filename)  
    image = image.resize((224, 224))  
    image = np.asarray(image)          # (224, 224, 3)  
    image = np.moveaxis(image, 2, 0). # (3, 224, 224)  
    image = image[np.newaxis, :].     # (1, 3, 224, 224)  
    return image
```

6. Then, we import the DLR and load the compiled model from the `resnet50` directory:

```
from dlr import DLRModel  
model = DLRModel('resnet50')
```

7. Then, we load a dog image... or an image of a cat. Your choice!

```
image = process_image('dog.jpg')  
#image = process_image('cat.png')  
input_data = {'data': image}
```

8. Finally, we predict the image 100 times, printing the prediction to defeat any lazy evaluation that MXNet could implement:

```
import time  
total = 0  
for i in range(0,100):  
    tick = time.time()  
    out = model.run(input_data)
```

```
print(out[0])
tock = time.time()
total+= tock-tick
print(total)
```

The following dog and cat images are respectively predicted as [2.554065e-09 1.000000e+00] and [9.9967313e-01 3.2689856e-04], which is very nice given the validation accuracy of our model (about 84%):



Figure 13.5 – Test images (source: Wikimedia)

Prediction time is about 1.2 seconds per image, which is slow but certainly good enough for plenty of embedded applications. Predicting with the vanilla model takes about 6–7 seconds, so the speed-up is very significant.

As you can see, compiling models is a very effective technique. In the next section, we're going to focus on one of Neo's targets, **AWS Inferentia**.

Deploying models on AWS Inferentia

AWS Inferentia is a custom chip designed specifically for high-throughput and low-cost prediction (<https://aws.amazon.com/machine-learning/inferentia>). Inferentia chips are hosted on **EC2 inf1** instances. These come in different sizes, with 1, 4, or 16 chips. Each chip contains four **NeuronCores**, implementing high-performance matrix multiply engines that speed up typical deep learning operations such as convolution. NeuronCores also contain large caches that save external memory accesses.

In order to run on Inferentia, models need to be compiled and deployed with the Neuron SDK (<https://github.com/aws/aws-neuron-sdk>). This SDK lets you work with TensorFlow, PyTorch, and Apache MXNet models.

You can work with the Neuron SDK on EC2 instances, compiling and deploying models yourself. Once again, SageMaker simplifies the whole process, as inf1 instances are part of the target architectures that Neo can compile models for.

You can find an example at https://github.com/awslabs/amazon-sagemaker-examples/tree/master/sagemaker_neo_compilation_jobs/deploy_tensorflow_model_on_Inf1_instance.

To close this chapter, let's sum up all the cost optimization techniques we discussed throughout the book.

Building a cost optimization checklist

You should constantly pay attention to cost, even in the early stages of your machine learning project. Even if you're not paying the AWS bill, someone is, and I'm sure you'll quite quickly find out who that person is if you spend too much.

Regularly going through the following checklist will help you spend as little as possible, get the most machine learning-happy bang for your buck, and hopefully keep the finance team off your back!

Optimizing costs for data preparation

With so much focus on optimizing training and deployment, it's easy to overlook data preparation. Yet, this critical piece of the machine learning workflow can incur very significant costs.

Tip #1

Resist the urge to build ad hoc ETL tools running on instance-based services.

Obviously, your workflows will require data to be processed in a custom fashion, such as applying domain-specific feature engineering. Working with a managed service such as **Amazon Glue**, **Amazon Athena**, or **Amazon SageMaker Data Wrangler**, you will never have to provision any infrastructure, and you will only pay for what you use.

As a second choice, **Amazon EMR** is a fine service, provided that you understand how to optimize its cost. As much as possible, you should avoid running long-lived, low-usage clusters. Instead, you should run transient clusters and rely massively on **Spot Instances** for task nodes. You can find more information at the following sites:

- <https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-plan-longrunning-transient.html>

- <https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-instance-purchasing-options.html>

The same advice applies to **Amazon EC2** instances.

Tip #2

Use SageMaker Ground Truth and automatic labeling to cut down on data labeling costs.

If you need to label large unstructured datasets, enabling automatic labeling in **SageMaker Ground Truth** can save you a significant amount of time and money compared to labeling everything manually. You can read about it at <https://docs.aws.amazon.com/sagemaker/latest/dg/sms-automated-labeling.html>.

Optimizing costs for experimentation

Experimentation is another area that is often overlooked, and you should apply the following tips to minimize the related spend.

Tip #3

You don't have to use SageMaker Studio.

As explained in *Chapter 1, Introducing Amazon SageMaker*, you can easily work with SageMaker Python SDK on your local machine or on a local development server.

Tip #4

Stop Studio instances when you don't need them.

This sounds like an obvious one, but are you really doing it? There's really no reason to run idle instances; commit your work, stop them, and then restart them when you need them again. Storage is persisted.

Tip #5

Experiment on a small scale and with instances of the correct size.

Do you really need the full dataset to start visualizing data and evaluating algorithms? Probably not. By working on a small fraction of your dataset, you'll be able to use smaller notebook instances. Here's an example: imagine 5 developers working 10 hours a day on their own m1.c5.2xlarge notebook instance. The daily cost is $5 \times 10 \times \$0.557 = \27.85 .

Right-sizing to `m1.t3.xlarge` (less RAM, burstable behavior), the daily cost would be reduced to $5 \times 10 \times \$0.233 = \11.65 . You would save \$486 per month, which you could certainly spend on more experimentation, more training, and more **automatic model tuning**.

If you need to perform large-scale cleaning and processing, please take the time to migrate that work to a managed service (see Tip #1) instead of working all day long with a humongous instance. Don't say, "Me? Never!" I know you're doing it!

Tip #6

Use local mode.

We saw in *Chapter 7, Extending Machine Learning Services with Built-In Frameworks*, how to use **local mode** to avoid firing up managed infrastructure in the AWS cloud. This is a great technique to quickly iterate at no cost in the experimentation phase!

Optimizing costs for model training

There are many techniques you can use, and we've already discussed most of them.

Tip #7

Don't train on Studio instances.

I'm going to repeat myself here, but it's an important point. Unfortunately, this antipattern seems to be pretty common. People pick a large instance (such as `m1.p3.2xlarge`), fire up a large job in their notebook, leave it running, forget about it, and end up paying good money for an instance sitting idle for hours once the job is complete.

Instead, please run your training jobs on **managed instances**. Thanks to **distributed training**, you'll get your results much quicker, and as instances are terminated as soon as training is complete, you will never overpay for training.

As a bonus, you won't be at the mercy of a clean-up script (or an overzealous admin) killing all notebook instances in the middle of the night ("because they're doing nothing, right?").

Tip #8

Pack your dataset in RecordIO/TFRecord files.

This makes it easier and faster to move your dataset around and distribute it to training instances. We discussed this at length in *Chapter 5, Training Computer Vision Models*, and *Chapter 6, Training Natural Language Processing Models*.

Tip #9

Use pipe mode.

Pipe mode streams your dataset directly from Amazon S3 to your training instances. No copying is involved, which saves on start-up time. We discussed this feature in detail in *Chapter 9, Scaling Your Training Jobs*.

Tip #10

Right-size training instances.

We saw how to do this in *Chapter 9, Scaling Your Training Jobs*. One word: **CloudWatch** metrics.

Tip #11

Use Managed Spot Training.

We covered this in great detail in *Chapter 10, Advanced Training Techniques*. If that didn't convince you, nothing will! Seriously, there are very few instances when **Managed Spot Training** should not be used, and it should be a default setting in your notebooks.

Tip #12

Use AWS-provided versions of TensorFlow, Apache MXNet, and so on.

We have entire teams dedicated to extracting the last bit of performance from deep learning libraries on AWS. No offense, but if you think you can `pip install` and go faster, your time is probably better invested elsewhere. You can find more information at the following links:

- <https://aws.amazon.com/blogs/machine-learning/faster-training-with-optimized-tensorflow-1-6-on-amazon-ec2-c5-and-p3-instances/>,
- <https://aws.amazon.com/about-aws/whats-new/2018/11/tensorflow-scalability-to-256-gpus/>
- <https://aws.amazon.com/blogs/machine-learning/amazon-web-services-aSchieves-fastest-training-times-for-bert-and-mask-r-cnn/>

Optimizing costs for model deployment

This very chapter was dedicated to several of these techniques. I'll add a few more ideas to cut costs even further.

Tip #13

Use batch transform if you don't need online predictions.

Some applications don't require a live endpoint. They are perfectly fine with **batch transform**, which we studied in *Chapter 11, Deploying Machine Learning Models*. The extra benefit is that the underlying instances are terminated automatically when the batch job is done, meaning that you will never overpay for prediction because you left an endpoint running for a week for no good reason.

Tip #14

Delete unnecessary endpoints.

This requires no explanation, and I have written "Delete the endpoint when you're done" tens of times in this book already. Yet, this is still a common mistake.

Tip #15

Right-size endpoints and use autoscaling.

Tip #16

Use a multi-model endpoint to consolidate models.

Tip #17

Compile models with Amazon Neo to use fewer hardware resources.

Tip #18

At large scale, use AWS Inferentia instead of GPU instances.

And, of course, the mother of all tips for all things AWS, which is why we dedicated a full chapter to it (*Chapter 12, Automating Machine Learning Workflows*).

Tip #19

Automate, automate, automate!

Tip #20

Purchase Savings Plans for Amazon SageMaker.

Savings Plans is a flexible pricing model that offers low prices on AWS usage, in exchange for a commitment to a consistent amount of usage for a one-year or three-year term (<https://aws.amazon.com/savingsplans/>).

Savings Plans is now available for SageMaker, and you'll find it in the console at <https://console.aws.amazon.com/cost-management/home#/savings-plans/>.

Built-in recommendations help you pick the right commitment and purchase a plan in minutes. Depending on the term and the commitment, you could save up to 72% (!) on all instance-based SageMaker costs. You can find a demo at <https://aws.amazon.com/blogs/aws/slash-your-machine-learning-costs-with-instance-price-reductions-and-savings-plans-for-amazon-sagemaker/>.

Equipped with this checklist, not only will you slash your machine learning budget but you will also build more robust and more agile workflows. Rome wasn't built in a day, so please take your time, use common sense, apply the techniques that matter most right now, and iterate.

Summary

In this final chapter, you learned different techniques that help to reduce prediction costs with SageMaker. First, you saw how to use autoscaling to scale prediction infrastructure according to incoming traffic. Then, you learned how to deploy an arbitrary number of models on the same endpoint, thanks to multi-model endpoints.

We also worked with Amazon Elastic Inference, which allows you to add fractional GPU acceleration to a CPU-based instance and find the right cost-performance ratio for your application. We then moved on to Amazon SageMaker Neo, an innovative capability that compiles models for a given hardware architecture, both for EC2 instances and embedded devices. Finally, we built a cost optimization checklist that will come in handy for your upcoming SageMaker projects.

You've made it to the end. Congratulations! You now know a lot about SageMaker. Now, go grab a dataset, build cool stuff, and let me know about it!



Packt.com

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

Why subscribe?

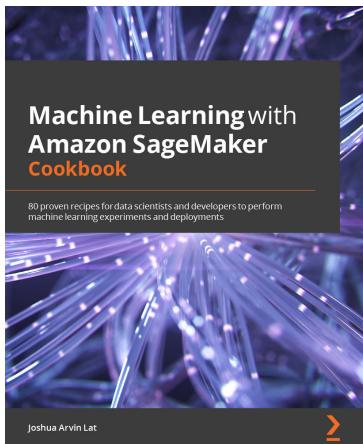
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print,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

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

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

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

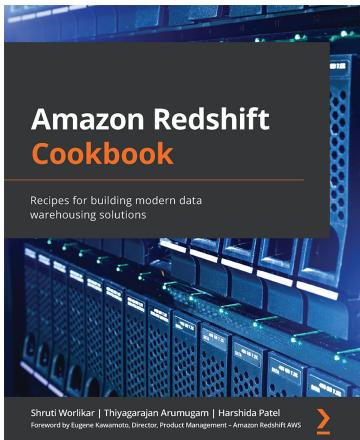


Machine Learning with Amazon SageMaker Cookbook

Joshua Arvin Lat

ISBN: 9781800567030

- Train and deploy NLP, time series forecasting, and computer vision models to solve different business problems
- Push the limits of customization in SageMaker using custom container images
- Use AutoML capabilities with SageMaker Autopilot to create high-quality models
- Work with effective data analysis and preparation techniques
- Explore solutions for debugging and managing ML experiments and deployments
- Deal with bias detection and ML explainability requirements using SageMaker Clarify
- Automate intermediate and complex deployments and workflows using a variety of solutions



Amazon Redshift Cookbook

Shruti Worlikar, Thiagarajan Arumugam, Harshida Patel

ISBN: 9781800569683

- Use Amazon Redshift to build petabyte-scale data warehouses that are agile at scale
- Integrate your data warehousing solution with a data lake using purpose-built features and services on AWS
- Build end-to-end analytical solutions from data sourcing to consumption with the help of useful recipes
- Leverage Redshift's comprehensive security capabilities to meet the most demanding business requirements
- Focus on architectural insights and rationale when using analytical recipes
- Discover best practices for working with big data to operate a fully managed solution

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 *Learn Amazon Sagemaker, 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

A

- accelerators 493
- alternative workflows
 - using 114
- Amazon Athena
 - about 374, 504
 - dataset, building with 471
- Amazon CloudWatch Logs 115
- Amazon Cognito
 - about 36
 - URL 36
- Amazon EC2
 - about 265, 341
 - reference link 298
- Amazon EC2 Spot Instances 341, 342
- Amazon ECR 271
- Amazon EFS
 - about 310
 - working with 331
- Amazon Elastic Container Registry (ECR)
 - about 115
 - reference link 115
- Amazon Elastic Container Service (ECS) 417
- Amazon Elastic File System (EFS)
 - reference link 330
- Amazon Elastic Inference
 - URL 493
 - used, for deploying model 492-496
- Amazon Elastic Kubernetes Service (EKS) 417
- Amazon EMR 504
- Amazon Fargate
 - about 417
 - models, deploying on 418
- Amazon FSx for Lustre
 - about 310
 - object detection, model training 337
 - reference link 330
 - working with 335, 336
- Amazon Glue 504
- Amazon Identity and Access Management (IAM)
 - reference link 14
- Amazon KMS key 41
- Amazon Linux 2 265
- Amazon Mechanical Turk
 - about 35
 - URL 35
- Amazon S3
 - dataset, uploading to 64, 65
- Amazon SageMaker

- advanced capabilities 6
- built-in algorithms, discovering 110
- built-in frameworks,
 - discovering 220, 221
- built-in open source frameworks 6
- capabilities 5
- computer vision built-in algorithms 146
- NLP built-in algorithms 184
- permissions, reference link 14
- setting up, on local machine 10
- Amazon SageMaker API
 - about 7
 - Amazon SageMaker SDK 9, 10
 - AWS language SDKs 8, 9
- Amazon SageMaker Autopilot
 - about 6
 - candidate generation notebook 103-107
 - data, analyzing 79
 - data exploration notebook 102, 103
 - discovering 78, 79
 - feature engineering 80
 - job artifacts 100, 101
 - job, cleaning up 100
 - job, launching in SageMaker
 - Studio 81-85
 - job, monitoring in SageMaker
 - SDK 98, 99
 - job, monitoring in SageMaker
 - Studio 86-89
 - jobs, comparing in SageMaker
 - Studio 89-93
 - learning 100
 - model, deploying 94-96
 - model, testing 94-96
 - model tuning 80
 - using, in SageMaker Studio 81
- Amazon SageMaker capabilities
 - building infrastructure 5
- deployment infrastructure 7
- training infrastructure, managing 6
- Amazon SageMaker Data Wrangler
 - about 504
 - dataset, loading 50-57
 - dataset, transforming 57-61
 - data, transforming 49
- Amazon SageMaker Debugger
 - profiling information, enabling 306-309
 - training challenges, solving 309-311
 - used, for monitoring training jobs 304
 - used, for profiling training jobs 304
- Amazon SageMaker Experiments 80
- Amazon SageMaker Ground Truth
 - about 149
 - data, labeling 34
- Amazon SageMaker Ground
 - Truth, for data labeling
 - reference link 49
- Amazon SageMaker JumpStart
 - used, for deploying models 22
 - used, for deploying one-click solutions 22
- Amazon SageMaker Model Monitor
 - features 409
 - used, for monitoring
 - prediction quality 410
- Amazon SageMaker Neo
 - about 497, 498
 - components 497
 - image classification model,
 - compiling on 498, 499
 - image classification model,
 - deploying on 499
 - models compiled, exploring with 500
 - models, compiling with 497

-
- Amazon SageMaker Pipelines
 - creating 474
 - end-to-end workflows,
 - building with 467, 468
 - model, creating 473, 474
 - model, registering 473, 474
 - running 475, 476
 - Amazon SageMaker Processing
 - about 80
 - batch jobs, running 63
 - Amazon SageMaker Processing API 64
 - Amazon SageMaker SDK
 - about 9
 - installing, with Anaconda 12, 13
 - installing, with virtualenv 10-12
 - reference link 9
 - Amazon SageMaker Studio
 - about 5
 - onboarding, onboarding with
 - quick start procedure 16
 - onboarding, options 16
 - onboarding, with quick start
 - procedure 16-20
 - reference link 16
 - setting up 15
 - Amazon Simple Notification Service
 - URL 36
 - Anaconda
 - installation link 12
 - used, for installing Amazon SageMaker SDK 12, 13
 - anomalies
 - detecting, with Random Cut Forest (RCF) 137-142
 - Apache MXNet
 - about 149, 179, 221
 - reference link 264
 - using, for model deployment 230
 - Apache Spark
 - working with 253
 - architecture search
 - automatic model tuning, using 359, 360
 - augmented manifest
 - about 46, 163, 164, 313
 - reference link 313
 - auto-generated notebooks 101
 - automated data labeling
 - reference link 42
 - automatic model tuning
 - about 80, 107, 350, 351, 506
 - hyperparameters, optimizing with 349
 - using, for architecture search 359, 360
 - using, with Keras 354
 - using, with object detection 351-354
 - Automatic Model Tuning 120
 - automation, with AWS Cloud Development Kit 446
 - automation, with AWS CloudFormation
 - about 428, 429
 - blue-green deployment 444
 - canary deployment 440-444
 - model, deploying to real-time
 - endpoint 432-435
 - second production variant, adding
 - to endpoint 438, 440
 - stack, moving with change set 435-438
 - template, writing 429-432
 - Auto Scaling 116
 - AWS
 - reference link 7
 - AWS CloudFormation
 - reference link 9
 - used, for automation 428, 429
 - AWS Glue Data Catalog 374
 - AWS Inferentia
 - models, deploying on 503, 504

AWS language SDKs
 about 8, 9
 reference link 8

AWS Marketplace
 URL 35

AWS permissions 14

AWS Service Catalog 467

AWS Single Sign-On (SSO) boarding
 reference link 16

AWS Sockeye
 reference link 187

AWS Step Functions
 end-to-end workflows,
 building with 452

B

bag-of-words (BoW) 186

batch jobs
 running, with Amazon SageMaker Processing 63

batch size
 updating 322

batch_skipgram 207

batch transform
 about 387
 feature 106
 models, deploying on 406, 407

Bayesian optimization 350

bias
 detecting, in datasets with SageMaker Clarify 376

binary classification 79

BlazingText
 about 111, 184, 300
 data for word vectors,
 preparing with 196

text, classifying 205-207
 used, for preparing data for classification 189-192
 word vectors, computing 207, 208

BlazingText algorithm
 about 185
 reference link 185

BlazingText classification model
 using, with FastText 208, 209

BlazingText models
 using, with FastText 208

BlazingText training job
 scaling 300-303

BlazingText version 2
 used, for preparing data for classification 193-195

BlazingText word vectors
 using, with FastText 209

blue-green deployment
 implementing 444
 implementing, with multiple endpoint 445
 implementing, with single endpoint 445
 URL 444

Boston Housing 452

boto3 168

boto3 SDK
 real-time endpoints, deploying with 402-406
 used, for managing real-time endpoints 402

bounding box 147, 161

BoW representation
 data, converting to 200, 201

build environment
 setting up, on EC2 266

built-in algorithm containers
 reference link 115

built-in algorithms
 about 104
 discovering, in Amazon SageMaker 110
 SageMaker SDK, using with 116
 used, for deploying models 112
 used, for training models 112
 working with 124

built-in algorithms, Amazon SageMaker
 supervised learning 110, 111
 unsupervised learning 111

built-in computer vision algorithms
 using 165

built-in frameworks
 using 238

built-in frameworks, Amazon SageMaker
 Apache MXNet 221
 Chainer 221
 example, running with
 XGBoost 222-224
 models, training and deploying
 locally 226, 227
 PyTorch 220
 Scikit-Learn 220
 Spark 221
 TensorFlow 220
 training, with script mode 227-229
 working, with containers 225
 XGBoost 220

C

Caltech-256 dataset
 reference link 170

canary deployment
 about 440-444
 URL 440

candidate generation notebook 80, 86, 103

candidate pipelines 80

CDK application
 creating 446-448
 deploying 450, 451
 writing 448, 450

central processing unit (CPU)
 instances 185

Chainer
 URL 221

change sets 429

checkpoints
 training, resuming from 347, 348

classification
 with XGBoost 125-127

Class Imbalance (CI) 379

Cloud Development Kit (CDK)
 installing 446

CloudFormation
 reference link 23

CloudWatch 441

COCO dataset
 URL 163

code
 dataset, processing 72

Command-Line Interface (CLI)
 reference link 340

comma-separated values (CSV) 186

computer vision built-in algorithms,
 Amazon SageMaker
 about 146

image classification algorithm 146

object detection algorithm 147

semantic segmentation algorithm 148

using, for training 149

container services
 models, deploying to 417

continuous BoW (CBOW) 207

cosine similarity
 reference link 187

-
- cost optimization checklist
 - building 504
 - data preparation 504
 - experimentation 505, 506
 - model deployment 508, 509
 - model, training 506, 507, 508
 - costs
 - comparing 340, 341
 - CSV 275
 - custom container
 - building 280
 - deploying, on SageMaker 281, 282
 - training, on SageMaker 281
 - custom transform 58
- D**
- data
 - analyzing 79
 - converting, to BoW
 - representation 200, 201
 - labeling, with Amazon SageMaker
 - Ground Truth 34
 - tokenizing 198-200
 - transforming, with Amazon
 - SageMaker Data Wrangler 49
 - uploading, for labeling 39
 - Databricks
 - URL 282
 - data drift 409
 - data exploration notebook 79, 86
 - data for classification
 - preparing, with BlazingText 189-192
 - preparing, with BlazingText
 - version 2 193-195
 - data for topic modeling
 - preparing, with LDA 197
 - preparing, with NTM 197
 - data for word vectors
 - preparing, with BlazingText 196, 197
 - data loading
 - simplifying, with MLIO 313
 - data parallelism 315
 - Data Science SDK
 - reference link 452
 - dataset
 - bias, detecting with SageMaker
 - Clarify 376
 - building, with Amazon Athena 471
 - building, with SageMaker
 - Feature Store 370
 - building, with SageMaker
 - Processing 471
 - features, managing with SageMaker
 - Feature Store 370
 - loading, in SageMaker Data
 - Wrangler 50-57
 - processing, with code 72
 - processing, with SageMaker
 - Processing 469, 470
 - processing, with scikit-learn 64
 - streaming, with pipe mode 311
 - transforming, in SageMaker
 - Data Wrangler 57-61
 - uploading, to Amazon S3 64, 65
 - datasets labeled
 - using, with SageMaker Ground Truth 203, 204
 - DeepAR 111
 - Deep Graph Library (DGL) 242
 - DeepLab v3 148
 - Deep Learning AMI 493
 - Deep Learning Container (DLC)
 - about 399
 - reference link 225

Deep Learning Containers images
reference link 265

deep learning (DL) models 185

Deep Learning Runtime (DLR)
URL 497

dependencies
libraries, adding for 233
libraries, adding for training 232, 233
managing 231
source files, adding for training 231

dependencies, for training
source files, adding 231

detection datasets
converting, to image format 153, 154

Difference in Positive Proportions
in Labels (DPL) 379

Difference in Positive Proportions in
Predicted Labels (DPPL) 379

Dirichlet distribution
reference link 186

DistilBERT model
reference link 246

Distributed Data Parallel
(DDP) library 324

Distributed Model Parallel
(DMP) library 325

distributed training
about 112, 158, 315, 506
for built-in algorithms 315
for built-in frameworks 316
for custom containers 316

distribution policy 310

Docker containers
packaging algorithms 115

Docker Hub
about 265, 280
URL 265

E

EBS optimized instance
reference link 310

EC2
build environment, setting up 266

EC2 Autoscaling
URL 482

EC2 instance
creating 332, 333

EFS volume
accessing 333
provisioning 331, 332

Elastic Block Store (EBS)
about 299
reference link 299

Elastic Fabric Adapter
reference link 310

Elastic File System (EFS)
object detection, model
training 334, 335

Elastic Inference Accelerator 494

embeddings 185

endpoint
autoscaling 482-486

end-to-end workflows
building, with Amazon SageMaker
Pipelines 467, 468
building, with AWS Step Functions 452
parameters, defining 468, 469

end-to-end workflows, building
with AWS Step Functions
implementing 453-459

Lambda function, adding 461

parallel execution, adding 460, 461

permissions, setting up 452, 453

execution role 462

Extract-Transform-Load (ETL) 253

F

factorization machines
about 111, 127, 128, 186
model, building on MovieLens 130-135
training, with pipe mode 314, 315
faster Pipe mode
reference link 312
FastText
BlazingText classification
model, using with 208
BlazingText models, using with 208
BlazingText word vectors,
using with 209
reference link 185
feature engineering 80
FIFOs 311
file layout
in SageMaker container 263
file mode 311
flask
reference link 274
framework code
workflow 233
framework code, running on
Amazon SageMaker
about 234
local mode, using 237, 238
managed infrastructure, using 238
script mode, implementing 235, 236
testing, locally 236
framework container
customizing 265
working with 225, 226
FSx filesystems, deployment options
persistent 335
scratch 335
Fully-Convolutional Networks

(FCNs) 148
fully custom container
building, for R language 277
building, for SageMaker
Processing 289, 290
building, for scikit-learn 272
deploying 274-277
training with 272-274

G

Gaussian process regression 350
gensim library
reference link 188
Global Vectors (GloVe) model 187
Gluon API
URL 221
Graph Convolutional Network 243
graphics processing unit (GPU)
instances 185
Graph Neural Network (GNN) 242

H

HPC applications 335
HTTPS endpoint 123
Hugging Face
dataset, preparing 246-249
source libraries 246
training, with SageMaker DDP 328
training, with SageMaker DMP 329
URL 245
Hugging Face model
deploying 251-253
fine-tuning 249-251
working with 245
hyperparameter optimization 80

hyperparameters
about 120
optimizing, with automatic
model tuning 349

I
identifier (ID) 186

image classification

about 146
algorithm 146, 147

image classification dataset

converting, to image format 150-153
converting, to RecordIO 157, 158

image classification model

compiling, on SageMaker 498, 499
deploying, on Raspberry Pi 501-503
deploying, on SageMaker 498, 499
fine-tuning 170, 171
instances, adding 323
output 324

scaling, on ImageNet 317

training 165

training, in image format 165-169

image datasets

preparing 150

image files

working with 150

image format

detection datasets, converting
to 153, 154

segmentation datasets,

converting to 154-157

training, in RecordIO format 169

ImageNet

dataset, preparing 317-319

image classification model, scaling 317

infrastructure requisites defining 319
input configuration, defining 319
reference link 317
training job, defining 319
training on 320, 321
URL 170

I
images

labeling 44, 46

imbalanced-learn open source library

reference link 382

inference containers

building 266-269

inference pipelines

about 387

models, deploying on 408, 409

input data

saving 201-203

IP Insights 111

J

JavaScript Object Notation (JSON) 389

JavaScript Object Notation

Lines (JSON Lines)

about 203

format 163

joblib

URL 393

JSON file 39

Jupyter server

URL 11

K

Keras

automatic model tuning, using with 354

callback, adding for early shopping 357-359
 custom metric, optimizing 355
 model, optimizing 356, 357
 URL 220
 used, for checkpointing 345, 346
 working with 239-242
Keras job
 debugging 366-369
 inspecting 366-369
k-means 111
K-nearest neighbors (KNN) 111

L

labeling
 data, uploading for 39
labeling job
 creating 39-43
Lambda function
 adding, to workflow 461
 permissions, setting up 462-466
 writing 464, 465
LDA algorithm
 about 185, 186
 data for topic modeling,
 preparing with 197
 topics modeling 210-213
lemmatization
 reference link 199
Linear Learner
 about 110
 algorithm 104
linear regression 79
list file 150
Local Mode
 about 271
 using 226, 227

long short-term memory (LSTM) 187
Lustre filesystem
 reference link 335

M

machine learning (ML)
 about 376
 reference link 4
managed infrastructure
 using 114
managed spot training
 about 342, 343
 used, for checkpointing 346, 347
 used, for optimizing training costs 340
 used, for training 346, 347
 using 345
 using, with object detection 344
manifest file 39, 40
mean average precision metric
 (mAP) 173, 174
mean intersection-over-union metric (mIOU) 177
MLeap
 reference link 253
MLflow
 about 282
 installing 282
 model, training 283, 284
 training 282
 URL 282
 used, for building SageMaker container 285
 used, for deploying model 285, 287
 used, for deploying model on SageMaker 287, 288
MLflow tutorial
 reference link 282

-
- mlp algorithm 104
 - model
 - compiling, with Amazon SageMaker Neo 497
 - creating, in Amazon SageMaker Pipelines 473, 474
 - deploying, from model registry 477-479
 - deploying, on SageMaker with MLflow 287, 288
 - deploying, to real-time endpoint 432-435
 - deploying, with Amazon Elastic Inference 492-496
 - deploying, with MLflow 285, 287
 - exporting 114
 - importing 114
 - registering, in Amazon SageMaker Pipelines 473, 474
 - training 472, 473
 - models artifact
 - about 113
 - built-in algorithms, examining 389, 391
 - built-in algorithms, exporting 389-391
 - built-in CV models, examining 391
 - built-in CV models, exporting 391, 392
 - examining 389
 - Hugging Face models,
 - examining 394, 395
 - Hugging Face models,
 - exporting 394, 395
 - scikit-learn models, examining 393
 - scikit-learn models, exporting 393
 - TensorFlow models, examining 394
 - TensorFlow models, exporting 394
 - XGBoost models, examining 392
 - XGBoost models, exporting 392
 - model deployment
 - about 107, 229
 - with Apache MXNet 230
 - with PyTorch 230
 - with TensorFlow 229
 - model parallelism 315
 - model registry
 - about 467
 - model, deploying from 477-479
 - models
 - compiling, with Amazon SageMaker Neo 497
 - deploying, on batch transformers 406, 407
 - deploying, on inference pipelines 408, 409
 - deploying, on real-time endpoints 396
 - deploying, to container services 417
 - deploying, with built-in algorithms 112
 - deploying, with other frameworks 231
 - exporting 389
 - training, with built-in algorithms 112
 - models, deploying on Amazon Fargate
 - about 418
 - configuring 419
 - model, preparing 418
 - task, defining 420-422
 - task, running 423-425
 - models, exploring with
 - SageMaker Debugger
 - about 360, 361
 - Keras job, debugging 366-369
 - Keras job, inspecting 366-369
 - XGBoost job, debugging 361, 362
 - XGBoost job, inspecting 362-364
 - MovieLens
 - Factorization Machines model,
 - building on 130-135
 - reference link 127

MovieLens dataset 314
 multi-class classification 79
 multi-GPU instances
 used, for scaling up 300, 315
 multi-label classification 147
 multi-model endpoint
 about 387, 487
 building, with scikit-learn 487-492
 deploying 487
 Multi-Model Server (MMS)
 about 230
 reference link 264
 multiple endpoint
 blue-green deployment,
 implementing with 445
 MXNet
 URL 389

N

named pipes 311
 natural language datasets
 preparing 188
 Natural Language Toolkit (NLTK) library
 reference link 188
 Neo-enabled containers
 URL 498
 NeuronCores 503
 Neuron SDK
 URL 503
 NLP built-in algorithms,
 Amazon SageMaker
 about 184
 BlazingText 185
 LDA 185-187
 NTM 186
 seq2seq 187

using, for training 188
 Node.js
 URL 446
 Notebook instances 5
 NTM algorithm
 about 185, 186
 data for topic modeling,
 preparing with 197
 topics, modeling 214-217

O

Object2Vec 111
 object detection
 about 146
 algorithm 147
 managed spot training, using with 344
 model, training with Amazon
 FSx for Lustre 337
 model, training with EFS 334, 335
 using, with automatic model
 tuning 351-354
 object detection dataset
 converting, to RecordIO 158-162
 object detection model
 training 172-175
 one-click solutions
 deploying, with Amazon
 SageMaker JumpStart 21-24
 ONNX format 497
 optimal hyperparameters, techniques
 grid search 349
 hyperparameter optimization
 (HPO) 350
 manual search 349
 random search 349

P

packaging algorithms, in Docker containers 115
 pandas library
 reference link 188
 Pascal VOC dataset 156
 performance modes
 general purpose 331
 Max I/O 331
 pipe mode
 about 158, 311, 507
 factorization machines,
 training 314, 315
 used, for streaming dataset 311
 using, with algorithms and
 frameworks 313
 using, with built-in algorithms 312
 plumber
 coding with 278, 279
 reference link 277
 prediction infrastructure
 creating 115
 prediction quality, monitoring with
 Amazon SageMaker Model Monitor
 about 410
 bad data, sending 414, 415
 baseline, capturing 411-413
 data, capturing 410, 411
 monitoring schedule, setting up 413
 violation reports, examining 415, 416
 predictions, monitoring with Amazon
 SageMaker Model Monitor
 data, capturing 411
 pre-trained models
 deploying, with Amazon
 SageMaker JumpStart 22-27

fine-tuning, with Amazon SageMaker JumpStart 28-31
 Principal Component Analysis (PCA)
 about 111, 135, 408
 using 135, 136
 private workforce
 about 35
 creating 36, 37, 38
 processing script
 running 69-72
 writing, with scikit-learn 65-68
 protobuf
 about 129
 reference link 129
 Pyramid Scene Parsing (PSP) 148
 PySpark
 about 221
 reference link 64
 Python 3.x environment 340
 Python Imaging Library (PIL) 178
 PyTorch
 reference link 220, 264
 using, for model deployment 230
 working with 242-245

R

Random Cut Forest (RCF)
 about 111, 137
 anomalies, detecting 137-142
 Random Forest algorithm 57
 Raspberry Pi
 image classification model,
 deploying on 501-503
 Ray RLib 221
 real-time endpoint
 deploying, with boto3 SDK 402-406
 managing, with boto3 SDK 402

-
- models, deploying 396, 432-435
 - real-time HTTPS endpoint 113
 - recommendation
 - with Factorization Machines 127
 - RecordIO
 - about 129, 310
 - image classification dataset,
 - converting to 157, 158
 - object detection dataset,
 - converting to 158-162
 - reference link 129, 149
 - RecordIO files
 - working with 157
 - RecordIO format
 - image classification model,
 - training in 169
 - RecordIO-wrapped protobuf 312
 - RecordIO-wrapped protobuf format 186
 - Recurrent Neural Networks 111
 - Reinforcement Learning 221
 - ResNet 148
 - Resnet 18 model 28
 - ResNet-50 147
 - ResNet convolutional neural network 146
 - R language
 - about 277
 - coding with 278, 279
 - fully custom container, building for 277
 - rollback trigger 442
 - root mean square error (RMSE) 126
 - runnable notebook 103
- S**
- SageMaker
 - code, invoking 262
 - custom container, deploying 281, 282
 - custom container, training 281, 282
 - custom deployment, options 264, 265
 - custom training, options 263
 - model, training on 418
 - spam classification model,
 - building with 257, 259
 - working with 331, 335, 336
 - SageMaker AutoML
 - reference link 96
 - SageMaker Autopilot SDK
 - job, launching 97, 98
 - using 96
 - SageMaker Clarify
 - bias analysis, running 378
 - bias, detecting in datasets 376
 - bias metrics, analyzing 379
 - bias, mitigating 382-384
 - explainability analysis, running 380, 381
 - used, for configuring bias
 - analysis 376, 377
 - SageMaker container
 - building, with MLflow 285
 - file layout 263
 - SageMaker data parallel libraries
 - training with 324
 - SageMaker Data Wrangler 374
 - SageMaker Data Wrangler pipeline
 - exporting 62, 63
 - SageMaker DDP
 - used, for training Hugging Face 328
 - used, for training TensorFlow 325-328
 - SageMaker Debugger
 - models, exploring with 360, 361
 - SageMaker DMP
 - used, for training Hugging Face 329
 - SageMaker Feature Store
 - about 72, 467
 - capabilities, exploring 375
 - data, ingesting 374

-
- engineering features 370, 371
 - feature group, creating 371-373
 - features, querying to build
 - dataset 374, 375
 - SageMaker Processing, used for
 - ingesting dataset in 470, 471
 - used, for building datasets 370
 - used, for managing datasets features 370
 - SageMaker Ground Truth
 - datasets labeled, using with 203, 204
 - files working with 163, 164
 - URL 505
 - SageMaker Inference Toolkit
 - about 264
 - URL 487
 - SageMaker instance types
 - reference link 297
 - SageMaker model parallel libraries
 - training with 324
 - SageMaker Processing
 - about 247, 289, 467
 - dataset, building with 471
 - dataset, ingesting in SageMaker
 - Feature with 470, 471
 - dataset, processing with 469, 470
 - fully custom container,
 - building 289, 290
 - SageMaker SDK 96
 - SageMaker SDK, used for managing
 - real-time endpoints
 - about 396
 - existing endpoint, invoking 401
 - existing endpoint, updating 401, 402
 - Hugging Face model, deploying
 - with PyTorch 399-401
 - Hugging Face model, importing
 - with PyTorch 399-401
 - TensorFlow model, deploying 398, 399
 - TensorFlow model, importing 398, 399
 - XGBoost model, deploying 396, 397
 - XGBoost model, importing 396, 397
 - SageMaker SDK, with built-in algorithms
 - cleanup 124
 - data, preparing 117, 118
 - model, deploying 123, 124
 - training job, configuring 119-121
 - training job, launching 121, 122
 - using 116
 - sagemaker_sklearn_extension module
 - reference link 104
 - SageMaker Studio
 - about 96
 - Amazon SageMaker Autopilot, using 81
 - monitoring information,
 - viewing 304-306
 - profiling information, viewing 304-306
 - SageMaker Training Toolkit
 - reference link 263
 - using, with scikit-learn 270, 271
 - SageMaker workflow
 - end-to-end workflow 113
 - savings plans
 - reference link 509
 - Scala 221
 - scalability 112
 - scaling
 - about 296
 - monitoring 298
 - techniques 296
 - training infrastructure 297, 298
 - training time adapting to
 - business requisites 297
 - scaling out 300

scaling up
about 299
with multi-GPU instances 300

scatter plot 92

scikit-learn
dataset, processing 64
deploying 230, 231
fully custom container, building for 272
multi-model endpoint,
building with 487-492
processing script, writing 65-68
reference link 64
SageMaker Training Toolkit,
using 270, 271

Scikit-learn
reference link 264

Scikit-learn model 452

script mode
about 263
models, training with 229
training with 227-229

segmentation datasets
converting, to image format 154-157

segmentation masks 148

semantic segmentation
about 146
algorithm 148
model training 175-180

seq2seq algorithm 187

Sequence to Sequence (seq2seq) 185

SHapley Additive exPlanations (SHAP) 88

Simple Storage Service (S3) 388

single endpoint
blue-green deployment,
implementing with 445

single-label classification 147

Single Shot MultiBox Detector
(SSD) architecture 147

skipgram 207

software development kit (SDK) 396

spaCy library
reference link 188

spam classification model
building, with SageMaker 257, 259
building, with Spark 257, 259

Spark
about 282
combining, with SageMaker 253, 254
spam classification model,
building with 257, 259

Spark combining, with SageMaker

Spark and SageMaker, stages
reference link 259

Spark Cluster
creating 254-256

sparse datasets 128, 129

sparse matrix 129

sparse matrix object 202

Spot Instances 341

state machines
reference link 452

stemming
reference link 199

storage services
Amazon EFS, working with 330
Amazon FSx for Lustre,
working with 335, 336

SageMaker, working with 330, 335, 336
using 330

subword embeddings 207

supervised learning (SL) 110,
111, 149, 184

Synthetic Minority Oversampling
Technique (SMOTE) 382

T

target attribute 78
 TensorFlow
 reference link 264
 training, with SageMaker DDP 325-328
 using, for model deployment 229
 working with 239-242
 TensorFlow inference container
 reference link 229
 TensorFlow Serving
 reference link 229
 text
 classifying, with BlazingText 205-207
 labeling 46-49
 TFRecord 310
 throughput modes
 bursting throughput 331
 provisioned throughput 331
 topic modeling 185
 topics
 modeling, with LDA 210-213
 modeling, with NTM 214-217
 topic uniqueness (TU) 187
 TorchServe
 reference link 264
 Training APIs
 reference link 72
 training containers
 building 266-269
 training costs
 optimizing, with managed
 spot training 340
 training infrastructure
 creating 115
 training jobs
 monitoring, with Amazon
 SageMaker Debugger 304

profiling, with Amazon
 SageMaker Debugger 304
 transfer learning 149, 170
 trust policy 462

U

UCI Machine Learning Repository
 reference link 376
 unsupervised learning (UL) 111, 184

V

vendor workforce 35
 VGG-16 147
 virtualenv
 used, for installing Amazon
 SageMaker SDK 10-12

W

Word2Vec
 reference link 185
 word embedding topic coherence
 (WETC) 187
 word vectors
 about 185
 computing, with BlazingText 207, 208
 workforces
 using 35
 Workshop on Statistical Machine
 Translation (WMT) dataset 187

X

XGBoost
 about 111
 deploying 230, 231

example, running with 221-225
reference link 264
XGBoost algorithm 125
XGBoost job
 debugging 361, 362
 inspecting 362-364
XGBoost model
 autoscaling, setting up 482-486

Z

Zachary Karate Club dataset
reference link 243

