Cloud Native AI and Machine Learning on AWS

Use SageMaker for building ML models, automate MLOps, and take advantage of numerous AWS AI services

Premkumar Rangarajan

David Bounds

Cloud Native AI and Machine Learning on AWS

Use SageMaker for building ML models, automate MLOps, and take advantage of numerous AWS AI services

Premkumar Rangarajan

David Bounds

Cloud Native AI and Machine Learning on AWS

Use SageMaker for building ML models, automate MLOps, and take advantage of numerous AWS AI services

Premkumar Rangarajan
David Bounds

www.bpbonline.com

First published: 2023

Published by BPB Online

WeWork

119 Marylebone Road

London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55513-267

www.bpbonline.com

Dedicated to

*My wife, **Sapna Mohan Kumar**,*
*my son **Harivatsa Premkumar** and*
*the darling of our home, our cat **Phoebe**.*

Your loving presence is an encouragement and
inspiration that brings out the best in me.

— Premkumar Rangarajan

*My wife, **Michelle Webb**.*

I would rather share one lifetime with you than
face all the ages of this world alone.

— David Bounds

About the Authors

Premkumar Rangarajan is a Principal AI/ML specialist solutions architect at Amazon Web Services and has previously authored the book Natural Language Processing with AWS AI services. He has 26 years of experience in the IT industry in a variety of roles, including delivery lead, integration specialist, and enterprise architect. He helps enterprises of all sizes adopt AI and ML, and in his spare time dabbles as a guest lecturer helping students shape their careers.

David Bounds is a Senior Solutions Architect at Amazon Web Services based out of Atlanta, Georgia. With more than 25 years in IT ranging from front-line support to Site Reliability Engineering, David has a focus on operationalization of workloads. He has an unrelenting passion for bringing AI/ML tools and capabilities to engineers of all experience levels and backgrounds. When he is not iterating over haiku-generation models, he runs, works on his 3-D printer, or watches movies with his boxer, Darby. He is preferential to action movies.

About the Reviewers

Radhakrishnan (Krishna) Gopal is a cloud evangelist, seasoned technology professional, and mentor with over 23 years of Industry experience. He has been helping organizations transform the way they work by adding value and helping to deliver on-time development practices.

Krishna is experienced in all major cloud hyper-scalers including AWS, Azure, and Google cloud. He holds two patents from the United States Patent and Trademark Office and many certifications & accreditations in the areas of Cloud, Data & Analytics. He served as a technical reviewer of the book "The Definitive Guide to Modernizing Applications on Google Cloud" from Packt. Krishna is passionate about building cloud-based, data-intensive applications at scale to deliver business value through cloud adoption and innovation.

Vamshi Krishna Enabothala is a Sr. Applied AI Specialist Architect at AWS. He works with customers from different sectors to accelerate high-impact data, analytics, and machine learning initiatives. He is passionate about recommendation systems, NLP, and computer vision areas in AI and ML. Outside of work, Vamshi is an RC enthusiast, building RC equipment (planes, cars, and drones), and also enjoys gardening.

Acknowledgement

There are a few people we want to thank for the continued and ongoing support they have given me during the writing of this book. First and foremost, we would like to thank our families for continuously motivating and supporting us in writing the book. They have been our pillars of support during this journey.

We are grateful to the wonderful customers and partners of Amazon Web Services AI/ML technologies whose feedback helped design some of the solutions that are discussed in this book. We gratefully acknowledge Mr. Gopal Krishna and Mr. Vamshi Krishna Enabothala for technically reviewing and validating the ideas we have discussed in the book.

Our gratitude also goes to the team at BPB Publications for ongoing support, encouragement and patiently answering our relentless questions on the publishing process, and in continuously educating us in this momentous journey.

Preface

The absorption of ML and AI into existing business processes is pretty successful. Data professionals and cloud engineers are eyeing solutions with cloud migration as part of the digital transformation strategy.

This book will present the readers with how data developers, data scientists, and cloud engineers can seamlessly drive the entire ML and AI on AWS, making maximum use of various AWS machine learning and AI services. In this book, we will create data lakes, prepare and train ML models, automate MLOps, prepare for maximum data reusability and reproducibility, and various other tasks of successful AI deployments.

The book covers use-cases demonstrating effective use of AWS AI/ML services. Readers will learn to leverage massive scale computing, manage large data lakes, train ML and AI models, deploy them into production, and monitor the performance of ML applications. The book also covers how readers can use the pre-trained models across various applications such as image recognition, automated data extraction, detection of images/videos, identifying anomalies, and more. Throughout the book, we use AWS capabilities such as Amazon Sagemaker, Amazon AI Services, frameworks such as Pytorch or TF.

This book is divided into **12 chapters** across three parts, namely Part 1 - Know your data, Part 2 - Whose model is it anyway, and Part 3 - To API or not to API. You will first read how ML and AI evolved in the last two decades, understand the foundational concepts of computational learning, and dive deep into what a ML workflow is and how the various stages in the workflow come together to help design, build and deploy a reliable, scalable and efficient solution to meet the most popular requirements for AI and ML today. You will learn by doing, and will be walking through python code samples, Jupyter notebooks, the AWS Management Console and APIs to build your solution through the various stages of the ML workflow. The chapter details are listed below.

Chapter 1 covers the evolution of ML and AI in the last couple of decades, and an introduction to the ML workflow that enterprises use today to build world-class ML applications. We will follow this with an introduction to the AWS AI/ML stack and briefly delve into some of the major services that address key stages of the ML workflow. These are the services that we will cover in the rest of this book.

Chapter 2 will introduce the importance of data and its role in building ML applications. We will learn why "data is the new oil" and learn to leverage data to unleash the tremendous potential in using ML to transform enterprises. We will look at the various ways data can be collected, harvested, curated and stored in a S3 Data Lake in AWS, the different options to analyze and make sense of the data, and finally prepare it for ML training and inference. We will follow step by step instructions in executing these tasks using easy to follow python code examples, Amazon SageMaker Jupyter notebooks, AWS Lambda and more.

Chapter 3 covers one of the most important aspects of designing a ML solution which is feature engineering. Features define the inputs and the output of the ML model and will have to be considered very carefully as they can directly influence the success of the model predictions. In this chapter, you will learn what features are, why they are important, how to select the features that matter, and different techniques for collection, preparation and usage. We will learn with comprehensive coding examples, how to apply feature engineering techniques for different ML domains and problem types. Finally, we will use AWS Glue to automate the feature engineering tasks.

Chapter 4 covers how to build a highly optimized collection of data pipelines using a combination of analytics and ML to harvest what we need for model training at scale but without the overhead of having to deal with large data volumes. We will show you with coding instructions how to build your own data pipelines for different types of ML use cases.

Chapter 5 discusses choices in algorithm selection when designing ML models. If you want to run deep learning you use neural networks but that's not a given. So, what are algorithms and neural nets? And how do you decide what to use? Why? What's the difference between the two anyway? Learn how to build powerful predictive models harnessing the power of Math and Science with detailed code examples in this chapter.

Chapter 6 will cover model training. By this time in the book, you know your ML problem, you know your data, you have decided what type of training you need to do and have the algorithm or the neural net ready. It's now time to see what happens when the rubber hits the road. For your specific ML problem, you first have to define the metric you have to evaluate your model against, and then iterate through a combination of model training and tuning to reach or exceed that metric. In this chapter, we will see with actual code examples how to build, train, tune, and evaluate your ML model using Amazon SageMaker, Jupyter notebooks and Python code samples.

Chapter 7 will cover AutoML techniques including advanced ML automation that is in high demand and becoming increasingly mainstream using Amazon SageMaker Autopilot and Amazon SageMaker Canvas. You will learn how to use SageMaker Canvas with its point and click visual interface and SageMaker Autopilot under the hood to automate the entire ML workflow all at the click of a button. We will also spend some time reviewing the notebooks automatically generated by SageMaker Autopilot to understand how the ML workflow was executed.

Chapter 8 covers model deployment strategies. By this time, you have already completed model training and are ready to use it to make predictions. You will learn how to predict at scale in real-time and also run batch predictions for high volume accumulated datasets. You will understand what needs to be considered when making deployment decisions, and what are some of the cloud-based deployment strategies to maximize efficiency and minimize costs.

Chapter 9 covers model inference design and implementation. You will learn considerations for how your model endpoint will be used, what is the scale it needs to support, and how you can build containers with model libraries to serve inference requests. You will learn how to do all of this in the most cost-effective way and at scale.

Chapter 10 covers sensory cognition AWS AI services. By this time in the book, you would have gained comprehensive knowledge and experience in building and deploying powerful ML models that operate at massive scale to solve diverse challenges across industries. In this chapter we will pivot from ML to AI to understand how we can leverage the AWS AI services. Using just an API call that provides ready-made intelligence for a variety of common use cases, we will build end-to-end solutions in a fraction of the time it took us to build ML workflows on our own.

Chapter 11 covers AWS AI services for industrial automation. You now understand how to use AWS AI service API calls to run powerful models for common sensory cognition use cases, and you will continue in this chapter to explore some of these AI services that are hugely popular in the area of industrial automation specifically in predictive maintenance, and time series forecasting

Chapter 12 covers MLOps. You are now an expert in knowing and applying AI/ML with AWS for your enterprise use cases. You know how to approach, design, build and deploy AI and ML solutions. But perhaps the most important step is learning how to prepare it for production deployment, and setting up an automated CI/CD pipeline for faster go-to-market cycles across your business analyst, development, data scientist and operations team. In this chapter we will use Amazon SageMaker pipelines and the SageMaker data science SDK to learn how to build end-to-end MLOps pipelines and in the process also learn some best practices for ML model development and deployment.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/oq1o1sb

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Cloud-Native-AI-and-Machine-Learning-on-AWS**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Table of Contents

CHAPTER 1
Introducing the ML Workflow

Introduction

Machine Learning or ML is simply the art and science of teaching machines to learn patterns in data (numbers, images, text, and so on) and using these patterns to either predict an outcome or infer a label. In comparison to regular programming that enables machines to perform a particular task really well, ML teaches machines to be intelligent and derive previously unseen relationships in data. Consider, for example, the process of withdrawing cash from your bank Automated Teller Machine (ATM). You insert your card, the machine asks for your PIN, you enter the PIN, the machine validates the PIN and if it is valid, presents you the banking menu. This is an example of regular programming. The bank's ATM software understands your card and the account number it's linked to, knows your PIN and validates if it's correctly entered, and finally, displays the banking menu. The software has been designed to perform this task repetitively and accurately.

However, the software is not learning from how users interact with its functions, nor is it trained to look for patterns in usage. So, even if this card is inserted into an ATM located in an entirely different country compared to where is it regularly used from, the ATM software will happily disburse cash if the PIN is found to be valid. This, of course, is a problem because it might not be you using the ATM but someone who has stolen your card details and is accessing your account without your knowledge. And this is where ML is really helpful; it can understand that there is something abnormal with this usage pattern, and it triggers an alert. Not only anomaly detection, ML models can be trained to detect fraud in banking and credit card transactions, classify images, predict movie genres, detect sentiment in text, detect objects from images and videos, recognize audio, track player movements in sports, and even drive vehicles autonomously.

In this book, we will learn with practical examples and detailed instructions how to build a ML workflow step by step and apply it for a variety of business use cases. But first, we need to understand some foundational concepts of ML and Artificial Intelligence or AI: how did it come about, how did it evolve, how did cloud computing make ML more accessible and how to leverage the Amazon Web Services (AWS) AI/ML services to build our ML solutions. We also need to learn some of the best practices to determine if ML is the right fit for a business problem, and how to navigate the plethora of options available to build ML solutions today. We will cover these introductory concepts in the following sections.

Structure

In this chapter, we will dive deep into the following topics:

- Evolution of AI and ML
- Approaching an ML problem
- Overview of the ML workflow
- Introducing AI and ML on AWS
- Navigating the ML highway

Objectives

The goal of this chapter is help raise awareness on what ML and AI mean at the foundational level and serve as a refresher for key concepts. Whether you are hearing about ML for the first time, or you are an experienced practitioner, this chapter will help you quickly come up to speed and be better prepared to get your hands dirty applying concepts and building your own ML solutions in the subsequent chapters. This chapter is primarily theoretical, so it is more reading and assimilating than doing. When we get to *Chapter 2, Hydrating Your Data Lake*, which will be hands-on, we will provide instructions on how to get started with AWS.

Evolution of AI and ML

It all started with the game of chess! It is indeed interesting as to why computer engineers often use chess to test their designs for intelligent computers. Why not say a game of monopoly or scrabble? It might probably be because chess is a game of skill rather than of recall or chance. The author is not an avid chess player but understands how the pieces move and what the rules are. In chess, the strategy of the consequence of the move is more important than the tactical act of the move itself. You need not only determine your move but also predict your opponent's retaliatory move, and then firm up strategies to counter that and so on. In this analogy, ML determines the play strategy, and regular programming executes it. But hold on, we seem to be digressing!! How is chess and ML connected? To understand this, let's time travel back to the 1960s, when the Beatles ruled the music industry.

While the world was in rapture with hits like "Yesterday" and "Let it be", Richard Greenblatt created the first artificial intelligence (AI) chess program to play humans in a tournament setting in a different corner of the world. Even though we have been fantasizing about creating an intelligent machine to play chess as early as the 18th century, the seeds for Greenblatt's work were sown in 1950, when Alan Turing, often considered the father of computer science and AI, wrote the world's first program to play chess. Since GUI was unheard of in those days, Turing himself had to play the role of the computer and translate moves based on outputs from his algorithm. The keyword here is "algorithm", which is at the core of how ML works. In the typical sense, an algorithm is an automation of a set of dependent actions that process inputs and derive outputs. Another important element without which an algorithm cannot work, is the data that constitutes the inputs and is a consequence of the outputs. Algorithms define what the task is and how should it be performed. In the context of chess, the algorithm would be aware of the rules, the roles of the various pieces, and how a move is to be executed. The data, on the other hand, would be the current position of the various pieces on the board, and the knowledge associated with historical game play strategies and moves that players train on.

As you can see, algorithm and data must work in tandem to create a successful play; this is true with ML as well. The learning aspect of ML is the algorithm powering through the data to uncover a generalized function that can help it interpret how the inputs (a set of columns in the data) can help derive an output (a target column we want to predict). When a function has been arrived at with adequate accuracy, the algorithm is considered learned and is exposed to previously unseen data of input columns only, to predict the target column or the output. And this is at the crux of machine learning or ML. Simple, right? It may appear so because all we seem to be doing is taking a few mathematical functions, coding them programmatically, throwing a bunch of data at them repeatedly, hoping that a pattern may be established. For example, a Decision Tree is a very common algorithm in ML used for classification (predict a class) or regression (predict a value) problems. In simple terms, Decision Trees (shown in *Figure 1.1*) are how a flow chart of decisions lead to an outcome, which can either be a class or a value based on the problem we are trying to solve. It primarily helps a model understand the generalization in the data that leads to a particular decision or a target value with the model learning from features that are tagged with decision labels in the training dataset:

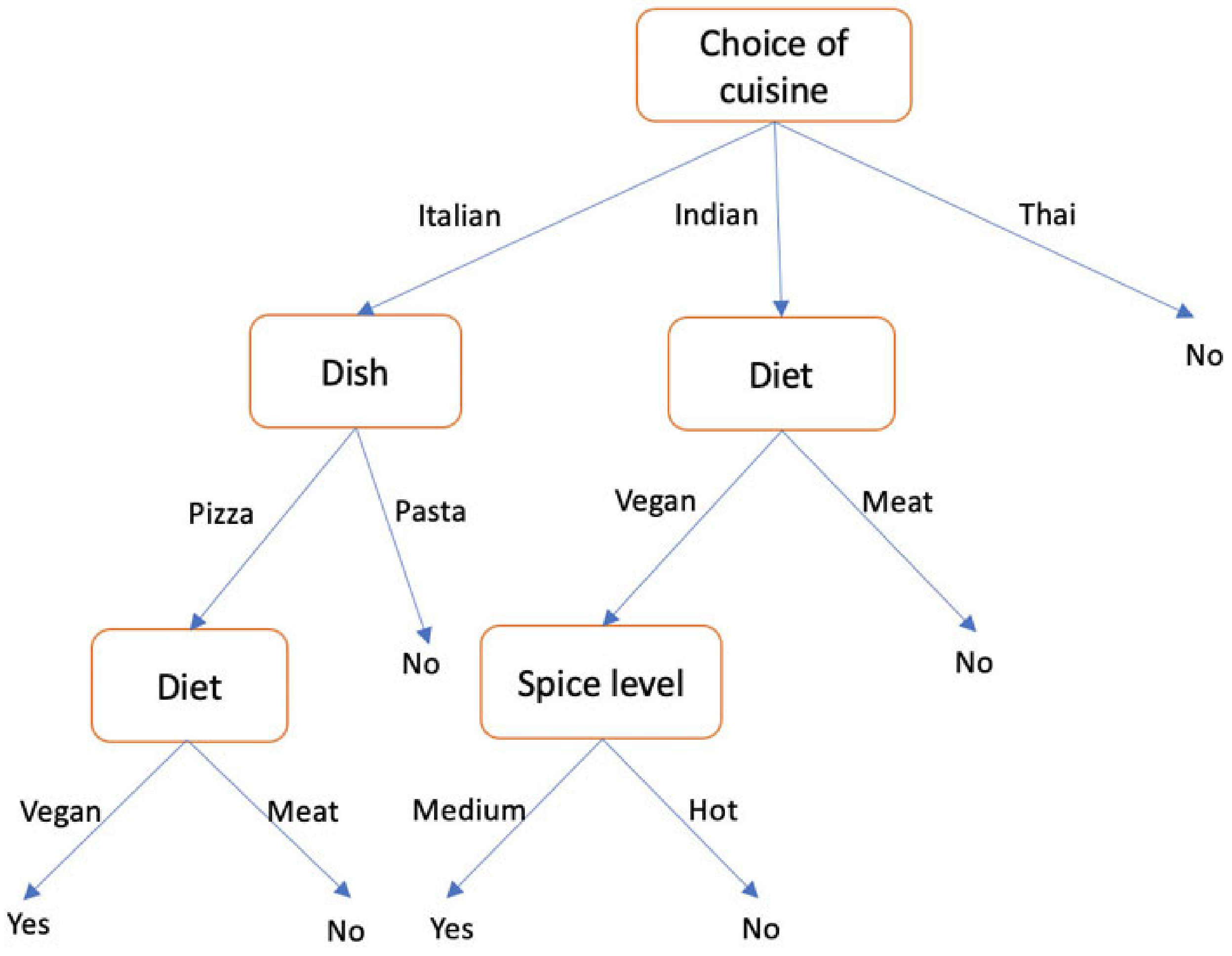


Figure 1.1: *Decision Tree to determine cuisine choices*

There are, however, major challenges to be overcome at every step of the way until we get to a working version of a ML model. The biggest of them is infrastructure or the hardware power needed to accomplish a ML training task. Until the mid-2000s, before the cloud gained popularity, only the largest and the most well-funded enterprises had the capability to set up infrastructure at the scale needed to run ML training. Costs were driven up by the need for petabyte scale data storage, high-performance computing hardware (lots of high-powered cores in a cluster) for distributed architecture to reduce training time, and processing power at the scale of TERAFLOPS (trillion floating point operations per second) or even petaflops. According to this interesting article (**https://visual.ly/community/Infographics/technology/cost-data-storage-through-years**) accessed on February 2022, the cost of 1MB of storage evolved from $10,000 in 1956 to $0.0006 in 2005. And yet, for large-scale ML training that needed 100s of terabytes or petabytes of data, storage costs alone were in the $100K to $500K range in the 2000s. Add this to the price of processing power, and it cost $82 per GFLOPS in 2003 (according to this article **https://aiimpacts.org/wikipedia-history-of-gflops-costs/** accessed in February 2022), which equals approximately $100K in compute alone; quickly, the initial investments for ML became unmanageable except for large enterprises.

That's why ML remained a peripheral technology until the advent of cloud computing. AWS is a pioneer in democratizing the power of high-performance computing and making it accessible for everyone, which paved the way for ML to become mainstream. Today, companies of all sizes and across industries are harnessing the power and capabilities of ML for a diverse set of use cases, and the list of ML applications keeps growing. With pay-as-you-go pricing and the agility of spinning up infrastructure in minutes globally, the possibilities are endless. For example, it only costs $28.15 to run an hour of ML training on a p3.16x.large instance that's one of the most powerful high-performance instances in the cloud today with 8 V100 Tensor Core GPUs, 64 vCPUs, and 25 Gbps of networking performance.

When we continue to examine some of the other challenges ML faced in the past, we see why AWS's cloud native AI/ML services have gained popularity and have been widely adopted across enterprises. After solving infrastructure challenges, organizations struggled to hire experts with the skill sets required to create ML algorithms, that is, a combination of advanced mathematics and programming. We are talking about scientists with PhD in Computer Science who were actively working in applied research. They were very rare to find (and still are), and affordability was an issue for all but big enterprises. The consequence of both the infrastructure and skill set requirement led to ML projects taking a long time to complete. Additionally, the data that was key to run ML training was often in diverse and disparate sources internal and external to the organization, and the technical capability to bring this data together for analysis and manipulation was both time-consuming and costly.

AWS AI/ML capabilities are built to remove these obstacles and make it easy for organizations to adopt ML and transform their business. First, we saw how infrastructure provisioning is easy and cost-effective with pay-as-you-go pricing and agility. To solve issues with ML skill shortages, AWS offers the AI services layer that provides pre-trained ML models available behind an API call for common ML use cases like image classification, object detection, and natural language processing. To make it easy for those developers and data scientists who want to build and train ML models without having to worry about complex mathematical functions, Amazon SageMaker is an AWS service that provides the capability to execute end-to-end ML workflow tasks within an integrated development environment built for ML and comes with algorithms. And since the AI/ML services are natively integrated with the rest of the AWS services, it is very easy to ingest data from various sources, transform and analyse the data, derive features, run pre-processing tasks, and then use it for ML training. We will see how to do this in detail in the subsequent sections and throughout this book.

As ML started becoming mainstream with the cloud gaining popularity, the scope of ML use cases increased in complexity with demand for more intelligent applications like autonomous driving, cancer detection, and climate research. This led to the development of deep learning models that could learn billions of parameters. This needed a layered architecture that could learn complexities in the data more deeply, so the neural network was born. It is deep learning and neural networks that powered the AI revolution in the past decade. Neural networks or **Artificial Neural Networks** (**ANN**) are a mathematical representation of how the human brain is understood to work in a compute context. Neurons are stacked in a combination of input, output and hidden layers. Each neuron in a hidden layer is provided an input of the sum product of outputs from the previous layer. Random co-efficients called as weights are also assigned to the neuron in the current layer to which a static bias is added and this combined value is passed to an activation function that determines the output that will be passed to the next layer. The following image shows a representation of how neural networks are constructed:

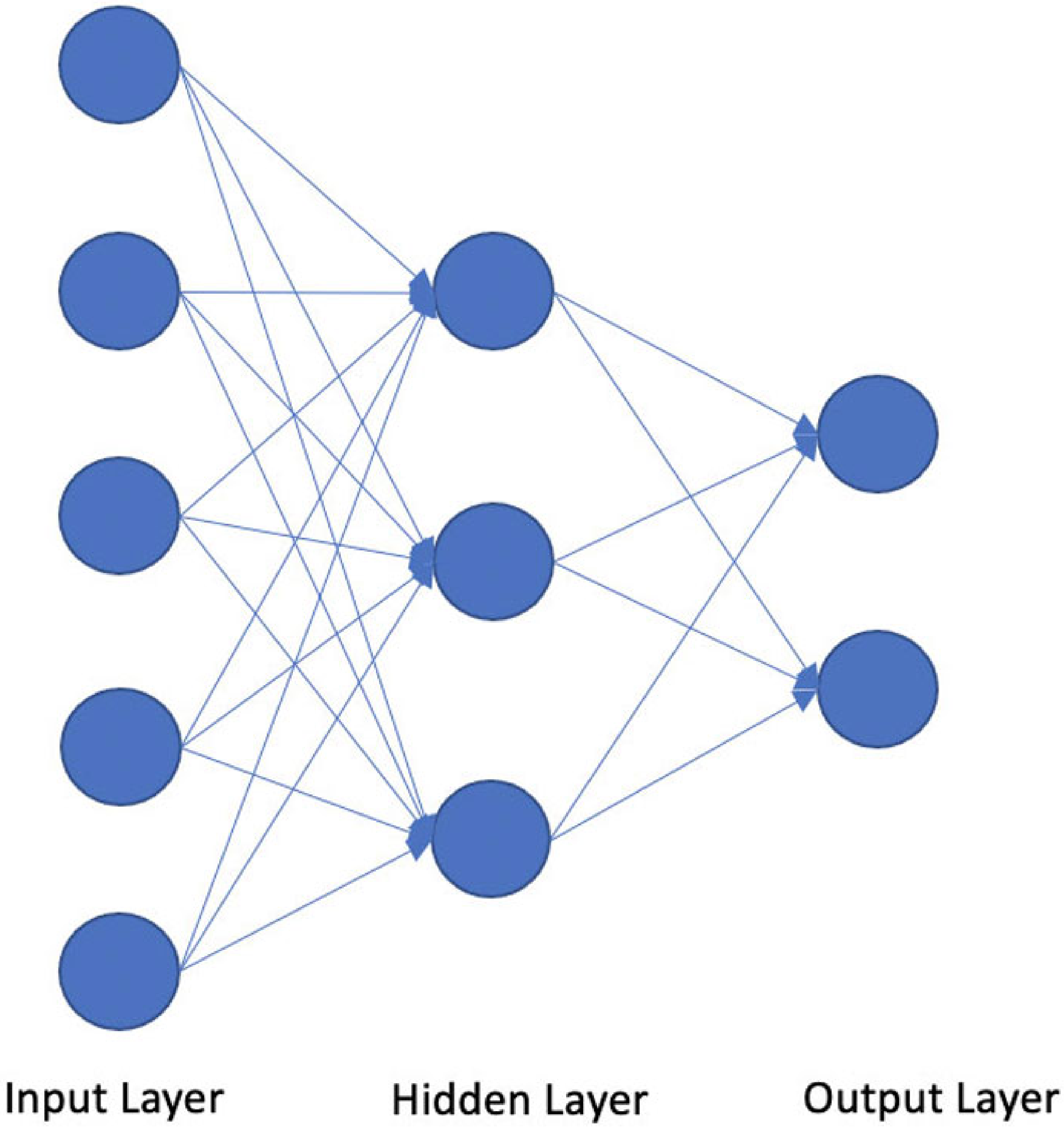


Figure 1.2: *A simple three-layer neural network for binary classification*

There are different types of activation functions, such as **Rectified Linear Unit** (**ReLU**), **Sigmoid and Hyperbolic Tangent**, that are used for deep learning networks; ReLU is the most commonly used today. There are different types of neural networks available based on the type of deep learning task, with some of the most popular ones being **Convolutional Neural Networks** (**CNNs**) for image processing, **Recurrent Neural Networks** (**RNNs**) for speech/text processing, and sequence to sequence that use a combination of RNNs for language translation tasks. For in-depth information about neural networks and activation functions, refer to the training curriculum offered by AWS Machine Learning University at **https://aws.amazon.com/machine-learning/mlu/**.

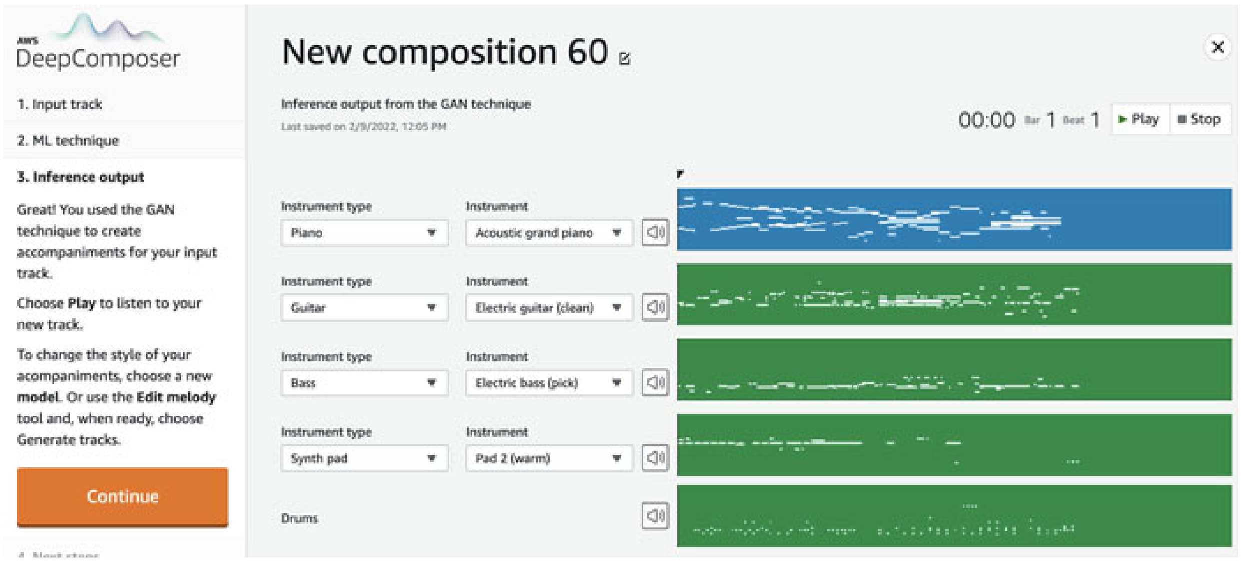
While deep learning was gaining momentum, companies were also developing ML frameworks to make it easy for developers and data scientists to assemble the algorithms and neural networks for ML training. Some of the most popular ML frameworks are **Apache MXNet** (**https://mxnet.apache.org/versions/1.9.0/**), **Google Tensorflow** (**https://www.tensorflow.org/overview/**), and **Facebook's PyTorch** (**https://pytorch.org/**). There are other frameworks like **Scikit Learn** (**https://scikit-learn.org/stable/**) for common ML tasks and Deep Graph Library (**https://www.dgl.ai/**) for graph-based neural networks that are also quite popular among ML enthusiasts. All these frameworks are supported by AWS for building your ML solution. Based on the type of ML requirement, you can use a combination of algorithms, neural networks and frameworks. For your reference, the following table shows how the ML frameworks are applied for common ML use cases. These should be used as suggestions, and depending on a scenario, there are multiple ways to do the same thing:

ML topic	ML use case	ML framework/library	Algorithm/Neural network
Computer Vision	Image classification	TensorFlow Keras, MXNet, PyTorch	CNN
	Object detection	TensorFlow object detection, MXNet GluonCV, PyTorch Detectron2	CNN, Single Shot Detector (SSD), You Only Look Once (YOLO)
	Instance segmentation	TensorFlow object detection, MXNet GluonCV, PyTorch	Mask R-CNN
Natural Language Understanding	Automated speech recognition	TensorFlow, MXNet LSTM, PyTorch	RNN, Long Short-Term Memory (LSTM)
	Machine translation	TensorFlow, MXNet, PyTorch	RNN, Neural Machine Translation,

			Sequence2Sequence
Natural Language Processing	Sentiment analysis	TensorFlow Keras, MXNet Gluon, PyTorch	RNN, CNN, Support Vector Machines (SVM)
Text classification	TensorFlow Keras, MXNet Gluon, PyTorch	Amazon SageMaker Blazing Text, FastText, SVM, RNN, Random Forest	
Tabular data	Regression	TensorFlow Keras, MXNet, PyTorch	Linear Regression, Decision Trees, XGBoost, Multilayer Perceptron (MLP)
	Classification	TensorFlow Keras, MXNet, PyTorch	Linear Regression, Decision Trees, XGBoost, MLP

***Table 1.1:** Mapping of ML use cases to frameworks and algorithms*

The alert reader in you might be wondering why we mentioned the music band Beatles in an earlier paragraph. How is music and ML connected? You will be surprised to know that the reach of AI and ML is not limited to industrial applications; it encompasses the artistic dimension as well. For example, **AWS DeepComposer**, a service to educate the capabilities of ML, not only enhances music composed by humans but also creates entirely new notes and tunes from a basic input piece that you provide. Take a look at the following figure, which shows how you can compose music using ML with DeepComposer:



New composition 60

Inference output from the GAN technique

Last saved on 2/9/2022, 12:05 PM

Great! You used the GAN technique to create accompaniments for your input track.

Choose **Play** to listen to your new track.

To change the style of your accompaniments, choose a new **model**. Or use the **Edit melody** tool and, when ready, choose Generate tracks.

Instrument type	Instrument
Piano	Acoustic grand piano
Guitar	Electric guitar (clean)
Bass	Electric bass (pick)
Synth pad	Pad 2 (warm)

Figure 1.3: *AWS DeepComposer Music Studio*

DeepComposer uses a variety of ML techniques, such as **Generative Adversarial Networks** (**GANs**), **Auto Regressive Convolutional Neural Network** (**AR-CNNs**) and Transformers to do this. GANs enable the creation of new music in the form of accompaniments to the original track by setting up two neural networks to compete against each other with one network using unsupervised learning and the other using supervised learning. With AR-CNNs, music generation is viewed as a time series problem, with notes from the past along with all embellishments that currently exist used to predict the notes that need to be generated. Transformers, introduced in 2017, are an advancement that overcomes constraints with CNNs and RNNs. The transformer architectrure allows data to be processed in parallel and accounts for variable length sequences and long-term dependencies in the data, which is important in music. You can check out **https://aws.amazon.com/deepcomposer/** if you are interested to learn more and try it out yourself.

Now that we have piqued your interest, it is time to take a deeper look at understanding where to start. ML has been around for a while, and the choices available make it overwhelming to determine the right roadmap for your ML project. That's why the approach is as important as the execution. In the next section, we will learn how to approach a business challenge and determine whether ML is the right fit for a solution, what some of the best practices to make this determination are, whether ML is indeed the right approach, and what the next steps are.

Approaching an ML problem

After having learned some core concepts of ML and how cloud computing made it popular, we need to ask ourselves if ML is the answer to every problem out there. Considering how accessible it is and the excitement of what's possible, we may be tempted to use ML for every challenge we are asked to solve, but this is not a rational approach, as ML projects take time because of their iterative nature and have cost implications depending on what's needed. So, the first question we need to ask is, "Is this really a ML problem"?

Suppose you are an AI/ML specialist at a big retailer, and your manager wants you to build an ML solution for inventory optimization as the retailer has recently suffered losses due to inadequate demand/supply management within stores and among suppliers. So, you first set out to investigate if this situation really requires ML. You identify two facets to this problem: issues with the inventory allocation system in correctly matching demand from the stores with the suppliers, and the lack of demand forecasting capability within the organization. When you perform a root cause analysis of the inventory allocation issue, you notice that there were demand spikes from the stores many times during the past year due to panic buying, the pandemic, and other external factors that severely impacted the price elasticity of several commodities. This caused the inventory allocation system to directly escalate to the suppliers who were already impacted by ongoing labour issues and input shortages.

Your investigation shows that you don't need ML to fix the inventory allocation issue. The system just needed to be tuned to moderate inventory replenishment in a much more streamlined fashion, along with a threshold established to alert supervisors whenever demand spikes are encountered. But you also realize that ML can add value in this situation by means of a demand forecasting solution that can predict inventory needs. And to take this forward, you start learning about Amazon Forecast (**https://aws.amazon.com/forecast/**), a fully managed automated ML powered time series forecasting service from AWS. We will cover this in *Chapter 11, AI for Industrial Automation*, of this book.

So, how do you know what ML is necessary for and what can be solved by regular programming? To answer this question, let us first understand the different types of ML, how they map to business challenges they can solve, and where do they fit in the broader context of solution development. You can think of ML as a collection of tools, processes, frameworks, techniques, and APIs that aid in the development of applied Artificial Intelligence solutions. Any entity that does not have consciousness but can learn and evolve from its inputs is considered to be Artificially Intelligent. In this context, ML enables AI to be what it is. But it does not stop there. As we saw in a previous section, with the advent of cloud computing and ML becoming very popular, technology evolved into Deep Learning (DL) to solve bigger problems and enable complex learning across billions of parameters. While ML dabbled with frameworks and algorithms, DL brought on the heavy equipment with neural networks, distributed training, data parallelism, and so on. And it was DL that took a previously relatively weak attempt at AI (for example, a robotic sounding voice in the case of text-to-speech) and made it more human-like, thereby truly revolutionizing the field of AI. Refer *Figure 1.4* that shows relationship between AI, ML and DL:

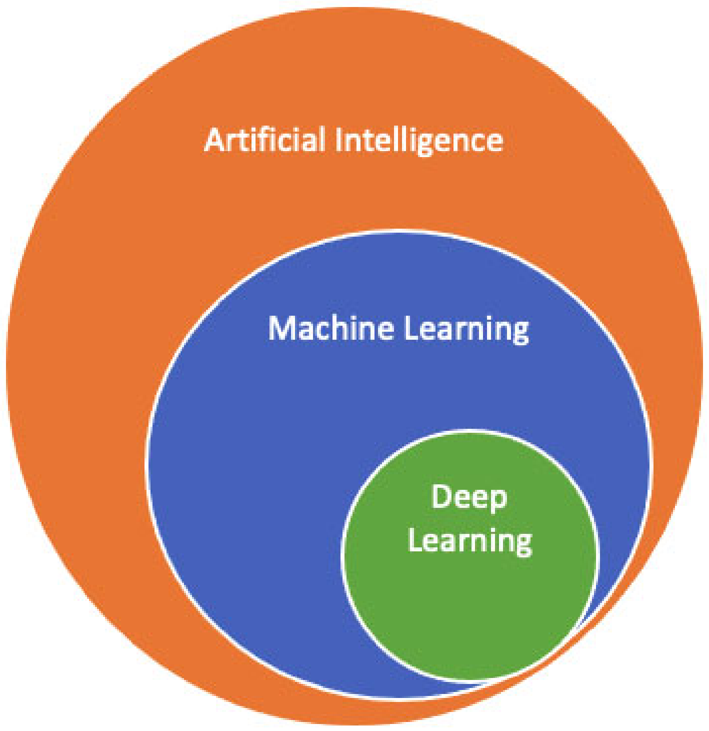


Figure 1.4: *AI, ML and DL*

In essence, DL is a subset of ML, and they both contribute to the realm of AI. When you build neural networks to train large-scale models, you are performing deep learning. An example is text summarization with natural language processing using Transformer models. When you are using algorithms to perform small- to medium-scale tasks like using Decision Trees to predict a particular outcome, it is ML. ML and DL can be further categorized based on how they learn a generalization approach. We will discuss the most common categories in the following subsections.

Supervised Learning

When you train a model by providing labeled data for the target column you want it to predict, it is called supervised learning. You are telling the model to learn an approximation function for predicting the target value using a set of labelled input values as a training dataset. Suppose the problem is to predict house prices; you train a model with historical data of house prices along with input values like the size of the house, location, schools nearby and accessibility. Since the historical data has the house price as labels in the training dataset, it can approximate the relationship between the input values and the house prices.

Example Algorithms: XGBoost, Linear Regression, Naïve Bayes, Support Vector Machines

Common Uses: sentiment analysis, movie genre prediction, cancer prediction, churn prediction

Unsupervised Learning

When you train a model without labels in the data, it is called unsupervised learning. This technique is primarily used for clustering or anomaly detection requirements. Suppose you have a large demographic dataset of people and want to determine the major factors that influence the grouping of individual datapoints; clustering is a good way to determine that. If there are commonalities in the data based on age, race, gender, employment status, and so on, the clustering will uncover that. Or if you want to determine what causes surge pricing to activate in your rideshare dataset, an unsupervised learning algorithm can help you isolate those spikes. Apart from anomalies and clustering, unsupervised learning applies to the nearest neighbours, dimensionality reduction and more.

Example Algorithms: K-means Clustering, K-Nearest Neighbours, Principal Component Analysis, Random Cut Forest

Common Uses: Fraud prediction, anomaly detection, data distribution analysis

Reinforcement Learning

Here, the learning is achieved by an iterative process of interaction and incentivization between an agent and its environment. A reward function determines the incentivization. An agent in an environment is rewarded if it takes the right action (from a list of actions available to the agent), and it is penalized if it takes the wrong action. There are two networks that work together to implement the learning. The agent's value network is set up to maximize reward and hence, the value. The policy network learns the outcome for every input action that the agent takes. This type of learning is similar to the real-life situation of training a dog. If it behaves well, you give it a snack; if it does something bad, you punish it. This way, the dog learns good behaviour.

Example Algorithms: Proximal Policy Optimization, Deep Q Network

Common Uses: Autonomous driving, gaming, drones

Now that we understand different types of ML and what they can be used for, let's tackle the question that you will face as you start helping your own customers adopt ML: "to ML or not to ML". To make your decision easier, the following table addresses common situations when you will face this question and the considerations that will help you make the right decision for your projects:

Customer need	Probable use case	To ML or not to ML?	Why?
Trading portfolio tracking	Track the portfolio position across several trades and provide status updates on gains and losses	No ML	ML works based on derivatives and predictions. This use case is an ask to keep track of gains and losses across trades. Regular programming can help with this.
Predicting stock prices	Predict future stock prices	ML	Here, we are asked to predict the future values based on historical performance, which is what ML excels at.
Determining tax codes for product categories	Look up tax codes based on product information	No ML	There is no need to predict or derive anything here. It is a lookup function of a pre-determined value based on a combination of input values.
Automating inspection of machine parts	Monitor, detect and alert defective machine parts	ML	This needs intelligence to analyze pictures or videos to determine the condition of what's

			being looked at, so ML is needed.
Determining new product features to support	Identify market interest for new product lines or features	No ML	While ML can assist in modeling information, the actual decision needs to be made by a logical analysis of the underlying factors.
Utility usage tracking and billing	Track customer usage of utilities and generate monthly bills	No ML	There are meters that can track usage and provide this information. The billing amount can be calculated based on rate categories and usage.

Table 1.2: *To ML or not to ML*

"If I had one hour to save the world, I would spend fifty-five minutes defining the problem and only five minutes finding the solution."

– Albert Einstein

A lot to absorb, right? Maybe. But that was us just scratching the surface of ML's potential. We are now about to get into the weeds of what ML really is, and how to build world-class ML applications that are secure, reliable and scalable. A methodical approach is helpful when setting out to solve a complex ML problem that often comprises a multitude of moving parts. Not only does it provide guidance on what needs to be done, but it also informs us how to do it, what is the sequence that must be followed, what needs to be prioritized and what are the tools that help us achieve the task. In the next section, we will learn about the ML workflow, the key components of ML solution building and how to use it to deliver success in your ML projects.

Overview of the ML workflow

If you have never worked on an AI/ML project before, you are in the right place. If you have worked on a ML project before but want to learn how to execute it end to end, like some of the largest enterprises, you are still in the right place. Simply put, everything starts, lives in and ends with the ML workflow. If there is such a thing as the 10 commandments here, that would be defined by the ML workflow. Why is it so important? Because it gives you a blueprint of everything you need to implement your project. Think of it as a checklist of tasks with prescriptive guidance on how to go about designing and implementing ML solutions. Without further ado, let's dive right in. Refer *Figure 1.5* for an overview of the ML workflow:

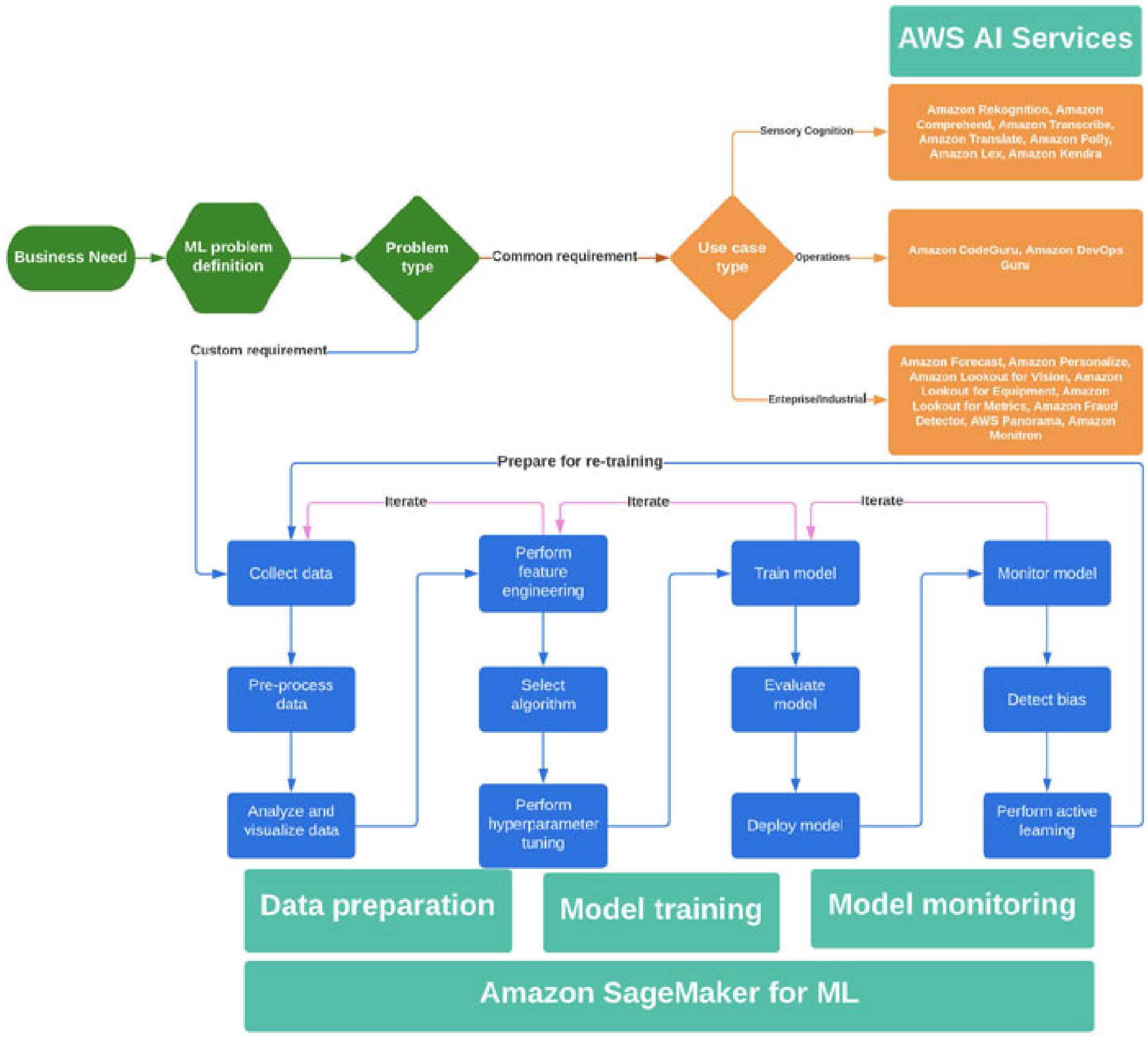

AWS AI Services
Sensory Cognition
Amazon Rekognition, Amazon Comprehend, Amazon Transcribe, Amazon Translate, Amazon Polly, Amazon Lex, Amazon Kendra
Business Need
ML problem definition
Problem type
Common requirement
Use case type
Operations
Amazon CodeGuru, Amazon DevOps Guru
Custom requirement
Enteprise/Industrial
Amazon Forecast, Amazon Personalize, Amazon Lookout for Vision, Amazon Lookout for Equipment, Amazon Lookout for Metrics, Amazon Fraud Detector, AWS Panorama, Amazon Monitron
Prepare for re-training
Iterate
Iterate
Iterate
Collect data
Perform feature engineering
Train model
Monitor model
Pre-process data
Select algorithm
Evaluate model
Detect bias
Analyze and visualize data
Perform hyperparameter tuning
Deploy model
Perform active learning
Data preparation
Model training
Model monitoring
Amazon SageMaker for ML

Figure 1.5: *AI/ML Workflow with AWS services*

As you can see, it's a pretty crowded picture with many boxes and arrows and a lot text. But there is a logical simplicity to the whole thing that we will attempt to highlight in the following subsections, which will hopefully demystify this picture for you.

Common versus custom ML

First things first. A business need must exist that prompts the creation of an ML project. Do not be tempted to try out ML in your organization just because it's a cool thing to do or because of the fear of missing out. If you just want to learn about the technology, of course, go ahead. You may expand on an existing business need to improve efficiency using ML. For example, when solving an inventory management issue, you may use ML to implement demand forecasting to increase efficiency in your operations. For more details, you can refer to the previous section, Approach*ing an* ML problem:

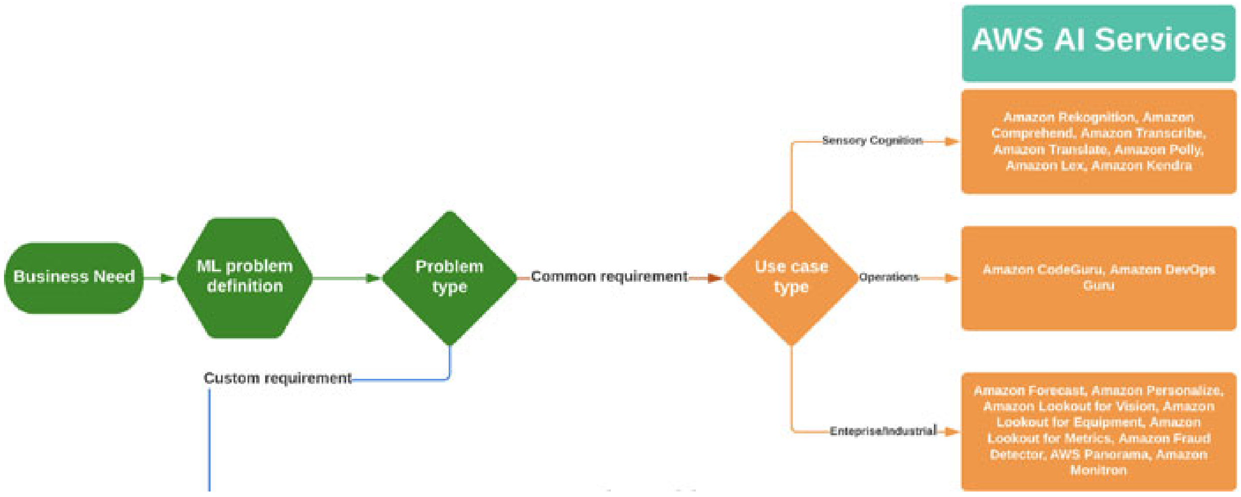

AWS AI Services
Business Need
ML problem definition
Problem type
Common requirement
Custom requirement
Use case type
Sensory Cognition
Operations
Enteprise/Industrial
Amazon Rekognition, Amazon Comprehend, Amazon Transcribe, Amazon Translate, Amazon Polly, Amazon Lex, Amazon Kendra
Amazon CodeGuru, Amazon DevOps Guru
Amazon Forecast, Amazon Personalize, Amazon Lookout for Vision, Amazon Lookout for Equipment, Amazon Lookout for Metrics, Amazon Fraud Detector, AWS Panorama, Amazon Monitron

Figure 1.6: *AWS AI services for common solutions*

Once there is a clear need and we have adequately understood the problem, we must determine if it can be addressed by a common ML solution or one that needs custom development. The following table guides you on the common problem types for which AWS provides pre-trained ML models called AWS AI services by means of an API that you can easily include in your application without having to worry about model training. We will discuss the AWS AI services in the next section:

ML problem type	ML sub type	ML use case
Sensory Cognition	Computer Vision	Image classification, object detection
	Speech	Speech to text
		Text to speech

	Text	Sentiment detection, entity recognition, text classification
		Machine translation
		Text extraction
		Intelligent search
	Chatbots	Conversational interfaces
Enterprise/Industrial	Business tools	Fraud detection
		Recommendations
		Forecasting
		Anomaly detection
	Industrial intelligence	Predictive monitoring
		Predictive analytics
	Healthcare	Text analytics
		Speech analytics
		Healthcare analytics
Operations	Development intelligence	Code optimization
		IT operations optimization

Table 1.3: *Common ML problems and use cases*

If the ML problem needs a custom solution, that is, if it cannot be addressed by a common solution discussed earlier, we will have to execute the steps in the ML workflow, starting with data preparation and followed by model training, deployment and optionally, monitoring. That is why our ML workflow is grouped into three major areas: data preparation, model training and model monitoring. Each of the tasks share functions among these groups. Note, as shown in the following image, that as you proceed to each of the tasks in the groups, you have to repeatedly iterate back, as indicated by the pink arrows:

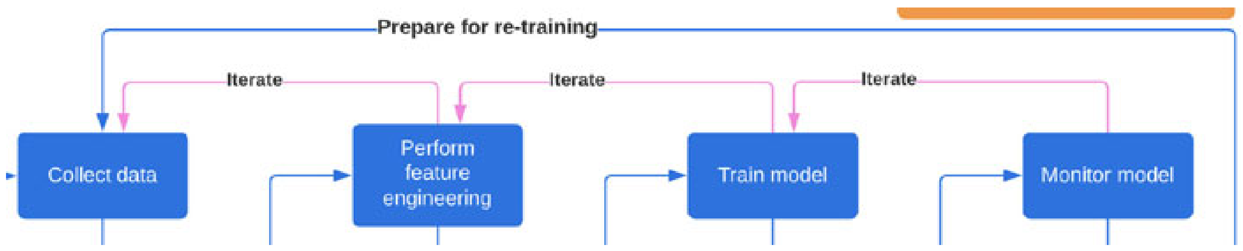


Figure 1.7: *ML workflow is an iterative process*

This is because ML is, by default, an iterative process. You start by setting a target accuracy (or other metrics based on the algorithm) your model needs to reach, and you work through the tasks iteratively to reach that accuracy. Sometimes you think you have all the data you need, and the algorithm seems to be right for the data and the problem, but your model may end up overfitting during training. It means you need to come back to data collection to introduce errors/outliers in the data; that's why we have the repeated iterations. We will discuss the tasks of the ML workflow in the next few subsections.

Data preparation

In this book, we will be showing you how to perform each step with the help of live code examples, but in short, data collection is the process of determining what data you need to train a model, identifying where this data resides in your enterprise, and determining how you can source this data into a Data Lake storage on AWS. In the next chapter, we will discuss how you can build your own data lake on AWS and how you an ingest data into this data lake. Data lakes are built on Amazon S3 object stores that can grow to any scale and provide eleven 9s of durability for your data. For more details, you can refer to **https://aws.amazon.com/products/storage/data-lake-storage/**. In the following image, we see the main steps that we need to perform as part of the data preparation phase of the ML workflow:

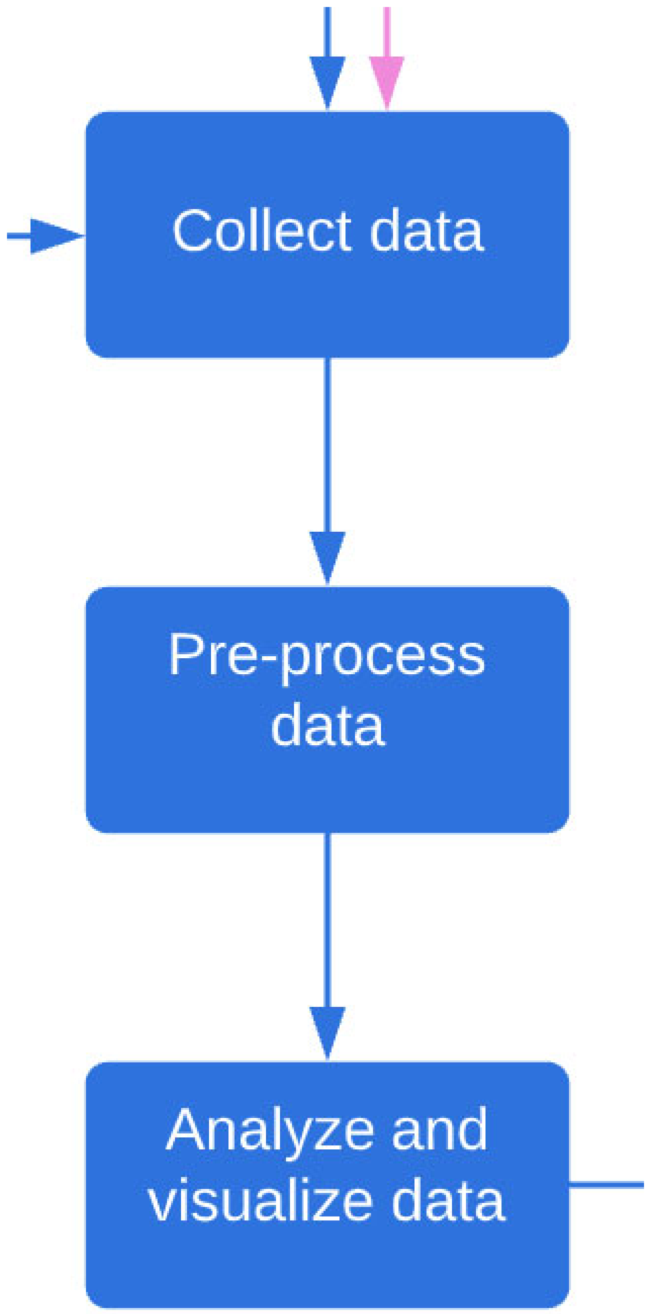


Figure 1.8: *Prepare data*

Now that you have the data you need in your data lake, the next step is to pre-process the data. When data arrives in a data lake, it is not transformed from its source format and is not immediately usable for your ML training. You typically perform transformations at the time of data extraction. So, this step is crucial in getting you the right quality of data. If your ML task type is Supervised Learning, this is the step when you add labels (values for the target column that you want the model to predict) to your dataset if they do not already exist. AWS provides purpose built services to help you with data pre-processing tasks, such as Amazon SageMaker Ground Truth (**https://aws.amazon.com/sagemaker/data-labeling/**) for data labelling, AWS Glue for managed ETL (**https://aws.amazon.com/glue**), Amazon EMR (**https://aws.amazon.com/emr/**) for map reduce, SageMaker Data Wrangler (**https://aws.amazon.com/sagemaker/data-wrangler/**) that provides pre-built transformations, and SageMaker Processing (**https://docs.aws.amazon.com/sagemaker/latest/dg/processing-job.html**) that provides a fully managed infrastructure for running data transformation, scaling, normalization and other data manipulation tasks using its built-in Scikit Learn container or PySpark applications.

In order to validate that our data pre-processing task has indeed yielded the right collection of columns/features that best correlates with what we want to predict and that we have the requisite volume, range, and scale of data for our ML training, we need to visualize the data, understand what the outliers are, determine the distribution topology and prune the data so that we get it to the shape we need. This is the next task in our ML workflow, and it is called data analysis and visualization. AWS provides Amazon QuickSight (**https://aws.amazon.com/quicksight/**), a BI service for creating intelligent dashboards from your data, including the ability to create histograms, heatmaps, scatter plots, box plots, and so on.

NOTE: There is an interim decision to be made before you proceed with the next task when you are implementing your ML workflow. If you want to fully automate the task of feature engineering, model training and evaluation, you will use an AutoML option like the Amazon SageMaker Autopilot (https://aws.amazon.com/sagemaker/autopilot/) or AutoGluon (https://auto.gluon.ai/stable/index.html). We will cover this in detail in *Chapter 8, Let George Take Over*. On the contrary, if you like the flexibility that you have when training and tuning the model yourself, you can proceed with the rest of the tasks in the workflow.

Training preparation

At this stage in the workflow, you have your data transformed and ready, and you have a reasonable idea of the model you want to train. The next set of tasks are shown in the following image:

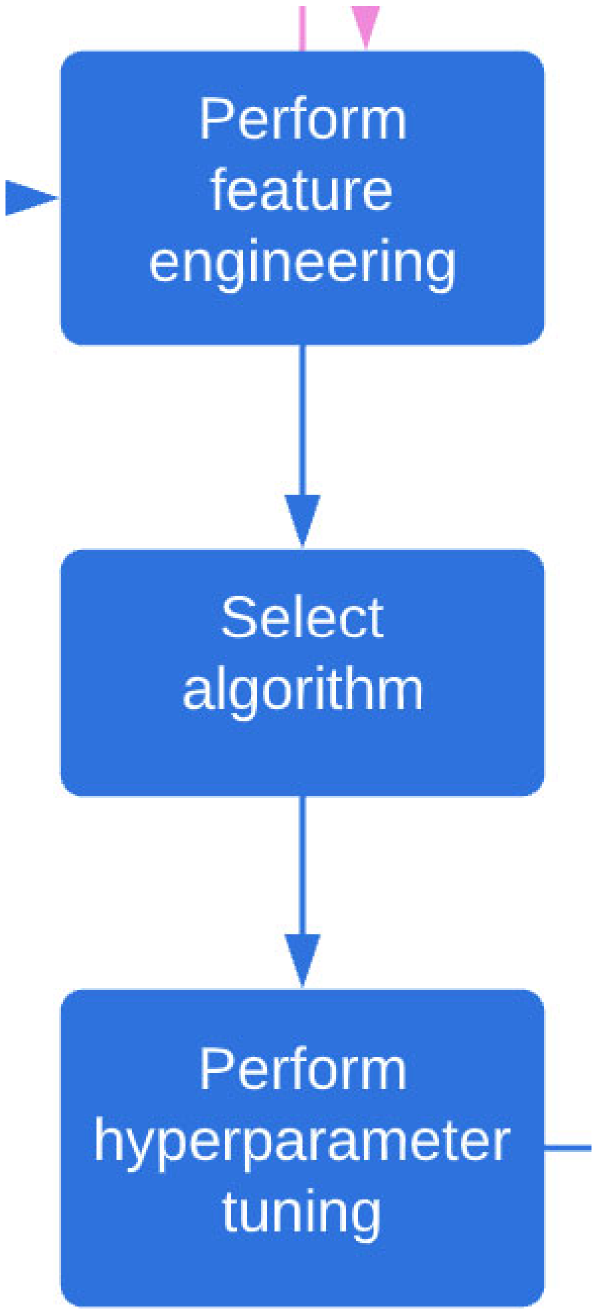


Figure 1.9: *Prepare for training*

Before you can begin training, you need to validate that the input data columns (also called **features**) in your training dataset are good enough for the model to approximate to the target values you want to predict. The technique we follow to do this is called feature engineering. Based on the problem and the data, there are different types of feature engineering tasks, such as scaling, normalization for tabular numerical datasets, stop word removal, stemming, lemmatization for text-based datasets, scaling, and resizing for images. You can also perform feature selection, removal, adjusting for missing values, and so on. AWS provides a feature to help with this as part of the Amazon SageMaker service; it is called Feature Store (**https://aws.amazon.com/sagemaker/feature-store/**). With Amazon SageMaker Data Wrangler and the Feature Store, you can integrate the data pre-processing and the feature engineering tasks, collect your features, and even reuse them for similar ML tasks.

We now get into an exciting area of the workflow: algorithm selection. To be fair, you might have already decided on an algorithm when you defined the ML problem. However, this is a good time to revisit that decision considering all that you know about your data now. And the choices are many. For example, if you are trying to train a classifier, do you want to use Amazon SageMaker's built-in Linear Regression or XGBoost, or do you want to bring in a Support Vector Machine, or Naïve Bayes or Random Forest? How do you know what will work best with your data? You may have an idea, but you need to be sure.

Hint: An alternative approach at this stage would be to use AutoML, which trains candidate models using many algorithms and hyperparameter (specific configuration parameters that define how a model learns) combinations and recommends the best performing model.

So how do you move forward? It’s simple. You iterate through different algorithms and select the best performing one based on the target accuracy you need. Remember that the keyword in ML is iteration!

The next step goes hand-in-hand with algorithm selection. They are not two separate tasks and are often executed jointly. Hyperparameter tuning or hyperparameter optimization is the process of training several candidate models with varying hyperparameter configurations (parameters that define how a model learns) to determine which combination of hyperparameters yields the best results for the algorithm and the data in meeting our target accuracy. Thankfully, Amazon SageMaker provides an easy-to-use API (**https://docs.aws.amazon.com/sagemaker/latest/dg/automatic-model-tuning.html**) that you can leverage to add hyperparameter tuning during model training.

Model Training

The next step is **model training**; for the most part this, can be considered a placeholder task because you might have already begun training earlier. If you used AutoML or if your hyperparameter tuning task yielded the desired result, you can skip this step and directly go to model evaluation and deployment. If not, use this step to try different combinations of algorithms and hyperparameters to reach the right model based on the target evaluation metric. The tasks to be performed during this stage are shown in the following image:

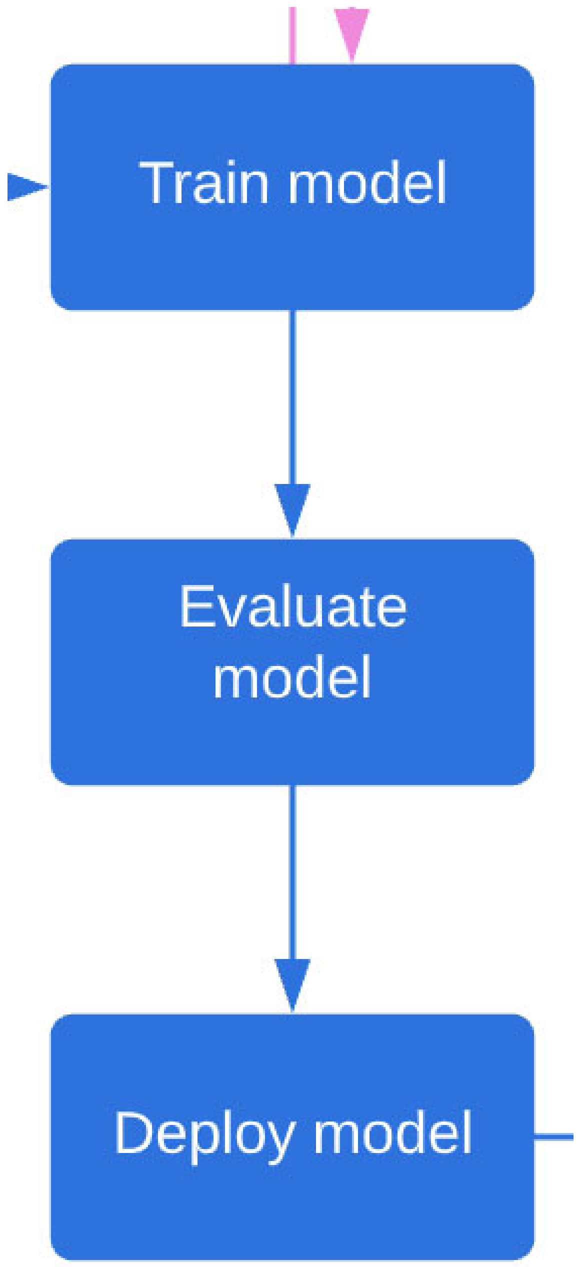


Figure 1.10: *Model training and deployment*

Also, in this stage, you would use the full scope of the training dataset, split up a percentage for blind test at evaluation time, and run the model training activity. Amazon SageMaker (**https://docs.aws.amazon.com/sagemaker/latest/dg/how-it-works-training.html**) provides an SDK (**https://sagemaker.readthedocs.io/en/stable/overview.html**) and several class of training instances (**https://aws.amazon.com/sagemaker/pricing/**)

that you can use to define your estimators, hyperparameters and run fit to start model training on managed infrastructure.

Before you trained your model, you split your data to create a blind test dataset of say 15% to 20% of your total training data. The model has not seen this blind test data yet because it was removed from your training dataset. In this step, we will run model evaluation on this data and compare model predictions with the original training labels (also called ground truth data) for the blind test data to compute metrics and evaluate if the model is good enough to be deployed to production or needs more work. We talked about accuracy as one of the metrics earlier. For classification problems, accuracy has the tendency to be influenced by distribution of the class labels and hence, is not a good neural metric. A better choice would be AUC (area under the receiver operating characteristic curve) that plots True Positive Rates and False Positive Rates (for more details, refer to **https://docs.aws.amazon.com/machine-learning/latest/dg/binary-model-insights.html**) for various classification thresholds and is more balanced. So, based on what type of model you are training, decide the evaluation metric, run your blind test data against the model, get the predicted labels, compare them with the original training labels, calculate the AUC, and compare it with your target value to complete the evaluation.

You are almost at the end of the ML workflow. At this stage, you have collected and prepared the data, identified features, selected the algorithm, performed hyperparameter tuning, and trained and evaluated your model. It appears that it meets the business need based on the evaluation metric. You are now all set to deploy the model into production for inference. Amazon SageMaker provides several options based on your infrastructure needs, inference types, and project requirements. With SageMaker deployment (**https://docs.aws.amazon.com/sagemaker/latest/dg/how-it-works-deployment.html**), you can set up endpoints (APIs with infrastructure provisioned for your trained model) for real-time inference. You can also use serverless inference with SageMaker if the traffic is unpredictable. You can also configure asynchronous inference and batch transform based on request routing frequency. There are additional choices to optimize inference costs by setting up a multi-model endpoint or creating an inference pipeline, and you can also configure autoscaling for your endpoints.

At this stage, you have completed the main tasks in your ML project development, and your model is deployed in production and serving inference requests efficiently. Over time, you will see a deviation between the baseline distribution the model was trained with and the empirical distribution of its inference response. This is called a **data drift**; it occurs if models see new data coming in that is different from the training dataset, or if the underlying business context changes. The next few tasks help with monitoring model performance and automating the mitigation by means of active learning frameworks.

Model monitoring

Traditionally, monitoring was considered an optional step, but it is becoming more mainstream these days due to the proliferation of AI and ML in every aspect of how businesses are run today. As shown in the following image, the ML workflow recommends monitoring, detecting drift and bias, and implementing active learning to ensure that models are aligned with the requirements:

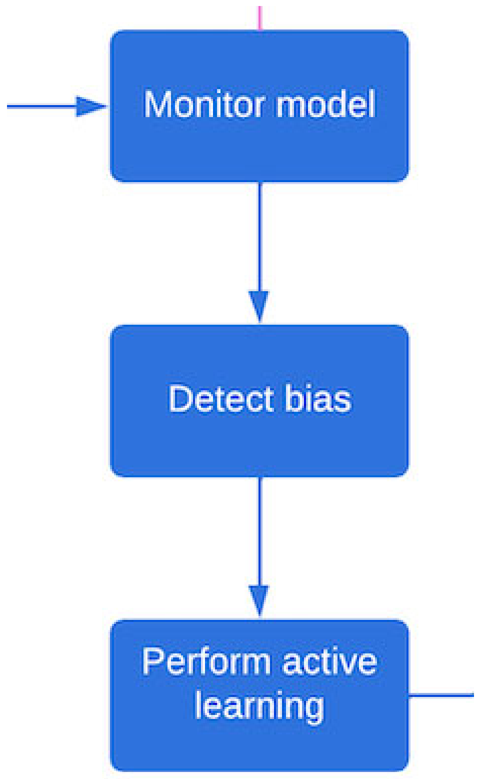


Figure 1.11: *Model monitoring stage in the AI/ML workflow*

You can easily set up a model monitoring solution with **Amazon SageMaker model monitor** (**https://docs.aws.amazon.com/sagemaker/latest/dg/model-monitor.html**) to monitor data quality at the feature or column level using statistics like mean, median, max, min, std_dev and more. You can also monitor model quality by comparing the evaluation metrics you originally calculated during training with the metrics the SageMaker model monitor can generate for you during inference based on the type of your ML problem.

An area that is being frequently discussed in the ML community is bias in models. Bias, as the term suggests, is the tendency to *lean* toward a particular outcome even though there is no influence toward this outcome in the data distribution. Amazon SageMaker provides a feature called **Clarify** (**https://aws.amazon.com/sagemaker/clarify/**) that helps you automatically monitor for bias in your models, throws alerts if it finds any bias and enables taking actions to correct it. You can monitor for bias drift, which is defined as the variance between the ground truth or baseline distribution and the distribution of the inference requests (or real-world data that the model did not see during training). You can also monitor for feature attribution drift, which is the variance in the correlation of specific features (or columns) in relation to the values in the predicted feature or column between the baseline distribution and the real-world data that the model sees during inference.

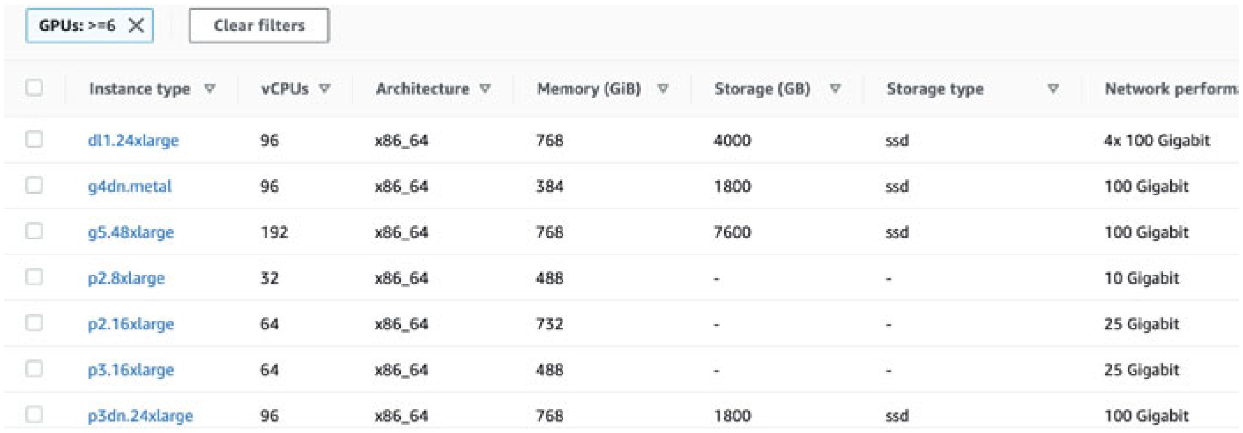
Both the model monitor and bias detection steps mentioned here alert us to issues with how the model is performing but do not autocorrect the model's behaviour. They do provide us with the options needed to automate the model update task. In addition to monitoring your models, there are instances at least initially to have a team of experts called as human in the loop for ML, reviewing/auditing your model predictions, and updating the values as required before sending the results to downstream applications. Amazon Augmented AI or Amazon A2I (**https://aws.amazon.com/augmented-ai/**) is a service that enables adding a human workflow to our ML process. You can take the corrected prediction values and add them back to the training dataset to automatically trigger retraining of your models. Typically, you set up a retraining pipeline that is triggered based on alerts from model monitoring or/and bias detection or/and the human workflow. This is called **active learning**.

That concludes our discussion of the ML workflow. It was a long section, and we thank you for your patience. Needless to say, knowing this workflow thoroughly will enable you to be highly successful as you build your own ML projects in your career. In this section, we reviewed the importance of a business need and the importance of determining if our ML problem was a common one or custom. For common requirements, we saw the different categories of solutions and saw how we have different AWS AI services to address specific areas. We also saw that in the case of a custom requirement, we need to navigate through the ML workflow, and we reviewed the steps to perform data preparation, feature engineering, model training, deployment, and model monitoring. We also learnt the names of the key AWS services for ML development. In the next section, we will dive a little deeper into the AWS AI/ML stack.

Introducing AI and ML on AWS

Welcome back dear readers!! We hope you are enjoying the book so far. In this section, we will introduce you to your toolbox that will help you become awesome ML engineers and data scientists. That's right, we are going to learn about the AWS AI/ML capabilities and offerings. We know that we have been name dropping some of these services in earlier sections and you may be wondering what all these service names and ML buzz words mean. Fear not, help is at hand. In this section, we will clarify (pun intended) what these services and features are for, and how to use them.

There are broad and deep offerings of services and features in the AI/ML space at AWS, and in order to make it easy for us to understand and select what we need, the capabilities are categorized into three layers stacked on top of each other, with the bottom layer offering maximum flexibility but minimal abstraction. This layer of capabilities is for ML experts who are interested in tinkering with ML frameworks, like to build their own algorithms and want to manage training and deployment infrastructure on their own. For compute options, AWS offers some of the most powerful compute instances available in the cloud today for ML training. The following picture shows compute instance options that have more than six GPUs each:

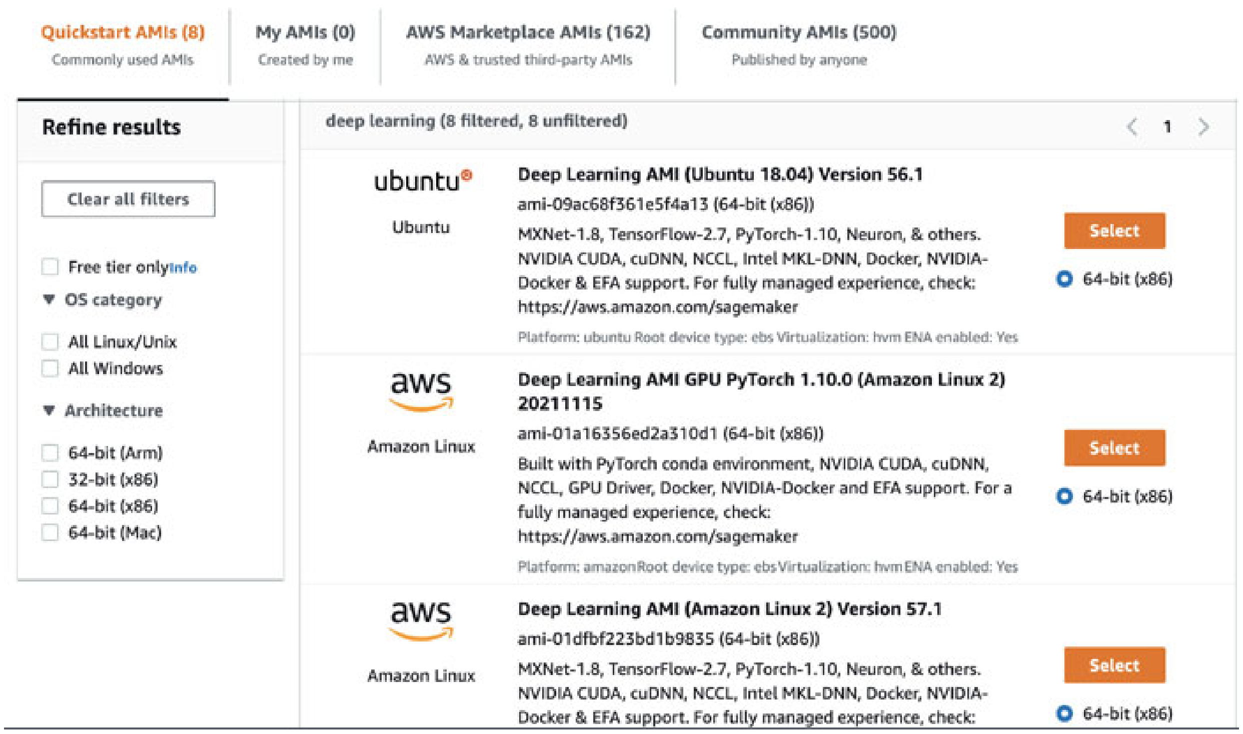


Instance type	vCPUs	Architecture	Memory (GiB)	Storage (GB)	Storage type	Network perform
dl1.24xlarge	96	x86_64	768	4000	ssd	4x 100 Gigabit
g4dn.metal	96	x86_64	384	1800	ssd	100 Gigabit
g5.48xlarge	192	x86_64	768	7600	ssd	100 Gigabit
p2.8xlarge	32	x86_64	488	-	-	10 Gigabit
p2.16xlarge	64	x86_64	732	-	-	25 Gigabit
p3.16xlarge	64	x86_64	488	-	-	25 Gigabit
p3dn.24xlarge	96	x86_64	768	1800	ssd	100 Gigabit

Figure 1.12: *Example list of GPU instances in AWS*

When we look at options for deep learning containers, AWS provides a comprehensive list of pre-built container images across all the major ML frameworks for various training needs. These are called Deep Learning Containers. A full list of available containers is provided in this GitHub repository at https://github.com/aws/deep-learning-containers/blob/master/available_images.md.

If containers are not your choice and you would rather deploy machine images on to your compute instances to set up your training, AWS has options for this with a selection of **Deep Learning AMIs** (**Amazon Machine Images**). The following picture provides an example list of these images. AWS Marketplace (a catalogue of tools and AMIs that you can purchase and deploy into your AWS account) and the community have an ever increasing list of options to choose from:



Deep Learning AMI (Ubuntu 18.04) Version 56.1
ami-09ac68f361e5f4a13 (64-bit (x86))
MXNet-1.8, TensorFlow-2.7, PyTorch-1.10, Neuron, & others. NVIDIA CUDA, cuDNN, NCCL, Intel MKL-DNN, Docker, NVIDIA-Docker & EFA support. For fully managed experience, check: https://aws.amazon.com/sagemaker
Platform: ubuntu Root device type: ebs Virtualization: hvm ENA enabled: Yes

Deep Learning AMI GPU PyTorch 1.10.0 (Amazon Linux 2) 20211115
ami-01a16356ed2a310d1 (64-bit (x86))
Built with PyTorch conda environment, NVIDIA CUDA, cuDNN, NCCL, GPU Driver, Docker, NVIDIA-Docker and EFA support. For a fully managed experience, check: https://aws.amazon.com/sagemaker
Platform: amazonRoot device type: ebsVirtualization: hvmENA enabled: Yes

Deep Learning AMI (Amazon Linux 2) Version 57.1
ami-01dfbf223bd1b9835 (64-bit (x86))
MXNet-1.8, TensorFlow-2.7, PyTorch-1.10, Neuron, & others. NVIDIA CUDA, cuDNN, NCCL, Intel MKL-DNN, Docker, NVIDIA-Docker & EFA support. For fully managed experience, check:

Figure 1.13: *Deep Learning AMIs*

In addition to compute, AWS offers custom designed chipsets to optimize your ML training (**https://aws.amazon.com/machine-learning/trainium/**) and inference (**https://aws.amazon.com/machine-learning/inferentia/**). So, the ML experts have a multitude of options to build their custom training and inference workflows at the bottom layer of the AWS AI/ML stack.

The **middle layer of the stack** is for ML practitioners who are willing to compromise on the flexibility a little bit to leverage the advantage of powerful abstractions that help their ML journey get to production quickly and easily. Amazon SageMaker is the main service that encompasses the middle layer with purpose-built features that address every step of the ML workflow we discussed in the previous section. Amazon SageMaker is a fully managed, secure, scalable and reliable service for all kinds of ML needs. To make it easier for you to think about this, the following table documents the key SageMaker feature for each step of our ML workflow. This is only a subset of features of Amazon SageMaker. As we go through the subsequent chapters in the book, we will learn about additional SageMaker features, such as using SageMaker Studio, which is the first IDE for ML development, SageMaker Autopilot for AutoML, and more. For a full list of SageMaker capabilities, you can refer to **https://aws.amazon.com/sagemaker/faqs**:

ML workflow task	Amazon SageMaker feature(s)	Feature overview
Data Labeling	GroundTruth	Automated data labeling as well as options to crowd source data labelers through Mechanical Turk, and also setting up your own private labeling team for image, text, tabular and other custom labeling requirements
Data Pre-processing	SageMaker Processing	Managed infrastructure and containers that support PySpark, Scikit-Learn and other packages for running various processing tasks, including data preparation, model evaluation, and feature engineering
Data Transformation	Data Wrangler	Provides 100s of pre-built transformation maps for common data processing needs in preparation for ML training
Feature Engineering	Data Wrangler and Feature Store	Provides pre-built transformations for many frequently used feature engineering tasks and the ability to store and reuse features using the purpose-built Feature Store for ML
Algorithm Selection	Built-in algorithms and bring your own	Provides support for 17 built-in algorithms across popular ML training use cases, and options to bring your own algorithms, or purchase from AWS Marketplace
Hyperparameter Tuning	Automatic Model Tuning	Easy-to-use API abstraction for hyperparameter tuning using random search and Bayesian regression methods
Model Training	Algorithms, Debugger, Experiments, Training containers	Apart from pre-built algorithms, support for a wide variety of deep learning containers, the ability to automate debugging and deep profiling of training jobs, and set up experiments and trials to track progress of training tasks
Model Evaluation	SageMaker SDK, SageMaker Processing	Chatbot implementation using ML with easy and intuitive interface to build highly intelligent conversational interfaces
Model Deployment	Production Variants, real-time endpoints, multi-model endpoints, inference pipelines, serverless inference, asynchronous	A wide variety of model deployment options, including serverless, asynchronous, real-time and batch, with the ability to perform A/B testing, Canary deployments, and automated version deployments

	inference, batch transform	
Model Monitoring	Model Monitor	Automated monitoring of models for data quality and model quality issues, and alerting
Bias Detection	Clarify	Automated monitoring of bias drift, feature attribution and alerting
Active Learning	Amazon A2I	Enables setting up a human workflow to review/audit and update ML predictions

Table 1.4: *SageMaker features mapping to ML workflow tasks*

At the top layer of the AWS AI/ML stack, we have services that are pre-trained ML models or/and provide easy-to-use abstractions for model training and deployment without the need for ML expertise to use these services. These are called AWS AI services. We covered this at a high level in the ML workflow section, but here we expand on the table to provide details on specific AI services that can be leveraged for common ML requirements:

ML problem type	**ML sub type**	**ML use case**	**AWS AI Service**	**Service overview**
Sensory Cognition	Computer Vision	Image classification, object detection	Rekognition	Pre-trained ML models available as an API for automating a variety of video and image processing tasks to infer intelligent insights
	Speech	Speech to text	Transcribe	Converts pre-recorded and streaming conversations to text transcripts using pre-trained ML models available as an API
		Text to speech	Polly	Creates life-like speech from text in various voices and inflections using pre-trained ML by means of an API
	Text	Sentiment detection, entity recognition, text classification	Comprehend	**Natural language processing** (**NLP**) service pre-trained for entity recognition, classification, sentiment, grammar and language detection with transfer learning capabilities
		Machine translation	Translate	Pre-trained ML service that provides language translations capabilities using an API

		Text extraction	Textract	Intelligent text extraction from a variety of document types and images using pre-trained ML models
		Intelligent search	Kendra	Enterprise search service powered by NLP indexes using pre-trained ML models with natural language query capabilities
	Chatbots	Conversational interfaces	Lex	Chatbot implementation using ML with easy and intuitive interface to build highly intelligent conversational interfaces
	Business tools	Fraud detection	Fraud Detector	Fully managed service for detecting and monitoring eCommerce fraud with a simplified interface for model building with no ML expertise required
		Recommendations	Personalize	Fully managed service for detecting and monitoring eCommerce fraud with a simplified interface for model building with no ML expertise required
		Forecasting	Forecast	Fully managed service for detecting and monitoring eCommerce fraud with a simplified interface for model building with no ML expertise required
		Anomaly detection	Lookout for Metrics	Fully managed service for detecting and monitoring eCommerce fraud with a simplified interface for model building with no ML expertise required
	Industrial intelligence	Predictive monitoring	AWS Panorama	Hardware and SDK for adding ML based intelligence to factory floor and on-premises camera systems
			Amazon Monitron	Sensors and pre-trained ML for industrial equipment monitoring and proactive maintenance
			Amazon Lookout for Vision	Custom ML pre-trained for industrial anomaly detection using computer vision
		Predictive analytics	Amazon Lookout for	ML service that provides intuitive capabilities for developing custom

			Equipment	models for detecting equipment anomalies with no ML expertise required
	Healthcare	Text analytics	Comprehend Medical	Pre-trained ML available as an API to extract insights from unstructured healthcare text inputs
		Speech analytics	Transcribe Medical	Pre-trained ML model with an API to convert speech to text, specifically designed for healthcare conversations
		Healthcare analytics	Healthlake	Intelligent content repository and healthcare analytics service with capabilities to query and analyze data using NLP, transcription, creating BI dashboards and running ML models at scale for a variety of business needs
	Developmen t intelligence	Code optimization	CodeGuru	Developer tool that uses ML to identify security issues and performance bottlenecks in code; no ML training required
		IT Operations optimization	DevOps Guru	Operations tool to help optimize operations and improve application availability using ML

Table 1.5: *AWS AI services for common ML use cases*

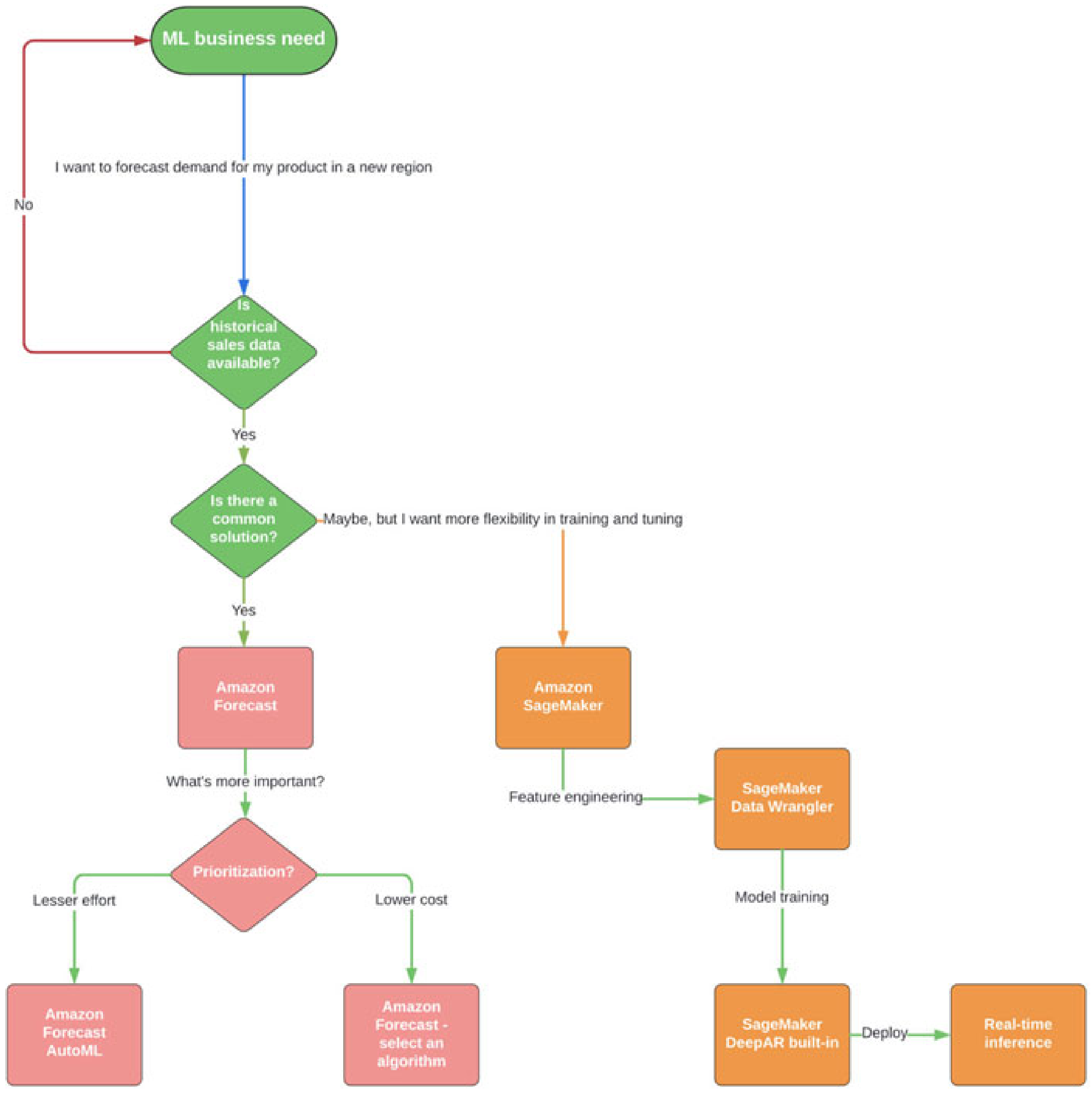
As you can see, we have a lot of choices and deep features within each AWS AI/ML service to help with our ML journey. It may seem overwhelming to begin with, but the good news is that you don't need to know all these services to build your ML solution. You just need to know what to choose when and deep dive into your selection of services and features for a specific project. We want to help you with the ability to make that determination, and we will cover that in the next section of this chapter.

Navigating the ML workflow

In this section, we will teach you how to prepare for ML solutioning. Considering the vastness of the AI/ML landscape, the ever-growing list of requirements it can solve for, and the countless choices we have in being able to design and develop a solution, it can be daunting when we start trying to organize the whole thing into a process. The ML workflow is designed to help you with exactly that. But even with the guidance of the workflow, how do you, for example, decide between using Amazon Comprehend for your NLP task and using Amazon SageMaker Blazing Text algorithm? What options do we have for time series forecasting, and how do we choose one over the other? In this section, we will take two of the most popular ML use cases (timeseries and NLP) and show you how to navigate toward a solutioning process for them. This is based on the author's own experience and curated over the years helping many customers adopt AWS AI/ML solutions. To make it fun, we will model this exercise using a Decision Tree algorithm. Feel free to convert this into a tabular dataset and train your own model to automatically predict the navigation pathway!!

Scenario 1 – Timeseries forecasting

In this scenario, once we have ratified the business need, the first thing we need to check is whether historical data is present, as it’s a timeseries problem. Without the historical dataset, this use case is non-starter. We then check whether it’s a common requirement, which it is, and discuss how the team wants to move forward. If the interest is to save time, we go with Amazon Forecast AutoML; if we want to use the common solution but try algorithms on our own, we go for the "select an algorithm" (**https://docs.aws.amazon.com/forecast/latest/dg/aws-forecast-choosing-recipes.html**) option in Amazon Forecast. On the other hand, if the team does not want to go with the common solution and wants to train on their own, we follow the SageMaker navigation pathway. It will take us through the steps in the ML workflow we discussed earlier. Both these options are depicted in the following image:



ML business need
I want to forecast demand for my product in a new region
No
Is historical sales data available?
Yes
Is there a common solution?
Maybe, but I want more flexibility in training and tuning
Yes
Amazon Forecast
Amazon SageMaker
What's more important?
Feature engineering
SageMaker Data Wrangler
Prioritization?
Lesser effort
Lower cost
Model training
Amazon Forecast AutoML
Amazon Forecast - select an algorithm
SageMaker DeepAR built-in
Deploy
Real-time inference

Figure 1.14: *Navigation pathway for timeseries forecasting*

Scenario 2 – Sentiment detection

For this scenario, we realize it's a text-based dataset and so, we understand it to be a Natural Language Processing (NLP) problem. Like before, we first check if it's a common requirement, which it is, and discuss the options to move forward. If the interest is to save time and cost, we go with Amazon Comprehend Detect Sentiment (**https://docs.aws.amazon.com/comprehend/latest/dg/API_DetectSentim**

ent.html), which is a ready-to-use API with a pre-trained model behind it. All we have to do is include the API call in our application. If the customer team wants to run their own training but are fine to use the common solution, they can train a custom classifier in Amazon Comprehend. If the team wants greater flexibility during model training, they can go the route of SageMaker and the BlazingText built-in algorithm (**https://docs.aws.amazon.com/sagemaker/latest/dg/blazingtext.html**). To enable active learning, we have the option to use Amazon A2I to configure a human workflow for reviewing model predictions and making updates, and this can be fed back for retraining the model with the updated dataset. These options are depicted in the following image:

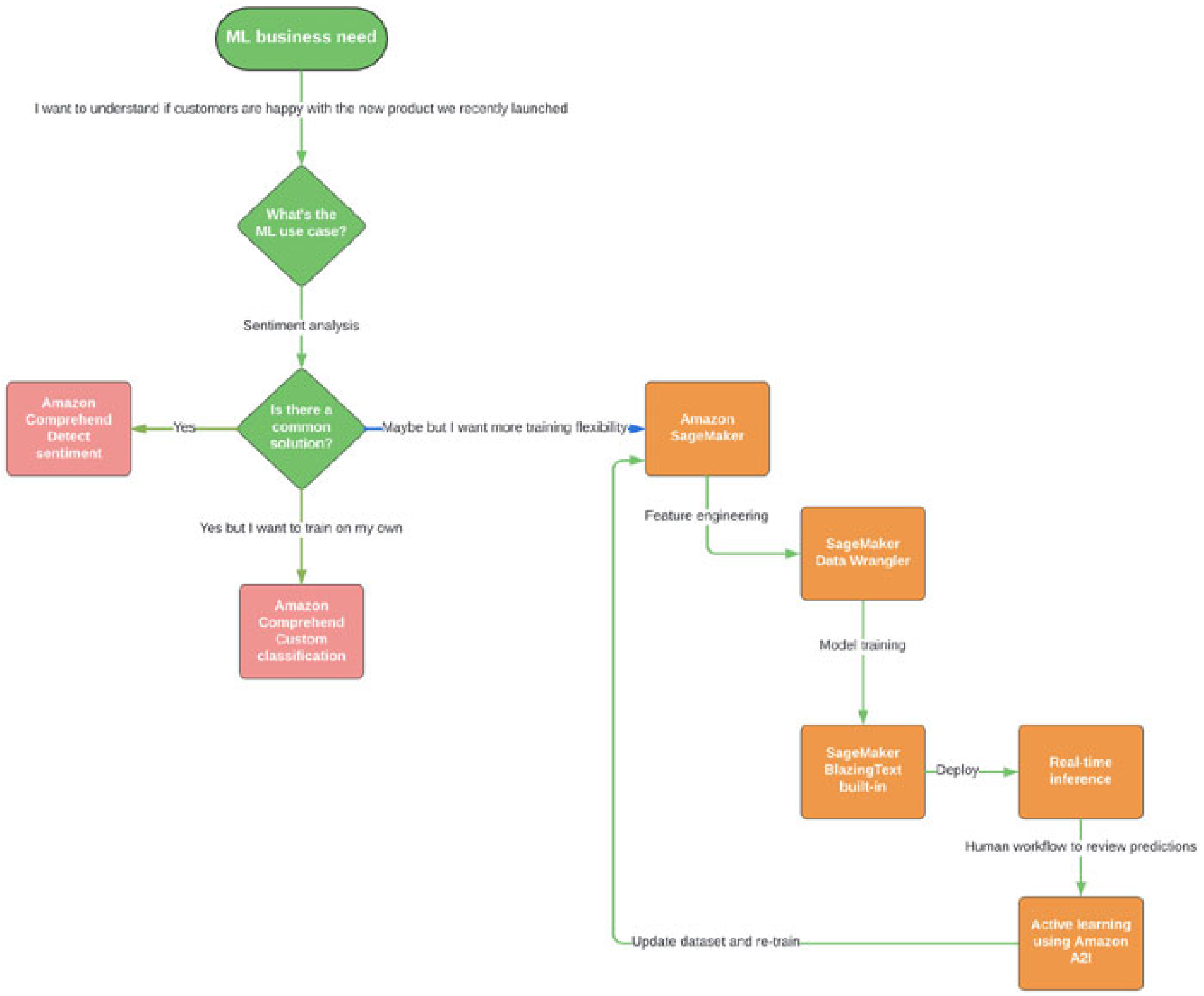

ML business need
I want to understand if customers are happy with the new product we recently launched
What's the ML use case?
Sentiment analysis
Is there a common solution?
Yes
Amazon Comprehend Detect sentiment
Maybe but I want more training flexibility
Amazon SageMaker
Yes but I want to train on my own
Amazon Comprehend Custom classification
Feature engineering
SageMaker Data Wrangler
Model training
SageMaker BlazingText built-in
Deploy
Real-time inference
Human workflow to review predictions
Active learning using Amazon A2I
Update dataset and re-train

Figure 1.15: *Navigation pathway for sentiment detection*

With time and experience, you will be able to reach a point when these navigation pathways are embedded into how you think. Keep in mind that the AI/ML landscape is evolving rapidly, so you need to keep yourself updated; this book is a great first step in that direction.

Conclusion

I know you are eagerly waiting to get hands-on with AI/ML on AWS and build some cool and innovative solutions. But just like in any academic education, you need to study a bit of theory before you can start practicing. And that was the purpose of this chapter, specifically the part about ML workflow, because that is what this whole book is about: how to build an end-to-end ML solution in the cloud by going through the various steps in the ML workflow. In the next chapter, we will tackle the first stage of the ML workflow with instructions and code snippets on what it takes to build an Amazon S3 data lake and hydrate the data lake by ingesting data from various sources. It is now time to get our hands dirty with some ML development. Fasten your seat belts!

Points to Remember

Here is a summary of the key points to remember from this chapter:

- We were first introduced to the concepts of ML and AI.
- We then learned how this technology evolved through the years, and how advancements in cloud computing led to the broad adoption of ML across organizations of all sizes and in all industries.
- We then read about deep learning, algorithms, neural networks and how you can compose music using ML with AWS DeepComposer.
- Subsequently, we discussed how to approach an ML problem and why the problem definition is one of the most important parts of an ML project.
- We followed this by discussing the ML workflow, the various stages and the tasks within these stages that guide us in running an ML project in depth.
- We discussed the different types of ML problems, use cases and how we can use AWS AI services with no ML expertise to build common

ML solutions.

- We then read about the AWS AI/ML stack, the objectives for each layer in the stack and how to leverage them based on our needs.
- We finally talked about solution navigation pathways and with the help of a couple of examples, learnt how to develop a navigation pathway for some popular ML use cases.

Multiple Choice Questions

Use these questions to enhance your knowledge of what we covered in this chapter.

1. **How are AI and ML related?**
    a. These are two independent techniques for training computers to perform new tasks
    b. ML is a subset of AI and is one way of how AI could be implemented
    c. AI is a subset of ML and is one way of how it could be implemented
    d. They are completely unrelated and mean different things
2. **What is an Artificial Neural Network?**
    a. It is an algorithm approach created by connecting neurons in layers, activated by mathematical functions for training machine learning and deep learning tasks.
    b. It is a technique to describe how the human brain works.
    c. It is a network of computers connected using neural approaches.
    d. It is an artificial computer.
3. **Which of these tasks is NOT a part of the ML workflow?**
    a. Feature engineering
    b. Data updates
    c. Algorithm selection
    d. Model training and tuning
4. **What is the most important task in the ML workflow?**

   a. Model training
   b. Inference
   c. Feature engineering
   d. Defining the ML problem

5. **You need to be an expert in data science or ML to use the AWS AI services. Is this true or false?**
   a. True
   b. False

Answers

1. **b**
2. **a**
3. **b**
4. **d**
5. **b**

Further Reading

- *Edwards, Benj. "A Brief History of Computer Chess." PCWorld, 6 May 2013, https://www.pcworld.com/article/451599/a-brief-history-of-computer-chess.html.*
- *"Richard Greenblatt." Richard Greenblatt - Chessprogramming Wiki, https://www.chessprogramming.org/Richard_Greenblatt.*
- *Alan Turing - Biography. Maths History. (n.d.). Retrieved February 13, 2022, from https://mathshistory.st-andrews.ac.uk/Biographies/Turing/*

CHAPTER 2
Hydrating the Data Lake

Introduction

Before we can get started training our machine learning model, we need to identify, assemble, curate, and prepare the data we will use as input. We could use that data where it rests, in various relational and non-relational data stores, data warehouses, object stores or file systems, but we know that we also need a data store of our own to keep our engineered feature sets, to share data between teams, and for it to act as input to our training. Many of the existing data stores require that we apply their schema when we write and have problems keeping up with the transfer speeds that machine learning needs to be efficient.

Enter the Data Lake. It is necessary not just for Machine Learning projects; a Data Lake is a data storage application on its own with features that make it suitable for Machine Learning projects. Primarily, it does not require that we apply any one schema to our data on write, meaning we can retrieve our data, transform it as we need for our projects, and then write it back transformed. It also uses fine-grained access controls that allow individual teams to have access and control of the data that they need.

Machine Learning projects do not need an object store like a Data Lake, but as an organization's AI/ML capabilities expand, using one ensures speed, agility, and flexibility. On Amazon Web Services, the kernel of the Data Lake is **Amazon Simple Storage Service** (**S3**). You will also use the **Amazon Key Management Service** (**KMS**) to encrypt the S3 Bucket at rest and the Amazon **Identity and Access Management** (**IAM**) service to control access to the bucket. Amazon has a pre-built solution for a Data Lake (linked in the *Further Reading* section) which your organization can use or extend to get started easily.

In this chapter, we will set up our Data Lake on AWS and then explore tools we can use to export data from its existing data stores and manage it on our Data Lake.

Structure

In this chapter, we will discuss the following topics:

- What is a Data Lake?
- Why does Machine Learning need a Data Lake?
- How do you create a Data Lake on AWS?
- How do you get data into your Data Lake?
- Additional considerations for Data Lake operations

Objectives

The goal of this chapter is to set the foundation for your Machine Learning projects by creating a repository for your source data, transformed data, and trained machine learning models. We will review the concept of the Data Lake itself and understand why it is a desirable storage method for Machine Learning projects. We will also review the methods for initial data moving to your Data Lake, often called hydration, and the different tools that can be used for the task. Finally, we will uncover methods for analyzing the data in your Data Lake and the similar services we can use to transform your data in preparation for your Machine Learning Projects.

To try the examples in this section, refer to the Technical Requirements section in *Chapter 1: Introducing the ML Workflow*, to sign in to the AWS management console, execute the steps create a SageMaker studio domain, and execute cloning the repository to SageMaker Studio to get started. Click on the folder that corresponds to this chapter number. If you see multiple notebooks, the section title corresponds to the notebook name for easy identification. You can also passively follow the code samples using the GitHub repository provided at the beginning of the book.

Chapter Scenario

Consider a database engineer who is part of a large enterprise company. They are solely responsible for the maintenance, management, and tuning of an on-premises legacy, monolithic relational database. They have kept this database running smoothly for over a decade, carefully curating and protecting it from abuse, but they have been given a new set of requirements that have left them unsure on how to proceed. They have been directed to create a schema-on-read Data Lake that includes fine-grained access controls, contains structured and unstructured data, keeps a version history of the data written to the Lake, is encrypted at rest, and supports analytics on the stored data without needing to specify a schema across the different data sets.

For this scenario, it is important to note that the database engineer is a Virgo.

A quick search across your trusted sources reveals a lot about Data Lakes and how they enable analytics, Machine Learning, data analysis, and other activities, but very little about how to manage the Lake itself.

The Data Lake

At its simplest, a Data Lake is a data repository. You could use a database, data store, storage device, or any other repository as a Data Lake, but there are several factors that make object storage optimal.

The first part of a Data LakeData Lake is the ability to store data in its raw or unaltered form. This enables schema-on-read versus applying a schema on writing to the lake like you would for a relational database. It also means that various teams can use the Data LakeLake to access the original data, the data at various stages of transformation, and in its final form ready to be ingested into their projects. Additionally, it means that any form of that data along that journey can be used as input for projects by that team or other teams.

This makes your data objects artifacts unto themselves, just like your code objects. These objects can (and should be) be versioned, packaged, and managed alongside your code or machine learning projects. Refer to *Figure 2.1*:

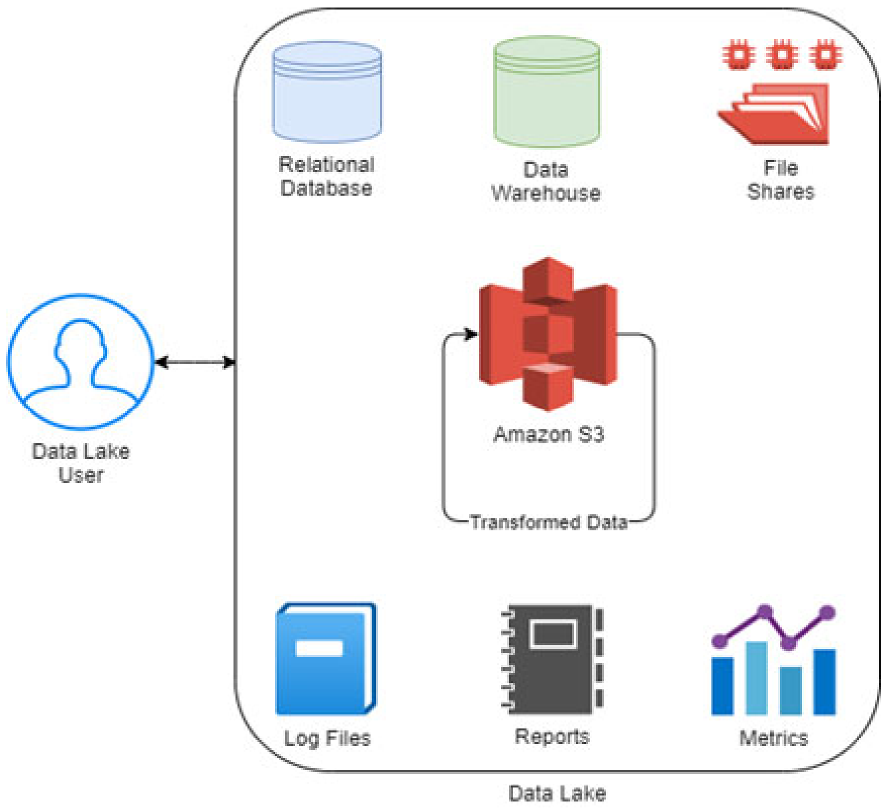


Figure 2.1: *Example diagram of a Data Lake*

The next part of a Data Lake is that it needs to be able to accept data in any format. It should be able to store structured data (csv, database exports, tabular files), unstructured data (images, audio files, text transcripts), or semi-structured data (json, yaml, xml). Finally, the Data Lake must be able to access the data on a per-object or even object version. This allows you to store controlled data in your Data Lake and manage who can view, access, or change individual data objects. For creating a Data Lake on AWS, the service of choice is the Simple Storage Service or S3. S3 is an object store that replicates your data across three Availability Zones to achieve eleven 9's of availability. S3 also offers different tiers with reducing costs that allow you to control the overall resiliency and cost of your stored objects.

On AWS, an Availability Zone is a collection of one or more Data Centers within a Region that are clustered together and connected via low latency dedicated metro fiber. They have redundant resources and are intended to provide high availability to applications within the Region.

Refer to *Figure 2.2*, which shows AWS Global Infrastructure layers:

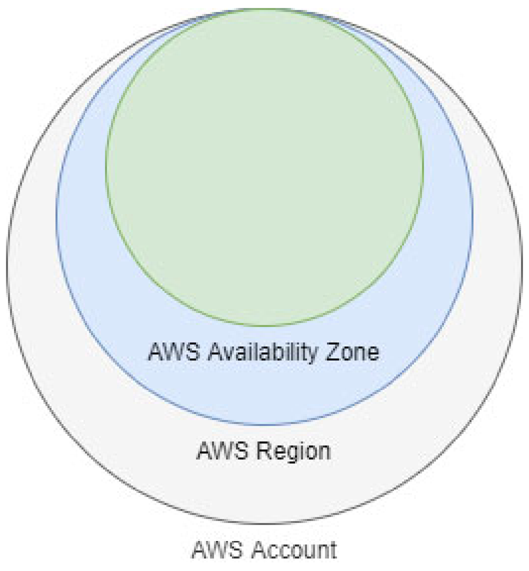


Figure 2.2: *AWS Global Infrastructure Layers*

This level of control and flexibility, along with the ability to control access on a per user, per bucket, per key, or per object level, make it the clear choice for a Data Lake on S3.

NOTE: Ransomware and Data Lakes - A key counter-tactic to project your data against Ransomware attacks is to keep an immutable data store. On S3, this can be accomplished in a few ways, but the most direct way is via a cross-region replication setup with versioning and Object-Locking active. This means that if your data objects are corrupted, you can retrieve the last uncorrupted version from your immutable data store.

Securing your Buckets

Social media and news sites are littered with examples of organizations exposing their data via S3 bucket configurations. Considering your security policy regarding your Data Lake now will save you heartache later. It is important to remember that by default, your buckets and all the objects within them are private. Making them public (readable by anyone in the world) requires action and should be carefully considered before activating or, more appropriately, being prevented entirely. Amazon S3 bucket access can be managed via policies attached to resources, such as buckets, keys, or tags, and policies attached to users. In the case of resource-based policies, also known as bucket policies, you write a policy document, an example of which is given as follows, and attach it to a specific bucket. In the case of user-based policies, and example of which is given as follows, you will create the policy in the **Identity and Access Management** (**IAM**) service and attach them to your users, roles or groups. In both cases, regular review of your policies is crucial to ensuring that you have the appropriately scoped permissions.

Here is an example bucket policy, allowing user **`EngineerCarl`** access to any files that start with the **`/carl/`** key in the S3 bucket data-lake-example-bucket:

```
{
  "Version": "2012-10-17",
  "Id": "BucketPolicyAllowCarlAccess",
  "Statement": [
    {
      "Sid": "ExampleStatement01",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::11223344556677:user/EngineerCarl"
      },
      "Action": [
        "s3:GetObject",
        "s3:GetBucketLocation"
      ],
      "Resource": [
        "arn:aws:s3:::data-lake-example-bucket/carl/*"
      ]
    }
  ]
}
```

Following is an example user policy that allows the user with the attached policy to access all buckets owned by the user, except the one called **private-bucket**:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
     "Effect": "Allow",
     "Action": ["s3:ListAllMyBuckets"],
     "Resource": "arn:aws:s3:::*"
    },
    {
     "Effect": "Allow",
     "Action": [
      "s3:ListBucket",
      "s3:GetBucketLocation"
     ],
     "Resource": "arn:aws:s3:::*"
    },
    {
     "Effect": "Deny",
     "Action": "s3:*",
     "Resource": [
      "arn:aws:s3:::private-bucket/*",
      "arn:aws:s3:::private-bucket"
     ]
    }
  ]
}
```

Securing your Data Lake

Beyond the core S3 bucket(s), the full details of creating, managing, and using a Data Lake are a whole book by itself, but the initial considerations should include the following:

- **Encryption at rest using AWS Key Management Service (KMS)**: Make sure that you identify the compliance or regulatory requirements of your data, especially if it contains **Personally Identifiable Information (PII)**.
- **Object Versioning**: Keeping a history of versions can be key if an accidental overwrite happens. You can also refer to the previous versions of an object when you need to identify the changing dimensions of your data.

- **Object Storage Classes**: Amazon S3 has several storage classes offering a range of availability and cost optimization. Additionally, S3 offers Storage Lifecycle Rules, allowing you to manage the storage class per object. You can move objects to a lower costing tier when they reach a certain age, including older versions of files. S3 also offers Intelligent Tiering, which uses your usage patterns to determine the most cost-effective tier.

Data Lakes for Machine Learning

Now you have a Data Lake, but why is it especially useful for Machine Learning Projects? To answer this question, we can look at the process of machine learning itself. The overall process is typically broken into two phases: a model build phase and a model deploy phase. In the early stages of your Machine Learning project, you will likely manage both phases in a single Jupyter Notebook.

On AWS, you can use SageMaker Notebook Instances, SageMaker Studio, a standalone EC2 instance, or Glue Development Endpoints to host your Jupyter Notebooks. In that notebook, you will usually combine the initial data collection, formatting, feature engineering, model training, model deployment, and model validation to allow you and your team to iteratively experiment with your project quickly. We will cover these items in further detail in the later chapters, but a Data Lake is ideal as a storage solution for this because it allows you to keep your data in its transformed state at each part of the process. This allows the same data to be used for multiple projects by different teams instead of recreating the data from the various sources. For example, suppose you are collecting temperature data for multiple geographic locations around the world. You could start by converting this data to JSON format, and then removing locations not relevant to you and selecting noon each day as your data set. You could save this data back to your Data Lake to be ingested by your model training process, which would allow another team to use the original temperature data for another project or use your transformed data as a starting point.

One important note on Data Lakes is that they are not intended to be replacements for all data repositories. They should be used for data objects and for your materialized data across various data sources, but they act as the center of the various data sources inside and outside your organization. This is typically referred to as Lake House architecture. In this setup, you might take some data from a relational database, join it with a third-party or external data set, enrich it with the data you have in your data warehouse, and finally, drop any data not associated with customers in your customer management system.

The Importance of Hydration

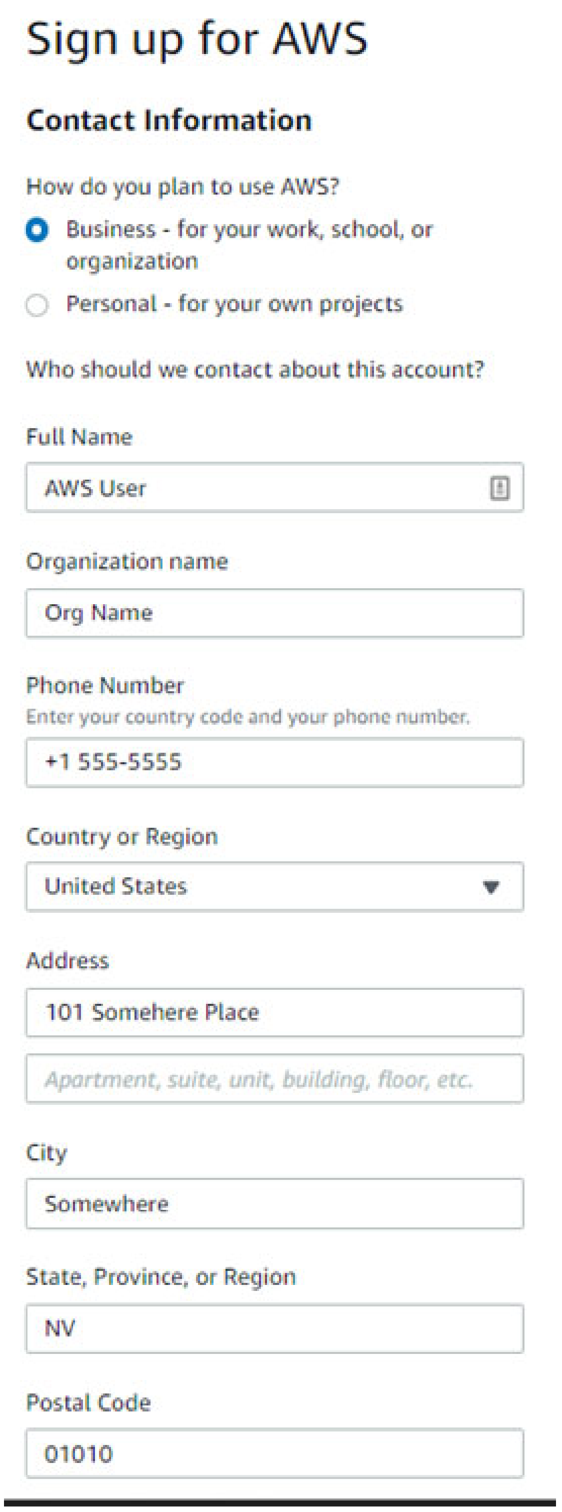
Once you have a Data Lake, the next phase is to populate it with data, often referred to as Hydrating the Data Lake. This is the process of initial movement of collected data that will be needed as separate objects to your Data Lake. For most projects, this will be downloading your selected Data Set and storing it in your Data Lake. It can also involve retrieving your dataset from an existing data source, such as a relational database, data warehouse, or external data store, performing any transformations as detailed later in the chapter, and adding it to your Data Lake. This initial set creates the initial step of treating your data objects as projects and deliverable artifacts on their own. As your data is decoupled from the workings of your applications and Machine Learning projects, it can be processed, versioned, transformed, and managed on its own, without being part of your Machine Learning life cycle.

Setting Up Your AWS Account

To build your Data Lake and hydrate it, you will need an AWS account. To get started, open the AWS home page (**https://aws.amazon.com/**) and choose Create an AWS Account.

NOTE: Root Account Email Address: If you are working in an organization or with a team, consider using a distribution list for the initial email address in case team members are added or removed.

On that page, you will enter your email address, a strong password, and a name for your AWS Account. Verify that the information is correct, and then click `Continue`. Now, you will enter your contact information and select if this is a Business or Personal account, as shown in *Figure 2.3*:



Sign up for AWS

Contact Information

How do you plan to use AWS?

[✓ Business - for your work, school, or organization]

Personal - for your own projects

Who should we contact about this account?

Full Name	AWS User
Organization name	Org Name
Phone Number (Enter your country code and your phone number.)	+1 555-5555
Country or Region	United States
Address	101 Somehere Place
	Apartment, suite, unit, building, floor, etc.
City	Somewhere
State, Province, or Region	NV
Postal Code	01010

Figure 2.3: *Signing up for an AWS Account*

Personal and Business accounts have the same services and settings enabled. For business accounts, make sure that you enter contact and phone numbers for your business rather than a personal number.

Once you click on `Continue`, you will add your contact information, and then select `Continue` and carefully read the AWS Customer Agreement before clicking on `Continue`. Once your account is created, you will get an email confirmation. Once you receive it, you can sign in using the email address and password you entered during account creation.

On first sign in, you will need to enter your payment method before using any of the AWS Services, and then you will need to verify the phone number you added during account creation. Once you agree to the verification, an automated system will call the listed phone number and read off a PIN that you will need to enter on the page.

Lastly, you need to choose your AWS support plan (add details) and click on `Complete Sign Up`. It can take up to 24 hours to activate your account, but it will typically complete in a few minutes.

Once your account is fully activated, you can sign in and use all the services.

This creates what is called a root account. This is a full-fledged account for all purposes, but it is a best practice to create an organizational structure or series of sub-accounts and account groupings called organizational units. This allows you to control not only what happens using an AWS account as a limiting radius but also to have easy visibility into the cost and usage of each account.

Tip: Multi-Factor Authentication: Before you go any further, ensure that you have an extremely secure password and have added multi-factor authentication on your root account. If someone gets access to your root account, they can lock you out and charge any amount they wish to the card you added during setup. Check the AWS subreddit, and you will see a non-stop list of posts with titles like "Surprise bill!" and requests on how to undo the damages. The short version is to use MFA and a very secure password. Set them up now.

The easiest way to set up guard rails for what can happen in your AWS accounts and your initial organizational structure is to use AWS Control Tower. Start by selecting the most appropriate region for your general use and navigating to the Control Tower landing page (**https://console.aws.amazon.com/controltower**). Choose the option to set up your landing zone and the wizard will guide you through the rest of the process. This will include setting up your **Organizational Units** (**OUs**), which are logical groupings of your accounts, as well as AWS Single Sign On access for any users you invite to your accounts.

Tip: Don't be Admin: Yes, it is easier. Don't do it. It is too easy to make mistakes, invoke things in regions you don't intend to, and interfere with infrastructure you don't mean to. Create a user for yourself and start with a managed policy for the job function you want to perform, adding permissions when you need them. After a while, you can use the AWS IAM Access Analyzer to create a custom policy that only includes permission to do the things you need to.

Finally, set up an account in an existing or new organizational unit that you will use for your Machine Learning experiments. It is a best practice to set up a billing alarm (**https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/monitor_estimated_charges_with_cloudwatch.html**) to ensure that you do not have any surprise charges and so that you get a reminder to shut down any unnecessary infrastructure. Thankfully, the newly released AWS console home page includes cost and usage metrics to make it easier to keep an eye on this.

Starting Datasets

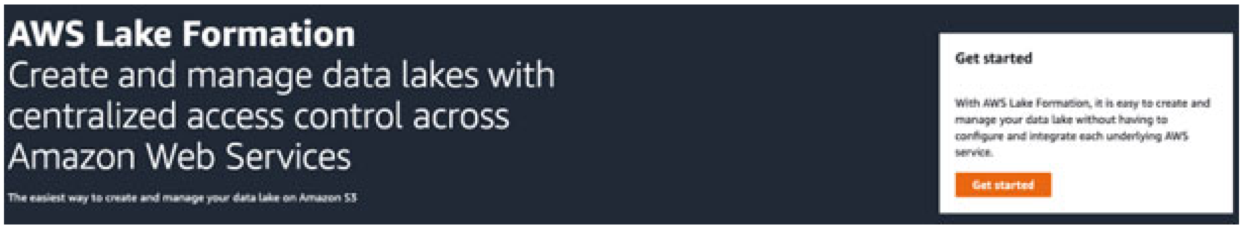
A dataset is simply a collection of data. There are several publicly available datasets, three of which we will be using in this and the following chapters. There are also several dataset search providers, such as Hugging Face Datasets, Kaggle, Google Dataset Search, AWS Public Datasets, and VisualData.

The datasets we will be using in the following chapters are listed here:

- **The Wine Quality Dataset**: **https://archive.ics.uci.edu/ml/datasets/wine+quality**
- **Titanic Dataset**: **https://web.stanford.edu/class/archive/cs/cs109/cs109.1166/problem12.html**
- **MNist Dataset**: **http://yann.lecun.com/exdb/mnist/**

- **PLAsTiCC Astronomical Classification Dataset**: **https://www.kaggle.com/c/PLAsTiCC-2018/data**

You can use these links to download these datasets and store them in your Data Lake, along with other sets that you choose to use with other projects. Refer to *Figure 2.4*:



AWS Lake Formation

Create and manage data lakes with centralized access control across Amazon Web Services

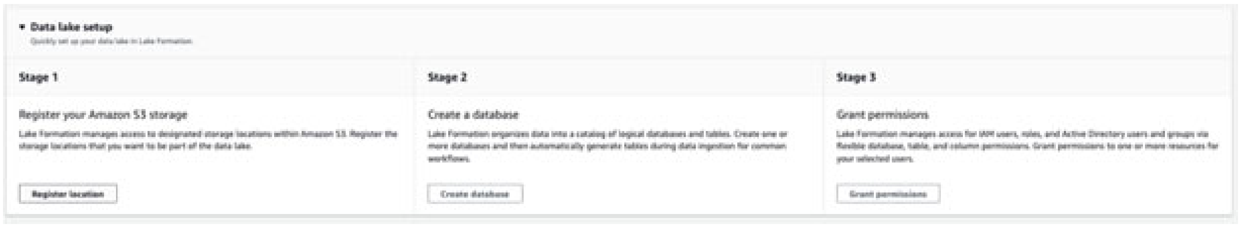
The easiest way to create and manage your data lake on Amazon S3

Get started

With AWS Lake Formation, it is easy to create and manage your data lake without having to configure and integrate each underlying AWS service.

Figure 2.4: *Welcome screen for Lake Formation*

There are several AWS Services that are considered in scope for Data Lake Hydration, and, like most things, there are many ways to accomplish the same goal in AWS. Returning to the scenario given at the beginning of the chapter, we can consider the task from the perspective of the database engineer who has chosen to use the AWS Lake Formation service to create the Data Lake. Refer to *Figure 2.5*:



Data lake setup

Quickly set up your data lake in Lake Formation.

Stage 1

Register your Amazon S3 storage

Lake Formation manages access to designated storage locations within Amazon S3. Register the storage locations that you want to be part of the data lake.

Stage 2

Create a database

Lake Formation organizes data into a catalog of logical databases and tables. Create one or more databases and then automatically generate tables during data ingestion for common workflows.

Stage 3

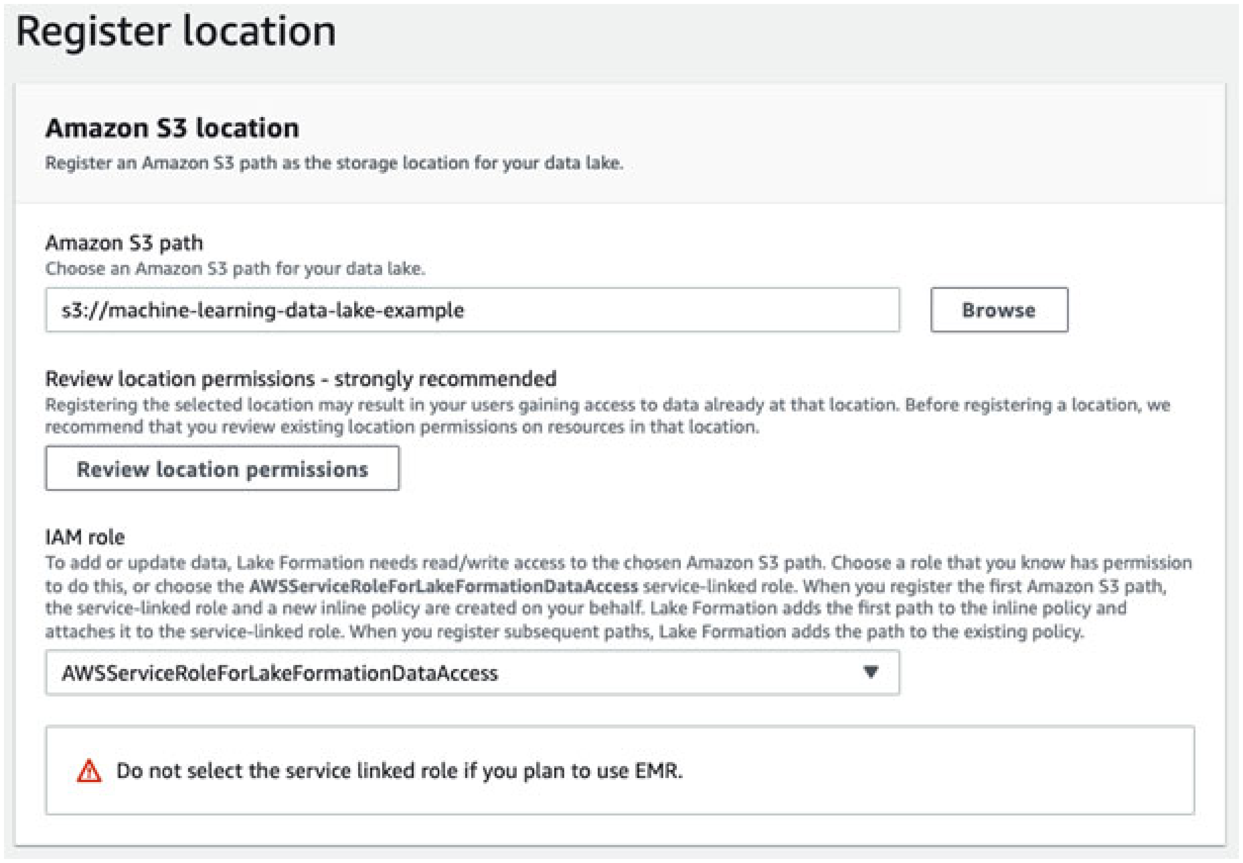
Grant permissions

Lake Formation manages access for IAM users, roles, and Active Directory users and groups via flexible database, table, and column permissions. Grant permissions to one or more resources for your selected users.

Figure 2.5: *Setting up your Data Lake with Lake Formation*

Our first task when getting ready to implement AWS Lake Formation is to create a Role role for the service to use on our behalf. When you sign in to AWS, you assume a Role, a type of persona that identifies the groups of permission sets we have in our accounts, also known as policies. AWS services are no different. Each service needs a role that it can assume (also known as a trust relationship) that also defines what it is allowed to do in your account. Just like users, it is important that services only have the permissions to do what they absolutely need to do with specific resources.

If you have existing S3 buckets, you can also register them as sources in your Data Lake, also known as Data Lake locations. This is particularly useful in cases where there are existing buckets that were used for data sources in the past that can now be consolidated in your Data Lake. Refer to *Figure 2.6*:



Register location

Amazon S3 location

Register an Amazon S3 path as the storage location for your data lake.

Amazon S3 path

Choose an Amazon S3 path for your data lake.

s3://machine-learning-data-lake-example

Review location permissions - strongly recommended

Registering the selected location may result in your users gaining access to data already at that location. Before registering a location, we recommend that you review existing location permissions on resources in that location.

IAM role

To add or update data, Lake Formation needs read/write access to the chosen Amazon S3 path. Choose a role that you know has permission to do this, or choose the **AWSServiceRoleForLakeFormationDataAccess** service-linked role. When you register the first Amazon S3 path, the service-linked role and a new inline policy are created on your behalf. Lake Formation adds the first path to the inline policy and attaches it to the service-linked role. When you register subsequent paths, Lake Formation adds the path to the existing policy.

AWSServiceRoleForLakeFormationDataAccess

Do not select the service linked role if you plan to use EMR.

Figure 2.6: *Registering a bucket with Lake Formation*

Our database engineer closely follows the AWS documentation for setting up Lake Formation (**https://docs.aws.amazon.com/lake-formation/latest/dg/getting-started-setup.html**), creating a role for the service, a Data Lake administrator (themselves), initial users (their team members), the actual S3 bucket used for storing objects, and the optional governed tables configuration. They then download the datasets linked above, loading them into the Data Lake for the upcoming machine learning projects.

Lake formation gives you several powerful optionsas shown in the following image:

- **Data Catalog**: This is the same as Glue Databases and Tables, described later in this chapter.

- **Register and ingest**: This is the ability to register S3 buckets as sources for your Data Lake, transform data using predefined templates and links to AWS Glue Crawlers and Jobs, described later in this chapter.
- **Permissions**: Centralized access control for your Data Lake. Controls can be applied for users and applications by role, which can be applied at the table and row level. Refer to *Figure 2.7*:

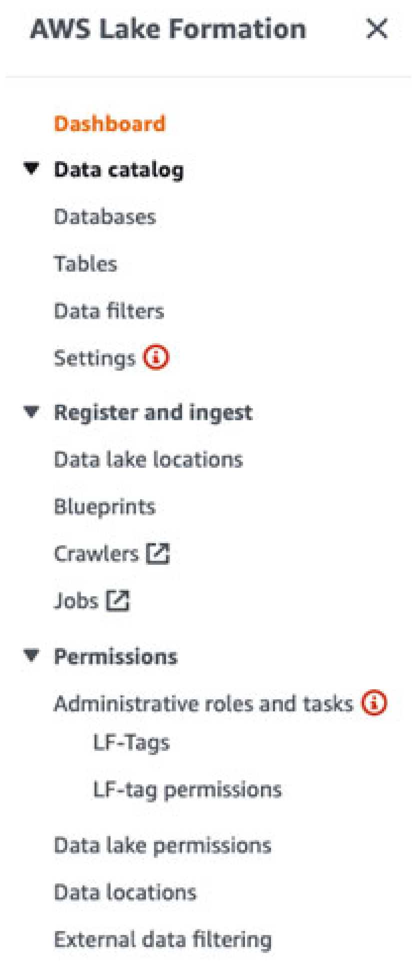


Figure 2.7: *Functions of Lake Formation*

Happy with their work, our database engineer checks their email to see that a brand new ticket has been assigned to them. IT Leadership, ecstatic with the quick implementation of the Data Lake, have identified the following additional data sources to add to the expanding lake:

- Clickstream data stored in DynamoDB in another AWS Account

- Current and historical log data for various cloud-native and on-premises applications
- Historical flat files in a network filesystem storage array in an aging colocation facility
- Cold-storage archival files in branch offices with extremely limited connectivity

In addition, several teams have requested the ability to view the characteristics of the data files loaded in the Data Lake and extract data from them without downloading them locally. Luckily, our database engineer, now an emerging Cloud Architect, has this book on hand and is ready to take on these new requests.

Streaming Data and the Data Lake

Clickstream data, or any ongoing data stream for that matter, can be ingested into the Data Lake via Amazon Kinesis. It is possible, of course, to create a one-time export of data from the existing data repository, but then an almost immediate request would come in for a refresh of the data, creating an endless loop of task repetition. In addition, some data sources receive rapid, ongoing updates, also known as streaming. These are particularly suited for a transport mechanism that can scale to support them.

Amazon Kinesis

Enter the AWS service family: Amazon Kinesis. A collection of similar services, Amazon Kinesis is a data transport, transform, and analysis mechanism. Using the services included, you can connect to data sources and scale as needed to deliver that data in its raw or transformed state to your target location. You can even chain the respective Amazon Kinesis services together to create more complex data streaming applications:

- Kinesis Data Streams allows you to serverless stream data sources like logs, processes, events, or updates to your target location. That location can be another data repository, such as S3, or another AWS service. The main benefit is that Kinesis scales to meet your needs, allowing you to decouple the respective portions of your workload. Kinesis Data

Streams allows for two modes of operation: on-demand and provisioned.

  - On-demand is exactly what it sounds like, no capacity planning and can scale up to gigabytes of read/write operations per minute. Kinesis handles the capacity management for you; use on-demand for spiky, unpredictable data streams.
  - Provisioned is where you specify the number of shards, which are uniquely identified data records in a single Kinesis stream. Shards have a defined capacity: max read of 2MB per second and max write of 1MB per second; use provisioned for predictable, steady-state data streams.

- Kinesis Data Firehose is a high-speed data delivery system custom made for moving data from its source to analytics services, data warehouses, or even Data Lakes.
- Kinesis Data Analytics allows you to create insights from your data mid-stream using the Apache Flink.
- Kinesis Video Streams is custom-made for managing, converting, and storing real-time video streams.

In this case, Amazon Kinesis Data Streams and Kinesis Firehose is the choice to migrate clickstream data from Amazon DynamoDB to our Data Lake. Kinesis Data Streams will act as the consumer, collecting and managing the input of item-level changes in DynamoDB, delivering them to Kinesis Firehose for high-speed delivery into the Data Lake. DynamoDB also features a convenient integration with Kinesis Data Streams, meaning we do not need to write any custom integration between the two services. In DynamoDB, simply click on the **`Overview`** of your chosen table, and then click on **`Manage streaming to Kinesis`** to get started. If needed, you can also integrate Kinesis Data Firehose with a compute resource like the serverless option, Amazon Lambda, to perform any data transformations that are needed, such as compressing, updating, or enriching the data before being written to its location in the Data Lake. An example Python3 script for creating a Kinesis Data Stream has been included in the GitHub repo associated with this book.

AWS DataSync

Now that the clickstream data is flowing into the Data Lake, we can focus on the log data stored in on–premises storage locations. We could again leverage Amazon Kinesis to migrate the data, but in this case, the data is historical and accompanied by application artifacts that provide context to the data. For this, we can leverage **AWS DataSync** to migrate the data into our Data Lake. DataSync is meant to be deployed as a virtual machine to an on-premises compute service that has an NFS, or SMB mount attached. Using the DataSync agent running on the virtual machine, files and folders can be identified for transfer to your AWS account, targeting S3, Amazon Elastic File System, or AWS FSx for Windows or Lustre.

Once we have the agent configured with the data we want to move and the target location, that is, our Data Lake, the DataSync agent connects to the DataSync service and writes the data to the selected location. AWS DataSync also supports scheduled tasks, so a daily transfer of any new log files can be created.

AWS Database Migration Service

In situations where our source data is in on-premises or in cloud databases like Oracle, MS SQL, MySQL, MariaDB, PostgreSQL, SAP, or DB/2, the AWS Database Migration Service can be leveraged for one-time or ongoing data migration (also referred to as change data capture, or CDC). In order to use the service, we first define a source object, giving the connection details and information about our source database. We then create a target object, giving connection details and information about our target location, which in this case is S3, the Data Lake. We also create a replication instance. This is the compute resource that will be performing the actual migration, conversion, mapping, and writing to the target location. You can use the same replication instance for multiple tasks, but make sure it has sufficient networking, CPU, and memory resources to keep up with the data being migrated.

Finally, you will create the replication task, tying the source, target, and replication instance together, specifying the type of replication: one-time full migration, change data capture only, or full migration with ongoing changes. AWS also recently released the Database Migration Studio, a visual interface for managing migration tasks, simplifying the overall process. At the time of this book being written, the Database Migration Studio is in preview, so make sure it has the features you need before using, especially in your production workloads. AWS DMS Fleet Advisor is a component that allows you to collect and manage data about multiple database environments, whether on-premises or in the cloud, and prepare it for migration tasks.

AWS Schema Conversion Tool

Another service that is in scope for this task is the AWS Schema Conversion Tool. Rather than a service that runs in your AWS Account, it is a tool you can download and use to assess existing database structure as well as map those database sctructures to new targets, identifying which portions can be converted automatically and which ones need manual intervention. This tool is particularly useful when you need to migrate from one data source with a strict schema to another with a similarly strict schema. Examples include from Oracle to Amazon Redshift, Apache Cassandra to Amazon DynamoDB, and SAP ASE to Aurora PostgreSQL. For our example, PostgreSQL to S3, our data will be written in CSV, which meets our business outcome needs, so we do not need to use the AWS SCT.

Once done with the database migration, our database engineer can move on to migrating the log and flat files from the various remote facilities. AWS DataSync is a possibility, but with limited internet bandwidth and the lack of local virtualization, it would be a slow process at best. The files aren't needed immediately, but they will be needed before the end of the current fiscal year.

Looks like it is time for a Snowstorm.

AWS Snow Family

AWS Snowcone, Snowball, and Snowmobile are series of physical devices that can be transported to your data where it is in the world, loaded with that data, and then transported back to AWS, where it will be loaded into the supported target location of your choice. Each of the devices are highly durable and are designed to withstand significant shipping incidents. They also come with multiple layers of encryption, preventing unauthorized access to your data in transit.

AWS Snowcone

The Snowcone is the smallest of the Snow family and is designed to connect to your network and either transfer data locally or via the internet using the embedded AWS DataSync client. Once your data has been loaded onto the Snowcone device, you ship it back to AWS. The device can support up to 14 terabytes of solid-state storage and has local compute capabilities that allow data processing before shipping it back to AWS.

AWS Snowball

Snowball is the mid-range of the Snow family, available in storage- or compute-optimized configurations. The Snowball Edge Storage configuration features up to 80 terabytes of included storage, and like the Snowcone, has included compute capacity to process data before shipping it back to AWS.

AWS Snowmobile

Snowmobile is a truck. Yes, a whole shipping container moved via a freight truck. It is designed for extremely large data transport tasks and up to exabyte-scale data sets. For our listed use case, the Snowcone has enough capacity, and since they have sufficient time to complete the task, our database engineer orders two from inside their AWS account console. They will coordinate getting them shipped to the first two remote colocation facilities, getting them hooked to the local network, loaded with the necessary data, and shipped back to AWS before ordering two more. One the tasks are finished and the queue is empty, our database engineer ends their workweek with a feeling of accomplishment and enjoys their weekend, getting some well-deserved rest. Come Monday morning, however, there is a brand-new ticket assigned to them: **Analytics**.

We will cover the majority of analytics in *Chapter 3, Predicting the Future With Features*, and *Chapter 4, Orchestrating the Data Continuum*. Designing a full-fledged analytics solution is far more than a single ticket. After a careful review, the task, at first blush, seems much simpler. The users of the Data Lake are delighted with the ease and simplicity of interacting with the individual data objects and the ability to store data in its native format. The only challenge reported so far is that it is difficult to understand the structure of the folders in the S3 bucket, where the data users they need might be, and what the starting format of the data is.

Users have suggested the creation of a data dictionary database that would identify each file, its location, and all the data about the data (also known as metadata). Our database engineer is tempted but thinks there might be a better way without creating a completely new data repository to manage.

NOTE: S3 Folders: A very important note about folders in S3 is that there aren't any. The user interface for S3 will show "folders" to allow the concept or organization, but the underlying object store only contains two pieces of location-based information about the objects: the bucket and the object's key. That key includes the entirety of the object's path. For example, if you have a bucket called "uniquetestbucket" with a folder called "testfolder" and a file called "testfile.jpg", the object is referenced as the bucket "uniquetest bucket" and the object key as "testfolder/testfile.jpg".

NOTE: S3 Key Optimization: Along with the previous note about S3 Folders is the concept of partitioned prefixes. An object's prefix is just the first part of the entirety of its key. When objects are read or scanned, they are scanned in blocks according to the similarity of the prefixes in the objects you are scanning. This means that if you have a set of files that all start with "nested/folder/structure/filename", followed by an incrementing number and the file extension, you will likely be scanning all of those similarly named files per operation. The best way to prevent this is to move the file or folder unique names as early as possible, akin to "nested/dateTime/folder/itemNumber/filename" where dateTime is the actual dateTime stamp and the itemNumber is an incremented number. This allows you to scan fewer items in your read operations, speeds up scans, and allows faster I/O operations in your Data Lake.

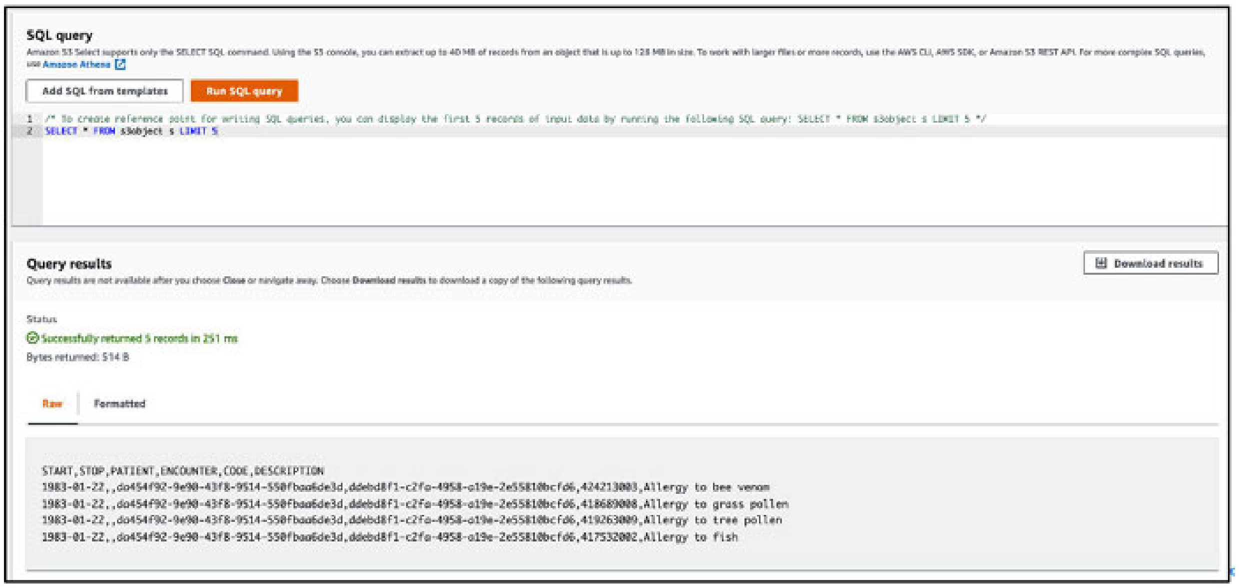
Uncovering Patterns

We will be getting into full-fledged analytics, feature engineering, and transforming data and preparing it for Machine Learning projects in *Chapter 3, Predicting the Future With Features*, and *Chapter 4, Orchestrating the Data Continuum*, but we want to provide some context to the data we have now. Both the Data Lake solution and the Lake Formation service allow us to add data about our data, also called metadata, to the individual data objects, which accelerate the data preparation phase of our Machine Learning projects. We can add additional context pieces, like the name of the team that owns the data, a minimum refresh date by which the data needs to be re-retrieved from the source system, the data security level of the data, or even a Boolean value if the data contains **Personally Identifiable Information** (also called **PII**). These are typically added as tags.

Tip: Tagging is not just for artists. Nearly everything that can be created in AWS can have tags created with it or added later. Tags are simple key-value pairs that add context or accompanying information to the created item. Beyond the context, it is a vital pattern to decide which tags are basic necessities for the things created in your account and which ones are optional. This can help you keep track of what resources are created for which purpose, what team is working on a particular service, or what part of a workload a service is used for. You can also use the AWS Tag Manager and Service Control Policies to enforce your decided tagging standards.

Amazon S3 Select

The AWS Simple Storage Service allows users to retrieve subsets of data from individual data objects using standard query language, also called SQL This means that using the console, the provided AWS software development kits, also known as SDK, or the S3 API, can retrieve selections of your files using standardized languages, as shown in the following examples. Normally, you would have to download the file and load it into query service, but S3 Select bypasses that and allows you to look at the structure of your files and programmatically retrieve subsections of them, ready to be used in many analytics, data engineering, and machine learning projects. Refer to *Figure 2.8*:


SQL query

Amazon S3 Select supports only the SELECT SQL command. Using the S3 console, you can extract up to 40 MB of records from an object that is up to 128 MB in size. To work with larger files or more records, use the AWS CLI, AWS SDK, or Amazon S3 REST API. For more complex SQL queries, use Amazon Athena

```
/* To create reference point for writing SQL queries, you can display the first 5 records of input data by running the following SQL query: SELECT * FROM s3object s LIMIT 5 */
SELECT * FROM s3object s LIMIT 5
```

Query results

Query results are not available after you choose **Close** or navigate away. Choose **Download results** to download a copy of the following query results.

Status

Successfully returned 5 records in 251 ms

Bytes returned: 514 B

```
START,STOP,PATIENT,ENCOUNTER,CODE,DESCRIPTION
1983-01-22,,da454f92-9e90-43f8-9514-550fbaa6de3d,ddebd8f1-c2fa-4958-a19e-2e55810bcfd6,424213003,Allergy to bee venom
1983-01-22,,da454f92-9e90-43f8-9514-550fbaa6de3d,ddebd8f1-c2fa-4958-a19e-2e55810bcfd6,418689008,Allergy to grass pollen
1983-01-22,,da454f92-9e90-43f8-9514-550fbaa6de3d,ddebd8f1-c2fa-4958-a19e-2e55810bcfd6,419263009,Allergy to tree pollen
1983-01-22,,da454f92-9e90-43f8-9514-550fbaa6de3d,ddebd8f1-c2fa-4958-a19e-2e55810bcfd6,417532002,Allergy to fish
```

Figure 2.8: *Querying a file in your Data Lake with S3 Select*

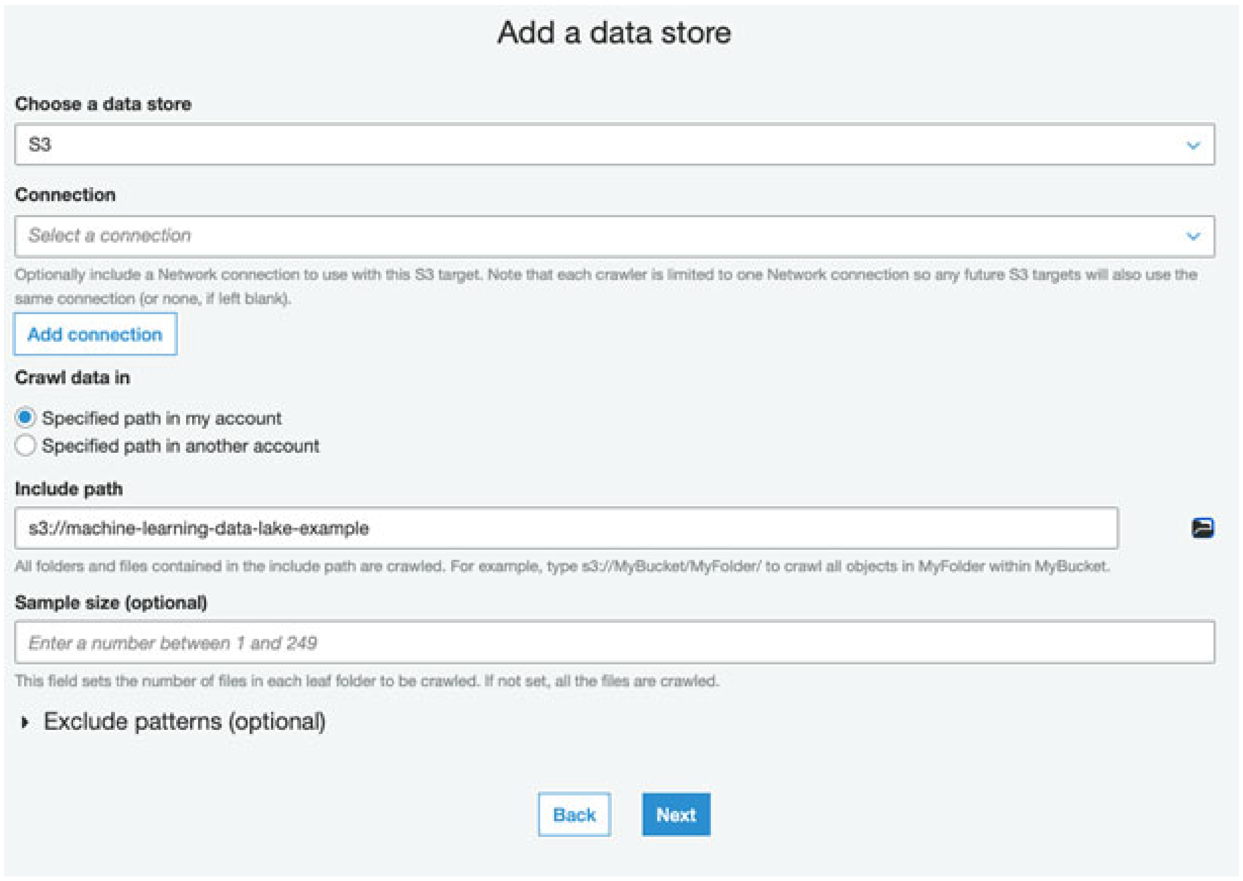
AWS Glue

Like Amazon Kinesis, AWS Glue is the name of a suite of tools and services designed to prepare, transform, and identify data. It is serverless, which means there are no compute or storage resources to manage, so your teams can run both their experiments, directly making changes to the data they need to run their projects and create automated jobs that repeat those steps for Machine Learning workflows.

We will only a few tools in the Glue family in this chapter, with more covered in *Chapter 4, Orchestrating the Data Continuum*.

Glue Crawlers

Metadata is data about your data, and the ability to easily manage a repository of metadata is critical to enabling teams consuming the data stored in your Data Lake. What fields are in a specific file, their data types, the number of entries, the file format, and the overall schema can be determined automatically and stored for review. The following image shows the AWS Glue Console when adding a new Crawler. The crawler will ask for database and table information to store the resulting metadata in. If you do not already have a database and table prepared, you can create one when you create your Crawler. Refer to *Figure 2.9*:



Add a data store

Choose a data store

S3

Connection

Select a connection

Optionally include a Network connection to use with this S3 target. Note that each crawler is limited to one Network connection so any future S3 targets will also use the same connection (or none, if left blank).

Add connection

Crawl data in

[✓ Specified path in my account]
Specified path in another account

Include path

s3://machine-learning-data-lake-example

All folders and files contained in the include path are crawled. For example, type s3://MyBucket/MyFolder/ to crawl all objects in MyFolder within MyBucket.

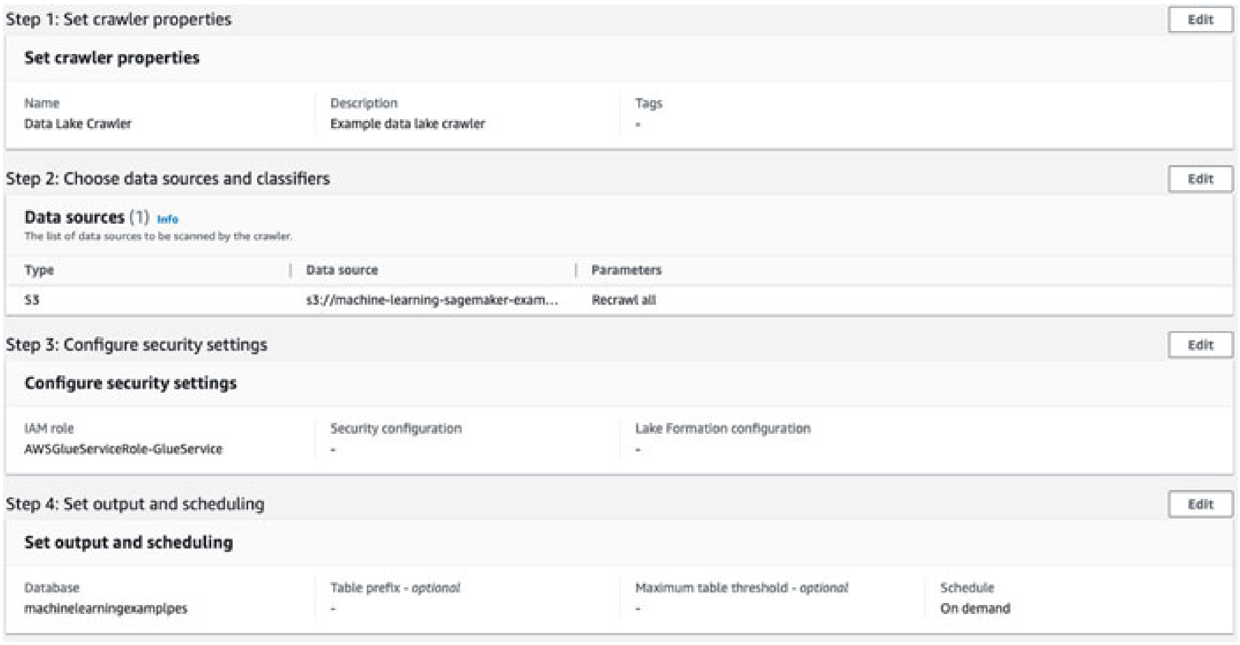
Sample size (optional)

Enter a number between 1 and 249

This field sets the number of files in each leaf folder to be crawled. If not set, all the files are crawled.

Figure 2.9: *Building a Glue Crawler*

You use AWS Glue Crawlers to scan the portions of your Data Lake you specify. The scanning process populates Tables in the Glue database with the collected metadata, as shown in *Figure 2.10*:



Step 1: Set crawler properties

Set crawler properties

Name	Description	Tags
Data Lake Crawler	Example data lake crawler	-

Step 2: Choose data sources and classifiers

Data sources (1)

The list of data sources to be scanned by the crawler.

Type	Data source	Parameters
S3	s3://machine-learning-sagemaker-exam...	Recrawl all

Step 3: Configure security settings

Configure security settings

IAM role	Security configuration	Lake Formation configuration
AWSGlueServiceRole-GlueService	-	-

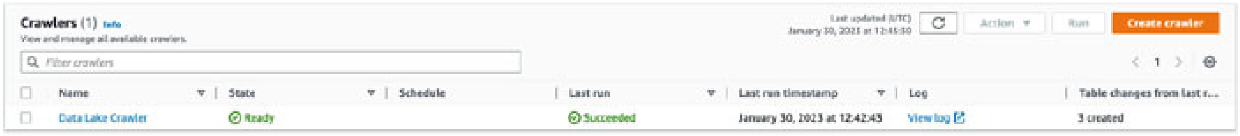
Step 4: Set output and scheduling

Set output and scheduling

Database	Table prefix - *optional*	Maximum table threshold - *optional*	Schedule
machinelearningexamplpes	-	-	On demand

Figure 2.10: *Review screen when creating a Glue Crawler*

Extract, Transform, and Load (ETL) jobs use the Glue Data Catalog as sources and targets, reading from the data stores defined in the tables. To create a Glue Crawler from the AWS Glue service home page, choose Crawlers. From there, you can follow the wizard or manage the individual elements of the crawler. Once the crawler starts, it will recursively scan the target location, reading in the individual data objects and collecting the metadata about them, writing it to the target Glue Table, as shown here (refer to *Figure 2.11*):



Crawlers (1)

View and manage all available crawlers.

Last updated (UTC) January 30, 2023 at 12:45:30

Name	State	Schedule	Last run	Last run timestamp	Log	Table changes from last r...
Data Lake Crawler	Ready		Succeeded	January 30, 2023 at 12:42:45	View log	3 created

Figure 2.11: *A created Glue Crawler*

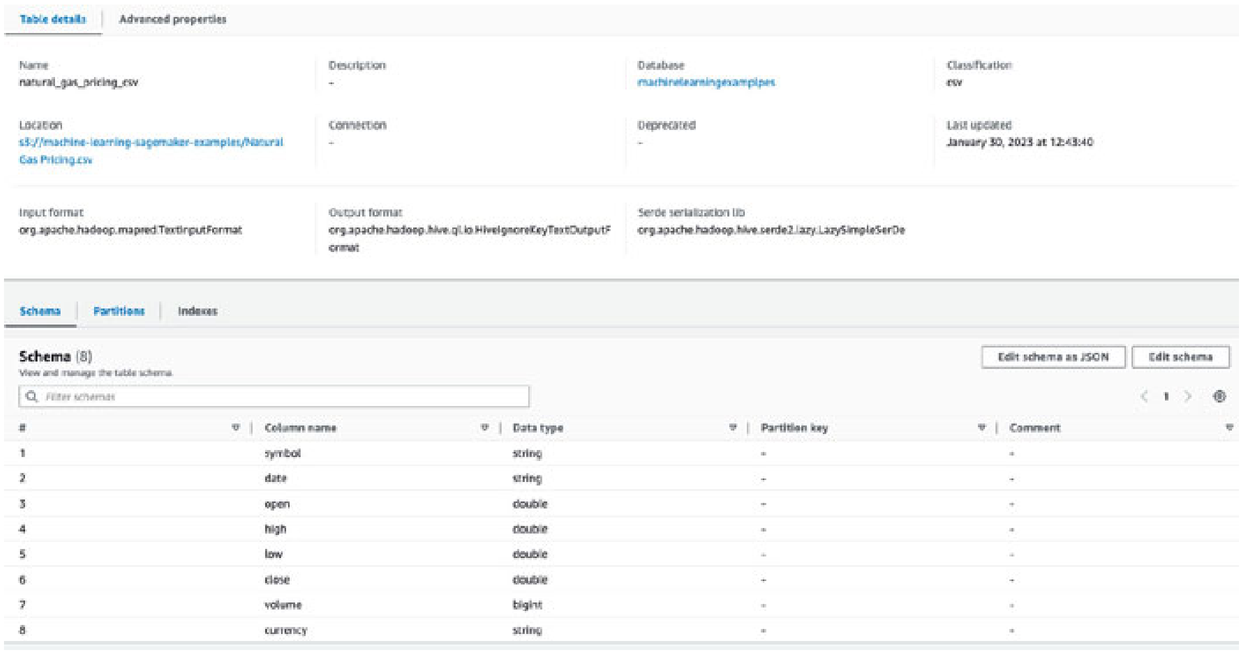
You can set crawlers to run on demand or on a schedule, enabling you to refresh your metadata table as new data lands in your Data Lake.

Glue Databases

AWS Glue Databases are collections of metadata tables in your AWS Glue service. Your AWS Glue Databases can contain tables that aggregate data from multiple sources in a single data repository or in multiple data repositories. This means you can collect metadata from multiple locations in your Data Lake that are logically associated even if they are stored in different key structures.

Glue Tables

Glue Tables are metadata from one or more sources. You can create tables manually, as a result of a Glue Crawler, using the CloudFormation infrastructure as Code service, or via Glue API, as shown in *Figure 2.12*:



Name natural_gas_pricing_csv	Description -	Database machinelearningexamples	Classification csv
Location s3://machine-learning-sagemaker-examples/Natural Gas Pricing.csv	Connection -	Deprecated -	Last updated January 30, 2023 at 12:43:40
Input format org.apache.hadoop.mapred.TextInputFormat	Output format org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat	Serde serialization lib org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe	

Schema (8)
View and manage the table schema.

#	Column name	Data type	Partition key	Comment
1	symbol	string	-	-
2	date	string	-	-
3	open	double	-	-
4	high	double	-	-
5	low	double	-	-
6	close	double	-	-
7	volume	bigint	-	-
8	currency	string	-	-

Figure 2.12: *A Glue table created from our crawler*

When you create the table manually, you can also define the table schema and the values of each table field. When it is created via a Glue Crawler, those parameters are created for you.

Amazon Athena

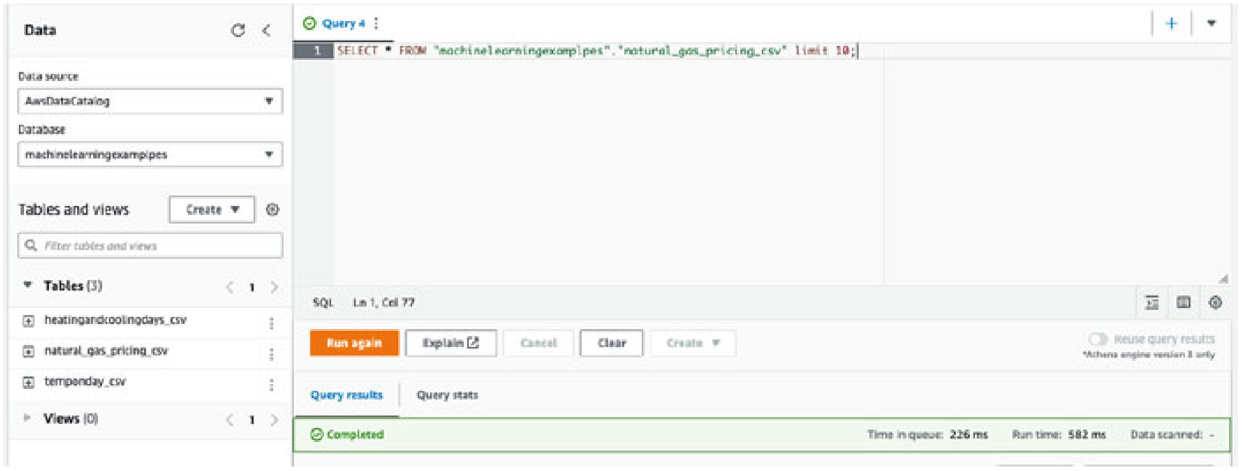
Amazon Athena is a serverless query service that simplifies interacting with data stored in Amazon S3, such as the data in our Data Lake. You use ANSI standard SQL to query your data without the need to move it from where it persists. Even better, Athena reads metadata from the tables in your Glue Data Catalog, which allows you to interact with that data without moving it from your Data Lake. With a few minor exceptions, the SQL standard for

Athena is based on the HIVEQL DDL: **https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL**.

Like most services, you can interact with Athena from the AWS console, choosing the database and table to query and then writing your queries directly into the provided web-based editor. Your results will be provided directly, allowing you to view, combine, and investigate your data. You can also use the Athena API and the SDK to build Athena interactions into your applications.

When you first use Athena, you will need to set up an S3 location that Athena can use to store your query results. This should not be in your Data Lake, since this data is considered ephemeral, but you can export your queries and their results to your Data Lake if it would be of use.

Figure 2.13 is an example of a simple query that can be run in the Amazon Athena console, selecting 10 rows from the chosen data source. This can be used to get an initial idea of the format of your columns, their names, and the data types:



Data source: AwsDataCatalog

Database: machinelearningexampipes

heatingandcoolingdays_csv
natural_gas_pricing_csv
temperday_csv

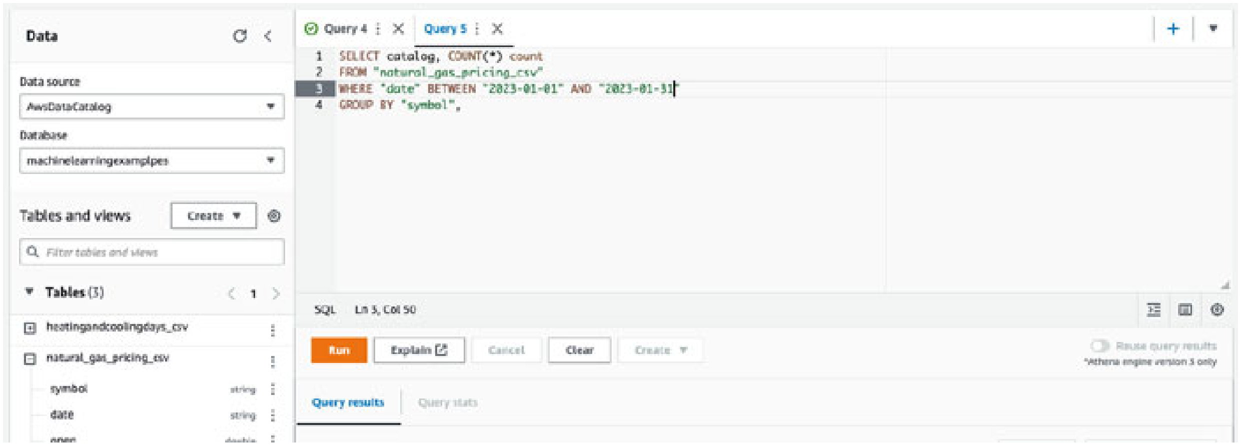
```
SELECT * FROM "machinelearningexamplpes"."natural_gas_pricing_csv" limit 10;
```

SQL Ln 1, Col 77

Completed Time in queue: 226 ms Run time: 582 ms Data scanned: -

Figure 2.13: *A basic query against a Glue Table in Athena*

While our primary use case for Athena is to provide visibility into our newly created Data Lake, it can also connect to relational databases using **Open Database Connectivity** (**ODBC**) and **Java Database Connectivity** (**JDBC**), external Hive metastores, and other Athena data source connectors, as defined in the provided documentation. *Figure 2.14* shows some of the types of ANSI standard SQL that can be used in the Amazon Athena console. The query is selecting a specific column, counting the entries in another where values in a third column are between specified valued, and finally grouped together by the first column:



```
SELECT catalog, COUNT(*) count
FROM "natural_gas_pricing_csv"
WHERE "date" BETWEEN "2023-01-01" AND "2023-01-31"
GROUP BY "symbol",
```

Figure 2.14: *More complex queries are possible with Athena*

While there is more to Athena that can be covered in this book, there is one additional key feature that can be implemented to empower the teams working on Data and Machine Learning projects: Workgroups.

Here are a few sample Athena queries:

1. Selecting the number of rows in the stellar_coordinate_data_table:

   ```
   SELECT count(*) as count FROM stellar_coordinate_data_table
   ```

2. Selecting data referencing a Glue Table, grouping our stellar data set and performing simple math expressions to derive additional details:

   ```
   SELECT DATE, CATEGORY, CONSTELLATION
    avg(stellar_coordinate) as coordinate,
    avg(stellar_coordinate/stellar_size) as
    average_stellar_momentum,
    avg(stars_in_constellation) as average_stars,
    stellar_size (stars_in_constellation, .99) 99th_percentile
   FROM stellar_coordinate_data_table
   WHERE DATE > '2002%' AND (TYPE = 'O' OR TYPE = 'A'
   GROUP BY CONSTELLATION
   ORDER BY 99th_percentile
   ```

Athena Workgroups allows you to separate users, teams, or even applications in Athena, controlling not only what data they have access to but also allowing them to share their queries and the results of their queries.

Since Athena is charged based on the amount of data scanned, the ability to set scanning limits in Athena is key to helping with cost control.

Conclusion

In this chapter, we covered the first major part of the ML workflow (which we learned about in *Chapter 1, Introducing the ML Workflow*): the data preparation phase. The most important part of a ML project is data, and first we need to determine what our data sources are, understand how data is generated, and how it needs to be curated. Then, we need to devise a mechanism to source this data into a centralized repository for collection, processing and consumption. We learned that we can do all this with an Amazon S3 Data Lake and using the several data ingestion mechanisms that AWS supports natively. We saw, with the help of code examples, how to set up a Data Lake, how to ingest data into the Data Lake (a process called hydration), how to query and view this data, and finally, how to do some processing before you can feature engineer your datasets for ML.

What's next you ask? Let's step into a time machine and take a look at the future in *Chapter 3, Predicting the Future With Features!*

Points to Remember

Here's a summary of what we learned in this chapter:

- In this chapter, we introduced the concept of a Data Lake, what purpose it serves, how to set it up, and how to enable your users to interact with the data objects in your Data Lake effectively.
- A Data Lake is a centralized data repository that allows you to store your data in its original format: structured, semi-structured, or unstructured. The Data Lake stores data from all aspects of your organization and enables users to set a schema to that data when it is read versus when it is written, meaning that the data is applicable to all users in all teams. This also allows you to form your data to the project at hand rather than trying to guess the schema needed.
- A Data Lake is also free from compute, memory, or legacy disk limitations. Without the need for a compute resource pool and memory allocation, your Data Lake can grow as much as it needs to while

providing the rules necessary to archive data when it is no longer needed.

- Data Lakes also provide functional and relevant security controls by user, role, project or team. Access, visibility, and permissions can be controlled down to individual rows of data.
- Lastly, Data Lakes integrate natively with tools meant to add additional contextual information about the data stored and those that allow visibility into the data.
- We also introduced a scenario detailing an experienced database engineer tasked with the creation of a Data Lake and enabling their team with its use. At the end of the current sprint, the database engineer was able to close all the assigned stories, having provided the requested functionality despite Mercury being in retrograde.

Multiple choice questions

Use these questions to challenge your knowledge of what you learned in this chapter.

1. **What is the difference between a Data Lake and a Data Warehouse?**
    a. These are just two names for the same concept.
    b. Data Lakes store data in their original format, and Data Warehouses store processed and transformed data for direct consumption.
    c. Data Lakes store processed and transformed data, and Data Warehouses store data in their original format.
    d. Data Lakes store data for reporting purposes, and Data Warehouses store data to help with operational processing.
2. **It is impossible to collect and ingest real-time streaming data directly into a Data Lake.**
    a. True
    b. False

3. **Amazon Athena is the service of choice to ingest batch data into an S3 Data Lake.**
    a. True
    b. False
4. **Which of these AWS services CANNOT be used for data ingestion or processing?**
    a. Amazon Polly
    b. Amazon Athena
    c. AWS Lambda
    d. Amazon Kinesis
5. **What does ETL stand for?**
    a. Elegant, Transformative, Long-lasting
    b. Engines, Transformers, Losses
    c. Extract, Transform, Load
    d. Empirical, TensorFlow, Latent

Answers

1. **b**
2. **b**
3. **a**
4. **a**
5. **c**

Further Reading

Here are a few additional resources for learning how to create and manage a Data Lake to enable your Machine Learning projects on AWS:

- The pre-built AWS Data Lake Solution: **https://aws.amazon.com/solutions/implementations/data-lake-solution/**

- Getting started with AWS Lake Formation: **https://aws.amazon.com/blogs/big-data/getting-started-with-aws-lake-formation/**
- Access and manage data from multiple accounts from a central AWS Lake Formation account: **https://aws.amazon.com/blogs/big-data/access-and-manage-data-from-multiple-accounts-from-a-central-aws-lake-formation-account/**
- A public Data Lake for analysis of COVID-19 data: **https://aws.amazon.com/blogs/big-data/a-public-data-lake-for-analysis-of-covid-19-data/**
- Anonymize and manage data in your Data Lake with Amazon Athena and AWS Lake: **https://aws.amazon.com/blogs/big-data/anonymize-and-manage-data-in-your-data-lake-with-amazon-athena-and-aws-lake-formation/**
- AWS Analytics Services Explained: From Data Lakes to Machine Learning: **https://aws.amazon.com/blogs/apn/aws-analytics-services-explained-from-data-lakes-to-machine-learning/**

CHAPTER 3
Predicting the Future With Features

Introduction

Do you believe in your stars? It is high time that all of us do. According to this article (**https://www.livescience.com/12856-astrology-science-indian-court-ruling.html**, published in 2011, accessed by the author on February 2022), the Bombay High Court has ruled that astrology is a science. Most, if not all of us, have looked up our star signs at some point in our lives. Whether or not we believe in the predictions, there is an inherent curiosity in all of us to understand what the future holds for us. While the future seems to be an outcome of a choice we make in the present in general, there are events that seem completely unrelated as well. And it is this uncertainty that drives us to define and understand it. Astrology is one way to help understand the future; it is the oldest, most accepted and time-tested method. We human beings do love to see what is in store, and if you feel the urge to look up predictions for your star sign, please do.

With machines, there is one big distinction. To understand this, let us look at the definition of Astrology. It is the art and science of predicting future events in our lives using the relative positions of stars and planets over time from our birth date and time as the starting point. A chart is created denoting the planetary positions at birth, and it is used to derive future events based on perceived positions of these planetary bodies for the future timeframe in question. So, there is a semblance of past data being used, but only to define a structure for evaluation (akin to designing data structures and algorithms in computing). In Machine Learning or ML, the origination timeframe for the data is not as relevant as the volume of historical data (or past events) needed. Even more important is the correlation between the individual attributes in the data to what we want to predict. These attributes are what are call Features in ML. In simple terms, these are the columns in your dataset.

For example, let's say you want to predict next year's house prices. To train a ML model to do this, you would need historical data with attributes like locality, house size, number of bedrooms, home loan interest rates, loan options, good schools nearby, accessibility to shops, does it have amenities like swimming pool, year of construction, and so on. These are the input features. The historical data will also the contain house prices. This is the target feature. The dataset you use for training the model (the historical data) will contain both input features and the target feature. The target feature serves as the label for supervised training of a ML model. The model will learn to approximate a function to predict the target feature from the input features in the data. After training, you can use the model to predict the house price (the target feature) by sending new input values for house characteristics (the input features).

And that's why the most important aspect of designing a ML solution is feature engineering or the technique of collecting, harvesting, and transforming features from your dataset. As we saw earlier, features define the inputs and the output of the ML model and will have to be considered very carefully as they can directly influence the success of the model predictions. In the previous chapter, you learned how to hydrate your data lake using various ingestion mechanisms from diverse data sources using AWS services. This data that you import contains the features we need to train our ML models. In this chapter, you will learn what features are, why they are important, how to select the features that matter, and different techniques for data collection, preparation and usage. We will dive deep into feature engineering techniques for various ML types, like Natural Language Processing or NLP, Computer Vision or CV, and Tabular data. We will learn by running Python code examples that show various feature engineering tasks in action and understanding the benefits of these techniques and what to use when.

Structure

In this chapter, we will dive deep into the following topics:

- Introducing feature engineering
- Feature engineering for NLP
- Feature engineering for computer vision
- Feature engineering for tabular data

Objectives

By the end of this chapter, you should have a very good grasp on why features are important in ML, the definition of feature engineering, how to perform some of the most common feature engineering techniques in different types of ML use cases with actual code examples such as tabular regression/classification, natural language processing or NLP, and object detection for computer vision tasks.

Technical Requirements

We are going to be very hands-on in this chapter, and access to an AWS account is mandatory. Follow the instructions in the Setting up your AWS account section of *Chapter 2, Hydrating Your Data Lake*, to sign up for an AWS account. Once you have signed up, log in to your AWS account using the instructions at **https://docs.aws.amazon.com/IAM/latest/UserGuide/console.html**.

Onboard to SageMaker Studio

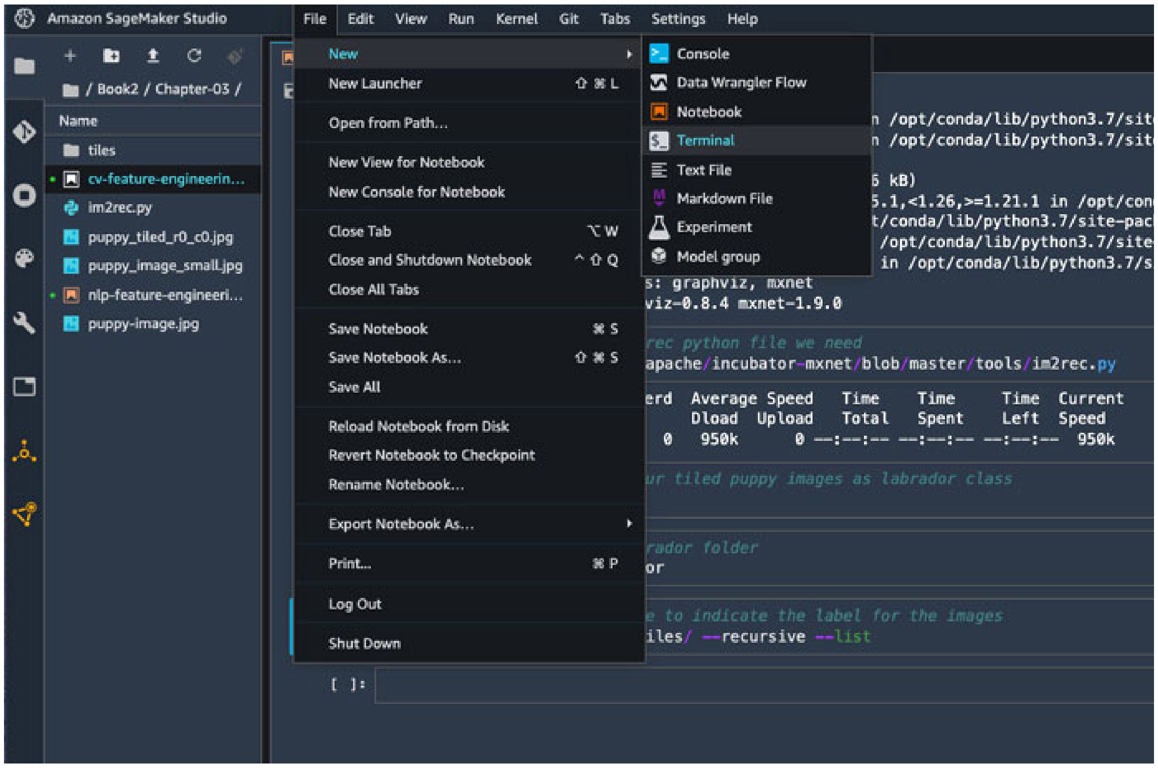
Next, you need to onboard to SageMaker Studio and open your Studio domain to get started. Follow the instructions in this documentation (**https://docs.aws.amazon.com/sagemaker/latest/dg/onboard-quick-start.html**) for step-by-step instructions; make sure you do not miss any step.

Hint: When you use SageMaker Studio to run ML training jobs or set up deployment instances, you will incur costs after the free tier limits have been exceeded. If you want to look at examples of how you can do ML training without an AWS login, you can try the Amazon SageMaker Studio Lab. This blog post contains the instructions on how to try it out: https://towardsdatascience.com/amazon-sagemaker-studio-lab-for-beginners-b5421b1550d3. Studio Lab is a good way to quickly prototype your ML use cases, but for trying out all the exercises in this book, you need access to SageMaker Studio through the AWS Management Console.

Cloning the repository to SageMaker Studio

Once you are logged in to the SageMaker Studio, you need to clone the book's repository to start running the examples. Follow the steps given here to do this:

1. When you are in SageMaker Studio, you will see a Launcher page with instructions to get started. From the top menu in the page, click on **`File`**, and then click on **`New`**, followed by **`Terminal`**. Refer to *Figure 3.1*:

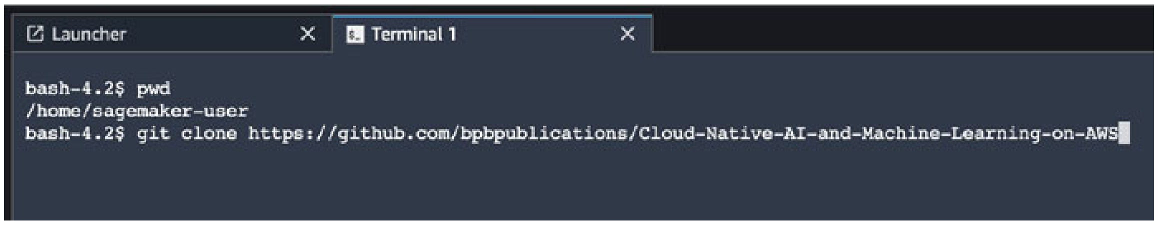


```
n /opt/conda/lib/python3.7/sit
n /opt/conda/lib/python3.7/sit
6 kB)
5.1,<1.26,>=1.21.1 in /opt/con
t/conda/lib/python3.7/site-pac
 /opt/conda/lib/python3.7/site
 in /opt/conda/lib/python3.7/s
s: graphviz, mxnet
viz-0.8.4 mxnet-1.9.0
rec python file we need
apache/incubator-mxnet/blob/master/tools/im2rec.py
erd  Average Speed   Time    Time     Time  Current
     Dload  Upload   Total   Spent    Left  Speed
  0   950k      0 --:--:-- --:--:-- --:--:--  950k
ur tiled puppy images as labrador class
rador folder
or
e to indicate the label for the images
iles/ --recursive --list
[ ]:
```

Figure 3.1: Open Terminal in SageMaker Studio

2. Now, git clone the GitHub repository by executing the command in the sagemaker-user directory:

   "git clone https://github.com/bpbpublications/Cloud-Native-AI-and-Machine-Learning-on-AWS"



```
bash-4.2$ pwd
/home/sagemaker-user
bash-4.2$ git clone https://github.com/bpbpublications/Cloud-Native-AI-and-Machine-Learning-on-AWS
```

Figure 3.2: Cloning the book GitHub repository

You can now close the `Terminal` window by clicking on `x` in the top-right corner of the window. Then, click on the folder icon on the left pane of the Studio window; this will bring up the folder for the book, along with the folders for each chapter. You can navigate to the chapter of your choice to run the examples.

Creating a S3 bucket and uploading objects

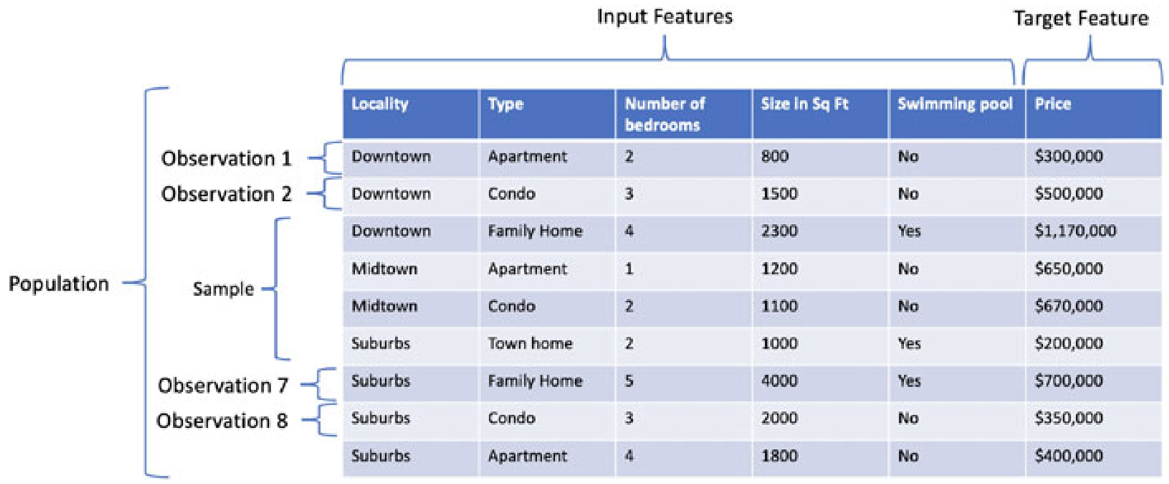
Follow the instructions in this documentation (**https://docs.aws.amazon.com/AmazonS3/latest/userguide/create-bucket-overview.html**) to create an Amazon S3 bucket. Almost all the coding chapters would need a S3 bucket, so it is better to create a bucket and then create multiple folders in the bucket one for each chapter.

Introducing feature engineering

Have you ever wondered (like the authors did) why do they (the people who invented ML buzz words) use complicated words like "feature engineering"? Granted it sounds cool to say it. If you were to say, "I specialize in feature engineering", you are sure to be treated with respect in the technology community because the words seem to have an important ring to them. But, why not something simple like data processing or data transform? It sure seems similar to what you would do to prepare your data for use in ML training right? Maybe. Maybe not.

Let's use the age-old adage of any demystification process and try to break it down. According to thesaurus.com (**https://www.thesaurus.com/browse/feature**), the word feature has several meanings, with the most prominent ones being aspect, item, quality, and attribute. As you can see, a feature refers to what is an important characteristic or what qualifies the data. These are the columns and their values in your dataset that enhance the significance of what your dataset represents as a whole. And that's exactly what we need for our ML training. There are other considerations before we arrive at the list of features we need; we will delve into that soon. Now that we understand the usage of the word *feature*, you might be thinking, why that specific word? Why not any other word that has a similar meaning? To answer this, we need to go to the subject of statistics, which is the foundation for the science behind ML. Statistics attempts to understand data by using measures like standard deviation, correlation and variance to explore how it is distributed and mapping the empirical relationships.

In statistics, datasets are referred to as populations, which are collections of data points that broadly lay out the scope for a particular question you want to answer or an experiment you are running. A sample in most cases is a random selection of data points from the population, but one that most closely represents the characteristics of the population of which the sample is a subset. In some instances, you may choose to be intentional about your samples, such as when you create an evaluation dataset. An observation is a single row of columns or attributes and indicates one instance of the sample or population. So, a sample is a collection of observations recorded across dimensions, such as time or other qualifying categories, that best represents the data. A feature is an attribute or column within an observation and denotes all recorded values for an attribute or column in a sample or population. You might be thinking hang on, that was also complex to understand. If so, our apologies. We will try an alternate approach. How about "a picture says more than a thousand words"? Refer to *Figure 3.3* to understand the function of a feature in the context of statistics:


Input Features
Target Feature
Population
Observation 1
Observation 2
Sample
Observation 7
Observation 8

Locality	Type	Number of bedrooms	Size in Sq Ft	Swimming pool	Price
Downtown	Apartment	2	800	No	$300,000
Downtown	Condo	3	1500	No	$500,000
Downtown	Family Home	4	2300	Yes	$1,170,000
Midtown	Apartment	1	1200	No	$650,000
Midtown	Condo	2	1100	No	$670,000
Suburbs	Town home	2	1000	Yes	$200,000
Suburbs	Family Home	5	4000	Yes	$700,000
Suburbs	Condo	3	2000	No	$350,000
Suburbs	Apartment	4	1800	No	$400,000

Figure 3.3: *Tabular data representation in the statistical context for ML*

We have talked about what a feature is long enough; let us move on to the second word: *engineering*. We all know what the word itself means, but what does it signify with respect to ML? It is simply whatever you do to get the features to a meaningful state and make them usable for ML training. Why do we have to do this? It is so because otherwise, you may not get a model that is able to predict anything well. The goal of ML is to minimize error in predictions, thereby meeting the needs of the business problem you are trying to solve. It doesn't matter how well the model trained; if the prediction is unreliable, it is not usable. And features are at the core of reliability of a model. So how do we engineer features? It depends on the ML use case (what are we trying to do?) and the type of learning required (how can we train the model based on data we have?).

If you recollect what we read in the *Approaching a ML problem* and *Overview of the ML workflow* sections of *Chapter 1, Introducing the ML Workflow*, we saw that based on the business or customer's need and the use case, you can select supervised learning, unsupervised learning or reinforcement learning, and the ML problem types/sub-types are varied, such as computer vision, speech, business tools, and industrial intelligence. These play a role in determining the types of features we work with, and the business need directly drives the features we select. In the following few subsections, we will explore common feature engineering techniques for these use cases.

Feature engineering for NLP

NLP is the approach or a collection of techniques to enable machines to intelligently cognize text-based data and derive contextual relationships. Some of the most popular uses of NLP are sentiment analysis (detecting the sentiment in a sentence or a paragraph), topic modeling (predicting the topic for a text document), classification (categorizing documents into classes or groups based on the document subject), and entity recognition (selecting words in documents that can be attributed to particular type, such as a person, place and date). The primary premise of a NLP model training involves converting text in the training document set into tokens or vector embeddings (machines do not understand text; they only understand numbers; embeddings are the numerical representations of words in a document based on frequency, similarity, and positioning of the word in relation to its neighbouring words). Depending on the learning objective (sentiment analysis or topic modelling for example), train to learn either the label corresponding to the group of words or common topics. In the subsequent chapters in this book, we will learn how to build some of these models; but first, we need to understand how we feature engineer for NLP training.

For example, let us consider an objective to train a model that can identify the author by providing quotes from their books as input. This means we need to train a text classifier. The following table is a sample of how our training dataset may look:

Quote	Author
Learning does not make one learned: there are those who have knowledge and those who have understanding. The first requires memory and the second philosophy.	Alexandre Dumas
For all evils there are two remedies - time and silence.	Alexandre Dumas
and as imagination bodies forth the forms of things unknown, the poet's pen turns them to shape, and gives to airy nothing a local habitation and a name	William Shakespeare

Table 1.1: *NLP text classification dataset*

Remember that the NLP model doesn't really care about how beautiful your text looks; it only needs to convert it into numbers (embeddings) and learn the relationships between vector embeddings of the Quote and the Author features. During feature engineering, we try and make training process easy by making the text as succinct as possible without losing the relationships or meaning. The following image depicts the various stages of a typical text feature engineering task. Use this as a general guideline, but you may want to perform additional steps based on your specific requirement. Refer to *Figure 3.4*:

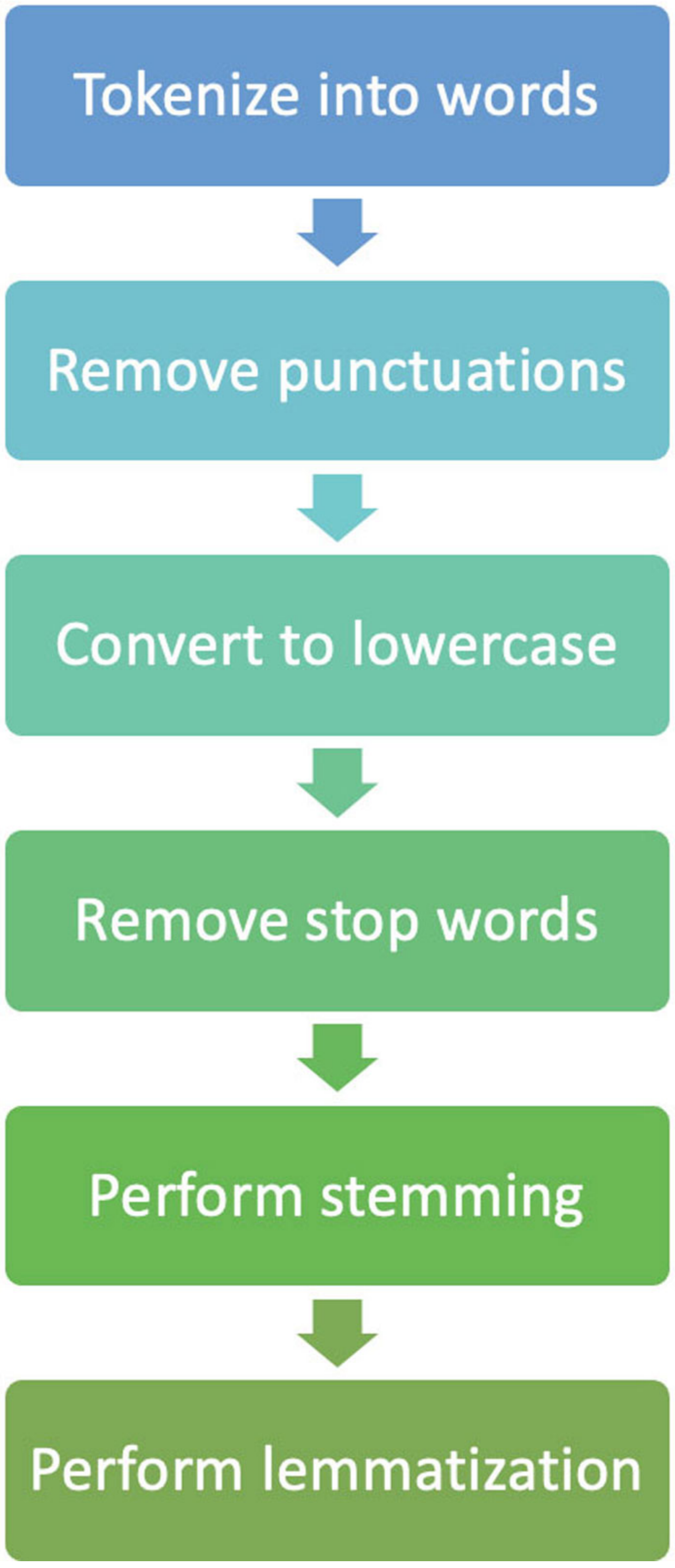


Figure 3.4: *NLP feature engineering stages*

Tokenize and remove punctuations

As a first step, let us tokenize our quotes into words (you may want to tokenize them into sentences before you tokenize into words) and remove any "unnecessary" punctuations (because we cannot convert them into numbers and even if we did, it would not help) from our text data above. To try the examples in this chapter, refer to the Technical Requirements section at the beginning of this chapter to sign in to the AWS management console, execute the steps in onboard to SageMaker studio, and execute cloning the repository to SageMaker Studio to get started. Click on the folder that corresponds to this chapter number. If you see multiple notebooks, the section title corresponds to the notebook name for easy identification. You can also passively follow the code samples using the GitHub repository provided at the beginning of the book.

If you are trying this locally, install Python (**https://wiki.python.org/moin/BeginnersGuide/Download**) before you proceed. We will install the Natural Language Toolkit (**http://www.nltk.org/**) Python library, import the data that the library needs, and run the code to remove the punctuations. Execute the following code after you have installed Python and nltk:

```
# import the nltk library after you have installed it
import nltk
# download the library subset that is needed for word tokenization
nltk.download('punkt')
# import the word tokenization function
from nltk.tokenize import word_tokenize
# creating a list of our quotes from the input dataset
quotes = ["Learning does not make one learned: there are those who
have knowledge and those who have understanding. The first requires
memory and the second philosophy.","For all evils there are two
remedies - time and silence.","and as imagination bodies forth the
forms of things unknown, the poet's pen turns them to shape, and
gives to airy nothing a local habitation and a name"]
# define a list for feature engineering output
fe1_quotes = []
for quote in quotes:
  # get the word tokens from each quote
  r_words = word_tokenize(quote)
  # remove punctuations
  p_words = [w for w in r_words if w.isalpha()]
  p_removed = ''
  # put the words with punctuations removed back into a sentence
  for p_word in p_words:
    p_removed += ' ' + p_word
  fe1_quotes.append(p_removed)
  print(p_removed)
```

The following table shows how our text dataset would look like with the punctuations removed:

Quote	Author
Learning does not make one learned there are those who have knowledge and those who have understanding The first requires memory and the second philosophy	Alexandre Dumas
For all evils there are two remedies time and silence	Alexandre Dumas
and as imagination bodies forth the forms of things unknown the poet pen turns them to shape and gives to airy nothing a local habitation and a name	William Shakespeare

Table 1.2: *NLP text classification dataset without punctuations*

Convert to lower case

During training, the NLP algorithm you choose will convert the words from your text into vector embeddings, as we discussed earlier in this chapter. A good approach would be to keep the inputs to this process as unambiguous as possible by converting text to one case only throughout the dataset. Execute the following code block to convert the quotes from your dataset to lower case. The Author column will be one-hot encoded (refer to the Tabular data feature engineering subsection) into integers prior to training:

```
fe2_quotes = []
for quote in fe1_quotes:
  fe2_quotes.append(quote.lower())
print(fe2_quotes)
```

The following table shows how our text dataset would look in lower case with punctuations removed:

Quote	Author
learning does not make one learned there are those who have knowledge and those who have understanding the first requires memory and the second philosophy	Alexandre Dumas

for all evils there are two remedies time and silence	Alexandre Dumas
and as imagination bodies forth the forms of things unknown the poet pen turns them to shape and gives to airy nothing a local habitation and a name	William Shakespeare

Table 1.3: *NLP text classification dataset in lower case*

Remove stop words

Stop words are frequently used words in text that do not mean anything on their own but help in structuring the sentence. In the preceding sentence, stop words were "are", "in", "on", "in", and "the". You get the gist. The **`nltk`** library provides a useful function to list stop words and also to remove these from our text dataset. Execute the following code to remove stop words:

```
# download the stopwords library
nltk.download('stopwords')
# import the word tokenization function
from nltk.tokenize import word_tokenize
# import the stop words function
from nltk.corpus import stopwords
# Define a new list for storing quotes with the stop words removed
fe3_quotes = []
# get the full list of stop words in the English language
stop_ws = set(stopwords.words('english'))
for quote in fe2_quotes:
  words = word_tokenize(quote)
  s_words = [word for word in words if not word in stop_ws]
  s_removed = ''
  for s_word in s_words:
    s_removed += ' ' + s_word
  fe3_quotes.append(s_removed)
print(fe3_quotes)
```

The following table shows how our text dataset would look with stop words removed:

Quote	Author
learning make one learned knowledge understanding first requires memory second philosophy	Alexandre Dumas
evils two remedies time silence	Alexandre Dumas
imagination bodies forth forms things unknown poet pen turns shape gives airy nothing local habitation name	William Shakespeare

Table 1.4: *NLP text classification dataset with stop words removed*

Perform stemming and lemmatization

As you can see, with NLP feature engineering, we are trying to make the text shorter and more succinct but without losing the meaning. The next steps in this process are stemming and lemmatization, which are techniques to prune words to get to the essence of what the word means and discarding the rest. The idea is to arrive at a "stem" or the "lemma" of a word without sacrificing its essence (to get the most accurate vector embedding for the word in relation to other words/similar words). The major difference between the two techniques is that a "stem" doesn't have to be an actual word, but a "lemma" needs to be. Execute the following code to perform stemming using the **`nltk`** library for our text dataset:

```
# import the stemming function
from nltk.stem.porter import PorterStemmer
# declare the porter stemmer, the most commonly used stemming
function
porter = PorterStemmer()
fe4_quotes = []
for quote in fe3_quotes:
  words = word_tokenize(quote)
  # stem each of the words for each of our quotes
  stem_words = [porter.stem(word) for word in words]
  stem_quote = ''
  for stem_word in stem_words:
    stem_quote += ' ' + stem_word
  fe4_quotes.append(stem_quote)
print(fe4_quotes)
```

The following table shows the quotes from our text dataset after stemming.

Note: See that some of the stemmed words are not actual words and hence, seem to have lost their meaning. However, during training, the NLP model will be able to derive contextual relationships from these words after vector embedding.

Quote	Author
learn make one learn knowledg understand first requir memori second philosophi	Alexandre Dumas
evil two remedi time silenc	Alexandre Dumas

imagin bodi forth form thing unknown poet pen turn shape give airi noth local habit name	William Shakespeare

Table 1.5: *NLP text classification dataset after stemming*

Let us now perform lemmatization using the **`nltk`** library. The following code block downloads the wordnet data library that provides the lemmatization function, imports the function, tokenizes the words from our quotes, and generates a lemma for each of the words:

```
# download the wordnet library for lemmatizer
nltk.download('wordnet')
# import the lemmatizer function
from nltk.stem import WordNetLemmatizer
# declare the lemmatizer
lemmatizer = WordNetLemmatizer()
fe5_quotes = []
for quote in fe3_quotes:
  words = word_tokenize(quote)
  # stem each of the words for each of our quotes
  lemma_words = [lemmatizer.lemmatize(word) for word in words]
  lemma_quote = ''
  for lemma_word in lemma_words:
    lemma_quote += ' ' + lemma_word
  fe5_quotes.append(lemma_quote)
print(fe5_quotes)
```

The following table shows the quotes from our text dataset after lemmatization.

NOTE: When you compare the output of stop word removal with the output of lemmatization, they look similar; however, notice that "remedies" became "remedy" in the second quote, and "bodies" became "body" in the third quote. Lemmatization gets to the essence of the word with the "lemma" still retaining meaning.

Quote	Author
learning make one learned knowledge understanding first requires memory second philosophy	Alexandre Dumas
evil two remedy time silence	Alexandre Dumas
imagination body forth form thing unknown poet pen turn shape give airy nothing local habitation name	William Shakespeare

Table 1.6: *NLP text classification dataset after lemmatization*

The techniques you learned in this subsection will get you started with your NLP project from a best-practices perspective. Based on your requirements and the type of NLP modelling you need, you may have to perform additional text transforms and pre-processing, such as parsing HTML web pages to extract text, and reading text off of images and PDF documents using Amazon Textract (**https://aws.amazon.com/textract/**), a fully managed service that provides easy text extraction using APIs. In the next subsection, we will learn some feature engineering techniques for image-based datasets.

Feature engineering for computer vision

Computer vision (CV) is the science of teaching computers how to see and understand. It refers to a collection of algorithms, concepts and approaches to classify images and videos, detect objects in images, and more. For example, with computer vision, your ML model can classify images to be dogs, cats or birds; it can identify a car in an image of a road, and it can distinguish between two cars in an image. There are several techniques, algorithms, neural networks and pre-trained models available as AWS AI services we can use to train CV models with high accuracy; we will cover these in the subsequent chapters. For now, what do you think are the features for an image dataset? Tabular datasets have columns, and it is pretty easy to determine the features, but how can you do this with images? As humans, we have a unique way of perceiving images; we look at form, shape, colour, and size and rely on our memory to understand what an image means, but computers understand only numbers. So, we use pixel positions and their colour representations (RGB for colour and L for grayscale), which results in an array of arrays for training computers to learn images. Let us now see this in action.

To try the examples in this section, refer to the Technical Requirements section at the beginning of this chapter to sign in to the AWS management console, execute the steps in onboard to SageMaker studio, and execute cloning the repository to SageMaker Studio to get started. Click on the folder that corresponds to this chapter number. If you see multiple notebooks, the section title corresponds to the notebook name for easy identification. You can also passively follow the code samples using the GitHub repository provided at the beginning of the book.

Let us first open the image and check it out. Execute the following code in the SageMaker Studio notebook to display the image using the Python Pillow (**https://pillow.readthedocs.io/en/stable/**), a versatile and easy-to-use libraries for all kinds of image processing tasks. We will use the Python3 Data Science kernel in SageMaker that already has these libraries installed. If you are trying locally, you must install Python and the Pillow package:

```
# Lets use the Python image processing Pillow library
from PIL import Image
img = Image.open('puppy-image.jpg')
display(img)
```

The following image is displayed when you execute the previous code:

Figure 3.5: *Sample image for feature engineering*

For image processing the feature engineering techniques are very different from what we do with text. Typically, we review the image size and resize the images so that all of them are the same size across the training and test datasets, cropping and tiling the images, rotating the images, and convert to grayscale if needed. The whole idea is to pre-process the images to get them to be the right quality and size for models to train well from them. In some cases, we may have to take a few additional steps, such as converting the images to the RecordIo Protobuf format (**https://docs.aws.amazon.com/sagemaker/latest/dg/cdf-training.html#cdf-recordio-format**) using an MXNet utility called im2rec (**https://mxnet.apache.org/versions/1.6/api/r/docs/api/im2rec.html**) in Python for image classification problems, or use a technique like Principal Component Analysis or PCA for dimensionality reduction (**https://docs.aws.amazon.com/sagemaker/latest/dg/pca.html**) to optimize training.

Resizing Images

A good idea would be to ensure that all images are of good quality at the same size. Images of different sizes in a dataset can confuse the model during training, and some algorithms may not accept different image sizes and may give you validation errors. So as a first step, we should print the image size and convert it to a smaller size, as needed, without impacting the content. Smaller is better because images that are big have a lot of pixels, which means the model will take significantly longer to converge, which directly relates to higher costs. Execute the following code in the SageMaker Studio notebook to check how resizing works:

```
# Print the size of the image as width, height
img.size
```

We get the output as `(1920, 1280)`, which indicates the width and height of the image. Now, execute the following code to resize the image and save it as a new image:

```
# Resize the image to a smaller size
img_smaller = img.resize((600, 400))
img_smaller.save('puppy_image_small.jpg')
print(img.size)
print(img_smaller.size)
```

We get the results as (1920, 1280) and (600,400), which is as expected. Now, execute the next line to display this resized image. As you can see, we have scaled down the image size without losing the content quality. To understand why resizing is important, let us review the image as an array of pixels. Execute the following lines of code to get the pixel representation in RGB (Red, Green and Blue) notation of the original image and print the first five pixels:

```
# Get the pixel values from the image width, height = img.size
pixels = {'R':[],'G':[],'B':[]}
# Let us now get the RGB value for each pixel position for w in
range(1,width):
  for h in range(1, height):
    pixpos = img.getpixel(((w,h)))
    pixels['R'].append(pixpos[0])
    pixels['G'].append(pixpos[1])
    pixels['B'].append(pixpos[2])
# Print the first 5 pixels in the first width position and up to 10
pixels height for i in range(1,6):
```

```
print("R: "+str(pixels['R'][i])+' '+"G: "+str(pixels['G'][i])+'
'+"B: "+str(pixels['B'][i])+' ')
```

We get the following print statement from the code execution:

```
R: 241 G: 238 B: 221
R: 240 G: 237 B: 220
R: 239 G: 236 B: 219
R: 239 G: 236 B: 219
R: 238 G: 235 B: 218
```

Now, let us print the size of the pixels dictionary to understand how many RGB pixel values we got from the original image. Execute the following code and review the results:

```
# How many pixels do we have?
len(pixels['R'])
```

We get the result as **`2454401`** for the count of pixels just for R. We will have similar counts for G and B. Let us now perform the same exercise for the resized image. Execute the following code in the SageMaker Studio notebook and review the results:

```
# Get the pixel values from the smaller image
width, height = img_smaller.size
pixels = {'R':[],'G':[],'B':[]}
# Let us now get the RGB value for each pixel position
for w in range(1,width):
  for h in range(1, height):
    pixpos = img.getpixel(((w,h)))
    pixels['R'].append(pixpos[0])
    pixels['G'].append(pixpos[1])
    pixels['B'].append(pixpos[2])
# How many pixels do we have?
len(pixels['R'])
```

We get the result as **`239001`**, which is 10 times smaller than the original pixel count. In this case, the model will process 10 times less data, which saves time and reduces costs significantly.

Cropping and tiling images

Another technique to reduce image data and improve training performance is to crop images and remove unwanted content in them. These can be the background or parts of the image that do not add value in recognizing what we want the model to learn. When we crop parts of the image, we automatically reduce the number of pixels, which reduces training time. Let us execute the following lines of code in the SageMaker Studio notebook to crop our image and review the results:

```
# Cropping and tiling our image
# define cropping coordinates
crop_coord = (130,75,460,350)
# crop the image
img_cropped = img_smaller.crop(crop_coord)
display(img_cropped)
```

We get the following image in the notebook as an output:

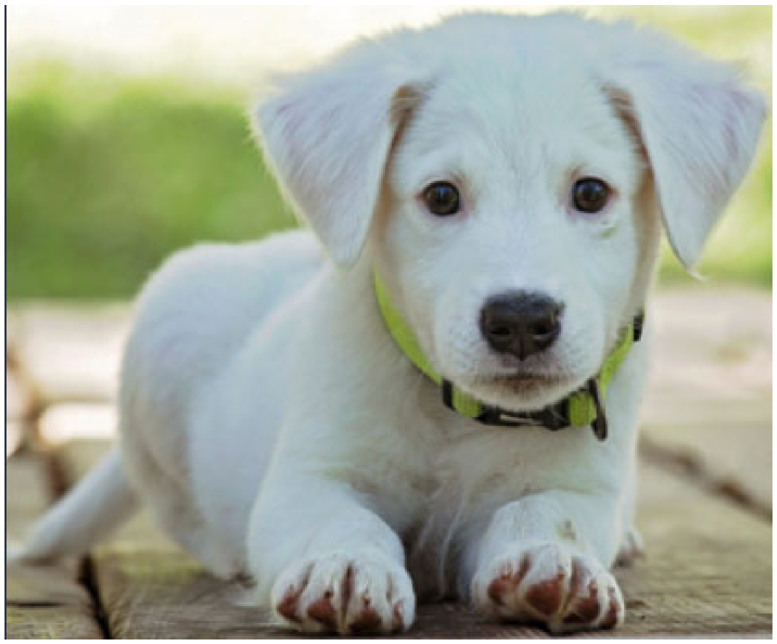


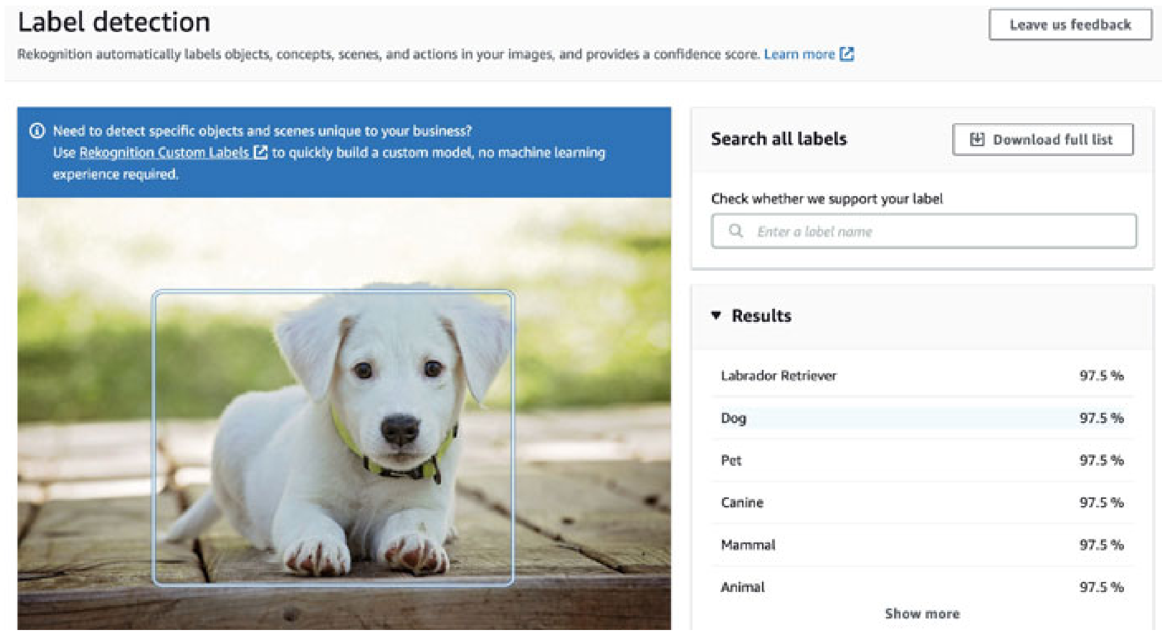
Figure 3.6: *Image cropped to remove unneeded portions*

The coordinates are the pixel positions (not the RGB representations for each pixel position) denoting the top-left corner and the distance to the bottom-right corner of the image we cropped (we used the resized image as the input). So, 130 is the pixel position that we started from in the left of the input image, and 430 is the pixel distance up to which we go in the right; 75 is the starting pixel position at the top of the image, and 350 is the pixel distance we go to the bottom of the image. Let us now understand the size of this image in comparison to the input image (the resized version). Execute the following code in the SageMaker Studio notebook and review the results:

```
# Get the pixel values from the smaller image
width, height = img_cropped.size
pixels = {'R':[],'G':[],'B':[]}
```

```
# Let us now get the RGB value for each pixel position for w in
range(1,width):
  for h in range(1, height):
    pixpos = img.getpixel(((w,h)))
    pixels['R'].append(pixpos[0])
    pixels['G'].append(pixpos[1])
    pixels['B'].append(pixpos[2])
# How many pixels do we have?
len(pixels['R'])
```

We get the result as 90146, which is less than half the pixel count of our resized input image (which was 239001). So, cropping can be really helpful as long we determine the coordinates correctly. An automated way to determine the coordinates for cropping will be to use an object detection model that can draw bounding boxes around objects in images. A great example of this is Amazon Rekognition (**https://aws.amazon.com/rekognition/**), a fully managed service using powerful pre-trained models for object detection and image classification for various use cases. We will learn more about Rekognition in *Chapter 10, Adding Intelligence With Sensory Cognition*. When we use the `Label detection` feature in Rekognition with the image of our puppy, we get the following results:



Label detection

Rekognition automatically labels objects, concepts, scenes, and actions in your images, and provides a confidence score. Learn more

Need to detect specific objects and scenes unique to your business? Use Rekognition Custom Labels to quickly build a custom model, no machine learning experience required.

Labrador Retriever	97.5 %
Dog	97.5 %
Pet	97.5 %
Canine	97.5 %
Mammal	97.5 %
Animal	97.5 %

Figure 3.7: *Amazon Rekognition analysis of the puppy image*

As we can see, Rekognition not only drew the bounding box accurately but was also able to predict that this was the image of a dog and its breed is Labrador

Retriever. Let us now learn about a related technique that is frequently used when working with large image sizes: image tiling.

Sometimes images are extremely large (running to 100s of MBs or GBs in file size) and contain complex details that prevent us from using the resizing or cropping techniques we learned earlier. For example, a satellite image of a residential neighbourhood will contain useful information in every area of the image and will lose context/quality if we resize and lose content if we crop away portions of the image. In this case, the best approach will be tiling. Tiling uses cropping, but for a different reason. In tiling, we crop a big image into many small sections, with each section becoming an independent image, and we store the coordinate positions of the individual images to be able to stitch together the full image again. It's like a jigsaw puzzle but with almost all the pieces (small sections of images) cropped into square or rectangular shapes and equal sizes. Execute the following code in your SageMaker Studio notebook under the title Tiling our image to tile our puppy image and review the results:

```
# We will now use the crop function to create multiple tiles of our
image import math
# get the dimensions of the input image - we are using the resize
image for this example
w, h = img_smaller.size
# How many images do we want? Feel free to change this value but it
should be a perfect squared number
# And the width and height of the input image must be divisible by
this square root
tiles = 16
divisor = int(math.sqrt(tiles))
left_pixel = 0
top_pixel = 0
right_pixel = w/divisor
bottom_pixel = h/divisor
# traverse from the left to right of the image or traverse through
the columns for i in range(divisor):
  if right_pixel <= w:
    # tiling position to attach to the image name referring to the
    row and column number
    # such as puppy_tiled_r0_c1.jpg, puppy_tiled_r0_c2.jpg…
    top_pixel = 0
    # initialize to first row here
    j = 0
    bottom_pixel = h/divisor
    tiled_img = img_smaller.crop((left_pixel, top_pixel, right_pixel,
    bottom_pixel))
    tiled_img.save('tiles/puppy_tiled_' + 'r'+str(j) + '_c'+str(i)
    +".jpg")
    # now traverse down the height of the image - traverse through
    the rows for each column for j in range(1,divisor):
      top_pixel += h/divisor
      bottom_pixel += h/divisor
      if bottom_pixel <= h:
        tiled_img = img_smaller.crop((left_pixel, top_pixel,
        right_pixel, bottom_pixel))
        tiled_img.save('tiles/puppy_tiled_' + 'r'+str(j) + '_c'+str(i)
        +".jpg")
    # increment pixel positions
    left_pixel += w/divisor
    right_pixel += w/divisor
```

This creates 16 tiled images of our input puppy image (that was resized earlier) into a separate folder called tiles in the current directory. Now, execute the code in the following cell in the notebook to display the tiled images in a grid. We will use subplots in matplotlib (**https://matplotlib.org/3.5.1/api/_as_gen/matplotlib.pyplot.subplots.html**) for this task:

```
# Lets use matplot lib to display the tiled images
import os
import matplotlib.pyplot as plt
images = []
for tile in sorted(os.listdir('./tiles')):
  if tile.endswith(".jpg"):
    images.append(plt.imread('./tiles/'+tile))
f, pltarr = plt.subplots(divisor, divisor, figsize=(12,8))
for j, row in enumerate(pltarr):
  for i, axis in enumerate(row):
    axis.imshow(images[j*divisor+i])
    axis.set_title(f'tile {j*divisor+i+1}')
title = 'Puppy tiled images'
f.suptitle(title, fontsize=16)
plt.show()
```

You can see the following image displayed as the output:

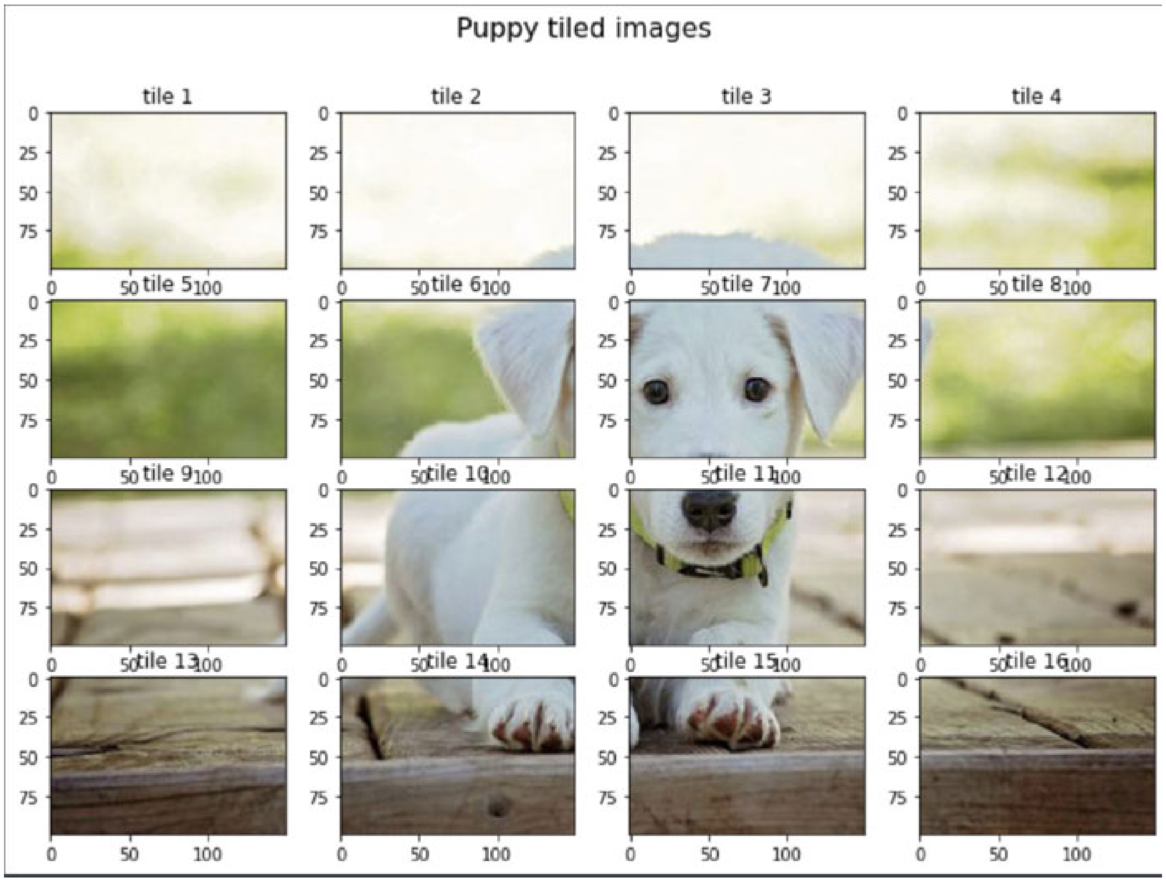


Figure 3.8: *Tiled images*

If you want to dive deeper into image processing utilities, check out ImageMagick (https://legacy.imagemagick.org/), an easy-to-use tool for comprehensive image manipulation.

Rotating images

Sometimes, the requirement might be to just rotate images across your dataset so that they are all of the same disposition. This is important because the model learns by understanding the RGB values for each pixel position. So, if you have two images of the same object at different angles, it can confuse the model. Execute the following code in the SageMaker Studio notebook to rotate your image to different angles and display it:

```
# rotate images to various angles
angles = [45,90,180,225]
for angle in angles:
  display(img_smaller.rotate(angle))
```

Converting to grayscale

In some cases, especially using deep learning for complex use cases, our image training dataset may be very large, with hundreds of thousands or even millions of images. We typically set up a distributed training environment with many computers in a cluster for this. But if we are looking to save on compute costs and time, there is a way for us to reduce the number of parameters a model needs to learn without reducing the volume of the images. For computer vision, we do this by converting our images to grayscale. Remember that the model learns the RGB values for pixels, which is an array of three elements per pixel. Depending on the size of the image, this can translate to millions of three-element arrays per image. Multiply that by a million images and very soon, it can become compute-intensive for training. So, we convert images to grayscale to reduce the number of data points the model has to learn. In grayscale, there is only one data point indicating the brightness level of the pixel for each pixel rather that the three values in RGB. Execute the following cell in the SageMaker Studio notebook to convert your image to a grayscale:

```
# Convert image to gray scale aka black and white
gray_img = img_smaller.convert('L')
display(gray_img)
```

The following image is displayed as the output:

Figure 3.9: *Grayscale image*

What we discussed so far are some common image processing techniques to improve the learnability of our CV-based ML model. This is by no means an exhaustive list of techniques to try out; there are other things you can do based on your use case, such as combining, change brightness, contrast, and colours. For a full list of what is possible, refer to **https://pillow.readthedocs.io/en/stable/reference/Image.html**. In the following subsections, we will learn some additional things we can do that do not involve direct image manipulation to make our CV training efficient.

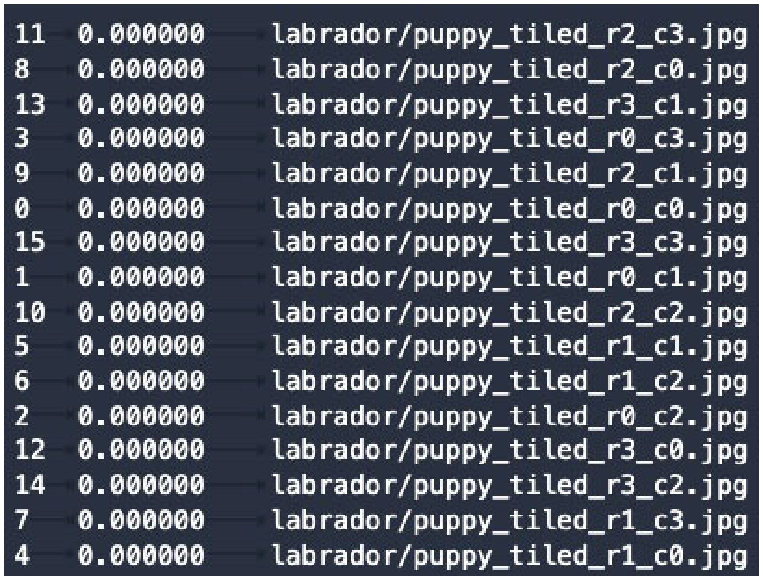
Converting to RecordIO format

Image formats like JPEG and PNG are popular because they are portable and compressible. However, when these images are used for computationally intensive tasks like ML training, IO reads from disks can result in a bottleneck because these formats are not tiled or layered. To improve training performance, you can convert the images to MXNet RecordIO format (**https://mxnet.apache.org/versions/1.7/api/python/docs/api/mxnet/recordio/index.html**). RecordIO can be used to pack image files as a sequence of records that are both compact and accessible. Conversion is easy with a couple of lines of code using the im2rec (**https://github.com/apache/incubator-mxnet/blob/master/tools/im2rec.py**) utility provided by MXNet. You can convert your entire image dataset to RecordIO for training as required. Let us use our tiled images as an example to try this out. We need to install MXNet for this activity. You can either use the following code to install MXNet, or you can change the kernel of your SageMaker Studio notebook to a MXNet kernel (**https://docs.aws.amazon.com/sagemaker/latest/dg/notebooks-run-and-manage-change-image.html**). Execute the following lines of code from the SageMaker Studio notebook to create the RecordIO format:

```
# first install mxnet. You can also use a kernel that has MXNet
already installed - select a kernel from the top right of SageMaker
Studio
!pip install mxnet
# we have to install opencv package for recordIo conversion
!pip install opencv-python-headless
# Download a copy of the im2rec python file we need
!curl -O https://raw.githubusercontent.com/apache/incubator-
mxnet/master/tools/im2rec.py
# create a folder to label our tiled puppy images as labrador class
!mkdir tiles/Labrador
```

```
# create a RecordIO list file to indicate the label for the images
!python im2rec.py ./puppy ./tiles/ --recursive –list
```

The preceding step first creates a list file that links the label identifier to the images. You can double-click on the file name on the left of your notebook pane to view the contents of this file, as shown in the following image:



```
11	0.000000	labrador/puppy_tiled_r2_c3.jpg
8	0.000000	labrador/puppy_tiled_r2_c0.jpg
13	0.000000	labrador/puppy_tiled_r3_c1.jpg
3	0.000000	labrador/puppy_tiled_r0_c3.jpg
9	0.000000	labrador/puppy_tiled_r2_c1.jpg
0	0.000000	labrador/puppy_tiled_r0_c0.jpg
15	0.000000	labrador/puppy_tiled_r3_c3.jpg
1	0.000000	labrador/puppy_tiled_r0_c1.jpg
10	0.000000	labrador/puppy_tiled_r2_c2.jpg
5	0.000000	labrador/puppy_tiled_r1_c1.jpg
6	0.000000	labrador/puppy_tiled_r1_c2.jpg
2	0.000000	labrador/puppy_tiled_r0_c2.jpg
12	0.000000	labrador/puppy_tiled_r3_c0.jpg
14	0.000000	labrador/puppy_tiled_r3_c2.jpg
7	0.000000	labrador/puppy_tiled_r1_c3.jpg
4	0.000000	labrador/puppy_tiled_r1_c0.jpg
```

Figure 3.10: *RecordIO list file contents*

Now, execute the following code to create the index file and the RecordIO file:

```
# now generate the recordIO file for our images which will be the
input for training
!python im2rec.py ./puppy ./tiles/ --recursive --pass-through --
pack-label
#you should see a puppy.rec file and a puppy.idx file created which
are inputs to your training
!ls -lt puppy*
```

It is really that simple to convert your images to the RecordIO format. Note that when you have a large collection of images, conversion may take up to a few hours, depending on the count and size of the images. We are now almost at the end of the CV feature engineering section.

Dimensionality reduction with Principal Component Analysis

Another pre-processing and feature engineering technique that you can apply to optimize your training is Principal Component Analysis or PCA (**https://docs.aws.amazon.com/sagemaker/latest/dg/pca.html**). It reduces the number of features in your dataset without compromising on the quality of the data. This is called dimensionality reduction because we are moving features from a high-dimensional space (lots of columns or features to represent an observation) to a low-dimensional space (reducing the number of features while retaining the meaning of the observation). With PCA, we can drastically reduce the number of columns in our dataset, thereby reducing the volume of data that needs to be processed during training, cutting costs and improving training efficiency. Typically, PCA is applied on tabular datasets, which we will explore in the following subsection of this chapter, but since we can represent images as a three-dimensional array of RGB pixel values, we can use PCA for images, provided the number of dimensions is reasonable. PCA transforms features into the number of components (which become the features for your actual ML training) that you specify. For example, if your dataset has 500 features, you can specify the number of components to be 40 as long as the number of observations is greater than 40 for your dataset. The number of components can be any number between 1 and the minimum of number of samples or the number of features. Let us see if PCA makes sense for our puppy image. Execute the following code blocks from the SageMaker Studio notebook:

```
# We need a numpy array of our image to get started. Import numpy
first
import numpy as np
# Let us convert our grayscale image to a transposed numpy array
first
seq = img_smaller.getdata()
img_pix_array = np.array(seq)
img_pix_array.shape
```

We get the result as `(240000, 3)`. There are 240000 rows or observations with three features (R, G and B). Let us now run the PCA analysis on this data using the Scikit Learn (https://scikit-learn.org/stable/index.html) decomposition module, as shown in the following code block:

```
# Let us use Scikit learn PCA to see the variability with all 3
features
from sklearn.decomposition import PCA
pca_3 = PCA(n_components=3)
pca_3.fit(img_pix_array)
print(pca_3.explained_variance_ratio_ *100)
```

We get the result as `[94.97632805 4.43190176 0.59177018]`. This indicates that the first feature accounts for 95% variability of the data and is good enough for our requirements. So, we will re-run the PCA with the **`n_components`** as 1:

```
# Component 1 accounts for 95% variability and is sufficient for us.
We will use number of components as 1 and run the transform
from sklearn.decomposition import PCA
pca_1 = PCA(n_components=1)
pca_1.fit(img_pix_array)
img_reduced = pca_1.transform(img_pix_array)
img_reduced.shape
```

We get the result as **`(240000, 1)`**, so we reduce the features from 3 to 1 with PCA. While this was interesting to learn, we can achieve the same results by converting our image to grayscale. For this specific example with just one image, PCA is not a good option to apply. If you get a dataset of 1000 images with similar characteristics as our image, it makes sense to use PCA. Here's a great example of how PCA can be used for ML problems using Amazon SageMaker: **https://sagemaker-examples.readthedocs.io/en/latest/introduction_to_amazon_algorithms/pca_mnist/pca_mnist.html**.

That brings us to the end of the subsection for feature engineering techniques in CV. In the next subsection, we will explore how to pre-process and feature engineer tabular data, which accounts for a majority of how ML training data is structured.

Feature engineering for tabular datasets

Data is ubiquitous and is in everything we do and experience. Data, in fact, provides the means by which we understand and interact with the world. If you stop for a moment and think about your day-to-day activities, you are using data and insights to make decisions every step of the way. For example, checking the weather to decide what you want to wear before you head out, finding the shortest route to your office from home, checking what time a grocery store closes to plan when you should leave from work, these are all part of our daily routine almost involuntary and yet reliant on data to help us decide. But hasn't this been what we have been doing all along? Why has it become so important suddenly? There are two major reasons for this: the cost of storage has come down significantly, and you can use data to uncover exciting possibilities with ML.

Hint: We discussed this in detail in the *Evolution of AI and ML* subsection in *Chapter 1, Introducing the ML Workflow*.

For data usage to remain relevant, it needs to depict variance in how it is distributed. Data that says the same thing repeatedly becomes immaterial. Further, we need a broad selection of variables that are not as much correlated with each other but together correlate toward generalizing a target feature you want the ML model to predict. Feature correlation and selection is a very important exercise to ensure that models are good at predicting values from data they have not seen before. Correlation is the extent to which an output feature/column's values are associated with or have relationship with a set of input feature values, such that this association can be used to approximate a prediction function. Features with good correlation are preferred, but if there is perfect correlation, the model may overfit (become very sensitive to the training data and perform poorly with test data); we must avoid this. On the other side of the spectrum, lesser correlation means the model may underfit (take a long time to learn or never be able to learn), so we want to be careful of that too. There is no magic number on what's the correlation threshold to aim for, and it is often the result of iterative experimentation, but a general rule is to select features with less than 85% correlation with each other.

When it comes to tabular datasets, the feature engineering techniques vary from what we have seen before. The end goal is the same, namely, prepare the features such that we achieve our goal metric for model performance. You will find that the techniques we use with tabular datasets are primarily statistical in nature, with a few alternate approaches based on the requirement. Alright, enough talk! Let's get started. Just like before, you need access to SageMaker Studio notebook to follow along and execute the code as we try out the various techniques.

To try the examples in this section, refer to the Technical Requirements section at the beginning of this chapter to sign in to the AWS management console, execute the steps in onboard to SageMaker studio, and execute cloning the repository to SageMaker Studio to get started. Click on the folder that corresponds to this chapter number. If you see multiple notebooks, the section title corresponds to the notebook name for easy identification. You can also passively follow the code samples using the GitHub repository provided at the beginning of the book.

We will use the Python3 Data Science kernel in SageMaker that has the libraries we need pre-installed. If you are trying locally, you must install Python and `pip install` the packages that we will import as part of our code blocks. We will start with simple techniques and progressively move on to advanced approaches. The example dataset we will use is the wine magazine dataset from Kaggle (**https://www.kaggle.com/pabloa/wine-magazine**), which contains 140K+ observations with nine input features and one target feature. The GitHub repository contains the data files required for running through the next few subsections. Go ahead and execute the code blocks in the notebook section Load data to create a Pandas DataFrame (**https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html**) object.

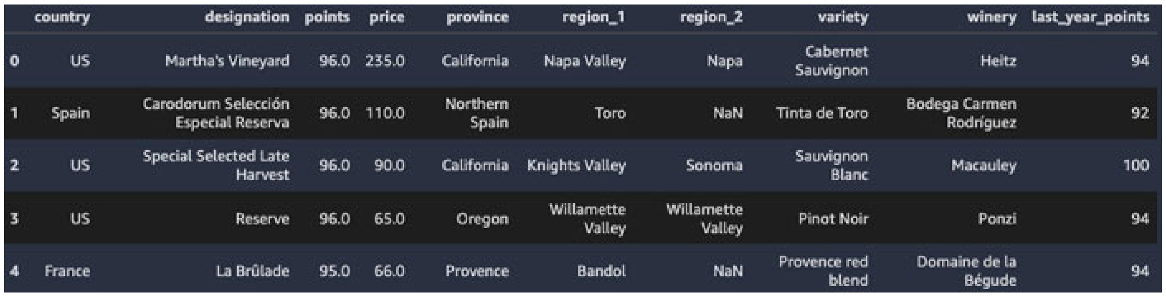
Exploring the data

First, we will load and walk through the data to understand what columns/features are there and what they mean. Ensure that you have executed the very first code cell to update the Scikit Learn package (**https://scikit-learn.org/stable/**) and restart the SageMaker Studio Kernel (from the top of the notebook page; then, click on **`Kernel`** and then on **`Restart Kernel`** and **`Clear All Outputs`**) before navigating to the **`Load data`** section. Execute the code block to unzip the wine magazine dataset (available at **https://www.kaggle.com/pabloa/wine-magazine**) provided in the GitHub repository available to you when you cloned the repository. Execute the following code to load the data to a Pandas DataFrame (**https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html**) to explore the date.

NOTE: The kernel we use in the SageMaker Studio notebook is a Python 3 Data Science version that includes most of what we need pre-installed.

```
import pandas as pd
# Let's first load the data into a Pandas dataframe so it is easy
for us to work with it
wine_raw_df = pd.read_csv('./winemag-data_first150k.csv',
sep=';',header=0)
wine_raw_df.shape
```

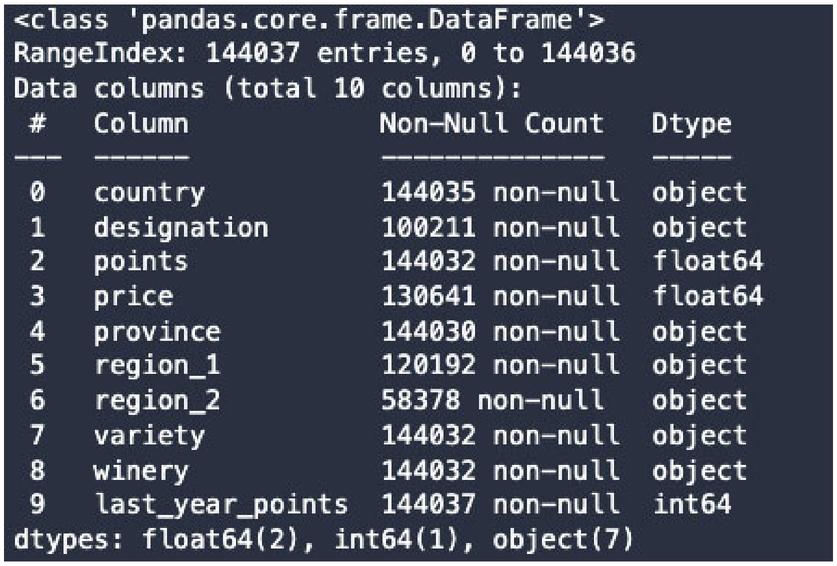
We get the result as (144037, 10), which means there are 144037 rows/observations/samples and 10 columns or features. Execute the next few cells to get more details on the data. When we run **`wine_raw_df.head()`**, we get the first five rows from the dataset with the following results:



	country	designation	points	price	province	region_1	region_2	variety	winery	last_year_points
0	US	Martha's Vineyard	96.0	235.0	California	Napa Valley	Napa	Cabernet Sauvignon	Heitz	94
1	Spain	Carodorum Selección Especial Reserva	96.0	110.0	Northern Spain	Toro	NaN	Tinta de Toro	Bodega Carmen Rodríguez	92
2	US	Special Selected Late Harvest	96.0	90.0	California	Knights Valley	Sonoma	Sauvignon Blanc	Macauley	100
3	US	Reserve	96.0	65.0	Oregon	Willamette Valley	Willamette Valley	Pinot Noir	Ponzi	94
4	France	La Brûlade	95.0	66.0	Provence	Bandol	NaN	Provence red blend	Domaine de la Bégude	94

Figure 3.11: *Wine magazine data first 5 rows*

Execute the **`wine_raw_df.info()`** code in the next cell to check whether there are any missing values in the dataset and review the results. Refer to Figure 3.12:



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 144037 entries, 0 to 144036
Data columns (total 10 columns):
 #   Column            Non-Null Count   Dtype
---  ------            --------------   -----
 0   country           144035 non-null  object
 1   designation       100211 non-null  object
 2   points            144032 non-null  float64
 3   price             130641 non-null  float64
 4   province          144030 non-null  object
 5   region_1          120192 non-null  object
 6   region_2          58378 non-null   object
 7   variety           144032 non-null  object
 8   winery            144032 non-null  object
 9   last_year_points  144037 non-null  int64
dtypes: float64(2), int64(1), object(7)
```

Figure 3.12: *Wine data checking for columns with null values*

Except the **`last_year_points`** feature, almost all columns have null values. The total number of rows is 144037, as shown in **`RangeIndex`**. If the count of Non-Null for each feature is less than 144037, the difference indicates the number of nulls we have for that feature. We need to use the technique of imputation to determine how to fill the missing values for these columns. Based on this information, our feature engineering strategy will be as shown in the following image:

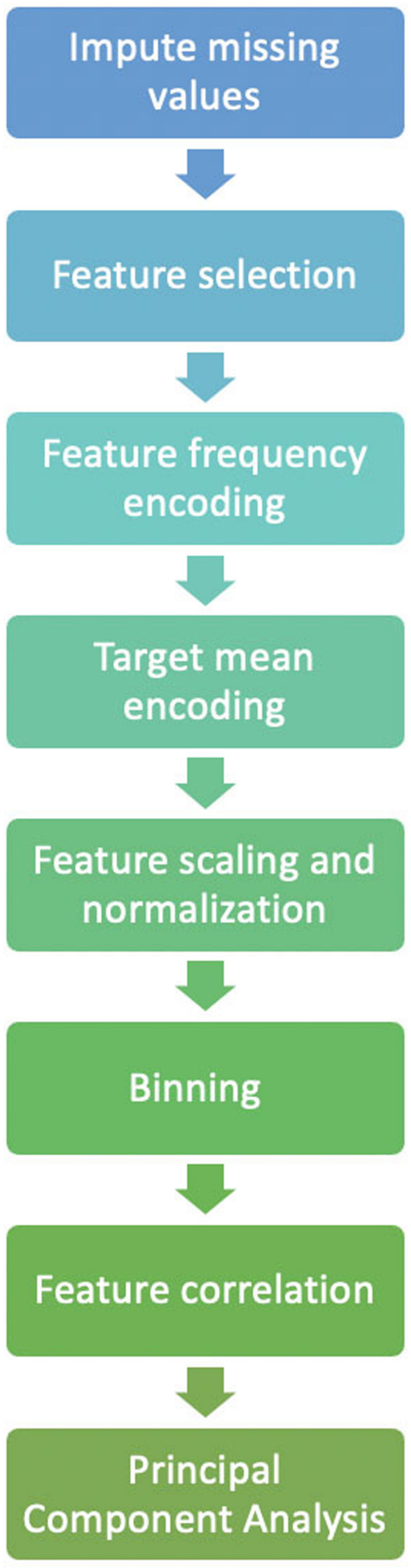

Impute missing values
Feature selection
Feature frequency encoding
Target mean encoding
Feature scaling and normalization
Binning
Feature correlation
Principal Component Analysis

Figure 3.13: *Tabular feature engineering tasks*

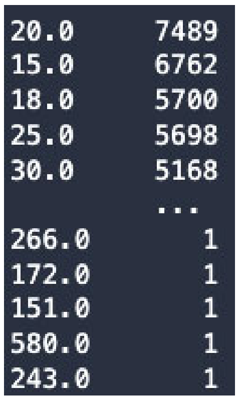
The sequence may vary based on your specific need, but this is what we will follow in this chapter, and we will see hints as we go along if a different order is warranted. First, we will impute the missing values in the price feature (this will be our target feature) and drop the rows where the features points, winery, country and variety are null because they are very few and dropping these rows won't impact the quality of training. Next, we will determine the encoding strategy for features country, designation, variety and winery because these have categorical tendencies. We will drop the features **`region_1`**, **`region_2`** and **`province`** because the country feature already provides the aggregation for them, and we are not interested in the level of detail these offer. Based on what the encoding yields, we may apply scaling, normalization, or binning and will finish up with dimensionality reduction with Principal Component Analysis (**https://docs.aws.amazon.com/sagemaker/latest/dg/how-pca-works.html**). In this chapter, we will use the Scikit Learn's PCA package (**https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html**) rather than SageMaker. Let us get going.

Imputing missing values

Often, when we get started, a dataset will be far from the shape we need it to be in for our ML training. The most common issue you will face is null or missing values in your dataset. The imputation technique helps you address this issue. Imputation is the process of intelligently determining the means to fill in the missing values. For numerical data, you can use the mean or median (if data is normally distributed; more on this later) of all the values for the feature to replace the missing values. If it’s text data, you can use the mode (most frequently occurring value) to fill missing values. In our case, we are imputing the price feature, which is numeric, so we will use mean as the measure. Execute the following code to get the count of rows for each unique value of price:

```
wine_raw_df['price'].value_counts()
```

We get the following results:



```
20.0     7489
15.0     6762
18.0     5700
25.0     5698
30.0     5168
          ...
266.0       1
172.0       1
151.0       1
580.0       1
243.0       1
```

Figure 3.14: *Price feature value counts*

Run the following code to check the count of null values first:

```
# Let's get the count of null values before imputation
wine_raw_df['price'].isnull().sum()
```

We get the result as `13396`. Next, run the following code to impute the missing value, which is the mean of all the prices across the dataset, and check the number of nulls again:

```
# we have prices ranging from $20 to $243 per bottle of wine - nice
# We can use a simple pandas imputation technique to replace the
missing values in price to a mean value
wine_raw_df['price'] =
wine_raw_df['price'].fillna(wine_raw_df['price'].mean())
# count the null values to see if anything remains - 0 is good
wine_raw_df['price'].isnull().sum()
```

We get the result as 0, which means there are no more nulls for price. We also see that the designation feature has many null values. We will impute this with the mode of the designation feature (most commonly occurring value). Execute the next few code blocks:

```
wine_raw_df.query('designation != designation')
wine_raw_df['designation'].mode()
```

We get the result as `Reserve`, which is the most common designation:

```
# designation also has a lot of null values let us impute using mode
wine_raw_df['designation'] =
wine_raw_df['designation'].fillna('Reserve')
# count the null values to see if anything remains - 0 is good
wine_raw_df['designation'].isnull().sum()
```

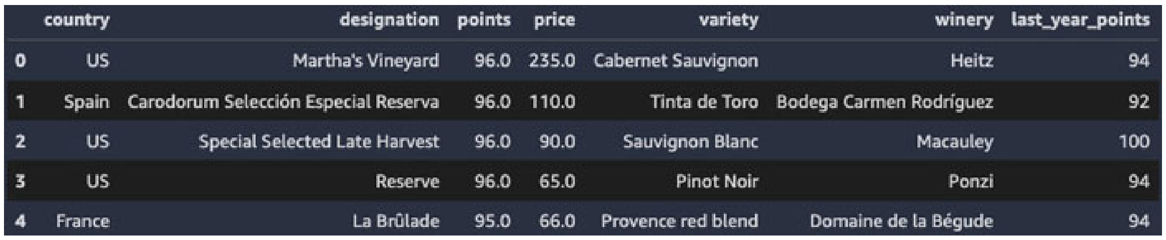
We get the result as `0`. Now, let us move on to feature selection and see if we can remove some of the features that are not needed and also trim down some of the observations.

Feature selection

Feature selection is implicitly tied into all the techniques we apply in feature engineering because that's what we are ultimately trying to do with all these approaches: arrive at an optimal list of features for our ML training. Here, we will intentionally remove the features and observations we don't need because they do not contribute to our objective. Let us execute the following code cells to remove the **`region_1`**, **`region_2`** and **`province`** features from our dataset.

```
wine_raw_df =
wine_raw_df.drop(['province','region_1','region_2'],axis=1)
wine_raw_df.head()
```

We get the results as shown in the following image:



	country	designation	points	price	variety	winery	last_year_points
0	US	Martha's Vineyard	96.0	235.0	Cabernet Sauvignon	Heitz	94
1	Spain	Carodorum Selección Especial Reserva	96.0	110.0	Tinta de Toro	Bodega Carmen Rodríguez	92
2	US	Special Selected Late Harvest	96.0	90.0	Sauvignon Blanc	Macauley	100
3	US	Reserve	96.0	65.0	Pinot Noir	Ponzi	94
4	France	La Brûlade	95.0	66.0	Provence red blend	Domaine de la Bégude	94

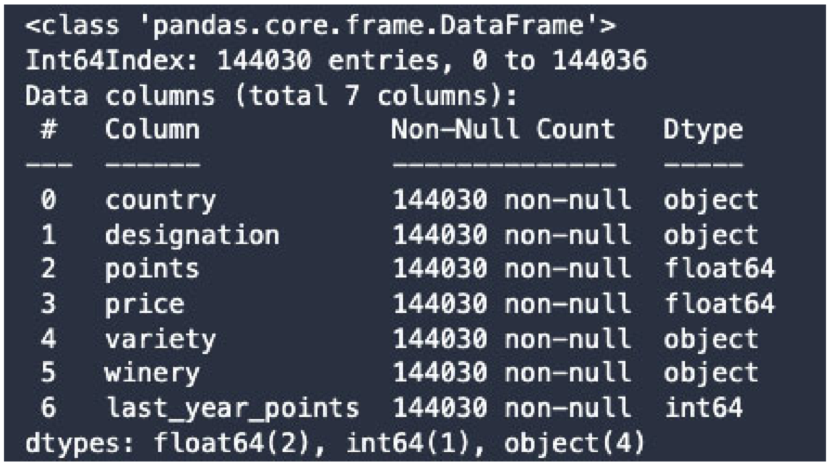
Figure 3.15: *wine dataset after dropping features*

Now, we will remove (rather than impute) some observations that have null values but are minimal in count. Execute the following code blocks:

```
# Let us drop the rows where country, points, variety and winery
have null values - there are just a few rows
# first get index values where country is NULL
wine_raw_df.query('country != country')
# drop the rows where country is null
wine_raw_df = wine_raw_df.dropna(axis=0, subset=['country'])
# check and drop the null value rows for points column
# looks like this will address null values in the other columns of
interest as well
wine_raw_df.query('points != points')
# drop the rows with null values
wine_raw_df = wine_raw_df.dropna(axis=0, subset=['points'])
# Let us check the null values once again
```

```
wine_raw_df.info()
```

We get the following results:



```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 144030 entries, 0 to 144036
Data columns (total 7 columns):
 #   Column           Non-Null Count   Dtype
---  ------           --------------   -----
 0   country          144030 non-null  object
 1   designation      144030 non-null  object
 2   points           144030 non-null  float64
 3   price            144030 non-null  float64
 4   variety          144030 non-null  object
 5   winery           144030 non-null  object
 6   last_year_points 144030 non-null  int64
dtypes: float64(2), int64(1), object(4)
```

Figure 3.16: *Null check after feature selection*

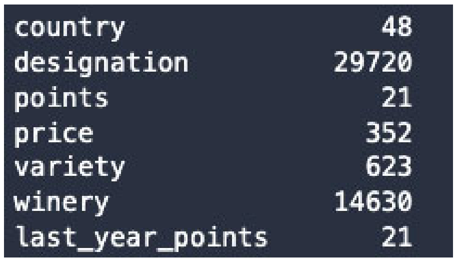
That brings us to the end of feature selection. We will now address the text value features in our dataset and perform encoding to convert them to numeric values needed for ML training. Depending on the feature and the number of observations, there are a few encoding approaches we can use, which we will discuss in the following sections.

Feature frequency encoding

Frequency encoding converts categorical features to a numerical representation that indicates the feature's frequency of occurrence across all observations. We will use this approach for the features designation and winery because they have many categorical values. Execute the following code block to check the unique values for each of our features:

```
wine_raw_df.nunique()
```

We see the results displayed in the following image with the unique value counts being high for **`designation`**, **`winery`** in that order:



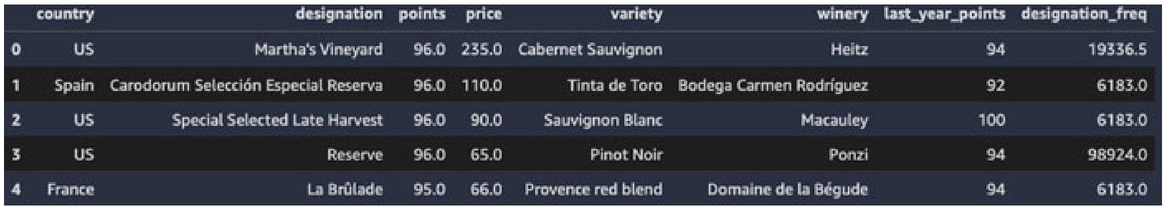
```
country               48
designation        29720
points                21
price                352
variety              623
winery             14630
last_year_points      21
```

Figure 3.17: *Unique value counts for each feature*

Run the following code block to frequency encode the designation feature with ranking included to prevent confusion when we have the same number of observations for multiple designation values:

```
from scipy.stats import rankdata
# Frequency encoding for designation
# first get a list of value counts for each designation type
desg_val_counts = wine_raw_df['designation'].value_counts()
# how frequently does each value occur across the entire dataset
desg_freq = desg_val_counts/len(wine_raw_df)
# Finally let us use a ranking of these frequency encoded values to
prevent issues with categories with similar frequencies
wine_raw_df['designation_freq'] =
rankdata(wine_raw_df.designation.map(desg_freq))
wine_raw_df.head()
```

We get the following results, with designation frequency encoded:



	country	designation	points	price	variety	winery	last_year_points	designation_freq
0	US	Martha's Vineyard	96.0	235.0	Cabernet Sauvignon	Heitz	94	19336.5
1	Spain	Carodorum Selección Especial Reserva	96.0	110.0	Tinta de Toro	Bodega Carmen Rodríguez	92	6183.0
2	US	Special Selected Late Harvest	96.0	90.0	Sauvignon Blanc	Macauley	100	6183.0
3	US	Reserve	96.0	65.0	Pinot Noir	Ponzi	94	98924.0
4	France	La Brûlade	95.0	66.0	Provence red blend	Domaine de la Bégude	94	6183.0

Figure 3.18: *Frequency encoding designation feature*

Let us now do the same thing for winery feature. Run the following code:

```
# Frequency encoding for winery
# first get a list of value counts for each winery type
win_val_counts = wine_raw_df['winery'].value_counts()
# how frequently does each value occur across the entire dataset
win_freq = win_val_counts/len(wine_raw_df)
# Finally let us use a ranking of these frequency encoded values to
prevent issues with categories with similar frequencies
wine_raw_df['winery_freq'] =
rankdata(wine_raw_df.winery.map(win_freq))
wine_raw_df.head()
```

Let us now apply target encoding for the variety feature, as shown in the following section.

Target mean encoding

In target mean encoding, we encode an input feature based on the mean of the target feature across all observations for that input feature value. From our dataset, we can see that **`variety`** and **`price`** do not have a linear relationship (or uncorrelated), so target encoding can be utilized. A word of advice though: target encoding may lead to overfitting (if a correlation already exists) because we are intentionally enforcing a correlation with the target feature. An alternative is to use Bayesian target encoding: **https://www.kaggle.com/mmotoki/hierarchical-bayesian-target-encoding**.

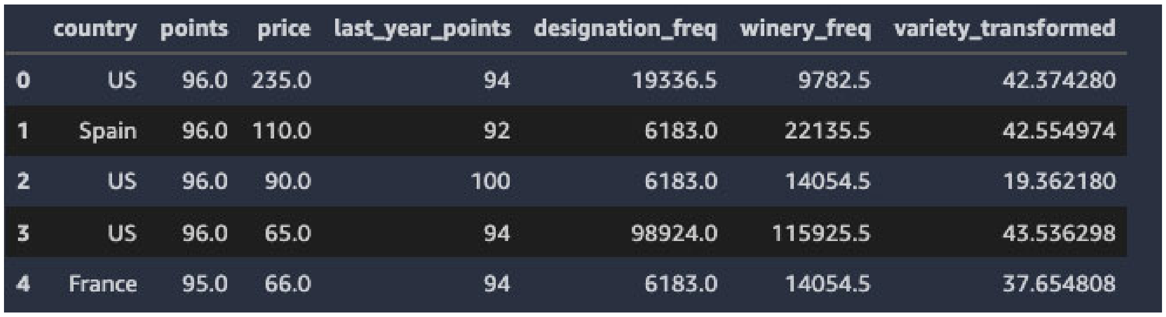
First, install categorical encoders package by running the following:

```
pip install category_encoders
```

Now, execute the following code blocks to apply target encoding for the **`variety`** feature:

```
import category_encoders as ce
tar_enc = ce.TargetEncoder(wine_raw_df['variety'])
wine_raw_df['variety_transformed'] =
tar_enc.fit_transform(wine_raw_df['variety'], wine_raw_df['price'])
# lets us move to a new dataframe and drop the categorical columns
we transformed
wine_ready_df = wine_raw_df.drop(['designation','variety','winery'],
axis=1)
wine_raw_df.head()
```

We get the following results:



	country	points	price	last_year_points	designation_freq	winery_freq	variety_transformed
0	US	96.0	235.0	94	19336.5	9782.5	42.374280
1	Spain	96.0	110.0	92	6183.0	22135.5	42.554974
2	US	96.0	90.0	100	6183.0	14054.5	19.362180
3	US	96.0	65.0	94	98924.0	115925.5	43.536298
4	France	95.0	66.0	94	6183.0	14054.5	37.654808

Figure 3.19: *Target encoding applied for variety feature*

We are almost there, but the country feature is still a text value. And since this has only 48 unique values, let us apply one hot encoding for this feature.

One hot encoding

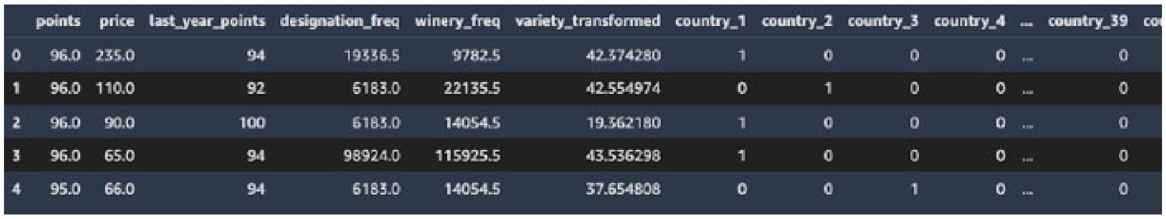
With one hot encoding, we prevent the common issue with regular categorical encoding. When we do categorical encoding, we convert text values to numbers.

Often, this is just a sequence of numbers. For example, if we have colours as a feature with values being [red, blue, green, yellow], categorical encoding will convert this to [1,2,3,4]. However, a machine will understand this as 4 > 3 > 2 > 1 or yellow > green > blue > red even though no such weightage is intended in the dataset. This will lead to poor model performance. With one hot encoding, every **`feature_value`** is converted to a new feature with 1 or 0 assigned as per original observation values for the feature. For more details, you can refer to https://scikit-learn.org/stable/modules/preprocessing.html#preprocessing-categorical-features.

Execute the following code block to one hot encode the **`country`** feature in our dataset:

```
one_enc = ce.OneHotEncoder(wine_ready_df['country'])
result = one_enc.fit_transform(wine_ready_df['country'])
wine_encod_df = pd.concat([wine_ready_df,result],axis=1)
wine_encod_df.drop(['country'], axis=1, inplace=True)
wine_encod_df.head()
```

We see the following results displayed:



	points	price	last_year_points	designation_freq	winery_freq	variety_transformed	country_1	country_2	country_3	country_4	...	country_39	co
0	96.0	235.0	94	19336.5	9782.5	42.374280	1	0	0	0	...	0	
1	96.0	110.0	92	6183.0	22135.5	42.554974	0	1	0	0	...	0	
2	96.0	90.0	100	6183.0	14054.5	19.362180	1	0	0	0	...	0	
3	96.0	65.0	94	98924.0	115925.5	43.536298	1	0	0	0	...	0	
4	95.0	66.0	94	6183.0	14054.5	37.654808	0	0	1	0	...	0	

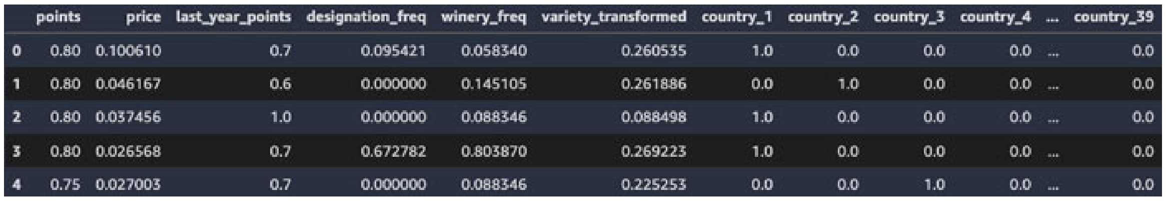
Figure 3.20: *One hot encoded result for country feature*

Feature scaling

You can perform scaling for the numerical features first, then perform one hot encoding for the country feature and finally, merge them together. In our case, we performed the one hot encoding earlier, and we will now scale our numerically converted dataset to see what results we get. The features we encoded and transformed are of different scales/bounds (check the value range of **`designation_freq`** and **`price`**, or **`winery_freq`** and **`variety_transformed`**). This will confuse the ML model, so let us scale them to comparable ranges between 0 and 1. We will use SciKit Learn packages (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html) because they are easy to work with. We will use **`MinMaxScaler()`** and convert all our numerical features to values between 0 and 1. Run the following code:

```
# Define the min max scaler to scale features to values between 0
and 1
from sklearn.preprocessing import MinMaxScaler
wine_features_scaler = MinMaxScaler()
wine_scaled_arr = wine_features_scaler.fit_transform(wine_encod_df)
# after scaling the results are in numpy array let's have a look at
the first row
# assign scaled results back to a dataframe and voila your feature
engineering dataset is ready
wine_scaled_df = pd.DataFrame(data=wine_scaled_arr,
columns=wine_encod_df.columns)
wine_scaled_df.head()
```

We get the results shown in the following image:



```
   points     price  last_year_points  designation_freq  winery_freq  variety_transformed  country_1  country_2  country_3  country_4  ...  country_39
0    0.80  0.100610               0.7          0.095421     0.058340             0.260535        1.0        0.0        0.0        0.0  ...         0.0
1    0.80  0.046167               0.6          0.000000     0.145105             0.261886        0.0        1.0        0.0        0.0  ...         0.0
2    0.80  0.037456               1.0          0.000000     0.088346             0.088498        1.0        0.0        0.0        0.0  ...         0.0
3    0.80  0.026568               0.7          0.672782     0.803870             0.269223        1.0        0.0        0.0        0.0  ...         0.0
4    0.75  0.027003               0.7          0.000000     0.088346             0.225253        0.0        0.0        1.0        0.0  ...         0.0
```

Figure 3.21: *Features scaled*

Let us take a look at normalization in the following section.

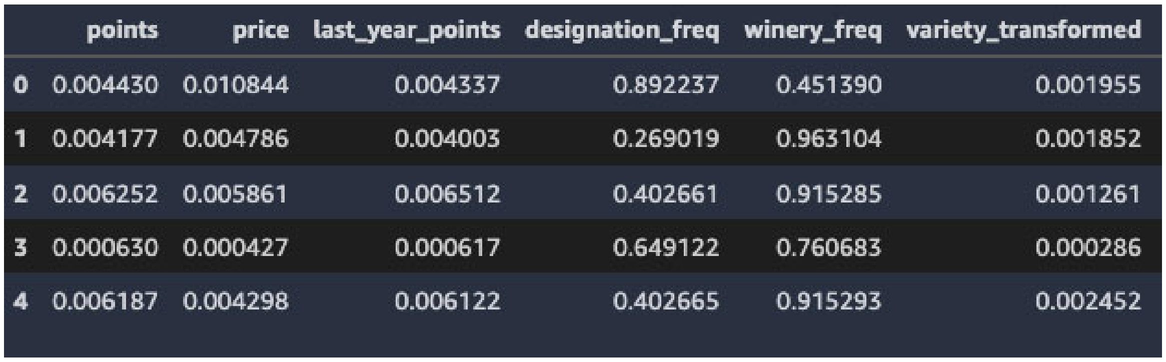
Feature normalization

Feature scaling and normalization are typically applicable for different types of ML problems. Scaling converts the range of values, while normalization changes the dataset to a normal distribution with most values around the mean and not extending beyond three standard deviations. We use scaling for regression and classification problems, and we use normalization for clustering problems. Execute the following code to normalize our dataset. We will select a version of the data before we scale it:

```
# with normalization
from sklearn.preprocessing import Normalizer
wine_features_normalizer = Normalizer()
wine_normalized_arr =
wine_features_normalizer.fit_transform(wine_encod_df)
# assign normalized results back to a dataframe and voila your
feature engineering dataset is ready
```

```
# technically you can also scale or normalize your numeric dataset
and then one hot encode the categorical data and merge them
wine_normalized_df = pd.DataFrame(data=wine_normalized_arr,
columns=wine_encod_df.columns)
wine_normalized_df.head()
```

We get the results shown in the following image. We should not normalize one hot encoded values; we should only normalize the numeric fields and then merge them with the one hot encoded values. In this example, we ran our normalization code on our entire dataset for the sake of simplicity:



	points	price	last_year_points	designation_freq	winery_freq	variety_transformed
0	0.004430	0.010844	0.004337	0.892237	0.451390	0.001955
1	0.004177	0.004786	0.004003	0.269019	0.963104	0.001852
2	0.006252	0.005861	0.006512	0.402661	0.915285	0.001261
3	0.000630	0.000427	0.000617	0.649122	0.760683	0.000286
4	0.006187	0.004298	0.006122	0.402665	0.915293	0.002452

Figure 3.22: *Normalized features*

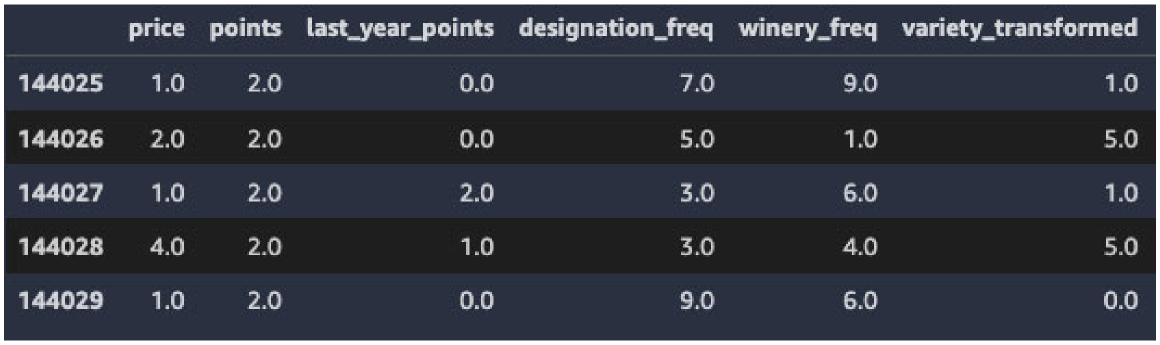
Binning

Binning, also called discretization, helps group continuous numerical variables into categories or bins, helping us understand how our data is distributed and whether there are any boundary conditions we need to watch out for. Binning helps model non-linear relationships better, improving the accuracy for regression use cases. Binning results in groups of feature values. After bins are created, we can use one hot encoding to further flatten the data, but only if the number of observations is minimal. In this example, we cannot do that, so we will settle for ordinal bin identifiers as the encoded value for our features. Execute the following code block to apply binning to our dataset:

```
# Let us use scikit learn package KBinsDiscretizer to group the data
into a different count of bins for each feature of interest
# and then encode them as an ordinal which will return the bin
identifier for each feature per row
# we will exclude the already one hot encoded country features and
bin the rest of the features
from sklearn.preprocessing import KBinsDiscretizer
```

```
wine_binner = KBinsDiscretizer(n_bins=[5,3,3,10,10,6],
encode='ordinal')
wine_binned_arr =
wine_binner.fit_transform(wine_encod_df[['price','points','last_year
_points','designation_freq','winery_freq','variety_transformed']])
# assign binned features back to a dataframe and voila your feature
engineering dataset is ready
wine_binned_df = pd.DataFrame(data=wine_binned_arr, columns=
['price','points','last_year_points','designation_freq','winery_freq
','variety_transformed'])
# concat back the country features
wine_binned_df =
pd.concat([wine_binned_df.reset_index(drop=True),wine_encod_df.iloc[
:,6:].reset_index(drop=True)],axis=1)
# display it
wine_binned_df.tail()
```

We will get the following results:



	price	points	last_year_points	designation_freq	winery_freq	variety_transformed
144025	1.0	2.0	0.0	7.0	9.0	1.0
144026	2.0	2.0	0.0	5.0	1.0	5.0
144027	1.0	2.0	2.0	3.0	6.0	1.0
144028	4.0	2.0	1.0	3.0	4.0	5.0
144029	1.0	2.0	0.0	9.0	6.0	0.0

Figure 3.23: *Binned features*

Feature correlation

After encoding and scaling, we now have a fully numeric dataset; so, we can run correlation algorithms to check how our features are correlated with each other. The less correlated the features are to the target feature, the poorer the model will perform and longer it will take to train. At the same time, if all features are highly correlated to the target feature, the model may not learn an approximation function and may tend to overfit on the training data. Balance is key; feature selection should use the correlation data to finalize the list of features we will use for ML training. But how do we understand the correlation between features. Fortunately, we have easy-to-use packages that can build the correlation and help us visualize it. We will use the Seaborn (**https://seaborn.pydata.org/**)

visualization library to create and view the correlation for our dataset. Execute the following code block to build the correlation for our dataset before scaling or normalization but after categorical encoding:

```
# Assuming we want to predict the price for a bottle of wine using
the input features let's see how they are correlated with each other
# we will use the seaborn visualization library
import seaborn as sns
# correlation of frequency and target encoded dataset
# we removed the country one hot encoded features for the sake of
displayability
sns.heatmap(wine_encod_df.iloc[:,0:6].corr(), annot=True)
```

We get the following heatmap as the result; 1.0 means fully correlated, and 0.0 indicates no correlation:

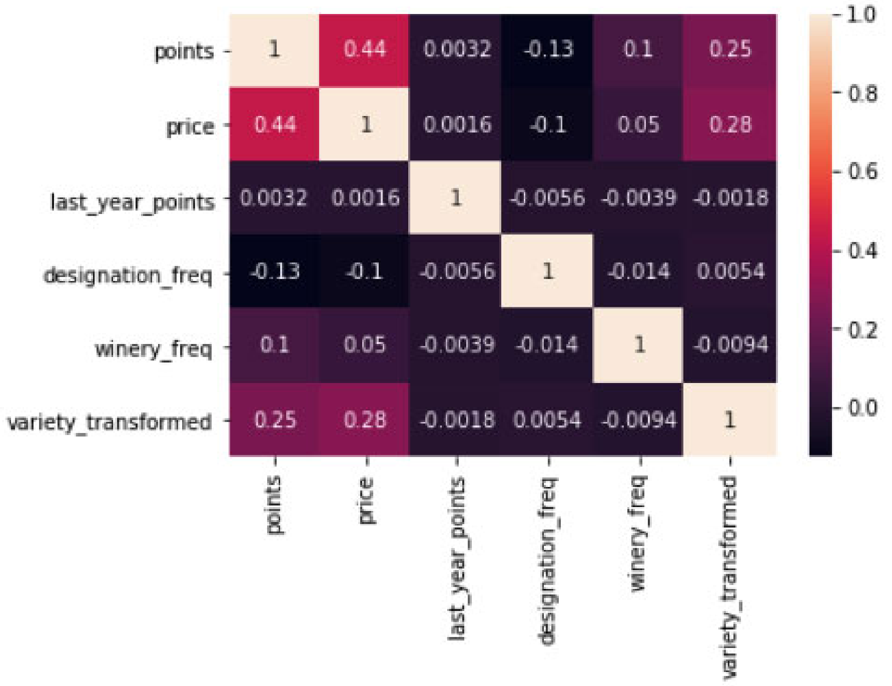


Figure 3.24: *Correlation heatmap for wine dataset after encoding*

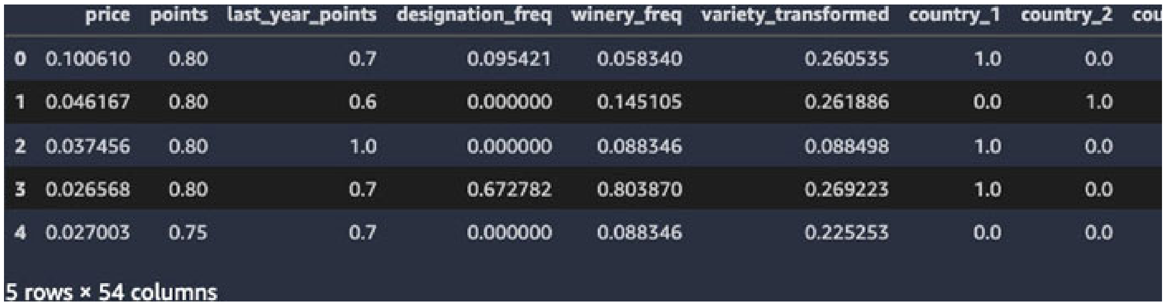
Continue to execute the rest of the code blocks in this section to visualize the correlation for scaled, normalized and binned dataset versions. Compare and contrast them to understand the variations. In the last section for tabular data feature engineering, we will apply dimensionality reduction using Principal Component Analysis (PCA) to reduce the number of input features we need for training.

Principal Component Analysis

We will check whether PCA can be used to reduce the number of features of our scaled dataset in preparation for training. We will use the same Scikit Learn package that we used in CV feature engineering. Execute the following code block to move the price feature in the first column because this will be our target feature. We will run PCA on our input features only:

```
# Lets check if PCA can help with our scaled dataset
# You can try this exercise for the rest of your datasets on your
own
# we will first make the price feature as the first column as its
our label
move_price = wine_scaled_df.pop('price')
wine_scaled_df.insert(0,'price',move_price)
wine_scaled_df.head()
```

We get the following result with the price feature as the first column now:



	price	points	last_year_points	designation_freq	winery_freq	variety_transformed	country_1	country_2	cou
0	0.100610	0.80	0.7	0.095421	0.058340	0.260535	1.0	0.0	
1	0.046167	0.80	0.6	0.000000	0.145105	0.261886	0.0	1.0	
2	0.037456	0.80	1.0	0.000000	0.088346	0.088498	1.0	0.0	
3	0.026568	0.80	0.7	0.672782	0.803870	0.269223	1.0	0.0	
4	0.027003	0.75	0.7	0.000000	0.088346	0.225253	0.0	0.0	

5 rows × 54 columns

Figure 3.25: *Moving price to be the first column in preparation for PCA*

We have a total of 54 features, with 53 input features and 1 target feature. We are trying to reduce the number of input features with PCA. First, we will run the following code block to set up a PCA model to create 20 components from our 53 input features to check how much variability in data these components handle:

```
# Let us use Scikit learn PCA to see the variability with 20
features
# only input features will be used, price will be ignore as its the
target feature
import numpy as np
from sklearn.decomposition import PCA
pca_20 = PCA(n_components=20)
#ignore the first column as its target feature
pca_20.fit(wine_scaled_df.iloc[:,1:])
print(pca_20.explained_variance_ratio_ *100)
```

We get the results shown as follows. We can see from the following array that the first 12 components account for 88% of the variability in our dataset. So, we will select the number of components to be 12:

```
[27.61619465 14.09297552 8.76254336 8.53355052 7.86265684 7.66539351
  4.08749879 3.39716159 3.25183729 3.02518259 2.35018136 2.08445308
  1.88961942 1.61447355 1.41769812 0.60665283 0.44511272 0.39916116
  0.16620317 0.1273224 ]
```

Let us transform our dataset using PCA with 12 components. Execute the following code block. We get the resulting shape of our dataset as **`(144030,12)`**, which we can use for training:

```
#88% of variance in the first 12 components, so thats what we will
choose
#wine_reduced will become our input train dataset after PCA
pca_12 = PCA(n_components=12)
wine_reduced = pca_12.fit_transform(wine_scaled_df.iloc[:,1:])
wine_reduced.shape
```

And that concludes our discussion on tabular data features and trying out the various feature engineering approaches for the three types of ML use cases we discussed in this chapter.

Conclusion

If you ask a data scientist to select one task that they think is of prime importance in ensuring model accuracy and performance, they will probably select feature engineering, which is what you learned about in this chapter. You may now start appreciating the complexity of the ML landscape and the importance of having a standardized and structure ML workflow to guide us through the myriad of tasks to build an ML solution. In this chapter, we selected three common ML domains, namely, NLP, Computer Vision and Tabular, and learned how to build and run feature engineering tasks for these with step-by-step guides. Technically, the next step in the ML workflow after feature engineering is algorithm selection, but we have to do one more thing before we get there: setting up a continuous data pipeline for feeding in training data for our ML training. We will learn how to build our data pipelines in the next chapter.

Points to Remember

Find below the summary of what we learned in this chapter:

- In this chapter, we learned the importance of features, what is feature engineering, and how we can use it to prepare data for ML training.
- We reviewed the feature engineering techniques applicable for different types of ML use cases and learned how to apply these techniques using sample datasets.
- We started by learning NLP feature engineering tasks and also tried them in action using sentence snippets from different books by performing stemming and lemmatization, tokenization, stop word removals and more.
- We then learned how to perform computer vision feature engineering by taking an image and going through tasks like sizing, rotating, converting to grayscale, and tiling.
- We then pivoted to tabular data feature engineering tasks. We used a wine dataset and applied techniques like scaling, normalization, categorical encoding, frequency encoding, and binning.
- While we walked through various feature selection steps sequentially for the benefit of learning, note that you may only execute some of these steps based on source data quality in a real-world situation.

Multiple Choice Questions

Use these questions to challenge your knowledge of what we learned in this chapter:

1. **Which of these tasks is NOT used for tabular feature engineering?**
    a. Binning
    b. Normalization
    c. Frequency encoding
    d. Stemming
2. **What is feature engineering?**
    a. The process of taking requirement features and engineering them to build applications
    b. The elevation view of a bridge as drawn by an engineer
    c. The process of extracting, selecting, and preparing features for ML training
    d. A key concept in astrology to predict a person’s future
3. **NLP models cannot directly work with text data.**

   a. True
   b. False

4. **What are features and observations in ML?**
   a. An important component of a model that is needed to observe how a model works
   b. The ability to pick selected samples from the training dataset
   c. Randomization techniques for model training
   d. The columns and rows of the training dataset
5. **What is the technique of converting categorical text-based data to numeric features called?**
   a. One-hot encoding
   b. Nominal encoding
   c. Neural topic modeling
   d. Categorical embedding

Answers

1. **d**
2. **c**
3. **a**
4. **d**
5. **a**

Further Reading

Pace, R. Kelley and Ronald Barry, Sparse Spatial Autoregressions, Statistics and Probability Letters, 33 (1997) 291-297.

API design for machine learning software: experiences from the scikit-learn project, *Buitinck* et al., *2013*

BibTex Citation

```
@inproceedings{sklearn_api,
author = {Lars Buitinck and Gilles Louppe and Mathieu Blondel and
Fabian Pedregosa and Andreas Mueller and Olivier Grisel and Vlad
Niculae and Peter Prettenhofer and Alexandre Gramfortand Jaques
Grobler and Robert Layton and Jake VanderPlas andArnaud Joly and
Brian Holt and Ga{\"{e}}l Varoquaux},
title = {{API} design for machine learning software: experiences
from the scikit-learnproject},
booktitle = {ECML PKDD Workshop: Languages for Data Mining and
Machine Learning},
year = {2013},
pages = {108--122},
}
```

CHAPTER 4

Orchestrating the Data Continuum

Introduction

Are you a space aficionado? If yes, you probably already know the basic premise of a continuum. Simply put, it refers to sequential events that evolve continuously, perhaps even infinitely. The space of this universe, in which all the planets and stars appear, has been continuously expanding since time immemorial. Even though we don't realize it, our entire solar system is moving at a speed of 200 KMs per second as the space all around us widens with the expansion of the universe. The known (we are still only scratching the surface on how large the observable universe is) part of our universe is greater than 13.5 billion light years wide (or the distance it takes light to travel in 13.5 billion years, with the speed of light being 30,000 KMs per second), and it only continues to grow. So, space is a continuum.

Time is a continuum too for even though we perceived its existence only a few thousand years ago, it had its beginnings billions of years ago (at least based on what we know today), and it continuously evolves toward infinity (or until there is none of us left to acknowledge the existence of time). Apart from evolution and continuity, a characteristic of a continuum is that it is self-sustaining. That means it does not need or depend on anything else for its existence. Space does not depend on stars or planets for its existence or expansion, and time does not depend on events. Space and time are the dimensions that contain, help qualify and understand the events and objects that appear and exist within them. Without space and time, we cannot experience life the way we perceive it. Similarly, data is the continuum that provides the dimensions that describe and characterize the occurrence of events that an ML model tries to understand and approximate.

In other words, ML models are the engines of continuum evolving in intelligence (improving their prediction accuracy) as they encounter and learn from a steady flow of ever-changing data during training and inference, assimilating approximation patterns between what is and what is needed, and consequently, becoming a data originator in itself cloaked as insights. In other words, predictions of one ML model are a source of data to train a different ML model. The main perspective we are trying to address here is that data is the link that holds the continuum of insights powered by ML models together. Increasingly, and in part due to technological advancements in the last few decades, data continues to be the fabric of our perceived reality influencing us in unforeseen ways.

Targeted advertising is one such example. You search for flights online to a favourite holiday destination that you have been meaning to go for a while now, and surprisingly, you see an ad showing attractions in that same destination when you catch up on your mobile video playlist later in the day. You might think it is providence, but it is not divinity that is watching over you; an ML model is tracking your browsing activity (data) and showing you ads related to your areas of interest. There are examples like this in literally everything we do these days. And data is the fuel that drives the engines behind these ML models, helping the continuum thrive.

In the previous chapters, you learned what the ML workflow is, how to source and ingest data into an Amazon S3 Data Lake (**https://aws.amazon.com/products/storage/data-lake-storage/**), what the importance of features in a dataset is, and how to perform feature engineering tasks for various ML use cases. In this chapter, we will learn how to build our own continuum by orchestrating automated data flows to collect, process, feature engineer and transform data for ML modelling. We will first go more deeper into the role of data in the continuum and understand why data processing automation is important for ML. We will then build our own data orchestration pipeline using AWS Glue ETL jobs (**https://aws.amazon.com/glue**) for the tabular feature engineering use case. Finally, we will learn how to perform data profiling, processing and clean up using AWS Glue DataBrew (**https://aws.amazon.com/glue/features/databrew/**). Let's get started.

Structure

In this chapter, we will dive deep into the following topics:

- Demystifying the data continuum
- Running feature engineering with AWS Glue ETL
- Data Profiling with AWS Glue DataBrew

Objectives

By the end of this chapter, you will have experience building data orchestration flows to process, feature engineer and transform your datasets in preparation for ML training. While *Chapter 2, Hydrating Your Data Lake*, and *Chapter 3, Predicting the Future With Features*, addressed the *what* of data preparation and feature engineering steps in the ML workflow we were introduced to in *Chapter 1, Introducing the ML Workflow*, this chapter addresses *how* to automate these steps, which is a requirement for operationalization of our ML solution.

NOTE: To run the examples in this chapter, you need access to an AWS account. You also need to create an Amazon S3 bucket, onboard to Amazon SageMaker Studio, and run Jupyter notebooks. Refer to the *Technical Requirements* section of *Chapter 3, Predicting the Future With Features*, for instructions on how to meet these pre-requisites.

Demystifying the data continuum

We understand that the data continuum drives the intelligence in ML modelling, but how do we build one for our needs? How do we create a self-sustaining, continuously evolving, automated data ecosystem that we refer to as the continuum? And even if we were to create it, of what use is it beyond ML? Even the greatest idea is only useful if it can be practically applied and if it yields results. So, how do we put the data continuum in practice? We can do this by iteratively building serverless (refers to the ability to run code without needing to manage infrastructure provisioning in cloud environments) data orchestration pipelines from your Amazon S3 data lake, automating data flows to your ML models and downstream applications using a modular hub and spoke design approach. Nothing too complicated, it is what airlines use when they plan their route network. The idea is to minimize the number of point-to-point connections. For example, consider the following image and assume that each node is a city and that the arrows indicate routes:

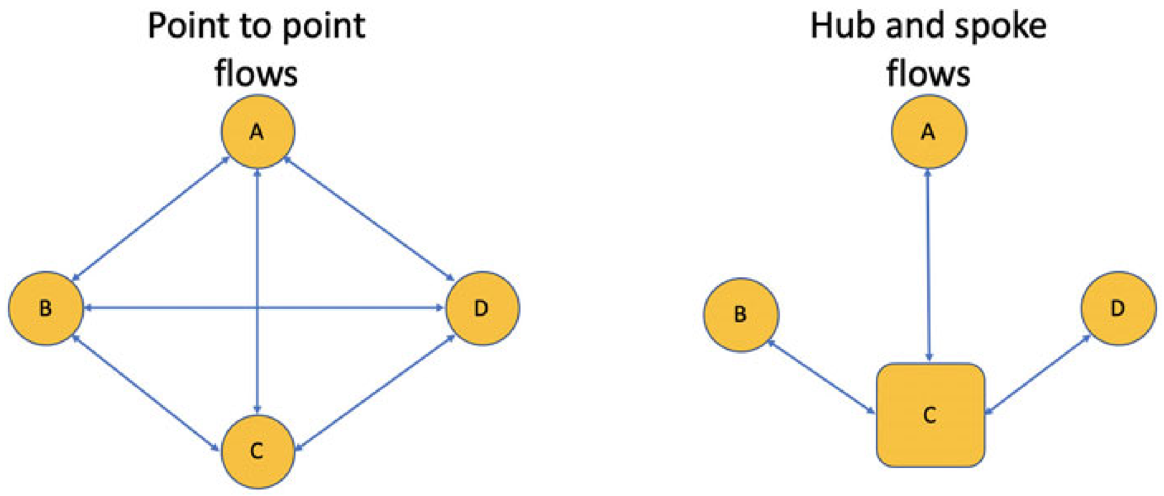


Figure 4.1: *Travel routes between cities*

As you can see, the point-to-point approach has more routes (more overhead, higher costs, and lower efficiency) than the hub and spoke approach. A direct connection may be quicker, but it is not a scalable approach as more cities and routes are added. The same concept applies to data flows between systems. In our case, node C is the Amazon S3 data lake, and nodes A, B and D are either ML models or applications or a combination of both. When we design a data continuum, we iteratively add data orchestrations or flows to/from the data lake, applications, ML models, data warehouses and other operational data stores. We always try and identify the most optimal nodes that can be considered hubs. The data orchestrations (or flows) are fully automated, with options to be executed in scheduled frequencies or in response to events, with an on-demand configuration enabled as well.

For ML models, the data orchestration flows (or spokes of the hub and spoke data continuum) are typically batch for delivered data for model training/re-training and real-time for inference requests, though it can be either way based on the requirements. The architecture defines the modalities of how data is served and consumed, with the keyword being automation. The following table briefly describes the different types of modalities, with some guidance on what to use when:

Interaction type	Interaction mode	Interaction paradigm	Usage
Real-time	Synchronous	Request/Response	For data flows between two systems (A and B), with A requesting B for information by either passing or not passing a parameter in its request, for example, a news website calling a weather webservice passing a city as a parameter
Real-time	Synchronous	Put/Get	A sending data or messages via a queue to B and waiting for a response from B
Real-time	Asynchronous	Publish/Subscribe	A sending messages or data to a topic that multiple consumers have subscribed to
Real-time	Asynchronous	Put/Get	A sending messages or data via a queue but not waiting for a response from B; also called fire and forget
Real-time	Broadcast	Publish/Subscribe	A sending message to a topic that has a large number of subscribers with no expectation of an acknowledgement or a response
Real-time	Fan-out	Publish/Subscribe	As part of a larger flow or orchestration, A performs a task that results in data or

			messages being sent to multiple consumers in parallel
Batch	Asynchronous	Request/Ack/Response	Large volume data is grouped into batches, processed and transferred from A to B, with B sending an acknowledgement to indicate that it has started data receipt and sends a response once data has been received and processed
Batch	Asynchronous	Push/Pull	A pushes a large volume of data to B based on a scheduled frequency or a trigger, or B pulls data from A again based on a frequency, trigger or on-demand

Table 4.1: *Design modalities for data flow orchestrations*

When designing a data continuum for your enterprise, you will use one or more of these modalities for your data orchestration flows. A word of caution though: do not try and bite off more than you can chew. Just like in ML modelling, you should follow an iterative approach here. First determine, the hub nodes for your continuum. You can consider this as a clustering or nearest neighbours problem (we will talk about ML algorithms in detail in *Chapter 5, Casting a Wider Net*). Your hub nodes are akin to your cluster centres, and they typically represent data stores or data lakes to/from which data orchestration flows consume or deliver data. We can, in fact, also model this a graph data (**https://aws.amazon.com/neptune/**) problem and use a graph neural network (**https://docs.aws.amazon.com/sagemaker/latest/dg/deep-graph-library.html**) to approach the data continuum. This goes to show that even the design elements of our data continuum can be represented as data and modelled using ML and validates the principle of using a continuum as a design paradigm.

AWS offers a lot of flexibility and choices in designing serverless data orchestration pipelines. We have room in this chapter to only cover two such options, but we will read about alternative approaches to building data orchestration for ML, along with links to instructions, in the Conclusion section of this chapter.

In this section, we explored the concept of a data continuum deeply and understood that it comprises data orchestration flows between participating systems that can be designed in various approaches based on requirements.

In the next section, we will use AWS Glue ETL (**https://docs.aws.amazon.com/glue/latest/dg/author-job.html**), a serverless, managed analytics service to create an automated job for atomically running the tabular feature engineering tasks on input data and creating an output dataset with features for ML training.

Running feature engineering with AWS Glue ETL

In this section, we will learn how to automate the feature engineering tasks we learned and executed in Chapter 3, Predicting the Future With Features, using AWS Analytics services. When we use Jupyter notebooks to run our code, we are in the experimentation phase. Notebooks provide a convenient way for us to build and test each line of code until we are sure that all the steps are working as intended. But when we get into development and production, we need to automate the execution of feature engineering tasks so that we have a fully functional data pipeline for ingestion, feature engineering, transformation and storage for ML training requirements. That is what we will build in this section. The architecture for our solution build is shown in the following image:

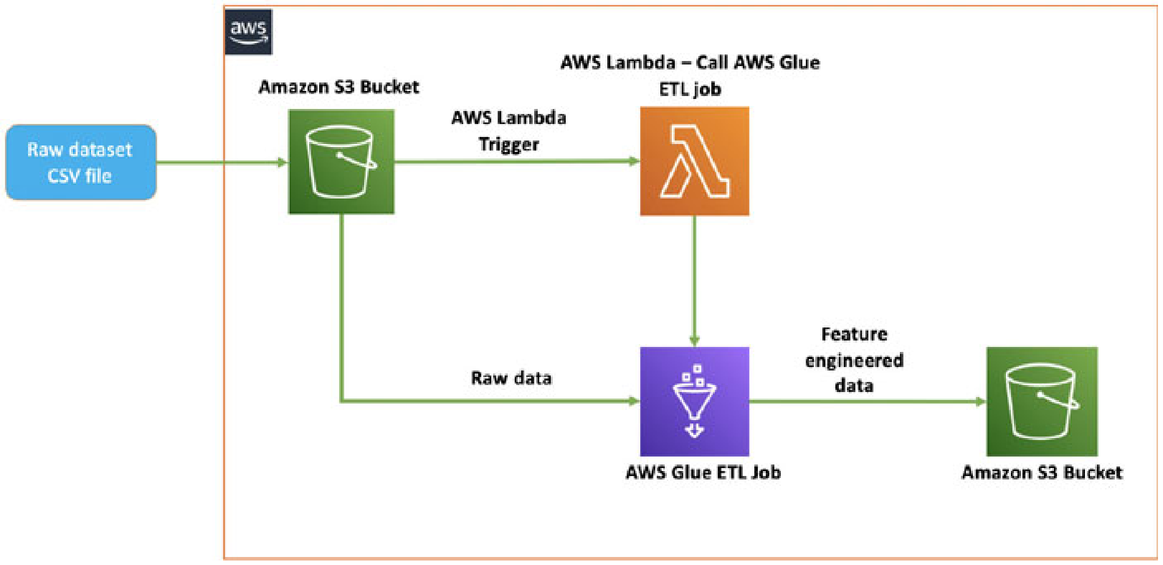


Figure 4.2: *Feature engineering using AWS Glue ETL*

As you can see, there are three major components to our build. We will build these pieces using the AWS management console, Amazon SageMaker Studio notebooks, and AWS Lambda, along with code snippets from our GitHub repository. Our build will consist of the following tasks that we will cover in independent sections subsequently for ease of understanding:

- Building an AWS Glue ETL job with the feature engineering code
- Creating an AWS Lambda function with an Amazon S3 trigger

- Triggering the AWS Glue ETL job from the AWS Lambda function when the raw dataset is uploaded to the Amazon S3 bucket

In this section we reviewed the solution architecture for building a feature engineering data orchestration flow using AWS Glue ETL. Our solution is similar to a batch push/pull approach with the raw data upload triggering a processing of the data and being pushed to a target S3 bucket location. Follow the instructions in the next section for the build tasks.

Building the AWS Glue ETL job

AWS Glue (**https://aws.amazon.com/glue**) is a fully managed, serverless, secure, and scalable data orchestration service that makes it possible to build a myriad of automated and elastic integration pipelines across diverse data sources, both on-premises and in the AWS cloud. AWS Glue is one of 14 (as of March 2022) analytics services available for data integration and processing, in addition to 20+ services available for data storage and migration. The following image shows a snapshot of these services available in the AWS Management Console:

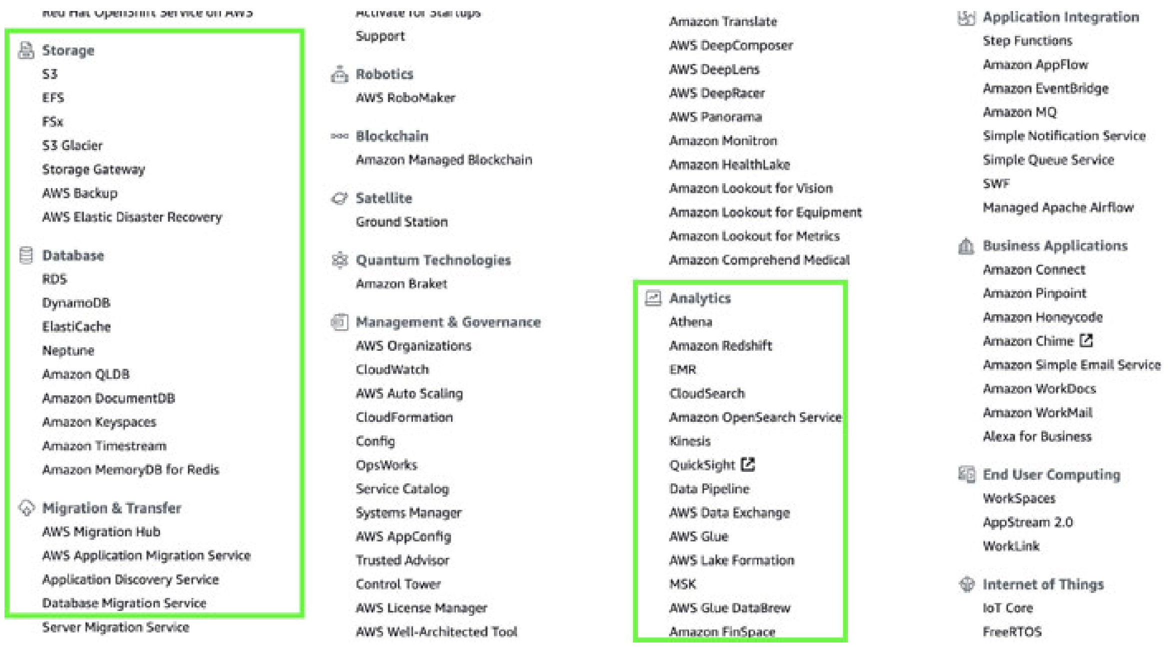


Figure 4.3: *AWS Storage, Database and Analytics services*

So, there are multiple options of how we can build data orchestration flows between various data stores, applications and ML solutions. Going into each of them will take up the whole book. To get you prepared for ML training, we will focus on using AWS Glue ETL jobs for data orchestration and AWS Glue DataBrew for data profiling and orchestration in this chapter. For now, we will set up the Glue job as a first step, as shown in the following image, from our solution architecture:

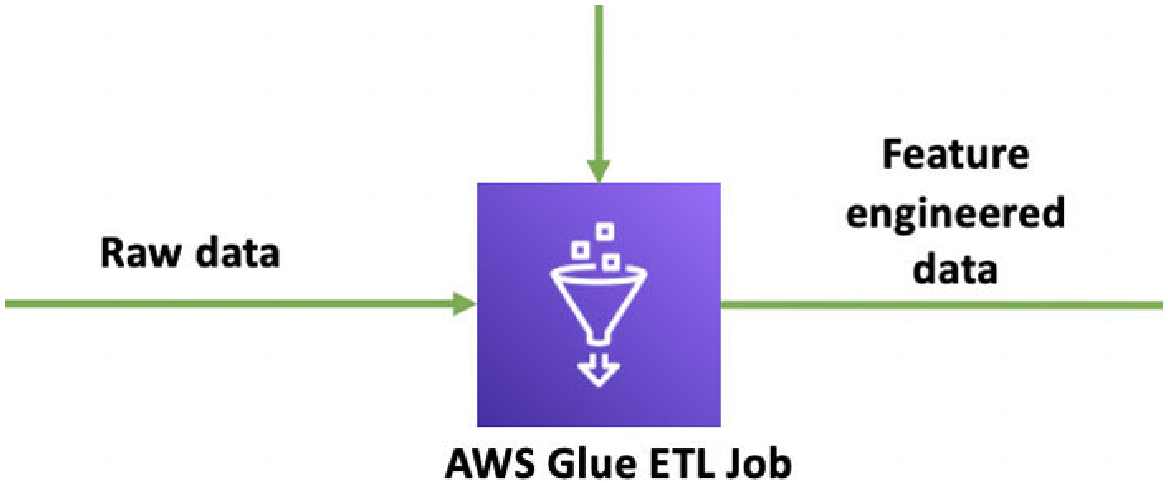


Figure 4.4: *AWS Glue ETL*

Follow the instructions in the *Technical Requirements* section to ensure that you have signed up for an AWS account and are logged in (**https://portal.aws.amazon.com/billing/signup**). Execute the following steps to complete this section:

1. You first need to create an Amazon S3 bucket (**https://docs.aws.amazon.com/AmazonS3/latest/userguide/create-bucket-overview.html**) and note down its name.
2. Next, follow the instructions in the *Technical Requirements* section in *Chapter 3, Predicting the Future With Features*, to onboard to an Amazon SageMaker Studio domain, clone the book's GitHub repository given at the beginning of the book, and click on Chapter-04 to open it up.
3. Open the SageMaker Studio notebook titled `glue-script-loader.ipynb` to execute the cells in it step by step.

   NOTE: Do not execute the cells under the heading starting with IMPORTANT in the notebook right now. We will execute this cell later.

4. Provide the name of your S3 bucket, as shown in the following code snippet, in the notebook cell and execute the cell either by pressing *Shift* + *Enter* or by clicking on the triangular play button at the top:

```
# import the boto3 python SDK for AWS
```

```
import boto3
# declare the S3 handle to create prefixes and load our files
s3 = boto3.client('s3')
# declare variables
bucket = '<your-bucket-name>'
prefix = 'aiml-book/chapter4/'
```

5. Execute the next cell to upload the Python wheel files (external libraries that we need to run our solution) to the S3 bucket. When the cell is executed, it will print the S3 locations of the wheel files; copy the print statement and save it. We will need this when we build the Glue ETL job:

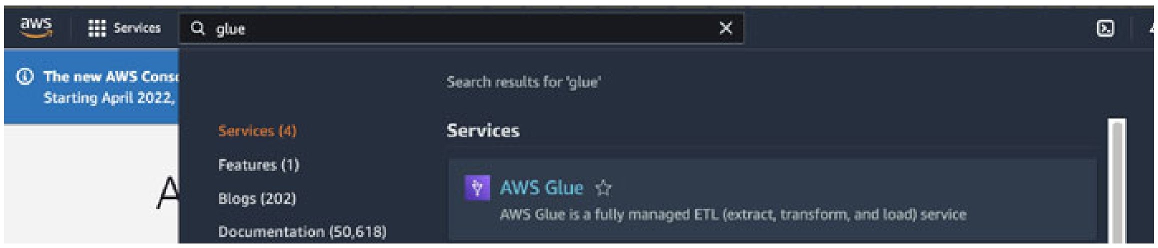
```
# upload the wheel files required by Glue to S3 bucket
ce_wheel = 'category_encoders-2.4.0-py2.py3-none-any.whl'
wr_wheel = 'awswrangler-2.14.0-py3-none-any.whl'
s3.upload_file(ce_wheel,bucket,prefix+ce_wheel)
s3.upload_file(wr_wheel,bucket,prefix+wr_wheel)
print("please copy these s3 file locations for AWS Glue job
creation")
print("Libraries - Python library path: " +
"s3://"+bucket+'/'+prefix+wr_wheel)
print("Libraries - Referenced files path: " +
"s3://"+bucket+'/'+prefix+ce_wheel)
```

We get the following output:

```
please copy these s3 file locations for AWS Glue job creation
Libraries - Python library path: s3://<bucket>/aiml-
book/chapter4/awswrangler-2.14.0-py3-none-any.whl
Libraries - Referenced files path: s3://<bucket>/aiml-
book/chapter4/category_encoders-2.4.0-py2.py3-none-any.whl
```

6. You can exit this notebook now, but leave it open as we will need it in a little while. As a next step, navigate to the book's GitHub repository () in your web browser and open the folder **`Chapter-04`**:

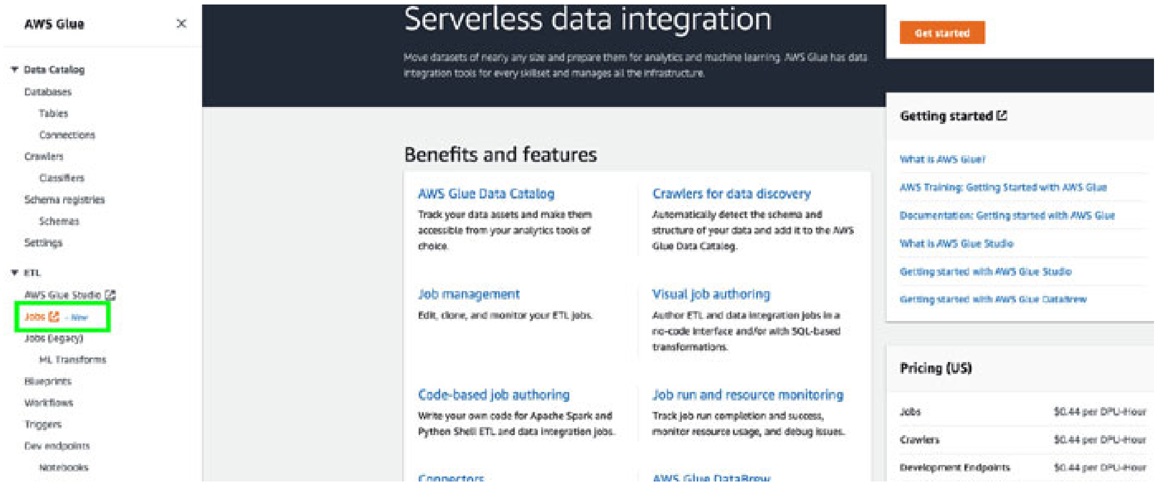
   Open the file named **`tabular-features-glue-etl.py`** and copy its contents. This is the code script for the AWS Glue ETL job that we will build now. Go to your **`AWS Management Console`**, type **`Glue`** in the **`Services`** search bar at the top and select **`AWS Glue`** to go be navigated to the Glue console. Refer to *Figure 4.5*:



AWS Glue is a fully managed ETL (extract, transform, and load) service

Figure 4.5: *Navigate to the AWS Glue console*

7. Select **`Jobs`** under ETL in the left pane of the Glue console, as shown in the following image:



Move datasets of nearly any size and prepare them for analytics and machine learning. AWS Glue has data integration tools for every skillset and manages all the infrastructure.

AWS Glue Data Catalog
Track your data assets and make them accessible from your analytics tools of choice.

Crawlers for data discovery
Automatically detect the schema and structure of your data and add it to the AWS Glue Data Catalog.

Job management
Edit, clone, and monitor your ETL jobs.

Visual job authoring
Author ETL and data integration jobs in a no-code interface and/or with SQL-based transformations.

Code-based job authoring
Write your own code for Apache Spark and Python Shell ETL and data integration jobs.

Job run and resource monitoring
Track job run completion and success, monitor resource usage, and debug issues.

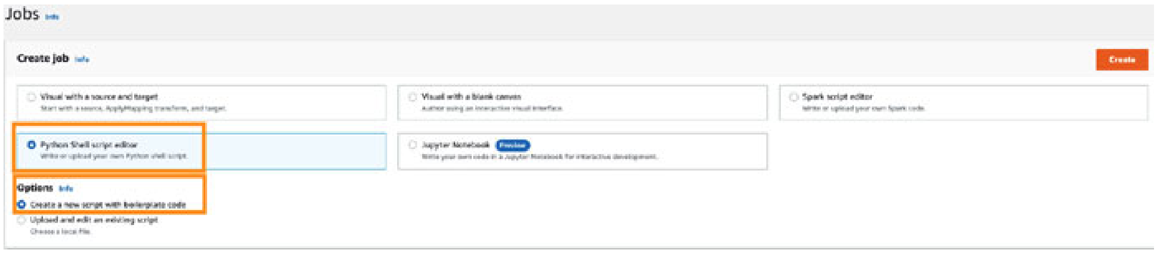
Connectors

AWS Glue DataBrew

Jobs	$0.44 per DPU-Hour
Crawlers	$0.44 per DPU-Hour
Development Endpoints	$0.44 per DPU-Hour

Figure 4.6: *Select Jobs*

8. This opens up the AWS Glue Studio console to author and run ETL jobs. Under **`Create job`**, select **`Python Shell script editor`**, and under **`Options`**, select **`Create a new script with boilerplate code`**. Click on the **`Create`** button in the top-right corner, as shown in the following image:



Visual with a source and target
Start with a source, ApplyMapping transform, and target.

Visual with a blank canvas
Author using an interactive visual interface.

Spark script editor
Write or upload your own Spark code.

[✓ Python Shell script editor]
Write or upload your own Python shell script.

Jupyter Notebook
Write your own code in a Jupyter Notebook for interactive development.

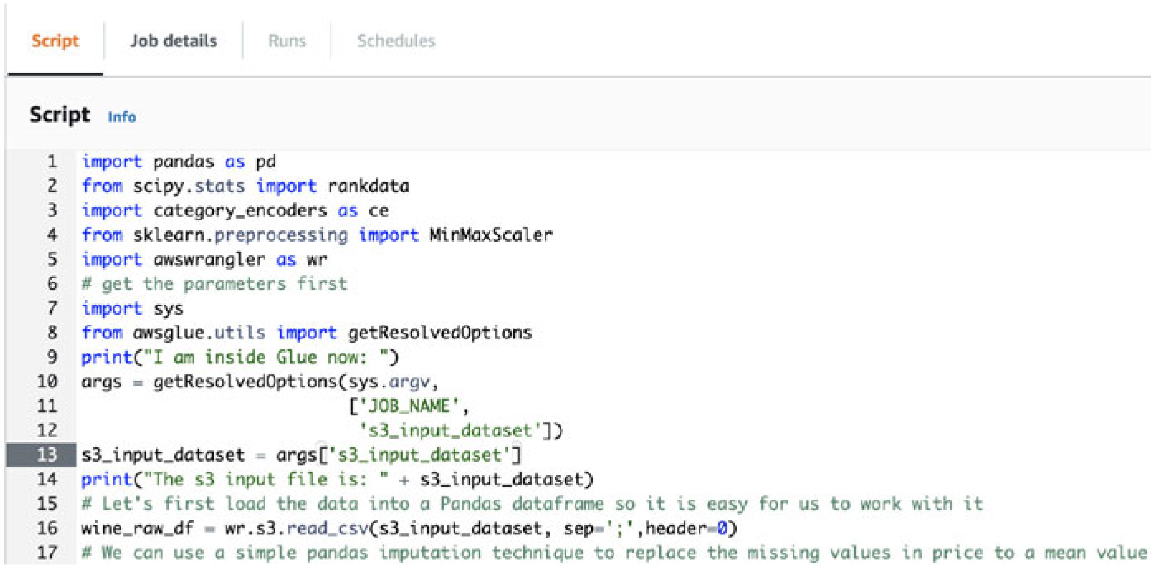
[✓ Create a new script with boilerplate code]

Upload and edit an existing script
Choose a local file.

Figure 4.7: *Create Python Shell job*

9. In the window that opens up, in the **`Script`** tab, paste the contents of the **`tabular-features-glue-etl.py`** file you copied a few steps ago in this section, as shown in the following image. This code snippet contains all the

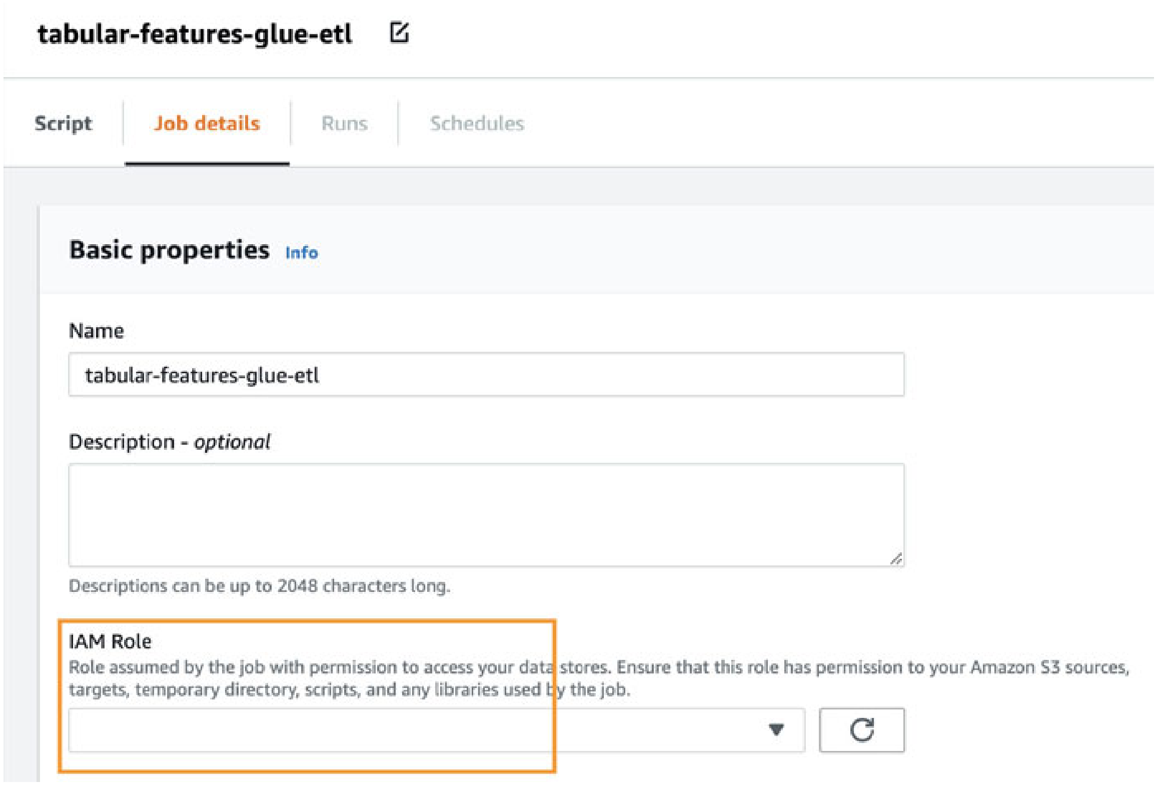
steps you executed in the tabular data feature engineering SageMaker Studio notebook you ran in Chapter 3, Predicting the Future With Features:



```
import pandas as pd
from scipy.stats import rankdata
import category_encoders as ce
from sklearn.preprocessing import MinMaxScaler
import awswrangler as wr
# get the parameters first
import sys
from awsglue.utils import getResolvedOptions
print("I am inside Glue now: ")
args = getResolvedOptions(sys.argv,
                          ['JOB_NAME',
                           's3_input_dataset'])
s3_input_dataset = args['s3_input_dataset']
print("The s3 input file is: " + s3_input_dataset)
# Let's first load the data into a Pandas dataframe so it is easy for us to work with it
wine_raw_df = wr.s3.read_csv(s3_input_dataset, sep=';',header=0)
# We can use a simple pandas imputation technique to replace the missing values in price to a mean value
```

Figure 4.8: *Paste the Python Shell script for tabular feature engineering onto the Script editor*

Now, click on the **`Job details`** tab and provide the name for this ETL job as **`tabular-features-glue-etl`**. We also need to provide an IAM role here that we will use to run the ETL job. As we do not already have this IAM role, we will be creating one. Refer to *Figure 4.9*:



Name

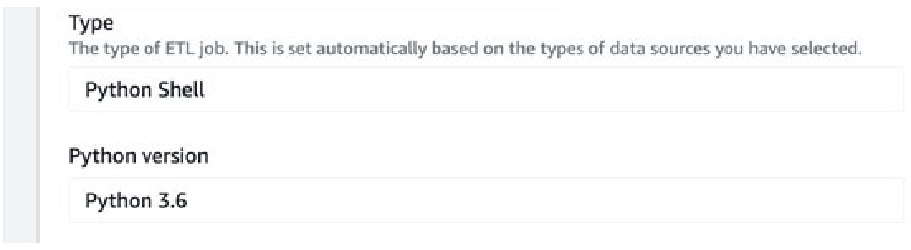
tabular-features-glue-etl

Description - *optional*

Descriptions can be up to 2048 characters long.

IAM Role

Role assumed by the job with permission to access your data stores. Ensure that this role has permission to your Amazon S3 sources, targets, temporary directory, scripts, and any libraries used by the job.

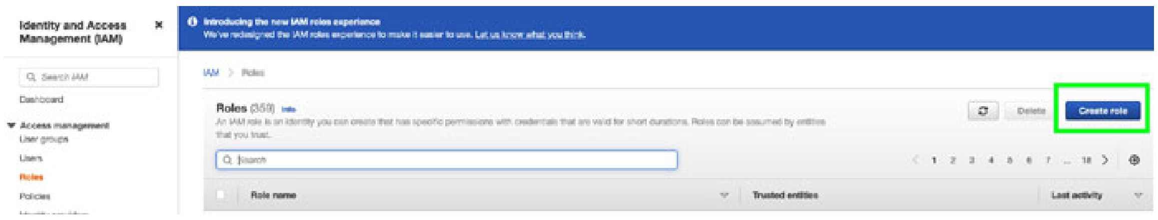


Type
The type of ETL job. This is set automatically based on the types of data sources you have selected.
Python Shell

Python version
Python 3.6

Figure 4.9: *Enter job details*

In the **`Services`** search bar at the top, type **`IAM`** and select **`IAM`**; right-click and open it in a new tab to be navigated to the AWS IAM console. Click on **`Roles`** in the left pane under **`Access management`** and click on the **`Create role`** button on the right. Refer to *Figure 4.10*:

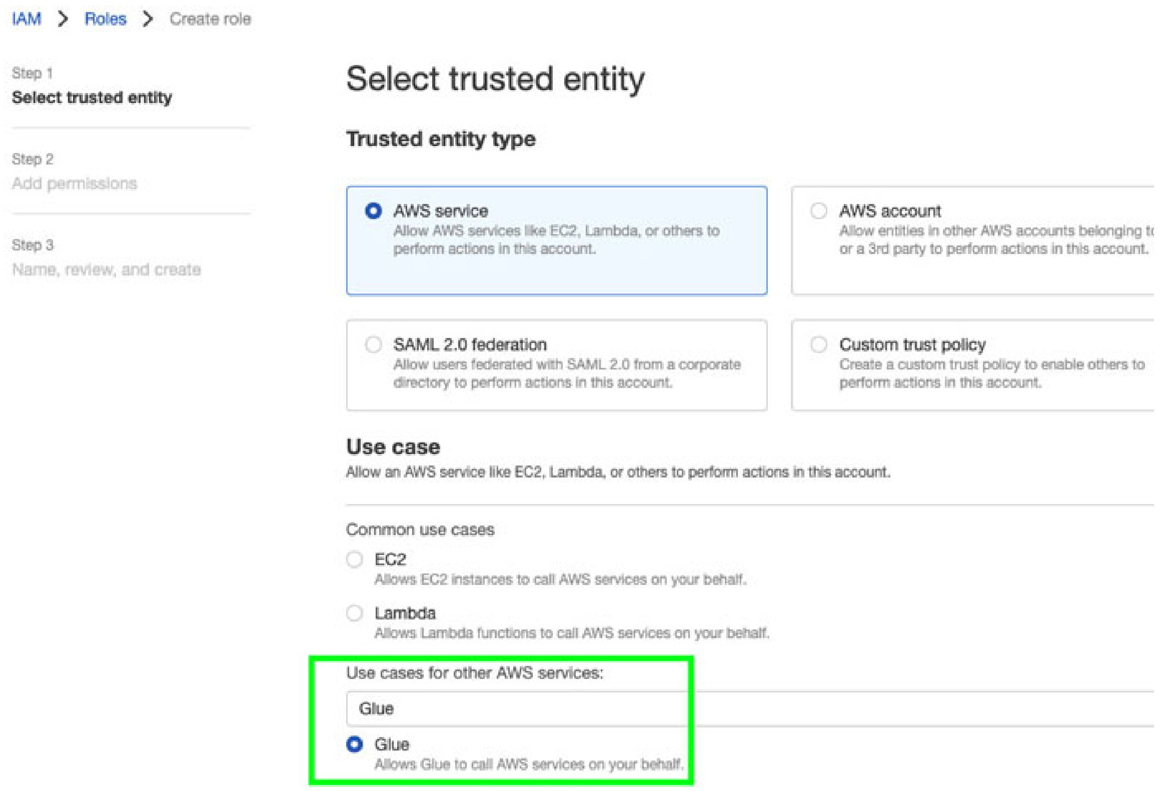


Introducing the new IAM roles experience
We've redesigned the IAM roles experience to make it easier to use. Let us know what you think.

Roles (359) Info
An IAM role is an identity you can create that has specific permissions with credentials that are valid for short durations. Roles can be assumed by entities that you trust.

Figure 4.10: *Create an IAM role*

On the **`Select trusted entity`** page, scroll down under Use case and then to Use cases for other AWS services, and click on the list box with the tip **`Choose a service to view use case`**. Select **`Glue`** from the list and click on the radio button next to Glue, as shown in the following image, and then click on **`Next`**:



IAM > Roles > Create role

Step 1
Select trusted entity

Step 2
Add permissions

Step 3
Name, review, and create

Select trusted entity

Trusted entity type

(●) AWS service
Allow AWS services like EC2, Lambda, or others to perform actions in this account.

( ) AWS account
Allow entities in other AWS accounts belonging t
or a 3rd party to perform actions in this account.

( ) SAML 2.0 federation
Allow users federated with SAML 2.0 from a corporate directory to perform actions in this account.

( ) Custom trust policy
Create a custom trust policy to enable others to perform actions in this account.

Use case
Allow an AWS service like EC2, Lambda, or others to perform actions in this account.

Common use cases

( ) EC2
Allows EC2 instances to call AWS services on your behalf.

( ) Lambda
Allows Lambda functions to call AWS services on your behalf.

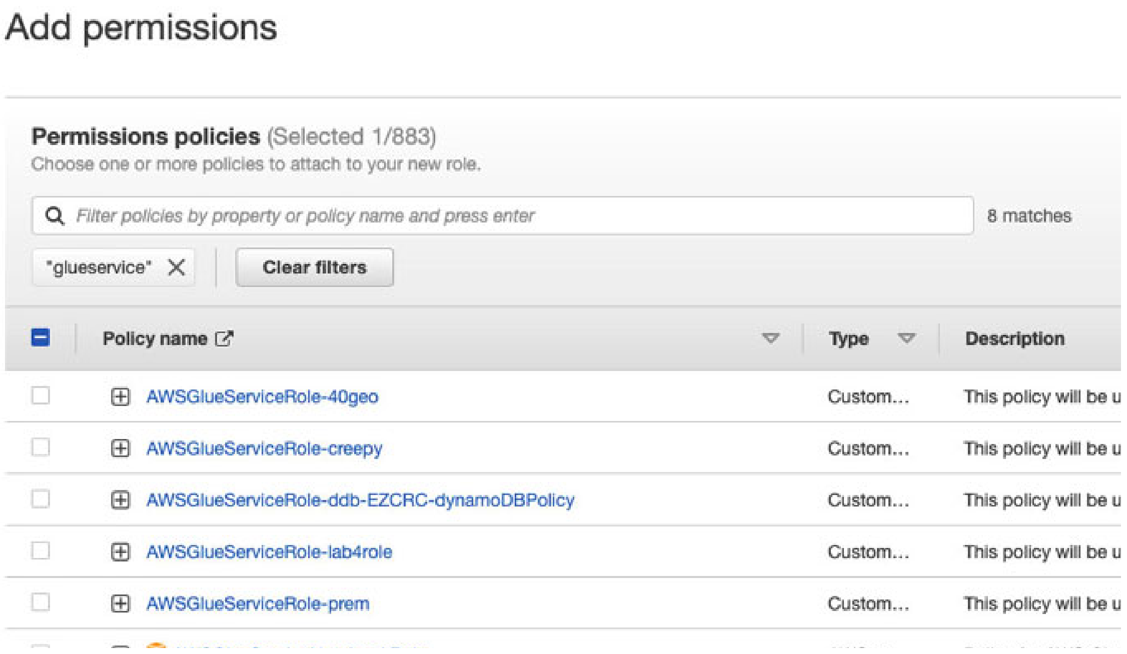
Use cases for other AWS services:

Glue

(●) Glue
Allows Glue to call AWS services on your behalf.

Figure 4.11: *Select trusted entity*

10. In the **`Add permissions`** page, type **`glueservice`** in the search bar and select **`AWSGlueServiceRole`**, as shown in the following image:



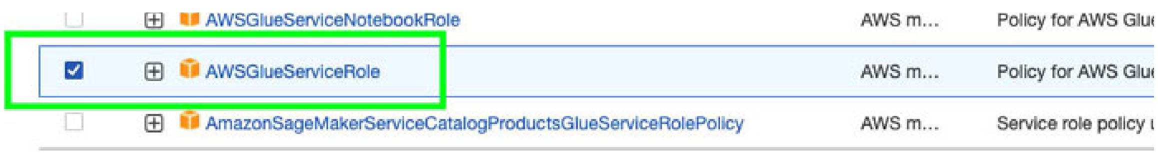
Add permissions

Permissions policies (Selected 1/883)
Choose one or more policies to attach to your new role.

Filter policies by property or policy name and press enter — 8 matches

"glueservice" ✕ — Clear filters

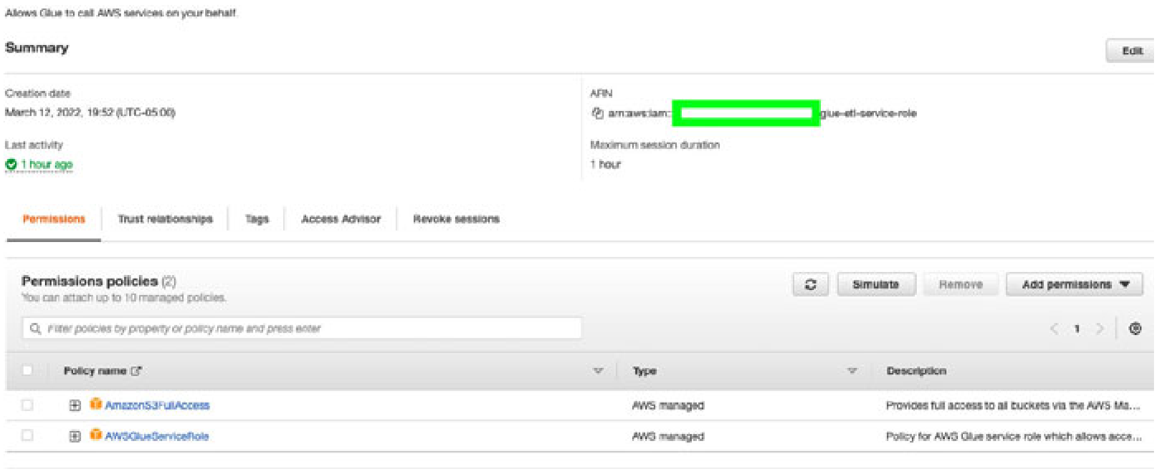
	Policy name	Type	Description
☐	AWSGlueServiceRole-40geo	Custom...	This policy will be u
☐	AWSGlueServiceRole-creepy	Custom...	This policy will be u
☐	AWSGlueServiceRole-ddb-EZCRC-dynamoDBPolicy	Custom...	This policy will be u
☐	AWSGlueServiceRole-lab4role	Custom...	This policy will be u
☐	AWSGlueServiceRole-prem	Custom...	This policy will be u



☐	AWSGlueServiceNotebookRole	AWS m...	Policy for AWS Glu
☑	AWSGlueServiceRole	AWS m...	Policy for AWS Glu
☐	AmazonSageMakerServiceCatalogProductsGlueServiceRolePolicy	AWS m...	Service role policy

Figure 4.12: Attach Glue Service permissions

11. Now, click on **`Clear filters`** under the policy search bar, type **`S3`**, and press *Enter* to bring up the S3 permissions. Scroll down, select **`AmazonS3FullAccess`** and click on **`Next`** in the bottom-right corner. In the **`Role details`** page, type a name for the role, such as **`glue-etl-role`**, and make note of the name you provide. Scroll down the page and click on **`Create role`**. Now, type the role name in the **`Roles`** search bar and review the permissions, as shown in the following image:



Allows Glue to call AWS services on your behalf

Summary

Creation date
March 12, 2022, 19:52 (UTC-05:00)

Last activity
1 hour ago

ARN
arn:aws:iam:[redacted]glue-etl-service-role

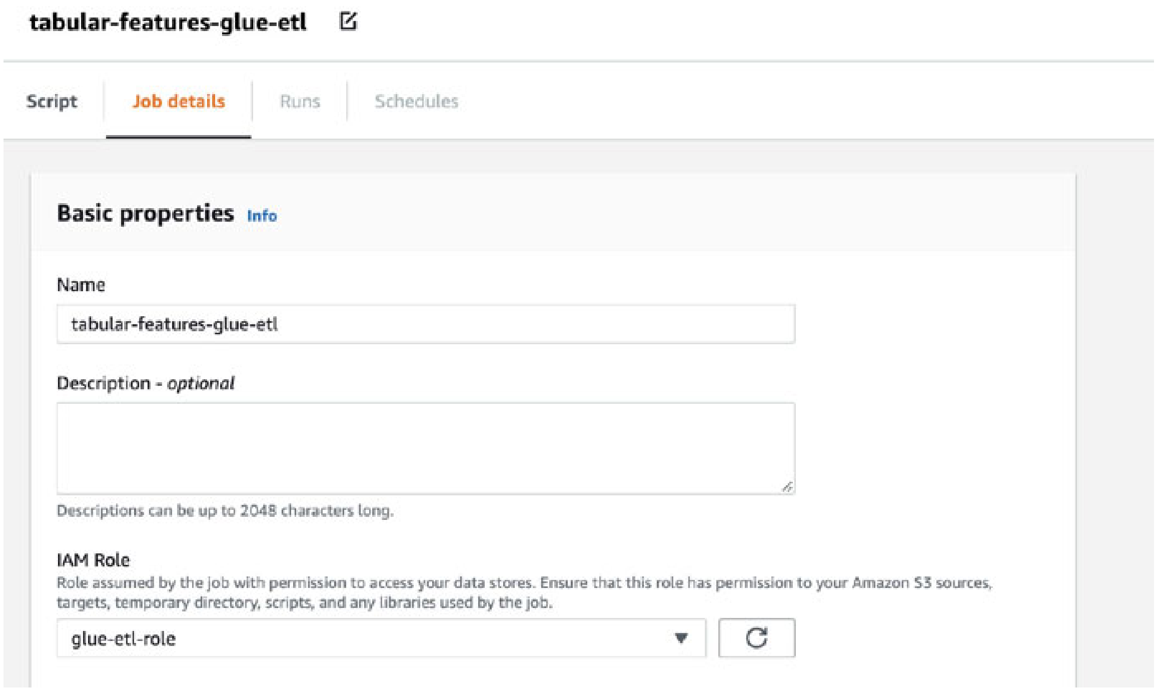
Maximum session duration
1 hour

Permissions policies (2)
You can attach up to 10 managed policies.

Policy name	Type	Description
AmazonS3FullAccess	AWS managed	Provides full access to all buckets via the AWS Ma...
AWSGlueServiceRole	AWS managed	Policy for AWS Glue service role which allows acce...

Figure 4.13: Permissions policies added to Glue role

12. Now go back to the Glue Studio console and provide the name of the IAM role you created just now in the IAM role list box on the **`Job details`** page for your Glue ETL job, as shown in the following image. If you do not see your newly created role appearing, just click on the **`Refresh`** button next to the role list box. Refer to *Figure 4.14*:



Name
tabular-features-glue-etl

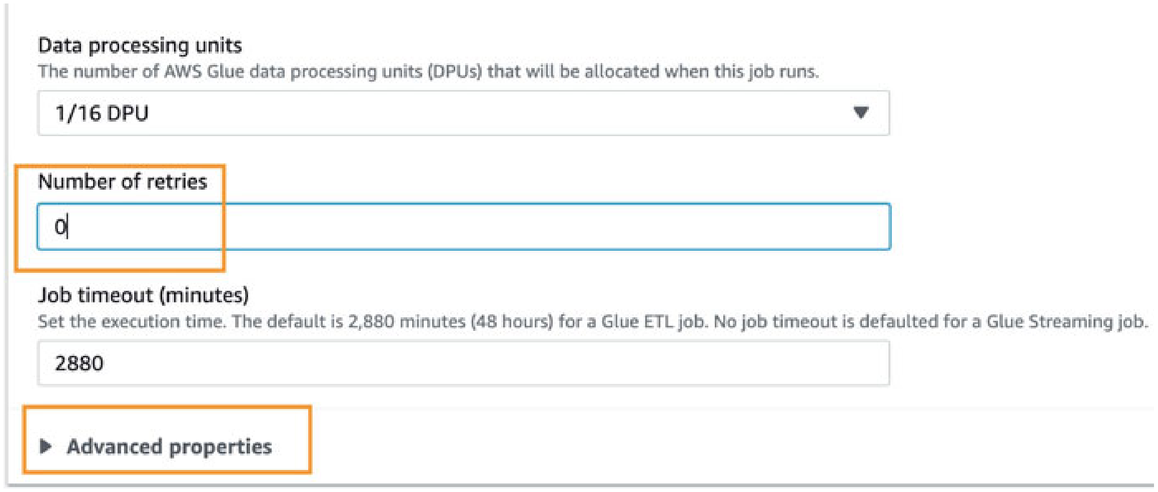
Description - *optional*

Descriptions can be up to 2048 characters long.

IAM Role
Role assumed by the job with permission to access your data stores. Ensure that this role has permission to your Amazon S3 sources, targets, temporary directory, scripts, and any libraries used by the job.
glue-etl-role

Figure 4.14: *Provide the IAM role in Job details*

13. Scroll down the **`Job details`** page and change the **`Number of retries`** to **`0`**. Click on **`Advanced properties`** at the bottom of the page, as shown in the following image:



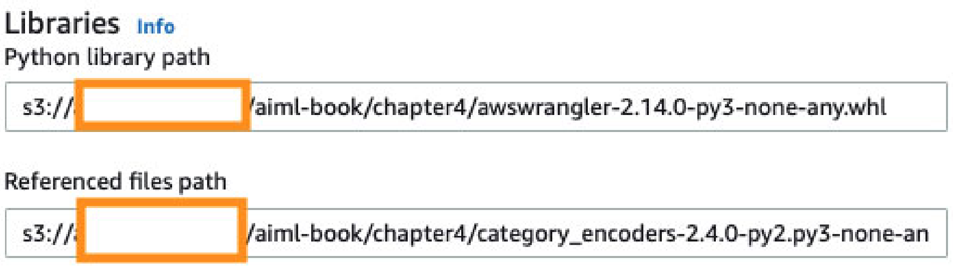
Data processing units
The number of AWS Glue data processing units (DPUs) that will be allocated when this job runs.
1/16 DPU

Number of retries
0

Job timeout (minutes)
Set the execution time. The default is 2,880 minutes (48 hours) for a Glue ETL job. No job timeout is defaulted for a Glue Streaming job.
2880

Figure 4.15: *Change job configuration and scroll to Advanced properties*

14. Keep scrolling all the way down the **`Advanced properties`** without changing any of the default settings. Go to the **`Libraries`** section. Paste the S3 URI for the Python library path that you copied from the cell print in the SageMaker Studio notebook, **`glue-script-loader.ipynb`**, in the input field under Python library path. Paste the S3 URI for the **`Referenced files path`** that you copied from the SageMaker Studio notebook in the input field under **`Referenced files path`**, as shown in the following image:



Libraries Info

Python library path

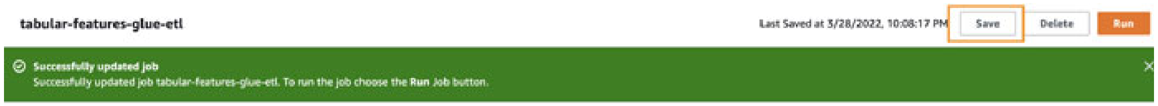
s3://[redacted]/aiml-book/chapter4/awswrangler-2.14.0-py3-none-any.whl

Referenced files path

s3://[redacted]/aiml-book/chapter4/category_encoders-2.4.0-py2.py3-none-an

Figure 4.16: *Add library files to the Glue job*

Now scroll back up all the way to the top of the page and click on the **`Save`** button in the top-right corner. Refer to *[Figure 4.17](#)*:



tabular-features-glue-etl

Last Saved at 3/28/2022, 10:08:17 PM

Successfully updated job

Successfully updated job tabular-features-glue-etl. To run the job choose the Run Job button.

Figure 4.17: *Glue job saved*

15. Do not click on the **`Run`** button as you will get errors. The script expects parameters that will be sent by the AWS Lambda function that we will build in the next section.

Creating AWS Lambda function with an Amazon S3 trigger

In this section, we will execute the tasks to create our AWS Lambda (**https://aws.amazon.com/lambda/**) function. AWS Lambda allows you to easily build and execute your code in response to events. You don't have to create or provision any infrastructure. All you have to do is write your code in one of the programming languages (**https://aws.amazon.com/lambda/faqs**) that AWS Lambda supports and upload it by creating a function in the AWS Lambda console. You can either attach a trigger (**https://docs.aws.amazon.com/lambda/latest/dg/lambda-invocation.html**) for the code to be executed as a response, you can directly invoke it using https, or you can put it behind an API using the Amazon API Gateway (**https://aws.amazon.com/api-gateway/**) service.

We will choose the Amazon S3 create trigger for our Lambda function. Whenever we upload a new file or upload an updated version of the same file to the S3 bucket, the Lambda function will be triggered. This Lambda function will call the AWS Glue ETL job we set up previously, as shown in the following image of our solution architecture.

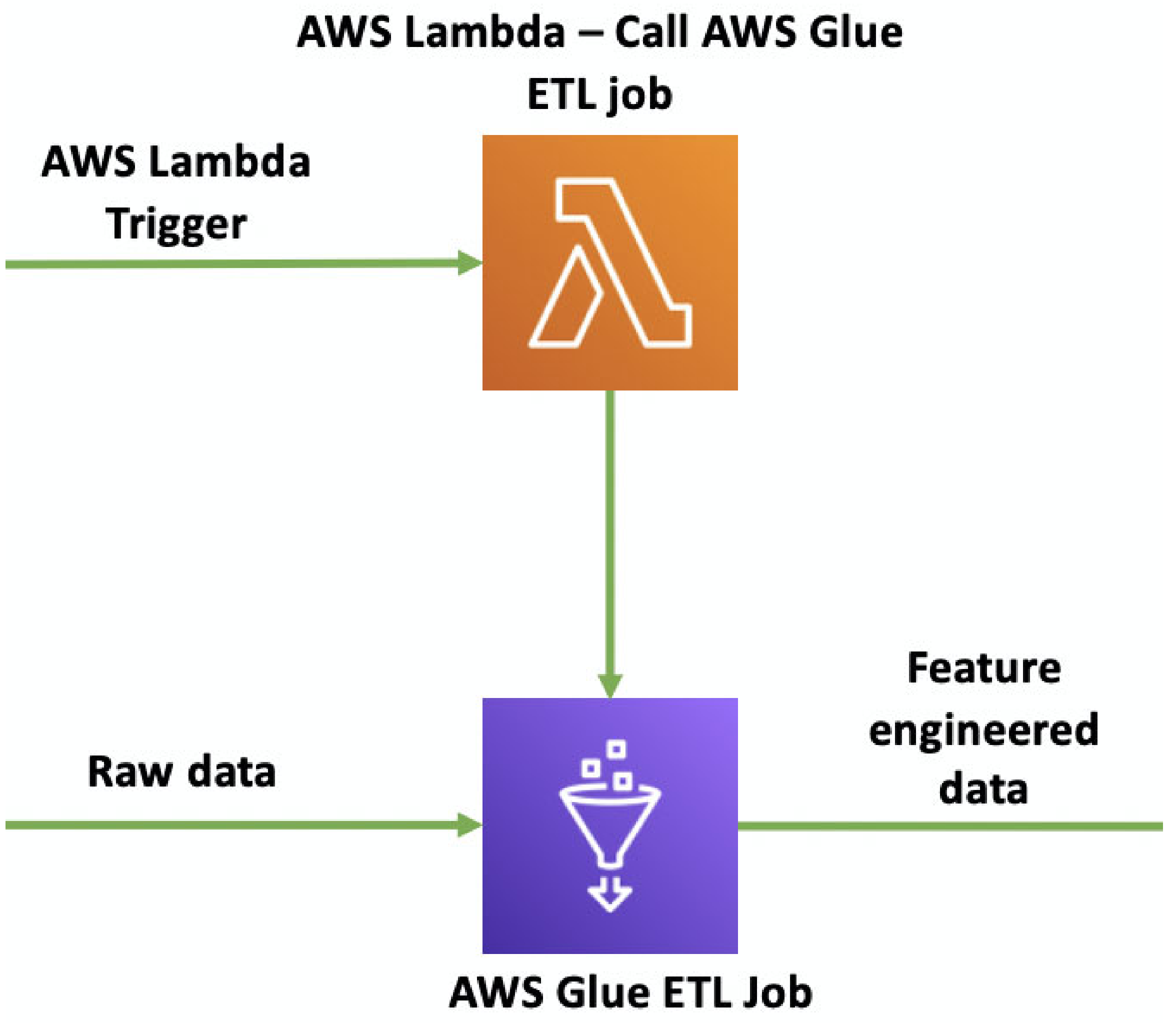
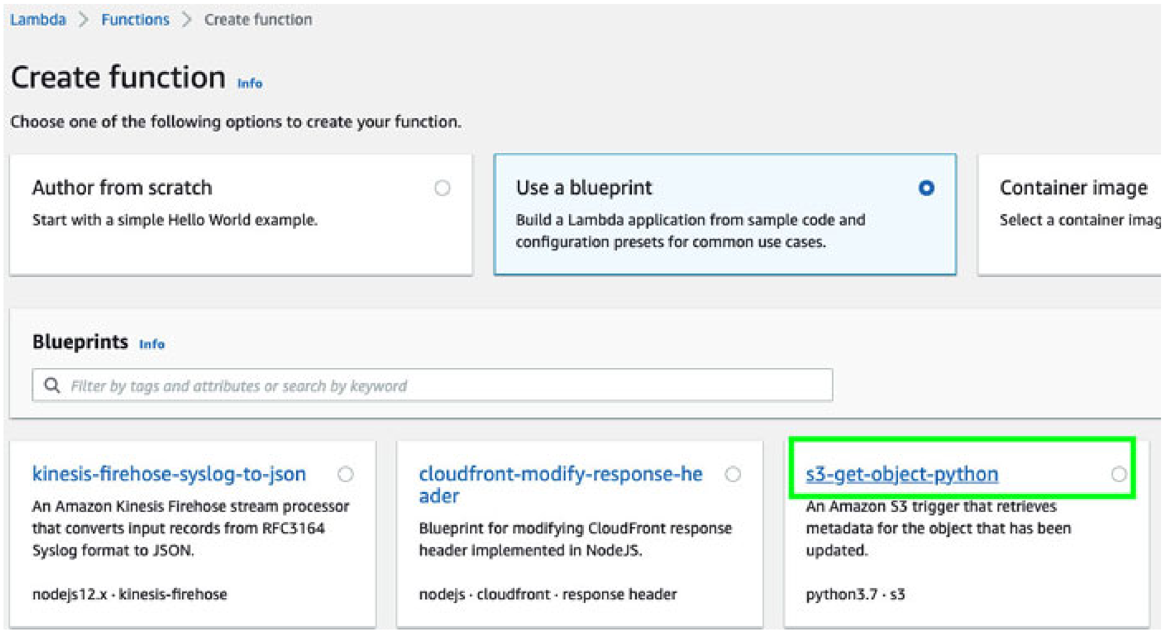


Figure 4.18: *AWS Lambda function triggered from S3 to call AWS Glue ETL job*

As a first step, we will use an existing blueprint to create an Amazon S3 trigger-based AWS Lambda function. In the AWS Management Console, type **`Lambda`** in the services search bar at the top of the page and click on **`AWS Lambda`** to be navigated to the Lambda console. Click on **`Functions`** either by expanding the menu on the left or by clicking on the **`Create function`** button in the top-right corner of the page. On the **`Create function`** page, click on **`Use a blueprint`**, select **`s3-get-object-python`** and click on the **`Configure`** button. Refer to *Figure 4.19*:


Lambda > Functions > Create function

Create function Info

Choose one of the following options to create your function.

Author from scratch
Start with a simple Hello World example.

Use a blueprint
Build a Lambda application from sample code and configuration presets for common use cases.

Container image
Select a container imag

Blueprints Info

Filter by tags and attributes or search by keyword

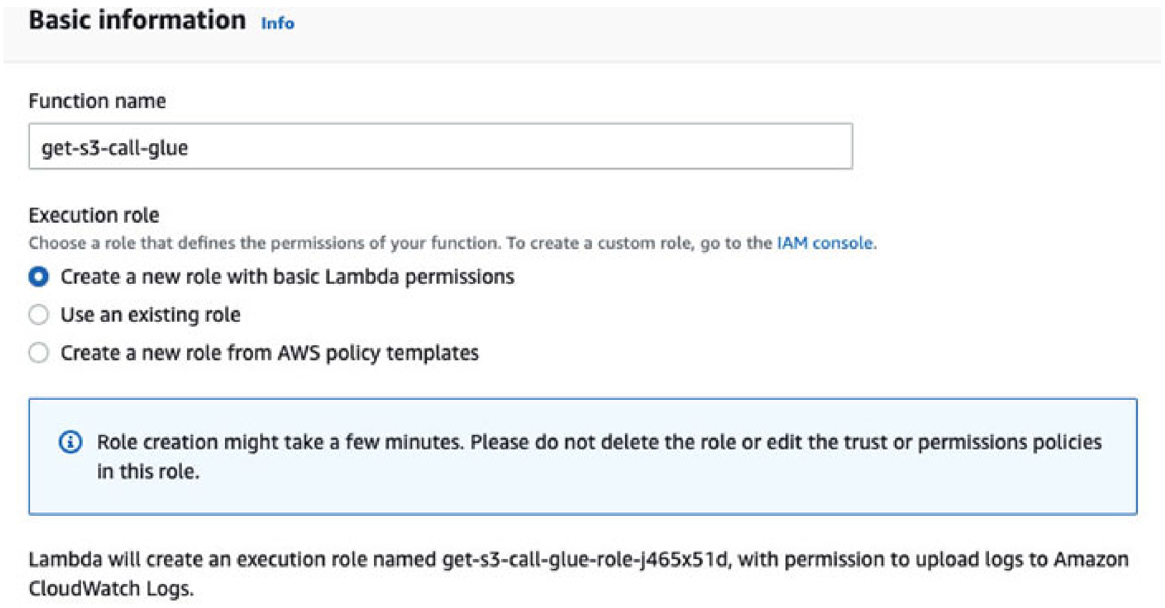
kinesis-firehose-syslog-to-json
An Amazon Kinesis Firehose stream processor that converts input records from RFC3164 Syslog format to JSON.
nodejs12.x · kinesis-firehose

cloudfront-modify-response-header
Blueprint for modifying CloudFront response header implemented in NodeJS.
nodejs · cloudfront · response header

s3-get-object-python
An Amazon S3 trigger that retrieves metadata for the object that has been updated.
python3.7 · s3

Figure 4.19: *Create a Lambda function*

Provide a **`Function name`** and select **`Create a new role with basic Lambda permissions`** under **`Execution role`**. This will allow Lambda to automatically create a new IAM role for this function. Refer to *Figure 4.20*:



Basic information Info

Function name

get-s3-call-glue

Execution role

Choose a role that defines the permissions of your function. To create a custom role, go to the IAM console.

[✓ Create a new role with basic Lambda permissions]

Use an existing role

Create a new role from AWS policy templates

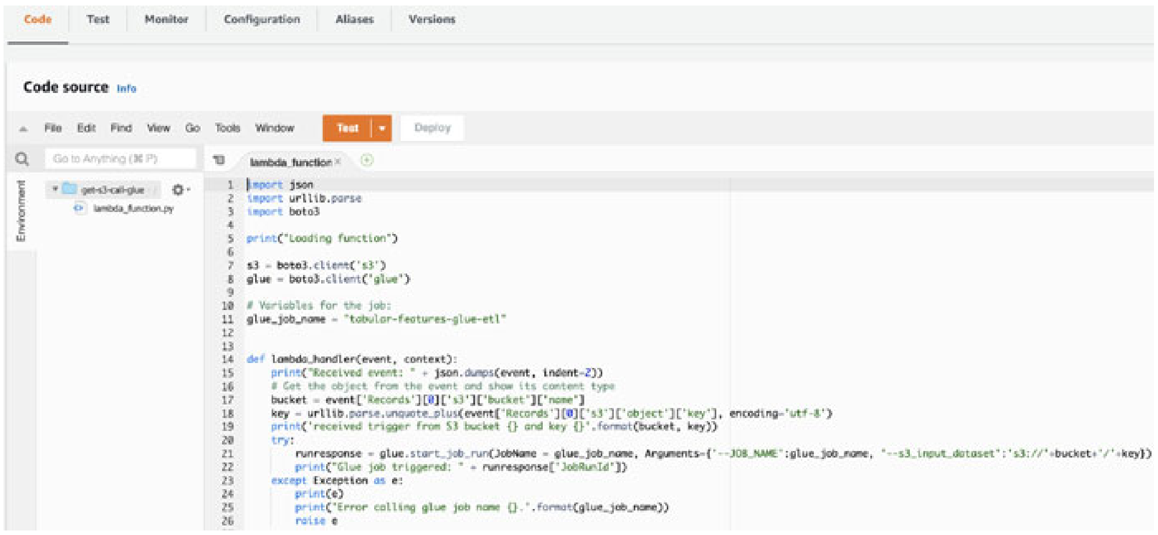
Role creation might take a few minutes. Please do not delete the role or edit the trust or permissions policies in this role.

Lambda will create an execution role named get-s3-call-glue-role-j465x51d, with permission to upload logs to Amazon CloudWatch Logs.

Figure 4.20: *Configure Lambda function*

1. Scroll down all the way to the bottom of the page (you do not have to enter the S3 trigger details here) and click on **`Create function`**. Ignore the error message you get at the top of the page for now; we will add the S3 trigger shortly. Scroll down to the **`Code`** section and copy/paste the Lambda code contents from the **`get-s3-call-glue.py`** file in the GitHub repo (**https://github.com/cloud-native-aiml-on-aws/Chapter-04/get-s3-call-glue.py**), as shown in the following image:

   Tip: Take a moment to review the code we pasted to the Lambda function. The code receives the event from the S3 bucket, extracts the bucket name, and the key to where the file resides, and starts the Glue job by passing a job name and the location to where the raw dataset we uploaded to S3 resides.



```
import json
import urllib.parse
import boto3

print("Loading function")

s3 = boto3.client('s3')
glue = boto3.client('glue')

# Variables for the job:
glue_job_name = "tabular-features-glue-etl"


def lambda_handler(event, context):
    print("Received event: " + json.dumps(event, indent=2))
    # Get the object from the event and show its content type
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']['key'], encoding='utf-8')
    print('received trigger from S3 bucket {} and key {}'.format(bucket, key))
    try:
        runresponse = glue.start_job_run(JobName = glue_job_name, Arguments={'--JOB_NAME':glue_job_name, "--s3_input_dataset":'s3://'+bucket+'/'+key})
        print("Glue job triggered: " + runresponse['JobRunId'])
    except Exception as e:
        print(e)
        print('Error calling glue job name {}.'.format(glue_job_name))
        raise e
```

Figure 4.21: *Copy and paste the Lambda code*

2. In order for our code to work, we need to provide additional permissions for Lambda to pass the role to Glue and also to execute the Glue APIs in the code. Click on the **`Configuration`** tab, click on **`Permissions`** on the left and right-click on the link provided under **`Role name`**; open in a new tab to be navigated to the AWS IAM console for the role. Refer to *Figure 4.22*:

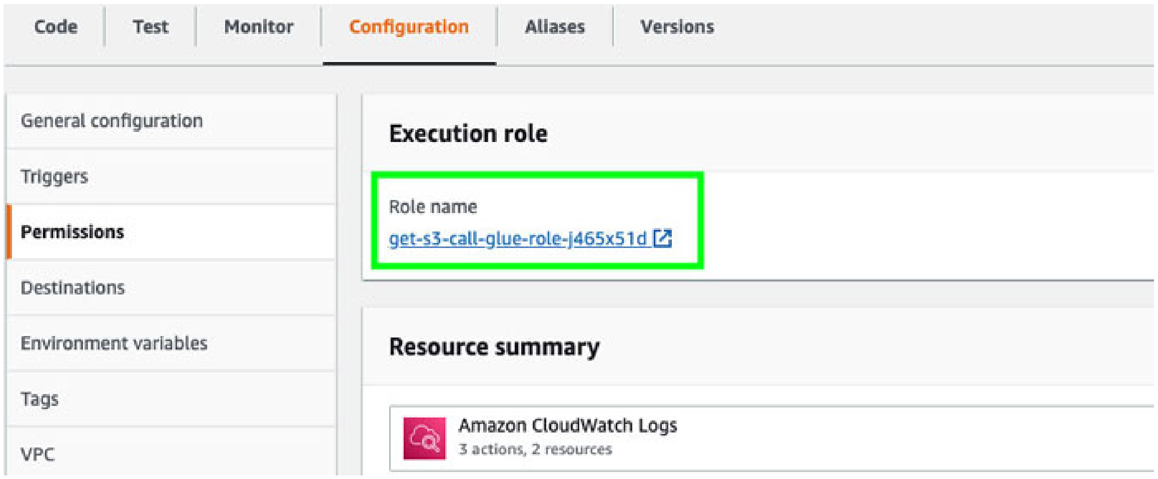
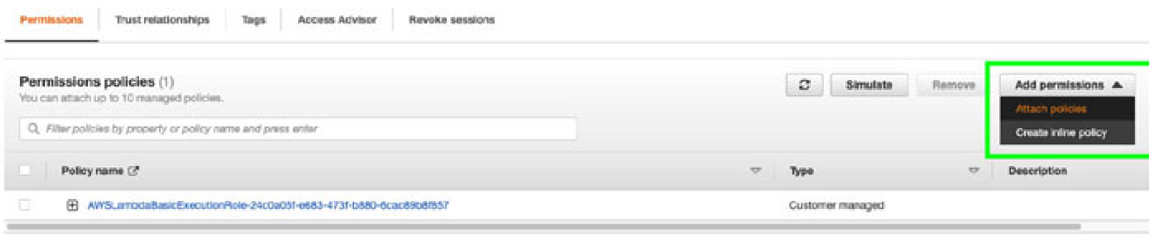


Figure 4.22: *AWS Lambda function's IAM role*

3. In the IAM console, in the **`Permissions`** tab (selected by default), click on the **`Add permissions`** button and then on **`Attach policies`**, as shown in the following image:

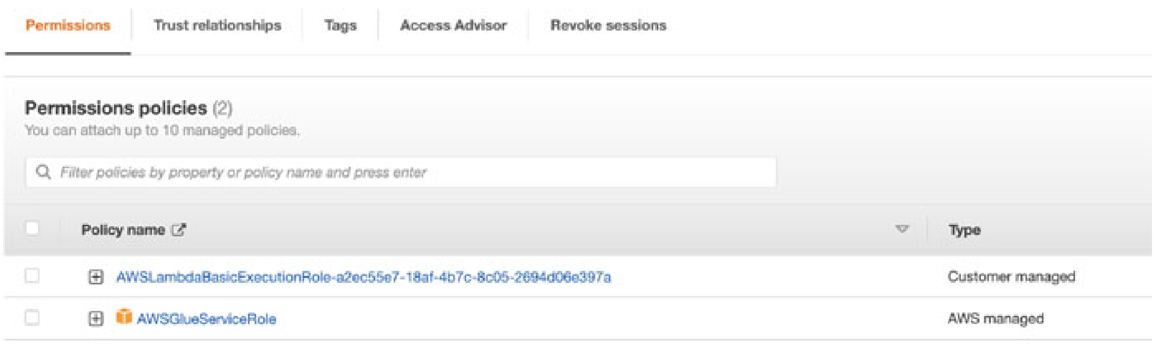


Permissions policies (1)
You can attach up to 10 managed policies.

Policy name	Type	Description
AWSLambdaBasicExecutionRole-24c0a05f-e683-473f-b880-6cac89b8f857	Customer managed	

Figure 4.23: Attach policies

4. In the **`Other permissions policies`** search bar, type **`glue`** and press *Enter*. Scroll down the list of permissions and select **`AWSGlueServiceRole`**. Scroll down to the bottom-right corner of the page and click on **`Attach policies`**. Your permissions should be updated, as shown in the following image:

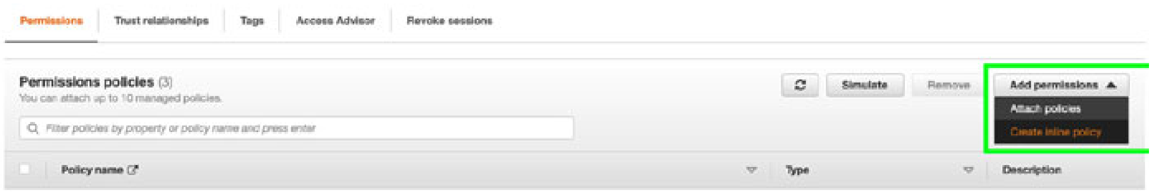


Permissions policies (2)
You can attach up to 10 managed policies.

Policy name	Type
AWSLambdaBasicExecutionRole-a2ec55e7-18af-4b7c-8c05-2694d06e397a	Customer managed
AWSGlueServiceRole	AWS managed

Figure 4.24: Glue policy attached to Lambda role

5. Now, click on the **`Add permissions`** button again and click on **`Create inline policy`**. We need to add a permission for Lambda to be able to pass this role to Glue during code execution. Refer to *Figure 4.25*:



Permissions policies (3)
You can attach up to 10 managed policies.

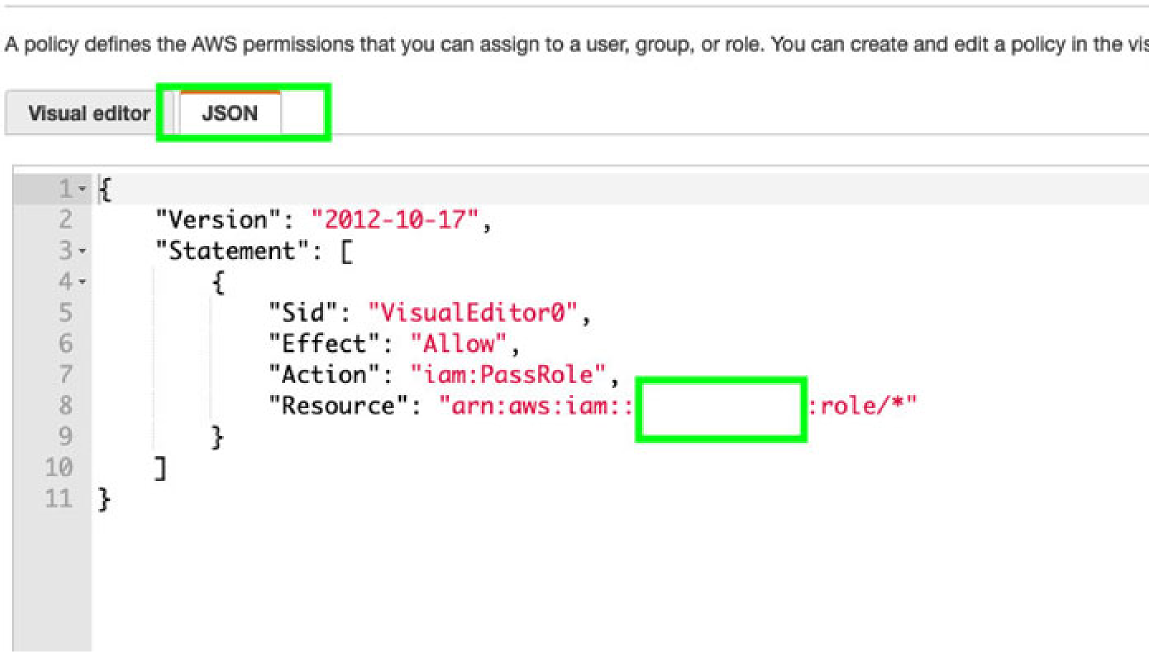
Policy name	Type	Description

Figure 4.25: Add inline policy

6. Click on the **`JSON`** tab at the top (next to Visual editor) and paste the following JSON policy statement in the input area, making sure to replace **`<aws-account-number>`** with your AWS account number (found on the top-right corner of the page in the **`your-user-name@aws-account-number`** format):

```
{
 "Version": "2012-10-17",
 "Statement": [
    {
     "Sid": "VisualEditor0",
     "Effect": "Allow",
     "Action": "iam:PassRole",
     "Resource": "arn:aws:iam::<aws-account-number>:role/*"
    }
  ]
}
```

This should be as shown in the following image:

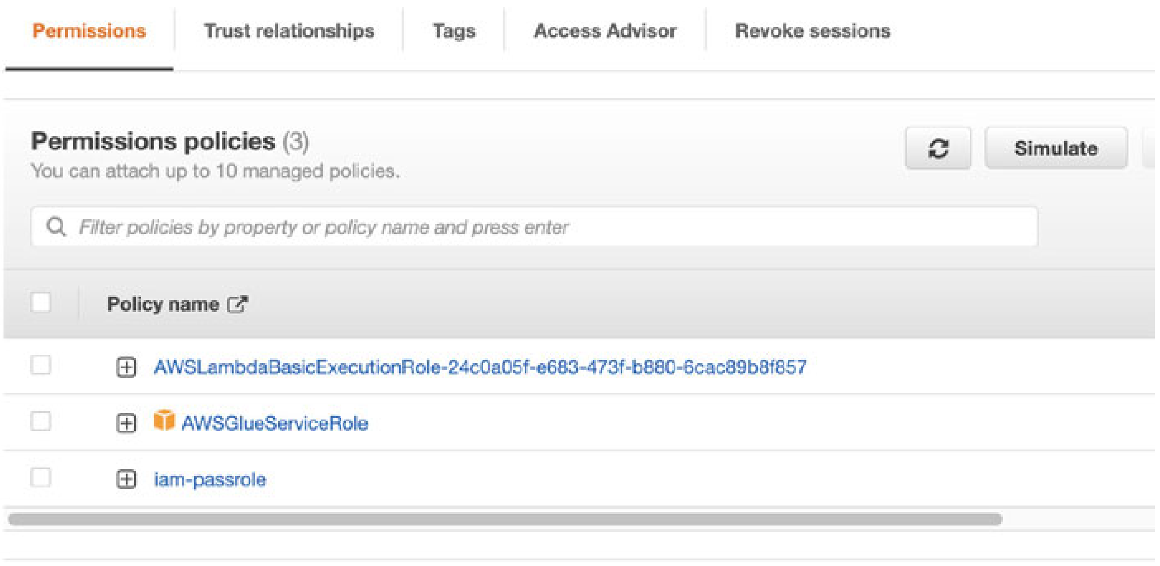


A policy defines the AWS permissions that you can assign to a user, group, or role. You can create and edit a policy in the vis

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": "iam:PassRole",
            "Resource": "arn:aws:iam::                :role/*"
        }
    ]
}
```

Figure 4.26: *Add inline policy*

7. Now, click on **`Review policy`** at the bottom-right corner of the page. Provide a name for this policy, such as **`iam-passrole`**, and click on the **`Save changes`** button in the bottom-right corner of the page. Your updated role should look as shown in the following image:



Permissions policies (3)

You can attach up to 10 managed policies.

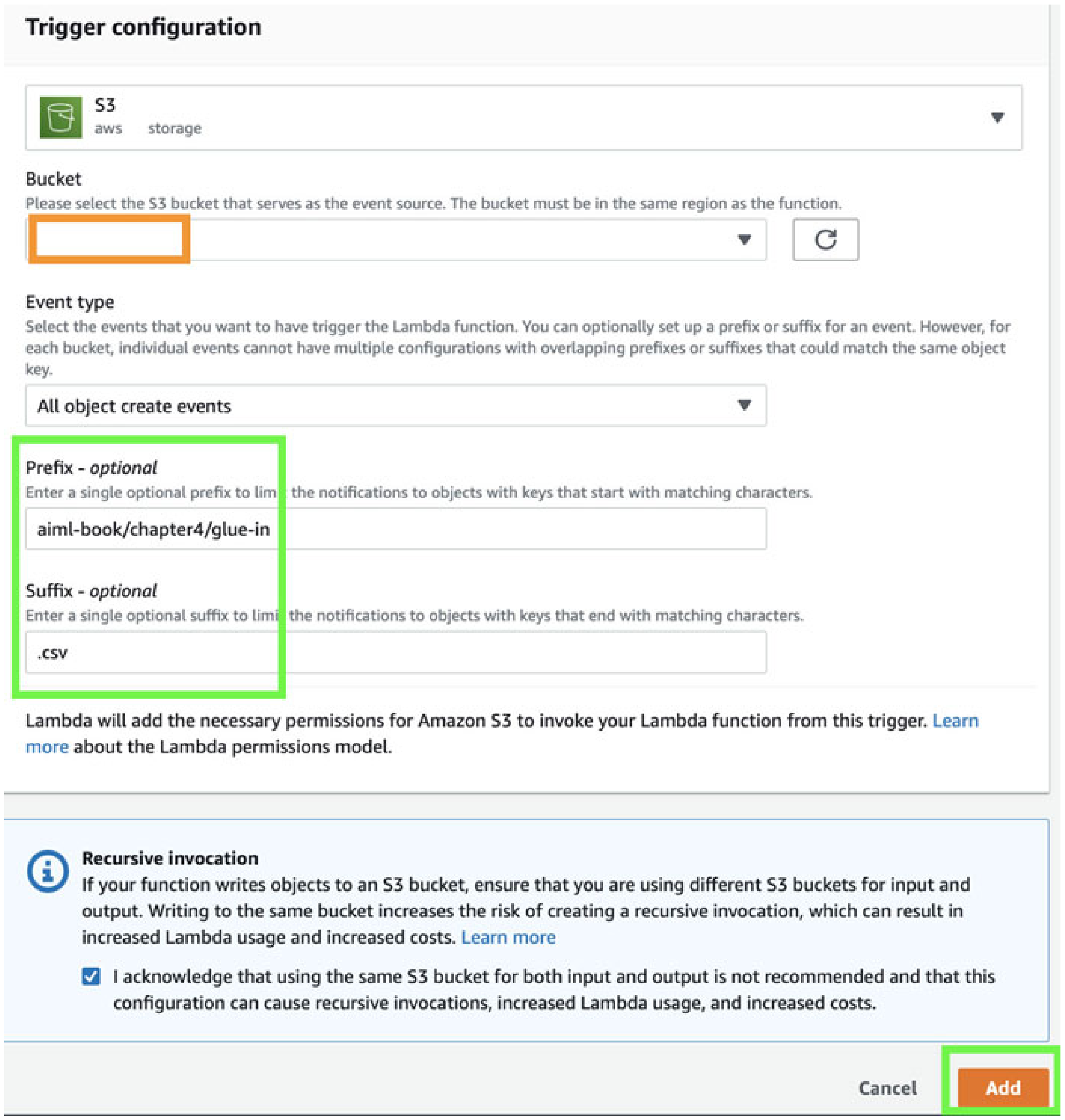
AWSLambdaBasicExecutionRole-24c0a05f-e683-473f-b880-6cac89b8f857

AWSGlueServiceRole

iam-passrole

Figure 4.27: *Lambda role policies attached*

Now go back to the AWS Lambda console (this should be open in a separate tab in your browser, or simply type **`Lambda`** in the services search bar and go to the Lambda console), and in the **`Function overview`** section at the top of the page, click on the **`Add trigger`** button on the left. On the **`Add trigger`** page, click on the list box underneath **`Trigger configuration`** and select **`S3`**. In the list box under **`Bucket`**, type the name of the bucket you created and select it. Leave the default value in the **`Event type`** and enter **`aiml-book/chapter4/glue-in`** in the Prefix – optional input field. Enter **`.csv`** in the Suffix – optional field. Scroll down and select the checkbox to acknowledge the **`Recursive invocation`** message. Click on the **`Add`** button in the bottom-right corner of the page. Refer to *Figure 4.28*:


Trigger configuration

S3
aws storage

Bucket
Please select the S3 bucket that serves as the event source. The bucket must be in the same region as the function.

Event type
Select the events that you want to have trigger the Lambda function. You can optionally set up a prefix or suffix for an event. However, for each bucket, individual events cannot have multiple configurations with overlapping prefixes or suffixes that could match the same object key.

All object create events

Prefix - *optional*
Enter a single optional prefix to limit the notifications to objects with keys that start with matching characters.

aiml-book/chapter4/glue-in

Suffix - *optional*
Enter a single optional suffix to limit the notifications to objects with keys that end with matching characters.

.csv

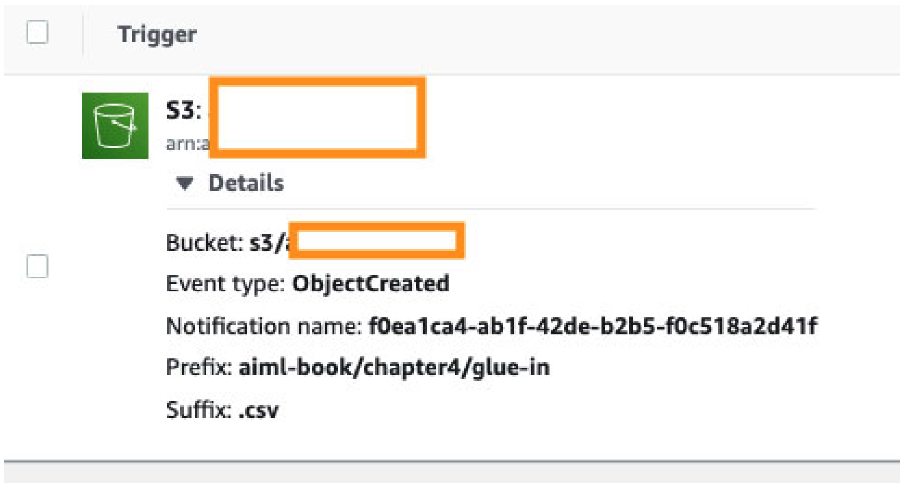
Lambda will add the necessary permissions for Amazon S3 to invoke your Lambda function from this trigger. Learn more about the Lambda permissions model.

Recursive invocation
If your function writes objects to an S3 bucket, ensure that you are using different S3 buckets for input and output. Writing to the same bucket increases the risk of creating a recursive invocation, which can result in increased Lambda usage and increased costs. Learn more

☑ I acknowledge that using the same S3 bucket for both input and output is not recommended and that this configuration can cause recursive invocations, increased Lambda usage, and increased costs.

Figure 4.28: *Attach S3 trigger to Lambda function*

Once the trigger is added, it is reflected in the Function overview section, and if you click on the **`S3`** pane, you can see the details, as shown in the following image. Now, whenever you upload a file to your S3 bucket in the **`aiml-book/chapter4/glue-in`** location, this Lambda function will be automatically invoked. Refer to *Figure 4.29*:



S3: [redacted]

arn:a

Bucket: s3/[redacted]

Event type: ObjectCreated

Notification name: f0ea1ca4-ab1f-42de-b2b5-f0c518a2d41f

Prefix: aiml-book/chapter4/glue-in

Suffix: .csv

Figure 4.29: *S3 trigger attached*

That brings us to the end of all the configuration steps we needed for building the solution. In the next subsection, we will test our solution and verify the results.

Running the solution

We will now test our solution by uploading the raw dataset to the trigger location in the S3 bucket we created. This will trigger the AWS Lambda function, which, in turn, will start the AWS Glue job. The job will extract the file and run it through a series of feature engineering steps and finally, generate a processed and scaled dataset ready for ML training. To understand how the Glue job works, take a moment to review the Glue ETL code available in tabular-features-glue-etl.py in our GitHub respository (**https://github.com/cloud-native-aiml-on-aws/Chapter-04/tabular-features-glue-etl.py**). The following code blocks receive parameters from the AWS Lambda function and read the raw dataset we uploaded to the S3 bucket:

```
args = getResolvedOptions(sys.argv, ['JOB_NAME',
's3_input_dataset'])
s3_input_dataset = args['s3_input_dataset']
print("The s3 input file is: " + s3_input_dataset)
# Let's first load the data into a Pandas dataframe so it is easy
for us to work with it
wine_raw_df = wr.s3.read_csv(s3_input_dataset, sep=';',header=0)
```

The subsequent lines of code will process and transform the data as part of feature engineering. Finally, a new location called glue-out is created, and the processed dataset is uploaded to this location, as shown in the following code snippet:

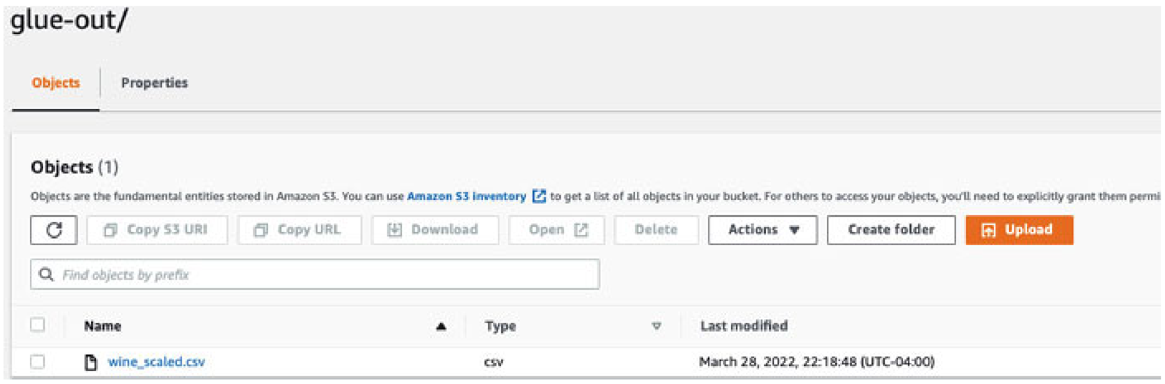
```
s3_glue_out = ''
```

```
temp = s3_input_dataset.split('/')[6]
s3_mod = s3_input_dataset.replace(temp,'')
if 'glue-in' in s3_mod:
  s3_glue_out = s3_mod.replace('glue-in','glue-out')
wr.s3.to_csv(wine_scaled_df, s3_glue_out+'wine_scaled.csv',
header=True, index=False)
```

To upload the raw dataset, open the SageMaker Studio notebook, navigate to **`Chapter-04`**, and open the **`glue-script-loader.ipynb`** notebook. Execute the code in the last cell underneath the heading IMPORTANT. This will take the raw data file that was copied to the notebook when you originally cloned the GitHub repository in the notebook and upload it to the S3 bucket and the location you specified in the first cell of this notebook. If it is unable to recognize the bucket and prefix variables, re-execute the first cell and try executing the last cell again:

```
# upload csv file for testing the end to end solution
infile = 'winemag-data_first150k.csv'
s3.upload_file(infile,bucket,prefix+'glue-in/'+infile)
```

Now go to your AWS Management Console, type S3 in the services search bar, go the **`s3`** console, type your bucket name in the search bar, and open your S3 bucket. Open the **`aiml-bucket`** prefix and then **`chapter4`**. You should see two new folders appear here: **`glue-in`** and **`glue-out`**. The glue-in folder appears as soon as you execute the cell to upload the raw dataset, and the glue-out folder appears when the Glue has successfully runs the feature engineering task and uploads the processed dataset. Click the **`glue-out`** folder and review the **`wine_scaled.csv`** file by selecting and downloading it locally. Refer to *Figure 4.30*:



Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permi

Name	Type	Last modified
wine_scaled.csv	csv	March 28, 2022, 22:18:48 (UTC-04:00)

Figure 4.30: *Processed dataset after feature engineering*

If you face any issues while running the solution or if it does not create the processed dataset, check the following two things:

1. Check whether the AWS Glue job name matches the **`glue_job_name`** parameter you specified in your AWS Lambda function.
2. Check whether the S3 bucket name and prefix location for the dataset match in the AWS Lambda S3 trigger, in the SageMaker Studio notebook and in the actual S3 bucket.

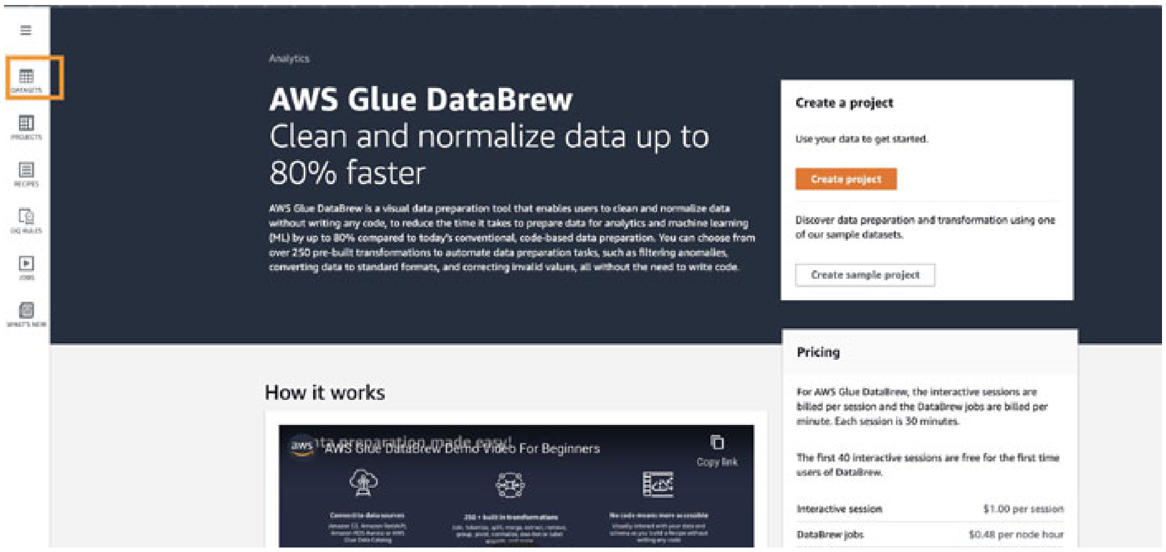
You can also review the AWS Glue and AWS Lambda logs to get a closer look at how the solution ran. In the AWS Lambda console, click on your function name, click on the **`Monitor`** tab in the middle of the page, and then on button **`View logs`** button in **`CloudWatch`**. This will navigate you directly to the Amazon CloudWatch (**https://aws.amazon.com/cloudwatch/**) log group where all your Lambda function logs are stored. Open the latest log to review the run status. Similarly, go to the AWS Glue console, click on **`Jobs`** in the left menu, click on your **`Glue job`** name, and then click on the **`Runs`** tab at the top of the page. Under the Recent job runs, you will see a CloudWatch logs heading; click on **`All logs`** under that. This will again take you to CloudWatch but for the Glue job.

That brings us to the end of using AWS Glue ETL jobs for creating a data orchestration flow for feature engineering. In the next section, we will use a new service called AWS Glue DataBrew (**https://aws.amazon.com/glue/features/databrew/**) for taking a closer look at our data, understanding its characteristics and composition, and using a visual interface to perform some of the feature engineering tasks we learned in *Chapter 3, Predicting the Future with features*. Finally, we will learn how to create jobs to automate the tasks we performed in the Glue DataBrew UI and run it on demand.

Data profiling with AWS Glue DataBrew

Glue DataBrew (**https://us-east-1.console.aws.amazon.com/databrew/home?region=us-east-1#landing**) is an interactive UI tool for data processing, transformation and performing comprehensive data manipulation tasks, including feature engineering for ML. The advantage of DataBrew is the comprehensive visualization options that help in exploring our datasets quite deeply, generating columnar and row statistics, understanding data distributions, and quickly identifying outliers, all of which are important inputs in finalizing our feature engineering strategy for our raw dataset. In this section, we will run a series of data processing tasks we are now familiar with, but we will use DataBrew instead of Glue ETL or SageMaker notebooks. Let's get started.

Like earlier, you need access to AWS Management Console, so review and follow the *Technical Requirements* section to sign up for/sign in to your AWS console. In the services search bar at the top, type **`DataBrew`** and click on **`AWS Glue DataBrew`** to navigate to the DataBrew console. In the console, click on the **DATASETS** option in the left menu, as shown in the following image:



AWS Glue DataBrew

Clean and normalize data up to 80% faster

AWS Glue DataBrew is a visual data preparation tool that enables users to clean and normalize data without writing any code, to reduce the time it takes to prepare data for analytics and machine learning (ML) by up to 80% compared to today's conventional, code-based data preparation. You can choose from over 250 pre-built transformations to automate data preparation tasks, such as filtering anomalies, converting data to standard formats, and correcting invalid values, all without the need to write code.

Create a project

Use your data to get started.

Discover data preparation and transformation using one of our sample datasets.

Pricing

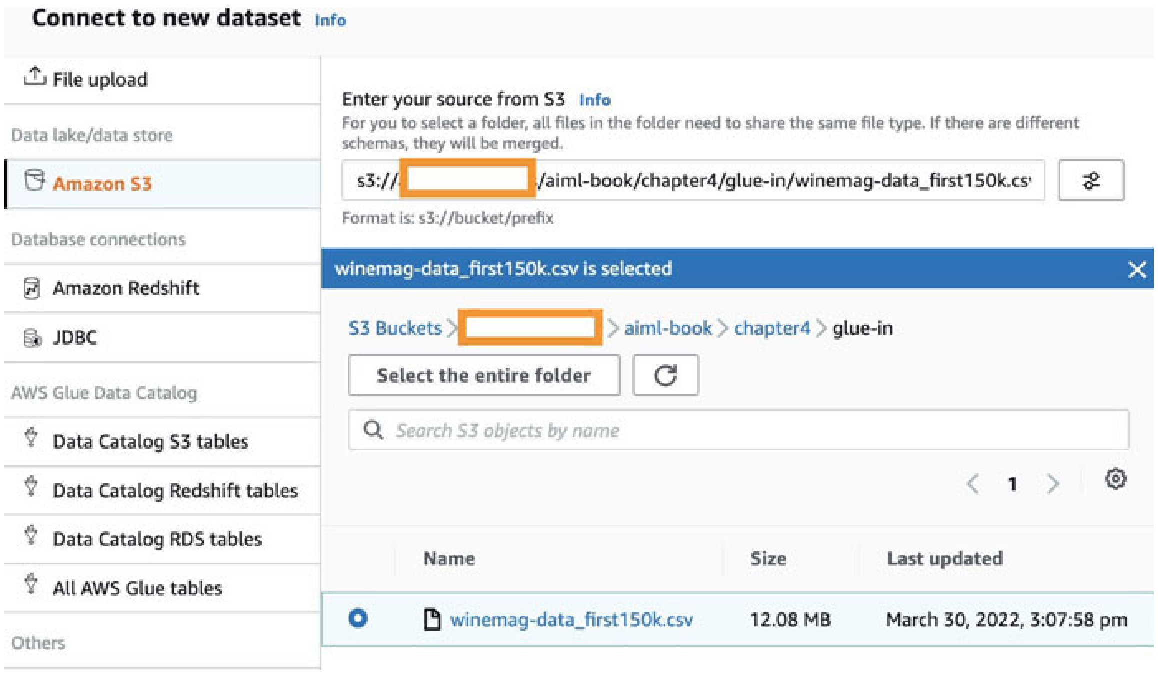
For AWS Glue DataBrew, the interactive sessions are billed per session and the DataBrew jobs are billed per minute. Each session is 30 minutes.

The first 40 interactive sessions are free for the first time users of DataBrew.

Interactive session	$1.00 per session
DataBrew jobs	$0.48 per node hour

Figure 4.31: *AWS Glue DataBrew console*

1. Click on the **`Connect new dataset`** button on the right of the page. In the **`New connection`** page that appears, enter a **`Dataset`** name under **`New dataset details`**. In the **`Connect to new dataset`** section, leave the default selection of **`Amazon S3`** as is, and click on the input area under **`Enter your source from S3`** and provide the path as **`s3://<your-bucket>/aiml-book/chapter4/ glue-in/winemag-data_first150k.csv`**. You can also browse to your S3 bucket prefixes using the search bar at the bottom of the input area, as shown in the following image:



Connect to new dataset Info

File upload

Data lake/data store

Amazon S3

Database connections

Amazon Redshift

JDBC

AWS Glue Data Catalog

Data Catalog S3 tables

Data Catalog Redshift tables

Data Catalog RDS tables

All AWS Glue tables

Others

Enter your source from S3 Info

For you to select a folder, all files in the folder need to share the same file type. If there are different schemas, they will be merged.

s3://[redacted]/aiml-book/chapter4/glue-in/winemag-data_first150k.csv

Format is: s3://bucket/prefix

winemag-data_first150k.csv is selected

S3 Buckets > [redacted] > aiml-book > chapter4 > glue-in

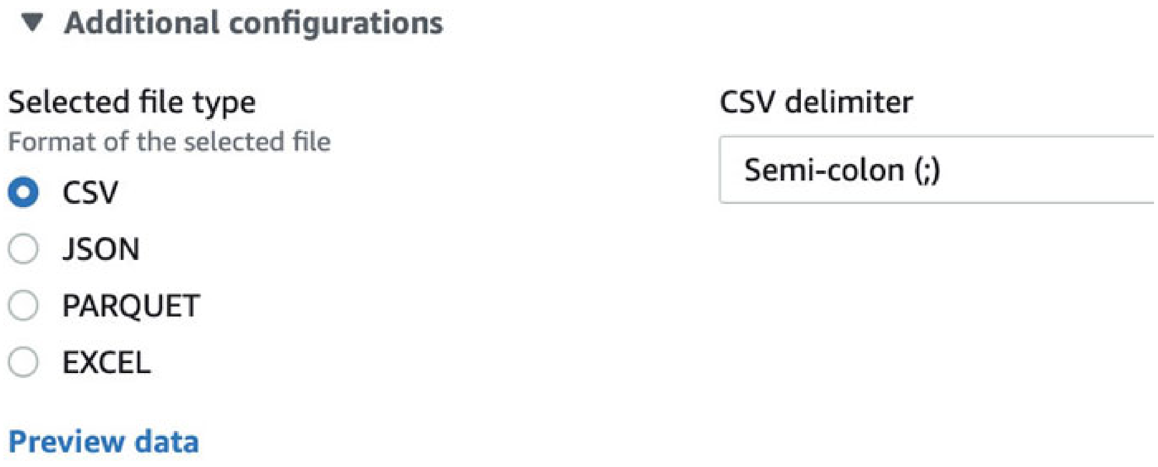
Select the entire folder

Search S3 objects by name

	Name	Size	Last updated
●	winemag-data_first150k.csv	12.08 MB	March 30, 2022, 3:07:58 pm

Figure 4.32: *Connect to a new dataset from Glue DataBrew*

2. Scroll down this page and change the **`CSV delimiter`** under **`Additional configurations`** to a **`Semi-colon`**, as shown in the following image:



▼ Additional configurations

Selected file type

Format of the selected file

[✓ CSV] JSON PARQUET EXCEL

CSV delimiter: Semi-colon (;)

Preview data

Figure 4.33: *Change dataset delimiter to Semi-colon*

3. Leave the rest of the defaults as they are and click on **`Create dataset`** in the bottom-right corner. You will be taken to the **`Datasets`** page, where your newly created dataset will appear. Click on the name of your new dataset to view the details of your dataset. Click on the **`Data profile overview`** tab and then on **`Run data profile`**, as shown in the following image:

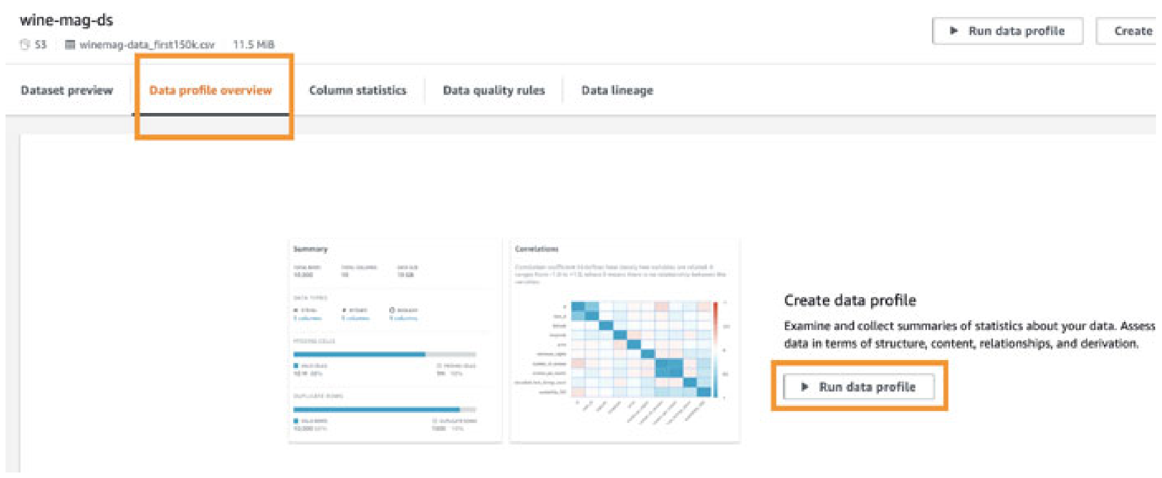
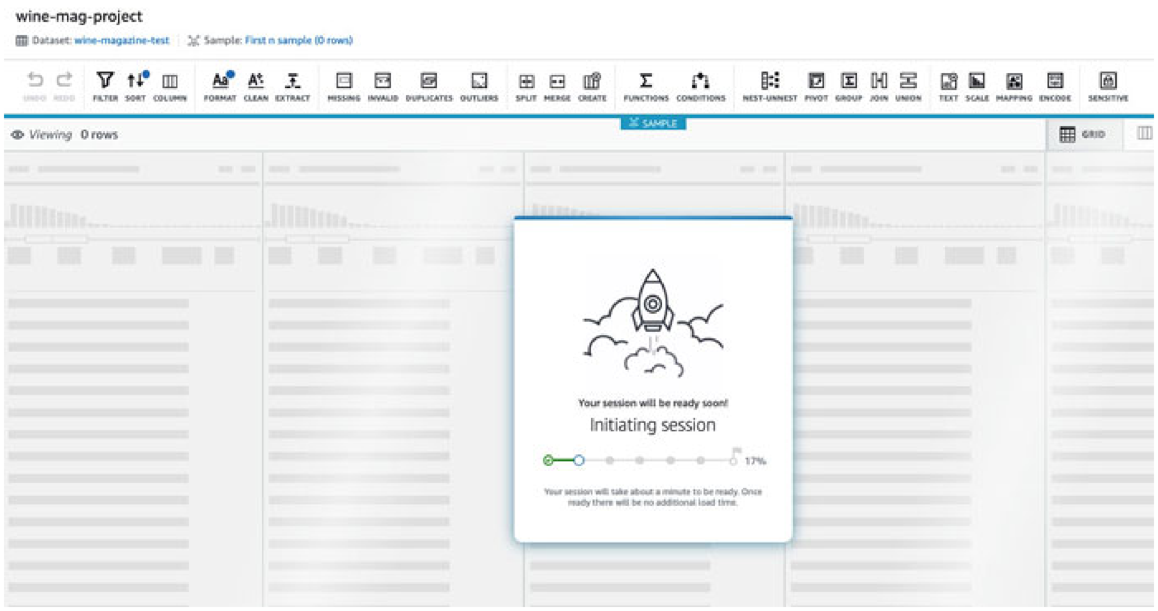


Figure 4.34: *Run data profile*

4. In the `Create job` page that opens up, scroll down to S3 location in the Job output settings and type `s3://a2i-experiments/aiml-book/chapter4/glue-out/` (you should have executed the previous section for the glue-out folder to appear). Now, scroll down to the `Permissions` section at the bottom; click on the list box and select `Create a new IAM role`. Type `chapter4` in the New IAM role suffix and click on `Create and run job`. The job will take a few minutes to complete.
5. In the meantime, click on `PROJECTS` in the left pane and then on the `Create project` button. Enter a `Project` name and scroll down to `Select a dataset`. Select your newly created dataset under `My datasets`. Scroll down to `Permissions` and select the IAM role (ending with `chapter4`) you created in the previous step (when you created the data profile job). Click on `Create project`. Refer to *Figure 4.35*:



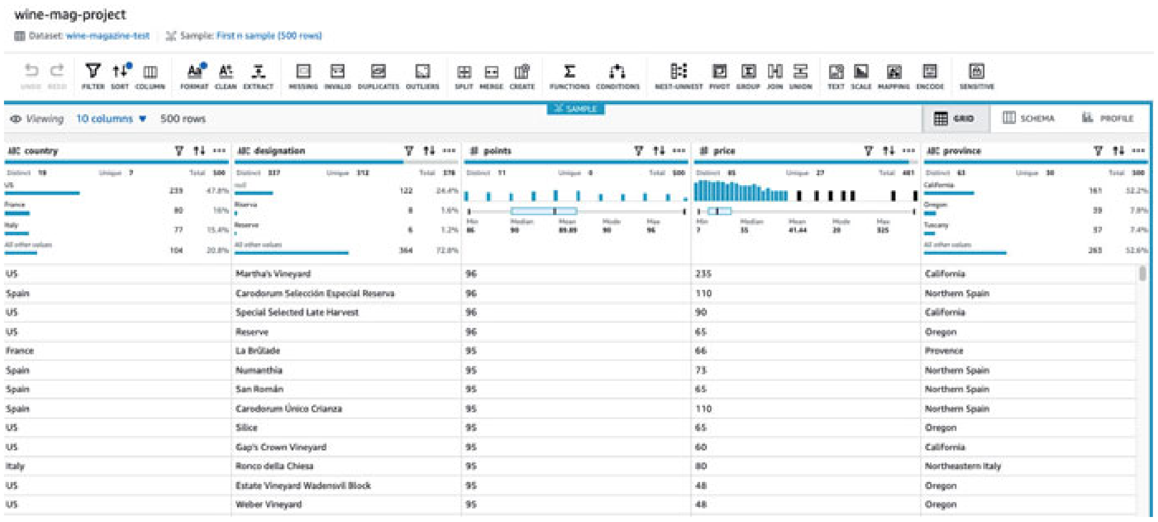
Your session will be ready soon!

Initiating session

Your session will take about a minute to be ready. Once ready there will be no additional load time.

Figure 4.35: *DataBrew project creation in progress*

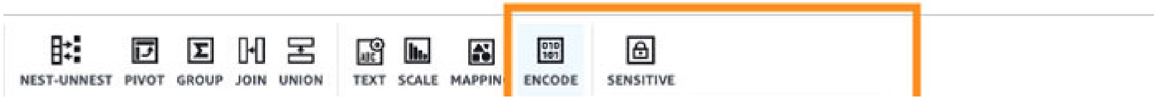
6. When the project is created, you will be able to see column statistics for a sample from your raw dataset. Every action we perform here is tracked by DataBrew as part of a **Recipe** that represents a collection of data processing tasks for a project. Refer to *Figure 4.36*:

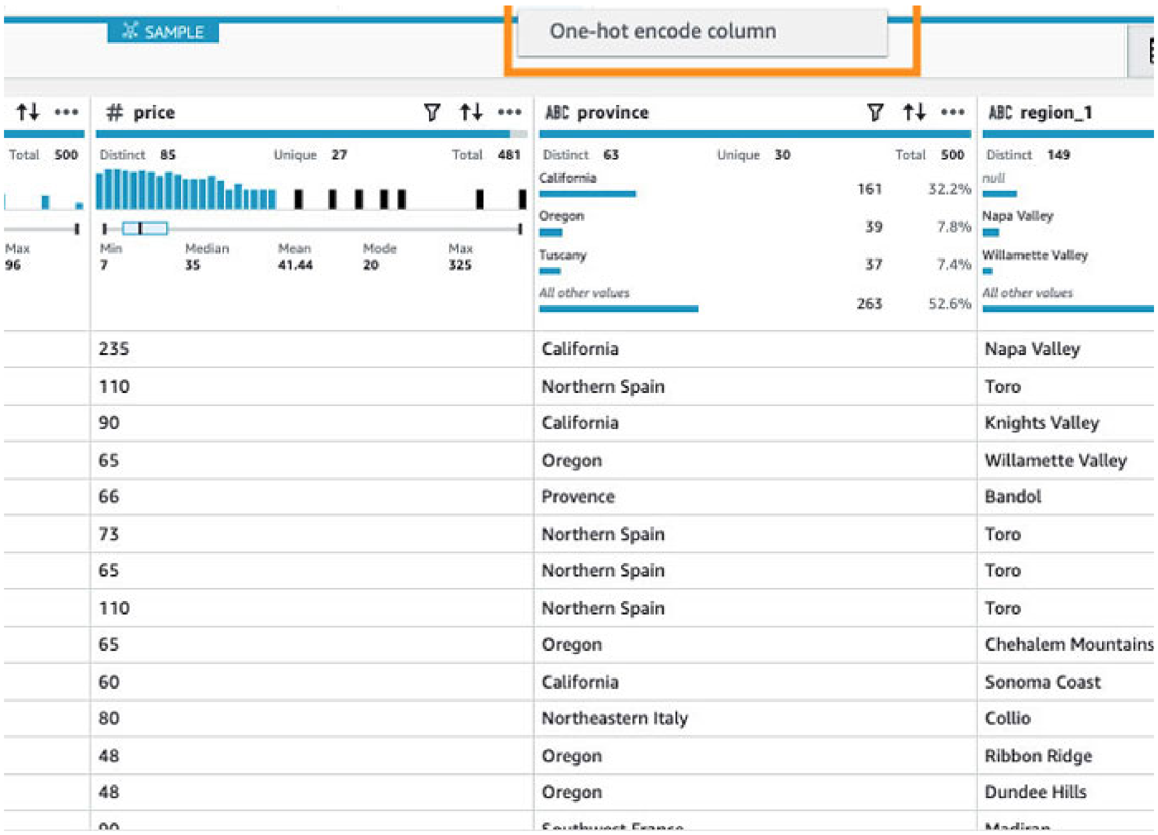


country	designation	points	price	province
US	Martha's Vineyard	96	235	California
Spain	Carodorum Selección Especial Reserva	96	110	Northern Spain
US	Special Selected Late Harvest	96	90	California
US	Reserve	96	65	Oregon
France	La Brûlade	95	66	Provence
Spain	Numanthia	95	73	Northern Spain
Spain	San Román	95	65	Northern Spain
Spain	Carodorum Único Crianza	95	110	Northern Spain
US	Silice	95	65	Oregon
US	Gap's Crown Vineyard	95	60	California
Italy	Ronco della Chiesa	95	80	Northeastern Italy
US	Estate Vineyard Wadensvil Block	95	48	Oregon
US	Weber Vineyard	95	48	Oregon

Figure 4.36: *DataBrew project*

7. The country column is categorical and has a limited set of unique values. Let us perform one-hot-encoding for this. Click on **ENCODE** in the top menu on the right and select **One-hot encode column**. Refer to *Figure 4.37*:

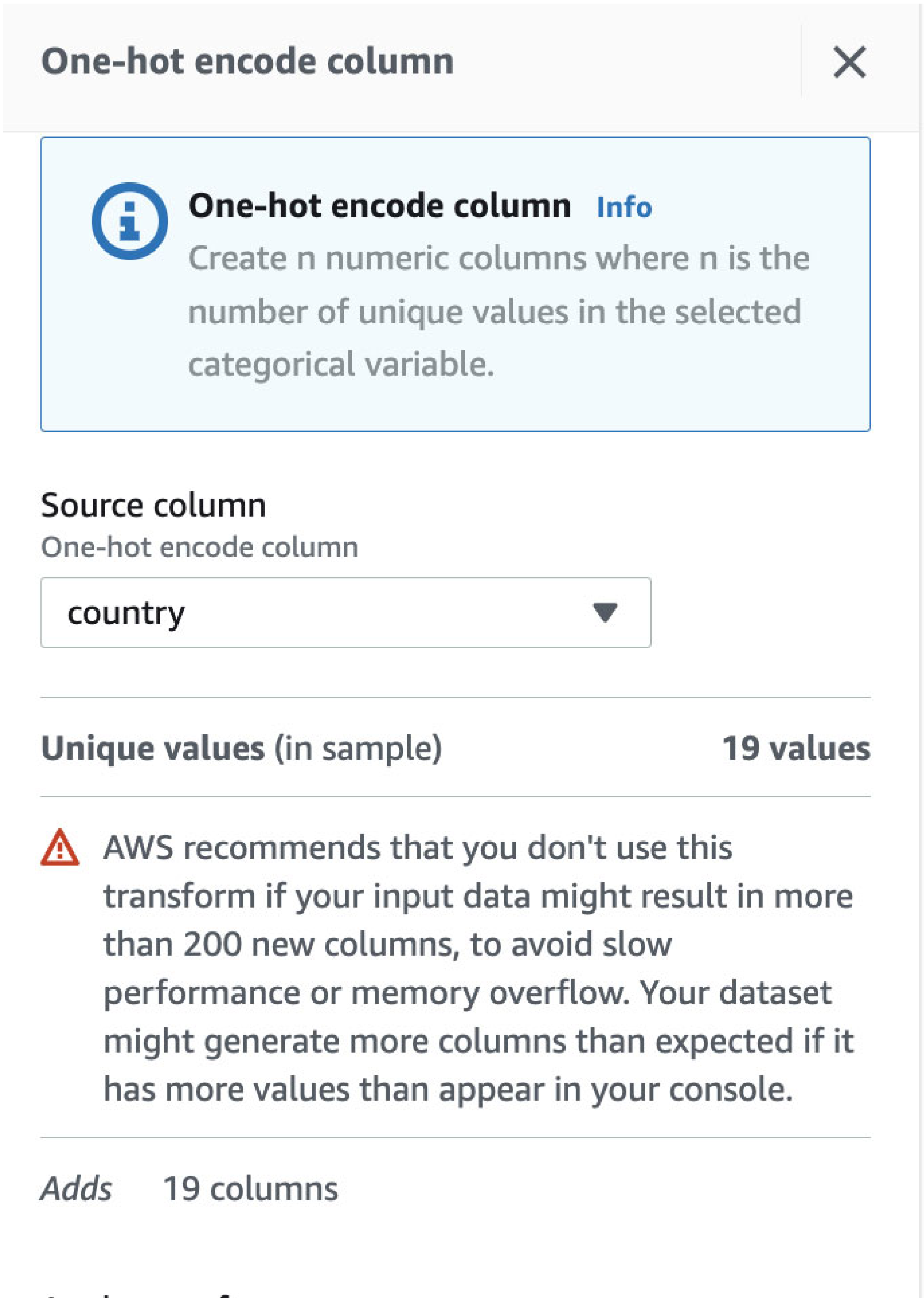




	# price	ABC province	ABC region_1
Total 500, Max 96	Distinct 85, Unique 27, Total 481; Min 7, Median 35, Mean 41.44, Mode 20, Max 325	Distinct 63, Unique 30, Total 500; California 161 32.2%; Oregon 39 7.8%; Tuscany 37 7.4%; All other values 263 52.6%	Distinct 149; null; Napa Valley; Willamette Valley; All other values
	235	California	Napa Valley
	110	Northern Spain	Toro
	90	California	Knights Valley
	65	Oregon	Willamette Valley
	66	Provence	Bandol
	73	Northern Spain	Toro
	65	Northern Spain	Toro
	110	Northern Spain	Toro
	65	Oregon	Chehalem Mountains
	60	California	Sonoma Coast
	80	Northeastern Italy	Collio
	48	Oregon	Ribbon Ridge
	48	Oregon	Dundee Hills

Figure 4.37: *Categorical encoding of columns/features*

8. You can configure the encoding on the right of the page. In the **`Source column`** list box, select country. Scroll down and click on **`Apply`**. Refer to *Figure 4.38*:



One-hot encode column

One-hot encode column Info

Create n numeric columns where n is the number of unique values in the selected categorical variable.

Source column

One-hot encode column

country

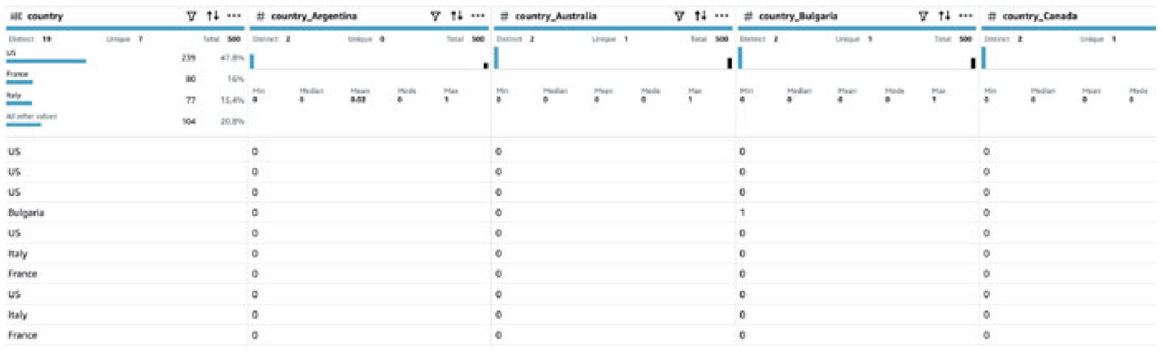
Unique values (in sample) 19 values

AWS recommends that you don't use this transform if your input data might result in more than 200 new columns, to avoid slow performance or memory overflow. Your dataset might generate more columns than expected if it has more values than appear in your console.

Adds 19 columns

Figure 4.38: *Perform one-hot encoding of country column/feature*

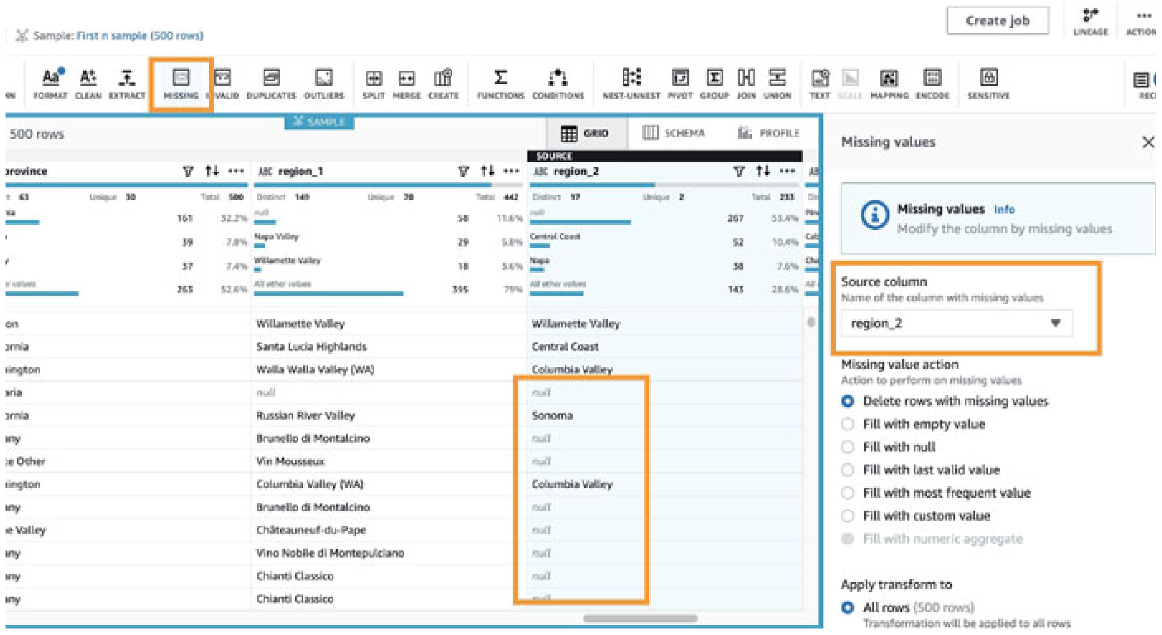
9. The country is encoded into multiple features or columns, one column for each unique country in our original dataset, as shown in the following image:



country	country_Argentina	country_Australia	country_Bulgaria	country_Canada
US	0	0	0	0
US	0	0	0	0
US	0	0	0	0
Bulgaria	0	0	1	0
US	0	0	0	0
Italy	0	0	0	0
France	0	0	0	0
US	0	0	0	0
Italy	0	0	0	0
France	0	0	0	0

Figure 4.39: *One-hot encoding results*

10. Now we will perform a missing values analysis for our columns. Select **`MISSING`** in the top pane and click on **`Remove missing rows`**. On the right, select the **`Source`** column as **`region_2`** and leave the default action of deleting rows with missing values from our dataset. Refer to *Figure 4.40*:



region_1	region_2
Willamette Valley	Willamette Valley
Santa Lucia Highlands	Central Coast
Walla Walla Valley (WA)	Columbia Valley
null	null
Russian River Valley	Sonoma
Brunello di Montalcino	null
Vin Mousseux	null
Columbia Valley (WA)	Columbia Valley
Brunello di Montalcino	null
Châteauneuf-du-Pape	null
Vino Nobile di Montepulciano	null
Chianti Classico	null
Chianti Classico	null

Missing values

Missing values Info
Modify the column by missing values

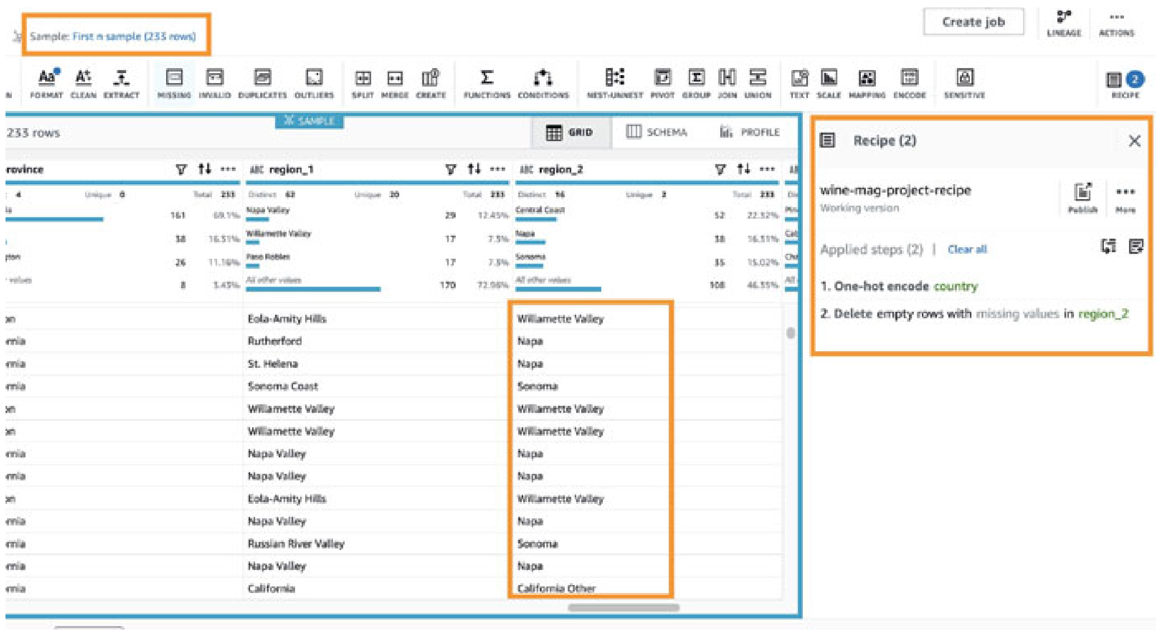
Source column
Name of the column with missing values
region_2

Missing value action
Action to perform on missing values
- [✓ Delete rows with missing values]
- Fill with empty value
- Fill with null
- Fill with last valid value
- Fill with most frequent value
- Fill with custom value
- Fill with numeric aggregate

Apply transform to
- [✓ All rows (500 rows)]
  Transformation will be applied to all rows

Figure 4.40: *Perform missing values analysis*

11. Leave the rest of the default values as they are, scroll down and click on **`Apply`**. The rows where **`region_2`** had missing values are deleted from the dataset, and our **`Recipe`** is now updated to show the two tasks we performed so far. Refer to *Figure 4.41*:



wine-mag-project-recipe
Working version

Applied steps (2)

1. One-hot encode country
2. Delete empty rows with missing values in region_2

Figure 4.41: Delete missing value rows

12. Now select **`SCALE`** in the top menu and click on **`Binning`**. Refer to [Figure 4.42](#):

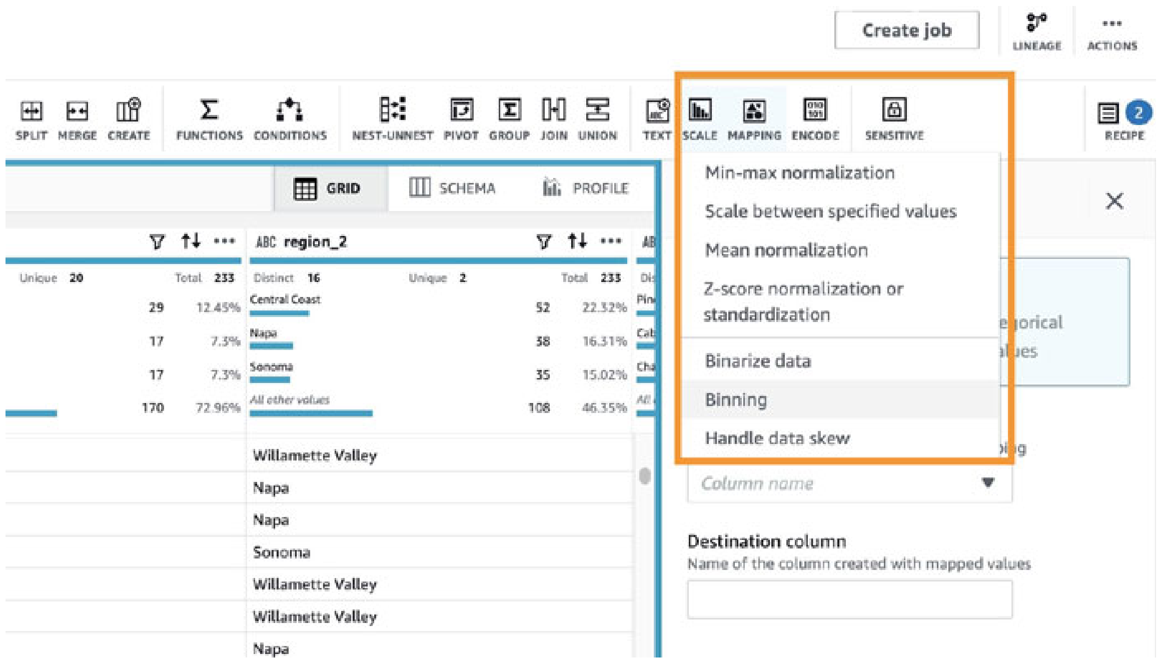
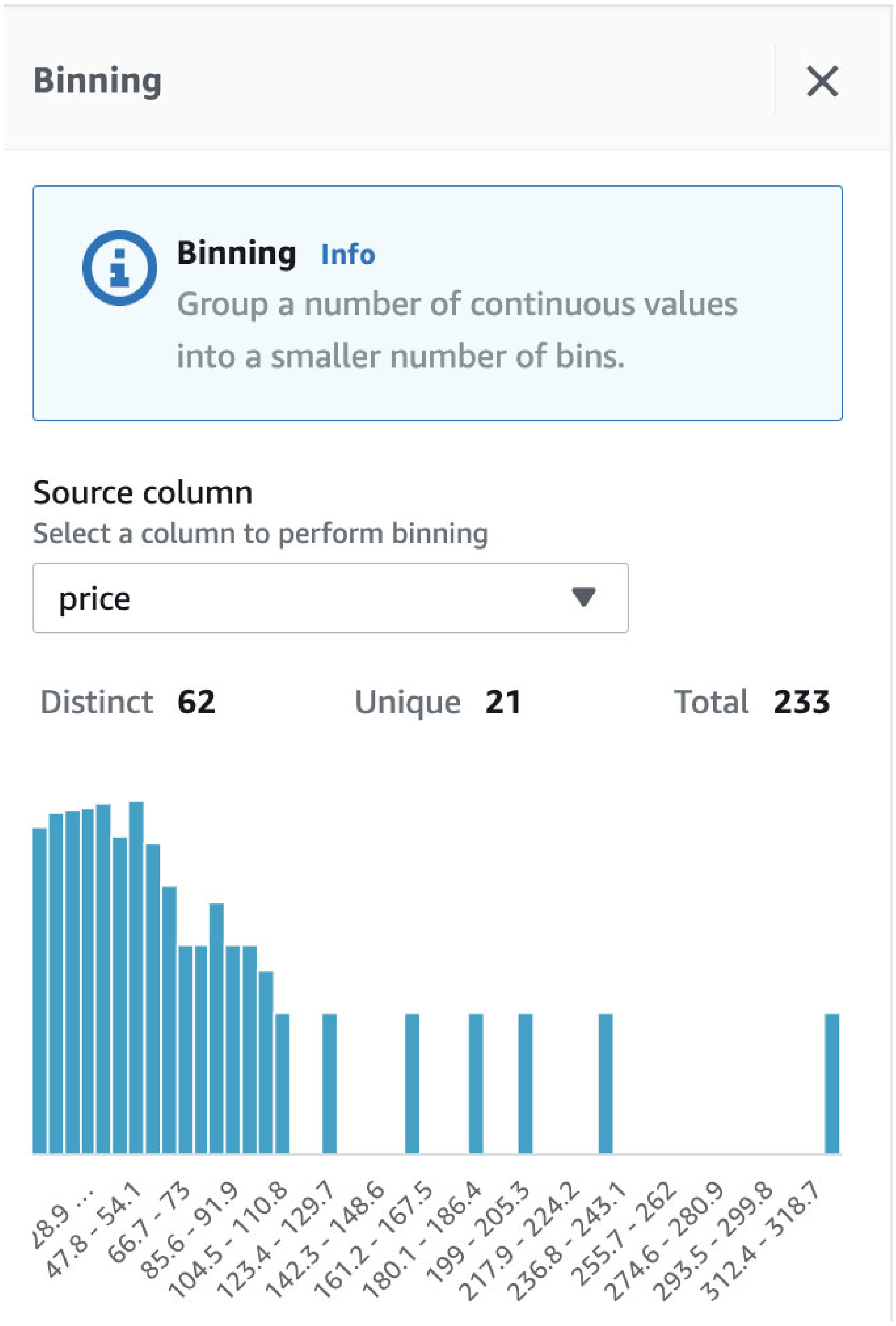


Figure 4.42: Perform Binning

13. Select the **`Source column`** as **`price`**. DataBrew immediately calculates the bins and the price ranges for each of the bins, which is displayed under **`Source column`**. Refer to [Figure 4.43](#):



Binning

Binning Info
Group a number of continuous values into a smaller number of bins.

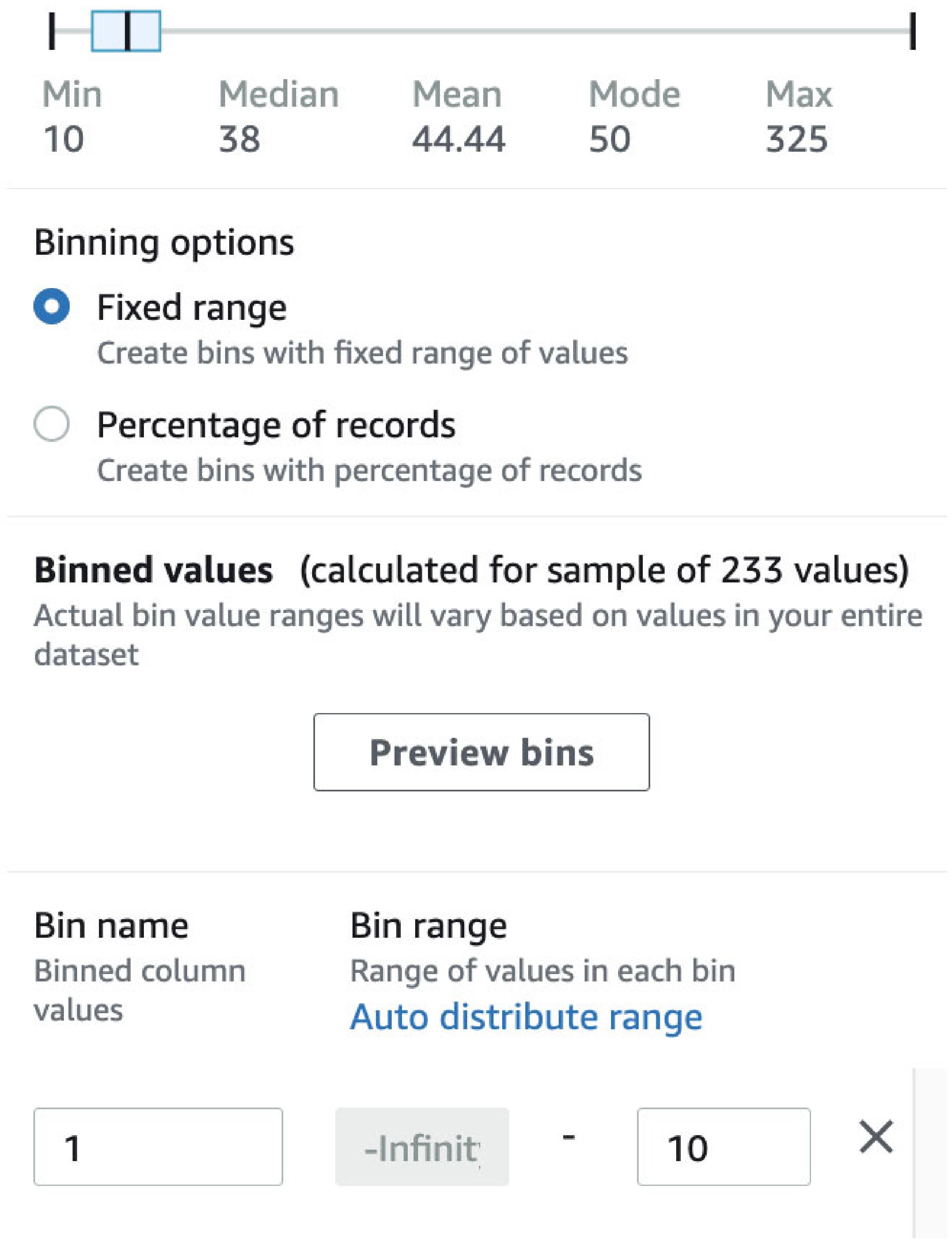
Source column
Select a column to perform binning

price

Distinct 62 Unique 21 Total 233

Figure 4.43: *Bins for price*

14. Scroll down to review the `Binning options` and the `Bin name`; you can also change the Bin name if required. If you retain the text names for the bins, make sure you perform one-hot encoding of these text values. Refer to *Figure 4.44*:



Min	Median	Mean	Mode	Max
10	38	44.44	50	325

Binning options

[✓ Fixed range] Create bins with fixed range of values

Percentage of records — Create bins with percentage of records

Binned values (calculated for sample of 233 values)

Actual bin value ranges will vary based on values in your entire dataset

Bin name — Binned column values

Bin range — Range of values in each bin

Bin name	Bin range		
1	-Infinit	-	10

Figure 4.44: *Binning options*

15. Scroll down and click on **`Apply`**. Once the binning is completed for your dataset, the **`Recipe`** is updated to show this new task. Click on **`Publish`** in the top-right corner to publish the recipe. We need to publish it before we create a job to automate the running of these tasks for future datasets. Refer to *Figure 4.45*:

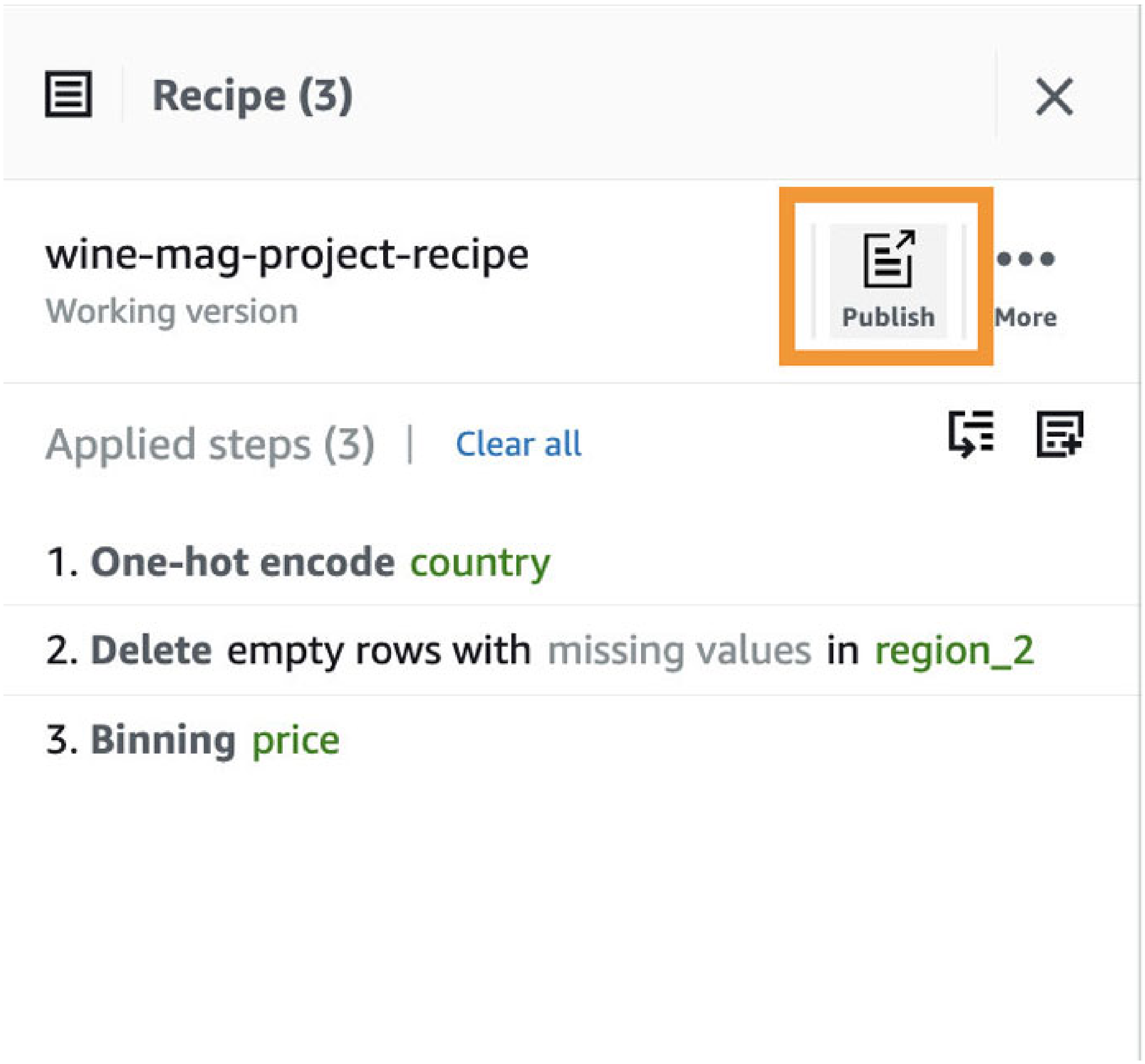
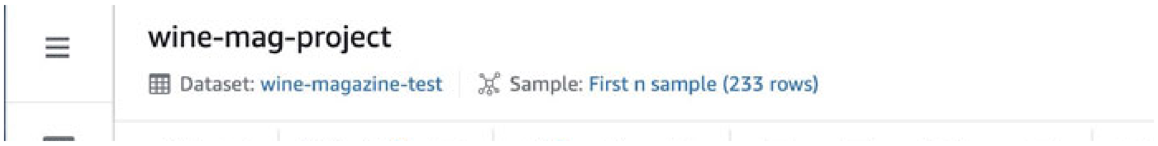
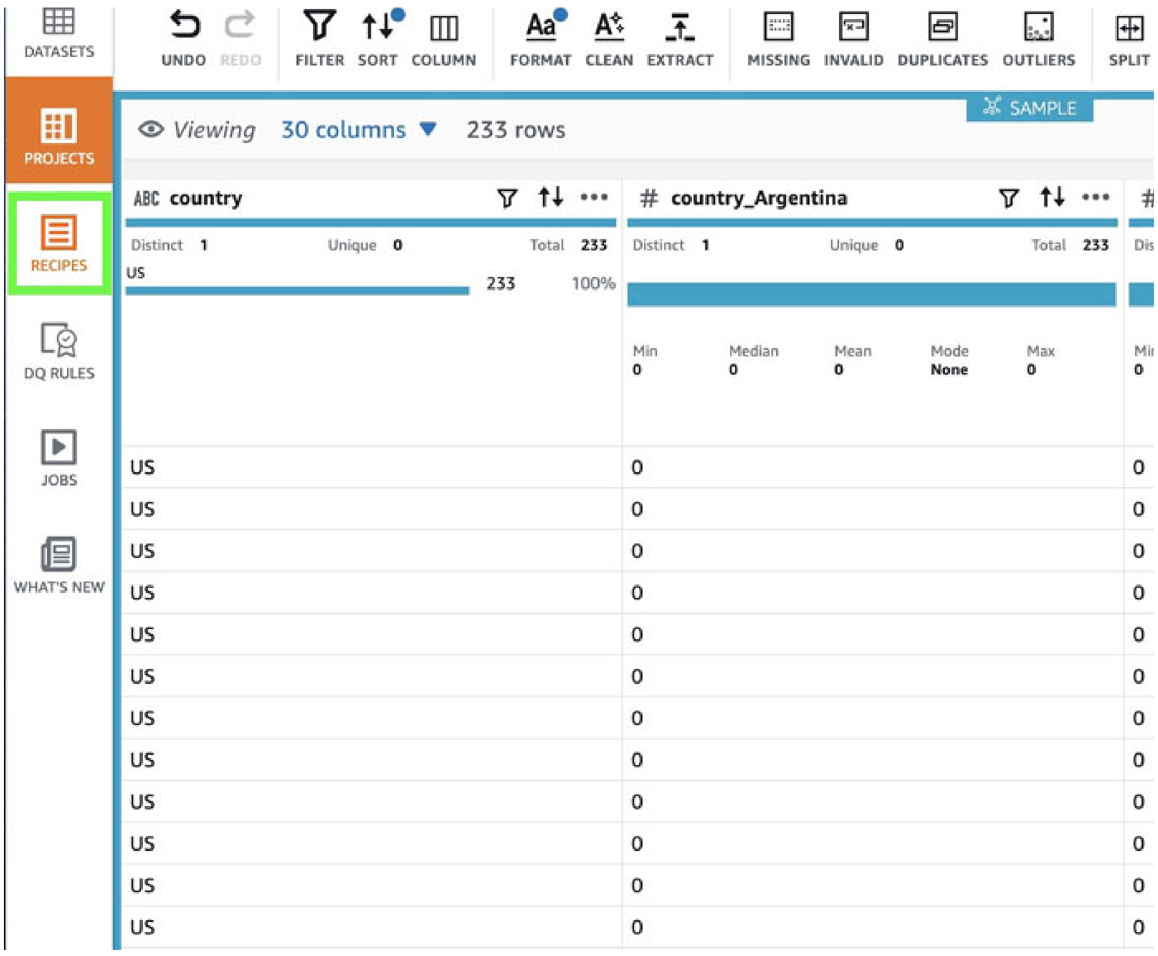


Figure 4.45: *Publish recipe*

16. In the **`Publish recipe`** window that appears, scroll down and click on the **`Publish`** button. Now click on the **`RECIPES`** option in the left pane. Refer to *Figure 4.46*:


wine-mag-project
Dataset: wine-magazine-test
Sample: First n sample (233 rows)

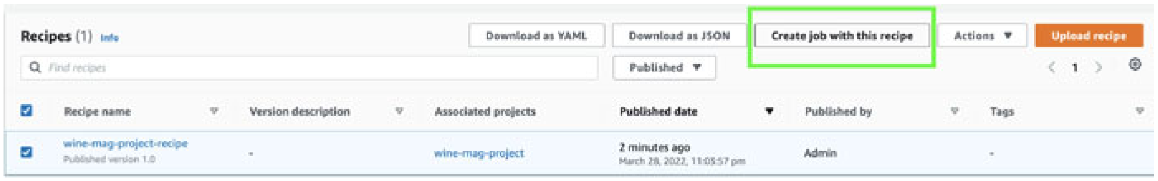


Viewing 30 columns ▼ 233 rows

ABC country	# country_Argentina
Distinct 1 Unique 0 Total 233	Distinct 1 Unique 0 Total 233
US 233 100%	Min 0 Median 0 Mean 0 Mode None Max 0
US	0
US	0
US	0
US	0
US	0
US	0
US	0
US	0
US	0
US	0
US	0
US	0

Figure 4.46: *Navigate to Recipes page*

17. We can see that our newly created recipe is automatically available in this page. Click on the check box next to our recipe name to select it, and click on **`Create job with this recipe`** in the top-right corner of the page. Refer to *Figure 4.47*:

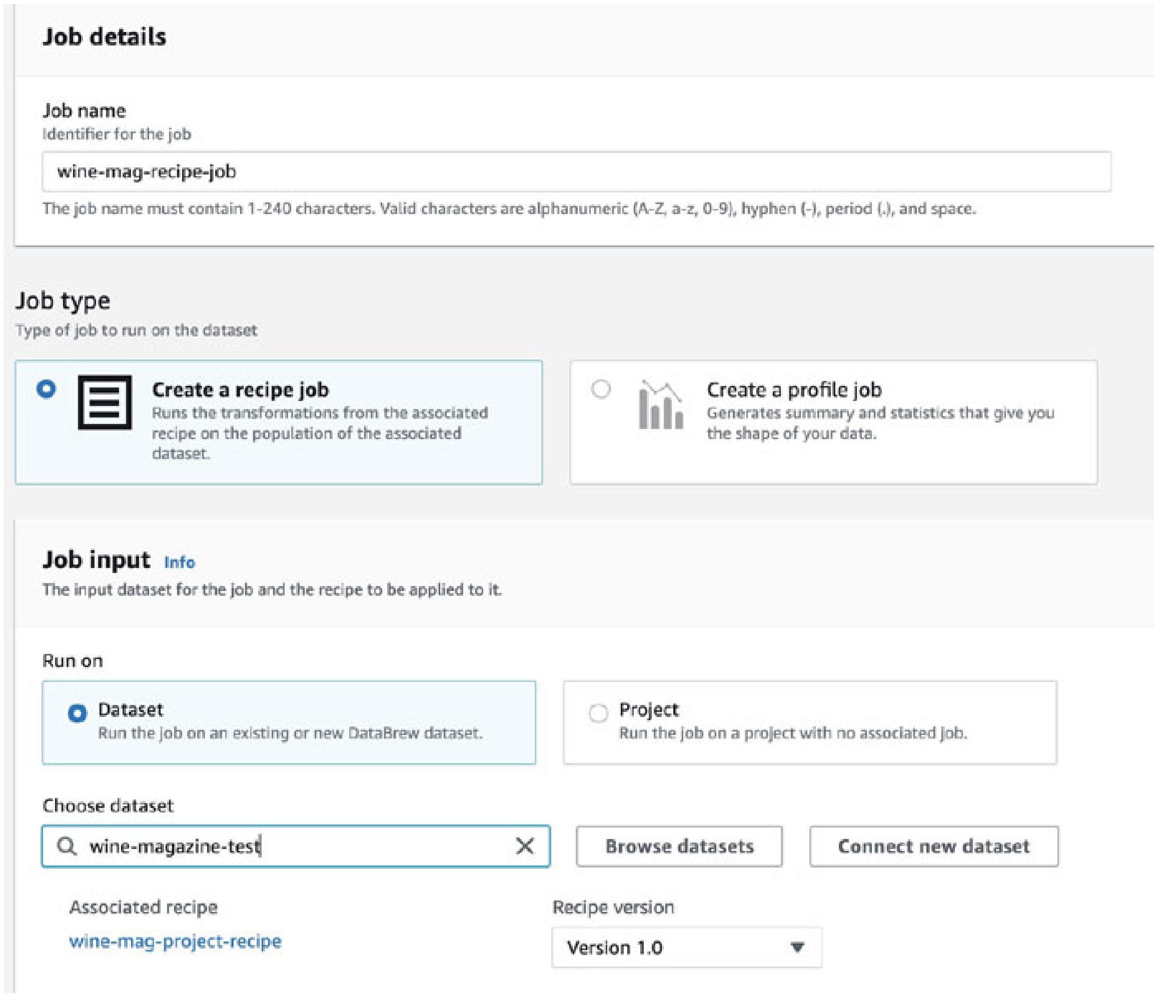


☑	Recipe name	Version description	Associated projects	Published date	Published by	Tags
☑	wine-mag-project-recipe Published version 1.0	-	wine-mag-project	2 minutes ago March 28, 2022, 11:03:57 pm	Admin	-

Figure 4.47: *Create job with recipe*

18. Provide a name for the job under **`Job name`**, leave **`Create a recipe job`** selected, and in **`Choose dataset`**, type or select the name of the dataset you

created at the beginning of this section, as shown in the following image:



Job details

Job name
Identifier for the job
wine-mag-recipe-job
The job name must contain 1-240 characters. Valid characters are alphanumeric (A-Z, a-z, 0-9), hyphen (-), period (.), and space.

Job type
Type of job to run on the dataset

[✓ Create a recipe job] Runs the transformations from the associated recipe on the population of the associated dataset.
Create a profile job — Generates summary and statistics that give you the shape of your data.

Job input Info
The input dataset for the job and the recipe to be applied to it.

Run on
[✓ Dataset] Run the job on an existing or new DataBrew dataset.
Project — Run the job on a project with no associated job.

Choose dataset
wine-magazine-test

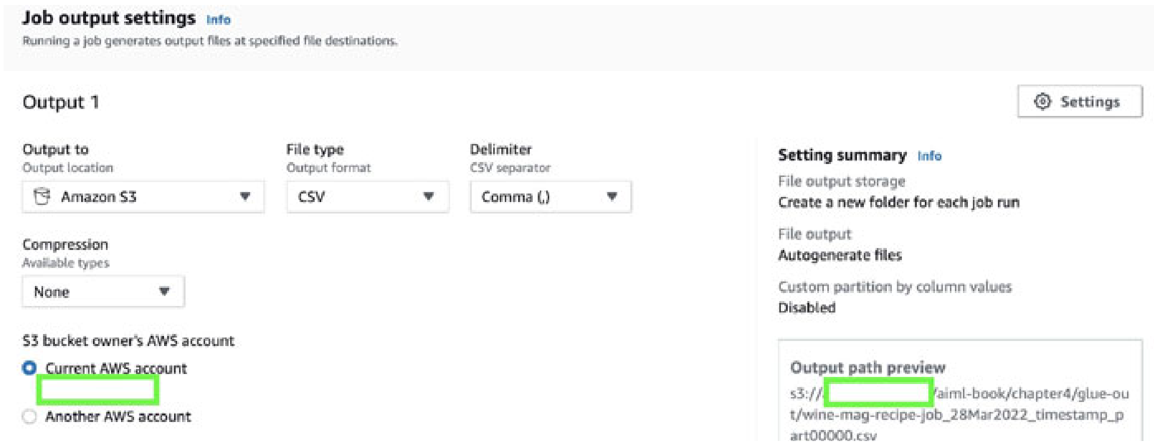
Browse datasets | Connect new dataset

Associated recipe
wine-mag-project-recipe

Recipe version
Version 1.0

Figure 4.48: *Provide inputs to create a recipe job*

19. Scroll down to **`Job output settings`** and type the S3 location as **`s3://<your-bucket>/aiml-book/chapter4/glue-out/`**, as shown in the following image:



Job output settings Info
Running a job generates output files at specified file destinations.

Output 1

Output to
Output location
Amazon S3

File type
Output format
CSV

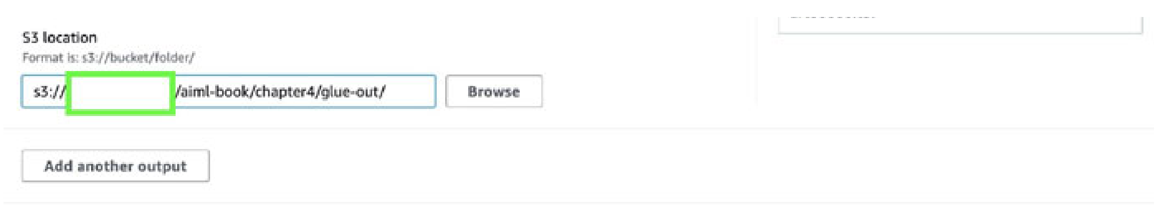
Delimiter
CSV separator
Comma (,)

Compression
Available types
None

S3 bucket owner's AWS account
[✓ Current AWS account]
Another AWS account

Setting summary Info
File output storage
Create a new folder for each job run
File output
Autogenerate files
Custom partition by column values
Disabled

Output path preview
s3://[redacted]/aiml-book/chapter4/glue-out/wine-mag-recipe-job_28Mar2022_timestamp_part00000.csv

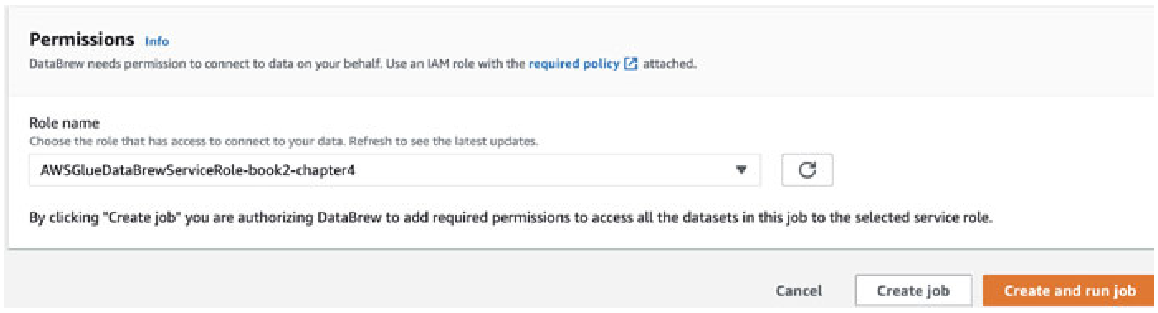

S3 location

Format is: s3://bucket/folder/

s3:// /aiml-book/chapter4/glue-out/

Figure 4.49: Provide job output settings

20. Scroll down to **`Permissions`** and select the IAM role you created earlier in this section. Refer to [*Figure 4.50*]():


Permissions

DataBrew needs permission to connect to data on your behalf. Use an IAM role with the required policy attached.

Role name

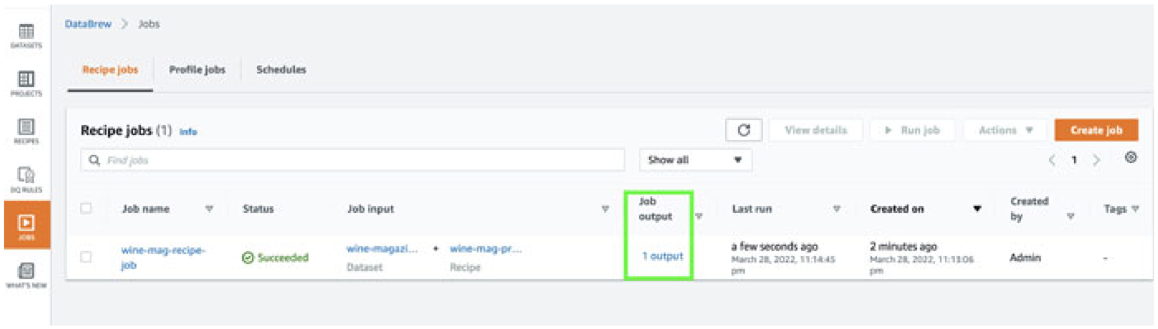
Choose the role that has access to connect to your data. Refresh to see the latest updates.

AWSGlueDataBrewServiceRole-book2-chapter4

By clicking "Create job" you are authorizing DataBrew to add required permissions to access all the datasets in this job to the selected service role.

Figure 4.50: Select the DataBrew IAM role

21. Click on **`Create and run job`**. This job will automatically run the tasks in the recipe against the input dataset and upload the output to the S3 bucket location we provided. To fully automate this step programmatically, you can use the Glue DataBrew API (**https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/databrew.html#GlueDataBrew.Client.create_recipe_job**) to start a job using the recipe you created.
22. Once the job completes, you can see this appearing in the **`Jobs page`** under **`Recipe jobs`**. Click on the link under the **`Job output`** to go to the S3 location and review the processed dataset that was created, as shown in the following image:


DataBrew > Jobs

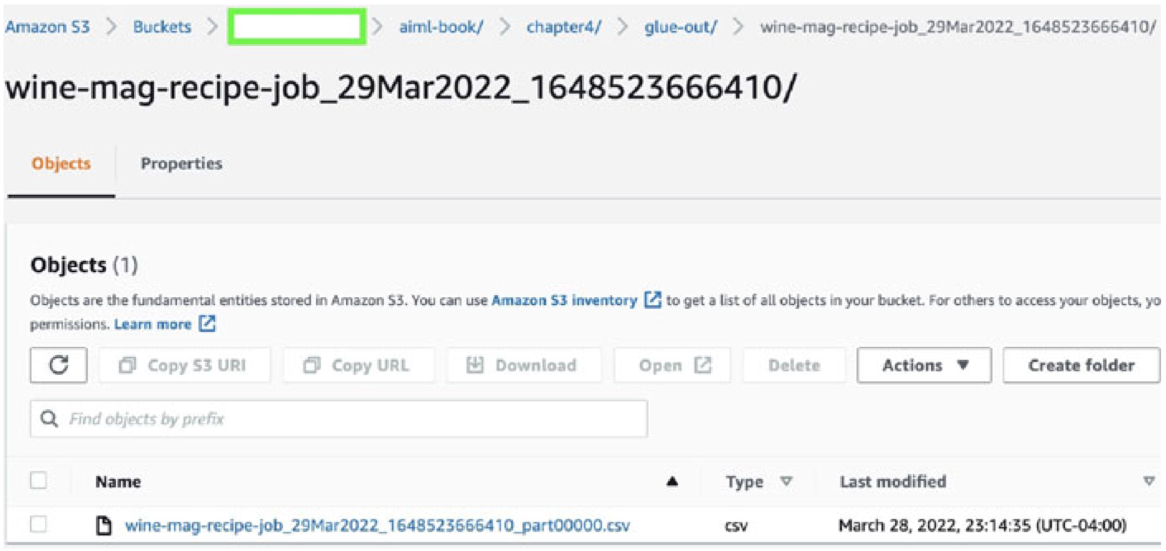
Recipe jobs | Profile jobs | Schedules

Recipe jobs (1) Info

	Job name	Status	Job input		Job output	Last run	Created on	Created by	Tags
	wine-mag-recipe-job	Succeeded	wine-magazi... Dataset	wine-mag-pr... Recipe	1 output	a few seconds ago March 28, 2022, 11:14:45 pm	2 minutes ago March 28, 2022, 11:13:06 pm	Admin	-

Figure 4.51: *Review the job run and navigate to job output*

23. When you navigate to the S3 location available in the **`Job output`** link, you can see that a new CSV file has appeared in the S3 bucket, as shown in the following image:


Amazon S3 > Buckets > [redacted] > aiml-book/ > chapter4/ > glue-out/ > wine-mag-recipe-job_29Mar2022_1648523666410/

wine-mag-recipe-job_29Mar2022_1648523666410/

Objects | Properties

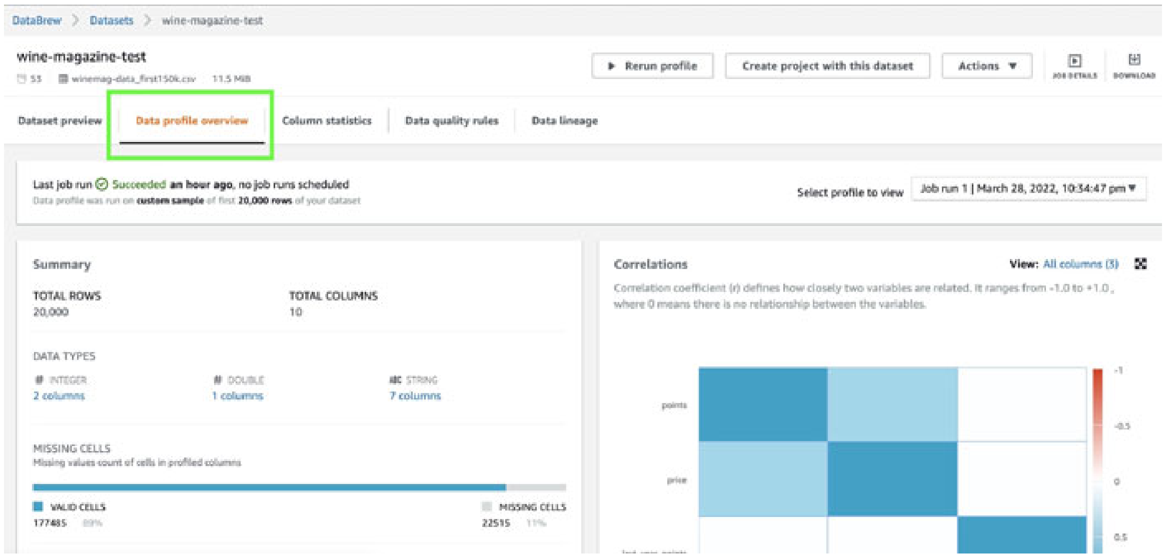
Objects (1)

Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, yo permissions. Learn more

	Name	Type	Last modified
	wine-mag-recipe-job_29Mar2022_1648523666410_part00000.csv	csv	March 28, 2022, 23:14:35 (UTC-04:00)

Figure 4.52: *Processed dataset appears in the S3 bucket*

This is how you use Glue DataBrew to create **`Recipe Jobs`** to run your feature engineering tasks. Let us now go back and review the results of the **`Data Profiling`** run we were submitting earlier when we came to the Glue DataBrew console. Click on **`DATASETS`** in the left pane, and in the **`Datasets`** page, click on **`View data profile`** next to your dataset name. This brings up the dataset profile page and shows the results of the profiling run. Refer to *Figure 4.53*:


DataBrew > Datasets > wine-magazine-test

wine-magazine-test
53 winemag-data_first150k.csv 11.5 MiB

Last job run Succeeded an hour ago, no job runs scheduled
Data profile was run on custom sample of first 20,000 rows of your dataset

Select profile to view: Job run 1 | March 28, 2022, 10:34:47 pm

Summary

TOTAL ROWS	TOTAL COLUMNS
20,000	10

DATA TYPES

INTEGER	DOUBLE	STRING
2 columns	1 columns	7 columns

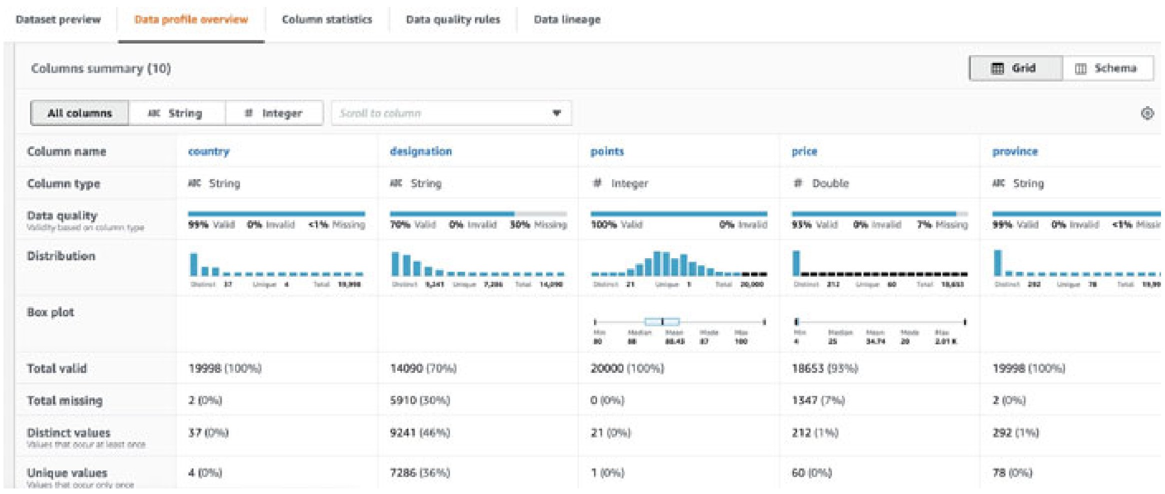
MISSING CELLS
Missing values count of cells in profiled columns

VALID CELLS	MISSING CELLS
177485 89%	22515 11%

Correlations — View: All columns (3)
Correlation coefficient (r) defines how closely two variables are related. It ranges from -1.0 to +1.0, where 0 means there is no relationship between the variables.

Figure 4.53: *Review results of the data profiling job*

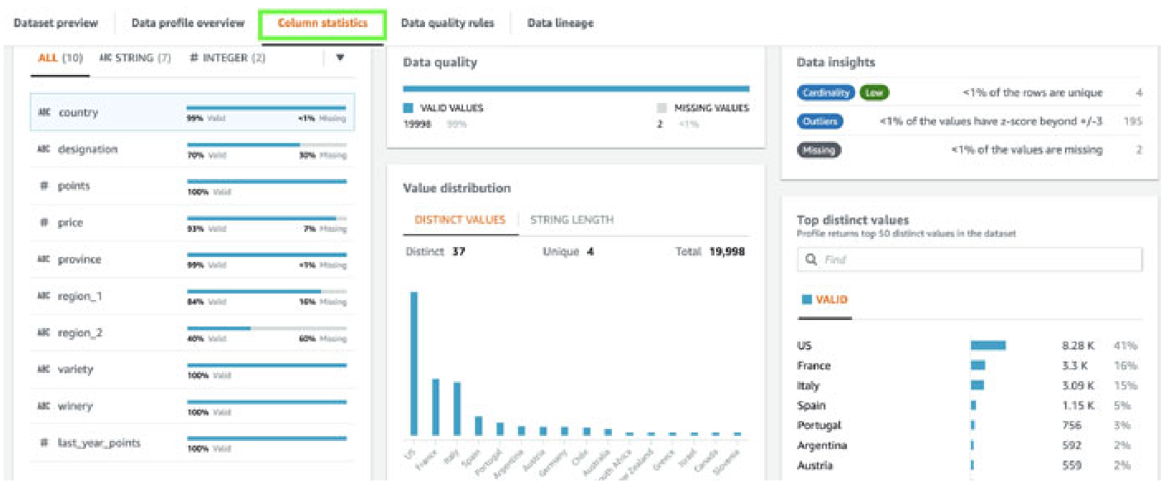
You can see row and column summaries, along with correlation matrices for the numerical columns in the dataset to begin with. Scroll down this page to view summary metrics for each column in the dataset under the **`Column Summary`** section. Refer to *Figure 4.54*:


Columns summary (10)

Column name	country	designation	points	price	province
Column type	String	String	Integer	Double	String
Data quality Validity based on column type	99% Valid 0% Invalid <1% Missing	70% Valid 0% Invalid 30% Missing	100% Valid 0% Invalid	93% Valid 0% Invalid 7% Missing	99% Valid 0% Invalid <1% Missing
Distribution	Distinct 37 Unique 4 Total 19,998	Distinct 9,241 Unique 7,286 Total 14,090	Distinct 21 Unique 1 Total 20,000	Distinct 212 Unique 60 Total 18,653	Distinct 292 Unique 78 Total 19,998
Box plot			Min 80 Median 88 Mean 88.43 Mode 87 Max 100	Min 4 Median 25 Mean 34.74 Mode 20 Max 2.01 K	
Total valid	19998 (100%)	14090 (70%)	20000 (100%)	18653 (93%)	19998 (100%)
Total missing	2 (0%)	5910 (30%)	0 (0%)	1347 (7%)	2 (0%)
Distinct values Values that occur at least once	37 (0%)	9241 (46%)	21 (0%)	212 (1%)	292 (1%)
Unique values Values that occur only once	4 (0%)	7286 (36%)	1 (0%)	60 (0%)	78 (0%)

Figure 4.54: *Review column summary*

Click on the **`Column statistics`** tab at the top to look at additional statistics and insights for each column. Refer to *Figure 4.55*:



country	99% Valid	<1% Missing
designation	70% Valid	30% Missing
points	100% Valid	
price	93% Valid	7% Missing
province	99% Valid	<1% Missing
region_1	84% Valid	16% Missing
region_2	40% Valid	60% Missing
variety	100% Valid	
winery	100% Valid	
last_year_points	100% Valid	

VALID VALUES	MISSING VALUES
19998 99%	2 <1%

Distinct 37	Unique 4	Total 19,998

Cardinality Low	<1% of the rows are unique	4
Outliers	<1% of the values have z-score beyond +/-3	195
Missing	<1% of the values are missing	2

US	8.28 K	41%
France	3.3 K	16%
Italy	3.09 K	15%
Spain	1.15 K	5%
Portugal	756	3%
Argentina	592	2%
Austria	559	2%

Figure 4.55: *Review column statistics*

You can also run a **`Data quality job`** to generate detailed data quality analyses for your entire dataset. You can do this to get a closer look at your data as per your requirements. Finally, click on the **`Data lineage`** tab in the top pane to display what processes caused changes to our dataset, as shown in the following image:

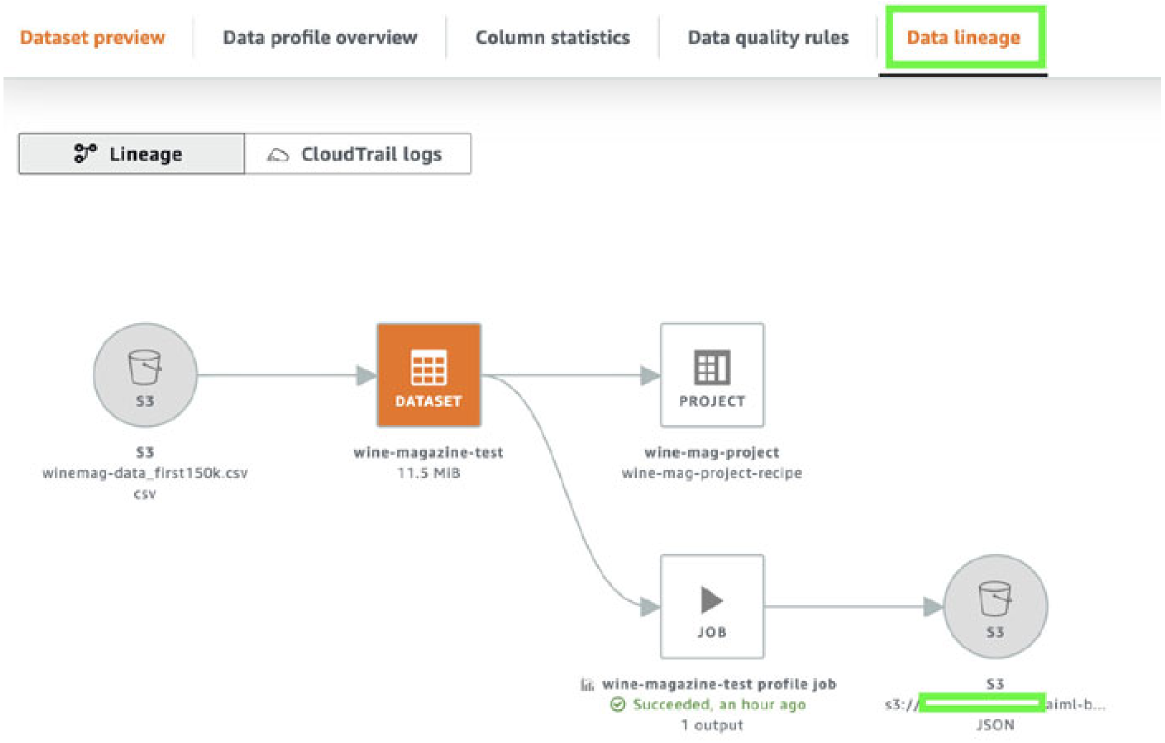


Figure 4.56: *Review data lineage*

And with that we come to the end of this section and this chapter. Glue DataBrew is a powerful tool in our experience and can provide you with a lot of information that is essential for understanding your data better. Remember that your data will determine how well your model can learn, so this type of detailed analysis is expected and required.

Conclusion

Phew!! This was a long chapter, and we did cover quite a bit of ground in this one. Take note that by the time you complete the solution build for this chapter, you will be well on your way to becoming a Cloud-Native AI/ML expert. You have now learned how to identify, source, collect, prepare, pre-process, feature engineer, analyse and transform your datasets for ML.

In the next chapter, we will dive deep into algorithms and neural networks. We will learn the different types of ML algorithms available, discuss common algorithms, talk about how to architect a neural network, and learn hands-on training models using algorithms and neural networks.

Points to Remember

Make a note of these key points with respect to data orchestration that we learned in this chapter:

- There are multiple ways in which you can build data orchestration pipelines for ML training and inference.
- We learned in this book a couple of options we have for building data orchestrations.
- Over the last couple of years, Amazon SageMaker (**https://aws.amazon.com/sagemaker/**), a fully managed service for building end-to-end ML workflows, added several new features that help integrate ML pipelines with the data ecosystem.
- One such feature is the Amazon SageMaker Data Wrangler, which has numerous pre-built transformations for standard feature engineering and data pre-processing tasks.
- Data Wrangler provides hundreds of pre-built transforms, and you can also write custom transformation maps using popular languages.
- Another highly sought-after launch was the Amazon SageMaker Feature Store, a managed service that provides a reusable repository for storing, maintaining, and accessing ML features across different ML projects.

- The purpose-built features within AWS AI/ML that help enable end-to-end automation and scaling across data sources, ML models and consumers of these models is one of the biggest advantages of the cloud computing model provided by AWS.

Multiple Choice Questions

Use these questions to challenge your knowledge of what we learned in this chapter.

1. **What is the right combination of data interaction types and modes we discussed in this chapter? (Select one or more right options).**
    a. Batch and synchronous
    b. Real-time and asynchronous
    c. Real-time and fan-out
    d. Batch and asynchronous
2. **Which of the following choices is NOT a true statement about the synchronous data interaction mode?**
    a. It is used for data flows between two systems (A and B), with A requesting B for information by either passing or not passing a parameter in its request.
    b. A sending data or messages via a queue to B and waiting for a response from B.
    c. It is primarily used for real-time calls between a service and its consumer.
    d. A sending messages or data to a topic that multiple consumers have subscribed to.
3. **AWS Glue supports big data processing and ETL tasks.**
    a. True
    b. False
4. **Which of these options about AWS Glue is NOT true?**
    a. AWS Glue is a serverless, managed analytics service to create an automated job for feature engineering tasks.
    b. An important task in setting up an AWS Glue ETL job is to select the right compute we need to properly run our jobs.

   c. AWS Glue Studio provides an easy way to author and manage ETL jobs.
   d. AWS Glue provides support for a number of languages, including Python, to author ETL jobs.

5. **You cannot run Min-Max normalization with AWS Glue DataBrew, but you can run the rest of the feature engineering tasks.**
   a. True
   b. False

Answers

1. **b,c,d**
2. **d**
3. **a**
4. **b**
5. **b**

Further Reading

- **https://www.space.com/33527-how-fast-is-earth-moving.html#:~:text=The%20sun%20and%20the%20solar,way%20around%20the%20Milky%20Way**
- Using SageMaker Data Wrangler and SageMaker Feature store together **https://catalog.us-east-1.prod.workshops.aws/workshops/63069e26-921c-4ce1-9cc7-dd882ff62575/en-US/lab1/option1**

CHAPTER 5

Casting a Deeper Net (Algorithms and Neural Networks)

Introduction

The author is a self-proclaimed fan of superheroes, especially Iron Man (what a performance by Robert Downey Jr.!). If you explore the offerings from the Marvel Cinematic Universe (MCU), a majority of the heroes and villains are either altered by a super serum or mutation, or they are gods or aliens. Only a handful of them are humans with extraordinary potential, and Iron Man is one of them. What makes Iron Man special is his genius and his flying suit powered by an AI called Jarvis. With capabilities like automatic speech recognition, natural language understanding, flight control, autopilot, navigation, weapons command and control, threat assessment, and strategic/tactical improvisation, Jarvis was versatile, and the interactions between Tony Stark (Iron Man) and Jarvis during the fight sequences were a treat to watch. The suit's power source was an alien technology, but the intelligence that made these tasks or capabilities possible was a collection of algorithms. Tony designs these modules with algorithms in the first Iron Man movie, and new capabilities are added as requirements evolve (remote suit control in Avengers, for example). Metaphorically speaking, algorithms can be thought of as the brains of an AI system. An AI's intelligence is measured by the efficacy with which it can learn and perform tasks to achieve a desired outcome; so, algorithms are at the core of an AI system's usefulness.

Consider, for example, an AI application that autosuggests words as you type a sentence. An algorithm is selected and trained against a large collection of documents to create a model that predicts the next words as the previous set of words are passed as input. The algorithm contains the computational logic and semantics to determine how to tokenize the words in the corpus to numeric vectors, how to embed the vectors in a latent space such that relationships between words and the sentence context are captured, and how to approximate a function to predict the next word a result of the learning process. In effect, an algorithm is a set of mathematical equations that work together to minimize the differential between actual vs. predicted values in the data iteratively (supervised learning), identify logical groupings or patterns in the data (unsupervised learning), and determine an action based on incentives (reinforcement learning). Thankfully, due to the increasing demand for ML and AI, and advancements in cloud computing making it more accessible, we can now work with algorithms without having to worry about getting a PhD in Mathematics. We can leave the math and science to the scientists/researchers and use abstracted APIs in popular programming languages, made available via ML frameworks, to select the algorithm of choice for our needs to set up ML training. With AWS AI services, we have an even greater level of abstraction with using APIs to directly embed AI into our applications without the need for ML skills or model training. We will cover the AI services in detail in *Chapter 10, Adding Intelligence With Sensory Cognition*, and *Chapter 11, I for Industrial Automation*.

In the previous chapters, we set the stage by introducing AI and ML, walking you through what the ML workflow looks like, why and how to set up data lake storage in AWS, how to ingest data into your data lake, how to run data processing tasks such as feature engineering, and how to automate data orchestration pipelines for your ML workloads using AWS Glue ETL and AWS Glue DataBrew. These chapters helped you learn how to work with your data, and how to get it prepared and ready for ML training. The next set of tasks correspond to the model training phase of the ML workflow, as shown in the following image:

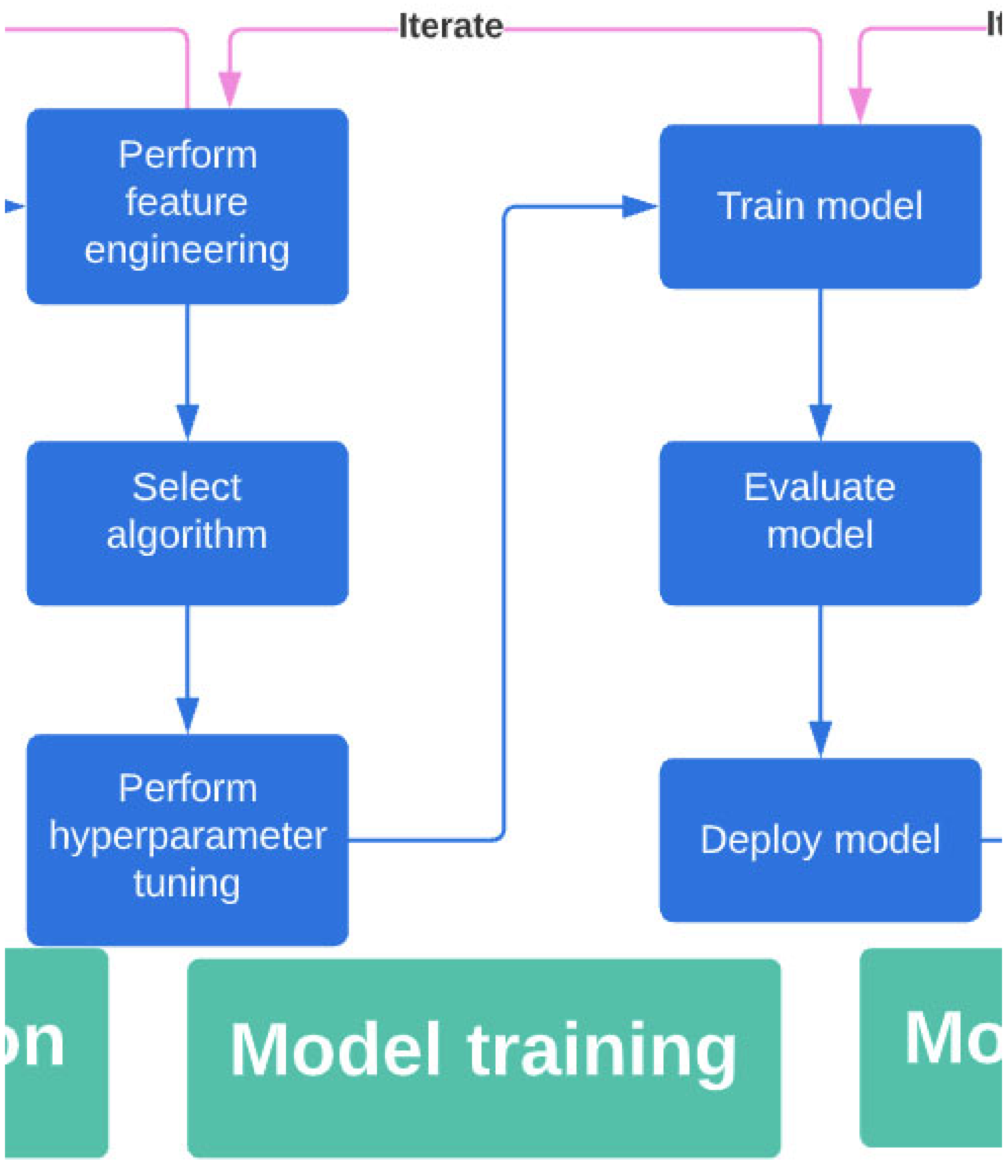


Figure 5.1: *Model training phase of the ML workflow*

In this chapter, we will learn what algorithms and neural networks are, learn what to use when, and use Python code samples to understand how to set up estimators and fit models for different types of ML use cases.

Structure

In this chapter, we will dive deep into the following topics:

- Introducing Algorithms and Neural networks
- Simplifying the Algorithms vs. Neural networks conundrum

- Building ML solutions with Algorithms and Neural networks

Objectives

So, why do we have a chapter on algorithms? Why not directly jump to model training and be done with it? If our objective was to just show you how to build and run a ML project, we probably would have. But, in this book we want to inspire you with the art of the possible. We want you to experience the breadth and depth of solution options that AI/ML encompasses today. We would like you to learn first-hand how some of the biggest challenges in the world are solved using AWS AI/ML; that's why we designed this chapter to strengthen your knowledge in algorithms and neural networks, which is the core of ML. In this chapter, we will first learn how algorithms came to be, how they evolved, how neural networks were born, and how complex problems are solved today by large-scale ML using a technology called deep learning. We will then review a matrix of choices to navigate to a specific algorithm based on the use case, building on top of what we learned in Chapter 1, Introducing the ML Workflow. Finally, we will get hands-on experience and play with popular algorithms and neural networks, setting up estimators and running basic training tasks.

Introducing Algorithms and Neural networks

The history of algorithms predates the history of computers. According to the Online Etymology Dictionary (**https://www.etymonline.com/word/algorithm**), the word algorithm is a confluence of words from the Arabic, French, Greek and Latin languages that all refer to numbers and computation. The word dates back to the 17th century, and the meaning did not change much until the 20th century (the era of computers) when it became widely used and accepted. In essence, it refers to a set of mathematical functions that follow a logical sequence to perform a particular task. That's great, but what does it mean really? And why did it take on a new form in the context of computers? At its root, the idea is the simplification of complex computations. Inventions and discoveries are closely tied to the evolution of who we are as human beings. Some of our biggest scientific and engineering breakthroughs in the past few centuries have mathematics at their core, and solving the math is an important part of building the invention. But these are extremely complex tasks, are computation intensive and cannot be solved manually. So, while inventing things that transform our lives, we have also been trying to invent tools that help us calculate easier and faster.

Deterministic Algorithms

The Abacus is one such tool. It is not clearly known who invented it and when. Ancient texts refer to the Abacus as early as 2500 BC, and it has been mentioned in texts from many parts of the world, even as recent as the 17th century. It was quite popular (and some still like to use it even today), until it was replaced by the sliding rule, and then recently, the faster calculator and computers. For a detailed read on the history of Abacus and the sliding rule, refer to **https://www.sliderulemuseum.com/Abaci.htm**. The Abacus can perform foundational arithmetic operations like add, subtract, divide multiply, square root and cubic root. The design is simply a wooden frame, with wires/thin rods running vertically or horizontally, with beads attached to them. The beads can be moved along the wire/rod, the position and count of beads indicate the calculation and the results of an operation. Surprisingly, the Abacus can perform these calculations for large numbers running up to the billions. Today, the Abacus is used in primary schools to teach young children how to calculate in a fun way and is also used to teach the visually impaired because of the touch-based learning.

If (hopefully, this never happens) our sun were to emit an electro-magnetic pulse that is powerful enough to disable all the electronics in the world, the following is how an algorithm would look like because we would have nothing else but our trusted Abacus or a slide rule to help with a solution.

Problem: Write an algorithm to determine how long it will take to travel from Earth to Mars in light speed.

Algorithm:

1. Light travels at the speed of 30,000 kilometers per second.
2. Light speed equals to 30,000 multiplied by 3600 per hour.
3. The distance between Earth and Mars is 225M kilometers; the time it takes to travel from Earth to Mars in light speed is the result of *Step 3* divided by result of *Step 2*.

This may seem like oversimplification, but remember, the whole idea of an algorithm is to break complex calculations into simple logical sequence of steps to arrive at an outcome, given an input. The preceding example is a classic deterministic algorithm. As the word indicates, deterministic means the algorithm's course is pre-determined, or for humour's sake, the algorithm follows its destiny. There is nothing outside the bounds of the algorithm that can influence its direction or outcome, and it will consistently produce the same outcome given the same inputs. The algorithm can be conditional, of course, but conditions and actions have to be sequenced into its structure. Consider, for example, that you walk up to your bank's **Automated Teller Machine** (or **ATM**) to withdraw cash, and you insert your card. The machine reads the magnetic stripe on your card, determines your account details, and when you enter your PIN, verifies that the PIN you entered is the same as what is set up for your card. If the PIN is correct, it presents you the main menu, and you can perform a number of transactions, including withdrawing money. If we visualize an algorithm that defines the cash withdrawal flow, it will be similar to the following image:

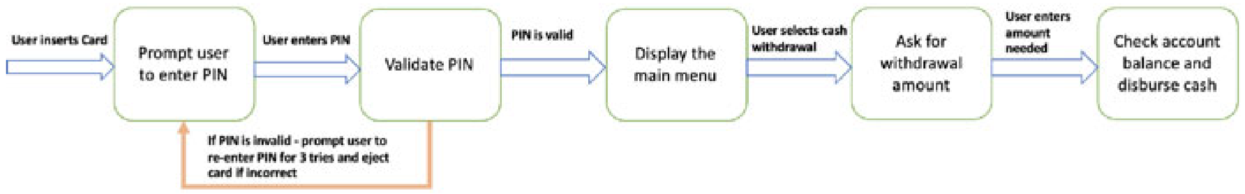
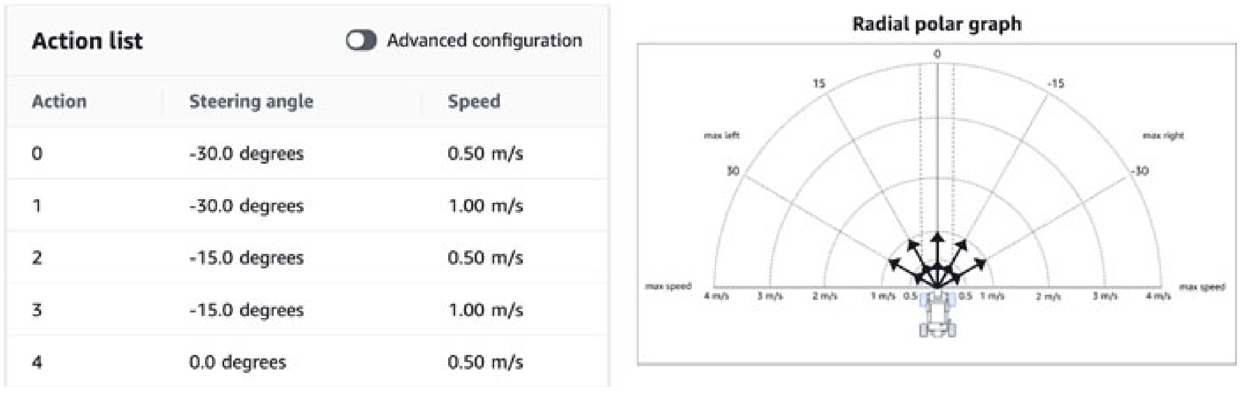

User inserts Card
Prompt user to enter PIN
User enters PIN
Validate PIN
PIN is valid
Display the main menu
User selects cash withdrawal
Ask for withdrawal amount
User enters amount needed
Check account balance and disburse cash
If PIN is invalid - prompt user to re-enter PIN for 3 tries and eject card if incorrect

Figure 5.2: *Example of a deterministic algorithm*

In this case, as long as the inputs to the algorithm remain the same, the processing and outputs will remain consistent. This also means that the algorithm has no attention span or memory and will not learn from the data it processes. So, the algorithm will not be able to differentiate between a genuine user and a fraudulent user if the right process steps are followed.

Probabilistic Algorithms

The other type of algorithm that's equally popular, especially in AI and ML, is probabilistic. In contrast to deterministic, an algorithm that can derive a generalization or approximation function can be termed probabilistic. In essence, there is a randomness at play, which can influence an outcome in either direction, but this same random nature gives way to assimilation by iteration. Hence, probabilistic algorithms are well suited for ML. A classic example is reinforcement learning (**https://docs.aws.amazon.com/sagemaker/latest/dg/reinforcement-learning.html**), in which an agent learns by taking actions while interacting with an environment and being rewarded if the action leads to the goal in the quickest possible way. At AWS, the DeepRacer (**https://docs.aws.amazon.com/deepracer/latest/developerguide/what-is-deepracer.html**) is a fun implementation of reinforcement learning to make learning ML hands-on and exciting. The action space determines the possible actions the agent can take in the environment, which is the outcome of the learning process during training, with the feedback provided by the reward function. The agent can take one action from a list of possible actions (in the discrete action space) or select an action within specific attribute bounds (in the continuous action space). Either way, the actions are probabilistic, and the combination of rewards and the iteration drive the learning aspect of the model. Refer to *figure 5.3*:


Action list

Advanced configuration

Action	Steering angle	Speed
0	-30.0 degrees	0.50 m/s
1	-30.0 degrees	1.00 m/s
2	-15.0 degrees	0.50 m/s
3	-15.0 degrees	1.00 m/s
4	0.0 degrees	0.50 m/s

Figure 5.3: *Discrete action space for AWS DeepRacer*

An algorithm, in its natural state, is just words and numbers, purely theoretical. It comes to life when its semantics are converted to instructions and executed with tools like the Abacus, slide rule, the calculator and the computer. In simple terms, and use case or a problem explains the "why?", the algorithm is an answer to the "what?", and the tools help with the "how?". So, the algorithm is the blueprint of the solution or the task performed to arrive at a desired result. When we use computers to do something, the first step is to create an algorithm for the task; then, we write programs in a language of our choice containing instructions codifying the algorithm. Today, we have ready-made programs and apps for most of our tasks, but remember that behind every successful program, there is an algorithm following its destiny (deterministic) or making a new one (probabilistic). And hopefully, this helps you understand how and why algorithms became so important in the computing era in the last century or so.

Decision Trees

At the beginning of this chapter, we discussed that an algorithm is the brain of a computer. This is more so in the case of ML, where we train computers to learn approximations to predict an output given an input. ML algorithms define the learning path of the prediction function, and during training, the mathematical equations comprising the algorithm are applied against the training dataset to enable the creation of a model that understands the rules, leading to an output based on an input, thereby becoming intelligent. Compare and contrast this with a deterministic algorithm that does not involve learning or modelling but an atomic execution of pre-determined instructions. We will cover the different types of ML algorithms in the next section, but let us briefly talk about the most popular algorithm family in supervised learning: decision trees. We covered decision trees briefly in *Chapter 1, Introducing the ML Workflow*, and the premise is using mathematics to build a tree from the training dataset, which lays out the decision rules that the model learns and then uses it to make predictions from inputs it has not seen before. The most common form of decision trees are the classification and regression trees or CART, and as the name implies, we use CART for classification (to predict a class or category) or regression (to predict a value) problems. Refer to *figure 5.4*:

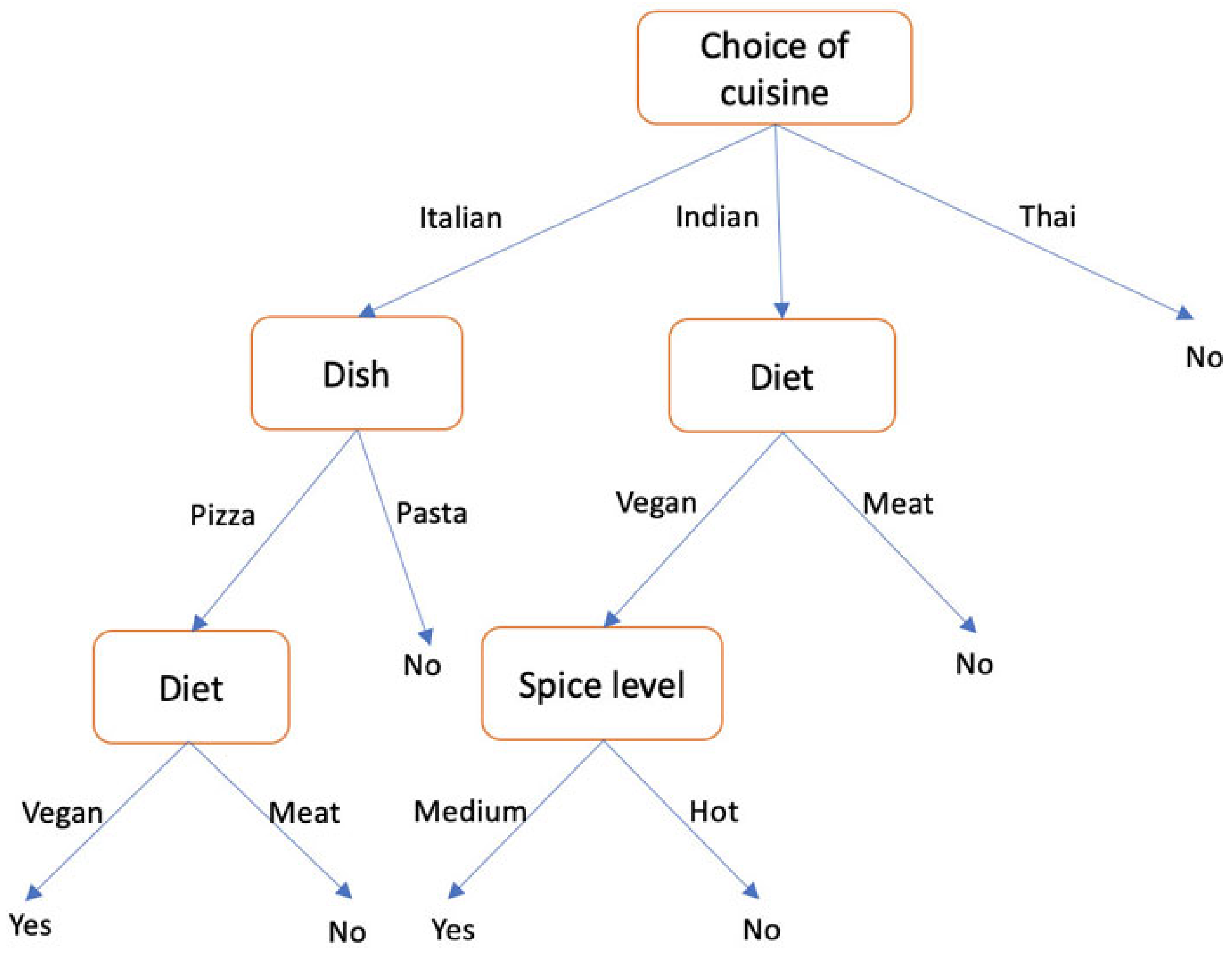


Figure 5.4: *Decision tree*

During training, the labeled input dataset (yes, decision trees use supervised learning) is fed to the algorithm for a tree representing the training dataset to be built. The most common mathematical functions used in building decision trees for classification problems are Entropy, Information gain and Gini Impurity, and that for regression is Reduction in Variance. These functions are akin to the DNA for a decision tree and determine how a tree is structured, how many nodes it can have, what are these nodes from the feature set, how many branches it can have, what are the branches, and finally, how many leaf nodes it terminates with. For additional reading, visit this link: **https://scikit-learn.org/stable/modules/tree.html**.

Entropy and Information gain

At a high level, entropy is a measure of how chaotic or disorderly the data distribution is across classes for a classification use case. Generally, it is a value between 0 and 1, but it can be greater than 1 in some cases. The closer the value is to 0, the less chaotic the data distribution, and vice versa. If entropy is biased toward either direction, it indicates that the dataset does not have adequate sampling of class distributions to enable approximation. If entropy strongly leans toward 0, it means that the model may overfit, and if it leans toward 1, it may underfit.

NOTE: Entropy, as discussed here, should not be confused with cross-entropy loss, which includes predicted class distributions and is a measure to determine how well or poorly a model has learned during training.

Typically, datasets need to be hierarchical for decision trees to be effective. There should be a main feature and dependent features that are related to the values of the main feature and cannot be used independently. Information gain is the difference between entropy of the main feature and the weighted average of entropy of the dependent features. By calculating the entropy and information gain of our tabular dataset, we can design a decision tree representation for our data iteratively. To understand how to do this, let us consider a tabular representation of our cuisine choices decision tree. The dataset looks as shown in the following table:

Cuisine choice	Dish	Diet	Spice level	Decision
Italian	Pizza	Vegan	Any	Yes
Italian	Pizza	Meat	Any	No
Italian	Pasta	Any	Any	No
Indian	Any	Vegan	Medium	Yes
Indian	Any	Vegan	Hot	No
Indian	Any	Meat	Any	No
Thai	Any	Any	Any	No

Table 5.1: *Cuisine choices dataset*

In our current example, it is obvious that some features are dependent on "Cuisine choice", such as "Dish" and "Diet", but when we have large datasets with hundreds of features and millions of rows, it is manually impossible to organize them into a decision tree; that's why we use these measures to automatically calculate and structure them. The approach is to first calculate the entropy of "Cuisine choice" based on the probability of each "Decision" of "Yes" or "No" as ultimately that is what we want the model to predict. The formula for entropy here is as follows:

Entropy = -(P("Yes")*log2P("Yes") + P("No")*log2P("No"))

Entropy = -((2/7)*log2(2/7) + (5/7)*log2(5/7))

Using a logarithm calculator (**https://www.rapidtables.com/calc/math/Log_Calculator.html**), we get the following results:

Entropy = -((0.285)*(-1.81) + (0.714)*(-0.486)) = -(-0.515 - 0.347) = 0.862

Next, we calculate the average of the weighted entropy for the three classes (or values) in "Cuisine choice". We use the same entropy formula as earlier but determine the P("Yes") and P("No") for each class in "Cuisine choice", such as "Italian", "Indian" and "Thai". For example, for "Italian", P("Yes") is 1/3 and P("No") is 2/3. We weigh the entropy for each class by its own probability. For example, the probability of a cuisine choice to be "Italian" in the dataset is 3/7. So, the overall formula here for average weighted entropy turns out to be as follows:

Probability of Italian * Entropy of Italian + Probability of Indian * Entropy of Indian + Probability of Thai * Entropy of Thai

[(3/7)*-(1/3*$\log_2$(1/3) + 2/3*$\log_2$(2/3))] + [(3/7)*-(1/3*$\log_2$(1/3) + 2/3*$\log_2$(2/3))] + [(1/7)*-(1/1*log2(1/1) + 0*$\log_2$(0))] = 0.389 + 0.389 + 0 = 0.778

Finally, information gain is the calculated difference between the two, which is as follows:

0.862 - 0.778 = .084**.**

To build our decision tree, we calculate the information gain for all the main features (that have dependent features) and select the feature with the highest information gain to be the root node for our tree. We then select each class for the root node, and repeat the entropy and information gain calculation (for each class, calculate its entropy and calculate the average weighted entropy for the features related to this class) for all dependent features; then, select the one with the highest information gain to be the next node in the tree. We repeat this until we have determined the leaf node (the tree terminates in the leaf node and will not have any further child nodes). We will now examine a different way to build the decision tree: a measure called Gini impurity.

Gini Impurity for decision trees

The other method to split decision trees for classification problems is Gini Impurity, which indicates the probability of wrong classification for a feature value, provided the dataset has been sampled randomly. It can have any value between 0 and 0.5, with 0 denoting no impurity or perfect classification or a pure feature, and 0.5 being totally impure. We use Gini impurity values to determine the nodes of a decision tree. The feature that has Gini impurity of 0 is a leaf node of a decision tree, and the feature with a lower Gini impurity among the rest of the main features will be the root node. We then calculate the Gini impurity for the dependent features for each main feature to understand which one will be the next node and so on. The formula for Gini impurity is as follows:

1 - SUM[(Probability of occurrence of a class instance i)2]

Reduction in variance for regression

We use a different measure to decide how to split a decision tree for regression problems, but we use this measure in the same way as classification problems, that is, the next node in a decision tree is selected based on the feature with the lowest variance. The variance we refer to here is the statistical variance of a feature in a dataset. We first calculate the variance of the main features and select the feature with the lowest variance as the root node; then, we calculate the variance of the dependent features, select the feature with the lowest variance to split the tree, and continue until we reach features with variance = 0, which is the leaf node. The formula for calculating the variance is as follows:

[SUM(feature value - mean of the feature value across the dataset)2/count of feature values in the dataset]

ML algorithms like decision trees are really good for use cases like predicting house prices, detecting fraudulent transactions, predicting whether a customer will buy a product, and other common regression or classification tasks. But what if we need a ML model to read and understand an image or a video, for example, in operating a self-driving car that can navigate through traffic or looking at tissue samples to detect whether a tumour is benign. Or let's say we want to build a model that can create a summary paragraph for each page in this chapter, automatically. These are examples of what is called **deep learning**, the ability to learn large numbers of parameters very comprehensively. If you recollect *Chapter 3, Predicting the Future With Features*, for the computer vision task, each pixel of the image was a feature, which, in turn, was a vector of colour coordinates and the position of the pixel itself. Images can have millions of pixels or even more, depending on the complexity of what is being conveyed. An image classification task may require thousands of images (or even more), and models need images of high quality for accurate predictions. Consequently, we need a much bigger, broader, and deeper algorithmic canvas to work with for models to learn the dataset adequately. This is where neural networks come in.

Neural Networks

We were introduced to neural networks in *Chapter 1, Introducing the ML Workflow*. We learned that a neural network is a programmatical and mathematical construct that mimics the function of human brain. We first define a neural network architecture for our ML training, which, at a high level, is a combination of the type of network (can be decided based on our ML problem), the number of layers in our network, the number of neurons in each layer, and the activation function to use for each layer. There is a lot of flexibility in how we set up our neural network, allowing extensive experimentation and iteration to meet our goal metric. The learning corresponds to how the values of the weights (randomly assigned coefficients) of each of the neurons evolve during the training process, especially during back propagation. Let us take an example to better understand these concepts, namely, our decision tree data to determine cuisine choices, but in the context of using a neural network to solve the same problem.

Let us denote each of the four input features as follows: F1=cuisine choice, F2=dish, F3=diet and F4=spice level, and the class label decision as O. Let us consider that each of our neurons is assigned an initial random weight (W), with weights in the first hidden layer denoted as WI and the weights in the second hidden layer denoted as WH, both of which will be updated during training, and a bias (b), which is initially set to 0 and will also be updated during the training process. We will use the ReLU (**https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html#relu**) activation functions for our hidden layers and the Sigmoid (**https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html#sigmoid**) activation function for our output layer to predict the probability of the decision being a Yes or a No. The sigmoid function is useful as a logistic regression function for classification problems. And for the sake of simplicity, let us call the neurons in our first hidden layer that receive the inputs IN1, IN2 and so on. Similarly, the neurons in the second hidden layer are designated as HN1 and HN2, and the output layer neuron is ON1.

NOTE: There are different schools of thought on whether the bias should be a constant value throughout the training or it should be assigned an initial value and expected to be updated during training, as you would a weight. In this example, let us assume that the bias is updated during the training process, during back propagation.

Our neural network architecture for predicting a decision class for this problem will look as shown in *Figure 5.5*:

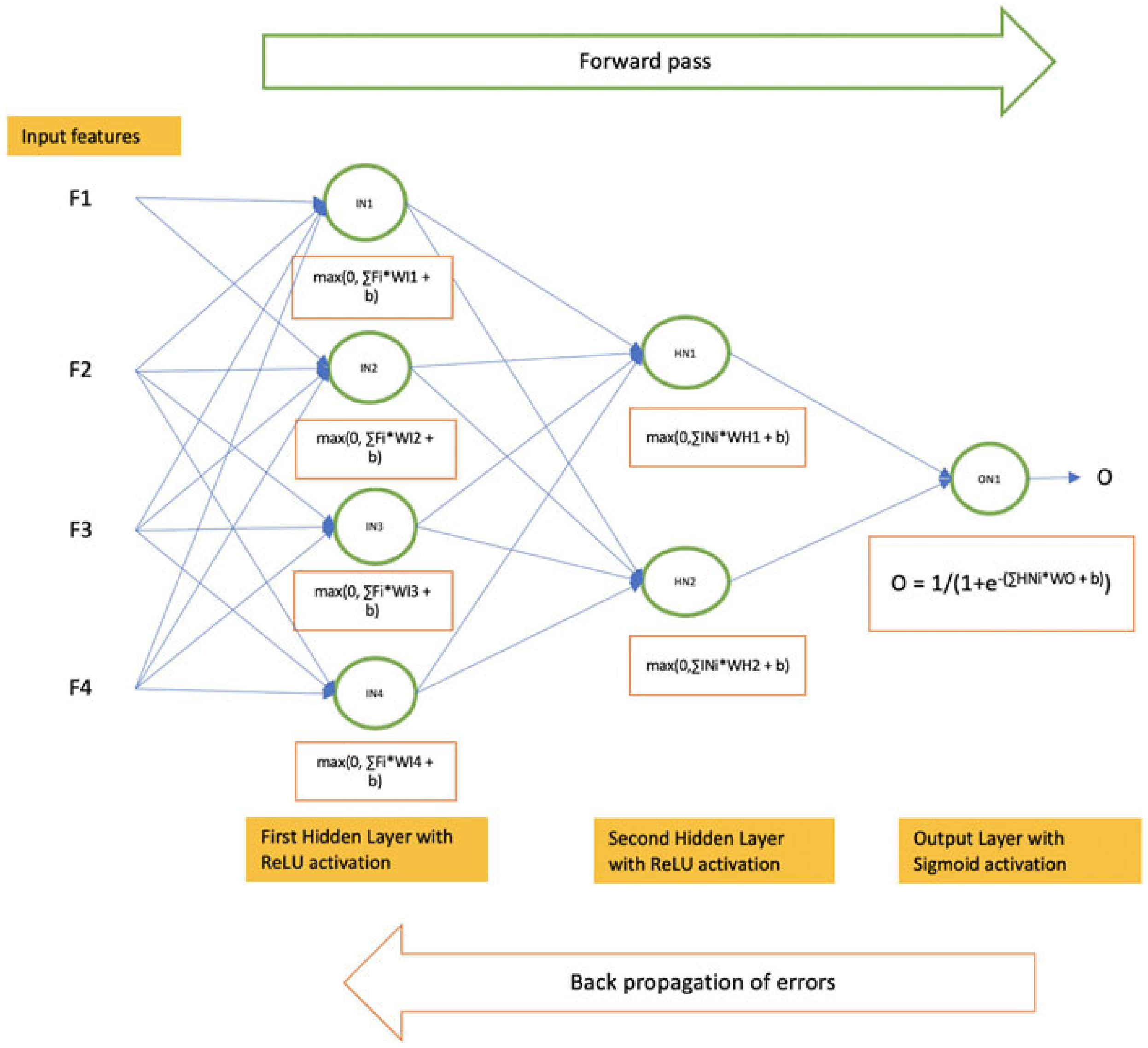


Figure 5.5: *Neural network architecture with logistic regression for binary classification*

NOTE: The 'e' represented in the Sigmoid function is a mathematical constant ~ 2.718 and is called Euler's number.

During the training process, the input features from our dataset will be fed to the neural network in samples. When the neural network has gone through one pass of our training dataset in its entirety, it is called an epoch. In this example, values for each observation (row) for each of the four input features are fed to each neuron in the first hidden layer using the sum product of the input feature values (Fi) and the weights assigned to the neurons (WI1, WI2…). The bias (b) is then added to this sum product. This calculated value is then passed as an input to the ReLU activation function, which returns the maximum value between 0 and the calculated input. If the calculated input is a negative value, ReLU returns 0, and if it is a positive value, ReLU returns the positive value. We represent ReLU for the first hidden layer as *max(0, ∑Fi*WIi + b)*. We repeat this process in the second hidden layer; however, this time, the inputs are not feature values but the activated value from each neuron in the preceding layer. We apply the ReLU once again, denoted by *max(0, ∑INi*WHi + b)*. We send both the activated values to the output layer, which calculates the sum product of the input values and the bias, and then passes it to a Sigmoid function that converts the calculated value to a probability. This is what happens during the forward pass of the neural network during training.

The output predicted by the neural network is then compared to the class label, and an error is estimated using a loss function like cross entropy (**https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html**) for classification problems or a mean square error (**https://keras.io/api/losses/regression_losses/**) for regression problems. We then use a different algorithm (with its own set of mathematical functions) called the back propagation of errors to calculate how the weights should be updated from the output layer all the way back through each of the hidden layers to compensate for the estimated loss. Once the weights are updated, we iterate through the training dataset once more, calculate the output value, estimate the cross-entropy loss, and again back propagate errors to update the weights. We repeat this process until the model has converged and reached our target metric. For a detailed explanation of how back propagation works, refer to the article at **https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd**.

That was a quick explanation of what algorithms and neural networks are. We probably need a chapter or maybe even an entire book to dive deep into the various ways in which you can set up a neural network, and configure and optimize it. Additionally, there are different types of neural networks and algorithms, based on the use case, and you can easily lose yourself in this maze. In order to make it easy for you to understand what is popular and what algorithm or neural network you should use based on your need, to help you choose between an algorithm or a neural network, and to aid other considerations you may have, such as hyperparameter tuning or the evaluation metric to use, the next section will provide guidelines on what to use when. We will also look at what some of these algorithms are, how and why to use them, and other considerations in detail.

Simplifying the Algorithm versus Neural network conundrum

You might be thinking this is all great, but how do I use this information? How do I actually get started deciding on an algorithm or a neural network? When should I use an algorithm, and when should I use a neural network? These are perfectly valid and in fact, valuable questions. Technology companies and the open-source community are innovating at such a rapid pace that it is very hard to be keep up with what is happening. In fact, at AWS, we added more than 250 new features in AI/ML in 2021 based on customer feedback. Fortunately, help is at hand; in this section, we will first discuss what parameters help us make ML algorithm (and neural net) choices, how to pick an algorithm, and finally, how to determine whether we made the right choice.

When you are about to finalize the candidate list of algorithms or neural nets you want to go with, you should already have answers to some key considerations in your ML workflow, as shown in *Figure 5.6*:

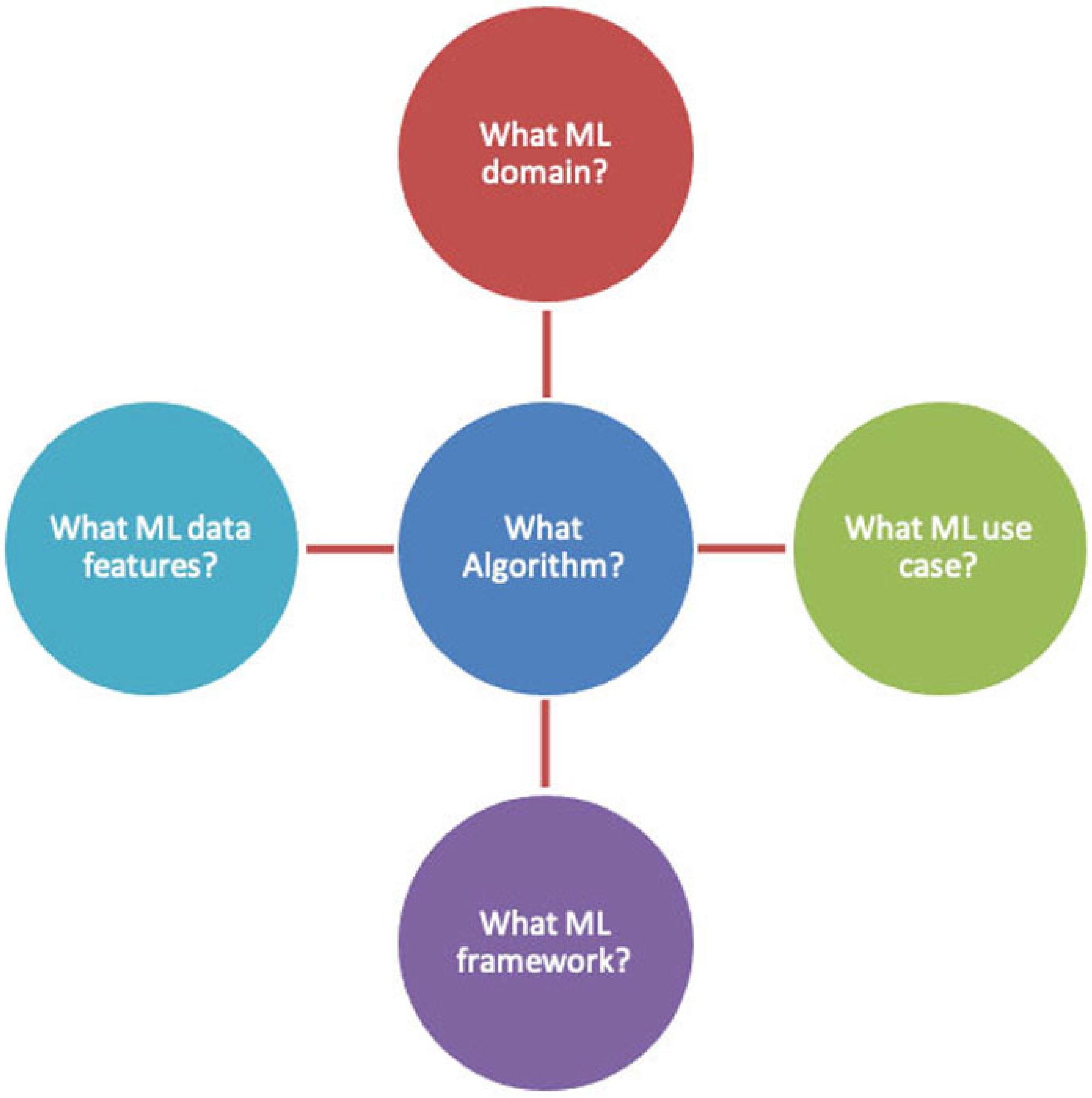


Figure 5.6: *Considerations for Algorithm selection*

What ML domain?

First, we look at the business challenge we are trying to solve and qualify if it is really a ML problem. If our requirement can be addressed by a deterministic approach (events trigger actions that follow a predetermined path based on conditions and lead to a consistent outcome every time), we do not need ML. If we need to predict a future outcome based on historical data, derive a category for unseen inputs, or uncover patterns from ambiguous data points, we probably need an ML solution. In this case, you need to first understand the ML domain (what type of a ML problem) you are looking at. If you need ML for classifying images or detecting objects in images, or segmenting images, your ML domain is Computer Vision or CV. If your requirement is to understand human speech, your domain is Natural Language Understanding or NLU. If your requirement is to process text, derive insights from text, or classify based on text, your domain is Natural Language Processing or NLP. If you are working with business data in tables and need to predict or derive insights from them, your domain is tabular data processing.

What ML use case?

Once we know our ML domain, the next step is to dive a little bit deeper to understand the use case. For example, if our domain is CV, what is it we want our ML model to do in CV? Do we want our model to look at an image and classify it as belonging to a predefined category (image classification)? Do we want our model to look at objects within an image, draw a bounding box and call out what that object within the bounding box is (object detection)? Do we want our model to look at an image and segment objects by classifying the pixels (semantic segmentation or instance segmentation)? Or do we want to look at a video and identify objects or classify specific frames (image classification or object detection)? Or we may want to process an image to determine a next course of action for an agent interacting with an environment (policy optimization with reinforcement learning; for more details, visit **https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html**). As you can see, the use case has a direct relationship with the algorithm we will choose for our experimentation. But we cannot stop with just this information; we need to know what framework we should use and if the data has useful features.

What ML framework?

As is the norm in technology, there are quite a few ML frameworks out there; sometimes deciding on what we need may appear overwhelming. For example, you can use both Tensorflow (**https://www.tensorflow.org/**) and PyTorch (**https://pytorch.org/**) for tasks like regression, image classification and reinforcement learning, but you can use only Deep Graph Library or DGL for link prediction problems. And you can use sci-kit learn (**https://scikit-learn.org/**) for almost any task, but only GluonTS (**https://ts.gluon.ai/**), Recurrent Neural Network or RNN (**https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks**), Sequence to Sequence or Seq2Seq (**https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html**) in an ML framework for time series forecasting. As you can see, there are no distinct indicators for what to use when, and definitely no one technique that fits all. As a general guideline (remember that ML tech is expanding at a faster rate than our universe, and change is the only constant factor in this space), the author suggests using popular frameworks like MXNet (**https://mxnet.apache.org/versions/1.9.1/**), Tensorflow or PyTorch for all ML domains due to extensive availability of helper functions, APIs, support and examples to get started. For common ML tasks, the author recommends trying out the Amazon SageMaker built-in algorithms (**https://docs.aws.amazon.com/sagemaker/latest/dg/algos.html**), which removes the undifferentiated heavy lifting of having to select, configure, test, and finalize algorithms and frameworks.

What ML data features?

Data is the fuel that powers your ML model. A model's "intelligence" is its interpretation of the relationship between input features (column values of your data across rows) and the target feature (what we want the model to predict). The choice of an algorithm depends on the meta data of features we select from our dataset, such as data structure, data format and data type. For example, if we have relational data in a tabular format with numerical data, we are either trying to perform a time series prediction or a regression; so, we will choose algorithms that can perform those functions. If we have tabular data structure with categorical data, we are most likely performing classification, so we would choose an algorithm like naïve bayes (**https://scikit-learn.org/stable/modules/naive_bayes.html**) or XGBoost (**https://xgboost.readthedocs.io/en/stable/**). If we are working with unstructured text data, we need NLP algorithms; if we are working with images, we need CV algorithms. So, the choice of the features we use for our ML modeling also plays a role in deciding what algorithm to use.

In *Chapter 1, Introducing the ML Workflow*, we learned how to approach a ML problem, saw examples of ML use cases, frameworks and algorithms that can used to solution for a ML problem, and finally, looked at scenario-based navigation pathways we can adopt to build the solution. These are essential initial steps, but they are not enough to start committing time and money to ML training. We need to dig deeper to create a list of candidates for experimentation (iterative training and evaluation of models).

Airline pilots call this a *final approach* prior to landing. While we may be blissfully unaware enjoying an in-flight movie or a pre-landing snack, the pilots are busy executing a series of manoeuvres to bring the plane at the right heading, altitude and speed to position it in the landing hold queue. When they get the clearance for landing, they bring the plane to a *final approach*, which involves deploying the under carriage or the landing gear (the plane's wheels), adjusting the flaps to maximum, taking manual control of the joystick, and reducing altitude and speed to land the plane. The final approach leads to landing, and this is when the rubber hits the road (we start incurring ML training costs). So, what we do in this *final approach* or finalizing the algorithm and hyperparameters is very important for ensuring a satisfactory ML outcome.

For example, consider the following rows from the table we reviewed in *Chapter 1, Introducing the ML Workflow*, in the *Evolution of AI and ML* section:

ML topic	ML use case	ML framework/library	Algorithm/Neural network
Tabular data	Regression	TensorFlow Keras, MXNet, PyTorch, Sci-Kit Learn	Linear Regression, Decision Trees, XGBoost, Multilayer Perceptron (MLP), RNN
	Classification	TensorFlow Keras, MXNet, PyTorch, Sci-kit Learn	Linear Regression, Decision Trees, XGBoost, MLP, CNN

Table 5.2: *Mapping of ML use cases*

In this case, the ML domain (or topic) and the ML use case are clear; we can select a specific framework that we are comfortable with, but we still have a lot of algorithm options, and if we have to try out every single one, we may be spending a lot of time and cost in experimentation alone. So, before we do that, we continue to dig a little deeper and understand what these algorithms actually mean, how are they used, are they configurable, what use case specific metrics do they support, and so on. This will help narrow our algorithm selection a bit further so that we only focus on the most essential ones.

NOTE: In *Chapter 7, Let George Take Over*, we will explore an approach to automate the algorithm selection and model training process using a technique called AutoML. It is, however, important to understand how to do algorithm selection iteratively via experimentation as circumstances of our ML project may dictate one approach or the other.

For the tabular regression use case, in the previous table, we have algorithms like linear regression; XGBoost; decision trees; and neural nets like Multilayer Perceptron or MLP and RNN mentioned. Let us understand what this actually means by looking at their hyperparameters and evaluation metrics. Hyperparameters are configuration variables that can used to control the behaviour of an algorithm during training. Evaluation metrics define the goal that, once met, indicates that our model has trained sufficiently for our task. Refer to the following table for a breakdown of specific metadata for the algorithms we discussed earlier. This is just a drop in the ocean in terms of the choices available in ML. We need multiple books to cover every single algorithm or neural network out there, and by the time we write about them, more algorithms would have been added because of the pace of innovations in ML these days.

Name	**Type**	**For**	**Important Hyperparameters**	**Common Evaluation Metrics**
Linear Learner	Statistical model or function	Classification or Regression tasks	Number of classes, predictor type, binary classifier model selection criteria, early stopping patience, epochs, learning rate, loss	RMSE [Root of the Mean Square Error between predicted and expected values in a dataset] Confusion matrix [for classification that validates classifier performance as predicted vs actual for each

				classification label]
Extreme Gradient Boosting or XGBoost	Boosted decision trees	Regression or classification tasks	eta or learning rate, max depth, min child weight, alpha, lambda	RMSE, AUC [or area under the curve measures classifier performance at all classification threshold points] or confusion matrix
Random Forest	Ensemble of decision trees	Regression or classification tasks	Number of trees, split criterion, max depth, max features, min weight fraction	RMSE or confusion matrix
Multilayer Perceptron or MLP	Neural network	DeepLearning image classification, object detection, time series forecasting and more	Number of hidden layers, count of neurons per hidden layer, activation function, optimizer, epochs, batch size, loss function	Depends on ML use case. For image classification - AUC or confusion matrix. For object detection - mean Average Precision or mAP, which compares labeled bounding box to predicted bounding box. For time series - MAPE or mean absolute percentage error.
Recurrent Neural Networks or RNN	Neural network	DeepLearning Text classification, text translation, NLU, NLP and more	Number of hidden layers, count of neurons per hidden layer, activation function, optimizer, epochs, batch size, loss function	Text classification - confusion matrix. Text translation - Bilingual Evaluation Understudy or BLEU
Convolutional Neural Networks or CNN	Neural network	DeepLearning Text or image classification,	Number of hidden layers, count of neurons per hidden layer,	Confusion matrix or mAP

		object detection and more	activation function, optimizer, epochs, batch size, loss function	

Table 5.3: *Example of considerations for finalizing algorithms*

Statistical models or decision tree considerations

If our use case is to predict house prices using a tabular housing dataset, we will most likely select a regression algorithm. On the other hand, we will select a binary classification algorithm to predict the customer churn or a multi-class classification algorithm if we need to predict whether a picture is that of a cat from a possible list of animals. If we need to predict all possible genres for a movie, we will use a multi-label classification algorithm. For regression or classification problems, you can use either a statistical algorithm or a neural network. For complex deep learning type of tasks with several parameters and high data volumes (not necessarily high-dimensional features), we use a neural network. For tabular classification problems, including both continuous and categorical high dimensional features, algorithms like linear learner, naïve bayes and decision trees perform well. Decision trees struggle a bit with regression problems, but that is resolved using a boosted tree algorithm like XGBoost (**https://xgboost.readthedocs.io/en/stable/**). Let us now look at some of the hyperparameters of these algorithms to understand how they help in controlling the training behaviour.

For the linear learner algorithm, predictor type helps you specify the ML use case: binary classification, multi-class classification or regression. The number of classes is applicable for multi-class only and indicates how many classes are to be learned. The binary classifier model selection criteria specify the deciding metric to select a model from a list of models trained, when the use case is binary classification, such as, accuracy, or F1 score, or a loss function. The early stopping patience indicates the number of epochs to monitor for an improvement to a specific goal metric (either what was specified in binary classifier model selection criteria or loss) before ending training. Epochs indicate the maximum number of passes through the training data a model must traverse. The learning rate is a number between 0 and 1 that indicates the quantity by which the step size of a model is updated during training after the loss function determines the error between the predicted value and the label. The loss indicates the mathematical function using which the loss for the current training step of the model is calculated, such as squared loss for regression and logistic or softmax loss for classification.

Decision trees like XGBoost also support learning rate, aka eta, the size by which weights are updated after learning to prevent overfitting. For random forest, which is an ensemble of decision trees, the number of trees is an important hyperparameter. We can also specify the split criterion for a random forest that indicates the split method to use, such as Gini Impurity or Entropy. The max depth indicates the size or depth of the tree in terms of how many levels the tree can split into, the max features indicate the total number of features the tree should use at each node to calculate the split for that node. The min weight fraction for a random forest is the same as min child weight for boosted decision trees, which indicates the criteria for node partitioning. If the weight for the partitioned node is less than the minimum weight specified here, there will be no further partitioning, and the node will become the leaf node. Alpha and lambda are the regularization factors for weights, the higher these values, the lower the weight updates and more conservative the model.

The evaluation metric or the goal metrics for a model depend on the ML use case, and the choice of such metrics are pretty much common and standardized based on what the model is trained for. For classification problems, the goal metric is almost always AUC or area under the receiver-operator characteristics curve for binary classification or logistic regression use cases. The AUC measures the classifier performance at all classification thresholds for the dataset. For multi-class classification, the goal metric that works best is a confusion matrix that plots the actual vs. predicted classifications for each class label. For regression use cases, the most popular metric is RMSE, or root mean square error, which is the square root of the mean squared error between the predicted and actual values.

Your choice of an algorithm should be made after careful consideration of the hyperparameters available and the goal metric you want to achieve. Depending on your business scenario, you may be required to use a confusion matrix over AUC, or you may need to use accuracy as an evaluation metric for your classifier because you are more interested in True Positives and True Negatives. You may be interested to control the learning rate for your decision tree, but if you select the random forest algorithm, it is not supported. So, it is important that you investigate the hyperparameters and evaluation metrics that are available for an algorithm before finalizing your algorithms for modeling.

Deep Learning and neural network considerations for Algorithms

NOTE: One thing that should be immediately obvious is that we use neural networks when we want to do deep learning.

Think of deep learning as a subset of ML but tailored for much more complex and large problems. We use neural networks because they can leverage high-performance computational capability when using CPU (with a lot of cores) or GPU instances, and the ability to process billions of model parameters during training, which translates to learning complex relationships from data. For example, think about training an object detection model to recognize cats in pictures. Let us assume we have a dataset of 5000 images, with each image being 300 * 512 in size. This translates to 153,600 pixels for each image. Each picture has a RGB coordinate, which is a three-dimensional vector that encodes the colour of that pixel. So, the shape of each image, when vectorized, is [153600, 3]. You can imagine the amount of data the model will have to learn during training for 5000 images. In typical object detection training, the images are of much higher resolution and of a significantly higher volume, so a statistical algorithm simply does not provide the learning depth for this scenario. However, a neural network can learn this data volume quite easily with even CPU instances. For certain NLP models that learn from entire corpuses, such as Wikipedia (6.5M articles in English alone), we need complex neural networks with multiple layers and neurons, GPU compute power, and distributed learning (parallel computing) to complete the training in a reasonable amount of time. For example, GPT-3 (**https://openai.com/blog/gpt-3-apps/**) is an NLP model that has the capacity to learn 175 billion parameters, can produce human like text, can do text summarization, can answer questions and complete other NLP tasks, but it is not the largest ML model. By the time this book is published, we may even see a model with 1 trillion or more parameters.

From the previous table, we can also see that the hyperparameters of the different types of neural networks are fundamentally the same. We may need additional hyperparameters, such as the number of filters for a CNN, but at a high level, these are the standard hyperparameters. Hyperparameters are a set of controls enabling you to define how you would like your model to train. Since you best know your data and your objective, you use these controls or levers to:

- Tell your model to learn faster or slower (learning rate)
- Define larger neural networks to sufficiently capture complex data relationships (number of hidden layers)
- Allow the collection of features in your dataset to be processed iteratively through the layers (number of neurons per layer)
- Use specific math functions to determine how inputs are processed and outputs are generated (activation functions)
- Encourage the learning progress by optimizing the approximation parameters learned to reduce loss (optimizers)
- Specify the number of samples processed through each iteration (batch size) and specify the number of passes through the neural network for the entire dataset (epochs).

The evaluation parameters are more dependent on what we are doing with the neural network, such as confusion matrix or AUC, for tabular/text classification, mean average precision or mAP for object detection, and mean absolute percentage error or MAPE for time series forecasting. Again, we use neural networks only for deep learning, but once we know that, we can select the type of neural network (or combine multiple neural networks in a single model; for example, Sequence to Sequence model uses two RNNs). Then we decide the hyperparameters related to the network architecture; select a ML framework; build the neural network; and select other hyperparameters like optimizers, epochs, and batch size; and provide access to the input dataset and start training.

OK, that was a lot of theory on algorithms and neural networks!! Thank you for being patient with us so far. We know you are eager to get hands-on experience of what you learned so far. In the next section, we will explore code samples of some popular SageMaker algorithms for different ML domains and also look at examples of how to build and run neural networks.

Building ML solutions with Algorithms and Neural Networks

In the next chapter, Iteration *M*akes Intelligence, we will learn the step-by-step flow of how to iteratively train ML models by setting up training and test datasets, choosing algorithms, setting up hyperparameters, running training, and tuning models for the desired goal and evaluating model performance once trained. In this section, we will focus more on how to set up estimators (**https://sagemaker.readthedocs.io/en/stable/api/training/estimators.html**) in SageMaker and how to build some simple neural networks. Estimators are abstracted interface classes that provide an easy-to-use approach for setting up a wide variety of model training tasks in SageMaker. Depending on the ML framework you select, you can create a MXNet estimator, a Tensorflow estimator, a PyTorch estimator, or Scikit-learn estimator and more. Once you define the estimator, you pass parameters like the algorithm to use (provided as a docker container image), or an entry point script that contains the algorithm code in case you are bringing your own algorithm to SageMaker, compute instance types, permissions, encryption keys, output location to store model artefacts, hyperparameters, and evaluation metrics. Once the estimator is ready, you can simply call the fit method, pass the training dataset, and start the training job. We will do a step-by-step code walkthrough using SageMaker Jupyter notebooks, like we did in the previous chapters.

Follow the instructions in the Setting up your AWS account section in *Chapter 2, Hydrating Your Data Lake*, to sign up for an AWS account. Once you have signed up, log in to your AWS account using the instructions at **https://docs.aws.amazon.com/IAM/latest/UserGuide/console.html**. You first need to create an Amazon S3 bucket (**https://docs.aws.amazon.com/AmazonS3/latest/userguide/create-bucket-overview.html**) and note down the name of the bucket. Next, follow the instructions in the *Technical Requirements* section in *Chapter 3, Predicting the Future With Features*, to onboard to an Amazon SageMaker Studio domain, clone the book's GitHub repository given at the beginning of the book, and click on *Chapter 5, Casting a Deeper Net*, to open it up. Open the `sagemaker_algorithms.ipynb` Jupyter notebook by clicking on it. Execute the following instructions to continue with your build activity.

The first cell contains imports for SageMaker: boto3, a Python SDK for AWS (**https://boto3.amazonaws.com/v1/documentation/api/latest/index.html**); pandas, a python library for data processing (**https://pandas.pydata.org/**); numpy, a Python library for working with numeric data (**https://numpy.org/**); and other libraries we need. We also need to provide the name of the bucket we used in Chapter 4 of the book and get the handle to the Amazon S3 bucket so that we can download the output CSV file we created in the chapter. Either press *Shift* + *Enter* or click on the triangular play button after keeping your cursor in the first cell to execute it.

```
import boto3
import io
import os
import time
import pandas as pd
import numpy as np
import sagemaker
import sagemaker.amazon.common as smac
from sagemaker.amazon.amazon_estimator import RecordSet
from sagemaker import get_execution_role
# Enter the name of the bucket you used in Chapter 4 here
bucket = 'bucket-name-from-Chapter4'
prefix = 'aiml-book/chapter4/glue-out/'
s3 = boto3.client('s3')
```

Execute the next cell to download the scaled CSV file from *Chapter 4 – Orchestrating the data continuum*:

```
# download the scaled CSV file we created in Chapter 4
s3.download_file(bucket,
prefix+'wine_scaled.csv','wine_scaled.csv')
```

Let us execute the next cell to load the contents of this CSV file to a Pandas data frame, and let us review the shape of this data frame:

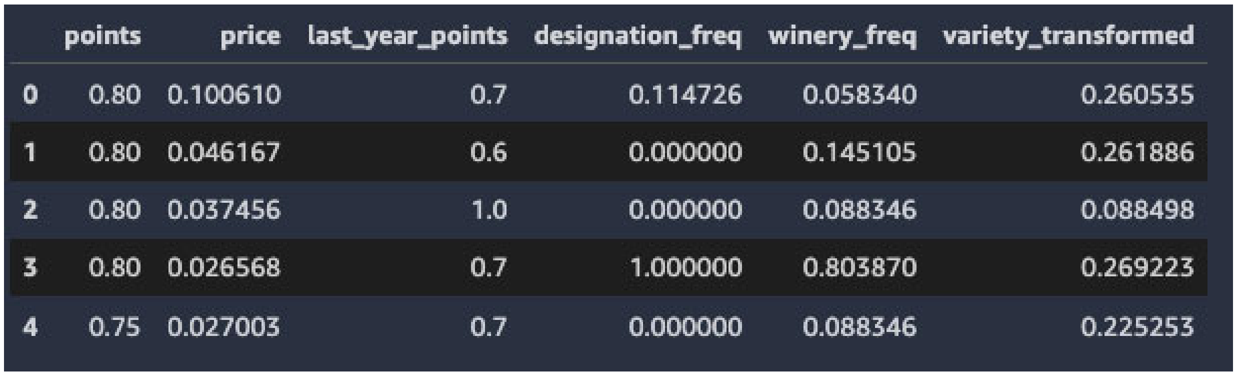
```
# Let's first load the data into a Pandas dataframe so it is
easy for us to work with it
wine_scaled_df = pd.read_csv('./wine_scaled.csv',
sep=',',header=0)
wine_scaled_df.shape
```

We should get the following output for the shape: `(144030, 54)`

To keep our example understandable, let us get rid of the one hot encoded country columns; this will reduce the number of features in our dataset from 54 to 6. We will select only the first 1,000 rows of our dataset. Execute the next cell.

```
# now we have our data, let us get rid of the country columns
and select a thousand rows to make it more understandable
wine_alg_df = wine_scaled_df.iloc[0:1000,0:6]
wine_alg_df.head()
```

You should see the following output:



	points	price	last_year_points	designation_freq	winery_freq	variety_transformed
0	0.80	0.100610	0.7	0.114726	0.058340	0.260535
1	0.80	0.046167	0.6	0.000000	0.145105	0.261886
2	0.80	0.037456	1.0	0.000000	0.088346	0.088498
3	0.80	0.026568	0.7	1.000000	0.803870	0.269223
4	0.75	0.027003	0.7	0.000000	0.088346	0.225253

Figure 5.7: *Scaled wine dataset with country features removed*

Now, execute the next few cells to reorder the columns so that our label feature (price) is the first column:

```
col_ord =
['price','points','last_year_points','designation_freq','winery
_freq','variety_transformed']
# reorder to move label (we want to predict the price of the
wine) to first position
wine_alg_ord_df = wine_alg_df.reindex(columns=col_ord)
wine_alg_ord_df.head()
```

So far, we have completed the steps to get our dataset ready. In the following section, we will look at the steps to follow to use SageMaker Linear Learner algorithm (**https://docs.aws.amazon.com/sagemaker/latest/dg/linear-learner.html**) to train a regression model to predict the wine price.

Using Amazon SageMaker Linear Learner Algorithm

As discussed earlier, we will first set up the Linear Learner estimator, and then pass hyperparameters and training data to start model training.

NOTE: We have also provided an alternative method to train your model using Amazon SageMaker but without using the estimator. This method uses SageMaker's Create Training Job API (https://docs.aws.amazon.com/sagemaker/latest/APIReference/API_CreateTrainingJob.html). This is available at the end of the notebook as an optional section for reference.

First let us get the ARN of the IAM execute role we need to pass for running our training job. Execute the first cell under the *Algorithm 1 - Linear Learner* section in the notebook.

```
# we need the IAM role from our notebook to run our training
job
role = get_execution_role()
print(role)
```

You should see the ARN of your IAM role printed. Now, execute the next cell to split our dataset into training and validation datasets. We will use 90% of our original dataset (approximately 900 rows) for training and 10% for validation. We will use the Numpy library to generate a random list to use as a reference to split our dataset.

```
split_list = np.random.rand(wine_alg_ord_df.shape[0])
t_list = split_list < 0.9
v_list = split_list >= 0.9
train_ds = wine_alg_ord_df[t_list]
val_ds = wine_alg_ord_df[v_list]
```

Execute the next cell to separate the train and validation datasets into a list of labels and features for each dataset. The label is the first column from our dataset, which is the price of wine, and the features are the remaining columns. The model will use the features to learn and the label to calculate the loss function, which is the difference between the model's predicted value and the label:

```
train_label = train_ds.iloc[:, 0].to_numpy()
train_features = train_ds.iloc[:, 1:].to_numpy()
val_label = val_ds.iloc[:, 0].to_numpy()
val_features = val_ds.iloc[:, 1:].to_numpy()
```

Now in the next cell, we will create the Linear Learner estimator, specify the training compute instance count, the instance size, what type of predictor we want to train (linear learner supports both classification and regression problems), the number of epochs or passes through our training dataset for our model to learn, and the loss function we want the model to use. We are using **`squared_loss`**, which indicates the sum of the squared difference between the predicted and the actual values. Execute the next cell to set up the estimator:

```
# Get the linear learner estimator and specify hyperparameters
estimator = sagemaker.LinearLearner(
  role=role,
  instance_count=1,
  instance_type="ml.m5.xlarge",
  predictor_type="regressor",
  epochs=10,
  loss="squared_loss",
)
```

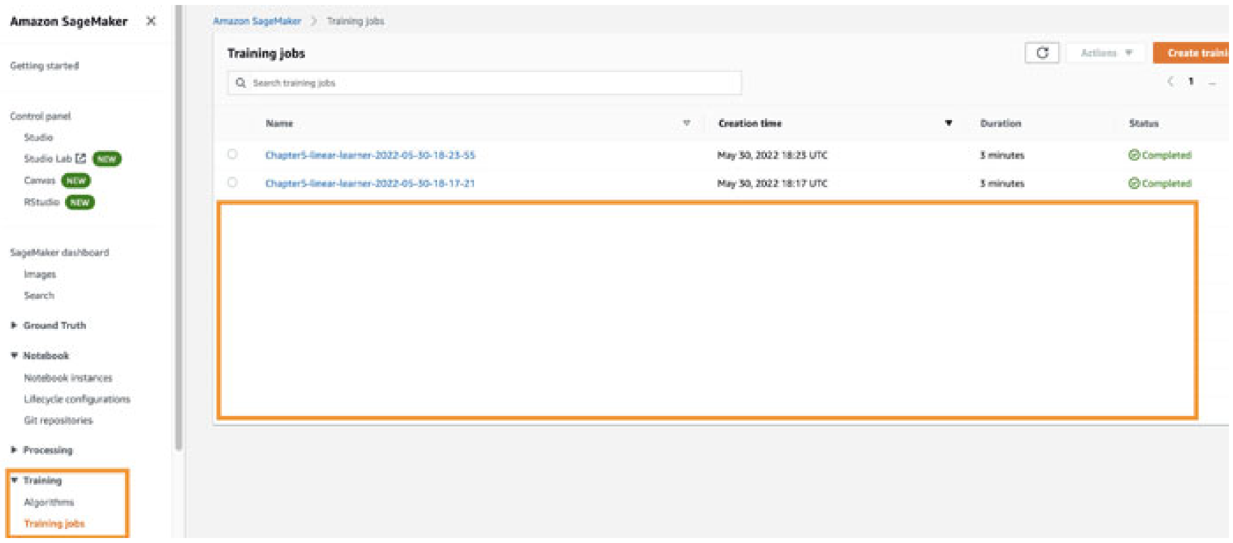
The Linear Learner estimator required data to be arranged in float32 representation of **`RecordSets`**. Execute the next cell to convert our datasets.

```
train_records =
estimator.record_set(train_features.astype("float32"),
train_label.astype("float32"), channel="train")
val_records =
estimator.record_set(val_features.astype("float32"),
val_label.astype("float32"), channel="validation")
```

We will now start the model training by running the fit method on our estimator. We pass a **`mini_batch_size`** parameter to indicate the sample size the model needs to use at a time for the learning process. Execute the next cell:

```
estimator.fit([train_records, val_records],mini_batch_size=50,
wait=False)
```

This will submit a training job in SageMaker. Navigate to Amazon SageMaker console by typing SageMaker in the services search bar in the AWS Management Console (console.aws.amazon.com). Now, click on **`Training`** and then on **`Training jobs`** in the left pane of the console to bring up the list of training jobs. The job that was submitted when we executed the fit command should appear as "**`linear-learner`**"-current timestamp. Click on the job name, as shown in the following image:



Name	Creation time	Duration	Status
Chapter5-linear-learner-2022-05-30-18-23-55	May 30, 2022 18:23 UTC	3 minutes	Completed
Chapter5-linear-learner-2022-05-30-18-17-21	May 30, 2022 18:17 UTC	3 minutes	Completed

Figure 5.8: *Linear learner training job in SageMaker console*

After you click on the job name, scroll down a few sections to the **`Monitor`** section and click on **`View logs`** to review the progress and the training steps of the model training job. That is how you use the Linear Learner algorithm in Amazon SageMaker using the estimator method. You can review the optional section at the bottom of the notebook to learn a different method for training that uses the SageMaker Create Training Job API (**https://docs.aws.amazon.com/sagemaker/latest/APIReference/API_CreateTrainingJob.html**). In the next section, we will see how to use a neural network to train a model for regression.

Using a Multi-layer Perceptron or MLP neural network

The MLP is an artificial neural network named so because it supports building multiple layers of artificial neurons to model a deep learning problem. In essence, almost all neural networks are MLPs. In the current example, we will be building the simplest form of MLP; advanced forms of MLPs are CNNs and RNNs. When you combine multiple CNNs or RNNs, you create much more advanced models for large and complex problems like natural language understanding or deep image recognition and more. For our regression problem to predict the wine price based on other characteristics, we will set up a three-layer neural network with two hidden layers and one output layer. The first hidden layer will receive the input features with ReLU activation (read the *Introducing Algorithms and Neural Networks* section in this chapter for more details), with random normal weights initialization, and optimized using a Stochastic Gradient Descent (**https://keras.io/api/optimizers/sgd/**) objective. We will use five neurons for the first hidden layer, three for the second hidden layer, and one for the output layer to receive the regressed prediction for wine price. We will use the mean squared error as the loss function, just like when we trained with the linear learner algorithm.

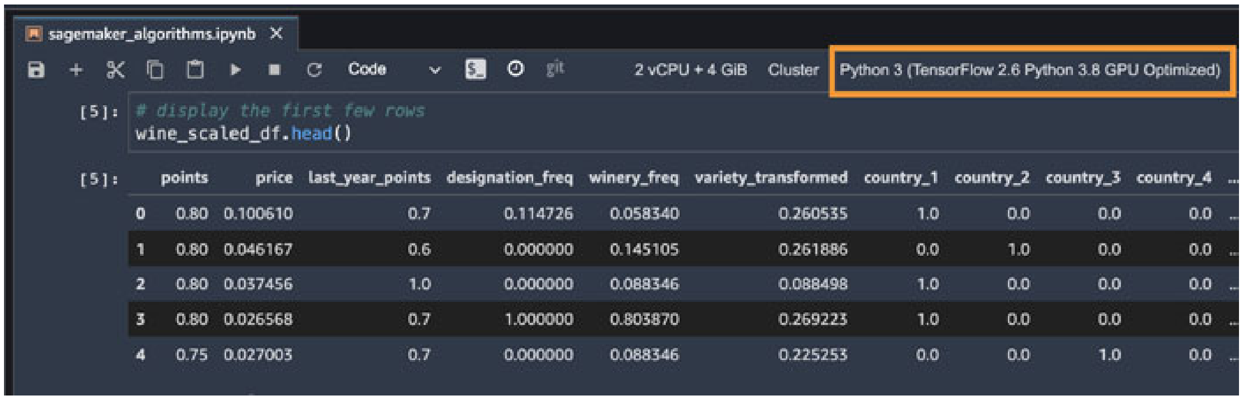
We will be reusing the train and validation datasets from the linear learner example. Let us first review the shapes of our train, validation labels and features. Execute the first cell in this section:

```
print("training label shape is: " + str(train_label.shape))
print("training features shape is: " +
str(train_features.shape))
print("validation label shape is: " + str(val_label.shape))
print("validation features shape is: " +
str(val_features.shape))
```

We should get the following output:

```
training label shape is: (906,)
training features shape is: (906, 5)
validation label shape is: (94,)
validation features shape is: (94, 5)
```

We will now use TensorFlow (**https://www.tensorflow.org/**) and Keras (**https://keras.io/examples/**). To ensure that we are able to execute the next cell successfully, we should be using a TensorFlow and Python kernel notebook in SageMaker Studio. You can verify this by checking the kernel type in the top-right corner of your notebook. Refer to *Figure 5.9*:



sagemaker_algorithms.ipynb

Code | 2 vCPU + 4 GiB | Cluster | Python 3 (TensorFlow 2.6 Python 3.8 GPU Optimized)

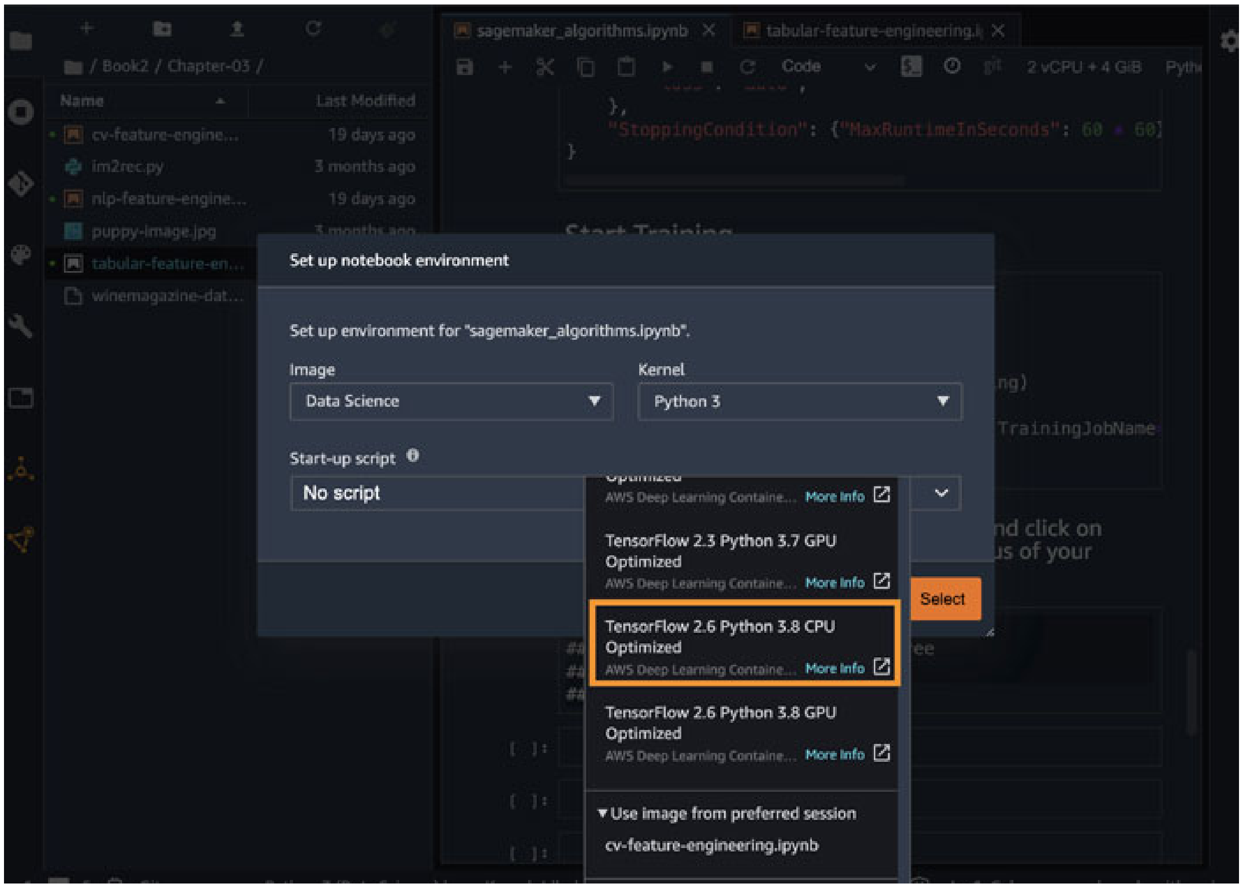
```
[5]: # display the first few rows
     wine_scaled_df.head()
```

[5]:

	points	price	last_year_points	designation_freq	winery_freq	variety_transformed	country_1	country_2	country_3	country_4	...
0	0.80	0.100610	0.7	0.114726	0.058340	0.260535	1.0	0.0	0.0	0.0	...
1	0.80	0.046167	0.6	0.000000	0.145105	0.261886	0.0	1.0	0.0	0.0	...
2	0.80	0.037456	1.0	0.000000	0.088346	0.088498	1.0	0.0	0.0	0.0	...
3	0.80	0.026568	0.7	1.000000	0.803870	0.269223	1.0	0.0	0.0	0.0	...
4	0.75	0.027003	0.7	0.000000	0.088346	0.225253	0.0	0.0	1.0	0.0	...

Figure 5.9: *TensorFlow Kernel in SageMaker Studio notebook*

If your notebook does not show a TensorFlow kernel, you can easily change this by clicking on the kernel name and choosing TensorFlow, as shown in the following image. For this example, it does not matter whether you select CPU optimized or GPU optimized, since our training data volume is limited. Refer to *Figure 5.10*:



Set up notebook environment

Set up environment for "sagemaker_algorithms.ipynb".

Image: Data Science

Kernel: Python 3

Start-up script: No script

/ Book2 / Chapter-03 /
Name
Last Modified
cv-feature-engine... 19 days ago
im2rec.py 3 months ago
nlp-feature-engine... 19 days ago
puppy-image.jpg
tabular-feature-en...
winemagazine-dat...
sagemaker_algorithms.ipynb
tabular-feature-engineering.i
AWS Deep Learning Containe... More Info
TensorFlow 2.3 Python 3.7 GPU Optimized
AWS Deep Learning Containe... More Info
TensorFlow 2.6 Python 3.8 CPU Optimized
AWS Deep Learning Containe... More Info
TensorFlow 2.6 Python 3.8 GPU Optimized
AWS Deep Learning Containe... More Info
▾Use image from preferred session
cv-feature-engineering.ipynb
Select

Figure 5.10: *Select a TensorFlow Kernel in SageMaker Studio notebook*

Execute the next cell to build our network with three layers. We pass the number of input features in the **`input_shape`** variable when we add the layer:

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
# set up the neural network layer by layer with 5 neurons in
1st hidden layer
# 3 neurons in second hidden layer
# output is 1 neuron
model = Sequential()
model.add(Dense(5, activation='relu',
kernel_initializer='random_normal', input_shape=
(train_features.shape[1],)))
model.add(Dense(3, activation='relu',
kernel_initializer='random_normal'))
model.add(Dense(1))
```

Execute the next cell to compile the neural network to create a model and then run the **`fit`** command to start the model training. We selected 25 epochs (number of passes through the full training dataset), and set our **`batch_size`** to 50:

```
# We will use the mean squared error as the calculated loss
between the label and predictions
# the model will try to minimize this loss during training
# we will use the stochastic gradient descent as the optimizer
method for learning
model.compile(optimizer='sgd', loss='mse')
# fit the model
model.fit(train_features.astype("float32"),
train_label.astype("float32"), epochs=25, batch_size=50)
```

When you execute this cell, the model will start training; you should see the loss printed for each epoch of training. You should also see the loss reducing gradually:

```
Epoch 1/25
19/19 [==============================] - 0s 2ms/step - loss:
3.2979e-04
Epoch 2/25
```

```
19/19 [==============================] - 0s 2ms/step - loss:
2.1591e-04
Epoch 3/25
19/19 [==============================] - 0s 2ms/step - loss:
1.6819e-04
Epoch 4/25
19/19 [==============================] - 0s 3ms/step - loss:
1.4893e-04
```

Now, let us calculate the overall loss of the trained model with the validation dataset. Execute the next cell to get the RMSE loss for our model:

```
import math
val_results = model.evaluate(val_features.astype("float32"),
val_label.astype("float32"))
print('Root Mean Squared Error or RMSE is: ' +
str(math.sqrt(val_results)))
```

We get the following result:

```
Root Mean Squared Error or RMSE is: 0.029675475229435623
```

That brings us to the end of this section. We saw how to use SageMaker to train a regression model using the Linear Learner algorithm, along with examples, and we looked at how to use Python to set up your own MLP neural network to build a regression model using a SageMaker Studio notebook. Linear learner is just one algorithm in SageMaker's collection of built-in algorithms. SageMaker also supports bring your own algorithms and models. For more details, refer to the documentation at **https://docs.aws.amazon.com/sagemaker/latest/dg/algos.html**.

Conclusion

We covered more complex ground in this chapter and learned how the brain of an ML model actually works (algorithm or neural networks). We also understood how to use an algorithm to build and train a ML model. The core building blocks of a ML solution are algorithms and the dataset, and with this chapter, you hopefully had a solid understanding of how to get both of them in place for your project.

In the next chapter, we will look at the big picture of how model training and tuning work end to end. We will take a few ML examples and use sample notebooks to walk the steps of building, training and evaluating models step by step, with detailed instructions and fun scenarios.

Points to Remember

Here's a summary of the key learning points for this chapter:

- In this chapter, we were first introduced to the concepts of an algorithm in the general context, the mathematical context, the computing context and finally, its role in ML and AI.
- We then learned how algorithms evolved through the years and understood the differences between algorithms like deterministic and probabilistic.
- We then dove deeper into the architecture of popular algorithms today, such as decision trees and neural networks.
- We learned how a decision tree works and what type of use cases can it be used for, and we discussed the fundamental concepts behind a decision tree.
- We then looked under the hood of a neural network architecture and discussed the learning process in detail.
- We then changed gears and discussed how to approach an ML problem from the perspective of choosing between an algorithm and a neural network.
- We looked at different types of ML use cases and learned which cases statistical algorithms are applicable, under what circumstances we can use decision trees and when to use neural networks.
- We then discussed the overall considerations in terms of the ML use case, the topic, the domain and the data, and looked at how all this ties back to the algorithm.
- We reviewed the hyperparameters and the evaluation metric for each type and looked at examples of algorithms and neural networks. We also discussed why to select one over the other.
- We then switched over to hands-on building by using the SageMaker linear learner algorithm, using an estimator to train a model for a regression use case with our wine prices dataset from *Chapter 4, Orchestrating the Data Continuum*.

- Finally, we built a Multi-Layer Perceptron (MLP) neural network from the ground up using TensorFlow and Keras in the SageMaker Studio notebook to train a regression model using the same dataset. We evaluated this model using the RMSE evaluation metric and reviewed the results.

Multiple Choice Questions

Use these questions to challenge your knowledge of what we covered in this chapter.

1. **What is the correct definition of information gain used to determine portioning or branching nodes in a decision tree?**
   a. Weighted average of entropy of the dependent features minus the entropy of the main feature
   b. Entropy of the main feature minus the weighted average of entropy for the dependent features
   c. Difference between entropies of related features
   d. Difference between entropies of randomly sampled features
2. **In a neural network, back propagation of errors is used to calculate how the weights should be updated from the output layer all the way back through each of the hidden layers to compensate for the estimated loss?**
   a. True
   b. False
3. **Which of the following options is the right choice for output layer activation in a binary classification problem?**
   a. Sigmoid
   b. ReLU
   c. Linear
   d. Softmax
4. **AUC is a good evaluation metric for regression problems.**
   a. True

   b. False

5. **Which of these is NOT an Amazon SageMaker built-in algorithm?**
   a. Linear Learner
   b. XGBoost
   c. Object Detection
   d. RoBERTa

Answers

1. **b**
2. **a**
3. **a**
4. **b**
5. **d**

Further Reading

- *Amazon SageMaker built-in algorithms*: **https://docs.aws.amazon.com/sagemaker/latest/dg/algos.html**
- Scikit Learn ML algorithms: https://scikit-learn.org/stable/
- The advent of the algorithm: **https://www.amazon.com/Advent-Algorithm-300-Year-Journey-Computer/dp/0156013916/ref=sr_1_13?gclid=CjwKCAiAy_CcBhBeEiwAcoMRHPkBYBDzv4zUl0emaYp8-JXcdUlnRZpbdrl8iuGsvxn8b-K3-tj-jRoCkB0QAvD_BwE&hvadid=256536957733&hvdev=c&hvlocphy=9010798&hvnetw=g&hvqmt=e&hvrand=7242570720563610897&hvtargid=kwd-432315633310&keywords=history+of+algorithms&qid=1671216379&s=books&sr=1-13**

CHAPTER 6
Iteration Makes Intelligence (Model Training and Tuning)

Introduction

By this time in the book, you must have properly framed your machine learning problem, know the structure and makeup of your data, have decided what type of training you need to do and have the appropriate algorithm ready. It is now time to see what happens when the machine learning rubber hits the computational road. For the specific machine learning problem, we must first define the metric used to evaluate your model, and then iterate through a combination of model training and parameter tuning to reach or exceed that metric. In this chapter, we will see how to build, train, tune, and evaluate your ML model using Amazon SageMaker, Jupyter notebooks, and Python, along with actual code examples.

We will be taking on the role of a data scientist who is extremely interested in expanding their technological portfolio into machine learning, starting with training and tuning models. The data scientist has more than a decade of experience in collecting, combining, transforming, and analyzing data but has historically handed that data off to other teams to use in various applications. Curious about becoming the machine learning version of a full stack developer, the data scientist begins exploring the different methods of training and fine-tuning the various models. As a classic Sagittarius, they are uniquely suited to this pursuit.

Structure

In this chapter we will discuss the following topics:

- Introduction
- Chapter Scenario
- Structure

Objectives

After reading this chapter and practicing with the associated notebooks, you should be able to understand the fundamentals of training a machine learning model. This includes the individual cycles of machine learning that can be broken into three components: training and tuning the model, registering and managing the resulting model, and deploying the model to an endpoint and performing inference. By the end of this chapter, you should be familiar with various methods of the first of those three: training and tuning your model. We will explore the process using several models and use a selection of metrics to determine overall model suitability.

To try the examples in this section, refer to the Technical Requirements section in *Chapter 1, Introducing the ML workflow* to sign in to the AWS management console, execute the steps in onboard to SageMaker studio, and execute cloning the repository to SageMaker Studio to get started. Click on the folder that corresponds to this chapter number. If you see multiple notebooks, the section title corresponds to the notebook name for easy identification. You can also passively follow the code samples using the GitHub repository provided at the beginning of the book.

The Meaning of Training

Before we go any further, let's pause and examine what training really means.

Training a machine learning model means we help an algorithm determine the good values for all the weights and the potential bias from our provided examples. In the case of supervised learning, we allow the model to learn from our provided examples, iterating until we have a resultant model that minimizes our loss. Loss can be thought of as the weight penalty for a bad decision. With a perfectly trained model, our loss would be zero (though that is rarely, if ever, the case). Our goal, then, should be to iterate over our model training process to achieve a loss that is close to zero.

Unsupervised learning uses the same process and methods but learns without a predefined example set. This allows the algorithm freedom to determine patterns, clusters, anomalies, or other information.

What Training Means for Deep Learning

As we saw in previous chapters, Deep Learning is a sub-discipline of Machine Learning (which is, itself, a sub-discipline of Artificial Intelligence, which is a sub-discipline of Information Technology). Deep Learning uses a series of Artificial Neural Networks patterned after the human brain to iteratively determine patterns in data. It does this by layering collections of nodes, or neurons, each of which have a weight and a threshold. If the output of any of the nodes is above the threshold value, then that node is activated, sending data to the next layer of nodes. There are specific types of layers of nodes: input layers, hidden layers, and output layers. Think of each of the individual nodes as a miniature linear regression model. It has input data, a selection of weights, a threshold (or bias) and an output.

By passing data through each of the layers in succession and adjusting the weights based on the node output, the network learns and gets progressively more understanding of the associations of the data being trained.

As you can probably guess, this takes a lot of iterations, which translates to a lot of computational power. This is also why **Graphical Processing Units** (**GPUs**) are uniquely suited for deep learning. Thanks to their large-scale parallel processing power and on-board memory, GPUs can train Deep Learning models faster and more efficiently.

GPU vs CPU

In the realms of processors, which one should be used? It’s not quite as simple as raw processing power or total FLOPS (Floating Operations per Second) that a processing can bring to bear. Like any set of tools, it depends on the task at hand. CPUs are often a good choice precisely because they are good at a very large range of tasks. Since so many workloads (ML and non-ML) use them, their costs are typically lower, and they do not suffer the same supply chain issues as other types of processors. On the other hand, GPUs are purpose-built to manage complicated mathematical processing for things like fluid dynamics, graphical rendering, and deep learning. The optimal processor type also depends on your chosen algorithm. Some algorithms (and some versions of them, always recheck!) are better suited for GPUs or CPUs during training and another for inference.

When using AWS SageMaker, as you will see in the following examples, you specify the compute type for training and a completely different compute type for inference. Additionally, you can use Amazon Elastic Inference to attach on-demand GPUs to your EC2 Instances, Deep Learning Containers, or SageMaker Instances.

AWS Trainium

AWS has gone beyond creating services, solutions, and mechanisms to accelerate Machine Learning practitioners in creating custom processing chips. Trainium is the second custom Machine Learning chip created by AWS (we will cover the first, that is, AWS Inferentia, in *Chapter 8, Blue or Green*) designed to provide purpose-build hardware. While it is in preview at the time of this book’s publishing, Trainium is designed for Deep Learning workloads like Natural Language processing, voice recognition, image classification and more.

In order to use AWS Trainium, sign up for the preview and use Trn1 instances for your training jobs.

Transfer Learning

Training a model from scratch takes time. In computational terms, this means it takes more processor time or more processors. We can save some of that time and those computational resources by using a model that has already been trained on a generalized data set. This is especially helpful in the realm of Natural Language Processing. We can take a model trained on a very general data source, such as Wikipedia, allowing it to understand the general structure of our chosen language, and then train it further on our more-specific training set.

The Mise en Place of Model Training

Mise en Place is a French culinary term that means to put everything in its place. It is used for a chef preparing everything they need and putting it in its place before starting service. In terms of Machine Learning, we have set most of our Mise in the previous chapters: carefully selecting our problem, combining our data, slicing and dicing with feature engineering, setting the transformations, and sharpening the data through analysis. To these, we can add our spices: training data emitted while our model is being trained and our chosen model parameter ranges. And just like that, service begins.

Defining Model Training and Evaluation Metrics

Since we have chosen Amazon Web Services as our Machine Learning platform, our service family is Amazon SageMaker. It is a collection of services intended to empower, simplify and enable Machine Learning practitioners. As we move through the steps necessary to train our chosen models, we will default to using the SageMaker services and methods where appropriate. Everything we will show you can be done locally or using a virtual workspace, but SageMaker can easily save you both time and money.

Machine Learning is an iterative process. Even when using tools that automate part of that process, it is likely that you will complete the same tasks multiple times in order to achieve the best outcomes. Most real-world Machine Learning projects tend to use more than one model, which exponentially increases the number of combinations, parameters, hyperparameters and other aspects of training alone. To keep track of, organize, and retain visibility into those iterations, we use SageMaker Experiments. There are two aspects of SageMaker Experiments: trials and experiments. Trials are collections of training steps involved in a singular training job. An experiment is a logical grouping of trails.

As we create our training jobs, we can pass in an additional parameter assigning the job to an experiment. Once you have then completed your training jobs, you can load the experiment and trials data into a pandas dataframe. This allows you to easily use plotting tools to visualize the results of your experiments.

In order to create our first trail or training job using Amazon SageMaker, we will need a few more things. The first is the URL of the S3 bucket where we have stored our training data and where we want to store our trained model (though these should not be the same prefix, they can be the same bucket if you need them to be), the second is the container or script where our training code is stored. We will cover more on containers and scripts for training later in the chapter, but for now, we will use one of the algorithms provided by SageMaker.

The last thing we will need is to decide the metric by which we will evaluate the quality of our model. Different models and uses will need different metrics to define the quality of the trained model. We will cover some here and give examples in the following code blocks, but make sure you carefully read the documentation of your chosen algorithm before starting training.

- **Absolute Values**:
    - **Accuracy**: It is the comparison between the number of accurately classified items and the total number of items.
    - **RMSE**: Root Means Squared Error is the distance between the vector of predicted values and the vector of observed values squared.
- **Relative Values**:
    - **AUC**: Area Under the Curve aggregates performance measurements across all possible classification thresholds.
    - **F1 Score**: It is the mean of precision and recall.

- **Ranking Measures**:
    - **NDCG**: Normalized Discounted Cumulative Gain is the normalized ratio of Discounted Cumulative Gain of the recommended order to the Discounted Cumulative Gain of the ideal order.
    - **MAP**: Mean Average position is the comparison of the ground truth bounding box to the detected box.
- **Algorithmic Properties**:
    - **Log-loss**: It refers to how close the predicted probability is to the corresponding actual true classification.

Additionally, you can use multiple metrics at once by identifying a composite metric that is the weighted sum of multiple individual metrics. Amazon SageMaker provides many built-in algorithms that come with various loss functions and metrics.

These are just a small sample of the methods of evaluating your chosen algorithm. If these seem confusing, don't panic! One of the fundamentals of Machine Learning is the iterative nature. Be prepared to sample different evaluation metrics to find out which one is suitable for your desired outcome. Like a good gourmand, we will sample a few in this chapter.

Setting Up Model Hyperparameters

Hyperparameters are the knobs and levers of machine learning models. They provide a way to finely control how your model behaves. The usual hyperparameter suspects are as follows:

- Number of Iterations or Epochs
- Learning Rate
- Momentum
- Regularization Rate or Lambda
- Mini-batch Size
- Storage and Compute Resources

To get optimal performance from your machine learning model, you need to carefully tune these hyperparameters. The goal is to find a combination that provides the best performance for your problem. The challenge is that this process can be quite time-consuming as it requires training your machine learning model multiple times with different hyperparameter combinations. Amazon SageMaker Experiments assists in not only grouping our trial iterations but also keeping track of those experiments, allowing us to analyze them and select the best one.

Additionally, it is important to carefully consider how you search the hyperparameter space. A brute force approach of trying every single combination is not recommended as it can be quite time-consuming and may never find the true global optimum. Instead, it is better to use a technique like Bayesian optimization, which can intelligently search the hyperparameter space.

There are many ways to tune machine learning models, but we will focus on using Amazon SageMaker's automatic model tuning functionality. This functionality uses Bayesian optimization under the hood and can be used to tune a wide variety of machine learning models.

Script vs Container

When training machine learning models, you have two main choices for how to run your training code:

- Running your training code in a script allows you to use any programming language you want. However, this can be slow as all code needs to be executed on the compute instances.
- Alternatively, you can choose to run your code in a container. A container is a self-contained environment that has everything needed to run your code, including the programming language and all its dependencies.

This can be faster as the compute instances only need to download and run the container.

Which option you choose will likely depend on what programming language you are most comfortable with and what is fastest for your problem. Amazon SageMaker supports both options and provides easy-to-use wrappers for both Python and R.

Training Data Storage and Compute

Training data is the bread and butter of machine learning models. It is the data that you use to train your machine learning model. Once you have trained your machine learning model, you will want to save it so that you can use it to make predictions. Saving your machine learning model is easy using Amazon SageMaker's built-in **`saveModel()`** functionality, identifying a model name. You can then reload your machine learning model at any time by calling the **`loadModel()`** function. In addition to saving your machine learning model, you will want to save your training data so that you can easily retrain your machine learning model if needed.

Amazon SageMaker makes saving your training data easy with the built-in functionality of Amazon S3. You can simply upload your training data to an Amazon S3 bucket and download it when needed. But why store it in S3 when we are just going to use it to train our model a few lines later? The answer lies in the power and flexibility that AWS gives us: abstraction.

When we run the code for our Machine Learning projects, we will usually be in an Integrated Development Environment of some kind. For instance, Eclipse, IntelliJ, Visual Studio Code. For our examples, we will continue to use Amazon SageMaker Studio, a cloud-based IDE that collects the various ML tools that SageMaker provides in one ML engineer-focused experience.

In our IDE of choice, we have some compute, represented by the CPU of the computer we are working on, or the compute attached to our SageMaker Studio instance. We also have some storage, represented by the hard drive of the computer we are working on or the storage attached to the SageMaker Studio instance. We use both for running our ML code and storing the libraries we need, but if we wanted to use them for training, we would have to either allocate the necessary resources to our IDE or buy a strong enough computer. These will work, but why pay for all that compute power if you are only going to use it while you are actively training your model?

Instead, we can leverage the flexibility and economies of scale that Amazon Web Services gives us and only allocate the resources we need for our IDE, abstracting our compute resources until we need them. Our compute resources are separate from our IDE, so we need a common storage location that is easily integrated with each: S3. Storing our training data in S3, we allow other teams, projects, and experiments to use the same data. We also get to be specific about the number of resources we leverage for our training, both vertically with larger instances (more processing and memory) and horizontally with additional instances.

Training Scenarios

In this chapter, we will train three different models, detailing the steps along the way. The code presented as follows is also in the GitHub repository linked from *Chapter 2, Hydrating the Data Lake*. Running these notebooks will incur charges in an AWS account, so make sure to clean up any resources that you end up allocating.

These examples also assume that you have already gone through the steps to understand your data and the feature engineering needed to refine the data and get it ready for the following models. If you need a refresher, read through *Chapter 3, Predicting the Future With Features*.

Linear Regression

Using our Wine Quality data set, we want to predict the quality of a white wine from the other attributes. Since this is predicting a numeric value, we can use the Amazon SageMaker built-in algorithm: XGBoost. XGBoost is a very popular ensemble algorithm, which means it uses a group of estimators from simpler models to create a faster and more accurate prediction.

That being said, let's dive into the start of our linear regression model. In the following examples, we are leaving out some of the starting parts where we import libraries, declare roles, list buckets, and other items that need to make the code work, but you can refer to the associated Jupyter Notebooks in the book's GitHub repository for the fully working notebook.

The first thing we do is set up our S3 client, download our dataset, read it into a pandas data frame, set the columns and look at the general details of our data:

```
s3 = boto3.client("s3")
!wget https://archive.ics.uci.edu/ml/machine-learning-
databases/wine-quality/winequality-white.csv
data = pd.read_csv("./winequality-white.csv", delimiter=";")
pd.set_option('display.max_columns', 20)
pd.set_option('display.max_rows', 50)
data.columns = [
"fixed acidity",
"volatile acidity",
"citric acid",
"residual sugar",
"Chlorides",
"free sulfur dioxide",
"total sulfur dioxide",
"Density",
"pH",
"Sulphates",
"Alcohol",
"quality"
]
print(data.shape)
display(data.head())
display(data.describe())
display(data.diagnosis.value_counts())
```

Now that we have the data loaded into a data frame, we can split it into test, train, and validation data sets:

```
train_data, validation_data, test_data =
np.split(data.sample(frac=1, random_state=5621), [int(0.7 *
len(data)), int(0.9 * len(data))])
```

XGBoost expects our data in CSV format without headers. For the validation data, it will throw errors if the feature we are predicting is included, so we drop that as well:

```
pd.concat([train_data['y_yes'], train_data.drop(['y_no',
'y_yes'], axis=1)], axis=1).to_csv('train.csv', index=False,
header=False)
pd.concat([validation_data['y_yes'],
validation_data.drop(['y_no', 'y_yes'], axis=1)],
axis=1).to_csv('validation.csv', index=False, header=False)
```

This does most of the work to set up our training and our environment, but now we get into the model hyperparameters. Recall that hyperparameters are configurations that are external to the model itself and the value of which cannot be estimated from data. This is counter to parameters, which are configurations that are internal to the model and can be estimated from data.

It is a best practice to always review the documentation carefully for the algorithm's hyperparameters, even for models you use frequently. Model versions change regularly, and the ranges, defaults, and number of hyperparameters can change.

For our chosen XGBoost algorithm, the initial hyperparameter values are as follows, with explanations:

- **`max_depth`**: It defaults to 6 and determines the depth of a given tree. XGBoost is a Decision Tree type of model, so we want to limit the depth (number of decisions from the tree root) that the model will iterate through. Too shallow and our model will not have sufficient complexity to learn; too deep and our model will iterate over minimal improvements. The higher you set this value, the more likely it is that your model will learn too much from your training data or become overfit.
- **`eta`**: This refers to how much shrink is applied at each step in order to prevent overfitting. It defaults to 0.3 and should be adjusted when you need to react to potential or actual overfitting.
- **`gamma`**: The lowest loss reduction that is acceptable to make another leaf node of the tree. It is used to determine when sufficient leaf nodes (branches in a sense) of your decision trees have been made. It defaults to 0, but we want our model to be a little conservative.
- **`min_child_weight`**: It is the lowest possible sum of an instance weight needed in a child node. In our case, this sets the lowest number of instances needed in each node. The default is 1, but we are setting it higher to again add some conservation in our model.
- **`subsample`**: This is the percentage of the training data that is sampled on the training instance. Setting it to 0.6 means that we are sampling 60% of our data. This is primarily used in preventing overfitting.
- **`verbosity`**: It controls the logging behavior of the model. We can set it to 0 for silent, 1 for warning messages, 2 for info messages, and 3 for debug messages, in that order for volume of messages. We will typically set this higher when we start our experiments to get the most logging and then lower it as our model performs as expected.
- **`objective`**: This is one of the more important hyperparameters. It specifies the earning task and the associated learning objective,

defining how we determine the training quality of our model. There are nearly 20 different objectives for XGBoost alone, and we are using **`re:squarederror`** for a linear regression using RMSE.

- **`num_round`**: This is the total number of rounds to run the training, a required parameter. We can limit this to check everything, but we will want to adjust it to get to the optimum evaluation metric value.

We can now set our hyperparameters for our model as follows.

```
Hyperparameters = {
   "max_depth":"4",
   "eta":"0.3",
   "gamma":"3",
   "min_child_weight":"7",
   "subsample":"0.6",
   "verbosity":"1",
   "objective":"reg:squarederror",
   "Num_round":"50"}
```

The next thing we can do is set the container we are going to use for our training environment. We could use our own script, but since XGBoost is a built-in SageMaker model, not only can we use the environment provided but can also use the managed container provided for this use. We don't need to know what container registry holds the model, just the region we are working in (**`region_name`**), the framework for the container we want to use (framework) and the version of the container we want (version). Versions are updated regularly, so we are going to use "latest" here, but always check which versions are available and, if your training suddenly stops working or changes behaviors and you are using "latest", check what version that is and if anything has changed.

```
Container =
sagemaker.image_uris.retrieve(region=boto3.Session().region_nam
e, framework='xgboost', version='latest')
```

Next, recall that we are not going to train our model on our local compute; we are going to use an instance we spin up just for this purpose. We need to tell that instance where to get our training, testing, and validation data:

```
s3_input_train =
sagemaker.inputs.TrainingInput(s3_data='s3://{}/{}/train'.forma
t(bucket, prefix), content_type='csv')
```

```
s3_input_validation =
sagemaker.inputs.TrainingInput(s3_data='s3://{}/{}/validation/'
.format(bucket, prefix), content_type='csv')
```

Next, we will create our Estimator, set our hyperparameters, and call the **`fit`** function to start our training, providing our validation data set:

```
session = sagemaker.Session()
xbg_estimator = sagemaker.estimator.Estimator(container, role,
instance_count=1, instance_type='ml.m5.xlarge',
output_path='s3://{}/{}/output'.format(bucket, prefix),
sagemaker_session=session)
xbg_estimator.set_hyperparameters(hyperparameters)
xgb_estimator.fit({'train': s3_input_train, 'validation':
s3_input_validation})
```

This command will train our model and store it in the **`output_path`** of our estimator. Remember, the result of our training job will also be stored in SageMaker, under the Training Jobs. This is vital to referring to the results of your experiments, especially the ones you do just before the weekend.

Once our model is trained, the instance we created for that purpose will be shut down, minimizing the charges we incur for it. We can observe the evaluation metric we chose and see if we want to keep it or adjust our hyperparameters and run the training again. This process of observing results and adjusting either our hyperparameters or input data until we get close to the results we want is very typical and expected. But what if we could shorten that iteration using the power of AWS? Of course, we can!

The previous command will list the output of our training model and report the total training time and the billing time that the training took. In order to see the overall inference performance of our model, we will need to deploy it and send data that the model has never seen before against it, which we will do in *Chapter 8, Model Deployment Strategies*. At the moment, we continue to focus on the training phase. We have a model, but how do we know if the hyperparameters are what they should be? As mentioned earlier, we can train, adjust, train, adjust, train, and continue this process until we get the best possible results.

Alternatively, we can tell AWS to do all that for us. Let's continue in the notebook to find out how we can automate the model hyperparameter tuning.

The first thing we do is declare the hyperparameters we want to tune, their parameter types, and their ranges:

```
from sagemaker.tuner import IntegerParameter,
CategoricalParameter, ContinuousParameter, HyperparameterTuner
hyperparameter_ranges = {‘alpha’: ContinuousParameter(0, 2),
‘min_child_weight’: ContinuousParameter(1, 10), ‘subsample’:
ContinuousParameter(0.5, 1), ‘eta’: ContinuousParameter(0, 1),
‘num_round’: IntegerParameter(1, 4000)
        }
```

The hyperparameters that you can tune, their types, and their ranges are typically found in the respective algorithm documentation, but we chose these because they have the highest impact on our model. The following code identifies the evaluation metric and the metric type:

```
objective_metric_name = ‘validation:rmse’
objective_type = ‘Minimize’
```

We’re choosing the Root Mean Squared Error because we want to minimize the distance between the predicted values and the actual values. We want the hyperparameters to result in an aggregated (means) prediction as close to our actual values as possible. We also identify that this metric is meant to minimize the difference:

```
xgb_hp_tuner = HyperparameterTuner(xgb_estimator,
objective_metric_name = objective_metric_name, objective_type =
objective_type, hyperparameter_ranges = hyperparameter_ranges,
max_jobs=5, max_parallel_jobs=5)
xgb_hp_tuner.fit({‘train’: s3_input_train, ‘validation':
s3_input_validation})
```

Here, we create the tuning job object, feeding in our variables, but we also set the number of tuning jobs we want to run at once and the maximum total number of jobs. This very clearly shows the awesome power that AWS can leverage for Machine Learning projects. We can run five different variations of our hyperparameters all at once. We could, of course, run them in sets of 2, 3, or 6, and set our total jobs to 10, 20, or even 30 jobs, but recall that in this chapter, we are an experienced data scientist and, more importantly, a Sagittarian. We have a history of getting things done right now.

The last part of the code shows how to automatically select the best set of parameters among those tuned:

```
xgb_hp_tuner.best_training_job()
```

We simply call the SageMaker SDK asking for the **best_training_job()** and put the results in a variable for use later. We could use that result to deploy our model to run inference if we wanted with a single line of code as shown below, but we will cover deployment in detail in *Chapter 8, Blue or Green*:

```
xgb_endpoint= xgb_hp_tuner.deploy(initial_instance_count=1,
instance_type="ml.m5.xlarge")
```

First task well in hand, the data scientist takes a break to check in on their assigned ticket backlog.

Natural Language Processing

Natural Language Processing is during something of a renaissance. Not too long ago, **Natural Language Processing** (or **NLP**) tasks were very challenging, time-consuming, and error prone. This mostly was due to the nature of language. Even if we break down a sentence into numerical values and graph the relationships of all the letters or words, those values can change wildly. Consider the word *set*. According to the Second Edition of the Oxford English Dictionary, *set* has 430 different meanings. Even more challenging is that the order of the words in a sentence can change their meaning and ability to be understood.

Rather than trying to understand language in a way a machine could understand it, researchers tried to do the opposite: define language the way humans understand it. This was the inception of self-attention, which is the process of differentially weighing each part of your training data. Self-attention has been formalized in Transformers as a type of learning model that uses it to process Natural Language Processing and other tasks.

Some of the unusual characteristics of Transformers is that they do not read a sentence from beginning to end but can provide context of any word in the sentence, allowing each word to have its weight and context reviewed at the same time.

This new type of model was a massive leap forward for Natural Language Processing tasks, but it was Hugging Face that provided a community hub to organize not only models but also data sets, documentation and implementations. Hugging Face democratized Natural Language Processing, making the use of Transformers as streamlined as single lines of code.

Upon checking their ticket backlog, our data scientist is surprised to see that at least a dozen have been assigned in the last day. They could sort by assignment date, priority, or the requester, but what if there was a way to analyze the contents of the tickets and get the sentiment, using it to determine actual severity? There is!

As with the linear regression model, we are skipping the steps of setting up libraries, modules, buckets, and so on, in the sake of saving page space, but you can find the full notebook in the GitHub repo associated with this book.

We start by setting a few variables that we will reference later, the most important one being the name of the Hugging Face tokenizer we will be using:

```
dataset_name = 'emotion'
num_labels=6
tokenizer_name='bert-base-uncased'
```

Bert is a model that has been pre-trained on a large amount (or corpus) of English data. We can use that pre-training to save ourselves a lot of training on the basics, and then use transfer learning to further train it on our chosen data set. The model is also uncased because it does not differentiate between cases. This also works for our usage since we can't be sure if someone writes urgent or URGENT, but we should generally treat them the same. Lastly, we are using the Hugging Face emotion data set. This data set is composed of Twitter messages (the text) and the associated basic emotions of those messages (the label). Since we want to identify the emotion of our ticket backlog, this data set should suit our needs.

```
dataset = load_dataset(dataset_name)
tokenizer = AutoTokenizer.from_pretrained(tokenizer_name)
def tokenize(batch):
  return tokenizer(batch['text'], padding=True, truncation=True)
dataset_encoded = dataset.map(tokenize, batched=True,
batch_size=None)
```

Next, we use the **`load_dataset`** from the datasets library to load the dataset. Then we need to break our text into smaller components and prepare it for our model training. This is called tokenizing, and we are using a pre-trained method that is included in the Hugging Face library:

```
model =
(AutoModelForSequenceClassification.from_pretrained(tokenizer_n
ame, num_labels=num_labels))
dataset_encoded.set_format("torch", columns=["input_ids",
"attention_mask", "label"])
Dataset_encoded["train"].features
```

Here, we create the model object including the tokenizer object, setting the format for PyTorch, and set the training data set from its features:

```
hyperparameters = {
  'model_name_or_path':tokenizer_name,
   'task_name': 'mnli',
  'output_dir':'/opt/ml/model',
}
```

These are the hyperparameters for this model, specifically, the name of the Hugging Face tokenizer we defined earlier, the task name, and the output directory of the model. We are using a Multi-Genre Natural Language Inference as a training task since it is associated with our chosen emotion data set. We also use the specific output directory since that is where the model will be exported from once our training instance has finished and before it is terminated:

```
git_config = {'repo':
'https://github.com/huggingface/transformers.git','branch':
'v4.17.0'}
huggingface_estimator = HuggingFace(
 entry_point='run_glue.py',
 source_dir='./examples/pytorch/text-classification',
 instance_type='ml.p3.2xlarge',
 instance_count=1,
 role=role,
 git_config=git_config,
 transformers_version='4.17.0',
 pytorch_version='1.10.2',
 py_version='py38',
 hyperparameters = hyperparameters,
  train_dataset=dataset_encoded["train"],
  eval_dataset=dataset_encoded["validation"]
)
```

Here we do something interesting. Instead of identifying a local training script or a pre-generated container, we specify a managed script in a remote Git repo. Hugging Face will download the script that we specify in the repo's branch and location before beginning training. This is especially useful when using frameworks like Hugging Face. We also want to use a single P3 2x instance to train our model; P3 instances are perfectly suited for Deep Learning and Natural Language Processing that require GPU. Additionally, we set the versions of our libraries, add in our hyperparameters, and specify the train and the validation data sets:

```
huggingface_estimator.fit()
```

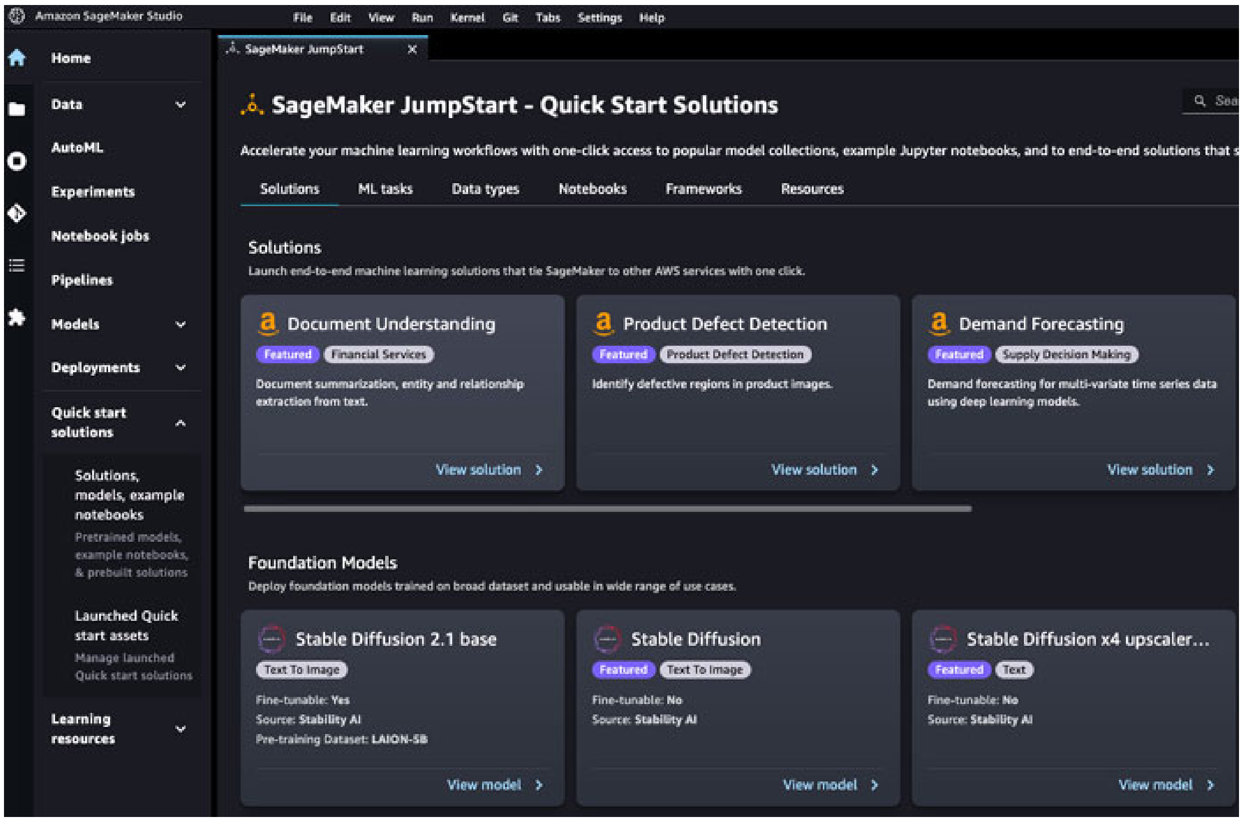
With this final line, we start our training job. Once it finishes, we can deploy the model and test it before iterating over our tickets to find their primary emotions.

Image Classification

Having managed at least the priority and disposition of their task list, our friendly data scientist can take a moment to review the assigned ticket that was identified as not only emotionally charged but with an alarming subject: "Image Classification Demo Tomorrow!"

Opening the ticket and reviewing further, one of the company's Vice Presidents had lunch with their executive friends and heard things about "all of this Machine Learning stuff"; they wanted a demonstration of how it could be used to identify objects in images.

Thinking quickly, the data scientist remembers something their AWS solutions architect mentioned in their last office hours. Logging in to their Amazon SageMaker Studio instance, they click on the JumpStart icon on the left panel launcher, and then on the **`Browse JumpStart`** button. The following image shows the SageMaker Studio Jumpstart initial options and icon:



SageMaker JumpStart - Quick Start Solutions

Accelerate your machine learning workflows with one-click access to popular model collections, example Jupyter notebooks, and to end-to-end solutions that

Solutions

Launch end-to-end machine learning solutions that tie SageMaker to other AWS services with one click.

Document Understanding

Document summarization, entity and relationship extraction from text.

Product Defect Detection

Identify defective regions in product images.

Demand Forecasting

Demand forecasting for multi-variate time series data using deep learning models.

Foundation Models

Deploy foundation models trained on broad dataset and usable in wide range of use cases.

Stable Diffusion 2.1 base

Fine-tunable: Yes
Source: Stability AI
Pre-training Dataset: LAION-5B

Stable Diffusion

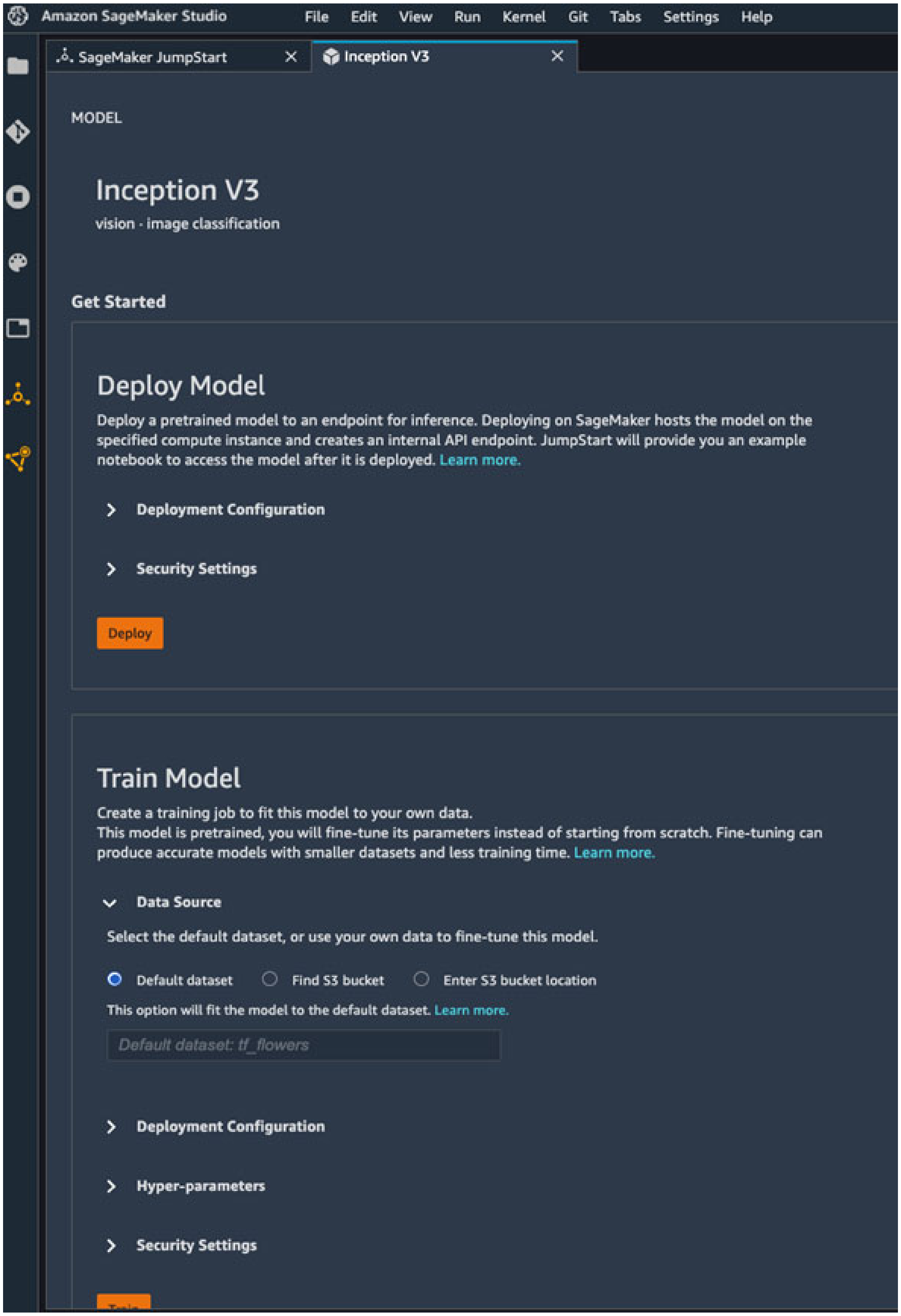
Fine-tunable: No
Source: Stability AI

Stable Diffusion x4 upscaler...

Fine-tunable: No
Source: Stability AI

Figure 6.1: *SageMaker Jumpstart*

There are several solutions listed on the page, but our data scientist sees one that, being a movie fan, catches their eye: Inception V3. A quick internet search confirms that this is a model used for image classification on the ImageNet data set, which will work perfectly in this case. This model is hosted on the TensorFlow Hub, and clicking on **`InceptionV3`** and reviewing the model's landing page gives two options: deploy the model and fine-tune the model. The following image gives an example of using SageMaker Jumpstart in SageMaker Studio to train and deploy an image classification model:



MODEL

Inception V3

vision · image classification

Get Started

Deploy Model

Deploy a pretrained model to an endpoint for inference. Deploying on SageMaker hosts the model on the specified compute instance and creates an internal API endpoint. JumpStart will provide you an example notebook to access the model after it is deployed. Learn more.

> Deployment Configuration

> Security Settings

Deploy

Train Model

Create a training job to fit this model to your own data.
This model is pretrained, you will fine-tune its parameters instead of starting from scratch. Fine-tuning 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: tf_flowers

> Deployment Configuration

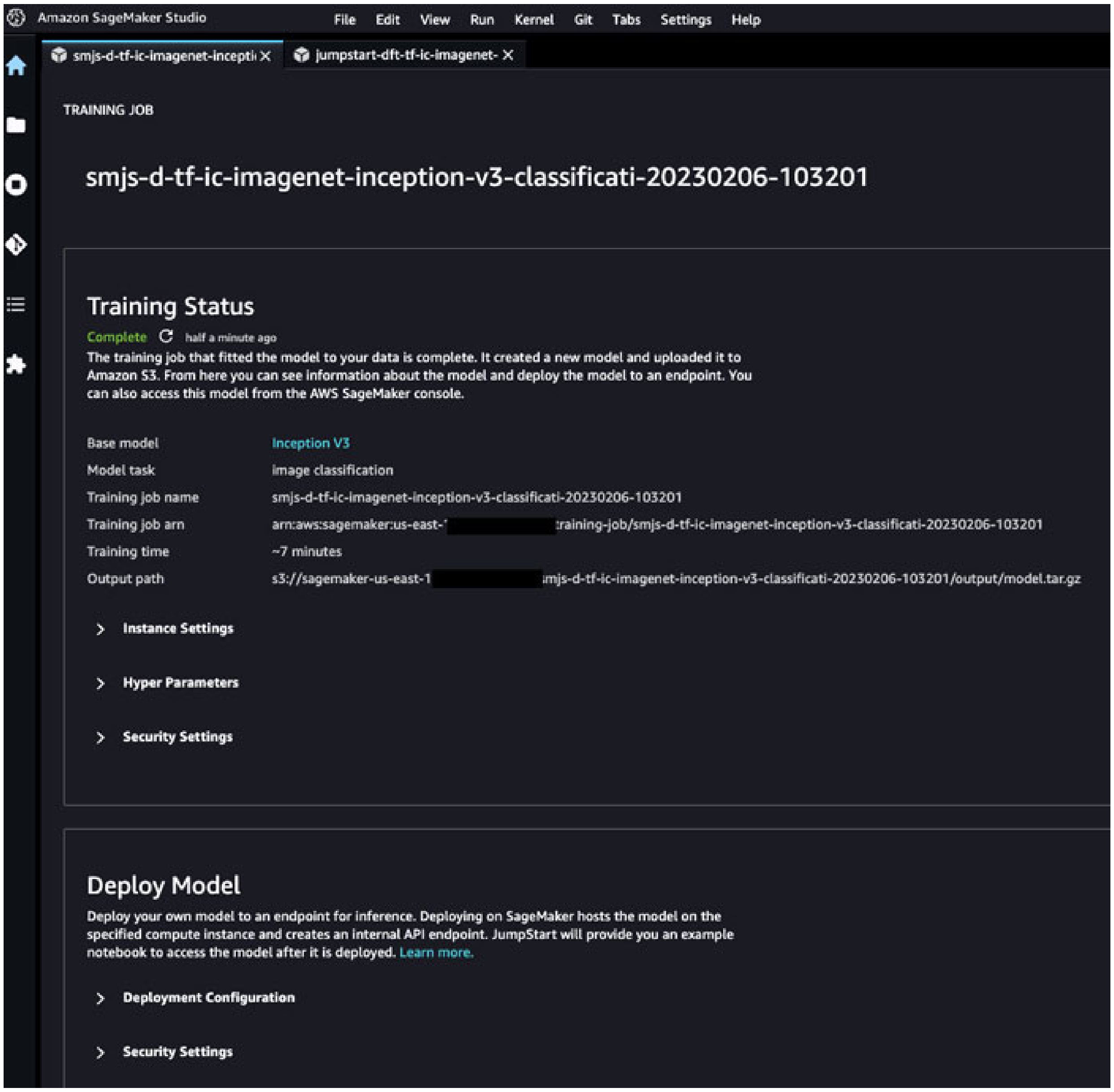
> Hyper-parameters

> Security Settings

Figure 6.2: *SageMaker Jumpstart options*

We will cover more details on this model later in this chapter, but the dataset the model was trained on, and instructions on how to use the model for inference. The option to fine-tune the model is tempting, but the demo looms close, so our data scientist chooses the option to Deploy the Model. Amazon SageMaker Jumpstart will show an update page while the model is deploying, with information about the deployment, which should take a few minutes.

Once the model is deployed, there will be options to delete the endpoint or open a Jupyter Notebook to repeat the steps. The following image shows the options for the image classification model deployment:



TRAINING JOB

smjs-d-tf-ic-imagenet-inception-v3-classificati-20230206-103201

Training Status

Complete half a minute ago

The training job that fitted the model to your data is complete. It created a new model and uploaded it to Amazon S3. From here you can see information about the model and deploy the model to an endpoint. You can also access this model from the AWS SageMaker console.

Base model	Inception V3
Model task	image classification
Training job name	smjs-d-tf-ic-imagenet-inception-v3-classificati-20230206-103201
Training job arn	arn:aws:sagemaker:us-east-[redacted]training-job/smjs-d-tf-ic-imagenet-inception-v3-classificati-20230206-103201
Training time	~7 minutes
Output path	s3://sagemaker-us-east-1[redacted]mjs-d-tf-ic-imagenet-inception-v3-classificati-20230206-103201/output/model.tar.gz

> Instance Settings

> Hyper Parameters

> Security Settings

Deploy Model

Deploy your own model to an endpoint for inference. Deploying on SageMaker hosts the model on the specified compute instance and creates an internal API endpoint. JumpStart will provide you an example notebook to access the model after it is deployed. Learn more.

> Deployment Configuration

> Security Settings

Figure 6.3: *SageMaker Jumpstart Model Deploy Options*

Our data scientist finishes the training of the image classification model, deploys it to an endpoint and tests it out against a few images. Satisfied that they are ready for their demo, they begin to ponder what something similar looks like without SageMaker Jumpstart. They absolutely could just change some of the values in the provided Notebook, but there is significant appeal in seeing a similar problem solved another way.

Image Classification Round 2

Our data scientist, having enjoyed working on the ticket summarization solution, wonders if it would be possible to use a Machine Learning model to classify images. You see, they have a local club they belong to, and other members are posting images to a central shared repository. These images are supposed to be of various kinds of aircraft. The problem is that a good number of the uploaded images are of random other things. If you can train a model to predict numbers and text, can you train one to look at an image and determine what it is?

Yep, you sure can. To accomplish this, we will be using an image classification model. Relevant parts of the code are reviewed as follows, but like in the other examples, you can check the notebook in the book's GitHub repo for the full code:

```
training_image = sagemaker.image_uris.retrieve(
  region=sess.boto_region_name, framework="image-classification"
)
print(f'Container: {training_image}')
```

Like before, we create a **`training_image`** object containing the Amazon SageMaker managed image-classification container location.

We will also use the CalTech 101 data set to train our model. This data set included 101 classes of images and some background noise. In the interests of time, we will only train on a few of these classifications, but there is no stopping you from taking the time to train with them all:

```
s3 = boto3.client("s3")
s3.download_file(
  "sagemaker-sample-files",
  "datasets/image/caltech-101/101_ObjectCategories.tar.gz",
  "ObjectCategories.tar.gz",
)
```

We also use the openly available **`im2rec.py`** script to convert the images from their default format to the **`RecordIO`** format. **`RecordIO`** is used in this case to encode the image data and the associated metadata into stream format, which is more optimized for disk operations. If we left the images in their native format, getting those images to the training instances might be slower than the GPUs can process the training batches. Using GPUs can be expensive, so taking effort to optimize how long they are needed is worth your time, especially when you are using distributed training methods:

```
!python im2rec.py --list --recursive caltech-101-train
caltech_101_train/ | sort
```

We do this with the images and the image categories. We could divine this from the folder names themselves, but we can also keep a separate list just in case the folder names have characters that we do not want in predictions like underscores or numbers.

Here, we create the estimator object for image classification:

```
s3_output_location = "s3://{}/{}/output".format(bucket, prefix)
ic_estimator = sagemaker.estimator.Estimator(
  training_image,
  role,
  instance_count=1,
  instance_type="ml.p3.2xlarge",
  volume_size=50,
  max_run=360000,
  input_mode="File",
  output_path=s3_output_location,
  sagemaker_session=sess
)
```

We feed in the training image and the SageMaker IAM role we created previously. We also identify the number of training images and the instance type. In this case, P3 instances are appropriate because we need to leverage GPUs to train our image classification model. AWS' P3 instances have NVIDIA V100 Tensor Core GPUs, which means our training will be faster and more efficient. We also set an appropriate EBS volume size (50 Gb), so our entire image library can be loaded into the image. We also set a max timeout for training in seconds, an input mode, where we want the model to be saved after training, and the SageMaker session.

Now we can set the model's hyperparameters:

```
ic_estimator.set_hyperparameters(
  num_layers=18,
  use_pretrained_model=1,
  image_shape="3,224,224",
  mini_batch_size=128,
  epochs=2,
  learning_rate=0.01,
  top_k=2,
  resize=256,
  precision_dtype="float32",
)
```

We first set the number of layers for the neural network. We set a low number for the layers to speed up training, but this hyperparameter can have a significant impact on your training time, so take a bit to adjust and see what the results on your accuracy metric will be. We also set the value of **`use_pretrained_model`** to 1 to set the first layer of our neural network model to be pre-trained. The other layers will have randomized initial weights, but the first one will be pre-set. The next hyperparameter is **`image_shape`**; it is a combination of the number of layers, and the height and width of the images. The number of channels is standardized at three, but the image dimensions details are static across all of your input images, so it is important to pre-process your images.

If your input image is smaller than the listed dimensions, the training will fail. If it is larger, your image will be cropped down to the listed dimensions. For the **`mini_batch_size`**, this is the number of training samples each GPU will handle. Next, we set number of epochs or the number of passes through the neural network with all training data. This is another hyperparameter where adjusting can have a significant impact, so be prepared to tune here.

Next, we set **`learning_rate`**, which is the degree with which individual node weights are updated during training. This can typically be set between 0.0 and 1.0, and the value controls how quickly the model adapts to the presented problem. A lower value may require more epochs before learning to understand the training data. You may be tempted to set a very high learning rate, but the higher the rate, the larger the risk of your model converging too quickly. Similarly, too low a rate can mean your model never actually learns to understand your data. We have identified a few other hyperparameters that are useful for adjusting in your training experiments, but **`learning_rate`** may be the one that can have the greatest impact on your model's training time and eventual accuracy.

The next hyperparameter reports the **`top_k`** accuracy during training. During training, the model will make a number of guesses with regard to the classification of the current image. The value set for **`top_k`** reports that the classification is true if it is in the top *k* guesses.

We also set the resize hyperparameter to the pixels on the shortest side of our input images. This should also be set to a value larger than the height and width of the **`image_shape`** values.

The last is the **`precision_dtype`**, which is the precision of the weights used for training. For this algorithm, you can use either float16 for half precision or float32 for full. Float16 can be useful when lower memory footprint is needed.

Lastly, we upload our training and validation data, and then call fit to start the training job:

```
ic_estimator.fit(inputs=data_types, logs=True)
```

Once the training job finishes, we can deploy it and download some images from a search service to use as test data.

Conclusion

Machine Learning workflows can typically be broken into three parts: Model Training, Model Registration, and Model Delivery. Each one can be operated independently. This chapter focused on the first phase: Model Training. There is a key purpose in breaking these parts of the workflow apart, mostly because they should be iterated separately. It is common for the training phase to continue even after your model is moved into a live production environment. You could fill a whole bookcase to cover all the various training methods of all the algorithms of all the various tasks currently in existence. Instead, we took a small sample of different methods for training models to whet your appetite. From here, the experimentations continue with different training methods.

In this chapter, we took on the persona of a data scientist who wanted to expand their knowledge of Machine Learning. In pursuit of this goal, they used their existing knowledge to prepare data for machine learning models, and then used some of the tools offered by AWS and SageMaker to train, tune, and evaluate their models. They looked at what hyperparameters can do and some methods of tuning them to achieve the desired goals for machine learning. Finally, they took their trained models and deployed them to run test inference against them.

In the next chapter, we will leave our data scientist behind, safe in the knowledge that they have taken significant steps towards their goal.

In *Chapter 7, Let George Take Over*, we will take a look at different methods of automating the feature engineering, model training, and model testing phases of a Machine Learning project. Most importantly, we will examine when it is tactically advantageous to use automated methods and when it is better to proceed through the iterations yourself.

Points to Remember

As you progress in your Machine Learning journey, remember that building a complete solution is an iterative one.

In the first section, Model Training, examine the problem you are trying to solve and review the available models, evaluation metrics, and hyperparameter ranges associated with the model. Be prepared to experiment with different combinations to achieve the results you need.

Once you have a candidate model that meets your evaluation criteria, store the model in a location that allows you to record the associated experimentation details, such as evaluation metric, link to training data, and metadata.

Finally, once you are ready to deploy your model, consider the compute and storage requirements to determine the convergence between resources, speed, and cost optimization.

Multiple choice questions

1. **Which of these is not a type of model you can train on AWS SageMaker?**
    a. Reinforcement
    b. Semi-supervised
    c. Unsupervised
    d. Supervised
2. **Which of these is not a model deployment option on AWS SageMaker?**
    a. Real-time
    b. Batch

    c. Asynchronous
    d. Remote

3. **Which of the following is the AWS SageMaker's built-in method for saving your model?**
    a. retainModel()
    b. keepModel()
    c. storeModel()
    d. saveModel()
4. **The AWS Linear Learner algorithm is best used for which kind of problems?**
    a. Regression
    b. Repression
    c. Random
    d. Representation
5. **What kind of Natural Language Processing architectures use self-attention?**
    a. Autobots
    b. Unicorns
    c. Transformers
    d. Deep Learning

Answers

1. **b**
2. **d**
3. **a**
4. **a**
5. **c**

Further Reading

- *Kaggle is a gamified Machine Learning Platform*: **https://www.kaggle.com/alexisbcook/getting-started-with-kaggle**
- *Hugging Face: Quick Tour*: **https://huggingface.co/docs/transformers/quicktour**
- *Amazon SageMaker Training Documentation*: **https://docs.aws.amazon.com/sagemaker/latest/dg/train-model.html**
- *Ready to jump into a project?* **https://github.com/topics/machine-learning-projects**

CHAPTER 7
Let George Take Over (AutoML in Action)

Introduction

In *Chapter 5, Casting a Deeper Net*, we briefly discussed how an airline pilot lines up the plane for landing in a "final approach", a straight-line route to the runway with the pilot reducing speed and altitude until the plane's wheels are down. We used this metaphor to talk about ML projects making a final commitment to spend time and money to run model training experiments. Depending on your business requirements, the skillset of your teams, and the amount of time and money you have to complete your project, you may choose to either perform feature engineering, algorithm selection, model training and tuning on your own, or as we will see in this chapter, you may select a technique called AutoML. As the name indicates, AutoML stands for automated machine learning, which is a bit of an oxymoron according to the author because it represents the deterministic execution of a set of tasks to achieve probabilistic outcomes. In short, imagine a program that has the ML workflow steps as its functions, and these are executed without manual intervention from data processing to model training to deployment. What does this mean for our ML development project?

To understand how AutoML revolutionized traditional ML projects, let us go back to our airplane example. Though no one really knows why, in the aviation industry, the autopilot is affectionately referred to as "George". When you get on a plane, the pilots are flying the plane manually only during take-off and sometimes during landing. As soon as the plane reaches an altitude of 10K feet, they will most likely switch on the autopilot or "let George take over" the plane. So, whenever you fly, know that a highly trained intelligent ML model is the one that's actually flying the plane, not the pilots. What does autopilot control? Do pilots actually do anything while "George" is in control? When the plane is in the cruising stage of the journey, the controls that matter are heading (the angle/degree/direction the plane is moving in/toward), speed, and the altitude. The autopilot knows the destination and the plane's position, measures the value for these controls (heading, altitude, speed) continuously, and frequently updates them based on the prevailing conditions. The pilots meanwhile listen to the air traffic control inputs from stations in the path of their journey and provide updated instructions to the autopilot if needed. Autopilots today are highly advanced AI systems and can fully land a plane. Current AutoML technologies are equally advanced with regard to what they can do with your ML workflows.

In the previous chapters, we dove deep into the data collection and model training stages of the ML workflow, learned about data lake storage in AWS, learned the importance of feature engineering tasks, built our own feature extraction solutions for common ML domains, and automated data orchestration pipelines for our ML workloads. We also learned about algorithms and neural networks and saw how to set up model training in Amazon SageMaker with a working example. In Chapter 5, Casting a Deeper Net, we also discussed that the steps in the ML workflow will differ if we run model training ourselves as compared to using AutoML. In this chapter, we will learn what those differences are and discuss the common AutoML solution frameworks like Amazon SageMaker AutoPilot (**https://aws.amazon.com/sagemaker/autopilot/**) and AutoGluon (**https://auto.gluon.ai/stable/index.html**). Further on, we will build our own end-to-end AutoML solution for a tabular regression use case.

Structure

In this chapter, we will dive deep into the following topics:

- Running AutoML with Amazon SageMaker Canvas
- Using AutoGluon for AutoML

Objectives

In this chapter, we will learn how to use two popular AutoML services with working examples. First, we will read how to use Amazon SageMaker Canvas (**https://aws.amazon.com/sagemaker/canvas/**) and SageMaker Autopilot to leverage the power of AutoML for a point and click low code/no code approach for running end-to-end ML workflows.

Then, we will explore another popular AutoML framework called AutoGluon, and we will use a SageMaker notebook to try, with hands-on instructions, how to automate your ML projects using AutoGluon. As we learn these concepts, we will also briefly talk about how AutoML is prevalent in other areas of the AWS AI/ML stack.

Running AutoML with SageMaker Canvas

Automation is a key motivator for all our inventions. We are constantly looking for ways to reduce manual effort, complete our tasks quicker, cut costs, and improve efficiencies. For example, think of the washing machine. What used to take hours of effort (washing clothes by hand) and significant manual exertion is now done with no effort in less than half the time, improving our quality of life. It's the same thing with technology. You may have built the greatest product, but if it is complex and requires a lot of manual effort to maintain, it may not see much success. We live in a fast-paced world that requires us to finish our projects not only with successful outcomes but also quickly. Traditional ML was the playground of scientists and researchers who had the luxury of time and funding to be deliberate in training, tuning, and validating models, but this changed after cloud computing. AI/ML is now mainstream, and automation is the need of the hour. The philosophy for automating ML workflows is quite simple. Just identify the tasks that require undifferentiated heavy lifting (data processing, transformation, feature engineering, allocation of compute), that are repetitive (iterative model training and tuning), that can be inferred (if a continuous nominal feature is to be predicted, it is a regression problem). Then, bundle them up together and provide means for execution without the need for manual intervention. SageMaker Autopilot (**https://aws.amazon.com/sagemaker/autopilot/**) does exactly that. It is a fully managed feature in SageMaker with automated low-code ML workflow executions for tabular and time series forecasting problems. Refer to the following image to understand the key capabilities of the ML workflow that is automated in SageMaker Autopilot and is completely transparent to the user:

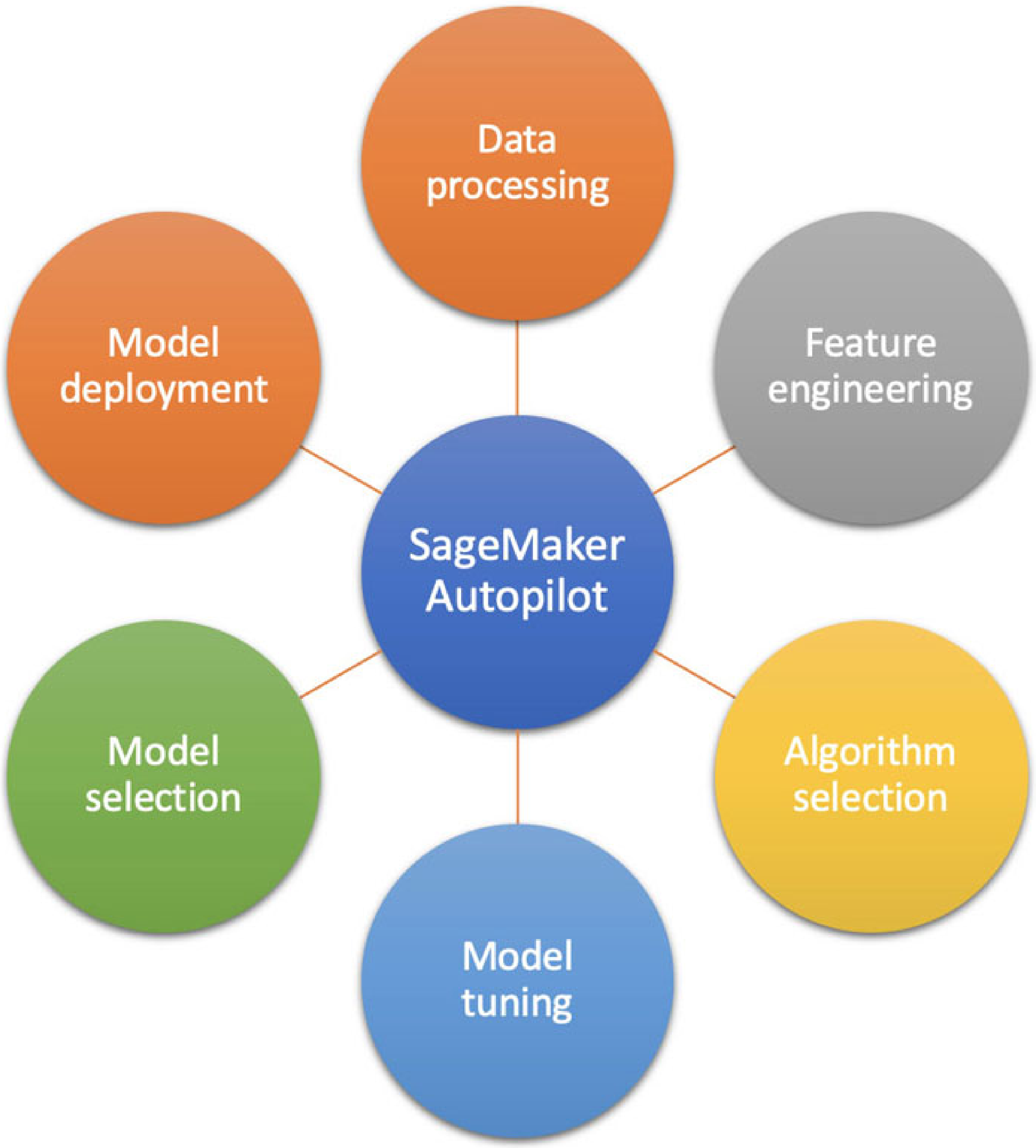


Figure 7.1: *ML workflow steps automated by SageMaker Autopilot*

Amazon SageMaker Canvas (**https://aws.amazon.com/sagemaker/canvas/**) uses Autopilot under the hood. SageMaker Canvas is a fully managed no-code ML solution with an interactive UI for building and running powerful ML models that you can use for various tasks. You can complete your entire ML workflow with just point-and-click interactions in Canvas. The instructions in this section will walk you through how to use Canvas with our wine features dataset to predict the price for a bottle of wine by directly working with the raw dataset.

Before we get started with using Canvas, we need to get our raw dataset and load it into an Amazon S3 bucket. In case you haven't signed up for an AWS account yet, now is the time to do so. Follow the instructions in the Setting up your AWS account section in *Chapter 2, Hydrating Your Data Lake*, to sign up for an AWS account. Once you have signed up, log in to your AWS account using the instructions here: **https://docs.aws.amazon.com/IAM/latest/UserGuide/console.html**. If you have not already done so, create an Amazon S3 bucket (**https://docs.aws.amazon.com/AmazonS3/latest/userguide/create-bucket-overview.html**). Now, go to the book's GitHub link, provided at the beginning of the book, and download the file - `Chapter-07/wine_canvas_ds.csv`. You can do this by going to the link, and in the top middle part of the page, clicking on View raw; then, right-click and save as, and when the file name appears, add the extension `.csv` to store the file in your local computer. Now, go to the Amazon S3 console in AWS (simply type S3 in the services search bar and select S3), create a folder called `chapter7`, and upload this file. Now we are ready to import our dataset and build our ML model. Execute the following instructions to proceed:

1. In the AWS Management Console, type `SageMaker` in the services search bar and select `SageMaker` to navigate to its console. In the SageMaker console, select `Canvas` under `Getting started` from the left menu pane, as shown in *Figure 7.2*:

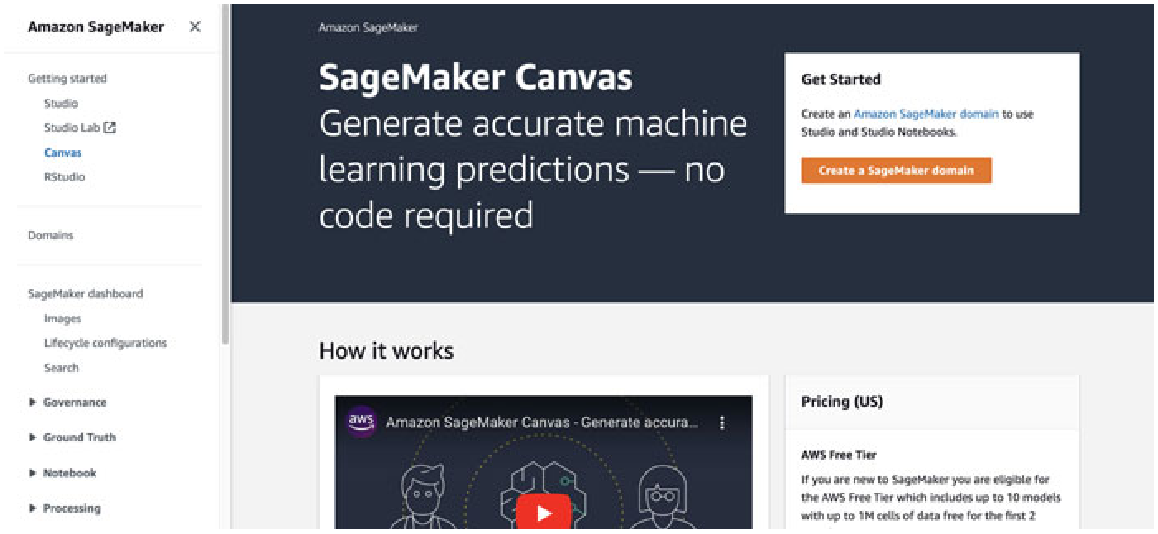
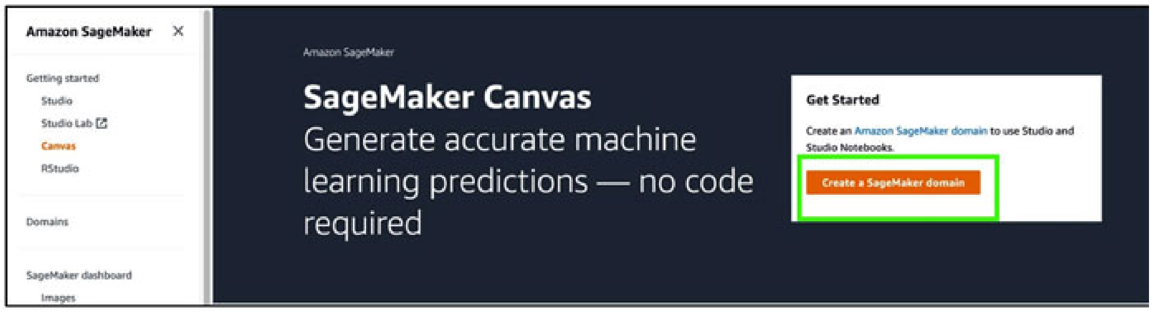

Amazon SageMaker
Getting started
Studio
Studio Lab
Canvas
RStudio
Domains
SageMaker dashboard
Images
Lifecycle configurations
Search
Governance
Ground Truth
Notebook
Processing
Amazon SageMaker
SageMaker Canvas
Generate accurate machine learning predictions — no code required
Get Started
Create an Amazon SageMaker domain to use Studio and Studio Notebooks.
Create a SageMaker domain
How it works
Amazon SageMaker Canvas - Generate accura...
Pricing (US)
AWS Free Tier
If you are new to SageMaker you are eligible for the AWS Free Tier which includes up to 10 models with up to 1M cells of data free for the first 2

Figure 7.2: *Amazon SageMaker Console*

2. Click on the `Create a SageMaker domain` button, as shown in *Figure 7.3*, if this is your first time in the SageMaker console and you don't have a domain created yet:

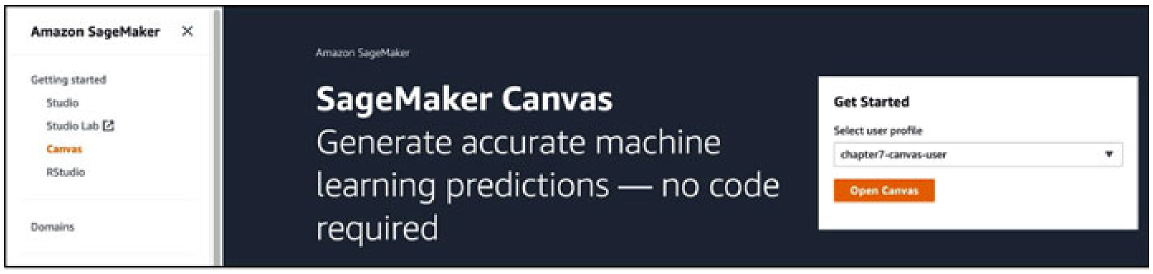


SageMaker Canvas
Generate accurate machine learning predictions — no code required

Get Started
Create an Amazon SageMaker domain to use Studio and Studio Notebooks.

Figure 7.3: *Create a SageMaker domain*

3. If this is not your first time in SageMaker, you would be prompted with a list box to select a user profile. You can pick a user profile and click on **`Open Canvas`** to get started. You can skip the next few steps in this case and start from *Step 9* in these instructions. Refer to *Figure 7.3a*:

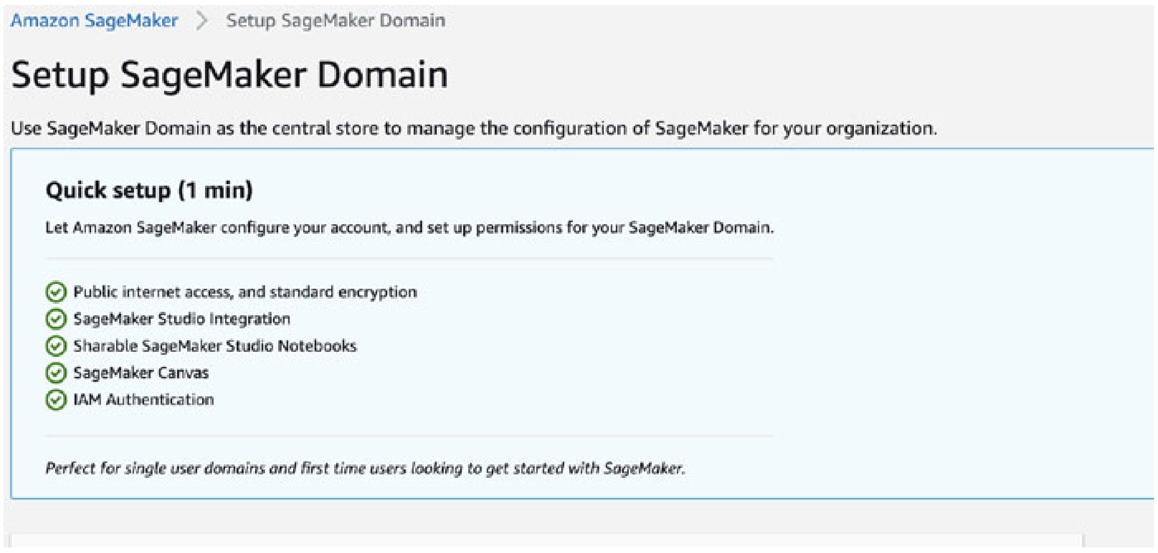


SageMaker Canvas
Generate accurate machine learning predictions — no code required

Get Started
Select user profile: chapter7-canvas-user

Figure 7.3a: *Open Canvas directly*

4. If this is your first time in SageMaker console, go ahead with the page Setup SageMaker Domain. Leave the default Quick setup option selected. Refer to *Figure 7.4*:



Setup SageMaker Domain

Use SageMaker Domain as the central store to manage the configuration of SageMaker for your organization.

Quick setup (1 min)

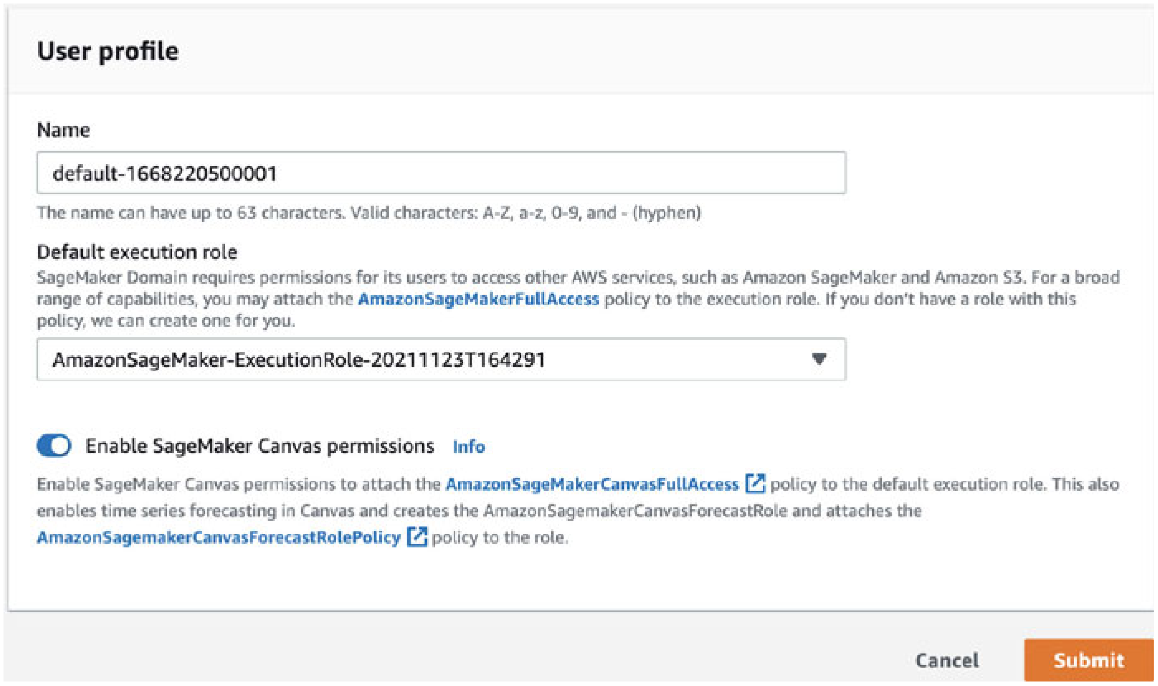
Let Amazon SageMaker configure your account, and set up permissions for your SageMaker Domain.

- ✓ Public internet access, and standard encryption
- ✓ SageMaker Studio Integration
- ✓ Sharable SageMaker Studio Notebooks
- ✓ SageMaker Canvas
- ✓ IAM Authentication

Perfect for single user domains and first time users looking to get started with SageMaker.

Figure 7.4: *Setup SageMaker Domain*

5. Provide a **`Name`** for the User profile, select **`Create a new role`** in the Default execution role, and click on **`Submit`** for SageMaker to create an IAM role for your user, as shown in *Figure 7.5*:



User profile

Name

default-1668220500001

The name can have up to 63 characters. Valid characters: A-Z, a-z, 0-9, and - (hyphen)

Default execution role

SageMaker Domain requires permissions for its users to access other AWS services, such as Amazon SageMaker and Amazon S3. For a broad range of capabilities, you may attach the AmazonSageMakerFullAccess policy to the execution role. If you don't have a role with this policy, we can create one for you.

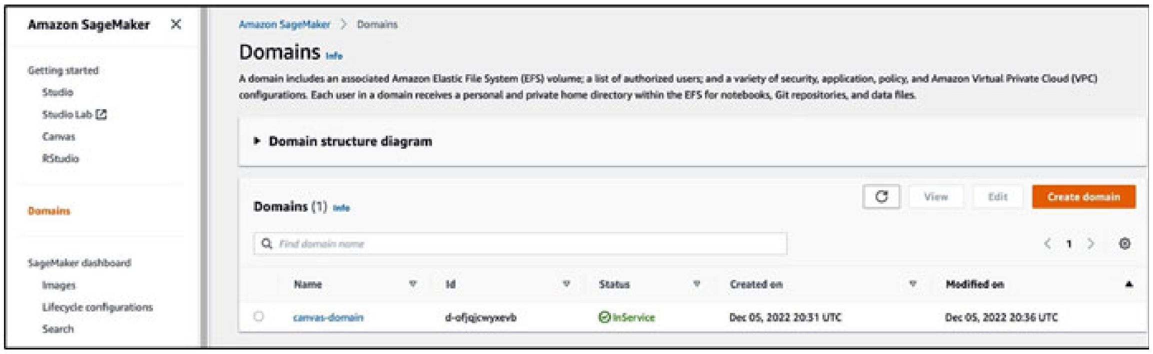
AmazonSageMaker-ExecutionRole-20211123T164291

Enable SageMaker Canvas permissions Info

Enable SageMaker Canvas permissions to attach the AmazonSageMakerCanvasFullAccess policy to the default execution role. This also enables time series forecasting in Canvas and creates the AmazonSagemakerCanvasForecastRole and attaches the AmazonSagemakerCanvasForecastRolePolicy policy to the role.

Figure 7.5: *Create SageMaker user*

6. You will be prompted to choose a VPC and a Subnet. Choose the default VPC from the list, choose a subnet from the list, and proceed to create the domain and the user.
7. Click on the **`Domains`** link in the left pane and refresh your browser to bring up the domain you just created. This will appear in the Domains page. Refer to *Figure 7.5a*:



Domains Info

A domain includes an associated Amazon Elastic File System (EFS) volume; a list of authorized users; and a variety of security, application, policy, and Amazon Virtual Private Cloud (VPC) configurations. Each user in a domain receives a personal and private home directory within the EFS for notebooks, Git repositories, and data files.

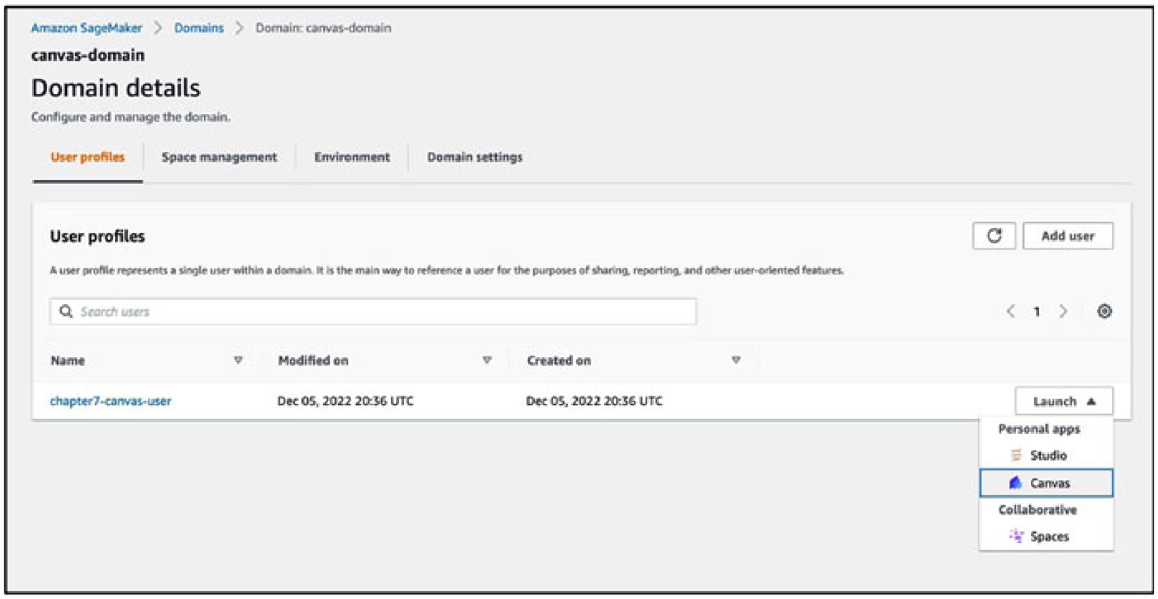
Domains (1) Info

Name	Id	Status	Created on	Modified on
canvas-domain	d-ofjqjcwyxevb	InService	Dec 05, 2022 20:31 UTC	Dec 05, 2022 20:36 UTC

Figure 7.5a: *Domains list in SageMaker*

8. Click on your domain to bring up the new user you created. Click on the **`Launch app`** list box on the right of your user and select **`Canvas`**, as shown in

Figure 7.6:



Domain details

Configure and manage the domain.

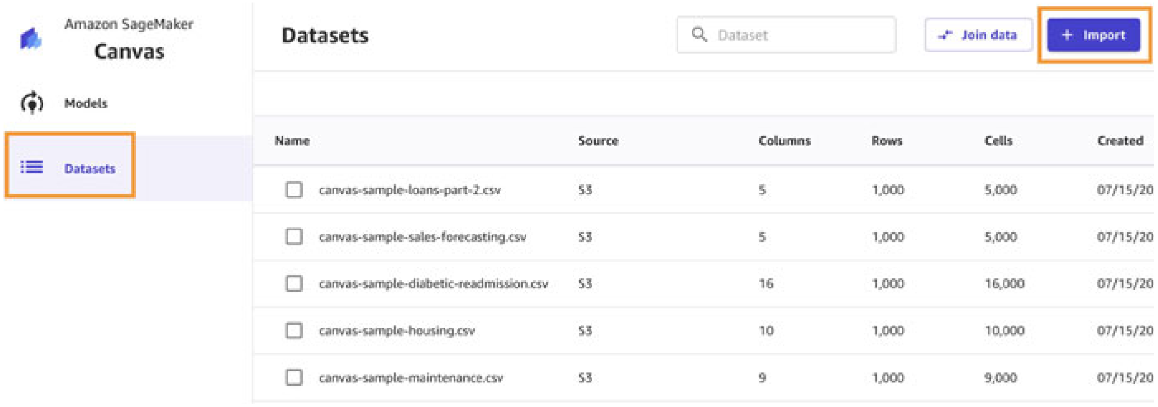
User profiles

A user profile represents a single user within a domain. It is the main way to reference a user for the purposes of sharing, reporting, and other user-oriented features.

Name	Modified on	Created on
chapter7-canvas-user	Dec 05, 2022 20:36 UTC	Dec 05, 2022 20:36 UTC

Figure 7.6: *Select Canvas*

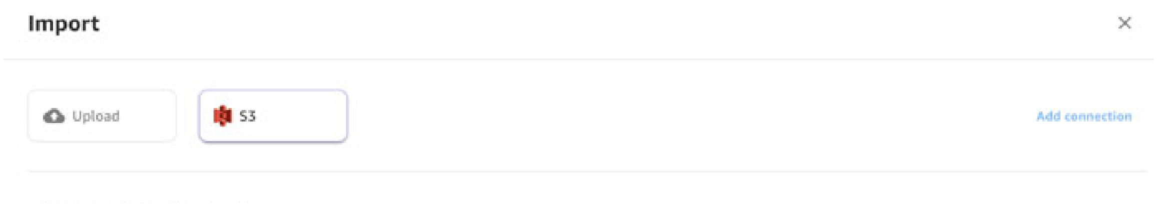
9. The SageMaker Canvas UI is launched. Click on **`Datasets`** in the left pane and then click on the **`Import`** button on the right, as shown in *Figure 7.7*:

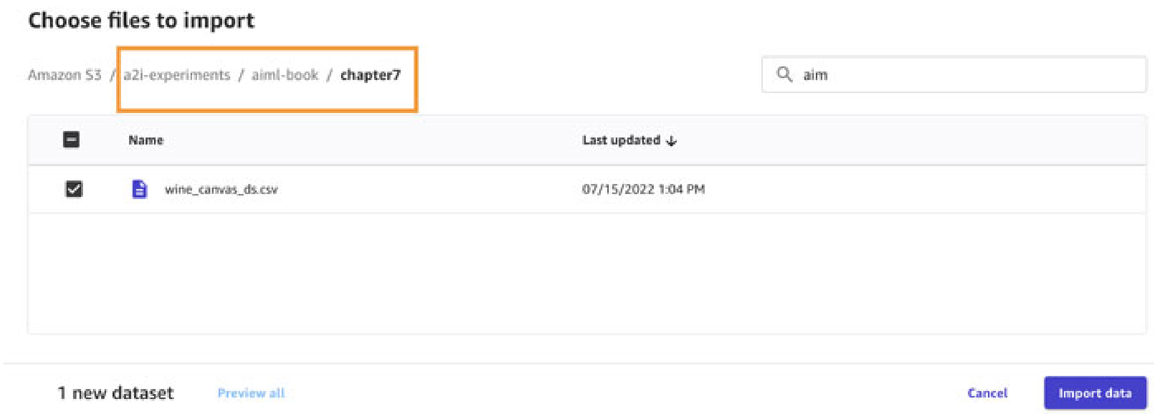


Name	Source	Columns	Rows	Cells	Created
canvas-sample-loans-part-2.csv	S3	5	1,000	5,000	07/15/20
canvas-sample-sales-forecasting.csv	S3	5	1,000	5,000	07/15/20
canvas-sample-diabetic-readmission.csv	S3	16	1,000	16,000	07/15/20
canvas-sample-housing.csv	S3	10	1,000	10,000	07/15/20
canvas-sample-maintenance.csv	S3	9	1,000	9,000	07/15/20

Figure 7.7: *Importing datasets in Canvas*

10. Type the name of the S3 bucket you created before launching SageMaker Canvas and navigate to the folder where you uploaded the raw dataset from this chapter's GitHub repository; then, click on **`Import`** data. Refer to *Figure 7.8*:




Choose files to import

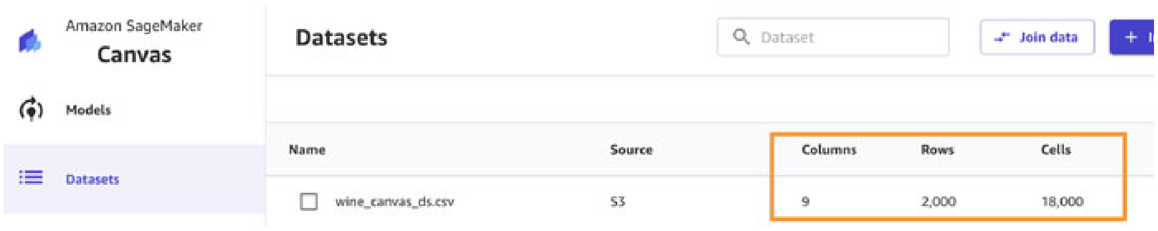
Amazon S3 / a2i-experiments / aiml-book / chapter7

Name	Last updated
wine_canvas_ds.csv	07/15/2022 1:04 PM

1 new dataset

Figure 7.8: Select dataset from Amazon S3 bucket

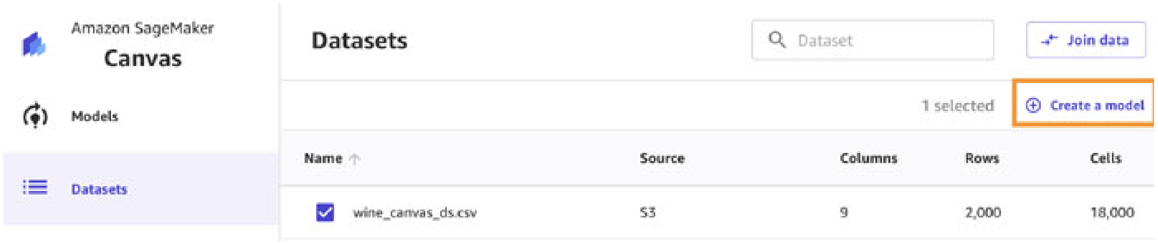
11. Once the dataset is successfully imported into Canvas, you will see the import metrics. Refer to *Figure 7.9*:



Name	Source	Columns	Rows	Cells
wine_canvas_ds.csv	S3	9	2,000	18,000

Figure 7.9: Dataset import metrics in Canvas

12. Now, select your dataset by clicking on the checkbox next to your dataset name and then on the **`Create a model`** button that appears in the top-right corner of the screen. Refer to *Figure 7.10*:



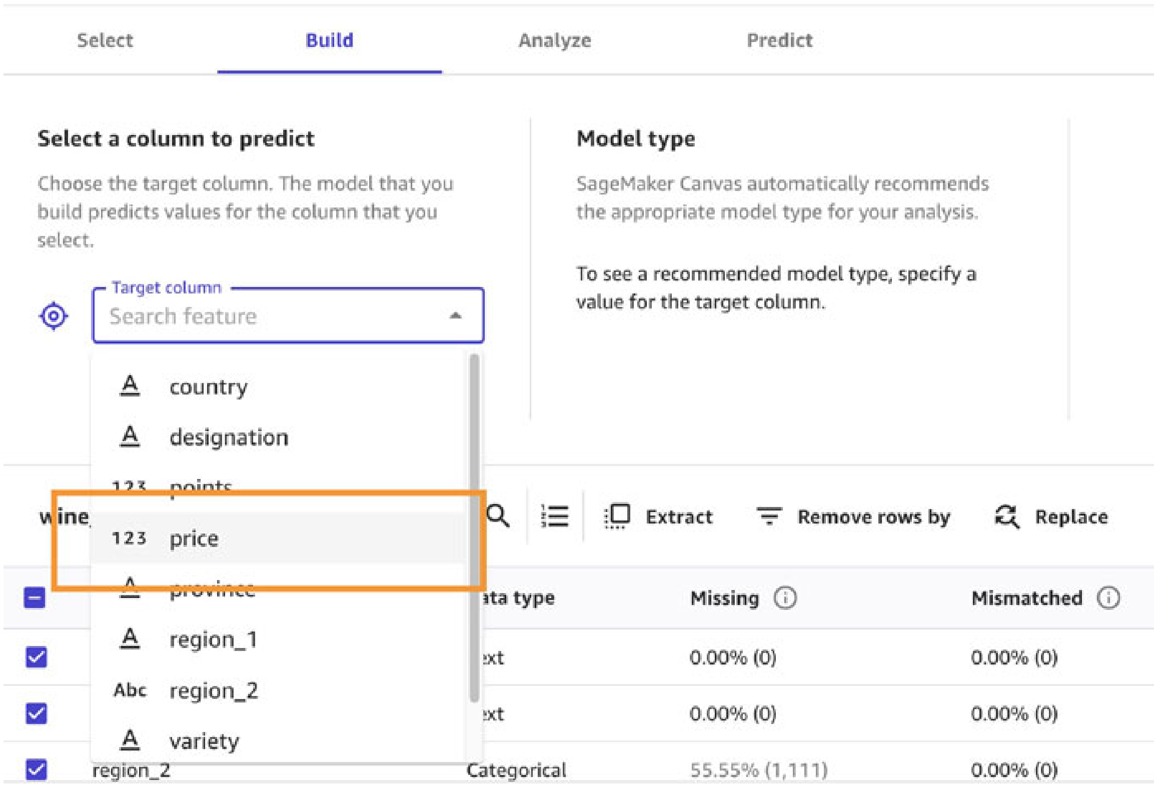
Name	Source	Columns	Rows	Cells
wine_canvas_ds.csv	S3	9	2,000	18,000

Figure 7.10: Create a ML model in Canvas

You will be prompted to enter a Model name; type a name for your model.

13. This will directly take you to the **`Build`** pane in the Model creation process in Canvas because you have already selected your dataset. Select the **`Target`**

column from your dataset that you want the model to predict. For our example, this is the price column that indicates the price of a bottle of wine. Refer to *Figure 7.11*:



Select Build Analyze Predict

Select a column to predict

Choose the target column. The model that you build predicts values for the column that you select.

Target column: Search feature

country, designation, points, price, province, region_1, region_2, variety

Model type

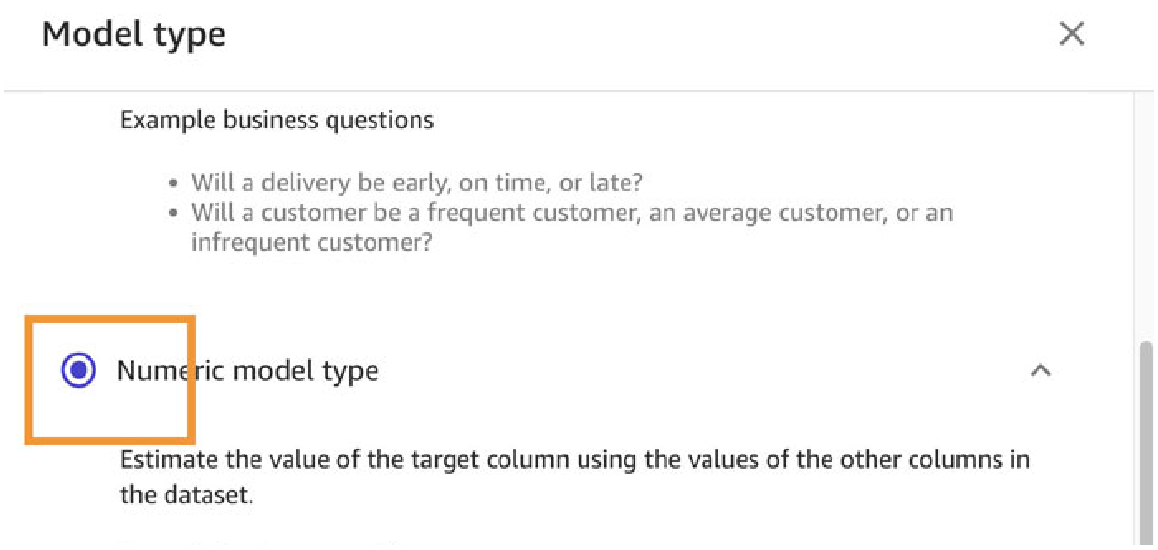
SageMaker Canvas automatically recommends the appropriate model type for your analysis.

To see a recommended model type, specify a value for the target column.

	Data type	Missing	Mismatched
	Text	0.00% (0)	0.00% (0)
	Text	0.00% (0)	0.00% (0)
region_2	Categorical	55.55% (1,111)	0.00% (0)

Figure 7.11: *Select target column or feature*

14. Then, select the **`Model`** type but choosing the **`Numeric model type`** which indicates this is a regression problem and click on **`Select model type`**. Refer to *Figure 7.12*:



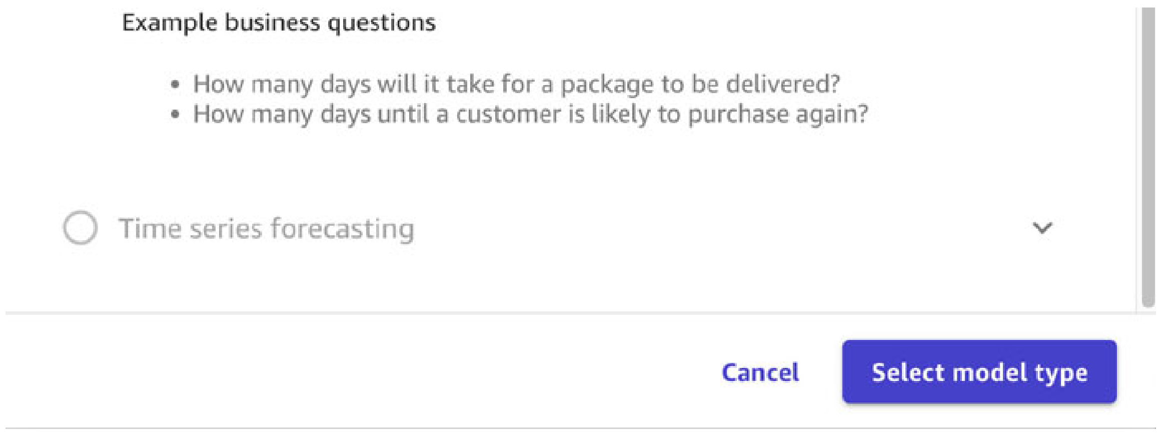
Model type

Example business questions

- Will a delivery be early, on time, or late?
- Will a customer be a frequent customer, an average customer, or an infrequent customer?

[✓ Numeric model type]

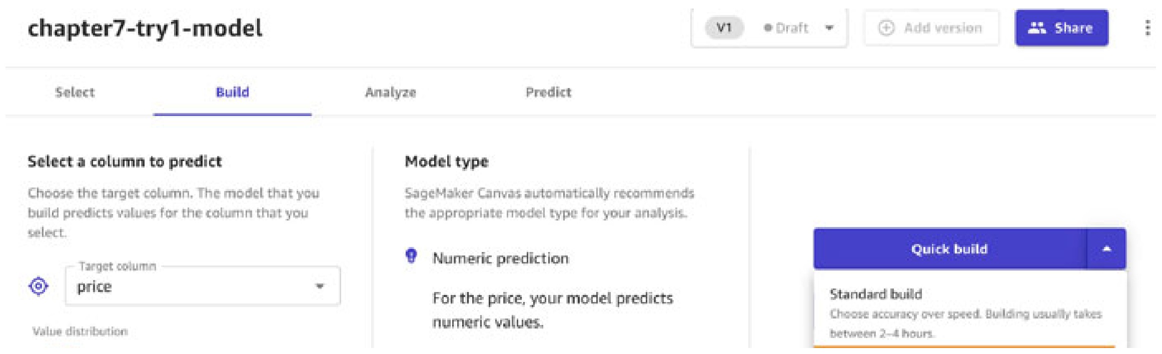
Estimate the value of the target column using the values of the other columns in the dataset.


Example business questions

- How many days will it take for a package to be delivered?
- How many days until a customer is likely to purchase again?

Figure 7.12: Select model type in Canvas

15. Then, click on the down arrow in the blue build button on the right of the Model type section, and ensure that you select Quick build. For your use case, you can also select **`Standard build`**, which will run for a couple of hours, but you will get the lowest error or the highest accuracy for our model. For this example, we will select **`Quick build`** to demonstrate the solution. Now, click on the **`Quick build`** button to start the model building process. Refer to *Figure 7.13*:



Select a column to predict

Choose the target column. The model that you build predicts values for the column that you select.

Target column: price

Value distribution

Model type

SageMaker Canvas automatically recommends the appropriate model type for your analysis.

Numeric prediction

For the price, your model predicts numeric values.

Choose accuracy over speed. Building usually takes between 2–4 hours.

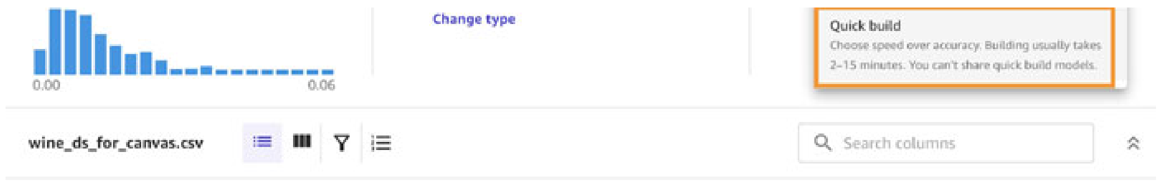

0.00
0.06
Change type
Quick build
Choose speed over accuracy. Building usually takes 2-15 minutes. You can't share quick build models.
wine_ds_for_canvas.csv
Search columns

Figure 7.13: *Quick build a model in Canvas*

16. You will be prompted to validate your data. Click on **`Start quick build`**.
17. The build process looks simple but does most of the AutoML heavy lifting behind the scenes. This will run a SageMaker Autopilot job, which performs data pre-processing, feature engineering, algorithm selection, hyperparameters selection, training several candidate models, evaluating results, and selecting a winning model that will be used for predictions. In the Canvas console, we will now see the results coming up in the **`Analyze`** pane. When the build completes, we will see a root mean squared error or RMSE (a popular evaluation metric for regression problems) displayed. For our quick build example, the RMSE is 23.75, which is pretty good considering that the whole build process took only a couple of minutes. You must select the standard build type to run for production scenarios. Refer to *Figure 7.14*:

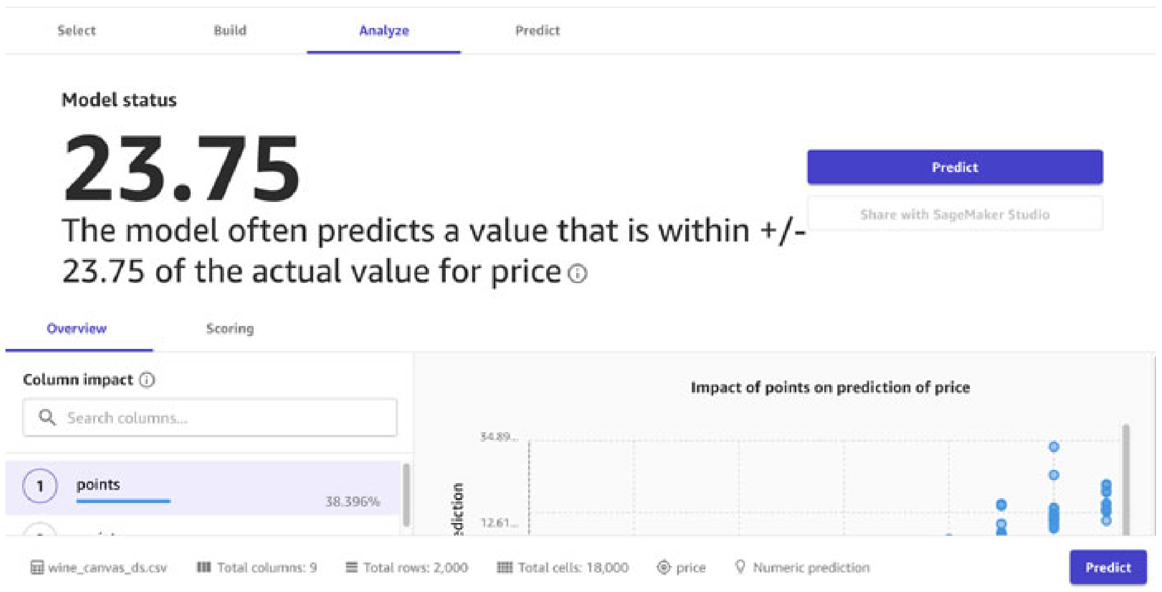
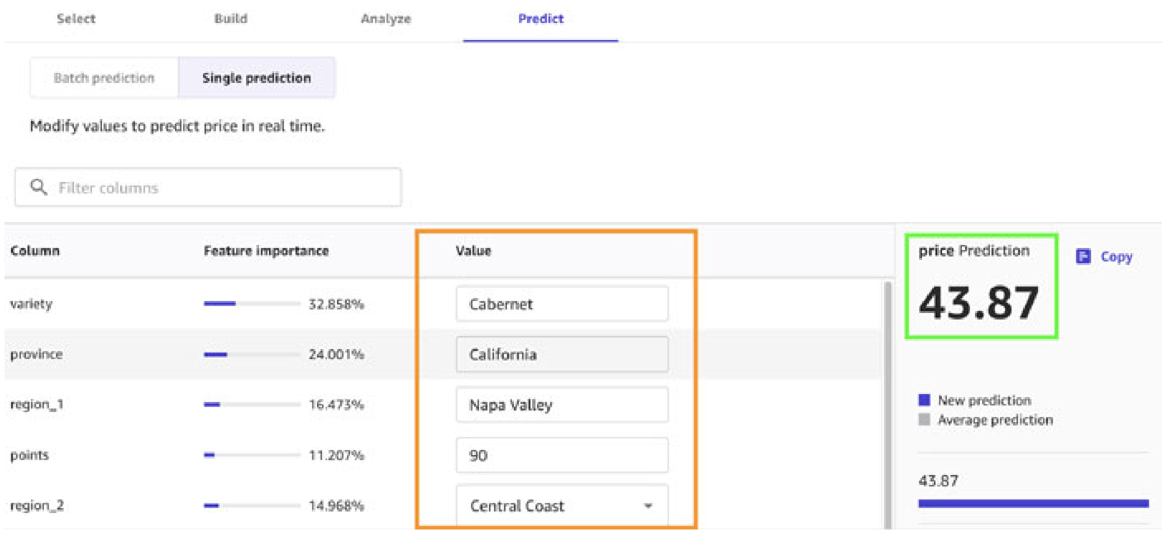


Figure 7.14: Evaluate model performance in Canvas

18. Click on the **`Predict`** button, and you can type values for the input features for the model to predict a wine price for us. This is a Single prediction. We can also do a Batch prediction, which requires us to select an imported test dataset in Canvas containing the input features for which the model predicts prices. Refer to *Figure 7.15*:



Column	Feature importance	Value
variety	32.858%	Cabernet
province	24.001%	California
region_1	16.473%	Napa Valley
points	11.207%	90
region_2	14.968%	Central Coast

Figure 7.15: Run real-time prediction in Canvas

With that, we conclude our section on how SageMaker Canvas makes it super easy to build ML models and run predictions for your use case. As we discussed earlier, Canvas is no code ML and is fully powered by AutoML. You saw that with Canvas, we directly worked with an UI to get going, and it was a fully abstracted experience usable by many, like business analysts, executives, business intelligence teams, and so on. A thing to note is that you do not have API access to Canvas outputs or models, so you cannot run predictions on Canvas models from your application, for example. If you would like to use SageMaker AutoML with the ability to programmatically access your models, you can use SageMaker Autopilot, which not only automates the end-to-end ML workflow but also provides you access to Jupyter notebooks it used for data processing and model training so that you can understand the steps that Autopilot performed. The winning model selected by Autopilot can be deployed to a SageMaker inference instance, and you can access it from your application. Here's an example notebook that walks you through Autopilot: **https://sagemaker-examples.readthedocs.io/en/latest/autopilot/sagemaker_autopilot_direct_marketing.html**.

Automated Hyperparameter Tuning

While we are on the topic of AutoML, let us not leave out an automated technique that is used to identify optimal hyperparameters for our model training: **hyperparameter optimization** (**HPO**) or automated model tuning. Though not strictly AutoML (which includes all aspects of the ML workflow, including data processing and feature engineering), HPO plays a crucial role and greatly simplifies manual model training efforts. HPO trains several candidate models for a specific problem type (classification or regression, for example) and an algorithm family (XGBoost for example) by applying a range of values for each hyperparameter to determine the most optimal values based on the best evaluation metric that can be reached by that candidate model. HPO uses a regression technique like Bayesian Search or seeds discrete values from a random range of values called Random Search. For an example of how to run an automated model tuning or hyperparameter optimization job in SageMaker, refer to **https://docs.aws.amazon.com/sagemaker/latest/dg/automatic-model-tuning-ex.html**.

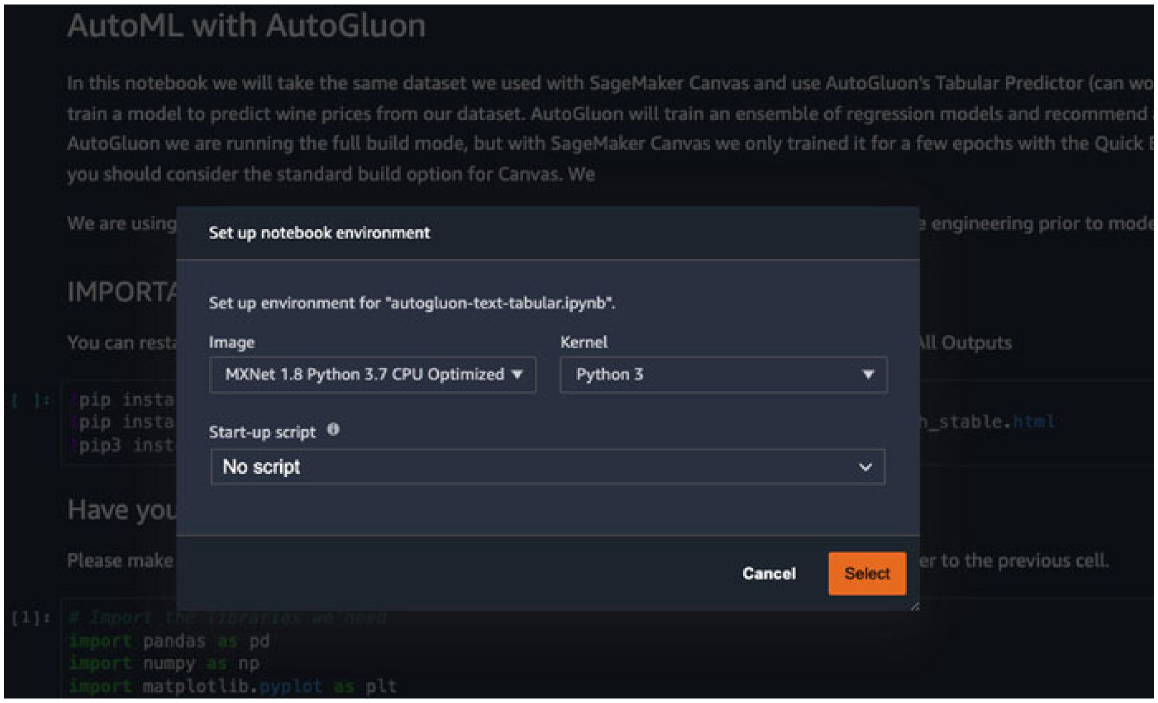
In the next section, we will learn how to train and evaluate a model using AutoGluon AutoML with the help of hands-on experiments. With AutoGluon, we will be executing code in a notebook, so while we don't have the same flexibility as with Canvas, we will have room to work with certain aspects of data preparation and training code. We will also be able to deploy these models to SageMaker inference endpoints, similar to Autopilot, and use the models in our applications.

Using AutoGluon for AutoML

We will now walk you through AutoGluon, which is a free abstracted AutoML technique published by AWS. When compared to Autopilot (tabular and timeseries), AutoGluon provides multiple modalities (image, text, tabular and time series) and access to the HuggingFace (**https://github.com/huggingface/transformers**) and openai (**https://github.com/openai/CLIP**) model zoos. Hence, it can be used for various ML problems. Further, you can easily create AutoGluon training and inference containers for SageMaker and build powerful ML solutions with SageMaker as your enterprise platform. Recently, AutoGluon Tabular was added as a built-in algorithm in SageMaker (**https://docs.aws.amazon.com/sagemaker/latest/dg/autogluon-tabular.html**), making model training even easier. Now, let us get started with trying out AutoGluon for text and tabular data types with our wine dataset. To keep it simple, we will use the same dataset we used with SageMaker Canvas.

To execute this example, we will use SageMaker Studio notebooks, and so, if not already done, you will have to onboard to SageMaker Studio and git clone our book repository. Follow the instructions in the Setting up your AWS account Section in Chapter 2, Hydrating Your Data Lake, to sign up for an AWS account. Once you have signed up, log in to your AWS account using the instructions here: **https://docs.aws.amazon.com/IAM/latest/UserGuide/console.html**. You need to create an Amazon S3 bucket (**https://docs.aws.amazon.com/AmazonS3/latest/userguide/create-bucket-overview.html**) if you have not already done so in the previous chapters; then, note down the name of the bucket. Next, follow the instructions in the *Technical Requirements* section in Chapter 3, Predicting the Future With Features, to onboard to an Amazon SageMaker Studio domain, clone (if you did not already do it in the earlier chaptesr) the book's GitHub repository, provided at the beginning of this book, and click on `Chapter-07` to open it. Open the Jupyter notebook autogluon-text-tabular.ipynb by clicking on it. Execute the following instructions to continue with your build activity:

1. Before we execute the notebook cells, double-check your notebook kernel shows Python 3 (MXNet 1.8 Python 3.7 CPU Optimized) in the top-right corner of the notebook tab. If it does not show this, from the menu at the top of the page, select `Kernel` `Change Kernel`, change the `Image` to `Python 3` (MXNet 1.8 Python 3.7 CPU Optimized), and click on Select in the pop-up, as shown in *Figure 7.16*:



AutoML with AutoGluon

In this notebook we will take the same dataset we used with SageMaker Canvas and use AutoGluon's Tabular Predictor (can wo
train a model to predict wine prices from our dataset. AutoGluon will train an ensemble of regression models and recommend
AutoGluon we are running the full build mode, but with SageMaker Canvas we only trained it for a few epochs with the Quick
you should consider the standard build option for Canvas. We

We are using   engineering prior to mode

IMPORTA

You can resta   ll Outputs

```
[ ]: !pip insta                                   n_stable.html
     !pip insta
     !pip3 inst
```

Have you

Please make   er to the previous cell.

```
[1]: # Import the libraries we need
     import pandas as pd
     import numpy as np
     import matplotlib.pyplot as plt
```

Set up notebook environment

Set up environment for "autogluon-text-tabular.ipynb".

Image: MXNet 1.8 Python 3.7 CPU Optimized

Kernel: Python 3

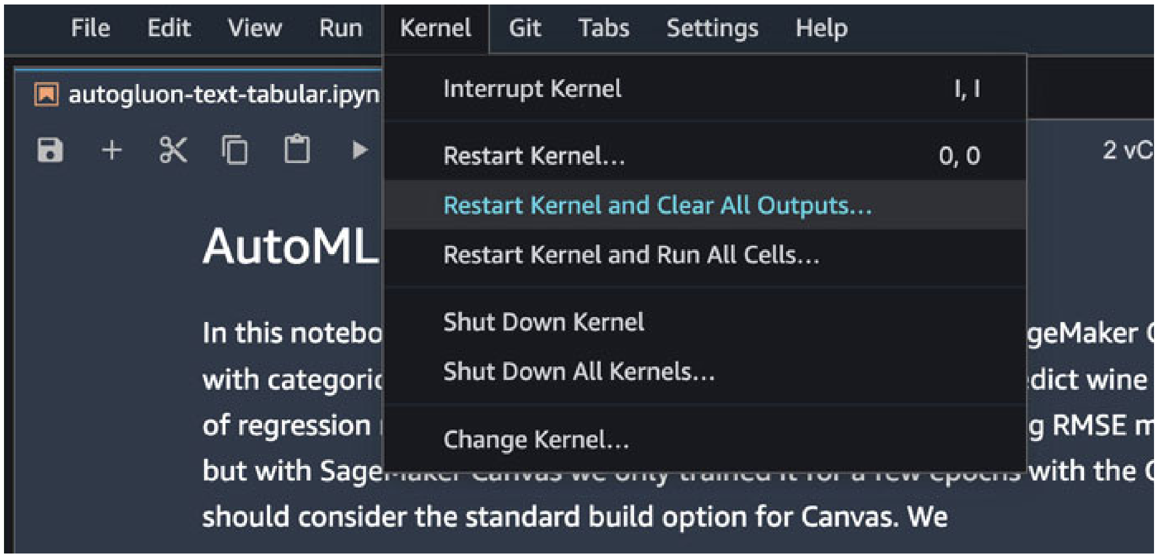
Start-up script: No script

Figure 7.16: *Change Studio Kernel*

2. First, we will have to install the **`autogluon`** package. Keep your cursor in the first cell of the notebook and click on the triangular play button at the top of the tab to execute the cell. This will install the dependent libraries and then the AutoGluon package. The commands will take 5 to 10 minutes to complete:

```
!pip install -U setuptools wheel
!pip install "torch>=1.0,<1.12+cpu" -f
https://download.pytorch.org/whl/cpu/torch_stable.html
!pip3 install autogluon
```

3. Restart the notebook kernel before proceeding. From the top menu, select **`Kernel`** □ **`Restart Kernel`** and **`Clear All Outputs`**. Refer to *Figure 7.17*:



AutoML

In this notebo ... geMaker with categoric ... dict wine of regression ... g RMSE m but with Sage... with the should consider the standard build option for Canvas. We

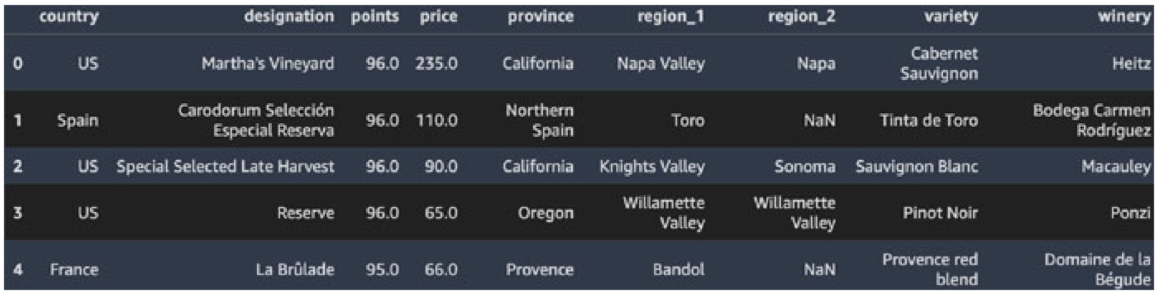
Figure 7.17: Restart Kernel

4. Then, execute the second cell in the notebook to import all the Python libraries we need to run our AutoGluon AutoML training.

5. For this training, we will use the same sample raw dataset we used with SageMaker Canvas. This file is available in our GitHub repository for this chapter (https://github.com/garchangel/AIMLwithAWS/tree/main/Chapter-07/wine_canvas_ds.csv) and was copied when you cloned the repository. Execute the next cell to load this dataset into a Pandas DataFrame.

```
# Let's first load the data into a Pandas dataframe so it is
easy for us to work with it
wine_canvas_raw_df = pd.read_csv('./wine_canvas_ds.csv',
sep=',',header=0)
wine_canvas_raw_df.head()
```

You should see the following output displayed, as shown in *Figure 7.18*:



	country	designation	points	price	province	region_1	region_2	variety	winery
0	US	Martha's Vineyard	96.0	235.0	California	Napa Valley	Napa	Cabernet Sauvignon	Heitz
1	Spain	Carodorum Selección Especial Reserva	96.0	110.0	Northern Spain	Toro	NaN	Tinta de Toro	Bodega Carmen Rodríguez
2	US	Special Selected Late Harvest	96.0	90.0	California	Knights Valley	Sonoma	Sauvignon Blanc	Macauley
3	US	Reserve	96.0	65.0	Oregon	Willamette Valley	Willamette Valley	Pinot Noir	Ponzi
4	France	La Brûlade	95.0	66.0	Provence	Bandol	NaN	Provence red blend	Domaine de la Bégude

Figure 7.18: Wine raw dataset

6. Execute the next cell to split this dataset into training dataset (1989 rows) and test dataset (10 rows).

```
# Let us reserve 10 rows to test our model and the rest will be
our training dataset
wine_train_df = wine_canvas_raw_df.iloc[:1989]
wine_test_df = wine_canvas_raw_df.iloc[1990:]
```

7. You can review the contents of the test dataset by executing the next cell in the notebook. The index should begin with row 1990.

8. In the next cell, we will submit the AutoML training job. From our raw dataset, we want to train a ML model that can predict the price of a wine bottle based on correlated input features. However, nore that our tabular dataset contains a mix of text and numbers. So, we have categorical and quantitative features. AutoGluon tabular (**https://auto.gluon.ai/stable/tutorials/tabular_prediction/index.html**) is designed to work with such a dataset, and we will use the TabularPredictor for our training needs.

```
# Data processing, feature engineering, setting up and running
training is just 3 lines of code
from autogluon.tabular import TabularPredictor
predictor = TabularPredictor(label='price',
path='winning_wine_predictor')
predictor.fit(wine_train_df)
```

9. Running ML training with AutoGluon is as simple as just the three lines of the preceding code. All you need to do is provide a training dataset, indicate which column in the dataset is the label and optionally, provide a folder name to store your winning model artefacts. AutoGluon performs data processing, feature engineering, algorithm selection, hyperparameter optimization, and model training and tuning by running an ensemble of regression algorithms and neural networks. It then selects a winning candidate based on the lowest error metric, or root mean squared error for our example. Let us now inspect the results of our training job. In the first part of the output, we see the data metrics and problem type selection, as highlighted here:

```
Beginning AutoGluon training …
AutoGluon will save models to "winning_wine_predictor/"
AutoGluon Version:  0.5.0
Python Version:     3.7.10
Operating System:   Linux
Train Data Rows:    1989
Train Data Columns: 8
Label Column: price
Preprocessing data …
Warning: Ignoring 100 (out of 1989) training examples for which
the label value in column 'price' is missing
```

AutoGluon infers your prediction problem is: 'regression' **(because dtype of label-column == float and many unique label-values observed).**

10. In the next part of the output (only a snapshot displayed here; refer to the notebook output under the executed cell for full results), we see the results of the feature engineering that AutoGluon performed:

```
Inferring data type of each feature based on column values. Set
feature_metadata_in to manually specify special dtypes of the
features.
Stage 1 Generators:
 Fitting AsTypeFeatureGenerator…
Stage 2 Generators:
 Fitting FillNaFeatureGenerator…
Stage 3 Generators:
 Fitting IdentityFeatureGenerator…
 Fitting CategoryFeatureGenerator…
  Fitting CategoryMemoryMinimizeFeatureGenerator…
Stage 4 Generators:
 Fitting DropUniqueFeatureGenerator…
Data preprocessing and feature engineering runtime = 0.36s …
```

11. AutoGluon then prints the performance metric it has selected for this job (the most popular metric for regression is RMSE, or root mean squared error):

```
AutoGluon will gauge predictive performance using evaluation
metric: 'root_mean_squared_error'
 This metric's sign has been flipped to adhere to being
 higher_is_better. The metric score can be multiplied by -1 to
 get the metric value.
 To change this, specify the eval_metric parameter of
 Predictor()
```

12. Next, AutoGluon selects a list of algorithms and runs ensemble training with optimal pre-selected hyperparameters. We have only displayed a snapshot here; refer to notebook outputs for the full list of models.

```
Fitting model: LightGBM …
```

```
 -22.5568   = Validation score   (-root_mean_squared_error)
 0.37s   = Training   runtime
 0.01s   = Validation runtime
Fitting model: CatBoost …
 -20.7315   = Validation score   (-root_mean_squared_error)
 2.73s   = Training   runtime
 0.01s   = Validation runtime
Fitting model: WeightedEnsemble_L2 …
 -19.8987   = Validation score   (-root_mean_squared_error)
 0.45s   = Training   runtime
 0.0s   = Validation runtime
```

13. Finally, AutoGluon prints the winning model as **`WeightedEnsemble_L2`** because it has the lowest RMSE of 19.89:

```
AutoGluon training complete, total runtime = 23.96s … Best
model: "WeightedEnsemble_L2"
TabularPredictor saved. To load, use: predictor =
TabularPredictor.load("winning_wine_predictor/")
```

14. In comparison to AutoGluon's RMSE of 19.89, the Canvas model with quick build had a RMSE of 23.75. If we had run Canvas in standard build, we would have a much lower RMSE and a better model. Now, we will run predictions with our newly trained model using the test dataset we had kept aside. Execute the next cell to create test data by dropping the price column from our original test dataset:

```
# Let us drop the price column from our test dataset so we can
get the model to predict that
wine_test_priceless = wine_test_df.drop(['price'], axis=1)
wine_test_priceless.head()
```

15. Run predictions using our model and display the results by executing the next cell.

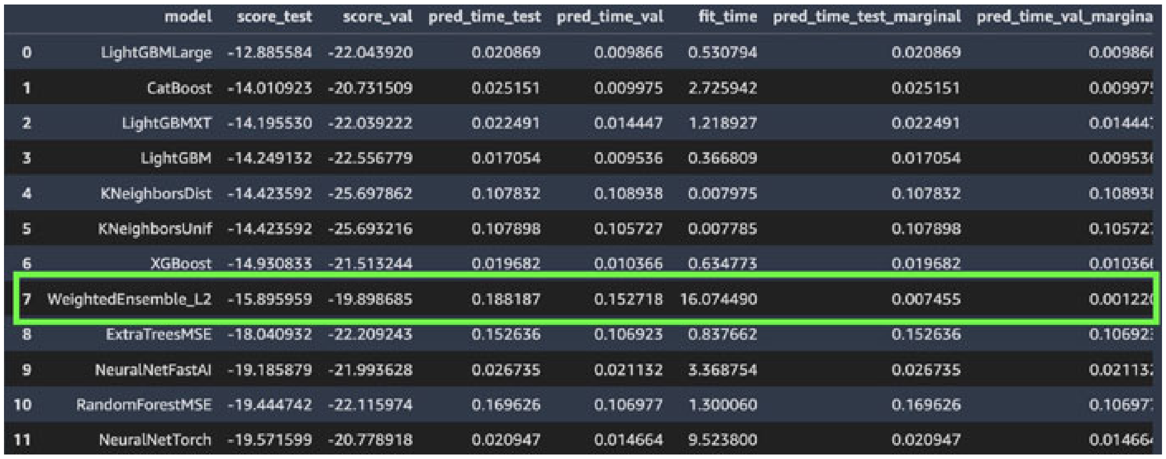
```
winning_predictions = predictor.predict(wine_test_priceless)
print("Winning Model Predictions for test
data:  "+str(winning_predictions))
```

We get the following output, which shows the wine price for each row index from our test dataset:

```
Winning Model Predictions for test data:  1990    45.047230
1991    28.318157
1992    29.502701
1993    76.576393
```

```
1994    35.715672
1995    36.933662
1996    28.318157
1997    23.321636
1998    34.547947
1999    47.772491
Name: price, dtype: float32
```

16. Finally, execute the next cell to print a Leaderboard of models that AutoGluon ran against our test data ranked based on model testing RMSE scores (not model validation RMSE, which determines our winning model). Refer to *Figure 7.19*:



	model	score_test	score_val	pred_time_test	pred_time_val	fit_time	pred_time_test_marginal	pred_time_val_margina
0	LightGBMLarge	-12.885584	-22.043920	0.020869	0.009866	0.530794	0.020869	0.00986
1	CatBoost	-14.010923	-20.731509	0.025151	0.009975	2.725942	0.025151	0.00997
2	LightGBMXT	-14.195530	-22.039222	0.022491	0.014447	1.218927	0.022491	0.01444
3	LightGBM	-14.249132	-22.556779	0.017054	0.009536	0.366809	0.017054	0.00953
4	KNeighborsDist	-14.423592	-25.697862	0.107832	0.108938	0.007975	0.107832	0.10893
5	KNeighborsUnif	-14.423592	-25.693216	0.107898	0.105727	0.007785	0.107898	0.10572
6	XGBoost	-14.930833	-21.513244	0.019682	0.010366	0.634773	0.019682	0.01036
7	WeightedEnsemble_L2	-15.895959	-19.898685	0.188187	0.152718	16.074490	0.007455	0.00122
8	ExtraTreesMSE	-18.040932	-22.209243	0.152636	0.106923	0.837662	0.152636	0.10692
9	NeuralNetFastAI	-19.185879	-21.993628	0.026735	0.021132	3.368754	0.026735	0.02113
10	RandomForestMSE	-19.444742	-22.115974	0.169626	0.106977	1.300060	0.169626	0.10697
11	NeuralNetTorch	-19.571599	-20.778918	0.020947	0.014664	9.523800	0.020947	0.01466

Figure 7.19: *AutoGluon Model Leaderboard*

That concludes our example on how to use AutoGluon to run AutoML training of your ML workflows. AutoGluon is really versatile and supports various ML domains. Ensure that you check out their tutorials page for more details (**https://auto.gluon.ai/stable/tutorials/index.html**). As you can imagine, AutoML is indeed a popular topic because of its ease of use, flexibility, and time and cost efficiencies you can achieve, and because it requires little to no ML expertise to run. We have only scratched the surface when it comes to AutoML in this chapter, with SageMaker Canvas and AutoGluon. There are other services within the AWS AI/ML stack that provide AutoML capabilities, such as Amazon Forecast AutoML and AWS AI Services.

Amazon Forecast (**https://aws.amazon.com/forecast/**) is a fully managed service for time series forecasting problems, with the option to use AutoML for data processing, feature engineering, and model training by selecting an algorithm from a collection of algorithms or recipes that are experimented upon. Amazon Forecast is an AWS AI service, which is a collection of powerful pre-trained models providing ML capabilities through APIs. AWS AI services cover a wide range of use cases, domains, and industries, including services for speech, text, vision, industrial AI, and business processes. We will cover AI services in detail in Chapter 10, Adding Intelligence With Sensory Cognition, and Chapter 11, AI for Industrial Automation.

Conclusion

In this chapter, we took a fork in the sequence of ML workflow stages we learned in the previous chapters to learn about AutoML. We saw that with AutoML and more recently, with the no-code ML options, most of the ML workflow stages are automated; all we need to take care of is data collection and some basic pre-processing to arrange the data based on the ML problem type. All the other tasks are fully automated with just some basic human intervention required. So AutoML is available as a route for you right after data collection, which we performed in *Chapter 2, Hydrating the Data Lake*. If you choose AutoML, you can automate the all the tasks until *Chapter 9, Wisdom at Scale With Elastic Inference*. In the next chapter, we will learn various deployment strategies for implementing our trained models for testing and production. We will learn how to approach and design model deployment techniques, how to build and update endpoints without dropping inference requests, and more.

Points to Remember

Here are the key takeaways from this chapter:

- In this chapter, we learned how to run fully automated ML workflows as an alternative to what we discussed in *Chapter 3, Predicting the Future With Features*; *Chapter 5, Casting a Wider Net*; and *Chapter 6, Iteration makes intelligence*.
- We learned that depending on the skill levels of the team, the requirements of the project, and the time and funding available, AutoML is a popular choice today for its ease of use and efficiency.
- We saw how to use two of the most popular AutoML choices today with SageMaker Canvas using SageMaker Autopilot implicitly and using AutoGluon with a SageMaker Studio notebook.
- We took our wine characteristics dataset that we first introduced in *Chapter 3, Predicting the Future With Features*, but used it in its unprocessed form, that is, the raw dataset, as input for our AutoML solutions.

- We saw that with SageMaker Canvas, we did not write a single line of code and were able to process datasets, execute feature engineering, and tune and train a ML model in under 2 minutes with the Quick build option, to predict wine prices with a RMSE of 23.75.
- We then used the same dataset with AutoGluon and trained a Tabular predictor, which transformed our dataset; executed feature engineering; trained, tuned, and selected a winning model (with the lowest RMSE) from an ensemble of 11 different models (of different family of algorithms).

Multiple Choice Questions

Use the following questions to challenge your knowledge of AutoML with AWS AI/ML services.

1. **What is Amazon SageMaker Canvas?**
    a. It is a new training algorithm in SageMaker.
    b. It is a no-code ML service that helps non-technical users build, train and use ML models.
    c. It is a new GUI from SageMaker that helps with drawing ML architecture diagrams.
    d. It is the new NLP service from SageMaker.
2. **What ML modalities does AutoGluon support?**
    a. Text and tabular only
    b. Image and tabular only
    c. Time series and text only
    d. Image, text, tabular and time series
3. **What are some of the ML modelling tasks (normally performed manually) that SageMaker Autopilot automates?**
    a. Feature engineering
    b. Algorithm selection
    c. Model training and tuning
    d. All of the above
4. **SageMaker Canvas uses SageMaker Autopilot under the hood.**
    a. True

    b. False

5. **AutoGluon requires feature engineering of text features in a tabular dataset before it can be used.**

    a. True

    b. False

Answers

1. **b**
2. **d**
3. **d**
4. **a**
5. **b**

CHAPTER 8

Blue or Green (Model Deployment Strategies)

Introduction

Blue or Green. Seems like a simple question, right? What about accuracy versus error rate? Horizontal versus vertical scaling? More memory or more compute? GPU or CPU? The possibilities before us do not end with training a machine learning model. The details of how you deploy your trained model can impact its usefulness and ability to accomplish the goal for which it was created. Making sure your model has the correct resources in the correct amounts and a plan to handle the expected traffic is necessary to complete your projects. In this chapter, we will discuss the considerations needed to deploy, scale, and monitor your models. This includes the rapidly converging DevOps (and DevSecOps!) concepts and the necessary steps to optimize your costs.

To try the examples in this section, refer to the *Technical Requirements* section in *Chapter 1 – Introducing the ML Workflow* to sign in to the AWS management console, execute the steps in onboard to SageMaker studio, and execute cloning the repository to SageMaker Studio to get started. Click on the folder that corresponds to this chapter number. If you see multiple notebooks, the section title corresponds to the notebook name for easy identification. You can also passively follow the code samples using the GitHub repository provided at the beginning of the book.

Chapter Scenario

In this chapter, we will be taking on the role of a site reliability engineer working alongside a group of highly motivated data scientists. These scientists have been working relentlessly for the past quarter to refine their methods of training various machine learning models, testing their accuracy, and getting them ready to use for various inference tasks. Now, planning for the next quarter's tasks, they are ready to take their models to production. Our SRE is well-versed in deploying other types of development-related artifacts but has never worked with machine learning models before. As an Aires, our SRE is not afraid of new tasks and assignments, and the pressure that comes with them.

Structure

In this chapter, we will discuss the following topics:

- Structure, considerations, and methods for deploying Machine Learning models
- Options for model deployment available with Amazon SageMaker Endpoints
- Ways to evaluate the different compute options available for Inference
- Implementations for automatically scaling resources available for your inference endpoint
- Steps needed to deploy a machine learning model for inference
- Considerations that need to be made in monitoring a deployed model
- Effective strategies for updating your deployed machine learning model

Objectives

We identified the three fundamental portions of a machine learning project in *Chapter 1, Introducing the ML Workflow*. We covered training and registering your model in previous chapters. The last section, deploying your model, is what we will review in this chapter. From choosing the correct compute options for your model to deciding on the type of inference you need to accomplish, this chapter will guide you through the choices, what they mean, and how to implement them.

Inference Options

Inference is defined as a conclusion reached based on evidence and reasoning. In the Machine Learning context, inference is the process of providing data to a machine learning model to calculate an output. This output can be a numerical score, a Boolean value, a classification, a prediction, or even a set of data points that are intended to be used as input for another application, possibly even another Machine Learning model. To perform this inference, we need to deploy a model and make it available to accept that data and, in turn, respond with its predictions. This act of deployment is what you would expect from application development projects, but it has some unique considerations that we will cover in this chapter. Our site reliability engineer will draw on their experience of deploying these kinds of projects to accomplish the deployment of a Machine Learning model.

At its most fundamental, a deployed Machine Learning model will first need a method to accept incoming structured data. The method chosen will largely be determined by the type of interaction your model will have with the system sending data to it. For evaluating your model, deploying a simple, single-threaded, locally accessible REST endpoint, especially to on-demand compute resources, is optimal. This allows you to send your validation data to your endpoint and evaluate it against your chosen metric, and then destroy the REST endpoint. Beyond this, the main consideration should be the necessary response time of your application. For applications that need to be able to respond to requests no matter when they are sent, real-time endpoints are preferable. Alternatively, asynchronous inference, where a data set is only periodically sent to your model, typically in a scheduled fashion, can take advantage of batch inference. Before moving on, let’s look at each of these in detail.

NOTE: What is a REST API? A REpresentational State Transfer (REST) describes an interface between two decoupled systems. In a monolithic application, the application transfers logic within itself, and the structure of the monolith itself manages the structure and

expectations of that logic. In decoupled applications, like Machine Learning services, applications externalize the method of interaction and structure into a service. REST interfaces work much like websites and use similar methods for interaction. REST interfaces work well for Machine Learning models because we can send predefined traffic to our REST endpoint, and the endpoint, in turn, can handle the interaction and the response from our model.

Testing and Validation

Typically, the first time a model is deployed is immediately after training has been completed. To validate that your trained model works and that it meets your desired needs, you need to be able to send traffic to it and measure the results. You can (and should) use your test data set for this validation. Testing your trained model is typically quick, with your model deployed, test traffic sent to the endpoint, and then the endpoint destroyed.

Real Time

Real time inference is defined by a model endpoint that is deployed and available whenever requests are sent to it. It is often measured not only by the evaluation metric of the model itself but also by the response time of the endpoint. Additionally, as with all model deployment methods, the scale of your inference size should be taken into consideration. A singular endpoint will begin to suffer latency as the number of concurrent requests increase, eventually becoming unable to respond to requests at all.

Batch

Also known as asynchronous, batch inference is used when you need to perform inference on a data set periodically. Since you pay for the compute associated with your model hosting while it is active, there is no need to leave a model deployed unless you are actively using it. By only deploying your model when you need it, you make the deployment object part of your process. Additionally, since we are deploying our model when we have data ready to use for inference, we can make decisions about the type of instance. If our workload allows for longer processing times, using smaller or fewer instances can allow for cost optimization. Similarly, if your chosen workload needs to be processed as quickly as possible, consider larger or more instances to optimize for speed.

Streaming

Streaming inference, though it appears like real-time, presents some unique challenges. The data involved is typically stored in a messaging or streaming bus, and this service holds the individual messages until a consumer application picks them up and acts on them. The streaming bus does not typically change the source messages, so like other inference options, a method of getting the message in the right format to send to the model endpoint is needed, along with a method to deliver them. In AWS, the first can be accomplished in-stream with Kinesis Data Analytics, transforming the message to be ready for inference. Similarly, you can use AWS Lambda as your stream consumer, sending the transformed messages to your Inference Endpoint and then handling the responses. For streaming inference, handling the volume of transactions is the key consideration for your hosting options.

Choosing your Compute

Now that we understand the different use cases for hosted models, we need to consider what compute we require. The resources associated with the compute option are also used to power your model, so the more power you provide, the more aggressive processing your model will be able to perform. This doesn’t necessarily scale linearly, though, with compute sources often topping out at a certain level no matter the raw processing or memory provided. To address this as well as cost optimization needs, multiple compute instances can be leveraged behind a load balancing method. In this style, a request will be routed to a single resource in a pool, and subsequent requests will be routed to different pool members.

At its most basic, the act of deploying a machine learning model is the creation of a web service to accept traffic, sending it to the model itself, and then returning the responses. To host this service, we need a compute resource, which is distinct from the similar resource we used to train our model. Many models operate at different efficiencies during inference than they do during training, so a one-size-fits-all approach does not apply. In addition, depending on the model, training and inference may need different resources. For example, XGBoost trains much faster on GPU resources and performs inference best on CPU instances from a memory utilization standpoint considering the decision tree ensembles are held in-memory at inference time.

Amazon SageMaker hosting allows us to choose the instance type when we deploy our model, and we can do the same thing if we host the model ourselves on an EC2 instance. After an initial estimation of necessary system resources based on the provided documentation associated with your chosen model, the instance type can be considered another type of hyperparameter. Like other hyperparameters during training and tuning of your model, finding the right combination of resources and instance type is an iterative experiment. There are tools that can help with this process, including AutoML, AutoGluon, and the AWS Inference Recommender. Let's look at these ahead.

Self-Hosted

In the case of practitioners who wish to manage the whole of their chosen compute environments, self-hosting is a strong option. In situations where multiple applications are sharing the same compute resource or where there are regulatory requirements preventing more managed options, managing your own compute is preferable. For these situations, allocating one or more EC2 instances to host your models is the best choice.

Using this method, you can create an independent auto scaling group for each of your model needs and the EC2 native user data capability to bootstrap each instance at start-up to install web server resources, download your models, and update local application packages. Following is an example of this kind of user data script for an Ubuntu style Amazon Machine Image:

```
#!/bin/bash
yum update -y
yum install -y amazon-linux-extras
amazon-linux-extras enable python3.8
yum install python3.8
python -m ensurepip --upgrade
pip install flask
```

```
aws s3 cp s3://replacewithyourbucket/model/model.tar.gz .
aws s3 cp s3://replacewithyourbucket/flask/flaskapp.py .
```

Amazon SageMaker Endpoint

Using an Amazon SageMaker endpoint gives additional flexibility over hosting the model yourself; this means more of the undifferentiated heavy lifting is handled by AWS. It also makes the process faster, allowing more rapid experimentation and handling individual endpoints as deployable objects.

Creating an endpoint in Amazon SageMaker differs slightly if you are using the SageMaker SDK or the AWS SDK for Python (boto3). The main difference is that using the SageMaker SDK requires the creation of an Endpoint configuration, while using the AWS SDK for Python does not. We will review the same implementation for both methods here.

The first step to creating an endpoint in SageMaker is to identify the location where your model is stored in S3 and the location for either the custom-made Docker image that contains your inference code or the framework and version of a SageMaker built-in container.

NOTE: While S3 buckets are global resources, they are created and hosted within specific regions. The bucket containing your model artifact and SageMaker endpoint must be in the same region.

Next, you will create the model object itself, referencing the container image location, the S3 model location, and the IAM role used for SageMaker to create the model:

```
from sagemaker.model import Model
model = Model(image_uri=container_url,
    model_data=s3_model_location,
    role=iam_role)
```

Now we create an endpoint configuration. This is only necessary when we are using the SageMaker SDK. You will need to specify a name for your endpoint configuration, the instance type of the compute used for your model, and a list of model variants, one for each model hosted by the endpoint. We will discuss multi-modal endpoints, where a single endpoint serves traffic for multiple models, in depth, but the following example is for a single model configuration. You will also specify the number of instances to use to service inference traffic:

```
endpoint_configuration = sagemaker.create_endpoint_config(
  EndpointConfigName=”Your Endpoint Config Name”
  ProductionVariants=[
   {
    "VariantName": “Variant Name”
    "ModelName": “Model Name”,
    "InstanceType": “ml.g4dn.xlarge”
    "InitialInstanceCount": 1 # Number of instances to launch
    initially.
   }
  ]
)
```

Finally, with the model object and the endpoint configuration, we can deploy the endpoint itself. You will need a name for the endpoint and the name of the endpoint configuration. The command will respond with Amazon Resource Name (ARN) of the created endpoint:

```
endpoint = sagemaker.create_endpoint(
      EndpointName=”Endpoint Name”,
      EndpointConfigName=”Endpoint Config Name)
```

If you need to programmatically send traffic to your endpoint, you can use the Amazon SageMaker SDK for the **`invoke_endpoint`** Python method:

```
Endpoint_response = sagemaker.invoke_endpoint(
      EndpointName=”Endpoint Name”,
      Body=bytes('{"features": ["Inference Payload"]}', 'utf-
      8')
     )
```

Inference Recommender

You could, of course, treat an instance type as a hyperparameter and iterate over different options, analyzing the output to determine the optimal combination of speed, cost and latency. If you have worked in software delivery teams before, this process might even seem familiar to you, though it was likely called another name: load testing. In essence, we are doing the same thing: putting our hosted model under pressure in the form of requests to observe the results in terms of used processor, memory and response times.

Thankfully, AWS includes an API accessible from the Command-Line Interface (CLI), SageMaker Studio, or the SDK for Python that can perform the load testing for us.

Using the AWS SDK for Python (boto3), you can create an inference recommendations job, identifying whether you would like simple inference recommendations (default) or a full-detail load test (advanced). You will also need the Amazon Resource Name (ARN) of the model you registered with SageMaker using the same method as the register model above.

```
sagemaker.create_inference_recommendations_job(
  JobName = “InferenceRecommendationJob”,
  JobType = “default”, ## Or Advanced for full
  RoleArn = RoleVariable,
  InputConfig = {
    'ModelPackageVersionArn': ModelArnVariable
  }
)
```

Once the job has been submitted, it will run and profile different options for the model you provided, and you can check the results in the AWS console or via the **`describe_inference_recommendations_job`** API, as follows:

```
InferenceRecommendations =
sagemaker.describe_inference_recommendations_job(
        JobName=”InferenceRecommendationJob”)
```

The response will detail useful items like cost per hour or per inference request for your endpoint. It will also contain the **`InstanceType`** field, which is the recommended compute option for your endpoint.

Serverless

On the far end of the managed scale from self-managed is serverless inference. It is well suited for inference workloads that are unpredictable or only happen periodically and those that can tolerate small delays in initial responses (referred to as cold starts), and it manages the entirety of the compute resource, scaling, and management of the chosen compute. On the cost side, you pay only for the time the endpoint is in use and not for any time it is idle or unused, making it an appealing option both for experimentation and for workloads that need to be intermittently available. The only required parameters for a serverless inference endpoint are a registered model definition, like the one we created earlier, and an endpoint configuration that includes a production configuration serverless config. That config should include available memory in MB for the endpoint and a maximum concurrency that defines how many concurrent invocations your endpoint can handle at once:

```
ServerlessEndpointConfig = sagemaker.create_endpoint_config(
  EndpointConfigName="ServerlessEndpointConfigName",
  ProductionVariants=[
    {
      "ModelName": "YourModelName",
      "VariantName": "ServerlessTraffic",
      "ServerlessConfig": {
        "MemorySizeInMB": 3072,
        "MaxConcurrency": 6
      }
    }])
```

NOTE: Cold Starts, are a situation where your endpoint has not received traffic recently and the compute resources behind it need to be restarted, your inference container needs to be re-downloaded, and your model needs to be re-deployed. When you are using a serverless inference endpoint, ensure that your calling applications can handle variable amounts of initial latency in their requests.

Autoscaling

With AWS offering on-demand compute resources for hosting our models, we can also use automated management of the necessary resources that are added when a metric of our choice exceeds a predetermined threshold. Similarly, we can configure resource scale down when the same metric falls below a lower threshold. With this in place, our endpoint will be able to scale to match demand when it is needed, and then back down to save on cost when it is not.

To create an autoscaling policy, you first need to consider your scaling metric and scaling method. In terms of methods, there are two options: target-tracking and step scaling. Target-tracking is the recommended approach; it means that AWS will scale out (adding compute resources) to your model to maintain your chosen metric. For example, if your metric is **`InvocationsPerInstance`** equal to 100, then as the metric exceeds 100, more compute instances will be added. Once it falls below that same amount for a predefined period, those resources will be removed. Similarly, for step scaling, you set an upper and a lower static that control when compute is added to the pool, taking inference requests with your model.

Once you are ready to add an auto scaling policy to your deployed endpoint, you choose your target metric, set the minimum and maximum levels for scaling, a cool down policy that controls how much, in seconds, the policy will wait before adding or removing instances, and the role the policy will use to perform its actions.

For your target metric, choose one that represents the middle ground of where your model needs to stay to perform within the established Service-Level Agreement (SLA). The default metric is **`InvocationsPerInstance`**, which will manage how many invocations each compute instance in the pool handles at once. If you are just starting with auto scaling, start with this metric.

Your minimum and maximum capacity sets the lower and upper limits of the scaling policy. You don't want to scale all the way to zero, and you also don't want to have no upper limit in case of a burst of traffic. Consider using the mentioned *Inference Recommender* job to determine the expected traffic patterns of your model to set a scaling metric and your maximum and minimum levels.

Your cooldown period controls the interval in which a scaling policy will pause before adding or removing instances. The main purpose of this period is to make sure instances are not added or removed from the pool before the previous scaling activity has completed. The default value is 300 seconds, and you can set the values independently for scale in and scale out. The following is an example of a scaling policy that you can use in the next step, applying the policy to your registered model:

```
{
   "TargetValue": 120.0,
   "PredefinedMetricSpecification":
   {
    "PredefinedMetricType":
    "SageMakerVariantInvocationsPerInstance"
   },
   "ScaleInCooldown": 700,
   "ScaleOutCooldown": 200
 }
```

One you have your model defined, as shown here, and a scaling policy defined, you can apply the scaling policy to the registered model using the AWS CLI or the Application Auto Scaling API:

```
 aws application-autoscaling put-scaling-policy \
   --policy-name AutoScalingPolicy \
   --policy-type TargetTrackingScaling \
   --resource-id endpoint/ProductionVariantName \
   --service-namespace sagemaker \
   --scalable-dimension sagemaker:variant:DesiredInstanceCount
   \
   --target-tracking-scaling-policy-configuration
   file://scalingPolicy.json
```

Inference at the Edge

So far, the examples we have provided have involved hosting models within a cloud environment and sending traffic to them in that same environment, much like we would for a standard application. There are several workloads that need the model and the associated inference to be closer to where the workload is deployed or even on dedicated hardware in the field. A camera watching an assembly line that uses computer vision to spot defects, a gateway device monitoring financial transitions at a remote site, and a real-time translation service in a conference room could all be examples of inference at the edge.

This approach and implementation bring unique challenges, like limited local resources and delays in messages being returned to your centralized systems. Additionally, managing the various devices, locations and environments your field-based models might encounter requires customization.

Thankfully, Amazon SageMaker has a method to manage these kinds of remote (called edge) deployments called the SageMaker Edge Manager. Models deployed in this style will need to have a special preparation called compilation, which we will cover in this section. Other than this, SageMaker's Edge Manager allows you to effectively package, deploy, execute and monitor your models in various remote locations.

In short, this setup is creating an Internet of Things (IoT) workflow. The whole of IoT application management is a complete book in itself, so make sure to review it in detail. That said, Edge Manager handles much of the heavy lifting for you. You will need to create distinct IoT roles separate from your SageMaker IAM role, but other aspects are managed.

Model Compilation

SageMaker Edge Manager required compiled models to be able to deploy to your edge locations. Using the Amazon SageMaker service, SageMaker Neo allows us to do this after we have trained and evaluated our model. The following is using Tensorflow (the job expects the framework in all caps) and is to be deployed to a Raspberry Pi3.

```
sagemaker.create_compilation_job(CompilationJobName="Compilati
onJobName", RoleArn=sagemaker_role_arn, InputConfig={
       'S3Uri': s3_model_location,
       'DataInputConfig': {"data":[1,3,224,224]},
       'Framework' : "TENSORFLOW"}, OutputConfig={
       'S3OutputLocation': s3_output_location,
       'TargetDevice': "rasp3b"}, StoppingCondition=
       {'MaxRuntimeInSeconds': 1200})
```

Next, you will need to package your compiled model. The packaging process adds all the necessary components to make sure your model is ready to be deployed to the remote edge location:

```
sagemaker.create_edge_packaging_job(
    EdgePackagingJobName="PackagingName",
    CompilationJobName="CompilationJobName",
     RoleArn=sagemaker_role,
     ModelName=model_name,
     ModelVersion="1.0.0",
     OutputConfig={
     "S3OutputLocation": packaging_s3_output
     })
```

Once you have the model package and have registered a device with AWS IoT, you can use AWS IoT Fleet Manager to deploy the model package and the associated model to the device. You can also use Fleet Manager to update your models, deploy additional models, and get monitoring metrics from your models.

Deployment Mechanics

Since we are using Amazon SageMaker, when we went through the training process, we created an estimator object and called .fit() on that object to start our training. Using the same object, and assuming we are using the Amazon SageMaker Python SDK, we can call **`.deploy()`** to create an inference endpoint to use for validating our trained model. The following examples are not complete; for the whole working code, check the associated GitHub repository provided at the beginning of the book.

Now that we have a solid understanding of the concepts associated with model hosting, let's look at a few of the options that our new-to-inference SRE will take to help the data scientists on their team. Checking their ticket queue, our energetic SRE sees that there are three requests already waiting for them.

The first is to deploy a model to help with narrative generation. Apparently, the data science team is tired of writing filler text to go along with their abstracts and would like to use some of the new models they keep hearing about to help.

The second is to deploy a linear regression model where they have already completed the training. This model will be supporting an ongoing experiment and needs to be able to respond quickly. They are expecting responses from the model to be received in less than 100 milliseconds per request.

The last model is a classifier that the team is going to use to sort through a massive backlog of user-submitted images. The good news is that the data science team already has the ground truth labeled data set and a trained model, but the incoming images for inference will be intermittent, and the results only need to be received once a week.

In all three cases, the team needs the inference completed within the requested time window, but it also wants to avoid any excess spending as they work mostly on grant money.

Time to get to work.

Narrative Generation

Deciding to take the tickets in order, our SRE decides to start with the narrative generation request. Interestingly, the team did not provide a model for the deployment. Luckily, our SRE has kept up on machine learning trends and has heard a lot of buzz about recent mesh transformers that have been producing spectacular results. Additionally, our SRE is already familiar with the Hugging Face community and model hub, so they decide to deploy one of the models from there. After a quick check, it looks like there is a model card that will work perfectly.

Our SRE decides to use the EleutherAI gpt-neo-1.3B model for this task and creates a Hugging Face model hub definition and model object for that model:

```
HubEnvironment = {
 'HF_MODEL_ID':'EleutherAI/gpt-neo-1.3B',
 'HF_TASK':'text-generation'
}
ModelObject = HuggingFaceModel(
 env=hub,
 role=SageMakerRole,
 transformers_version="4.6"
 pytorch_version="1.7",
 py_version="py36",
)
```

Once we have those defined, we provide the model object to a predictor definition, along with an instance count and SageMaker instance type. Since this model will only be used when the data science team needs to add some text to their draft papers for publishing, it doesn't need a huge pool. It only needs an instance large enough to hold the model in memory with available GPUs. Choosing a P3 instance is the best option, and our SRE knows that they can schedule time for the model to be deployed to avoid leaving it running continually.

```
PredictorObject = huggingface_model.deploy(
  initial_instance_count=1,
  instance_type="ml.p3.2xlarge",
)
```

Once the model endpoint is deployed, we can test it using the **`.predict()`** function and some text that simulates an abstract:

```
PredictorObject.predict({
'inputs': "The aim of this study was to review the prevalence
and epidemiology of diabetes in patients with systemic
mastocytosis (SM), as well as the impact of SM on diabetic
control, with an emphasis on the pathophysiology of insulin.",
  'parameters': {
    'max_length': 600,
    'temperature':1.0,
    "return_full_text": False,
  }
})
```

To use this endpoint, our SRE will need to add an actual web application that the users will interact with; the notebook in the book's GitHub repo has an example application using Streamlit.

Linear Regression

Checking the next ticket, our SRE can see that the model has already been trained and, according to the Jupyter notebook provided by the Data Science team that they used to train it, registered with SageMaker Model Registry:

```
s3_model_url = "s3://YourBucketName/prefix/model.tar.gz"
modelpackage_inference_specification = {
    "InferenceSpecification": {
      "Containers": [
        {
```

```
        "Image": '683313688378.dkr.ecr.us-east-
        1.amazonaws.com/sagemaker-xgboost:1.5-2',
    "ModelDataUrl": s3_model_url
      }
     ],
     "SupportedContentTypes": [ "text/json" ],
     "SupportedResponseMIMETypes": [ "text/json" ],
  }
}
ModelPackageInput = {
   "ModelPackageGroupName" : "ModelPackageGroupName",
   "ModelApprovalStatus" : "Approved"
}
ModelPackageInput.update(modelpackage_inference_specification)
CreateModelResponse =
sagemaker.create_model_package(**ModelPackageInput)
ModelPackageARN =
create_model_package_response["ModelPackageArn"]
```

From these details, our SRE can see that there is a **`ModelPackageARN`** object that we can use to refer to the registered model version and similarly, to deploy the model:

```
ModelResponse = sagemaker.create_model(
  ModelName ="ModelName",
  ExecutionRoleArn = role,
  Containers = ModelPackageARN
)
```

Once we have the create model response, we can use that object to create an endpoint configuration, just like we did earlier. One thing our SRE noticed about the model needs was that despite needing the responses to be very fast, they didn't mind some **initial** latency. The calling application had a built-in backoff that would just retry, and as long as more than 90% of the responses were faster than 1 second, it would meet their needs. Armed with this information, our SRE decides to deploy a serverless endpoint to meet the cost optimization needs as well.

```
create_endpoint_config_response =
sagemaker.create_endpoint_config(
 EndpointConfigName = "EndpointConfigName",
 ProductionVariants=[{
    "ModelName": "ModelName",
    "VariantName": "AllTraffic",
    "ServerlessConfig": {
     "MemorySizeInMB": 2048,
     "MaxConcurrency": 10
    }}]
)
```

Now that we have the endpoint configuration, we can call the **`create_endpoint()`** method, use **`EndpointResponse`** to get the deployed URL, and send that to the data science team:

```
EndpointResponse = SageMaker.create_endpoint(
  EndpointName="LinearRegressionEndpoint",
  EndpointConfigName="EndpointConfigName"
)
```

Computer Vision

Finished with the second ticket, our SRE reviews the last one. Since this only needs to run inference once a week, it seems the best suited for asynchronous or batch style inference. Since asynchronous is mostly used for very large inference payloads that won't finish by the time a real-time application can wait, our SRE decides to go for a batch style of deployment.

Since the source images would be stored in an S3 bucket, the SRE decided to consider that bucket the source location and would set up a separate bucket to store the processed images and a separate folder in the source bucket for any images that had an error during processing. Given this setup, each time the batch inference process started, it would list the items in the source bucket and use those as the agenda for inference.

Tempted to use the classic shell script triggered by a cron expression on an EC2 instance, our SRE considers for a moment before deciding to explore further. The EC2 cron approach would work, but it would violate the concept of cost optimization and did not suit the on-demand nature of cloud computing. Quickly checking to see if AWS has a capability to provide for state-machine style workflows, they find AWS Step Functions.

AWS Step Functions is a serverless orchestration service. It allows you to create workflows that need different paths, branches, and logic. It also natively supports interacting with many AWS APIs, `CreateEndpoint` in this case, which is exactly what we need. We will still need to have some helper jobs before inference to list the images ready for inference and to manage the interaction with the created SageMaker endpoint, as well as moving the images to the completed bucket, but those can easily be accomplished as AWS Lambdas (a fully managed serverless compute service from AWS - **https://aws.amazon.com/lambda/**).

Figure 8.1 shows a visual example of the AWS Step Function Flow that can be created for our use:

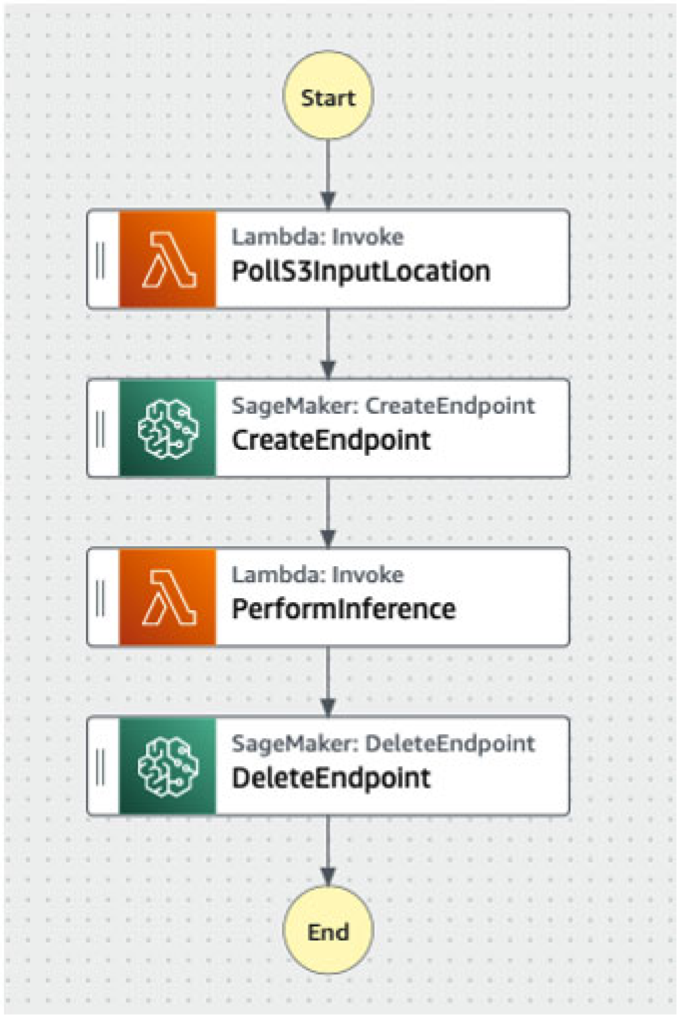


Figure 8.1: *Amazon SageMaker Step Function flow*

AWS Step Functions provides an easy-to-use graphical interface for creating workflows, but we can also represent those workflows as code. Following is an example of the state machine, triggered by an AWS CloudWatch Event Bridge cron expression, used to scan the incoming S3 bucket, deploy the endpoint, send each of the images for inference, and then destroy the endpoint. The associated Lambdas are also available in the book's associate GitHub repo:

```
{
 "Comment": "State Machine that Performs Inference",
 "StartAt": "PollS3InputLocation",
 "States": {
 "PollS3InputLocation": {
  "Type": "Task",
  "Resource": "arn:aws:states:::lambda:invoke",
  "OutputPath": "$.Payload",
  "Parameters": {
  "Payload.$": "$",
  "FunctionName": "arn:aws:lambda:us-east-
  1:123456789012:function:polls3function:latest"
  },
  "Retry": [
  {
   "ErrorEquals": [
   "Lambda.ServiceException",
   "Lambda.AWSLambdaException",
   "Lambda.SdkClientException"
   ],
   "IntervalSeconds": 2,
   "MaxAttempts": 6,
   "BackoffRate": 2
  }
  ],
  "Next": "CreateEndpoint"
 },
 "CreateEndpoint": {
  "Type": "Task",
  "Resource": "arn:aws:states:::sagemaker:createEndpoint",
  “Parameters”: {
  “EndpointConfigName”: “string”,
  “EndpointName”: “string”,
  “Tags”: [
   {
   “Key”: “string”,
   “Value”: “string”
   }
  ]
  },
  “Next”: “PerformInference”
 },
 “PerformInference”: {
  “Type”: “Task”,
  “Resource”: “arn:aws:states:::lambda:invoke”,
  “OutputPath”: “$.Payload”,
  “Parameters”: {
  “Payload.$“: “$”,
  “FunctionName”: “arn:aws:lambda:us-east-
  1:123456789012:function:performinference:latest”
  },
  “Retry”: [
  {
   “ErrorEquals”: [
   “Lambda.ServiceException”,
   “Lambda.AWSLambdaException”,
   “Lambda.SdkClientException”
   ],
   “IntervalSeconds”: 2,
   “MaxAttempts”: 6,
   “BackoffRate”: 2
  }
  ],
  “Next”: “DeleteEndpoint”
 },
 “DeleteEndpoint”: {
  “Type”: “Task”,
    "End": true,
    "Parameters": {
    "EndpointName": "MyData"
    },
    "Resource": "arn:aws:states:::aws-
    sdk:sagemaker:deleteEndpoint"
   }
   }
 }
```

Satisfied that they have the batch inference request in hand and are thoroughly testing their new Step Function, our SRE (and new Machine Learning Operations Engineer) relaxes and digs into their next task.

After the Deployment

Once your model is deployed and your team can get successful inference, there is more to be considered. Unlike typical application management, machine learning models will not normally return errors when invalid requests are sent; rather, their predictions will be less accurate. Additionally, since your model was likely trained from historical data, if your situation changes, that data may no longer relate to the live real-time data you are sending to your model for inference. Historically, machine learning models were considered *closed-box* applications with little to no understanding of why they made the predictions they did. Now, with modern model monitoring and baselining, we can observe the state of our model's predictions and the state of the data being sent to the model, and we can provide explainability as to why the model is making the predictions it is making.

Model Monitoring

AWS CloudWatch allows you to review, alert, and act on a number of metrics emitted by your SageMaker endpoint. You can monitor for the response code, resource utilization, logs generated, and similar metrics within CloudWatch. These are like standard application monitoring; while they should not be ignored, they only describe part of the picture of what is happening with your deployed model.

Beyond these structured metrics, we need to monitor for model quality, which can be accomplished with the aptly named Amazon SageMaker Model Monitor. For each of our monitoring scenarios, we will need to enable data capture. Enabling this feature stores the inputs and outputs of your inference in an s3 bucket. You can use this stored data to create baselines for each scenario. You can add the data capture configuration block after the production variant block in your endpoint configuration:

```
DataCaptureConfig= {
  'EnableCapture': True,
  'InitialSamplingPercentage' : 25, #Integer of percentage of
  requests sampled
  'DestinationS3Uri': “s3://EndpointDataCaptureBucket/”,
  'CaptureOptions': [{"CaptureMode" : “Input” ] # Can be Input,
  Output or both
  }
```

Once you deploy your model with the data capture configuration included in your endpoint configuration, the sampling percentage of transactions, along with metadata, will be captured in the listed S3 bucket. This is the data we will use to create baselines that, when compared against, determine the health of our model.

Data Drift

Once we have data capture enabled as we just mentioned, we can create a baseline job. As there are expected variations in the data sent to a machine learning model, it isn’t enough to check for types in the incoming data. The baselining job will create a profile of the incoming data that includes the expected types, and also includes things like standard deviation, median, mean, constraints, missing columns, and ongoing statistics. The following is an example of creating a baselining job based on the data you used to train your model. Since it is already indicative of the data your model will be performing inference on, it is a good way to get an idea of the details of that data:

```
ModelMonitor = DefaultModelMonitor(
  role=SageMakerModelRole,
  instance_count=1,
  instance_type='ml.c5.xlarge',
  volume_size_in_gb=20
 )
 ModelMonitor.suggest_baseline(
  baseline_dataset=’s3://your_data_buclet/training-dataset-
  with-header.csv',
  dataset_format=DatasetFormat.csv,
  output_s3_uri=’s3://your_data_bucket/modelmonitorresults/’
 )
```

Metrics emitted by the model monitoring job can be reviewed, graphed, and alerted upon within Amazon CloudWatch.

Model Drift

Model drift or bias drift can occur when the data sent to a model for inference differs informationally from the data used to train the model. This can happen gradually, as trends change, or suddenly, as a response to changes in your domain. Consider a model that calculates optimal supply chain schedules prior to 2020. The historical data it was trained on did not contain any of the sudden challenges associated with the COVID-19 pandemic, so it won’t be able to make quality predictions with data after that point.

Even in the best models, model conceptual drift is expected over time. Historically, this was managed by regularly retraining a model with freshly collected data, but it is also possible, using our enabled data capture, to monitor for drift in our predictions. To create a model drift monitoring job, a baseline of how and why predictions were made must first be created, and then a regular monitoring job can be run. The metrics emitted by this job can be graphed and alerted to inform your data science teams for the presence of drift:

```
 BiasMonitor = ModelBiasMonitor(
  role=SageMakerRole,
  sagemaker_session=sagemaker,
  max_runtime_in_seconds=900,
 )
 BiasMonitorDataConfig = DataConfig(
  s3_data_input_path=”s3://validation_dataset_location/input/”,
  s3_output_path=””s3://validation_dataset_location/output/,
  label=label_header,
  headers=all_headers,
  dataset_type=dataset_type,
)
```

Model Quality

Since we have already enabled data capture, we can also monitor for quality, comparing the predictions that the model makes against the ground truth labeled data that was used to train it. Like with Data Drift, we need to create a baseline job that does the ground truth labels against the predictions of the model and creates a statistical model of those comparisons. Then, you create a schedule of monitoring jobs that will compare the baseline against a capture of inference requests. These results can be viewed, graphed, and alerted within Amazon CloudWatch.

To create a model quality baseline job, set the compute used and the duration for the baseline job to run:

```
ModelMonitor = ModelQualityMonitor(
  role=SageMakerole,
  instance_count=1,
  instance_type='ml.c5.xlarge',
  volume_size_in_gb=30,
  max_runtime_in_seconds=2700,
  sagemaker_session=sagemaker
)
job = ModelMonitor.suggest_baseline(
  job_name=”ModelMonitorJob”,
  baseline_dataset=”s3://baseline_s3_location/input/”,
  dataset_format=example_dataset.csv(header=True),
  output_s3_uri = s3://baseline_s3_location/output/,
  problem_type=”Regression”,
  inference_attribute= "PredictionColumn",
  probability_attribute= "ProbabilityColumn",
  ground_truth_attribute= "LabelColumn" #
)
```

Updating a Deployed Model

Once your model has been deployed and is taking inference requests, you will, at some point, need to deploy another version of the same model. You have a new model trained on new data, a model with higher accuracy, or a model that accepts new formats of input data. Regardless of the reason, you will eventually need to replace your existing model, and, when you do, there are options for how you replace that model. The first and basic option is simply to deploy another model over your existing one. It certainly wins for simplicity but loses quite a bit in availability, since any requests to the model during deployment will fail. It also suffers from a lack of preview capability. If your model does not act like you expect once it is deployed, you won't know until it is deployed and you observe the issues either in your tests (you are testing your deployed models, right?) or in your data capture metrics. The last issue with this option is that it only gives you metrics associated with your testing to decide whether you want to replace your existing model at all.

What if there was a way to address these issues, gather data about your model, preview its performance, validate its functionality, and assess its accuracy, all without disrupting the traffic to your existing model beyond a preset threshold of your choosing? Not only is there a solution, but you've already used and interacted with the method we will use: the production variant.

In the previous examples, we used a single production variant when creating the endpoint configuration. By using multiple production variants, we can manage multiple machine learning models per endpoint, including what percentage of traffic we send to each variant.

Blue/Green Deployments

Our SRE is already familiar with Blue/Green deployments from supporting other workloads, and the same concepts apply here. Typically, a Blue/Green deployment strategy is when you create two identical environments and then switch from one environment that is running the existing application (blue) over to the environment running the new application (green). The color designations are used to identify where the traffic is flowing. We could add an abstraction layer on top of our model endpoint in the form of a network load balancer or API gateway, but in the interest of optimizing for cost and efficiency, we will leverage the built-in capability of production variants to accomplish it within a single endpoint.

We will use the same linear regression endpoint used earlier with the caveat that the data science team has provided a new model that they would like to switch traffic to without losing any from the initial model. Thus, our endpoint configuration becomes as follows:

```
CreateEndpointConfig = sagemaker.create_endpoint_config(
 EndpointConfigName = "EndpointConfigName",
 ProductionVariants=[{
   model_name="ModelNameBlue",
   instance_type="ml.m=c5.xlarge",
   initial_instance_count=1,
   variant_name='BlueModel',
   initial_weight=1
   },
 {
   model_name="ModelNameGreen",
   instance_type="ml.m=c5.xlarge",
   initial_instance_count=1,
   variant_name='GreenModel',
   initial_weight=0
   },
}])
```

Based on the weights mentioned, the blue model will take 100% of the traffic, and the green model will take 0%. Once we are ready to shift traffic to the green model, we can update the endpoint configuration as shown here:

```
sagemaker.update_endpoint_weights_and_capacities(
 EndpointName="EndpointName",
 DesiredWeightsAndCapacities=[
  {
   "DesiredWeight": 0,
   "VariantName": variant1["BlueModel"]
  },
  {
   "DesiredWeight": 1,
   "VariantName": variant2["GreenModel"]
  }])
```

The weights applied can also be any fraction of 100, allowing you to send percentages of traffic to your endpoint. This allows you to validate that your model functions as expected with a small percentage of traffic and also allows you to reverse course and shift the weight back down to zero if there is a problem.

A/B Testing

Using this method, you can perform A/B testing on your models to determine how they perform before implementing one and sending all your traffic to it. A/B testing refers to the practice of testing different variants of your model and comparing the respective performance, metrics, and responses to determine which model you wish to ultimately keep. You can set the weights of all production variants to 1, which means each model will get the same amount of traffic, allowing you to determine the ultimate performance of each option.

Additionally, if you have control on the calling application, you can set which of your production variants are called on each translation during the endpoint invocation by adding the **`TargetVariant`** parameter:

```
ModelInvocation = sagemaker.invoke_endpoint(
  EndpointName="ModelName",
  ContentType="text/json",
  Body=ModelPayload,
  TargetVariant=variant1["BlueModel"],
)
```

This is especially useful where certain users opt in as beta participants, allowing you to insert the **`TargetVariant`** parameter into their model requests.

Multi-Model Endpoints

As your machine learning use increases, it is likely that you will reach a point where a workload will need multiple machine learning models. There can be cases where one model calls another to further refine the data needed for a prediction, or where the trained models need to be trained on specific subsets of data that cannot be generalized across all data. For example, you have a dataset containing historical metrics across India for 28 states and 8 Union territories. Your model needs to be able to make predictions for users in specific states or union territories. In order to increase accuracy to a level that meets your business need, you split your training data by state and union territory. The result will be a unique model for each of the state and union territory leading to a total of 36 models.

In order to perform inference on all your 36 models, you would need an inference endpoint for each, meaning you would pay the compute costs for each. Instead, you can host all 36 models behind a singular endpoint and specify the model the request is intended at inference time. To do this, when we define the container used to host the model in the model definition, we add in the MultiModel mode parameter:

```
MultiModalModel = sagemaker.create_model(
     ModelName = 'ModelName',
     ExecutionRoleArn = SageMakerRole,
     Containers = {
       'Image': image_repo_location,
       'ModelDataUrl': 's3://model_bucket/model_path',
       'Mode': 'MultiModel'
       })
```

The models you will add to this endpoint should all be contained within the s3 location. Each model artifact will be added to the endpoint and hosted out of the container you specify in the Image parameter. If you need to host a multi-modal endpoint with models in separate containers, the specification will be as follows:

```
MultiModalModel = sagemaker.create_model(
     ModelName = 'ModelName',
     InferenceExecutionConfig = {'Mode': 'Direct'},
     ExecutionRoleArn = SageMakerRole,
     Containers = [{ 'Image': '123456789012.dkr.ecr.us-west-
     2.amazonaws.com/ecr_repo_1:latest', 'ContainerHostName':
     'ModelContainer_1'}, 'Image':
```

```
'123456789012.dkr.ecr.us-west-
2.amazonaws.com/ecr_repo_2:latest',
     'ContainerHostName': 'ModelContainer_2'
     }])
```

Once your model has been defined, you can create a Multi-model endpoint configuration and deploy the model:

```
MultiModalEndpointConfig = sagemaker.create_endpoint_config(
     EndpointConfigName = 'MultiModalEndpointConfig',
     ProductionVariants=[
       {
        'InstanceType': 'ml.c5.xlarge',
        'InitialInstanceCount': 2,
        'InitialVariantWeight': 1,
        'ModelName': 'MultiModalModel',
        'VariantName': 'AllTraffic'
       }
     ]
     )
MultiModalEndpoint = sagemaker.create_endpoint(
     EndpointName = 'MultiModelEndpoint',
     EndpointConfigName = 'MultiModalEndpointConfig')
```

The requests you send to the Multi-model endpoint should contain the target model parameter identifying the model artifact to route the request to. You can use the SDK to invoke the model as shown earlier or the AWS CLI.

```
Amazon SageMaker-runtime invoke-endpoint --endpoint-name
"MultiModelEndpoint" --body "{104.4, 102.2, 103.3}" --content-
type "text/json" --target-model "SpecificModelArtifact.tar.gz
InferenceResults.txt
```

Conclusion

In this chapter, we started our inference journey. Just like classic application deployments, the methods, means, and options for machine learning deployments are varied. This is a field of expertise that is expanding at a remarkable rate alongside the size and complexity of models. The deployment aspect of a machine learning model's life cycle should be treated as an interactive experiment apart from the training aspect. Singleton deployment is preferable for the training and validation steps of machine learning model creation, but more consideration is needed once your model is ready to be deployed and used for customers.

That consideration should include your deployment option between real-time, asynchronous, and batch. It should include specifying the exact compute instance type that will power your model or opting for a serverless implementation. It should include the type of data being sent to your model and how you want your model to respond to determine whether and how it should scale.

Finally, the deployment is just part of the journey. Monitoring the health, quality, and validity of your model is an ongoing process that requires some unique methods that can leverage the power of Amazon SageMaker to provide clarity.

Both we and our SRE have finished this part of our journey on deploying our models; out journey will continue in *Chapter 9, Wisdom at Scale With Elastic Inference*.

Points to Remember

During the initial experimental phase of a machine learning project, deploying a model locally is acceptable, but when you think of production deployments, you need to be better prepared. At its core, your model needs the support of a transactional system to supply it requests and return the responses. Not too long ago, simply putting your model behind a REST endpoint from your notebook was acceptable. Now, with Machine Learning applications gaining first-class citizen status alongside long-standing application development projects, a more mature, repeatable, and flexible approach is needed. Here are a few points to remember when considering your deployment approach:

- Your model will need to be able to accept the data profile sent to it. Sending a single feature is one thing, but large images, long blocks of text, or massive tensors are another; make sure your serving application can accept and manage the data payload being sent.

- Your model will likely be part of a larger, complete, end-to-end machine learning application. It is not uncommon that a request is first enriched with other data sources, transformed, passed through other models, or even (de) compressed before it gets to your model. Make sure you have complete visibility and that your model is ready for integration rather than simply being experimental.
- DevOps is more than a catchphrase. DevOps is a function of information. Make sure you monitor your model's platform metrics like memory, CPU, disk operations, and also the unique elements of machine learning models. Use this information to inform your deployment methods.
- You will rarely work in isolation on any project, much less on one focused on machine learning tasks. Make sure you design deployment methods that include change management approvals; operations considerations like documentation, runbooks, and automation; and your fellow data scientists' visibility.

Multiple Choice Questions

Use these questions to challenge your knowledge of deploying machine learning models on AWS.

1. **Which of these is not a deployment option available for Machine Learning models in Amazon SageMaker?**
   a. Real-time
   b. Part-time
   c. Batch
   d. Asynchronous
2. **What is the Amazon SageMaker capability that can be leveraged to get recommendations on endpoint configurations?**
   a. SageMaker Compute Recommender
   b. SageMaker Cost Optimizer
   c. SageMaker Inference Recommender
   d. SageMaker Endpoint Confabulator

3. **How can you scale the compute associated with your Machine Learning model deployment once it is in production?**
    a. SageMaker just sort of does it for you
    b. Application Auto Scaling
    c. Application Right Sizing
    d. SageMaker Load Balancer
4. **Which of the following is most accurate for the kind of models that can be hosted with Amazon SageMaker?**
    a. Compiled models
    b. Regression models
    c. Any model that can run on Graviton
    d. Any model that adheres to the documented container specification
5. **What is the service that allows you to deploy models to edge devices?**
    a. Amazon SageMaker Model Registry
    b. Amazon SageMaker Edge Pusher
    c. Amazon SageMaker Cog Master
    d. Amazon SageMaker Edge Manager

Answers

1. **b**
2. **c**
3. **b**
4. **d**
5. **d**

Further Reading

The following are select readings that expand on the topics presented in this chapter and are resources to continue your learning journey:

- *Machine Learing Ops using Edge Devices*: **https://aws.amazon.com/blogs/machine-learning/mlops-at-the-edge-with-amazon-sagemaker-edge-manager-and-aws-iot-greengrass/**
- *Scaling Inference with Yolo5*: **https://aws.amazon.com/blogs/machine-learning/scale-yolov5-inference-with-amazon-sagemaker-endpoints-and-aws-lambda/**
- *Accelerating Hugging Face Transformers*: **https://huggingface.co/blog/bert-inferentia-sagemaker**
- *Fine Tuning and Deploying Self-Managed Scripts*: **https://aws.amazon.com/blogs/machine-learning/fine-tune-and-deploy-a-summarizer-model-using-the-hugging-face-amazon-sagemaker-containers-bringing-your-own-script/**
- *Curated SageMaker Examples*: **https://github.com/aws/amazon-sagemaker-examples**
- *Understanding how to leverage Experiments and Pipelines*: **https://aws.amazon.com/blogs/machine-learning/organize-your-machine-learning-journey-with-amazon-sagemaker-experiments-and-amazon-sagemaker-pipelines/**

CHAPTER 9

Wisdom at Scale with Elastic Inference

Introduction

SageMaker is such a cool name for the AWS AI/ML service, don't you think? According to the author, there could not be a more apt name for the service. It literally means "the service that makes a sage", which we could paraphrase as "that which enables the creation and delivery of wisdom". In practical terms, we could presume that SageMaker will help us create models, aka "sages", with the ability to detect and predict new information, namely, "wisdom". The term "maker" indicates that the models ("sages") have to be trained ("made"), and this service enables us to do that in a standardized, easy-to-use manner. As we discussed in the previous chapters, Amazon SageMaker (**https://aws.amazon.com/sagemaker/**) is a managed AWS service providing end-to-end capabilities to build and run ML models at scale. SageMaker provides several broad and deep features to cater to every aspect of an ML workflow across domains, use cases and industries. If you recollect what we learned in *Chapter 1, Introducing the ML Workflow*, and some of the other chapters where we covered specific aspects of the workflow, you will see that SageMaker has removed all the guesswork and hard work associated with what traditionally used to be really complex tasks, including data pre-processing, feature engineering, algorithm selection, model training, inference and deployment.

In *Chapter 3, Predicting the Future with Features*; *Chapter 4, Orchestrating the Data Continuum*; *Chapter 5, Casting a Wider Net*; *Chapter 6, Iteration Makes Intelligence*; and *Chapter 7, Let George Take Over* of this book, we learned how to use Amazon SageMaker notebooks to perform feature engineering and feature selections and how to use AWS Glue ETL to build data orchestration pipelines. We also reviewed common approaches and best practices to select algorithms and neural network architectures, trained and tuned our ML model with SageMaker training, walked through using low-code, no-code ML options with SageMaker Canvas and AutoGluon, and learned about various options to get your trained model ready for inference with SageMaker Model Deployment. In this chapter, we will focus specifically on setting up enterprise scale inference endpoints for real-time predictions and batch transform options for non-real-time asynchronous predictions. We will learn how to create a model package from our trained model, use SageMaker for building real-time inference endpoints, run batch inference, and use the purpose-built feature AWS Elastic Inference and leverage the custom chipset AWS Inferentia.

Structure

In this chapter, we will dive deep into the following topics:

- Understanding SageMaker ML Inference options
- Running Inference with real-time endpoints
- Running Inference with serverless endpoints
- Running Inference with Batch Transform
- Running Inference with SageMaker Elastic Inference

Objectives

In this chapter, we will discuss ML inference and understand why is it important. We will learn about the different types of inference options we can set up with SageMaker hosting both real-time and batch. Further on, we will learn about Elastic Inference, an accelerator instance for ML inference. Finally, we will execute code samples to prepare, train models and try out some of the inferencing options that SageMaker supports.

Understanding SageMaker ML Inference options

Just like wisdom that can be imparted in one fell swoop instantly or can be gained as an outcome of a process that matures over time, SageMaker provides you several options to stand up your model to support inference requests. Consider the following image showing the various ways in which you can host and run inference of your ML models on SageMaker. As you can see, based on your business problem, your requirements, how you want to make your model accessible to consuming applications or users, and depending on the cost or capacity constraints you may have, you can configure SageMaker to support your model inference accordingly. We will dive deep into each of these options in this section. Refer to *Figure 9.1*:

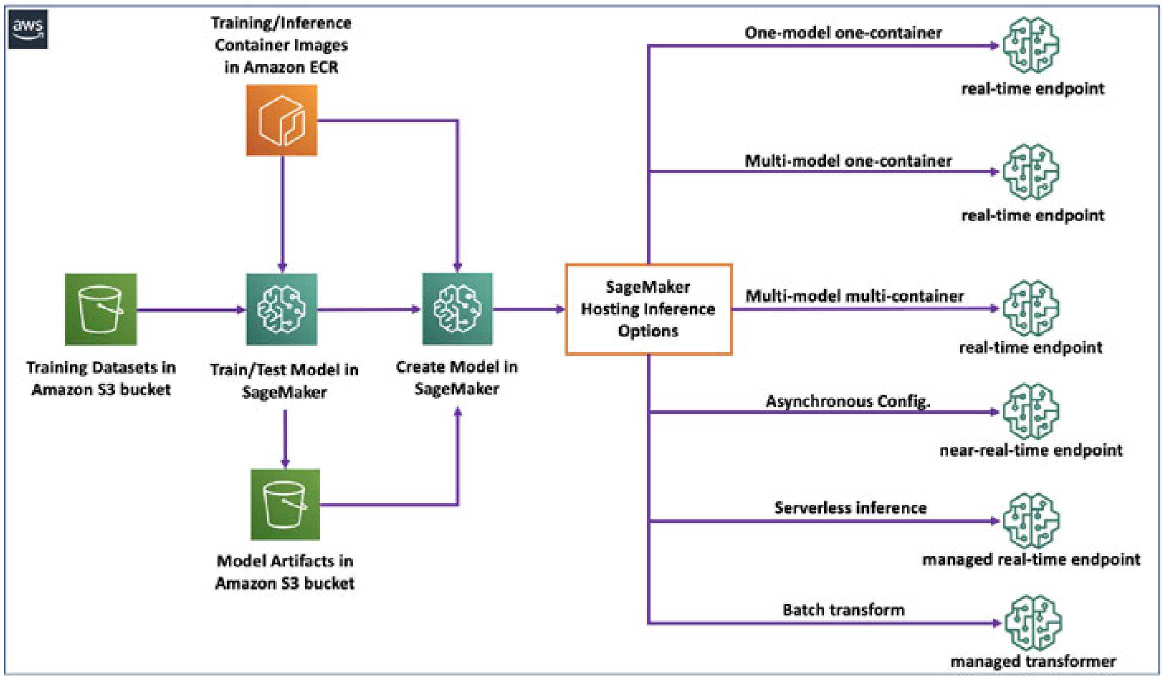

Training/Inference Container Images in Amazon ECR
Training Datasets in Amazon S3 bucket
Train/Test Model in SageMaker
Create Model in SageMaker
Model Artifacts in Amazon S3 bucket
SageMaker Hosting Inference Options
One-model one-container
real-time endpoint
Multi-model one-container
real-time endpoint
Multi-model multi-container
real-time endpoint
Asynchronous Config.
near-real-time endpoint
Serverless inference
managed real-time endpoint
Batch transform
managed transformer

Figure 9.1: *SageMaker Hosting inference options*

In addition to these hosting options, SageMaker provides purpose-built accelerators to help you improve model performance efficiencies, such as SageMaker Elastic Inference (**https://docs.aws.amazon.com/sagemaker/latest/dg/ei.html**) and the AWS Inferentia chipset (**https://aws.amazon.com/machine-learning/inferentia/**). We will learn about these accelerators in this chapter, along with code samples of how to use them for your ML needs. But first, let us review what the various inference options actually mean, why we need them, and how to use them. The biggest use of a model is its ability to detect useful information and predict future events. The detection and prediction capabilities of a model are realized during inference, after training is complete and the model is deployed or hosted. In SageMaker, the model hosting begins with executing the CreateModel API (**https://docs.aws.amazon.com/sagemaker/latest/APIReference/API_CreateModel.html**), which requires a Docker container image of model and algorithm dependencies, and the trained model artifacts to create a callable version of the model. After the model is created, it can be hosted on a compute resource called an endpoint and invoked with a request message to received prediction outputs. This function is called running an inference or making predictions. In this section, we will discuss the options available during model creation and hosting because the inputs vary based on a chosen configuration option for the hosting.

SageMaker real-time endpoints for one-model one-container

As the name indicates, real-time endpoints provide a request/response paradigm of interaction for running inference with the model, with the requestor waiting until a response is received. Remember the ATM cash withdrawal flow we reviewed in *Figure 5.2*: *Example of a deterministic algorithm* in *Chapter 5, Casting a Deeper Net?* Assume that you have a mobile banking application similar to that flow, using which you made a request to check your account balance. Until the response to that action is received, the mobile application holds that page and does not allow you to perform a new action. This is a basic example of a request/response. Another example is when you log in to your banking application. When your app makes the login request based on credentials you entered, its process thread cannot execute any other action until a response is received. There are several such examples, like when you check out at Amazon.com and click on **Submit**, the requesting page waits until the order is submitted and an acknowledgement is received. Similarly, when you set up a SageMaker real-time endpoint, you submit an inference request and wait until you receive a response from the endpoint before proceeding to execute the next line of code. Real-time endpoints are well suited for low latency, high throughput inferencing needs like fraud detection for banking transactions, stock price predictions, airplane autopilot course corrections, and real-time threat detection for security systems. Refer to *Table 9.1*:

API to use	Parameters for option	Value types for option
CreateModel	PrimaryContainer	Image ModelDataUrl
CreateEndpointConfig	ProductionVariants	InstanceType InitialInstanceCount
CreateEndpoint	EndpointName EndpointConfigName	

Table 9.1: *SageMaker Hosting option for one-model, one-container endpoints*

In a one-model, one-container option, we host a single model using a single serving container that contains all the dependencies for the model to create the callable model. We specify the serving container using `PrimaryContainer` in the `CreateModel` API. The model is then hosted by creating an endpoint configuration specifying a SageMaker ML instance type, the count of initial instances, along with the model name used to run the `CreateModel` API. Finally, we execute the `CreateEndpoint` API to instruct SageMaker to provision a new compute instance, start it up, deploy the container image on to this instance, load up the model artifacts, and create a callable version capable of accepting inference requests and responding with a prediction. All this heavy lifting is done by SageMaker behind the scenes, saving you both time and money.

SageMaker real-time endpoints for multi-model one-container

Commonly called Multi-Model endpoints, this hosting option provides the ability to use a single endpoint to host multiple models in real time, providing applications with the flexibility to serve requests in multiple dimensions (a model per dimension, for example, if you have a use case to predict housing prices for the whole of the US, you may train separate models for each city and host all of them in a multi-model endpoint) simultaneously and save on hosting costs. The trick is to provide an Amazon S3 URI location to a folder containing artifacts for multiple models (with each model in its own folder) to the **`CreateModel`** API, but you need to ensure that your container image for serving includes all the dependencies and code to host all the models you want to use for inference with this endpoint. With multi-model endpoints, you can host 1000+ models in a single endpoint. Refer to the following table for the parameters you need to fill for the multi-model, one-container hosting option.

API to use	Parameters for option	Value types for option
`CreateModel`	Containers - specify a single container object	• Image • ModelDataUrl - specify S3 folder containing artifacts for multiple models • Mode - 'MultiModel'
`CreateEndpointConfig`	`ProductionVariants`	• InstanceType • InitialInstanceCount - specify at least two instances
`CreateEndpoint`	`EndpointName` `EndpointConfigName`	

Table 9.2: *SageMaker Hosting option for multi-model one-container endpoints*

Multi-model endpoints are very good at saving costs when compared to creating separate endpoints to host each model. However, the decision to use a particular model artifact occurs at runtime, when an inference request is received by the endpoint; hence, you should see a higher response latency compared to dedicated endpoints for each model. The good news is that SageMaker takes care of memory management of endpoint resources across models based on incoming traffic and observed scaling patterns during peak and off-peak usage.

SageMaker real-time endpoints for multi-model multi-container

An advanced configuration of multi-model endpoints is the option to specify multiple containers, aka multi-container endpoints. This removes the constraint with the single container multi-model endpoints where all models had to use either the same ML framework, such as MXNet, TensorFlow or PyTorch, or an algorithm family like XGBoost or Linear Learner. For a list of all the supported algorithms and frameworks for multi-model (single-container) endpoints, refer to **https://docs.aws.amazon.com/sagemaker/latest/dg/multi-model-endpoints.html#multi-model-support**.

Multi-container endpoints support providing multiple container images (up to a maximum of 15 containers) at the time of creation, along with providing multiple model artifacts for the various models that will service the inference requests. By providing the flexibility to use separate container images, you can now create a single endpoint that can automatically serve models from different frameworks and algorithm families, enabling the provisioning of an enterprise scale inference framework. Refer to *Table 9.3*:

API to use	Parameters for option	Value types for option
`CreateModel`	Containers - specify multiple container objects, one for each container image. The image should be pre-built with model framework dependencies and model artifacts.	• Image • ContainerHostName
`CreateModel`	`InferenceExecutionConfig`	Mode
`CreateEndpointConfig`	`ProductionVariants`	InstanceType `InitialInstanceCount` - specify at least two instances
`CreateEndpoint`	`EndpointName` `EndpointConfigName`	

Table 9.3: *SageMaker Hosting option for multi-container endpoints*

As you can see from the previous table, the main difference between multi-model and multi-container options is how you configure the parameters for the **`CreateModel`** API. In this option, you specify a **`Containers`** list of dictionaries with multiple unique container images, one for each model served by this endpoint. You also specify a host name for the container that is used to reference a specific container at the time of inference. You do not specify an S3 URI for model artifacts because the expectation is that you pre-build your container with your trained model. You also need to specify an additional parameter called **`InferenceExecutionConfig`** with a **`Mode`** that can be either **`Direct`** or **`Serial`**. **`Direct`** indicates that at the time of inference, a specific container will be referenced when the endpoint is invoked, and **`Serial`** refers to executing each container in sequence, as you would in an inference pipeline.

SageMaker endpoints for asynchronous inference

So far, we have learned how to run a real-time request/response inference paradigm for our models on SageMaker. Now, we will learn how to run asynchronous (**https://aws.amazon.com/about-aws/whats-new/2021/08/amazon-sagemaker-asynchronous-new-inference-option/**) or near-real-time inference. Note that asynchronous does not mean batch inference, which we will cover in a subsequent section, but it is more suited for scenarios where an immediate low latency response is not needed and the requestor is not waiting to receive a response. SageMaker will queue inference requests, process them together and notify the requestor using Amazon Simple Notification Service or SNS (**https://aws.amazon.com/sns/**) about whether the requests were successful or ended up in error. The results are sent to an Amazon S3 bucket specified at request time. Refer to *Table 9.4*:

API to use	Parameters for option	Value types for option
`CreateModel`	`PrimaryContainer`	• Image • ModelDataUrl
`CreateEndpointConfig`	`ProductionVariants`	• InstanceType • InitialInstanceCount
`CreateEndpointConfig`	`AsyncInferenceConfig`	• S3OutputPath • NotificationConfig
`CreateEndpoint`	`EndpointName` `EndpointConfigName`	

Table 9.4: *SageMaker Hosting option for asynchronous endpoints*

The configuration for **`CreateModel`** in the case of asynchronous hosting is the same as when you create a single-model, single-container real-time endpoint. The difference is in how you specify the **`CreateEndpointConfig`**. You add a new configuration option called **`AsyncInferenceConfig`** when creating the endpoint config and provide an Amazon S3 bucket and prefix for where your inference response outputs are to be stored. You also should specify a **`NotificationConfig`** object that contains SNS topics for success and error messages. Asynchronous inference hosting supports larger request message sizes (as big as 1GB) and can handle sustained processing times of up to 15 minutes. Ideally, you should use an AWS Lambda function to be triggered on receipt of the SNS notification of inference request completion. The SNS notification will contain both the input and the output parameters from the inference job so that you can successfully map which responses are for which requests. So, if you don't want to go full batch mode but are OK to wait for a little bit to receive your inference requests, and you need the ability to handle bigger message sizes, then asynchronous hosting option is the best for your needs.

SageMaker endpoints for serverless inference

Having the ability to configure hosted endpoints for real-time and near-real-time inference requests is great, and you can specify how much you want to scale up to by providing an initial instance count. But what if you had no idea what instance type you needed or if you are cost constrained to pay for even a single initial instance. Granted that you can specify an instance count of zero for asynchronous endpoints, but let us say you don't need the ability to process large message sizes or for longer durations, and you still want to save on inference costs. With real-time and near-real-time endpoints, you keep accruing costs for the duration the instances are up and running, not just when they are serving requests. So the minute you spin up these endpoints, you are billed, whether or not they are in use. To save on endpoint infrastructure costs, you can use serverless inference. With SageMaker serverless inference (**https://docs.aws.amazon.com/sagemaker/latest/dg/serverless-endpoints.html**), you can stop worrying about provisioning and managing inference instance counts, autoscaling policies and up time, because you are charged for when the instances are serving requests and not for whenever they are running. This helps implement a truly pay-as-you-go model for ML inference. Refer to *Table 9.5*:

API to use	Parameters for option	Value types for option

`CreateModel`	Containers - specify a single container object	`Image`
		`ModelDataUrl`
		`Mode - 'SingleModel'`

`CreateEndpointConfig`	ProductionVariants - ServerlessConfig	`MemorySizeInMB` `MaxConcurrency`
`CreateEndpoint`	EndpointName	

	EndpointConfigName	

Table 9.5: *SageMaker Hosting option for serverless endpoints*

For serverless inference, we specify a single container object with a container image, the Amazon S3 location of where the model artifacts are stored (we can also use a SageMaker Model Registry and provide a **`ModelPackageName`**), with the Mode as a SingleModel when creating our model using the **`CreateModel`** API. We also need to provide a **`ServerlessConfig`** input to our **`CreateEndpointConfig`** API call, specifying the **`MemorySize`** for our serverless compute and the maximum count of concurrency requests it should handle. The **`CreateEndpoint`** remains the same. And this is how we set up serverless inference with SageMaker hosting.

SageMaker transformer for batch inference

If our need is to process inference for large input datasets, our preferred option is SageMaker Batch Transform (**https://docs.aws.amazon.com/sagemaker/latest/dg/batch-transform.html**). For example, suppose your company's leadership want to evaluate the overall performance of your contact centre agents based on historical call transcripts. You train a text classification model that could predict whether an agent successfully helped a customer based on certain keyword references in the transcript. You iteratively train the model and evaluate whether it performs well. You now need to run this model against your historical call transcripts but notice that you have more than a million calls that you need to analyse. You cannot use real-time or asynchronous endpoints because that will be too time-consuming and cost prohibitive. This is where a batch inference option can help you. SageMaker Batch Transform abstracts the heavy lifting associated with infrastructure provisioning and management of batch jobs, and it makes it easy to set up and run batch inference. Refer to *Table 9.6*:

API to use	Parameters for option	Value types for option
`CreateModel`	`PrimaryContainer`	`Image` `ModelDataUrl`
`CreateTransformJob`	`BatchStrategy`	`'MultiRecord'`
`MaxPayloadInMB`	`<= 100MB`	

`TransformInput`	`S3DataSource`
	`SplitType - 'Line'`	

`TransformOutput`	`S3OutputPath`	
`TransformResources`	`InstanceCount` `InstanceType`	

Table 9.6: SageMaker Hosting option for batch transform

Similar to what we did in the case of real-time, we first must use the `CreateModel` API to build a model. However, instead of creating endpoints, we use SageMaker Transformer libraries to create a transform job. You will need to decide on a batch strategy you want to use, such as `SingleRecord`, if you want to send one row at a time for inference, or `MultiRecord` if you want to send mini-batch of rows for inference. For your `BatchStrategy` to be considered, you need to specify the `SplitType` in `TransformInput`. For example, you can choose a `MultiRecord` strategy with the `SplitType` as `Line` to indicate that every line is a new record. You should specify the `S3DataSource` in `TransformInput` containing the S3 location of your input dataset for inference and specify the `S3OutputPath` in `TransformOutput` to indicate where you want the results to be saved. You should also specify `TransformResources` to let SageMaker know what instance type and counts to use when setting up the batch transform job.

And that brings us to the end of this section. As you can see, Amazon SageMaker provides several flexible and easy-to-use options to host inference for various request modalities. Do not worry if this was too much to learn or understand; we are not going to be just covering theory in this book. We will also try out some of these inference options and see them in action. So if you are in the mood for some coding and want to learn SageMaker just as the experts would, fasten your seat belts and move on to the next section.

Running Inference with SageMaker Hosting

In the previous section, we discussed how SageMaker provides a number of inference options for different types of request architectures. In this section, we will roll up our sleeves and dive right into a SageMaker Studio notebook to execute code samples to set up a single-model, single-container real-time endpoint for a trained model and send some inference requests to test it out. We will then ask SageMaker to configure us a serverless endpoint for our model and see how it works. Finally, we will set up a batch transform job using SageMaker and learn how to run batch inference for a trained model. Get ready, it's coding time!

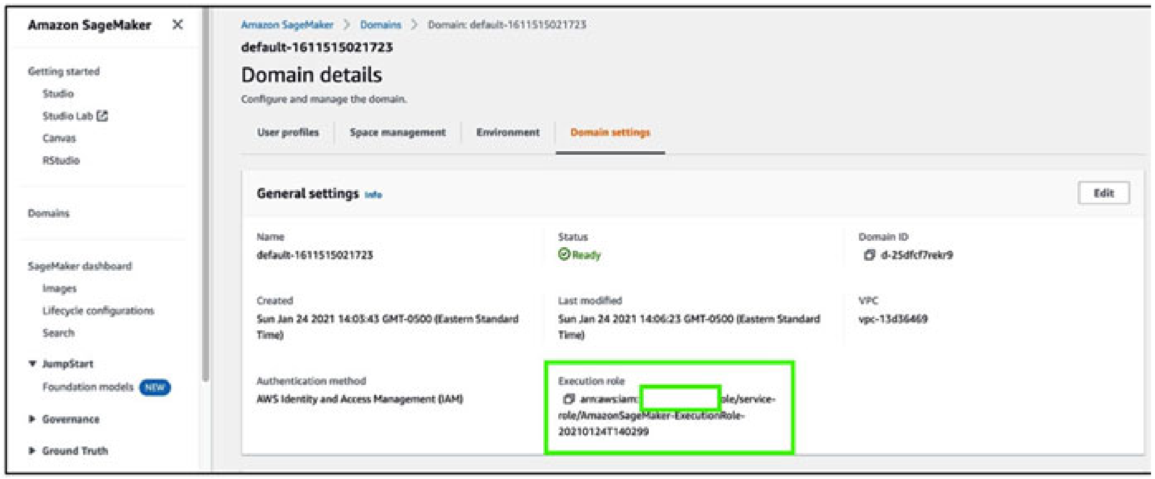
To execute the examples in this section, we will use SageMaker Studio notebooks; so, if not already done, you will have to onboard to SageMaker Studio and git clone our book repository. Follow the instructions in the Setting up your AWS account section of *Chapter 2, Hydrating Your Data Lake*, to sign up for an AWS account. Once you have signed up, log in to your AWS account using the instructions at

https://docs.aws.amazon.com/IAM/latest/UserGuide/console.html. You need to create an Amazon S3 bucket (**https://docs.aws.amazon.com/AmazonS3/latest/userguide/create-bucket-overview.html**) if you have not already done so in the previous chapters and note down the name of the bucket. Next, follow the instructions in the *Technical Requirements* section of *Chapter 3, Predicting the Future With Features*, to onboard to an Amazon SageMaker Studio domain and clone the book's GitHub repository (**https://github.com/garchangel/AIMLwithAWS**). Before we open our notebook, we need to set up IAM permissions for our SageMaker Studio role to allow working with our AWS AI services in this chapter.

Setting up IAM permissions

Our SageMaker Studio notebook comes up with default IAM permissions for working with SageMaker features and access S3 buckets, but it does not include access to the AI services we will use in this chapter. We will have to attach additional policies to our role to provide it with the necessary permissions. Execute the following instructions to augment your SageMaker role:

1. If not already, done, log in to AWS Management Console, type `SageMaker` in the services search bar at the top, and navigate to the SageMaker console. Under `Getting started` in the left menu, click on `Studio`. If you have already on-boarded to SageMaker Studio, you will see the username you created appearing under the Select user profile list box. Select your user profile and click on `Open Studio`. If you did not onboard to SageMaker Studio, do so by following the instructions in the *Technical Requirements* section of *Chapter 3, Predicting the Future With Features*, to onboard to an Amazon SageMaker Studio domain and clone the book's GitHub repository (**https://github.com/garchangel/AIMLwithAWS**).
2. In either case, navigate to the Amazon SageMaker console, click on `Domains` in the left pane, click on your domain name, and then on the `Domain settings` tab. Copy the Execution role ARN that is displayed in the `General settings` pane in the bottom-right corner of the page (by clicking on the two squares on the left of the role), as shown in the following image:



Amazon SageMaker > Domains > Domain: default-1611515021723

default-1611515021723

Domain details

Configure and manage the domain.

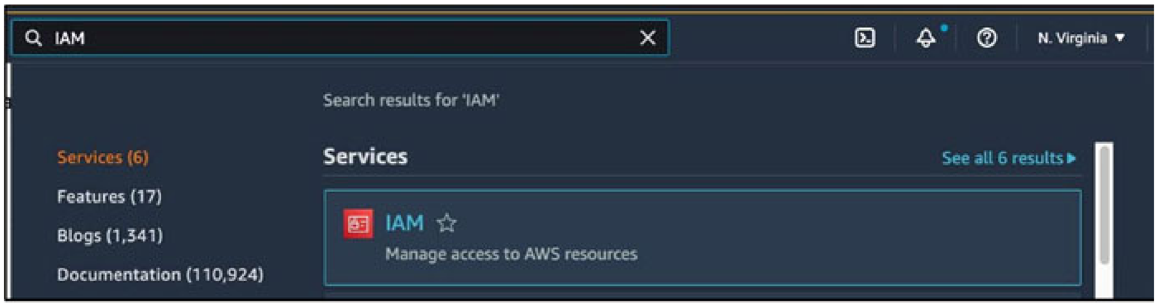
User profiles | Space management | Environment | Domain settings

General settings Info — Edit

Name	Status	Domain ID
default-1611515021723	Ready	d-25dfcf7rekr9
Created	Last modified	VPC
Sun Jan 24 2021 14:05:43 GMT-0500 (Eastern Standard Time)	Sun Jan 24 2021 14:06:23 GMT-0500 (Eastern Standard Time)	vpc-13d36469
Authentication method	Execution role	
AWS Identity and Access Management (IAM)	arn:aws:iam:[redacted]ole/service-role/AmazonSageMaker-ExecutionRole-20210124T140299	

Figure 9.2: *Copy Studio execution role ARN*

3. Now, type IAM in the services search bar and navigate to the IAM console, as shown in the following image:



IAM

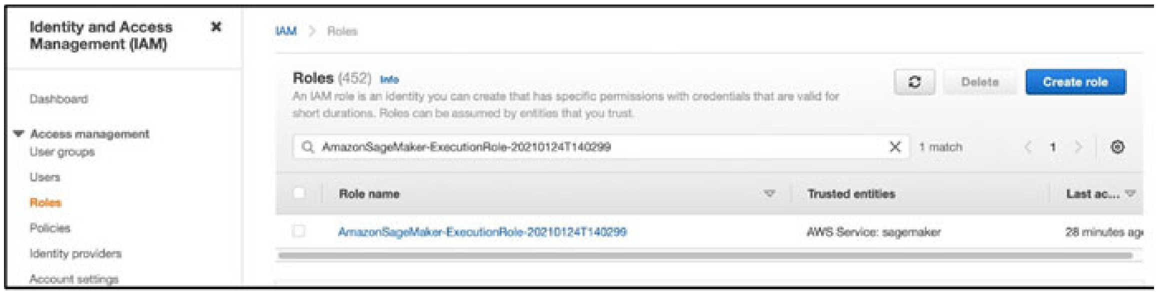
Search results for 'IAM'

Services — See all 6 results

IAM — Manage access to AWS resources

Figure 9.3: *Search for IAM service*

4. In the IAM console, select **`Roles`** from the left menu, and type only the role name from the ARN you copied, which is the portion of the ARN that begins with **`AmazonSageMaker-ExecutionRole`**. Press *Enter* to bring up the role. Refer to *Figure 9.4*:



IAM > Roles

Roles (452) Info

An IAM role is an identity you can create that has specific permissions with credentials that are valid for short durations. Roles can be assumed by entities that you trust.

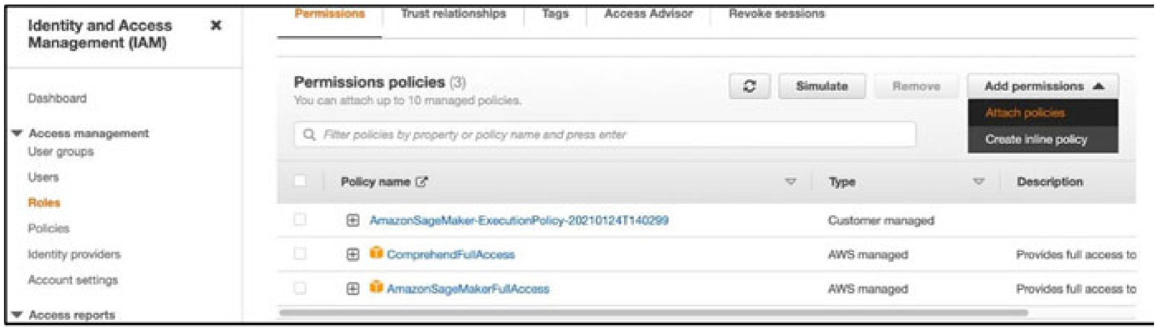
AmazonSageMaker-ExecutionRole-20210124T140299 — 1 match

Role name	Trusted entities	Last ac...
AmazonSageMaker-ExecutionRole-20210124T140299	AWS Service: sagemaker	28 minutes ag

Figure 9.4: *Lookup IAM role*

5. Click on the role name to select the role, and under **`Permissions policies`**, click on the **`Add permissions`** button and select **`Attach policies`**. Refer to

Figure 9.5:

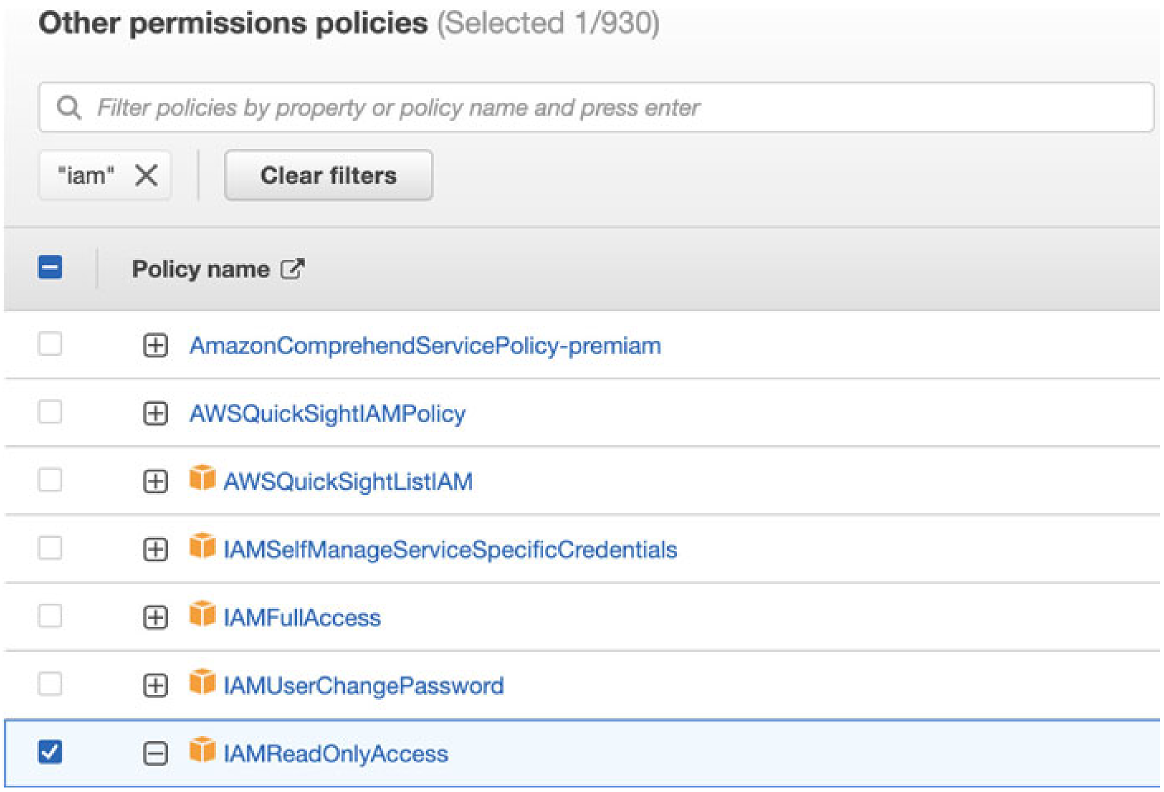


Permissions policies (3)
You can attach up to 10 managed policies.

Policy name	Type	Description
AmazonSageMaker-ExecutionPolicy-20210124T140299	Customer managed	
ComprehendFullAccess	AWS managed	Provides full access to
AmazonSageMakerFullAccess	AWS managed	Provides full access to

Figure 9.5: *Attach policies to role*

6. We need to add an IAM read only policy to our SageMaker role to allow us to use the execution role for some of our inference tasks. Type IAM in the policy search bar and select the **`IAMReadOnlyAccess`** policy. Click on **`Attach policies`** at the bottom of the screen, as shown in the following image:



Other permissions policies (Selected 1/930)

	Policy name
☐	AmazonComprehendServicePolicy-premiam
☐	AWSQuickSightIAMPolicy
☐	AWSQuickSightListIAM
☐	IAMSelfManageServiceSpecificCredentials
☐	IAMFullAccess
☐	IAMUserChangePassword
✓	IAMReadOnlyAccess

Figure 9.6: *Attach IAM Read Only policy*

Now, go back to your studio notebook and navigate to the left pane; click on the folder with the book name and then on the **`Chapter-09`** folder. Open the **`running-inference-sagemaker-hosting.ipynb`** Jupyter notebook by clicking on it. Before we can get to running inference, we need to pre-process our datasets and train a model. The initial steps are for model training, and then we will branch into separate sections to try each inference option. Execute the following instructions to continue with your build activity:

1. Execute the first two cells to upgrade the SageMaker packages and import **`boto3`**, **`numpy`** and the **`sagemaker`** libraries we need for running our code samples. Before you execute the second cell, provide an Amazon S3 bucket name (you must have created this based on instructions in the preceding paragraphs). You can leave the prefix as is. We are also defining the handle to the S3 service in this cell:

```
Import boto3
import io
import os
import time
import pandas as pd
import numpy as np
import sagemaker
import uuid
from sagemaker import image_uris
from sagemaker.inputs import TrainingInput
from sagemaker.estimator import Estimator
from sagemaker import get_execution_role
# Enter a bucket name - you can use the same bucket you used in
other chapters
bucket = '<specify-your-s3-bucket-name>'
prefix = 'aiml-book/chapter9'
s3 = boto3.client('s3')
```

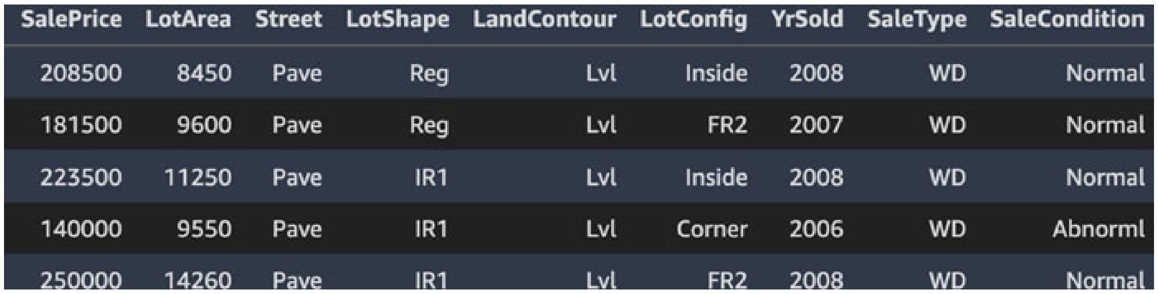
2. Now we will use the house prices dataset from Kaggle (**https://www.kaggle.com/c/house-prices-advanced-regression-techniques**), which is already provided as a CSV file in the GitHub repository and became available to you when you cloned the repository. Execute the next cell to load this CSV data into a Pandas DataFrame (**https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html**):

```
# load to dataframe
raw_df = pd.read_csv('kaggle-house-prices-dataset.csv',
header=0)
# drop columns containing null values
raw_df.dropna(axis=1, inplace=True)
raw_df.head()
```

3. In the following cell, we will create a low-dimensional dataset with and bring the **`SalePrice`** feature to the first column to act as the label/prediction feature:

```
# Create a low dimensional dataset with a few numeric and
categorical features for our example
small_df =
raw_df[['SalePrice','LotArea','Street','LotShape','LandContour'
,'LotConfig','YrSold','SaleType','SaleCondition']]
small_df.head()
```

4. We see the results as shown in *Figure 9.7*:



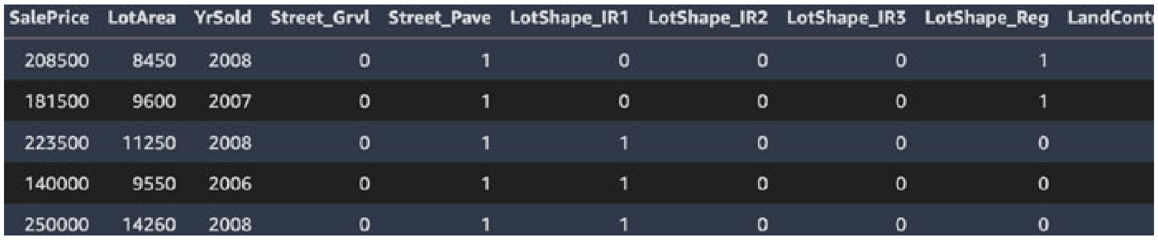
SalePrice	LotArea	Street	LotShape	LandContour	LotConfig	YrSold	SaleType	SaleCondition
208500	8450	Pave	Reg	Lvl	Inside	2008	WD	Normal
181500	9600	Pave	Reg	Lvl	FR2	2007	WD	Normal
223500	11250	Pave	IR1	Lvl	Inside	2008	WD	Normal
140000	9550	Pave	IR1	Lvl	Corner	2006	WD	Abnorml
250000	14260	Pave	IR1	Lvl	FR2	2008	WD	Normal

Figure 9.7: *Raw dataset*

5. Execute the next cell to perform encoding of our categorical features for all features to be of type numeric. We will use the Pandas **`get_dummies`** method to automatically one hot encode categorical features:

```
# perform numerical encoding of categorical features
encoded_df = pd.get_dummies(small_df)
encoded_df.head()
```

6. We will see the results shown in *Figure 9.8*:



SalePrice	LotArea	YrSold	Street_Grvl	Street_Pave	LotShape_IR1	LotShape_IR2	LotShape_IR3	LotShape_Reg	LandCont
208500	8450	2008	0	1	0	0	0	1	
181500	9600	2007	0	1	0	0	0	1	
223500	11250	2008	0	1	1	0	0	0	
140000	9550	2006	0	1	1	0	0	0	
250000	14260	2008	0	1	1	0	0	0	

Figure 9.8: *Encoded dataset*

7. Execute the following cell to split the dataset for training and testing and to remove the label feature from the test dataset:

```
# Select 90% as train data and 10% as test data
train_index=int(0.9 * len(encoded_df))
```

```
train_df = encoded_df.iloc[:train_index,:]
test_df = encoded_df.iloc[train_index:,:]
# remove the label feature from the test dataset
test_df_no_label = test_df.drop(['SalePrice'], axis=1)
print("Train dataset dimensions: " + str(train_df.shape))
print("Test dataset dimensions: " +
str(test_df_no_label.shape))
```

8. Now, create CSV files of the datasets and upload them to your Amazon S3 bucket:

```
# Create CSV files and upload to S3 bucket
train_df.to_csv('train.csv',index=False, header=False)
test_df_no_label.to_csv('test.csv',index=False, header=False)
s3.upload_file('train.csv',bucket,prefix+'/train/train.csv')
s3.upload_file('test.csv',bucket,prefix+'/test/test.csv')
```

9. Create a SageMaker Training Input object to abstract the train dataset location for use during model training:

```
# create a training input for SageMaker model training
train_input = TrainingInput('s3://{}/{}/{}/'.format(bucket,
prefix, 'train'), content_type='csv')
```

10. Now we are ready for model training. Execute the next cell to configure the hyperparameters for SageMaker XGBoost algorithm (**https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost.html**). SageMaker's XGBoost is a best-in-class adaptation of the popular boosted decision trees framework and is well suited for most regression and classification tasks:

```
# First let us define the hyperparameters
xgboost_hps = {
   "max_depth":"6",
   "eta":"0.3",
   "gamma":"2",
   "min_child_weight":"4",
   "subsample":"0.5",
   "objective":"reg:squarederror",
   "num_round":"30"
}
```

11. We are using SageMaker's built-in version of XGBoost, so all we need to do is refer to the container image uri at the time of training. Execute the next cell to load reference to this image in a variable:

```
# Next let us get the XGBoost built-in image
xgb_image = sagemaker.image_uris.retrieve(“xgboost”, ‘us-east-
1’, “1.5-1”)
```

12. Execute the next cell to define the output paths for the model artifacts:

```
# Define the output path to store model artifacts
model_prefix = prefix+’/xgboost-model’
model_output = ‘s3://{}/{}/output’.format(bucket, model_prefix)
```

13. Set up the XGBoost Estimator object that abstracts the heavy lifting in setting up training configuration, loading model artifacts, setting up containers, provisioning training infrastructure and more:

```
# Build XGBoost estimator
xgb_estimator = Estimator(image_uri=xgb_image,
hyperparameters=xgboost_hps,
        role=sagemaker.get_execution_role(),
        instance_count=1, instance_type=’ml.m5.2xlarge’,
        volume_size=10, # 10 GB
        output_path=model_output)
```

14. Execute the next cell to fit the estimator that will trigger the training job:

```
# Fit the estimator to run training
xgb_estimator.fit({‘train’:train_input})
```

15. We specified **`num_rounds`** as 30, indicating the number of epochs the model will train through our input training dataset. We will see the **Root Mean Squared Error** (**RMSE**), which a standard regression metric printed by the model for each round. The RMSE is an error metric, and it needs to reduce during each round, indicating that the model is learning well:

```
[24]#011train-rmse:49154.59375
[25]#011train-rmse:48825.31641
[26]#011train-rmse:48730.51953
[27]#011train-rmse:48351.53516
[28]#011train-rmse:47955.95703
[29]#011train-rmse:47543.60938
```

That brings us to the end of the training section of the notebook. Now we come to the main topic of this chapter: ML inference. We will now explore the different types of inference we can set up for our trained model, how to configure and build them, and how to run predictions using them.

Inference with real-time endpoints

In this section, we will build a single-model, single-container real-time endpoint, upload our trained model and run predictions using this endpoint. Execute the following instructions to proceed.

NOTE: You cannot directly execute this section; first, execute the model training section (Step 1 to Step 15) and then move on to the code here. If you try the following code directly, it will fail.

1. Execute the first cell in this section to print the location of where the trained model artifacts are stored in our Amazon S3 bucket:

    ```
    # Model artifacts are stored here
    print(xgb_estimator.model_data)
    ```

2. We see the following path printed:

    ```
    S3://<your-S3-bucket>/aiml-book/chapter9/xgboost-
    model/output/sagemaker-xgboost-2022-08-11-16-22-27-
    785/output/model.tar.gz
    ```

3. Execute the next cell to run the `CreateModel` API and build a SageMaker model from our trained model artifacts. Note that since we used SageMaker to train our model in the previous section, we can directly use the estimator object to deploy an endpoint. But in order to show you how you can use the `CreateModel`, `CreateEndpointConfiguration` and `CreateEndpoint` APIs, we take a low-level approach here. As you can see, we provide the XGBoost container image and the S3 URL for the model artifacts as input for this step:

    ```
    # get SageMaker boto3 handle
    sm = boto3.client('sagemaker')
    # our model name
    model_name = 'chapter9-xgboost-model-one-container'
    # now let us create a model based on the trained model
    artifacts
    model_res = sm.create_model(
          ModelName = model_name,
          ExecutionRoleArn = sagemaker.get_execution_role(),
          PrimaryContainer = {
            'Image': xgb_image,
            'ModelDataUrl': xgb_estimator.model_data,
         })
    ```

4. Execute the following cell to create a SageMaker Endpoint Configuration for our model deployment. This is a key step, and we provide the model

name, the endpoint infrastructure instance type and the instance count as inputs here. The `InitialInstanceCount` will ensure that there is at least as many instances always running as the count provided here:

```
Ep_config_name = model_name +'-epconfig'
epcfg_response = sm.create_endpoint_config(
EndpointConfigName=ep_config_name,
ProductionVariants=[{
    "VariantName": "chapter9-test-variant", # The name of the
    production variant.
    "ModelName": model_name,
    "InstanceType": 'ml.m5.xlarge', # Specify the compute
    instance type.
    "InitialInstanceCount": 1 # Number of instances to launch
    initially.
   }])
print("Endpoint Configuration successfully created: " +
epcfg_response['EndpointConfigArn'])
```

5. Execute the next cell to create our SageMaker real-time endpoint for our trained model:

```
Ep_name = model_name+'-ep'
ep_response = sm.create_endpoint(EndpointName=ep_name,
EndpointConfigName=ep_config_name
```

6. Endpoints typically take 15 to 20 minutes to be provisioned. Execute the following cell to check the Endpoint status. You can either refer to the Endpoint status in the SageMaker Consoler at **https://us-east-1.console.aws.amazon.com/sagemaker/home?region=us-east-1#/endpoints** (make sure you use the right region for your AWS console), or you can keep executing this following cell until the status changes to `InService`. Go to the next cell only after the status printed in this cell changes to `InService`:

```
# Wait until the print statement here shows InService
print(sm.describe_endpoint(EndpointName=ep_name)
['EndpointStatus'])
```

7. Now we will test both the endpoint and the model by invoking the endpoint with entries from our test dataset (without labels). Execute the following cell to create a `StringIO` buffer of our test entries from the CSV file:

```
# create a buffer for the csv request data from our test
dataset
```

```
from io import StringIO
inf_req = StringIO()
test_df_no_label.to_csv(inf_req,header=False, index=False)
```

8. Execute the following cell to invoke the endpoint to send input features from the test dataset to your model and print the results from our model:

```
# we need a runtime handler for SageMaker
sm_run = boto3.client("sagemaker-runtime")
# now call the endpoint
predictions = sm_run.invoke_endpoint(
        EndpointName=ep_name,
        Body=inf_req.getvalue(), # the values from the StringIO
        buffer we created in the previous cell
        ContentType='text/csv'
        )
#check if we getproper response - the predicted sale prices
print(predictions['Body'].read().decode('utf-8'))
```

9. We see the following results (only a snapshot of the results is reproduced here for brevity). These are predicted Sale Prices for houses from the test dataset we kept aside initially:

```
159152.59375
211584.296875
205173.65625
173063.265625
198776.09375
```

That brings us to the end of this subsection on how to build and use real-time provisioned endpoints in SageMaker for our trained models. We provided the instance type and count, but if we need to scale up our instance count because we suddenly receive a lot of traffic, we need to explicitly define an application autoscaling policy and add our endpoint as a dimension to this policy. Further, charges accrue for the duration the endpoint is up and running, irrespective of whether or not it receives inference requests. SageMaker's Serverless inference option enable configuring and using endpoints without worrying about infrastructure scaling.

Inference with serverless endpoints

In this section, we will learn how to build, deploy and use serverless endpoints. We will use the same XGBoost model we trained earlier but deploy a serverless inference option instead of a provisioned endpoint. Execute the cells in this section based on the following instructions:

1. Execute the first cell in this section to create a SageMaker model from our model artifacts, but specify the Containers objects instead of the **`PrimaryContainer`** dictionary we used in the previous section:

   ```
   # get SageMaker boto3 handle
   sm = boto3.client('sagemaker')
   # our model name
   model_name_serverless = 'chapter9-xgboost-model-serverless'
   # now let us create a model based on the trained model
   artifacts. For serverless we will use the Containers list
   rather than the PrimaryContainer
   model_res_serverless = sm.create_model(
        ModelName = model_name_serverless,
        ExecutionRoleArn = sagemaker.get_execution_role(),
        Containers = [{
          'Image': xgb_image,
          'Mode': 'SingleModel',
          'ModelDataUrl': xgb_estimator.model_data,
        }])
   print(model_res_serverless['ModelArn'])
   ```

2. Next, create a serverless endpoint configuration for our model by executing the following cell:

   ```
   Ep_config_name_serverless = model_name_serverless +'-epconfig'
   epcfg_response_serverless = sm.create_endpoint_config(
   EndpointConfigName=ep_config_name_serverless,
   ProductionVariants=[{
       'VariantName': 'chapter9-serverless',
       'ModelName': model_name_serverless,
       'ServerlessConfig': {
        "MemorySizeInMB": 3072,
        "MaxConcurrency": 25
     }}])
   ```

3. Execute the following cell to create the serverless endpoint:

   ```
   # The name of the endpoint
   ep_name_serverless = model_name_serverless+'-ep'
   ep_response_serverless =
   sm.create_endpoint(EndpointName=ep_name_serverless,
   ```

```
EndpointConfigName=ep_config_name_se
```

4. Execute the next cell a few times to check whether the endpoint is `InService` before proceeding to use the endpoint for predictions. You can also check the status in the SageMaker console at **https://us-east-1.console.aws.amazon.com/sagemaker/home?region=us-east-1#/endpoints**.

```
# Wait until the print statement here shows InService - should
take 3 to 5 mins
print(sm.describe_endpoint(EndpointName=ep_name_serverless)
['EndpointStatus'])
```

5. Create a `StringIO` buffer from the CSV file:

```
# create a buffer for the csv request data from our test
dataset
from io import StringIO
inf_req_svl = StringIO()
test_df_no_label.to_csv(inf_req_svl,header=False, index=False)
```

6. Finally, invoke the endpoint to run the predictions from our test entries:

```
# we need a runtime handler for SageMaker
sm_run = boto3.client("sagemaker-runtime")
# now call the endpoint
serverless_predictions = sm_run.invoke_endpoint(
        EndpointName=ep_name,
        Body=inf_req_svl.getvalue(), # the values from the
        StringIO buffer we created in the previous cell
        ContentType='text/csv'
        )
#check if we getproper response - the predicted sale prices
print(serverless_predictions['Body'].read().decode('utf-8'))
```

7. We get the following results printed. These are the Sale Prices predicted for housing inputs from the test dataset (only a few entries have been reproduced here for brevity).

```
159152.59375
211584.296875
205173.65625
173063.265625
198776.09375
176141.71875
```

That brings us to the end of this section. In the next section, we will learn how to perform batch inference with SageMaker hosting.

Inference with Batch Transform

When the need is to run inference for large datasets without the need for a low-latency response, for example, analyse call records and classify calls for analytics, you can use the SageMaker CreateTransformJob API to run a Batch Transform (**https://docs.aws.amazon.com/sagemaker/latest/dg/batch-transform.html**). Follow the instructions given here to execute the cells in this section to build and run a batch transform job for our XGBoost model:

1. First, let us define the S3 locations for our test dataset and output for our batch job; execute the first cell in this section:

```
# S3 location for our test dataset
s3_test = 's3://{}/{}/{}'.format(bucket, prefix,
'test/test.csv')
s3_batch_out = 's3://{}/{}/{}'.format(bucket, prefix,
'batch/output')
```

2. In the next few cells, we will build three dictionary objects: one for inputs to the batch transform job, one for storing the transform output configuration, and one for provisioning compute resources for the transform job. Execute the next cell to define the inputs to the job:

```
# input details for the Batch Transform
transform_input = {
  'DataSource': {
    'S3DataSource': {
      'S3DataType':'S3Prefix',
      'S3Uri':s3_test
    }},
'ContentType':'text/csv',
'SplitType':'Line'
}
```

3. Execute the next cell to define the output configuration for the batch transform:

```
# location for storing batch outputs
transform_output = {
'S3OutputPath':s3_batch_out,
'AssembleWith':'Line'
}
```

4. Execute the next cell to define compute resources:

```
# configure compute for the batch transform
transform_resources = {
'InstanceType':'ml.m5.2xlarge',
'InstanceCount': 1
}
```

5. Now, execute the following cell to start our batch transform job after providing additional inputs like a job name, the name of our XGBoost model, a batch strategy, which is **`MultiRecord`** in our case, using mini batches, with the **`SplitType`** as Line (we specified this in the transform input configuration cell earlier):

```
# run the batch transform job
batch_job_name = 'chapter9-batch-inference'
batch_res = sm.create_transform_job(
     TransformJobName=batch_job_name,
     ModelName=model_name,
     MaxPayloadInMB=1,
     BatchStrategy='MultiRecord',
     TransformInput=transform_input,
     TransformOutput=transform_output,
     TransformResources=transform_resources)
```

6. Once the batch job is complete, we will extract the outputs to review the inference results from the job. Execute the next cell to review the results:

```
# Read and print the outputs from the batch transform job
out_file = 'test.csv.out'
s3.download_file(bucket, prefix+'/batch/output/'+out_file,
out_file)
out_df = pd.read_csv(out_file,header=None)
out_df.head()
```

7. We get the following results printed. These are the predicted Sales Price for the houses:

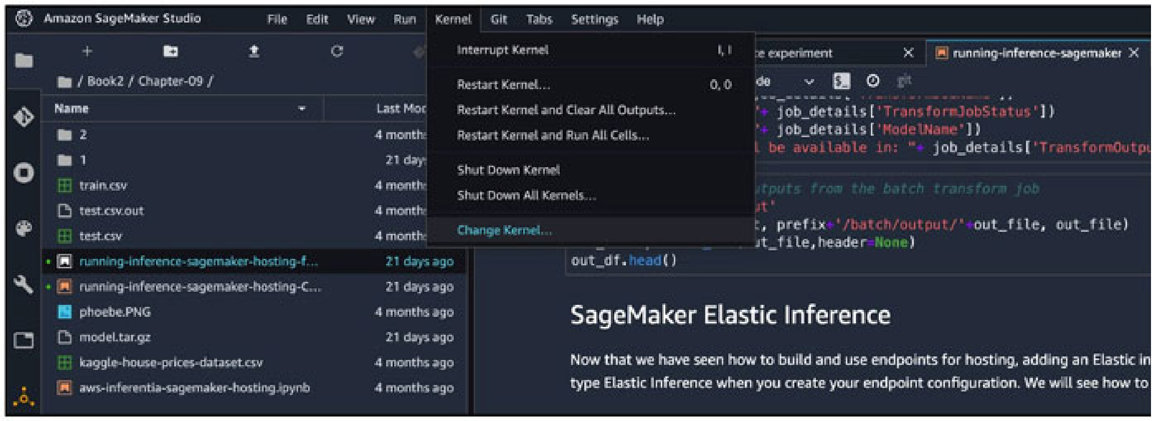
```
0
0 159152.593750
1 211584.296875
2 205173.656250
3 173063.265625
4 198776.093750
```

And that is how you set up batch inference for your trained models in SageMaker. In the next section, we will learn how to accelerate our inference performance by adding GPU burst capacity to a CPU instance using SageMaker Elastic Inference (**https://docs.aws.amazon.com/elastic-inference/latest/developerguide/what-is-ei.html**).

Adding a SageMaker Elastic Inference (EI) accelerator

When you use provisioned endpoints with compute that you specify, you get charged for the duration those instances are up and running. Suppose you have a high throughput real-time inference need but the traffic is unpredictable, provisioning a GPU instance to meet the throughput needs may work, but it is not cost-effective. With **Elastic Inference (EI)**, you configure a CPU instance endpoint, and whenever you need the burst GPU capacity for traffic spikes, EI will automatically supply that additional capacity. In this section of the notebook, we will learn how to add EI to your real-time endpoint to ensure GPU capacity when you need it for your inference traffic. Note that EI works only for computer vision ML domain, and the models have to be built using one of the standard ML frameworks, like Tensorflow, PyTorch, or MXNet. For our example, we will use the pre-trained ResNet50 (**https://www.tensorflow.org/versions/r2.6/api_docs/python/tf/keras/applications/resnet50/ResNet50**) image classification model from Tensorflow, and we will use Keras to load and deploy the model. Execute the following instructions to try out EI:

1. For this section, we need to use a different Kernel to import our pre-trained TensorFlow model. From your SageMaker Studio notebook, click on `Kernel` in the top menu, and then click on `Change Kernel`:



/ Book2 / Chapter-09 /

Name	Last Moc
2	4 month
1	21 day
train.csv	4 month
test.csv.out	4 month
test.csv	4 month
running-inference-sagemaker-hosting-f...	21 days ago
running-inference-sagemaker-hosting-C...	21 days ago
phoebe.PNG	4 months ago
model.tar.gz	21 days ago
kaggle-house-prices-dataset.csv	4 months ago
aws-inferentia-sagemaker-hosting.ipynb	4 months ago

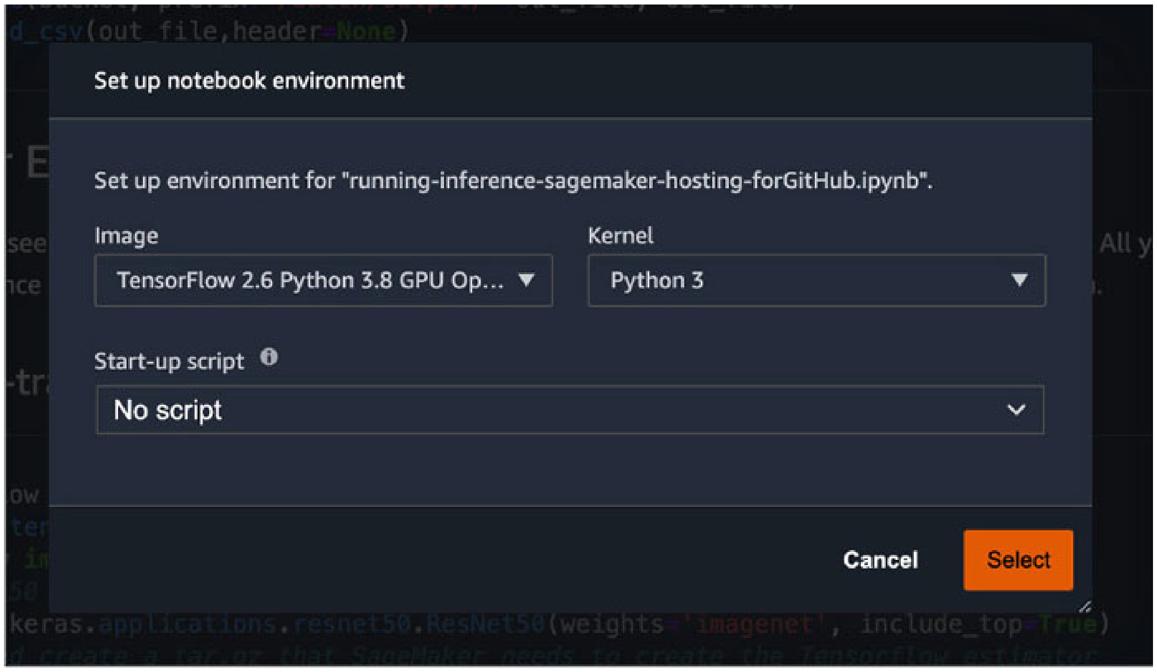
```
'+ job_details['TransformJobStatus'])
'+ job_details['ModelName'])
l be available in: "+ job_details['TransformOutpu

tputs from the batch transform job
ut'
t, prefix+'/batch/output/'+out_file, out_file)
ut_file,header=None)
out_df.head()
```

SageMaker Elastic Inference

Now that we have seen how to build and use endpoints for hosting, adding an Elastic in
type Elastic Inference when you create your endpoint configuration. We will see how to

Figure 9.8a: Change Kernel

2. In the popup that appears for Image, select TensorFlow 2.6 Python 3.8 GPU optimized, as shown in the following figure, and click on the `Select` button:



Set up notebook environment

Set up environment for "running-inference-sagemaker-hosting-forGitHub.ipynb".

Image: TensorFlow 2.6 Python 3.8 GPU Op...

Kernel: Python 3

Start-up script: No script

Figure 9.8b: Update the Kernel Image

3. Now, import the pre-trained model from the Tensorflow hub and save the model to a local directory. The directory name must be numeric to ensure that the save model action can associate a version to the model. Finally, create a tar file of the saved model artifacts so that we can use it for deployment. Execute the first cell to perform these tasks:

```
Import tarfile
import tensorflow as tf
from sagemaker.tensorflow import TensorFlowModel
from tensorflow import keras
# import resnet50
resnet_model =
keras.applications.resnet50.ResNet50(weights='imagenet',
include_top=True)
# save model and create a tar.gz that SageMaker needs to create
the Tensorflow estimator
m_dir = '1'
tf.saved_model.save(resnet_model,m_dir)
# open a tar file and save model contents
```

```
with tarfile.open('model.tar.gz','w:gz') as entry:
entry.add(m_dir)
```

4. Execute the next cell to load the model artifacts to your S3 bucket and create a **`TensorFlowModel`** object from the artifacts:

```
Tf_path = 'tensorflow/model/model.tar.gz'
tf_s3_path = 's3://{}/{}/{}'.format(bucket,prefix,tf_path)
s3.upload_file('model.tar.gz',bucket,prefix+'/'+tf_path)
# Create a Tensorflow estimator reference from the model
tf_model = TensorFlowModel(model_data=tf_s3_path,
framework_version='2.3', role=sagemaker.get_execution_role())
```

5. Execute the following cell to use the **`TensorFlowModel`** object to make a deploy call providing the compute instance type and count to create a real-time endpoint. What helps in attaching an EI instance to the deploy call is the **`accelerator_type`** parameter. Specify an EI instance type here, such as **`ml.eia2.medium`**. For a list of all available EI instances, you can visit **https://aws.amazon.com/machine-learning/elastic-inference/pricing/**.

```
# To deploy it to a SageMaker endpoint with an Elastic
Inference accelerator attached we simply pass this to the
deploy method
tf_endpoint = tf_model.deploy(instance_type='ml.m5.xlarge',
initial_instance_count=1, accelerator_type="ml.eia2.medium")
```

6. In the following cell, we will take an image of a cat and run predictions against our model to check whether it classified the image correctly. Execute the cell to pre-process the input image:

```
Import PIL
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
import matplotlib.pyplot as plt
cat_pic = load_img('phoebe.PNG', target_size=(224,224))
plt.imshow(cat_pic)
plt.show()
```

7. We get the following output image. It looks skewed because we adjusted the dimensions of the image to align with the model's input feature dimension.

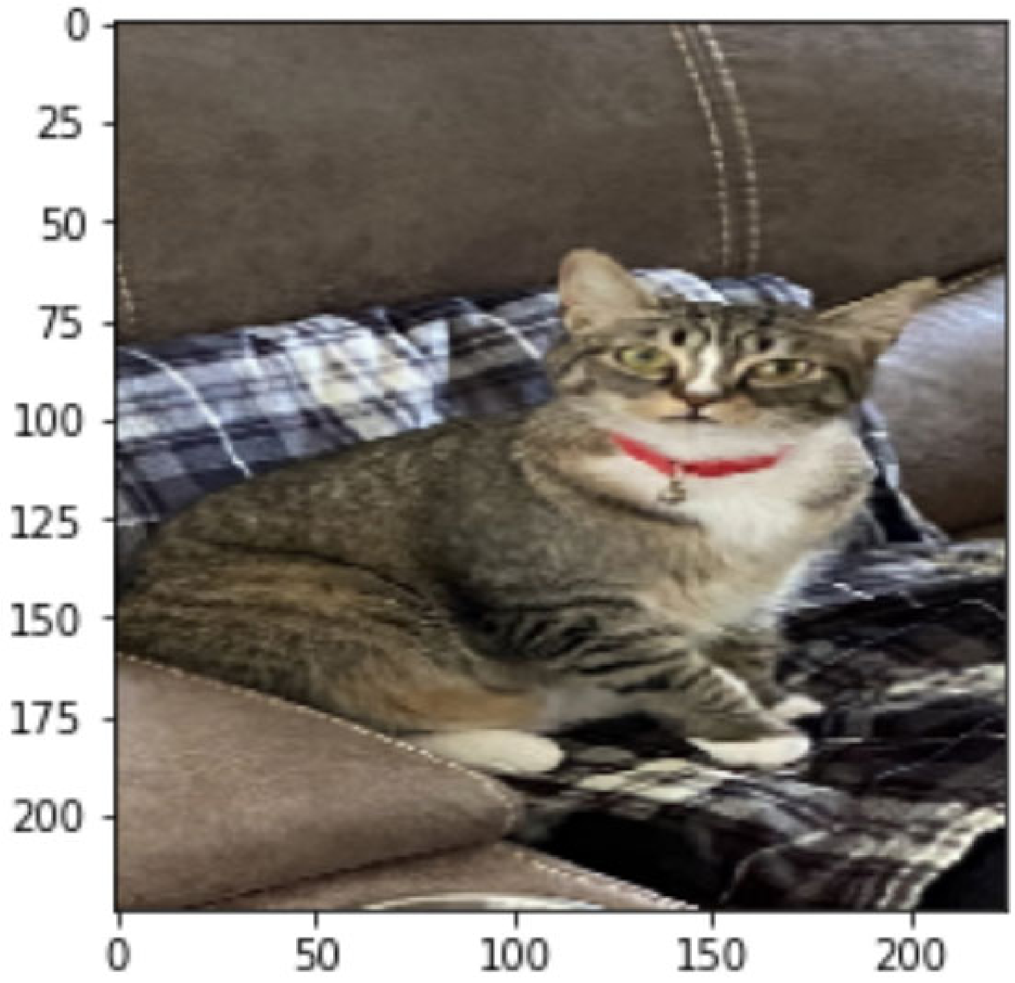


Figure 9.9: *Pre-processed image of my son's cat Phoebe*

8. Execute the next cell to get a numpy array of our image and expand the dimensions to get a batch version with a shape that matches the model's requirements:

```
np_arr = img_to_array(cat_pic)
arr_bat = np.expand_dims(np_arr, axis=0)
arr_bat.shape
```

9. We get the shape of the array for the image, as follows:

```
(1, 224, 224, 3)
```

10. In the following cell, we will use the **`preprocess_input`** and **`decode_predictions`** utility classes from Tensorflow for our network to improve prediction performance and convert model prediction outputs to a list of classes with maximum confidence rather than an array. Execute the cell and review the results:

```
# preprocess the image for prediction
b4_pred_img =
keras.applications.resnet50.preprocess_input(arr_bat.copy())
# make predictions and decode the output to a class
results = tf_endpoint.predict({"inputs": b4_pred_img.tolist()})
# convert to numpy array
new_res = np.array(results['outputs'])
# Get class predictions for the picture
print(keras.applications.imagenet_utils.decode_predictions(new_
res))
```

11. Review the printed results containing predictions from the ResNet50 model for our input cat image. As you can see, the model predicted the most probable breed for the cat quite correctly:

```
[[('n02124075', 'Egyptian_cat', 0.506083429), ('n02123045',
'tabby', 0.191357881), ('n02123159', 'tiger_cat', 0.1097463),
('n02091467', 'Norwegian_elkhound', 0.0174919851),
('n03594734', 'jean', 0.0171929821)]]
```

12. Navigate to the Amazon SageMaker console (**https://us-east-1.console.aws.amazon.com/sagemaker/home?region=us-east-1#/endpoints**) and delete the endpoints you created in this chapter if they are no longer used so that you do not incur costs. Make sure you select the correct region based on the AWS region you are in.

And that brings us to the end of this section and chapter. We hope you enjoyed trying out the different ways to host your models for inference with Amazon SageMaker. The options are built to cater to massive scale based on your enterprise needs and diverse ML use cases.

Conclusion

Congratulations! Pat yourself on the back, because you have now learned comprehensive skills required to build and run ML models on AWS using Amazon SageMaker. Up until now, we covered the major stages of the ML workflow, such as data preparation, model training and hosting. Model monitoring and active learning are not in scope of this book; they may need a separate book. We will include a few articles on these topics for further reading. Remember to check out *Chapter 12, Operationalized Model Assembly*, to learn about MLOps, an essential part of an ML project for enterprise CI/CD.

In the next chapter, we will pivot from using SageMaker to AWS AI services, which are pre-trained models abstracted as APIs that can be directly used for common ML use cases, without the need for ML expertise. In *Chapter 10 – Adding Intelligence with Sensory Cognition* we will learn about AI services for sensory cognition, and in *Chapter 11 – AI for Industrial Automation*, we will learn about AWS services for industrial AI automation.

Points to Remember

In this chapter, we learned one of the key phases in the ML workflow, the primary reason why a model is trained: predictions or inference. We covered the following topics:

- We started by understanding the different configuration options for setting up inference with SageMaker hosting.
- We learned about hosting options like real-time inference, multi-model and multi-container inference, asynchronous inference, serverless inference, and batch transform.
- We then executed code samples on how to pre-process datasets, train a model, and create a model package for deployment.
- We then went hands-on and used the SageMaker boto3 Python SDK to prepare model artifacts for deployment, which included creating a model package, creating an endpoint configuration, and creating an endpoint that serves as an API for inference.
- For real-time inferencing, we saw how to create real-time and near-real-time or asynchronous endpoints using SageMaker Python SDK. We also tested the hosting options using the model we trained earlier in this chapter.
- We configured the SageMaker Serverless inference, a newly launched feature, and ran an inference test using the serverless endpoint.
- We learned how to execute a transform job for batch inference in SageMaker for our trained model and tested it with a dataset stored in Amazon S3.
- Finally, we learned how to set up and use Amazon Elastic Inference, an advanced accelerator that provides GPU burst capacity for provisioned CPU instances to accelerate prediction performance.

Multiple Choice Questions

Use these questions to challenge your knowledge of Amazon SageMaker hosting options for ML models.

1. **What are some of the hosting options that SageMaker supports? (Select all correct options.)**
   a. Real-time inference
   b. Batch transform
   c. Serverless inference
   d. Messaging and queuing inference
2. **What is the order in which you configure steps for model hosting using SageMaker Python SDK?**
   a. Create endpoint config, create model package, and then create endpoint
   b. Create endpoint, create model package, and then create endpoint config
   c. Create endpoint config, create endpoint, and then create model package
   d. Create model package, create endpoint config, and then create endpoint
3. **What is SageMaker Elastic Inference used for?**
   a. To autoscale SageMaker training job capacity
   b. To run batch inference
   c. For supervised learning
   d. To automatically add GPU burst capacity to a CPU inference instance to support high volume of requests
4. **You cannot host a ML model in SageMaker if it is not trained in SageMaker.**
   a. True
   b. False
5. **What is the maximum payload size for SageMaker Batch Transform?**
   a. 500 MB
   b. 100 MB
   c. 250 MB

d. 6 MB

Answers

1. **a, b, c**
2. **d**
3. **d**
4. **b**
5. **b**

Further Reading

- Model monitoring with Amazon SageMaker: **https://docs.aws.amazon.com/sagemaker/latest/dg/model-monitor.html**
- Active Learning of document processing workflows: **https://aws.amazon.com/blogs/machine-learning/active-learning-workflow-for-amazon-comprehend-custom-classification-part-2/**
- Active Learning of Computer Vision workflows: **https://aws.amazon.com/blogs/machine-learning/using-amazon-rekognition-custom-labels-and-amazon-a2i-for-detecting-pizza-slices-and-augmenting-predictions/**
- Detecting bias in ML models: **https://aws.amazon.com/blogs/machine-learning/learn-how-amazon-sagemaker-clarify-helps-detect-bias/**

CHAPTER 10

Adding Intelligence with Sensory Cognition

Introduction

If you are a fan of science fiction books, you must have read at least one Isaac Asimov book during your delightful sojourn. The author is a self-proclaimed fan of Mr. Asimov's authentic dystopian portrayal of a not-too-distant future in which mathematical statistics and historical information can be used to predict the evolutionary trajectory of humanity, a branch of fictional science that Asimov called Psychohistory. Now, where have we heard using mathematics, statistics and historical data to make predictions before? If you are still guessing, right here, in what we call AI/ML today. With Psychohistory, Asimov probably took the most logical artistic liberty on how to apply math and history to predict the future, but it indeed has an uncanny similarity to how we approach and use AI/ML. If you have not tried an Asimov before, the author recommends starting with Foundation (**https://www.amazon.com/Foundation-Isaac-Asimov/dp/0553293354**).
Another concept that that Asimov dealt with in his books and that is thrillingly similar to what the world is moving toward is a future where humans are totally reliant on robots (**https://www.amazon.com/Isaac-Asimov-Collection-Robots-Empire/dp/9124369896**). To establish order, Asimov uses his three laws of robotics (https://www.britannica.com/topic/Three-Laws-of-Robotics) as a guiding design principle that all robots are deigned to follow, and he discusses ethical, moral, and humanitarian conflicts that arise in detail in his stories.

In our current world, there are a lot of parallels to Asimov's fiction (it was written in the 1950s), such as robots for vacuuming, dishwashing, mopping, industrial robots that can make cars, agricultural robots than can perform farming, and software robots that can fly planes. It is just that Asimov imagined his robots to be of humanoid appearance, but the robots we use today are not. Asimov's robot series is a fascinating read for sure, but what does that have to do with this chapter? Not a lot, but there is relevance. Let us take Asimov's humanoid robot with a positronic brain (**https://www.litcharts.com/lit/i-robot/terms/positronic-brain**). It is, of course, a fictional concept created to imbue the story with logic, but what would it really take to build a robot that could see, listen, speak and understand. In other words, how can we add intelligence to a robot with sensory cognition capabilities?

In the previous chapters, we learned every step of the ML workflow with real-world scenarios and comprehensive coding samples, and we built our own ML models using Amazon SageMaker and other AWS services. Specifically, we learned how to perform data collection and feature engineering, how to automate data orchestration pipelines, how to select and optimize an algorithm or neural net for model train, how to train and tune models, how to use AutoML to save on ML project time/costs, how to deploy our ML models, and how to run inference/predictions off of our models. That pretty much covers most of the steps of a ML workflow; in this chapter, we will pivot to a different branch of AWS AI/ML stack: the AI services layer. Here, we will learn about key AWS AI services for speech, text and vision domains, the use cases they can be applied to, and how to build applications to leverage these AI services. We will, of course, come back to our robot example and understand what AI services can be used to build a cognitive robot.

Structure

In this chapter, we will dive deep into the following topics:

- Introducing AWS AI services
- Using Amazon Rekognition for computer vision
- Using Amazon Transcribe for speech recognition
- Using Amazon Translate for language translation
- Using Amazon Polly for speech generation
- Using Amazon Comprehend for deriving insights
- Adding sensory cognition to your applications

Objectives

In this chapter, we will first get an overview of AWS AI services that are pre-trained ML models available behind an API for a wide variety of common ML use cases and domains. We will then focus on AWS AI services that provide sensory cognition functions, such as speech understanding, text to speech, vision, translation and deriving insights from textual data. We will understand the benefits of these services and what use cases these can be applied for, and then we will learn how to use these services in our applications with code samples.

Introducing AWS AI services

As we discussed in *Chapter 1, Introducing the ML Workflow*, in the Introducing AI and ML on AWS section, AI services are powerful pre-trained models that have been laser tuned to solve for specific use cases or domains well. Think of these as some of the most common problems you can use AI/ML to solve; organizations of all sizes are using them today. While this means that the services are highly abstracted (you do not have access to the underlying models), it also means that you can take your use case to production that much faster. This is because the AI services provide access to running predictions with pre-trained models by means of an API call. This means you do not have to process data, train, tune or deploy models, so no ML skills are required. You just write a few lines of code to make an API call, send your inputs and receive your predictions. It is as simple as that. And we will cover how to build intelligent applications using AI services in the later sections of this chapter. In terms of how these AI services are categorized, the author likes to think that the services are organized to address requirements in three major areas of interest: sensory cognition, enterprise/industrial, and technology operations. This chapter will focus on the AI services providing sensory cognition capabilities. In *Chapter 11, AI for Industrial Automation*, we will discuss the other categories. To refresh our memory (because we did learn about this in *Chapter 1, Introducing the ML Workflow*), we will now review the groups of AI services that are part of the sensory cognition category, from our list of AWS AI services. Refer to the following table:

ML problem type	ML sub type	ML use case	AWS AI Service	Service overview
Sensory Cognition	Computer Vision	Image classification, Object detection	Rekognition	Pre-trained ML models available as an API for automating various video and image processing tasks to infer intelligent insights
	Speech	Speech to text	Transcribe	Converts pre-recorded and streaming conversations to text transcripts using pre-trained ML models available as an API
		Text to speech	Polly	Creates lifelike speech from text in various voices and inflections using pre-trained ML by means of an API
	Text	Sentiment detection, entity recognition, text classification	Comprehend	Natural language processing (NLP) service pre-trained for entity recognition, classification, sentiment, grammar and language detection with transfer learning capabilities available

		Machine translation	Translate	Pre-trained ML service that provides language translations capabilities using an API
		Text extraction	Textract	Intelligent text extraction from a variety of document types and images using pre-trained ML models
		Intelligent search	Kendra	Enterprise search service powered by NLP indexes using pre-trained ML models with natural language query capabilities
	Chatbots	Conversation al interfaces	Lex	Chatbot implementation using ML with easy and intuitive interface to build highly intelligent conversational interfaces

Table 10.1: *AWS AI services for sensory cognition*

As you can see, AWS provides pre-trained AI services for common use cases in the areas of computer vision, speech, text and chatbots. In this chapter, we will dive deep into the core sensory cognition services like Amazon Rekognition, Amazon Transcribe, Amazon Polly, Amazon Comprehend, and Amazon Translate. And you will be surprised to see how it relates to all the discussion we had about robots in the introduction to this chapter. Refer to the following picture and try to visualize it representing a robot's head:

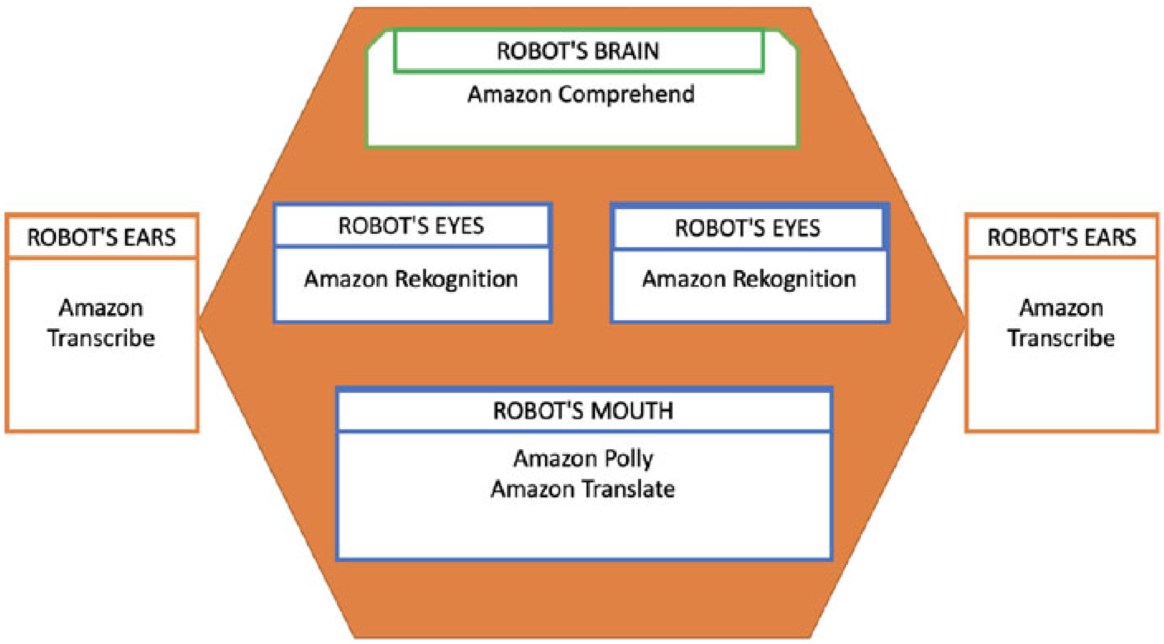
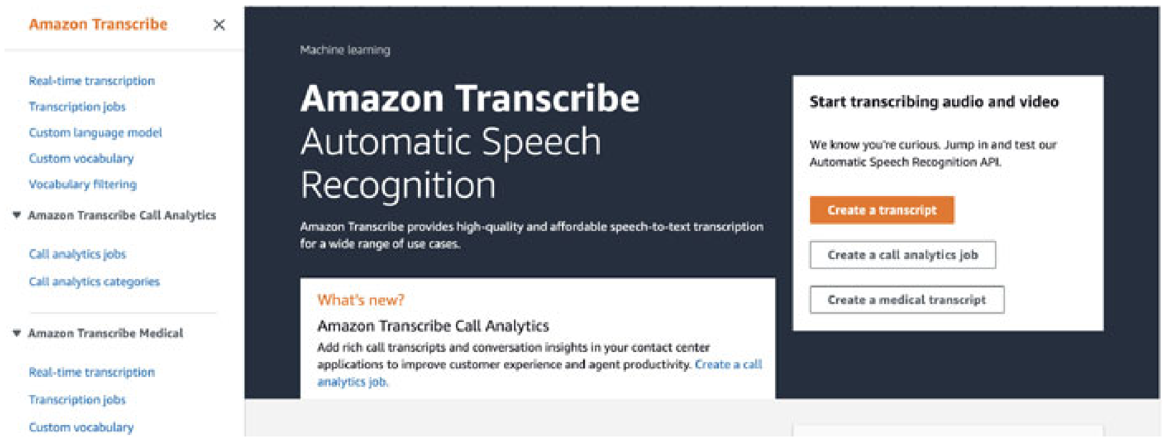


Figure 10.1: *AWS AI services for cognitive functions*

Amazon Transcribe for automatic speech recognition

The robotic ears should have the capability to detect voice in sound, and interpret and understand the meaning of the spoken words, also called as automatic speech recognition or ASR. Amazon Transcribe (**https://aws.amazon.com/transcribe/**) is a fully managed AI service with powerful pre-trained ASR models that can not only transcribe speech to text but can also help derive critical insights, supporting both real-time and batch transcriptions with the ability to customize transcription outputs using your own vocabulary or incrementally train ASR models unique to your business semantics using custom language models. Amazon Transcribe provides APIs (**https://docs.aws.amazon.com/transcribe/latest/APIReference/Welcome.html**) in different programming languages (for example, the Python SDK for Transcribe: **https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/transcribe.html**), using which you can easily transform your applications with powerful ASR capabilities. The following image shows the Amazon Transcribe console. We will use Amazon Transcribe APIs in the next section to learn how to build speech recognition.



Amazon Transcribe
Automatic Speech Recognition

Amazon Transcribe provides high-quality and affordable speech-to-text transcription for a wide range of use cases.

What's new?

Amazon Transcribe Call Analytics

Add rich call transcripts and conversation insights in your contact center applications to improve customer experience and agent productivity. Create a call analytics job.

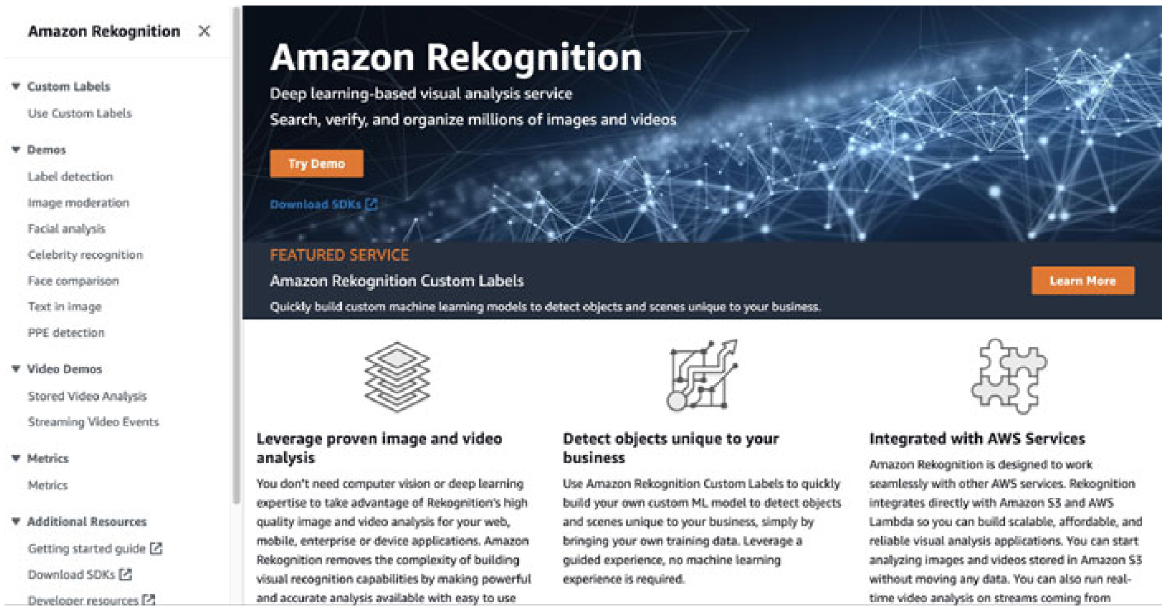
Start transcribing audio and video

We know you're curious. Jump in and test our Automatic Speech Recognition API.

Figure 10.2: *Amazon Transcribe Console*

Amazon Rekognition for computer vision

William Shakespeare said, "*Eyes are the window to your soul*", but he was, of course, not referring to the eyes of a robot, which are nothing but a set of curved glass plates that capture light from objects into an image or video, namely, the camera. More than just recording an image, our robot needs to be equipped with some form of intelligence that can detect and recognize images to be functionally useful. That way, our robot can take actions based on what it "sees" and react intelligently to its environment. And this is why, for our robot's eyes, we can use Amazon Rekognition (**https://us-east-1.console.aws.amazon.com/rekognition/home?region=us-east-1#/**), a fully managed computer vision AI service, with pre-trained models providing capabilities for analysing images and videos, detecting objects in images, facial feature comparison, content moderation and more. Due to the fact that these powerful computer vision models are available as API calls, you can make your applications vision AI capable with just a few lines of code, for a wide variety of common vision-based use cases, including detecting whether your employees are wearing **personal protective equipment** (**PPE**) as they walk into your facility. The following image shows the Amazon Rekognition console. We will use Amazon Rekognition APIs in the next section to learn how to build computer vision as shown in the following image:



Amazon Rekognition

Deep learning-based visual analysis service

Search, verify, and organize millions of images and videos

FEATURED SERVICE

Amazon Rekognition Custom Labels

Quickly build custom machine learning models to detect objects and scenes unique to your business.

Leverage proven image and video analysis

You don't need computer vision or deep learning expertise to take advantage of Rekognition's high quality image and video analysis for your web, mobile, enterprise or device applications. Amazon Rekognition removes the complexity of building visual recognition capabilities by making powerful and accurate analysis available with easy to use

Detect objects unique to your business

Use Amazon Rekognition Custom Labels to quickly build your own custom ML model to detect objects and scenes unique to your business, simply by bringing your own training data. Leverage a guided experience, no machine learning experience is required.

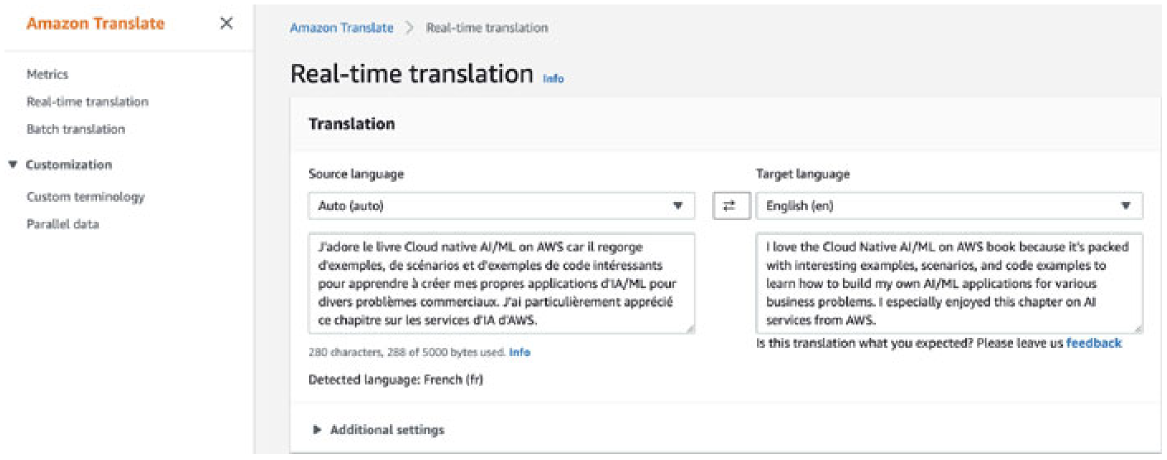
Integrated with AWS Services

Amazon Rekognition is designed to work seamlessly with other AWS services. Rekognition integrates directly with Amazon S3 and AWS Lambda so you can build scalable, affordable, and reliable visual analysis applications. You can start analyzing images and videos stored in Amazon S3 without moving any data. You can also run real-time video analysis on streams coming from

Figure 10.3: *Amazon Rekognition console*

Amazon Translate for machine translation

Father Thomas Keating said "God's first language is silence. Everything else is a translation". To find peace, silence might be sufficient, but to live and interact with the world, one needs language. Due to the evolution of multiple civilizations across the world, we now have numerous languages, each with its own cultural significance, societal embellishment, grammatical structure and inflection. With as many as 7000 spoken languages in the world today, it is indeed a herculean task, time intensive and highly cost prohibitive to manually translate languages with a high degree of accuracy. So, if you build a robot that knows English well but sell or deploy the robot in France or Germany, it will not be of much use. This is where you can use a machine translation service like Amazon Translate (**https://aws.amazon.com/translate/**), a fully managed AWS AI service for neural machine translation supporting 75 languages (with more being added) without the need for any ML training or expertise, by making a simple API call. You can also customize translations using parallel data files (**https://docs.aws.amazon.com/translate/latest/dg/customizing-translations-parallel-data.html**) or custom terminology (**https://docs.aws.amazon.com/translate/latest/dg/how-custom-terminology.html**) to introduce your own unique style. Amazon Translate can automatically detect the source language using another AWS AI service called Amazon Comprehend (the *brains* of our robot), which we will cover in a subsequent section. So, imagine we powered up your robot with Amazon Translate, enabling it to listen and respond to you in the language of your choice, wouldn't it be cool? You can also configure your robot to allow listening in one language and responding in a different language. In fact, using Amazon Transcribe and Amazon Translate, you can create a universal translator referred to in Star Trek (**https://memory-alpha.fandom.com/wiki/Universal_translator**) but for earth-based languages. That said, the sky's the limit on what you can do with Translate. The following image, *Figure 10.4*, shows the Amazon Translate console you can use to try real-time translations. In the next section, we will use Translate APIs to learn how to add machine translation to applications:



Real-time translation

Translation

Source language: Auto (auto)

J'adore le livre Cloud native AI/ML on AWS car il regorge d'exemples, de scénarios et d'exemples de code intéressants pour apprendre à créer mes propres applications d'IA/ML pour divers problèmes commerciaux. J'ai particulièrement apprécié ce chapitre sur les services d'IA d'AWS.

280 characters, 288 of 5000 bytes used. Info

Detected language: French (fr)

Target language: English (en)

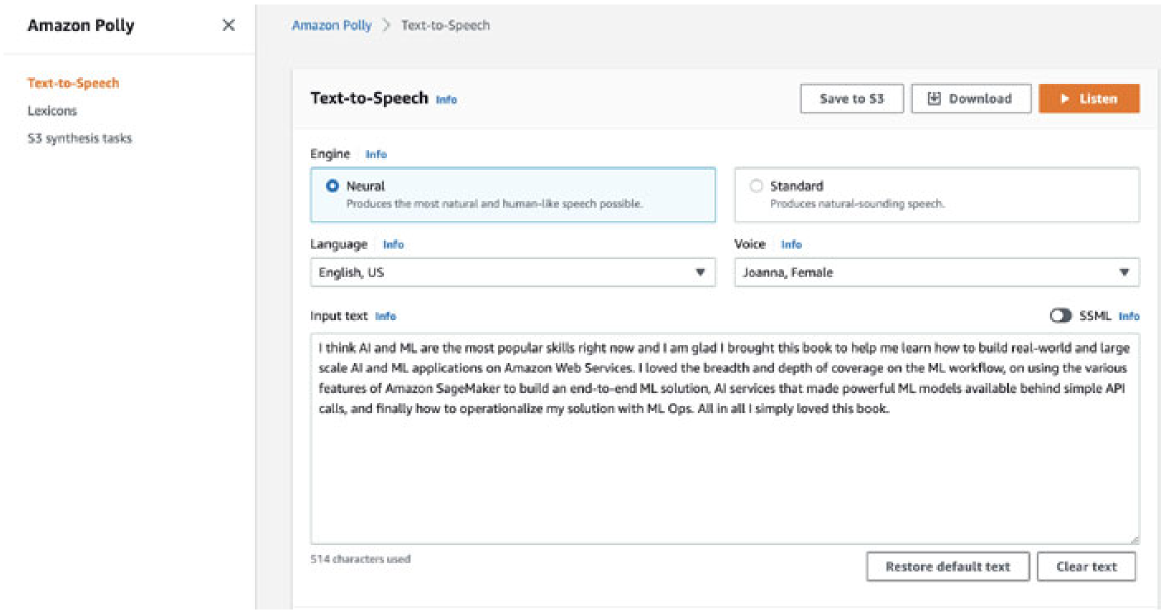
I love the Cloud Native AI/ML on AWS book because it's packed with interesting examples, scenarios, and code examples to learn how to build my own AI/ML applications for various business problems. I especially enjoyed this chapter on AI services from AWS.

Is this translation what you expected? Please leave us feedback

Figure 10.4: *Amazon Translate in action*

Amazon Polly for text to speech

By now, we have a robot that can hear you (using Amazon Transcribe), see you (with Amazon Rekognition), and translate what you spoke to other languages, but it cannot yet speak back to you. We can install an audio player and speaker in the robot, but we need to still find a way to synthesize text into an audio stream for the robot to respond to you in speech. Amazon Polly (**https://aws.amazon.com/polly/**) is a text to speech AI service providing capabilities to add speech to your applications with just an API call, without the need to train or manage ML models. Polly supports 33 languages, and 70+ voices as of July 2022 and continues to add more languages and voices. With Polly, you can work with the AWS team to create a custom voice for your brand, select from a choice of three different speaking styles (standard, neural or newscaster) and deliver real-life speaking experiences with your applications. So, Amazon Polly is a good choice for us to speech enable our robot, and the good news is that we have a wide variety of languages and both male and female voices to choose from. In the next section, we will use the Polly API to demonstrate how to build applications that use speech for response. The following image shows the Amazon Polly console with the option to test voices in real time:



Engine: Neural

Language: English, US

Voice: Joanna, Female

I think AI and ML are the most popular skills right now and I am glad I brought this book to help me learn how to build real-world and large scale AI and ML applications on Amazon Web Services. I loved the breadth and depth of coverage on the ML workflow, on using the various features of Amazon SageMaker to build an end-to-end ML solution, AI services that made powerful ML models available behind simple API calls, and finally how to operationalize my solution with ML Ops. All in all I simply loved this book.

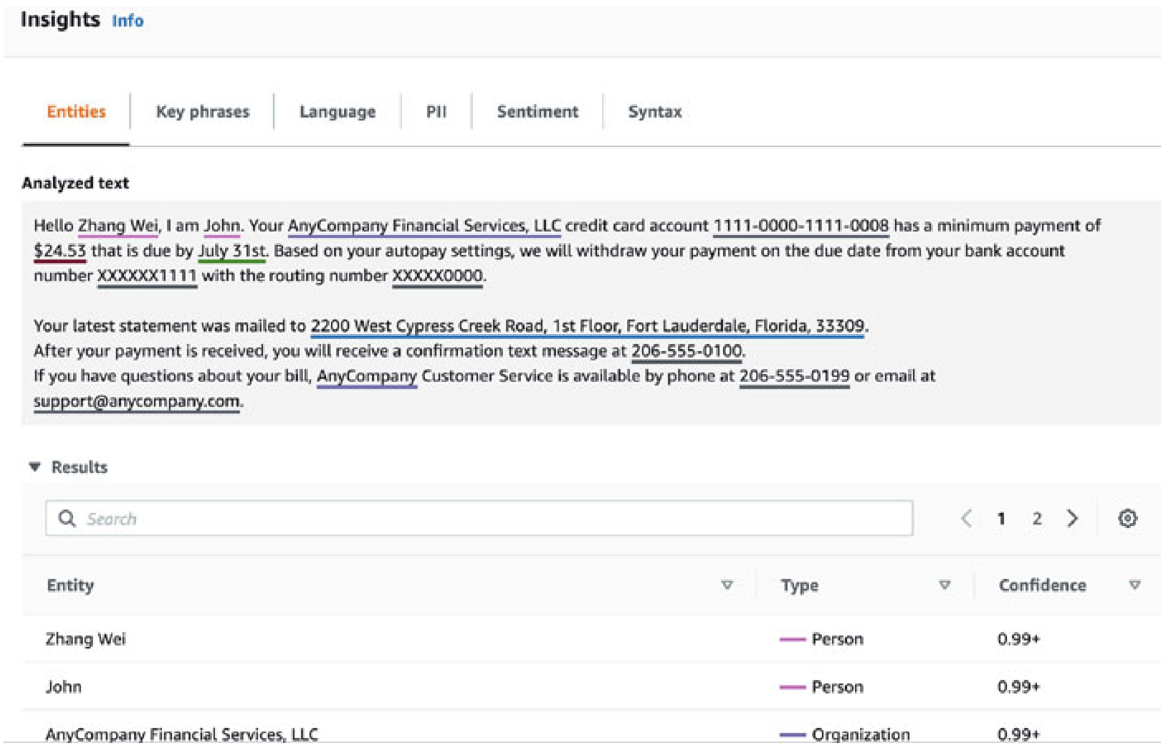
514 characters used

Figure 10.5: *Amazon Polly console*

Amazon Comprehend for deriving insights

While the sense organs provide inputs and respond to stimuli, the cognition aspect comes from the brain in the human body. However, in the case of our AI services, each service has a sensory component and a cognitive component combined to provide the capabilities they are designed for. For example, in Amazon Rekognition, you can think of the sensory component to be the layer that processes the input images, and the cognitive component to be the layer that derives meaning in the form of labels from the image. In the case of Amazon

Comprehend though, the service is tailored to uncover insights and patterns in text, so it is purely for comprehension or cognition without the sensory aspect. And that is why we chose Amazon Comprehend for the brains of our robot. With features to detect named entities from text, ascertain key phrases, detect sentiment, understand the grammatical syntax of text, detect dominant language, determine and obfuscate personally identifiable information from text, and much more, Comprehend is a core NLP service with rich features available behind a simple API call, without the need to train or tune ML models. In the next section, we will look at how to use Amazon Comprehend APIs to build NLP applications, but for now, take a look at the following image for a view of the insights generated by Comprehend in real time:



Analyzed text

Hello Zhang Wei, I am John. Your AnyCompany Financial Services, LLC credit card account 1111-0000-1111-0008 has a minimum payment of $24.53 that is due by July 31st. Based on your autopay settings, we will withdraw your payment on the due date from your bank account number XXXXXX1111 with the routing number XXXXX0000.

Your latest statement was mailed to 2200 West Cypress Creek Road, 1st Floor, Fort Lauderdale, Florida, 33309.
After your payment is received, you will receive a confirmation text message at 206-555-0100.
If you have questions about your bill, AnyCompany Customer Service is available by phone at 206-555-0199 or email at support@anycompany.com.

Results

Entity	Type	Confidence
Zhang Wei	Person	0.99+
John	Person	0.99+
AnyCompany Financial Services, LLC	Organization	0.99+

Figure 10.6: *Amazon Comprehend real-time insights*

Alright, it is time to fasten our seatbelts and start executing our code samples to learn by doing. In this section, we reviewed the AWS AI services and walked through the key AI services that simulate sensory cognitive functions by visualizing an example of what we need to design a robot. In the next section, we will run through code samples of using AWS AI service APIs for a variety of ML tasks, all without requiring any ML training or development.

Adding sensory cognition to your applications

In the previous section, we discussed how to add cognitive capabilities using AWS AI services for our robot example and what service we need to use for which specific function. We also briefly discussed some of the features that these services provide. We had a glimpse of what the AWS console looks like for these services. In this section, we will change gears and follow a programmatic approach to working with the APIs of these AI services; we will also learn how to use them to transform enterprise applications and inspire endless innovation. Before we get started though, we need to ensure that our pre-requisites are in place.

NOTE: If you have already signed in to your AWS console, created an Amazon S3 bucket, on-boarded to Amazon SageMaker Studio and cloned the book's GitHub repository, you can skip the following paragraph and go to the sub-section titled *Setting up IAM permissions*. If not, follow the instructions from the following paragraph onward.

In case you still haven't signed up for an AWS account, now is the time to do so. Follow the instructions in the Setting up your AWS account section of *Chapter 2, Hydrating Your Data Lake*, to sign up for an AWS account. Once you have signed up, log in to your AWS account using the instructions at **https://docs.aws.amazon.com/IAM/latest/UserGuide/console.html**. If you have not already done so, create an Amazon S3 bucket (**https://docs.aws.amazon.com/AmazonS3/latest/userguide/create-bucket-overview.html**). Next, follow the instructions in the *Technical Requirements* section of *Chapter 3, Predicting the Future With Features*, to onboard to an Amazon SageMaker Studio domain, and clone the book's GitHub repository using the link provided at the beginning of this book. We are interested in the *Chapter 10, Adding Intelligence With Sensory Cognition*, folder, but before we open our notebook, we need to set up IAM permissions for our SageMaker Studio role to allow working with our AWS AI services in this chapter.

Setting up IAM permissions

Our SageMaker Studio notebook comes with default IAM permissions for working with SageMaker features and access to S3 buckets. But it does not include access to the AI services we will use in this chapter. We will have to attach additional policies to our role to provide it with the necessary permissions. Execute the following instructions to augment your SageMaker role:

1. If not already done, log in to AWS Management Console, type `SageMaker` in the services search bar at the top, and navigate to the SageMaker console.

Under **`Control Panel`** in the left menu, click on Studio. Then, click on the **`Launch SageMaker Studio`** button on the right. If you have already onboarded to SageMaker Studio, you will see the username you created appearing under the Users section. If you have not onboarded to SageMaker Studio, you can do so by following the instructions in the *Technical Requirements* section in *Chapter 3, Predicting the Future With Features* to onboard to an Amazon SageMaker Studio domain and clone the book's GitHub repository (**https://github.com/garchangel/AIMLwithAWS**).

2. In the **`Control Panel`** page, copy the Execution role ARN that is displayed in the **`Domain`** pane on the right of the page (by clicking on the two squares on the right of the role), as shown in the following image:

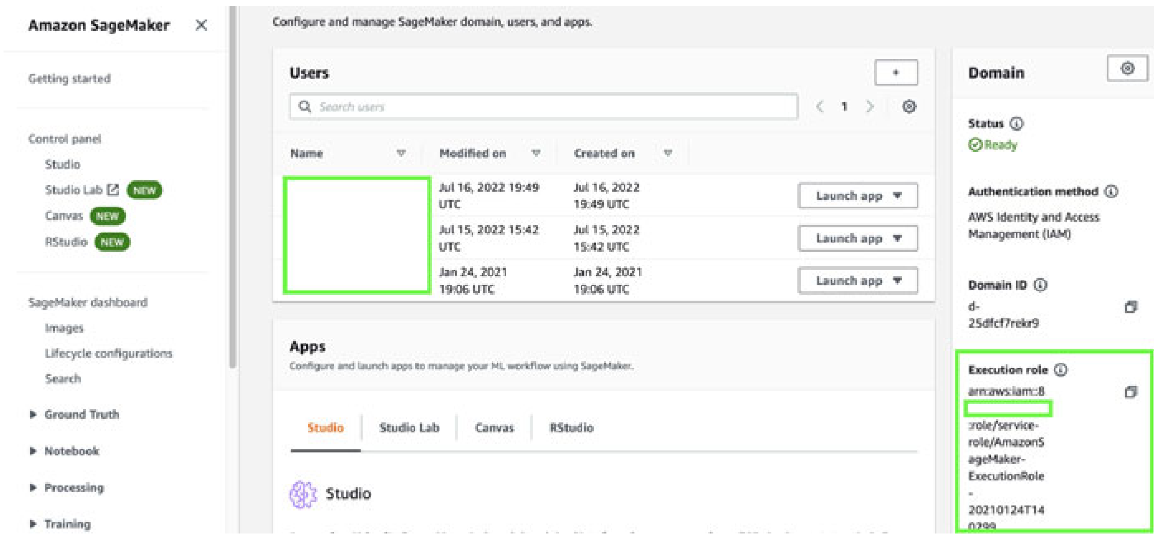


Figure 10.7: Copy Studio execution role ARN

3. Now, type IAM in the services search bar and navigate to the IAM console, as shown in the following image:

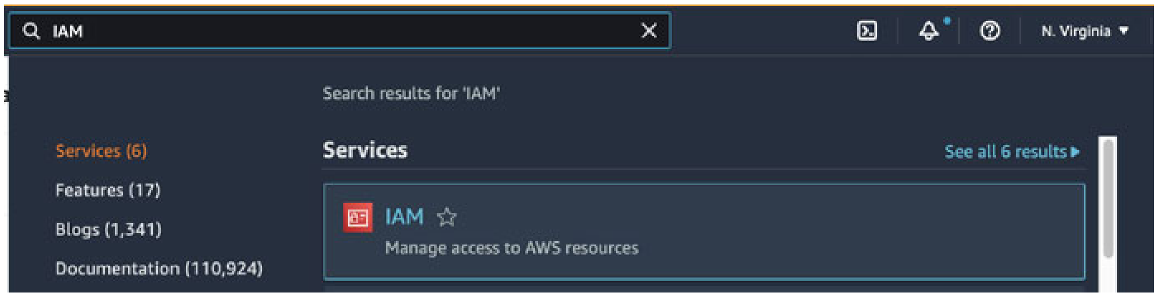
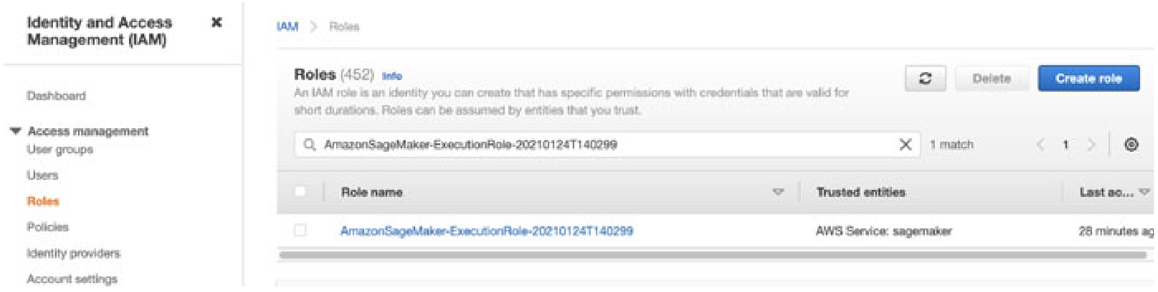


Figure 10.8: Search for IAM service

4. In the IAM console, select **`Roles`** from the left menu and type only the role name from the ARN you copied, which is the portion of the ARN that

begins with **`AmazonSageMaker-ExecutionRole-`**; then, press enter to bring up the role. Refer to *Figure 10.9*:


Identity and Access Management (IAM)

Dashboard

Access management
User groups
Users
Roles
Policies
Identity providers
Account settings

IAM > Roles

Roles (452) Info

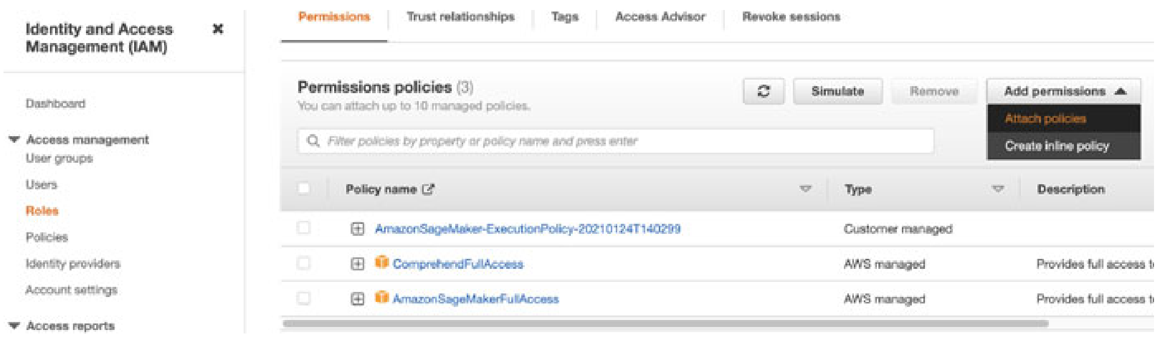
An IAM role is an identity you can create that has specific permissions with credentials that are valid for short durations. Roles can be assumed by entities that you trust.

AmazonSageMaker-ExecutionRole-20210124T140299 — 1 match

Role name	Trusted entities	Last ac...
AmazonSageMaker-ExecutionRole-20210124T140299	AWS Service: sagemaker	28 minutes ag

***Figure 10.9:** Lookup IAM role*

5. Click on the role name to select the role, and under **`Permissions`** policies, click on the **`Add permissions`** button and select **`Attach policies`**. Refer to *Figure 10.10*:


Permissions | Trust relationships | Tags | Access Advisor | Revoke sessions

Identity and Access Management (IAM)

Dashboard

Access management
User groups
Users
Roles
Policies
Identity providers
Account settings

Access reports

Permissions policies (3)

You can attach up to 10 managed policies.

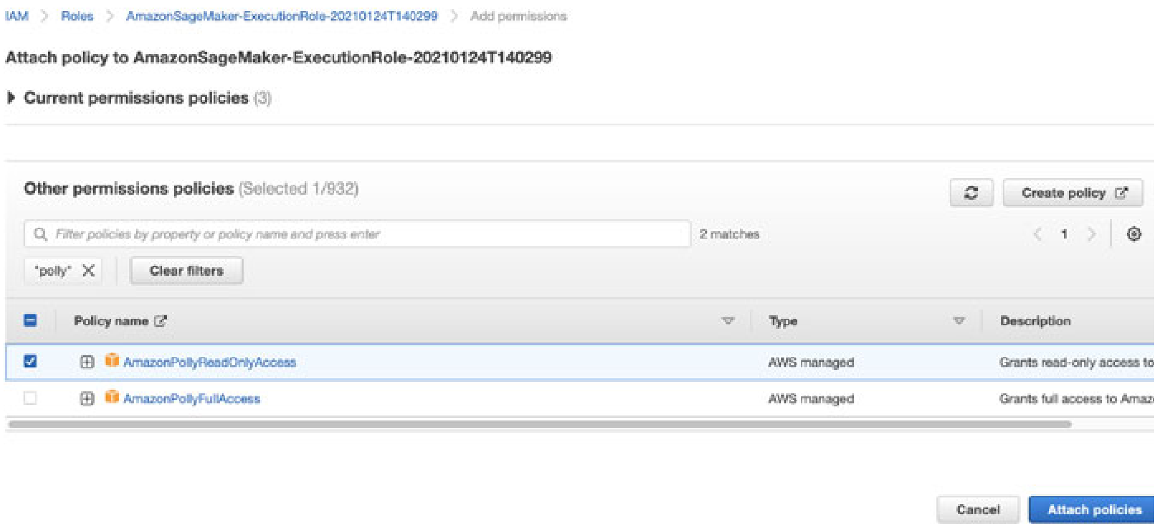
Filter policies by property or policy name and press enter

Simulate | Remove | Add permissions | Attach policies | Create inline policy

Policy name	Type	Description
AmazonSageMaker-ExecutionPolicy-20210124T140299	Customer managed	
ComprehendFullAccess	AWS managed	Provides full access t
AmazonSageMakerFullAccess	AWS managed	Provides full access t

***Figure 10.10:** Attach policies to role*

6. Now we will add a read only access policy for each service at a time. First, type polly in the policy search bar and click on the checkbox to the left of **`AmazonPollyReadOnlyAccess`** to select the policy. Do not click on the **`Attach policies`** button yet; we have a few more policies to attach, and we can do all of them at once here. Refer to *Figure 10.11*:



IAM > Roles > AmazonSageMaker-ExecutionRole-20210124T140299 > Add permissions

Attach policy to AmazonSageMaker-ExecutionRole-20210124T140299

Current permissions policies (3)

Other permissions policies (Selected 1/932)

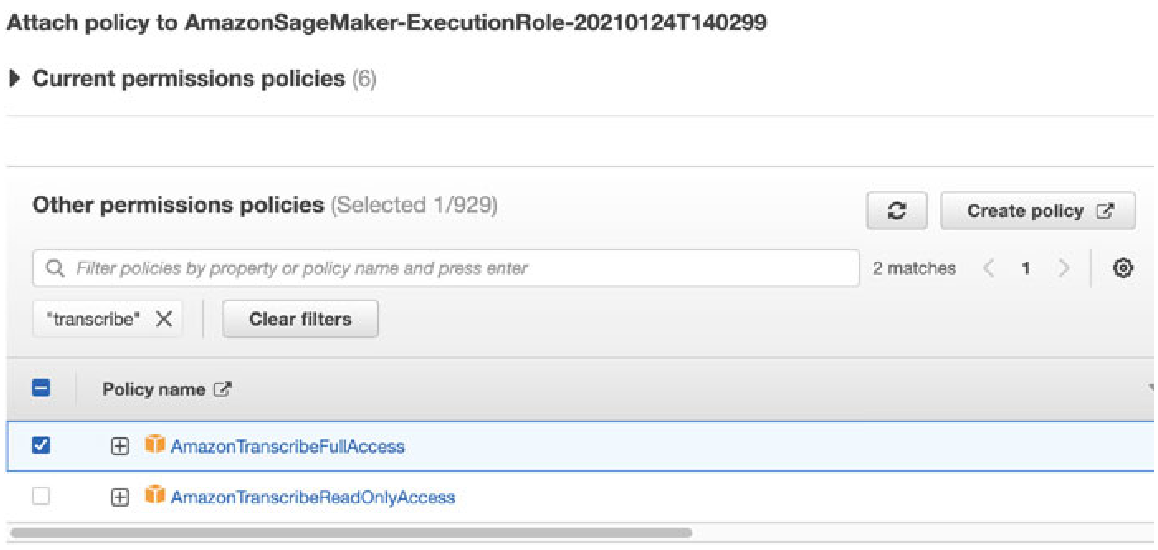
Filter policies by property or policy name and press enter — 2 matches

"polly" ✕ — Clear filters

	Policy name	Type	Description
✓	AmazonPollyReadOnlyAccess	AWS managed	Grants read-only access to
	AmazonPollyFullAccess	AWS managed	Grants full access to Amaz

Figure 10.11: *Add read only policy for Amazon Polly*

7. Clear the **`polly`** filter by clicking on the **`x`** next to it; then, type transcribe in the permissions search and add full access permissions for Transcribe because we will be using **`Transcribe API`** to submit a job. Refer to *Figure 10.12*:



Attach policy to AmazonSageMaker-ExecutionRole-20210124T140299

Current permissions policies (6)

Other permissions policies (Selected 1/929)

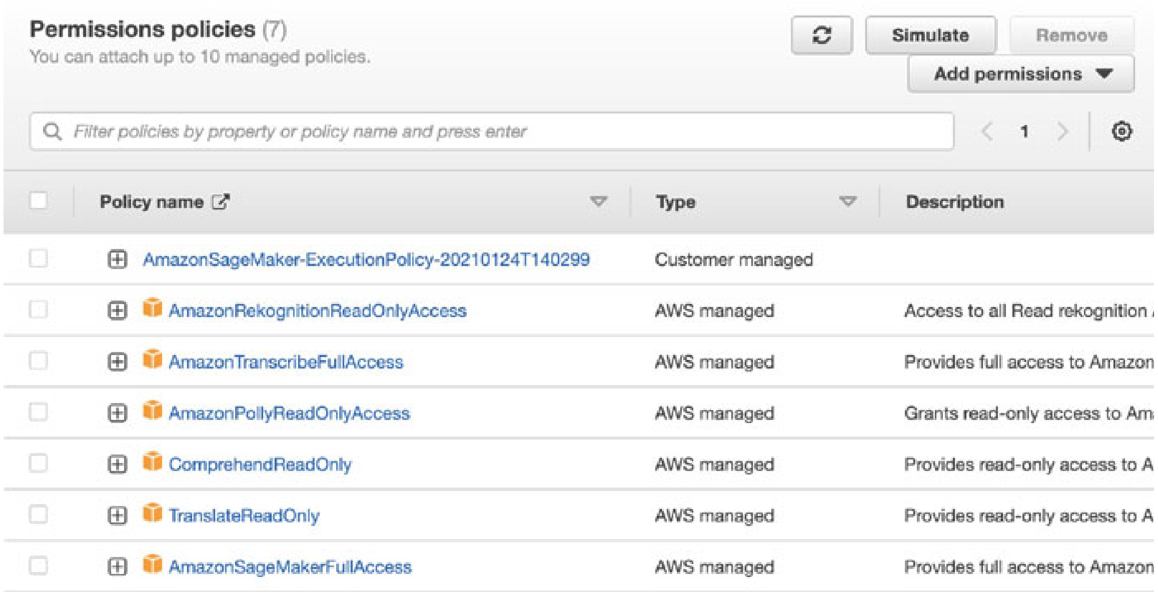
Filter policies by property or policy name and press enter — 2 matches

"transcribe" ✕ — Clear filters

	Policy name
✓	AmazonTranscribeFullAccess
	AmazonTranscribeReadOnlyAccess

Figure 10.12: *Select policy for Transcribe*

8. Now, repeat step 6 for each service name, one service at a time, for comprehend, translate, and rekognition, to add read-only policies for these services. Remember to clear the filter for the previous service before searching for the next service. Finally, click on the **`Attach policies`** button in the bottom-right corner of the page. When you are done, you should see the policies, as shown in the following image:


Permissions policies (7)

You can attach up to 10 managed policies.

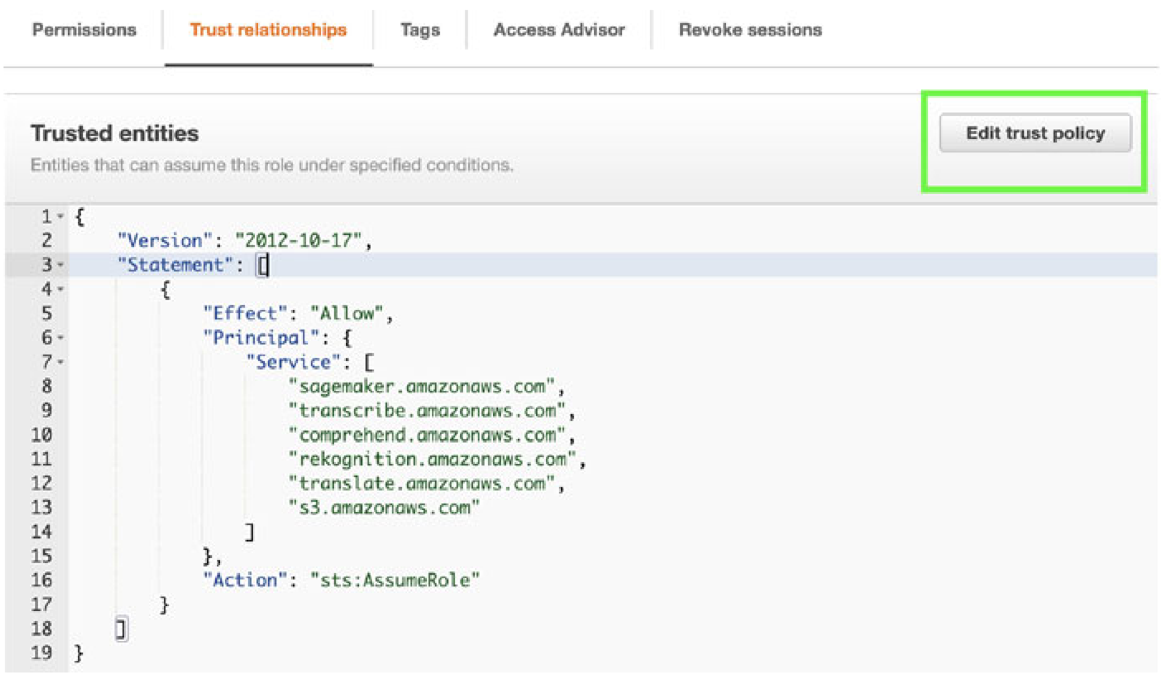
Policy name	Type	Description
AmazonSageMaker-ExecutionPolicy-20210124T140299	Customer managed	
AmazonRekognitionReadOnlyAccess	AWS managed	Access to all Read rekognition
AmazonTranscribeFullAccess	AWS managed	Provides full access to Amazon
AmazonPollyReadOnlyAccess	AWS managed	Grants read-only access to Am
ComprehendReadOnly	AWS managed	Provides read-only access to A
TranslateReadOnly	AWS managed	Provides read-only access to A
AmazonSageMakerFullAccess	AWS managed	Provides full access to Amazon

Figure 10.13: *AI services policies attached*

9. Now, click on the **`Trust relationships`** tab and then on **`Edit trust policy`**. Copy and paste the following JSON structure to add trusted relationships who can assume this execution role. After you have pasted the JSON, click on the **`Update policy`** button in the bottom-right corner of the page:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": [
          "sagemaker.amazonaws.com",
          "transcribe.amazonaws.com",
          "comprehend.amazonaws.com",
          "rekognition.amazonaws.com",
          "translate.amazonaws.com",
          "s3.amazonaws.com"
        ]
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

10. Once you have added the JSON, your Trusted entities should look as shown in the following image:



```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "sagemaker.amazonaws.com",
                    "transcribe.amazonaws.com",
                    "comprehend.amazonaws.com",
                    "rekognition.amazonaws.com",
                    "translate.amazonaws.com",
                    "s3.amazonaws.com"
                ]
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

Figure 10.14: *Trusted entities for the SageMaker execution role*

Alright, now we are ready to start playing with the AI services themselves. To execute the next set of steps, go back to your Amazon SageMaker console (type SageMaker in the services search bar and navigate to the console). Click on the `Studio` option under the `Getting started menu` on the left pane. Now, pick the user profile for your SageMaker Studio domain that you created in the previous chapters, from the list box and click on the `Open Studio` button. You should have already cloned the book's repository from GitHub at this juncture. If not, refer to the instructions at the beginning of this section. In your Studio page, on the left pane, click on the folder named after this book, and then click on `Chapter-10`. Now, click on the Jupyter notebook named `sensory-cognition-aws-ai.ipynb` to open it. First, we need to import the libraries we need to run the APIs for our services; execute the first cell to do this. As you can see, we are also declaring the boto3 (AWS Python SDK - **https://boto3.amazonaws.com/v1/documentation/api/latest/index.html**) handles for the various AI services in this step.

```
# declare the boto3 handles for each AI service
transcribe = boto3.client("transcribe")
rekognition = boto3.client("rekognition")
translate = boto3.client("translate")
polly = boto3.client("polly")
```

```
comprehend = boto3.client("comprehend")
# Amazon S3 (S3) client
s3 = boto3.client('s3')
```

Using Amazon Transcribe for speech recognition

We will first learn how to use Amazon Transcribe APIs to automate speech recognition and create text transcripts from audio files containing human speech. Execute the cells by reviewing the following instructions:

1. Execute the first cell in this section to declare the Amazon S3 bucket and the prefix we will use. Provide the name of the Amazon S3 bucket you created earlier:

   ```
   bucket = <your-S3-bucket-name>
   prefix = 'aiml-book/chapter10'
   ```

2. To run this code sample, we have provided you with an audio file of conversation between an agent and a customer at a contact center. Execute the next cell to upload this sample audio file to your S3 bucket:

   ```
   # First let us list our audio files and then upload them to the
   S3 bucket
   # we will use the example audio file we provided with the repo
   audio_dir = 'input/audio-recordings'
   for sdir, drs, fls in os.walk(audio_dir):
   for file in fls:
      s3.upload_file(os.path.join(sdir, file), bucket,
      prefix+'/transcribe/'+ os.path.join(sdir, file))
      uri = "s3://" + bucket + '/'+prefix+'/transcribe/' +
      os.path.join(sdir, file)
      print("Uploaded to: " + uri)
   ```

3. Now, we will use the Transcribe API to start the transcription job. In this example, we are using the asynchronous API, but Transcribe provides a number of APIs to run streaming transcriptions, call analytics and much more. For a full list of Transcribe APIs for Python, you can refer to **https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/transcribe.html**:

   ```
   # get the current time
   now = datetime.now()
   time_now = now.strftime("%H.%M.%S")
   job = 'transcribe-test-'+time_now
   ```

```
# start the transcription job
try:
transcribe.start_transcription_job(
    TranscriptionJobName=job,
    LanguageCode='en-US',
    Media={"MediaFileUri": uri},
    Settings={'MaxSpeakerLabels': 2, 'ShowSpeakerLabels': True}
    )
time.sleep(2)
print(transcribe.get_transcription_job(TranscriptionJobName=job
)['TranscriptionJob']['TranscriptionJobStatus'])
except Exception as e:
print(e)
```

4. The job status is printed as **`IN_PROGRESS`**. Visit **https://us-east-1.console.aws.amazon.com/transcribe/home?region=us-east-1#jobs** to navigate to the Amazon Transcribe console to review the job's progress. Kindly change the region from the previous link to the AWS region you are using.
5. The job will complete in 10 to 15 minutes. Once it is complete, execute the next cell to create an output directory we will use to store the transcription results:

```
# Create an output transcripts directory
dr = os.getcwd()+'/output-transcripts'
if not os.path.exists(dr):
os.makedirs(dr)
```

6. We will now execute the code to get the Transcript URI (the S3 bucket location where the job output is stored) and download the transcript to the notebook. We will then parse the JSON output to extract the transcript text segments we need, break it down to line-by-line structure, and write it into an output file. Execute the following cell:

```
# Our transcript is in a presigned URL in Transcribe's S3
bucket, let us download it and get the text we need
import urllib.request
response = transcribe.get_transcription_job(
TranscriptionJobName=job
)
out_url = response['TranscriptionJob']['Transcript']
['TranscriptFileUri']
infile = job+'-output.json'
```

```
urllib.request.urlretrieve(out_url, infile)
# declare an output file to store the transcripts
outfile = 'output-transcripts/'+job+'.txt'
with open(infile, 'rb') as t_in:
full = json.load(t_in)
entire_transcript = full["results"]["transcripts"]
lines = str(entire_transcript).split('. ')
i = 0
for line in lines:
   i += 1
   print("Line "+str(i)+": " + line)
# write the transcript to an output file
with open(outfile, 'w') as out:
  out.write(str(lines))
```

7. You will see the transcript results printed in the cell output. Only the first few lines are reproduced here for brevity:

```
Line 1: [{'transcript': "Thank you for calling Platinum Motors,
the Northwest Home for high end Autos
Line 2: I'm Michelle, how can I help you today? Uh Hi Michelle,
my name's Jack
Line 3: Uh I'm calling in today because about two weeks ago I
got a car from your dealership
Line 4: Um And uh the the service light came on already and I
know you guys handle a lot of your um your uh your fixing, your
mechanics are all in house and in shop and I'm just a little
baffled about how and why the light came on so soon
Line 5: So I wanted to give you guys a call
Line 6: Of course I am so sorry for the inconvenience
Line 7: Uh Jack what was your last name? So I can look you up
in the system
```

This is how you use Amazon Transcribe APIs to convert speech to text and create powerful transcripts from audio files. Transcribe supports multi-channel audio, can recognize different speakers in an audio file, supports changing transcription style using custom vocabulary and more. For details, please you can to the Amazon Transcribe documentation at **https://docs.aws.amazon.com/transcribe/index.html**.

Using Amazon Rekognition for computer vision

In this section, we will learn how to build computer vision applications using Amazon Rekognition with just API calls, without the need for ML training. Amazon Rekognition supports a broad range of computer vision use cases with specialized APIs for each task. For a full list of Amazon Rekognition features, read the documentation at **https://docs.aws.amazon.com/rekognition/latest/dg/what-is.html**. In our example here, we will show how to use the API to detect labels from images.

1. Execute the first cell in this section in the notebook to display our input image. This is the same image we used in Chapter 3, Predicting the Future With Features:

   ```
   # Lets use the Python image processing Pillow library
   from PIL import Image
   img = Image.open('./input/images/puppy-image.jpg')
   display(img)
   ```

2. The preceding code will display the image of the puppy:

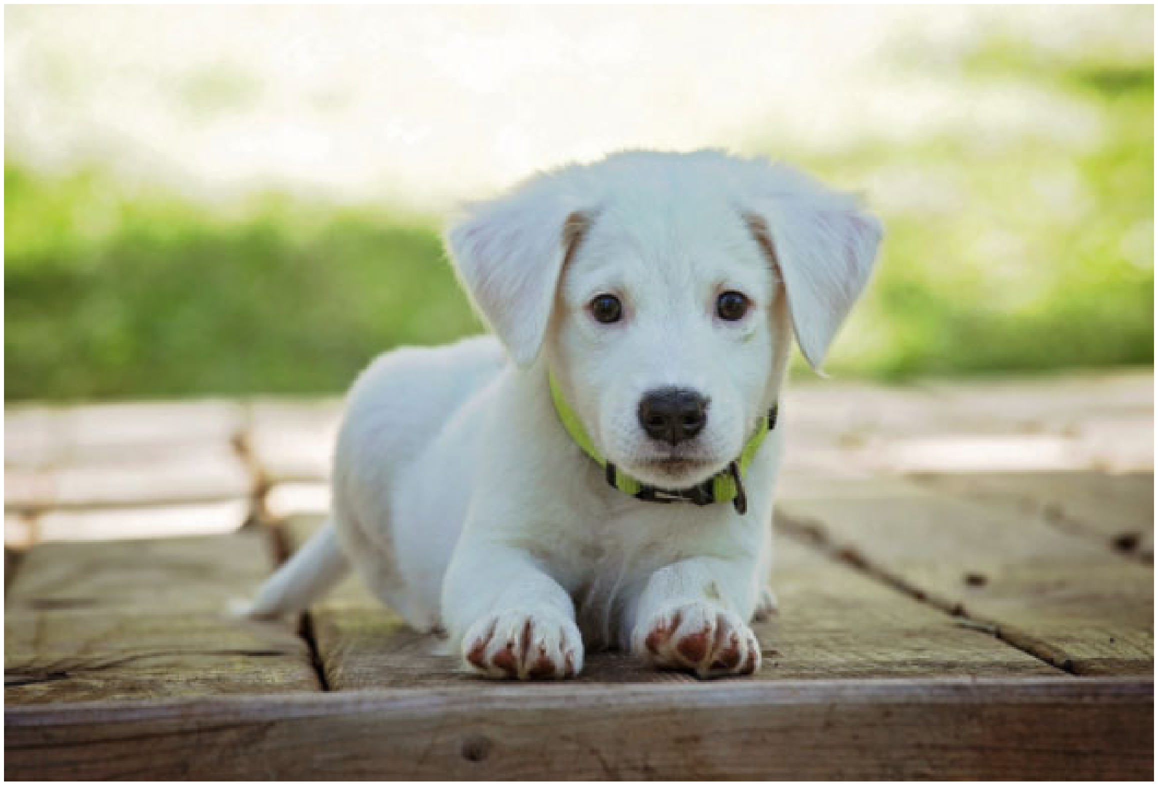


Figure 10.15: *Input image for Rekognition*

3. We will use the `DetectLabels` API (**https://docs.aws.amazon.com/rekognition/latest/APIReference/API_DetectLabels.html**) to understand what Amazon Rekognition sees in this

image. But first, we must upload the image of the puppy to an Amazon S3 bucket. Execute the following code cell to do so:

```
# First upload image to S3 bucket
input_dir = 'input/images'
prefix_uris = []
for sdir, drs, fls in os.walk(input_dir):
for file in fls:
   s3.upload_file(os.path.join(sdir, file), bucket,
   prefix+'/rekognition/'+ os.path.join(sdir, file))
   uri = "s3://" + bucket + '/'+prefix+'/rekognition/' +
   os.path.join(sdir, file)
   prefix_uri = prefix+'/rekognition/' + os.path.join(sdir,
   file)
   prefix_uris.append(prefix_uri)
  print("Uploaded to: " + uri)
```

4. Execute the next cell to invoke the DetectLabels API, passing the image as the input; we then print the results from Rekognition:

```
# Detect Labels
for prefix_uri in prefix_uris:
response = rekognition.detect_labels(
   Image={
    'S3Object': {
     'Bucket': bucket,
     'Name': prefix_uri
    }
   },
   MaxLabels=5,
)
for label in response['Labels']:
   print("Amazon Rekognition is
   "+str(round(label['Confidence'],0))+" confident that this
   picture is of a "+label['Name'])
```

5. When we look at the image, all we see is a cute puppy. But Rekognition sees much more than that. Review the results we print from Rekognition. We see only five print statements here because we specified **`MaxLabels=5`**, as shown in the highlighted section of the preceding code. If we increase the number of labels, we will get more information from Rekognition:

```
Amazon Rekognition is 98.0 confident that this picture is of a
Labrador Retriever
```

```
Amazon Rekognition is 98.0 confident that this picture is of a
Dog
Amazon Rekognition is 98.0 confident that this picture is of a
Pet
Amazon Rekognition is 98.0 confident that this picture is of a
Canine
Amazon Rekognition is 98.0 confident that this picture is of a
Mammal
```

DetectLabels is one of the many capabilities of Amazon Rekognition. The other APIs can analyze videos, compare faces, perform people pathing, detect personal protective equipment, recognize celebrities and more. For a full list of features, you can refer to the documentation at **https://docs.aws.amazon.com/rekognition/latest/dg/what-is.html**. And that's why we discussed using Amazon Rekognition to be our robot's eyes.

Using Amazon Translate for language translation

Previously, we learned how to use Amazon Transcribe for automatic speech recognition and Amazon Rekognition for computer vision applications. In this section, we will use Amazon Translate APIs to convert our English text from the transcripts Amazon Transcribe generated (refer to Step 7 in the *Using Amazon Transcribe for speech recognition* section) to Hindi and French without any need for ML training. Navigate to the Amazon Translate section in the notebook to follow instructions we will execute here:

1. Execute the first cell to create an output directory to store our translated text:

```
# Create an output translations directory
dr = os.getcwd()+'/output-translations'
if not os.path.exists(dr):
  os.makedirs(dr)
```

2. Since we will reuse some of the list variables we created when executing the Amazon Transcribe API calls, ensure that you have completed that section before you proceed. In this cell, we parse our transcript file and translate all text from even line numbers to Hindi and all text from odd line numbers to French. Run this cell:

```
x = 0
translate_out = 'output-translations/translations.txt'
t_list = []
```

```
# the lines list here was created when we executed the
Transcribe code sample earlier in this notebook.
# It contains lines of transcribed text
for line in lines:
 x += 1
 if (x % 2) == 0:
  result = translate.translate_text(Text=line,
  SourceLanguageCode='auto', TargetLanguageCode='hi')
 else:
  result = translate.translate_text(Text=line,
  SourceLanguageCode='auto', TargetLanguageCode='fr')
 t_list.append("Line "+str(x)+": "+result['TranslatedText'])
 with open(translate_out, 'w') as t_out:
 t_out.write(str(t_list))
# print the translation results
 for l in t_list:
  print(l)
```

3. The following output is printed. We will reproduce only a few lines of output in the book for brevity, but you can see the full output in the notebook:

```
Line 1: [{'transcript' : « Merci d'avoir appelé Platinum
Motors, le Northwest Home pour les automobiles haut de gamme
Line 2: मैं मिशेल हूं, आज मैं आपकी मदद कैसे कर सकता हूं? उह हाय मिशेल,
मेरा नाम जैक
Line 3: Euh, j'appelle aujourd'hui parce qu'il y a environ deux
semaines, j'ai reçu une voiture chez votre concessionnaire.
Line 4: उम और उह सेवा परकाश पहले से ही आ गया था और मुझे पता है कि
आप लोग आपके उम को अपने फिक्सिंग के बहुत सारे संभालते हैं, आपके यांत्रिकी
सभी घर में और दुकान में हैं और मैं बस इस बारे में थोड़ा चिकत हूं कि परकाश कैसे
और कयों आया इतनी जलदी
Line 5: Alors je voulais vous appeler.
Line 6: बेशक मुझे असुविधा के लिए बहुत खेद है
Line 7: Jack, quel était ton nom de famille ? Pour que je
puisse vous trouver dans le système
Line 8: यकीन है कि मेरा अंतिम नाम है, क्या आप कृपया मेरे लिए यह लिख
सकते हैं? एस आई
```

We also sent the translated output to a text file that is stored on the notebook that you can review and post-process as required.

We tried the real-time TranslateText API in this example. Amazon Translate provides features to customize your translation style using custom terminologies or parallel data, and you can run translations in real time or batch. Review the Amazon Translate documentation for more details: **https://docs.aws.amazon.com/translate/latest/dg/what-is.html**.

Using Amazon Polly for speech generation

Polly is an easy-to-use AI service to speech enable applications at any scale. It takes text as input and can speak this text in many voice styles, both male and female. Polly also supports multiple languages and has a unique voice collection tailored to language-based colloquialism. Navigate to the Amazon Polly section of the notebook and execute the following instructions in the notebook to see it in action.

1. Execute the first cell to define a text variable with a sample text we will use for testing:

```
input_text = "I think AI and ML are the most popular skills
right now, and I am glad I brought this book. It helps me learn
how to build real-world and large scale AI and ML applications
on Amazon Web Services. I loved the breadth and depth of
coverage on the ML workflow, on using the various features of
Amazon SageMaker, and the AI services that made powerful ML
models available behind simple API calls. Overall this books is
a very good learning resource"
```

2. In the next cell, we will create an output directory to store our audio file that Polly will generate:

```
# Create an output directory
dr = os.getcwd()+'/output-audio'
if not os.path.exists(dr):
os.makedirs(dr)
```

3. Execute the next cell to call the Polly API passing our input text to synthesize audio from the text. We will provide a voice, engine type and output format type as inputs along with the text:

```
response = polly.synthesize_speech(VoiceId='Kajal',
     OutputFormat='mp3',
     Text = input_text,
     Engine = 'neural')
```

```
mp3_file = open('./output-audio/chapter10-polly-test.mp3',
'wb')
mp3_file.write(response['AudioStream'].read())
mp3_file.close()
```

4. This generates an MP3 file and stores it in our output folder. In the next cell, we will play the audio using the IPython Audio utility. When you execute this cell, the audio player automatically plays the MP3 file, which will speak your input text in the voice you selected:

```
from IPython.display import Audio
Audio('./output-audio/chapter10-polly-test.mp3', autoplay=True)
```

That concludes the build of how to use Polly to speech enable your applications. Again, we used only one of the APIs. For a full list of Python APIs for Polly, you can refer to **https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/polly.html**. For more details on how Polly works and to learn about features like how to use SSML, Lexicons, and integrating with WordPress, you can refer to the developer guide at **https://docs.aws.amazon.com/polly/latest/dg/what-is.html**.

Using Amazon Comprehend for deriving insights

Amazon Comprehend is an NLP service with pre-trained models that provides a wide variety of text-based insights. We will use a few of Comprehend's real-time APIs to show how you can easily build applications that can generate these insights. To get started, navigate to the Amazon Comprehend section in the notebook and execute the following instructions to try the API calls.

1. Execute the first cell to get the name of the transcript file we generated when we worked with Amazon Transcribe. We will see what insights Comprehend can help derive from this transcript:

```
# For Comprehend we will take the Transcript output and see
what insights we can get from this text
transcript = 'output-transcripts/'+job+'.txt'
print(transcript)
```

2. We will read this transcript file and store the contents in a variable for easy reference:

```
# get the contents of the transcript into a text
with open(transcript, 'r') as comp_in:
in_text = comp_in.read().split(',')
# lets re-construct a full text from the list of sentences
```

```
full_text = ''
for text in in_text:
full_text += text+'. '
```

3. The first API we will try is the boto3 Python SDK version of Comprehend DetectEntities (**https://docs.aws.amazon.com/comprehend/latest/dg/API_DetectEntities.html**), which can surface unique references to people, places and things in text, without any type of ML training. Execute the next cell to call this API.

```
comp_res = comprehend.detect_entities(Text=full_text,
LanguageCode='en')
for entity in comp_res['Entities']:
print("Comprehend is "+str(round(entity['Score']*100,0))+"%
confident that "+entity['Text']+" is an entity of type
"+entity['Type']+" ")
```

4. We will see the following results printed. Once again, only the first few entities are shown here, but the notebook has the full list that Comprehend detects:

```
Comprehend is 100.0% confident that Platinum Motors is an
entity of type ORGANIZATION
Comprehend is 77.0% confident that Northwest Home is an entity
of type ORGANIZATION
Comprehend is 100.0% confident that Michelle is an entity of
type PERSON
Comprehend is 99.0% confident that today is an entity of type
DATE
Comprehend is 100.0% confident that Michelle is an entity of
type PERSON
Comprehend is 100.0% confident that Jack is an entity of type
PERSON
```

5. The next API we will try is **`DetectKeyPhrases ()`**, a collection of words in the text that is key to bringing out the meaning of a sentence. Execute the next cell and review the output:

```
# Read and print the key phrases
comp_res = comprehend.detect_key_phrases(Text=full_text,
LanguageCode='en')
for phrase in comp_res['KeyPhrases']:
print("Comprehend is "+str(round(phrase['Score']*100,0))+"%
confident that "+phrase['Text']+" is a key phrase")
```

6. We will see the following output:

```
Comprehend is 99.0% confident that about two weeks is a key
phrase
Comprehend is 100.0% confident that a car is a key phrase
Comprehend is 100.0% confident that your dealership is a key
phrase
```

7. Next, we will look at how we can use Comprehend APIs to detect sentiment in text. Execute the next cell and review its output:

```
sent_text = 'Also if you wanted to wait for the two days we
could also have a rental car available for you at no charge in
case you wanted that in case it takes a little bit longer to
fix that will be an option that we can plan out for you as well
but again you are also welcome to come in any time before then
and we will get you in as soon as we can'
comp_res = comprehend.detect_sentiment(Text=sent_text,
LanguageCode='en')
print(comp_res['Sentiment'])
print(comp_res['SentimentScore'])
```

8. We will see the following output:

```
POSITIVE
{'Positive': 0.4174244999885559, 'Negative':
0.018003448843955994, 'Neutral': 0.34651508927345276, 'Mixed':
0.21805694699287415}
```

9. The last API we will try in this notebook is the DetectSyntax API. Execute the next cell and review its output:

```
synt_text = 'Also if you wanted to wait for the two days we
could get a rental car'
comp_res = comprehend.detect_syntax(Text=synt_text,
LanguageCode='en')
for token in comp_res['SyntaxTokens']:
print("The Part of Speech for word: "+token['Text']+" :is:
"+token['PartOfSpeech']['Tag'])
```

10. We will get the following output:

```
The Part of Speech for word: Also :is: ADV
The Part of Speech for word: if :is: SCONJ
The Part of Speech for word: you :is: PRON
The Part of Speech for word: wanted :is: VERB
The Part of Speech for word: to :is: PART
The Part of Speech for word: wait :is: VERB
The Part of Speech for word: for :is: ADP
The Part of Speech for word: the :is: DET
```

That was the just the tip of the iceberg on what Comprehend can do. For a full list of the Comprehend APIs, refer to **https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/comprehend.html**.

With that, we conclude this chapter on how you can use AWS AI services with sensory cognition capabilities to innovate your applications. We covered the major AI services in this chapter, but there are several other complementary AI services like Amazon Lex (**https://aws.amazon.com/lex/**) for conversational AI, Amazon Kendra (**https://aws.amazon.com/kendra/**) for NLP powered enterprise search, and others that, when combined, can lead to powerful digital transformation of your enterprise footprint.

Conclusion

As you might have realized. AWS has automated and pre-trained ML models for various use cases already and abstracted them behind APIs. This greatly simplifies your approach and saves you time and costs. There is a demand for more automation and simplification in ML because traditionally, ML projects take a long time and require heavy compute resources, which is challenging, especially if you are on a budget. With the economic advantage provided by the cloud and pre-trained ML models like APIs, all it takes for you to ML enable your applications is a few lines of code. And this is only the beginning; it is an indicator of what we can expect in the next few years. In this chapter, we learned a set of AWS AI services for sensory cognition. In the next chapter, we will learn about a different set of AWS AI services, specifically for industrial automation like detecting anomalies in machine parts, predicting equipment issues, and shaping our predictive analytics strategy using APIs provided by these services without the need for long hours of ML development.

Points to Remember

Here's a summary of what we learned in this chapter:

- We were introduced to the AI services layer of the AWS AI/ML stack, and we reviewed key services that simulates sensory cognition capabilities.
- We used a fun example of a robot and looked at adding abilities for the robot to see, listen, speak, translate and infer using faculties of ears, eyes,

mouth and brain.

- We then learned key capabilities of the sensory cognition AI services as they map to satisfying the functions we needed for the robot.
- We then pivoted to using the AI services APIs to learn how to programmatically add AI capabilities to application builds.
- We executed Python code samples using Amazon Transcribe APIs for creating text transcriptions from audio files.
- We learned how to use Amazon Rekognition to detect labels from images.
- We learned how to use Amazon Translate for translating our transcript text to multiple languages.
- We learned how to use Amazon Polly to generate speech from text that we stored in a MP3 file.
- Finally, we learned how to use Amazon Comprehend to derive unique insights from text.

Multiple Choice Questions

Use these questions to challenge your knowledge in the AWS AI services you learned in this chapter.

1. **What is NOT the right option to detect insights from text data?**
   a. Using Amazon Comprehend to extract entities and key phrases from text
   b. Using Amazon SageMaker BlazingText algorithm for classification
   c. Using HuggingFace distilBERT containers with SageMaker for named entity recognition
   d. Using Amazon Translate to translate text into a different language
2. **With what AI service can you generate text labels of image data?**
   a. Amazon Comprehend
   b. Amazon Textract
   c. Amazon Rekognition
   d. Amazon Polly
3. **AWS AI services provide pre-trained inference ready models abstracted as an API.**
   a. True

    b. False

4. **Amazon Polly adds voice to applications, but it cannot be integrated with other AWS AI services and must be used on its own.**

    a. True

    b. False

5. **What AWS AI services allow you to use incremental training or transfer learning to train your own unique custom models without any ML expertise?**

    a. Amazon Textract

    b. Amazon Comprehend

    c. Amazon Translate

    d. Amazon Rekognition

    e. Amazon Transcribe

Answers

1. **d**
2. **c**
3. **a**
4. **b**
5. **b,d and e**

CHAPTER 11
AI for Industrial Automation

Introduction

Continuing our discussion about robots from *Chapter 10, Adding Intelligence With Sensory Cognition*, do you remember that factory scene from the movie "I, Robot," where a batch of newly built robots is waiting to be activated? Science fiction and movies may tempt us to imagine a fully automated factory to be teeming with humanoid robots doing all kinds of activities very similar to humans. In reality, most, if not all, of the factory floors are either semi or fully automated today; they are powered by AI and robotics. However, practical common sense dictates that AI doesn't have to be deployed in humanoid shapes (like an android robot) to be of use. In fact, today, with the power of ubiquitous internet and democratized AI/ML with AWS, advanced AI capabilities offered by the cloud can be easily leveraged from the factory floor for various predictive maintenance and analytics tasks. The keyword here is predictive. Why is this important? Earlier, when a critical piece of machinery had to be serviced, entire assembly lines had to be brought down, resulting in halted production and significant revenue loss. Further, the cost of servicing or repairing a defect was exacerbated because of having to troubleshoot and determine what was wrong in the first place.

Imagine if you could continuously monitor your machinery and exactly predict when a particular part or component would malfunction. You could then set up an automated notification to schedule an inspection/replacement for the part just before it breaks down. Knowing in advance also means you are in control of when to replace the part. This means you can take care of it during operational downtime rather than having to bring down production just to replace the part. Now, couple this scenario with the ability to automate quality control for your manufactured products. Automotive companies lose millions of dollars when they Recall car models because of manufacturing issues. Ideally, these are due to faults that should have been caught in quality inspections but were missed due to error or inadvertence. With AI, you can enable highly accurate fault checks for manufactured products, and you can also build a continuous improvement workflow for inspection and remediation. These are just two of the numerous possibilities of the ways in which AI can empower industrial automation, improve your operational efficiencies, help you cut costs, and increase your profitability.

Till now in this book, we walked through how to design, build, train and deploy ML workflows for various use cases. In the previous chapter, we were introduced to AWS AI services that are pre-trained models available behind APIs for several common ML tasks. We also learned how to build applications to leverage sensory cognition AI services without the need for any ML skills or expertise, just by using AWS Python SDK APIs. In this chapter, we will continue to learn about AWS AI services, especially the ones that enable industrial automation capabilities. We will first be introduced to two such AI services, and then we will build a predictive analytics solution with these services.

Structure

In this chapter, we will dive deep into the following topics:

- Overview of AI for Industrial Automation
- Predictive Analytics with Amazon Lookout for Equipment
- Quality control with Amazon Lookout for Vision

Objectives

In this chapter, we will first discuss industrial automation use cases and why these are important, and then we will look at how AI can help. We will learn about an AWS AI service for detecting abnormalities in industrial equipment called **Amazon Lookout for Equipment**. We will then pivot to learn about a different AWS AI service, called Amazon Lookout for Vision, for detecting faults in a product line using computer vision. Finally, we will walk through a solution built using both of these services for predictive analytics.

Overview of AI for Industrial Automation

In the previous chapter, we reviewed core AI services for use cases such as text-to-speech, speech-to-text, deriving insights from text, and detecting and recognizing image content. We saw that these AI services are pre-trained ML models abstracted as scalable and reliable APIs with high-performance capabilities for **automatic speech recognition** (**ASR**), **natural language understanding** (**NLU**), **natural language processing** (**NLP**), **computer vision** (**CV**), and more. We saw that we needed just a few lines of code to make an API call; we sent inputs and received predictions. This chapter will focus on AI services that help with automating industrial use cases, specifically, anomaly detection from sensor readings and detecting faults from product images for quality control checks.

"If everyone is moving forward together, then success takes care of itself."

- Henry Ford

In 1913, Henry Ford famously deployed his first moving assembly line for automotive manufacturing at Ford's Highland Park assembly plant (**https://corporate.ford.com/articles/history/moving-assembly-line.html**). Designed to bring products and activities to workers rather than the other way around, the assembly line changed mass production and significantly improved manufacturing efficiencies at the same time. It also cut operational costs, making the end product more affordable for consumers, thereby boosting revenue and profits. Today, after 100 years, the assembly line is still the most efficient way to make cars. While we haven't invented something simpler than assembly lines, we have made huge technological advancements in how we use them. If we have to distill all that we have learned in the last century or so on how to cut costs and make manufacturing more efficient, we can fit it into two keywords: "automation" and "continuous improvement".

Automation reduces the need for human intervention and dependence, cuts costs, and saves time. Automation also provides capabilities to measure, monitor, calibrate and improve manufacturing tasks. Traditionally, automated production required tasks to be iterative and repeatable, and generally needed them to follow the norms of a process. So, tasks that were deterministic and repetitive, such as assembling the body of a car or attaching tires to the axles in the chassis, were automated using industrial machines, but tasks that were probabilistic, such as inspecting the quality of the assembled car, running diagnostic tests of the controller area network, or checking the capacity of the compressor, were performed manually. Continuous improvement, or CI, refers to the ability to fine-tune process efficiencies so that production moves toward peak performance irrespective of changing scope and scale. CI requires active monitoring of tasks, measuring dips in performance, and the ability to influence factors that can improve production capacity, reduce time, and use resources efficiently. The Lean Six Sigma process framework provides a methodical approach to CI (**https://www.investopedia.com/terms/l/lean-six-sigma.asp**) and aims to reduce defects or wastage from a business process to improve its efficiencies. Lean was a process developed by Toyota to identify and remove tasks/steps that do not add value to a process. Six Sigma was developed by Motorola to define standards for business process improvement. However, traditional applications of the Lean Six Sigma relied heavily on human involvement, especially in the Define, Analyze and Improve phases (**https://goleansixsigma.com/wp-content/uploads/2012/02/DMAIC-The-5-Phases-of-Lean-Six-Sigma-www.GoLeanSixSigma.com_.pdf**).

Manufacturing processes have undergone a lot of changes over the years, but there is still room to improve efficiency, cut costs and achieve a higher degree of automation. This was made possible with the advent of AI technologies. So how can AI help with automation and continuous improvement in manufacturing and other industries? To answer this question, let us go back to our list of AWS AI services from *Chapter 1, Introducing the ML Workflow*, and review the AI services that are part of the industrial automation category. Refer to the following table.

ML problem type	ML subtype	ML use case	AWS AI Service	Service Overview
Enterprise/Industrial	Business tools	Fraud detection	Fraud Detector	Fully managed service for detecting and monitoring eCommerce fraud with a simplified interface for model building with no ML expertise required
		Recommendations	Personalize	Fully managed service for detecting and monitoring eCommerce fraud with a simplified interface for model building with no ML expertise required
		Forecasting	Forecast	Fully managed service for detecting and monitoring eCommerce fraud with a simplified interface for model building with no ML expertise required
		Anomaly detection	Lookout for Metrics	Fully managed service for detecting and monitoring eCommerce fraud with a simplified interface for model building with no ML expertise required
	Industrial Intelligence	Predictive monitoring	AWS Panorama	Hardware and SDK for adding ML-based intelligence to the factory floor and on-premises camera systems
			Amazon Monitron	Sensors and pre-trained ML for industrial equipment monitoring and proactive maintenance
			Amazon	Custom ML pre-trained for

			Lookout for Vision	industrial anomaly detection using computer vision
		Predictive analytics	Amazon Lookout for Equipment	ML service that provides intuitive capabilities for developing custom models for detecting equipment anomalies with no ML expertise required

Table 11.1: *AWS AI services for industrial automation*

Cost of Poor Quality or COPQ

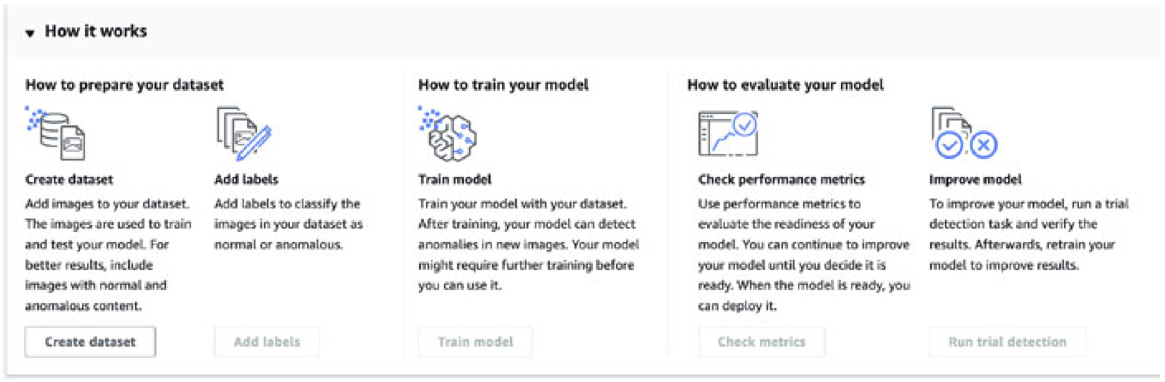
Cost of Poor Quality (COPQ) (**https://sixsigmastudyguide.com/cost-of-poor-quality/**) is a key metric that measures the performance efficiency of a manufacturing process by quantifying the costs associated with regressive effort and rework and impact on operational costs. Organizations constantly aim to reduce COPQ by trying to prevent rework. There are a couple of ways by which this can be done:

- Improve product quality
- Prevent manufacturing failures

Improve product quality

When a product is manufactured with high quality or there are processes in place that identify quality issues early and accurately, rework effort is automatically reduced. The cost of fixing defects increases exponentially with an increase in the time it takes to identify a defect or if it is identified late in the manufacturing cycle. A defect in the initial build phase is easier to fix than after the product has been manufactured. Further, proper identification and categorization of errors is equally critical. If an error is incorrectly labeled, it may not only impact the current process, but the fixing of the original error may introduce new and more troublesome issues. The accuracy of the system detecting the error determines the overall quality because it is OK to flag false positives (that there is an error) but not false negatives (the model assumes no issue even though there is an issue).

Amazon Lookout for Vision (**https://aws.amazon.com/lookout-for-vision/**) is a service that helps with this problem. It is a fully managed computer vision service that is purpose-built for automated quality control inspections, and it can work both on the cloud and at the edge directly in your factory floors or manufacturing facilities. It uses proprietary computer vision modeling techniques to build powerful anomaly detection models with less than 50 images in your dataset. The training dataset should contain images of the product when it is NOT anomalous, or when it is normal. At the time of testing, we send anomalous images to the model and review the prediction results. Take a look at the following image showing the process of working with Lookout for Vision.



▾ How it works

How to prepare your dataset

Create dataset
Add images to your dataset. The images are used to train and test your model. For better results, include images with normal and anomalous content.
Create dataset

Add labels
Add labels to classify the images in your dataset as normal or anomalous.
Add labels

How to train your model

Train model
Train your model with your dataset. After training, your model can detect anomalies in new images. Your model might require further training before you can use it.
Train model

How to evaluate your model

Check performance metrics
Use performance metrics to evaluate the readiness of your model. You can continue to improve your model until you decide it is ready. When the model is ready, you can deploy it.
Check metrics

Improve model
To improve your model, run a trial detection task and verify the results. Afterwards, retrain your model to improve results.
Run trial detection

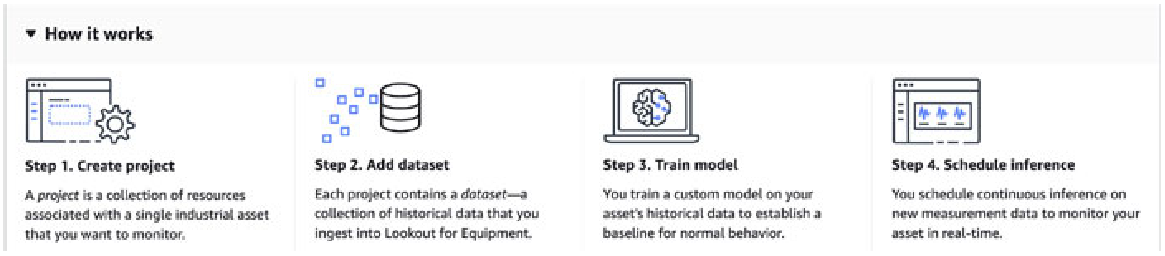
Figure 11.1: *Lookout for Vision modelling process*

In a subsequent section in this chapter, we will walk through an example of how to automate quality control inspections with Amazon Lookout for Vision, where we will simulate quality control for electronics printed circuit board manufacturing facility.

Prevent manufacturing failures

System failures in the assembly lines are those impacting the production lines. They are a huge deal in manufacturing facilities. According to an article: https://www.automation.com/en-us/articles/june-2021/world-largest-manufacturers-lose-almost-1-trillion (accessed on October 2022), manufacturing and industrial Fortune 500 companies, which include some of the biggest brands, lose a staggering $864 billion every year due to unplanned downtimes caused by faulty machines. So, predictive analytics, which is the art and science of monitoring and mitigating equipment failure in advance, or providing recourse to scheduled downtimes, is critical to reducing manufacturing costs and improving business profitability. Until the advent of ML and AI, we did not have predictive analytics, so to speak. Instead, we had a reactive approach that would continuously monitor machine performance and alert us when something was wrong. While an advanced alert was helpful, it did not provide a means to achieve a controlled environment to proactively address issues. With ML, you can train models to predict a threshold of when your machinery is tending towards failure and prioritize maintenance for these machines accordingly.

Amazon Lookout for Equipment (**https://aws.amazon.com/lookout-for-equipment/**) is a managed service that enables you to ingest time series data from equipment and machinery sensors, analyze the data, train a model either per sensor or for the machinery as a whole, and predict anomalies in sensor data either in real time or at scheduled intervals. The benefit of using Lookout for Equipment is that you do not need any ML skills at all. You can directly use the Amazon Lookout for Equipment console to go through the entire ML life cycle, with all the infrastructure provisioning, feature engineering, data transformation, algorithm selection, model tuning, and deployment completely abstracted from the user, making it easy to implement predictive analytics. Refer to *Figure 11.2*:



▾ How it works

Step 1. Create project
A *project* is a collection of resources associated with a single industrial asset that you want to monitor.

Step 2. Add dataset
Each project contains a *dataset*—a collection of historical data that you ingest into Lookout for Equipment.

Step 3. Train model
You train a custom model on your asset's historical data to establish a baseline for normal behavior.

Step 4. Schedule inference
You schedule continuous inference on new measurement data to monitor your asset in real-time.

Figure 11.2: *Lookout for Equipment modeling process*

In a subsequent section in this chapter, we will walk through Amazon Lookout for Equipment directly using the AWS console and also through the end-to-end modeling process. We will use time series sensor data collected for a duration of 4 months from compressor pumps at 1-minute intervals. We will create a project, add a dataset, train a model, and schedule inference, all without any ML expertise or provisioning any infrastructure.

Quality Control with Amazon Lookout for Vision

In this section, we will get hands-on experience; we will build a quality control inspection model using Amazon Lookout for Vision. We will use the **Visual Anomaly (VisA)** dataset introduced by this paper - **https://arxiv.org/abs/2207.14315**, specifically, the PCB1 category of printed circuit board (PCB) images, for fault detection of manufactured PCBs. After the training is complete, the model will have the ability to look at images of anomalous PCBs and correctly identify issues, and it will also have the ability to classify non-faulty images as normal. For this activity, we will use the AWS Management Console and see how we can build and deploy powerful computer vision models at scale with no ML expertise.

Before we get started, we need to ensure that our prerequisites are in place. In case you haven't signed up for an AWS account yet, now is the time to do so. Follow the instructions in the Setting up your AWS account section in Chapter 2, Hydrating Your Data Lake, to sign up for an AWS account. Once you have signed up, log in to your AWS account using the instructions at **https://docs.aws.amazon.com/IAM/latest/UserGuide/console.html**. If you have not already done so, create an Amazon S3 bucket (**https://docs.aws.amazon.com/AmazonS3/latest/userguide/create-bucket-overview.html**). Execute the following instructions to build, train and deploy your quality control model for PCB anomaly detection:

1. Once you have logged in to the AWS Management Console, type vision in the services search bar and select **`Amazon Lookout for Vision`** to be navigated to its console.
2. Click on the **`Get started`** button; you will be prompted to create an Amazon S3 bucket. Then, click on the **`Create S3 bucket`** button. In the Dashboard, in the **`Projects`** pane, click on the **`Create project`** button to start our model build process. Refer to *Figure 11.3*:

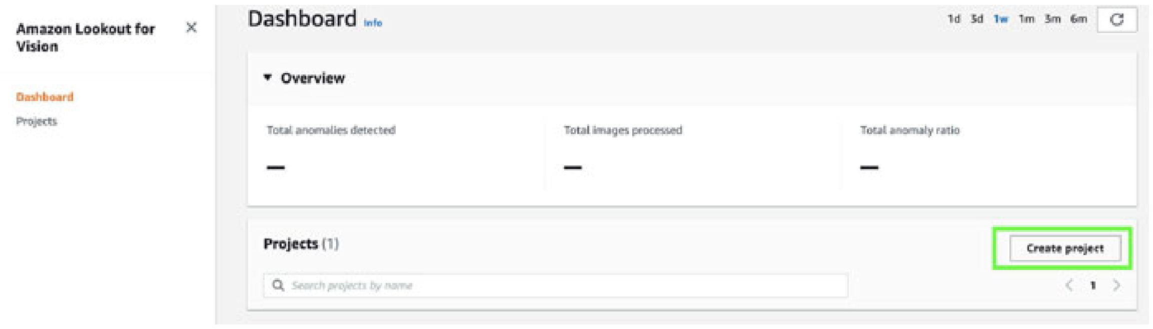
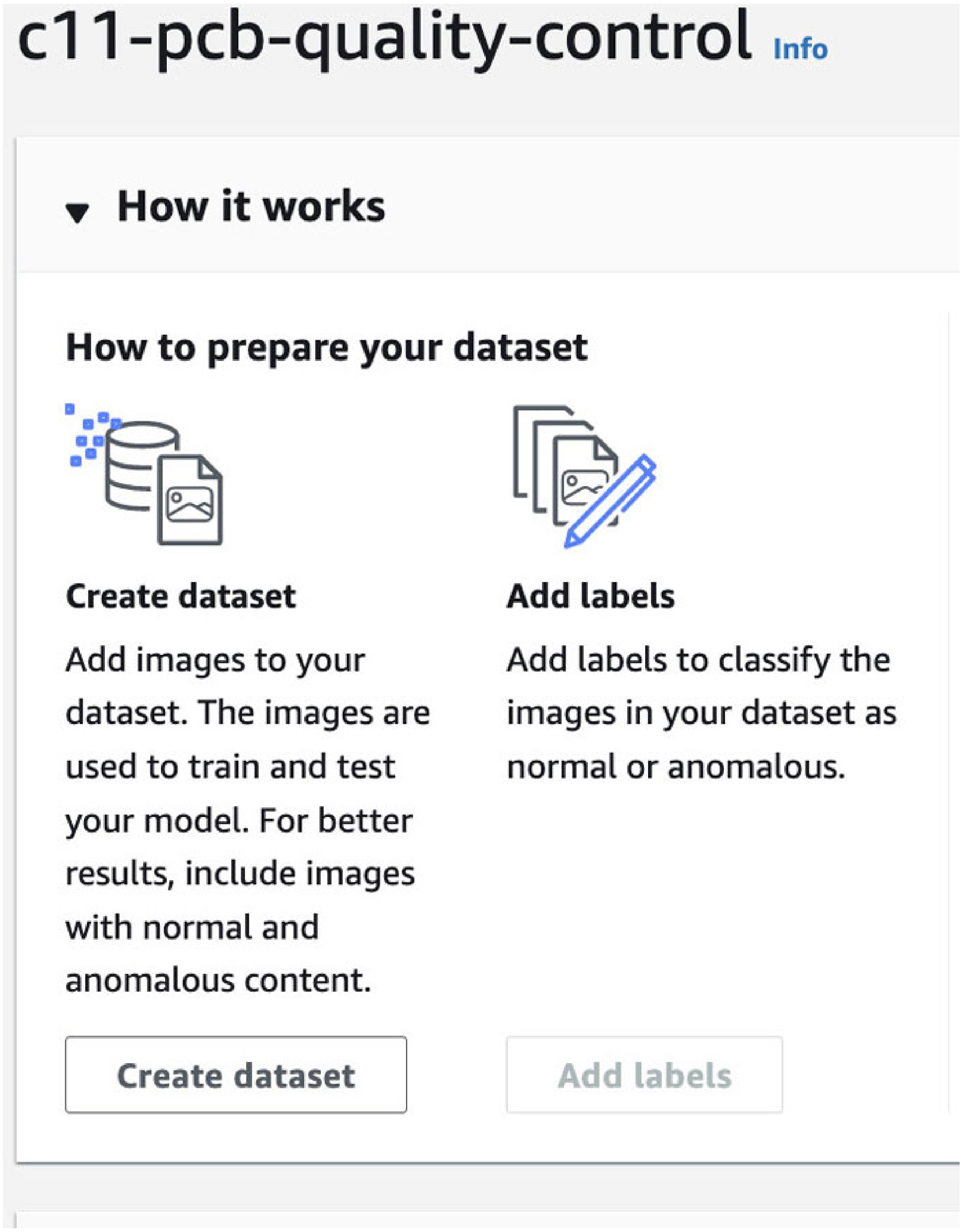


Figure 11.3: *Create a Lookout for Vision project*

3. In the **`Project details`** section, for the Project name, type **`c11-pcb-quality-control`** and click on **`Create project`**.
4. In the *How it works* section of your project, click on the **`Create dataset`** button. Refer to *Figure 11.4*:



c11-pcb-quality-control Info

▾ How it works

How to prepare your dataset

Create dataset

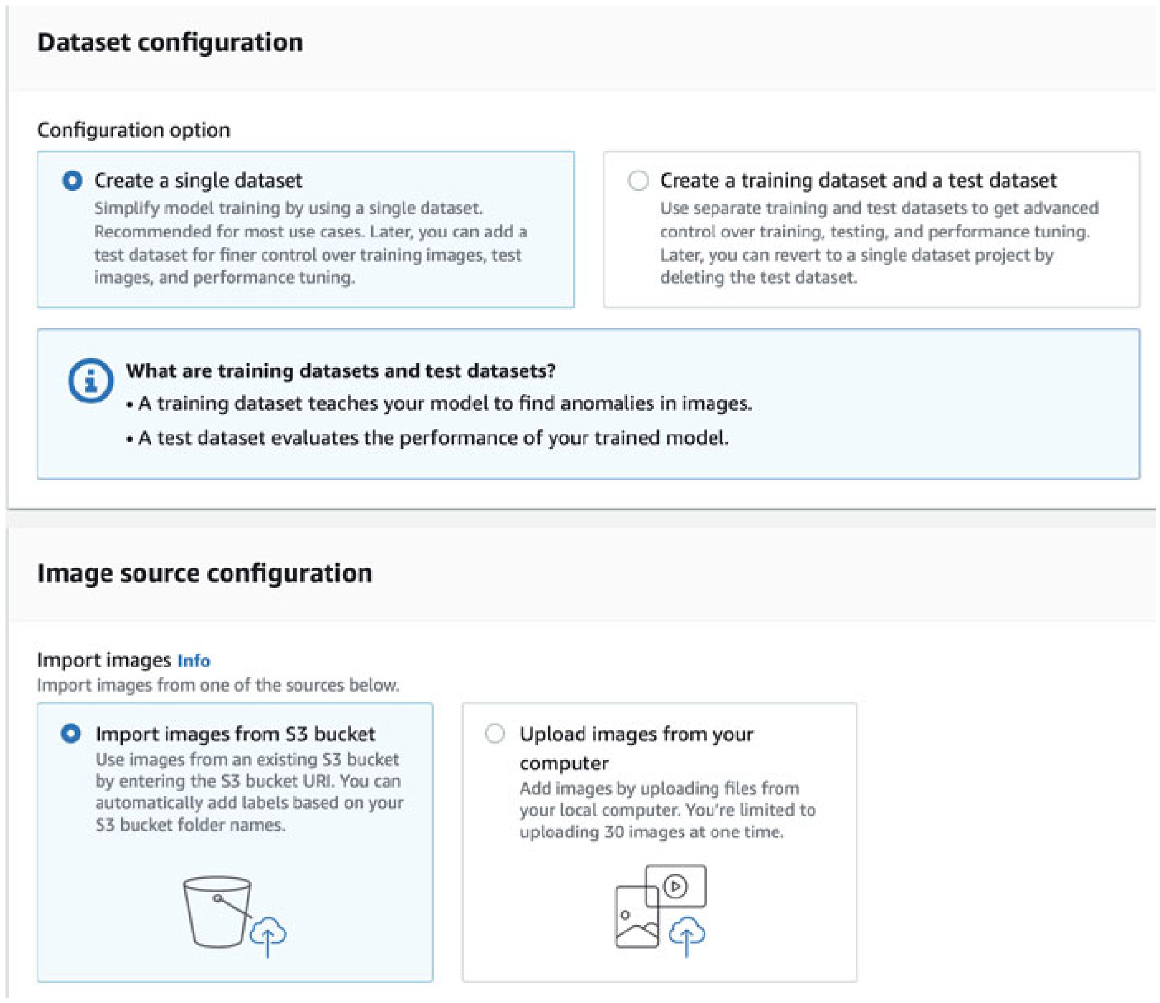
Add images to your dataset. The images are used to train and test your model. For better results, include images with normal and anomalous content.

Add labels

Add labels to classify the images in your dataset as normal or anomalous.

Figure 11.4: *Create dataset*

5. On the **`Create dataset`** page, leave the default selection of **`Create a single dataset`**. Scroll down to the **`Image source configuration`** section and select **`Import images from S3 bucket`**. Refer to *Figure 11.5*:


Dataset configuration

Configuration option

[✓ Create a single dataset]
Simplify model training by using a single dataset. Recommended for most use cases. Later, you can add a test dataset for finer control over training images, test images, and performance tuning.

Create a training dataset and a test dataset
Use separate training and test datasets to get advanced control over training, testing, and performance tuning. Later, you can revert to a single dataset project by deleting the test dataset.

What are training datasets and test datasets?
• A training dataset teaches your model to find anomalies in images.
• A test dataset evaluates the performance of your trained model.

Image source configuration

Import images Info
Import images from one of the sources below.

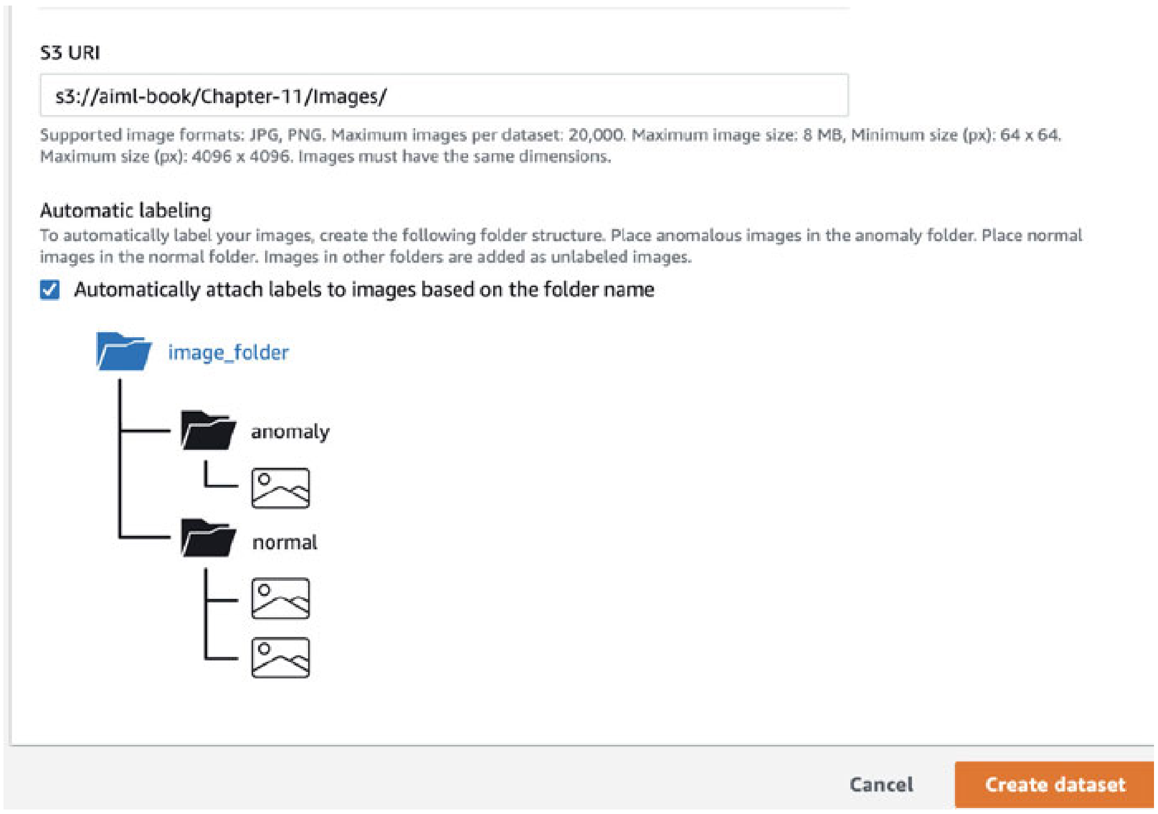
[✓ Import images from S3 bucket]
Use images from an existing S3 bucket by entering the S3 bucket URI. You can automatically add labels based on your S3 bucket folder names.

Upload images from your computer
Add images by uploading files from your local computer. You're limited to uploading 30 images at one time.

Figure 11.5: *Select images from S3 bucket*

6. Now, open a new tab in your browser and download the dataset from the book's GitHub repository provided at the beginning of the book. The folder and file name you are looking for are **`Chapter-11/chapter11-pcb-images.zip`**.
7. Extract the files from the ZIP file to your local computer. You should see the following folders, with each folder containing 45 to 50 images:
   - Images/Normal
   - Images/Anomaly
8. Create an Amazon S3 bucket (if not already done) using the instructions at **https://docs.aws.amazon.com/AmazonS3/latest/userguide/create-bucket-overview.html** and upload the Images folder to the S3 bucket using the **`Add folder`** option.

9. Provide the S3 URI of where your images are stored all the way up to the Images folder. Select the **`Automatic labeling`** checkbox and click on the **`Create dataset`** button. Refer to *Figure 11.6*:



S3 URI

s3://aiml-book/Chapter-11/Images/

Supported image formats: JPG, PNG. Maximum images per dataset: 20,000. Maximum image size: 8 MB, Minimum size (px): 64 x 64. Maximum size (px): 4096 x 4096. Images must have the same dimensions.

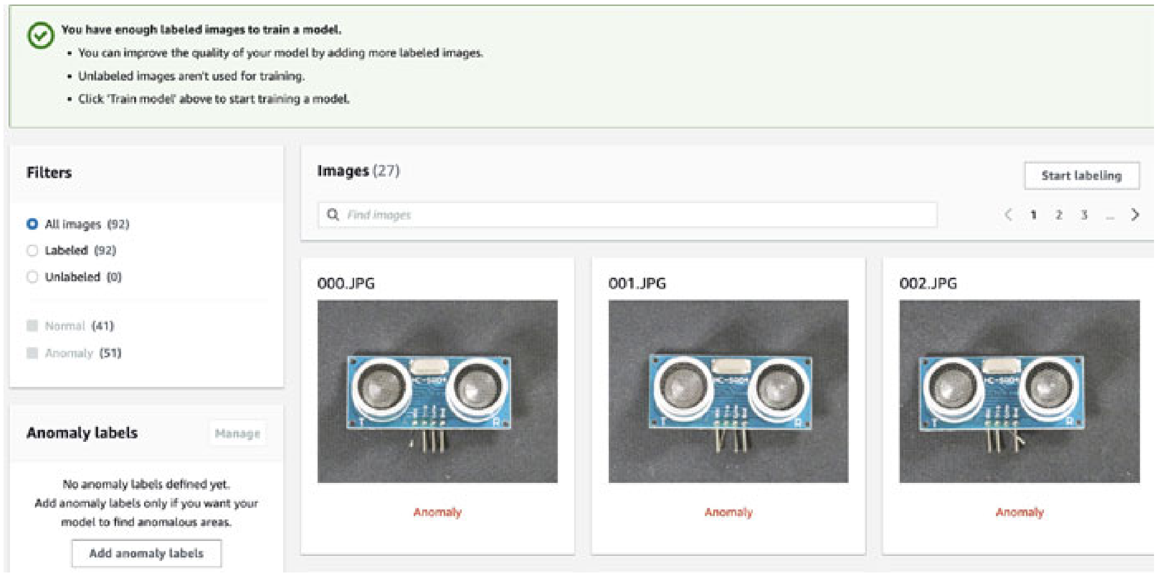
Automatic labeling

To automatically label your images, create the following folder structure. Place anomalous images in the anomaly folder. Place normal images in the normal folder. Images in other folders are added as unlabeled images.

☑ Automatically attach labels to images based on the folder name

Figure 11.6: *Provide S3 URI and select automatic labeling*

10. This takes you to the data labeling page, where you can review the dataset, label it if needed and initiate model training. Refer to *Figure 11.7*:



You have enough labeled images to train a model.

- You can improve the quality of your model by adding more labeled images.
- Unlabeled images aren't used for training.
- Click 'Train model' above to start training a model.

Filters

● All images (92)
○ Labeled (92)
○ Unlabeled (0)

Normal (41)
Anomaly (51)

Anomaly labels

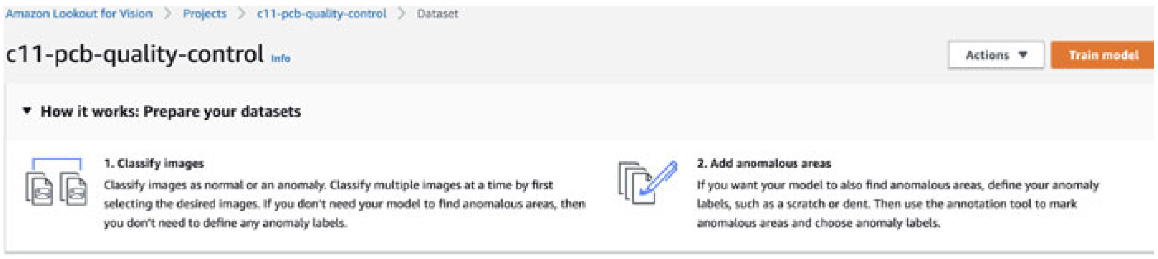
No anomaly labels defined yet.
Add anomaly labels only if you want your model to find anomalous areas.

Images (27)

000.JPG	001.JPG	002.JPG
Anomaly	Anomaly	Anomaly

Figure 11.7: Review dataset

11. Now click on the **`Train model`** button in the top-right corner of the screen to start model training. Refer to *Figure 11.8*:



Amazon Lookout for Vision > Projects > c11-pcb-quality-control > Dataset

c11-pcb-quality-control Info

Actions ▼ | Train model

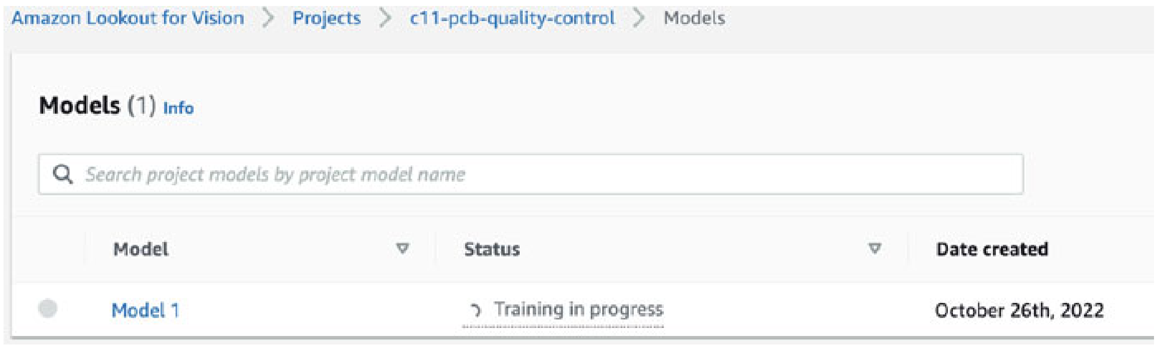
▼ How it works: Prepare your datasets

1. Classify images
Classify images as normal or an anomaly. Classify multiple images at a time by first selecting the desired images. If you don't need your model to find anomalous areas, then you don't need to define any anomaly labels.

2. Add anomalous areas
If you want your model to also find anomalous areas, define your anomaly labels, such as a scratch or dent. Then use the annotation tool to mark anomalous areas and choose anomaly labels.

***Figure 11.8:** Click the Train model button*

12. In the **`Train model`** page, leave the defaults as they are and click on the **`Train model`** button at the bottom of the page. Accept the confirmation popup to start model training.
13. The model now starts training, and you are navigated to the **`Models`** page. Refer to *Figure 11.9*:



Amazon Lookout for Vision > Projects > c11-pcb-quality-control > Models

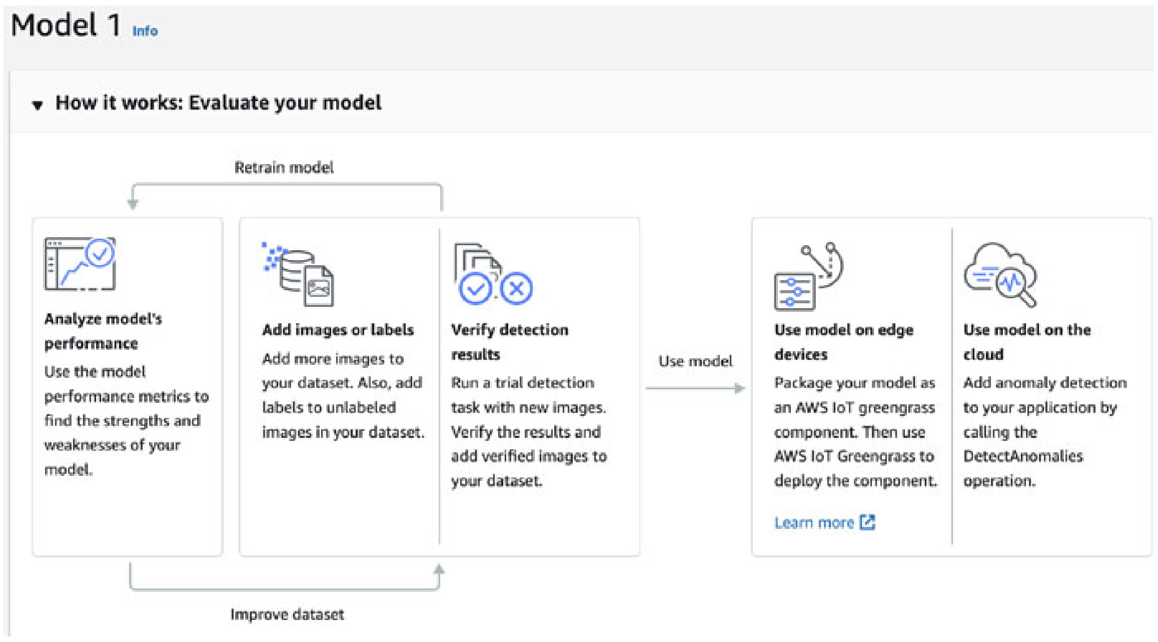
Models (1) Info

Search project models by project model name

	Model	Status	Date created
○	Model 1	Training in progress	October 26th, 2022

***Figure 11.9:** Model training progress*

14. The training will take about 30 minutes. It's time for a coffee or tea break!
15. When the training completes, the **`Models`** page is updated to display the model metrics, such as **`Precision`**, which is calculated as True Positive/(True Positive + False Positive), and **`Recall`**, which is calculated as True Positive/(True Positive + False Negative). Click on **`Model 1`** to go to the details page for our model. You can see the model evaluation process here. Refer to *Figure 11.10*:



Model 1 Info

How it works: Evaluate your model

Retrain model

Analyze model's performance
Use the model performance metrics to find the strengths and weaknesses of your model.

Add images or labels
Add more images to your dataset. Also, add labels to unlabeled images in your dataset.

Verify detection results
Run a trial detection task with new images. Verify the results and add verified images to your dataset.

Use model

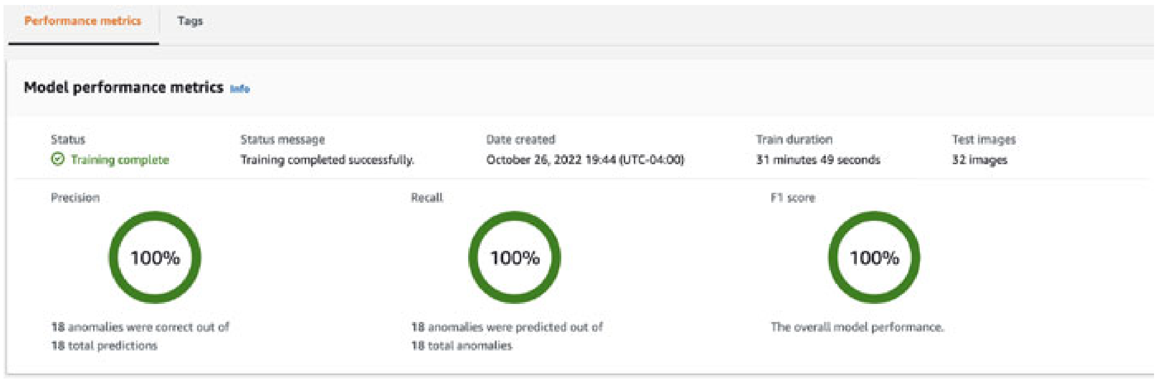
Use model on edge devices
Package your model as an AWS IoT greengrass component. Then use AWS IoT Greengrass to deploy the component.
Learn more

Use model on the cloud
Add anomaly detection to your application by calling the DetectAnomalies operation.

Improve dataset

Figure 11.10: *Model evaluation process*

16. Scroll down a bit to come to the Performance metrics panel, where you can see the details of your model for Precision, Recall, and the FI score, which is calculated as *(2*Precision*Recall)/(Precision+Recall)*. Refer to *Figure 11.11*:



Model performance metrics Info

Status	Status message	Date created	Train duration	Test images
Training complete	Training completed successfully.	October 26, 2022 19:44 (UTC-04:00)	31 minutes 49 seconds	32 images

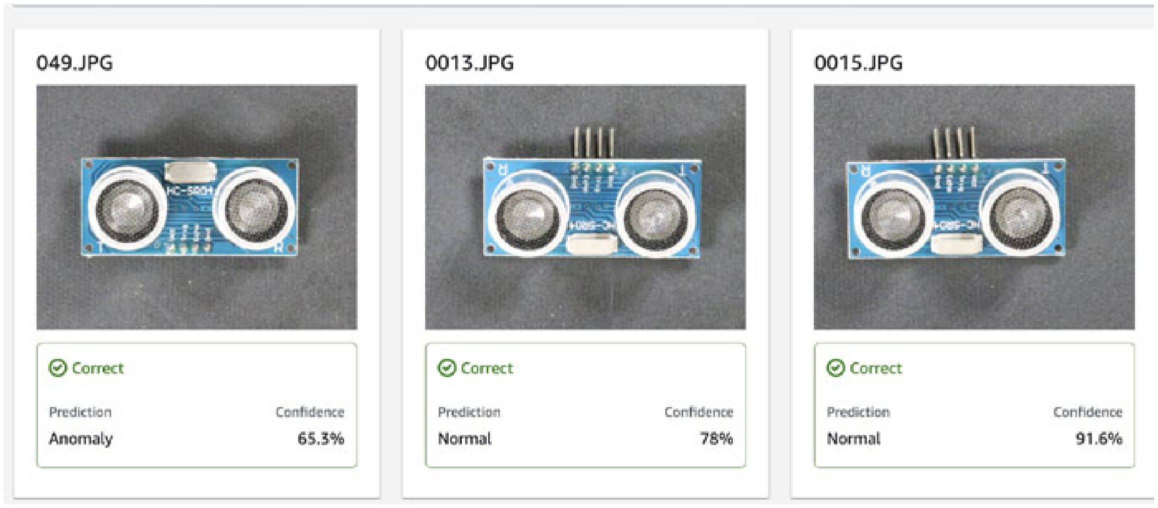
Precision
100%
18 anomalies were correct out of 18 total predictions

Recall
100%
18 anomalies were predicted out of 18 total anomalies

F1 score
100%
The overall model performance.

Figure 11.11: *Model performance*

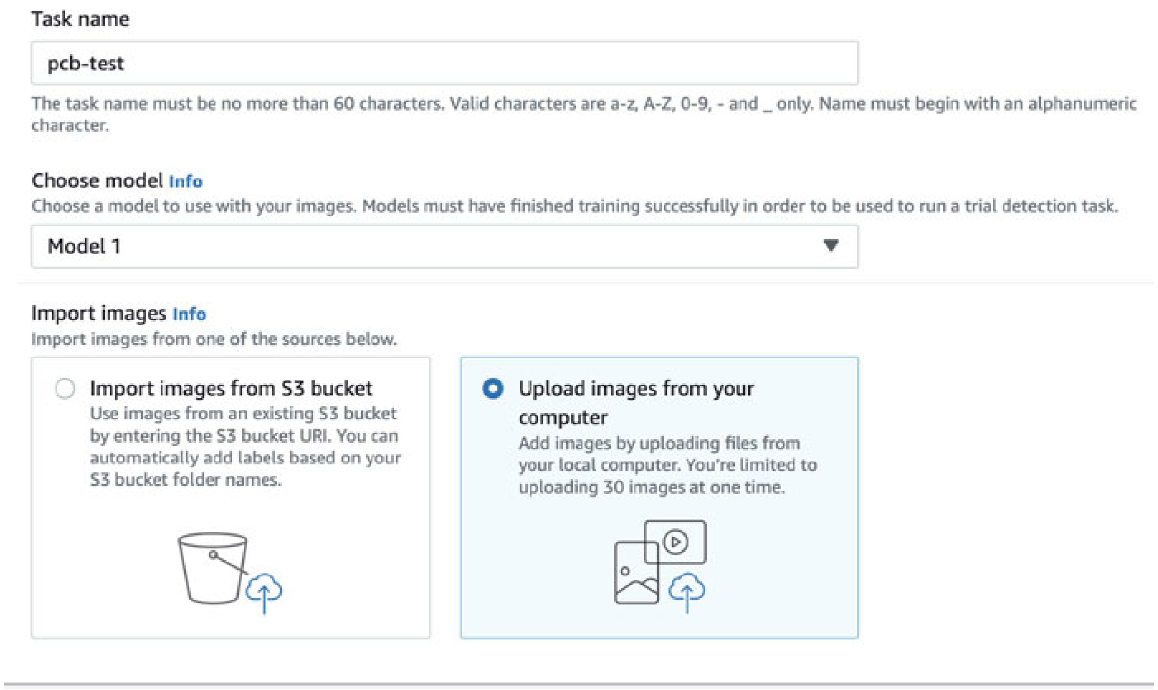
17. It is expected that we will get a 100% for Precision, Recall, and F1 because we used the same dataset for training and testing. This is for experimentation purposes, but if you want to evaluate the model with a blind test dataset, you should create a separate test dataset and choose the train and test option when we created the dataset.
18. When you scroll down, the model shows the prediction results on the test images. Refer to Figure 11.12:



049.JPG	0013.JPG	0015.JPG
Correct	Correct	Correct
Prediction: Anomaly	Prediction: Normal	Prediction: Normal
Confidence: 65.3%	Confidence: 78%	Confidence: 91.6%

Figure 11.12: *Model test results*

19. Now scroll up the page and click on the **`Run trial detection`** button. We will use this to test with a few blind test images, both normal and anomalous, that the model has not seen before. Download these images from our GitHub repository at **https://github.com/garchangel/AIMLwithAWS/tree/main/Chapter-11/l4v-blind-test**.
20. In the **`Run trial detection`** page, give the Task name as **`pcb-test`**, leave the **`Choose model`** field selection as is, and select the **`Upload images from your computer`** option. Refer to *Figure 11.13*:



Task name

pcb-test

The task name must be no more than 60 characters. Valid characters are a-z, A-Z, 0-9, - and _ only. Name must begin with an alphanumeric character.

Choose model Info

Choose a model to use with your images. Models must have finished training successfully in order to be used to run a trial detection task.

Model 1

Import images Info

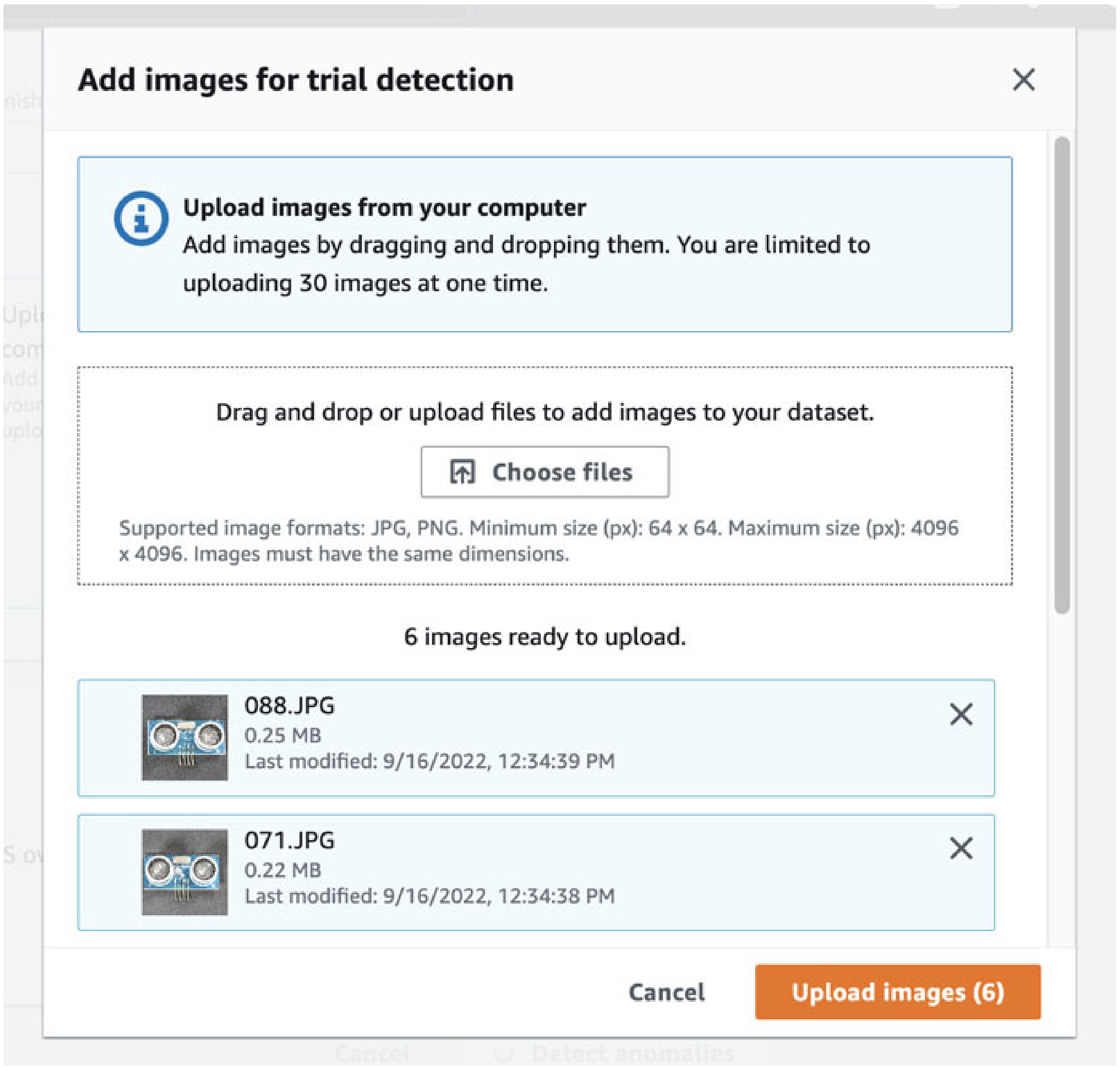
Import images from one of the sources below.

Import images from S3 bucket
Use images from an existing S3 bucket by entering the S3 bucket URI. You can automatically add labels based on your S3 bucket folder names.

[✓ Upload images from your computer]
Add images by uploading files from your local computer. You're limited to uploading 30 images at one time.

Figure 11.13: *Run trial detection - select images*

21. Click on the **`Detect anomalies`** button and accept the prompt by clicking on the **`Run trial detection`** button.
22. In the **`Add images for trial detection`** popup, click on **`Choose files`** and select the six images you downloaded from the GitHub repository in Step 19.
23. When the image previews are loaded, scroll down to review them and then click on the **`Upload images`** button. Refer to *Figure 11.14*:



Add images for trial detection

Upload images from your computer
Add images by dragging and dropping them. You are limited to uploading 30 images at one time.

Drag and drop or upload files to add images to your dataset.

Choose files

Supported image formats: JPG, PNG. Minimum size (px): 64 x 64. Maximum size (px): 4096 x 4096. Images must have the same dimensions.

6 images ready to upload.

088.JPG
0.25 MB
Last modified: 9/16/2022, 12:34:39 PM

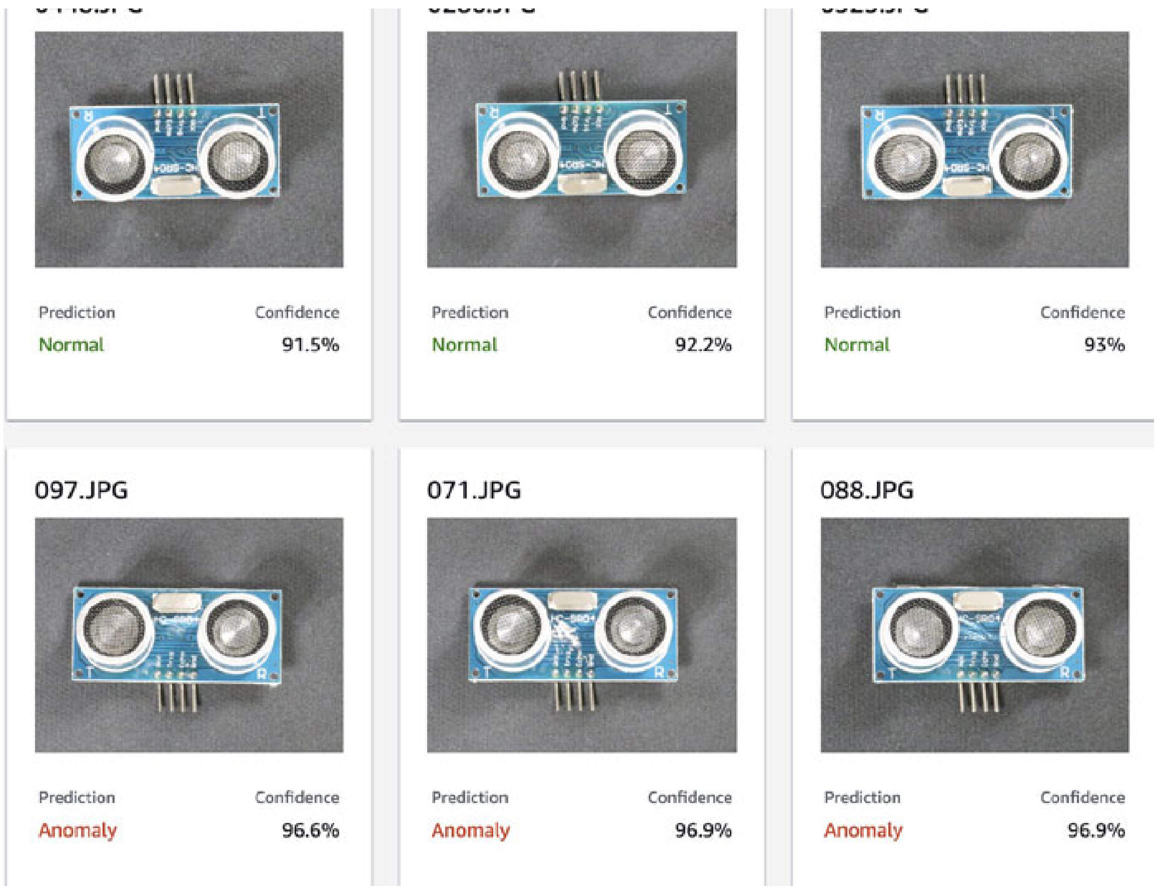
071.JPG
0.22 MB
Last modified: 9/16/2022, 12:34:38 PM

Cancel
Upload images (6)

Figure 11.14: *Select images for trial detection*

24. Lookout for Vision will navigate you to the **`Verify machine predictions`** page in **`Trial detections`** for your model. Once it is completed, the prediction results for each class type (normal or anomalous) are displayed.

As you can see, our model predicted the results accurately. Refer to *Figure 11.15*:


097.JPG

071.JPG

088.JPG

Prediction	Confidence
Normal	91.5%
Normal	92.2%
Normal	93%
Anomaly	96.6%
Anomaly	96.9%
Anomaly	96.9%

Figure 11.15: *Results of a trial prediction run*

25. When you are done and happy with your model's performance, you can deploy it as an API in the cloud, or you can create a model package and deploy it over the air to an edge device using AWS Greengrass IoT (**https://aws.amazon.com/greengrass/**). AWS Greengrass IoT is a cloud and on-edge AWS service for building, deploying and managing your IoT device fleets, enable MQTT integration between on-edge, and cloud, and perform over-the-air deployments of software to your edge devices, with low latency, high reliability, and at scale. With Greengrass IoT, you can train your models in the cloud and deploy them to your edge devices for inference, which helps with low latency inference, the ability for models to be closer to data sources, and for data privacy requirements. Refer to *Figure 11.16*:

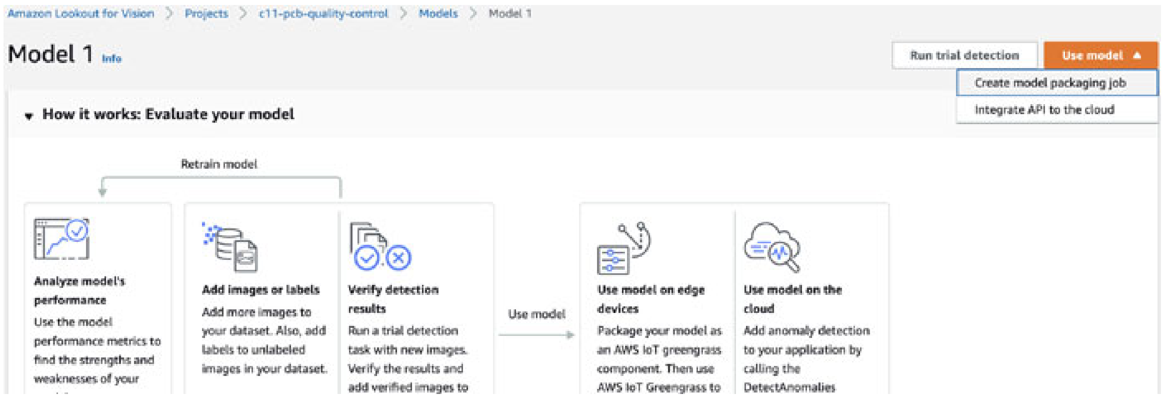

Amazon Lookout for Vision > Projects > c11-pcb-quality-control > Models > Model 1
Model 1 Info
Run trial detection
Use model
Create model packaging job
Integrate API to the cloud
How it works: Evaluate your model
Retrain model
Analyze model's performance
Use the model performance metrics to find the strengths and weaknesses of your
Add images or labels
Add more images to your dataset. Also, add labels to unlabeled images in your dataset.
Verify detection results
Run a trial detection task with new images. Verify the results and add verified images to
Use model
Use model on edge devices
Package your model as an AWS IoT greengrass component. Then use AWS IoT Greengrass to
Use model on the cloud
Add anomaly detection to your application by calling the DetectAnomalies

Figure 11.16: *Model usage*

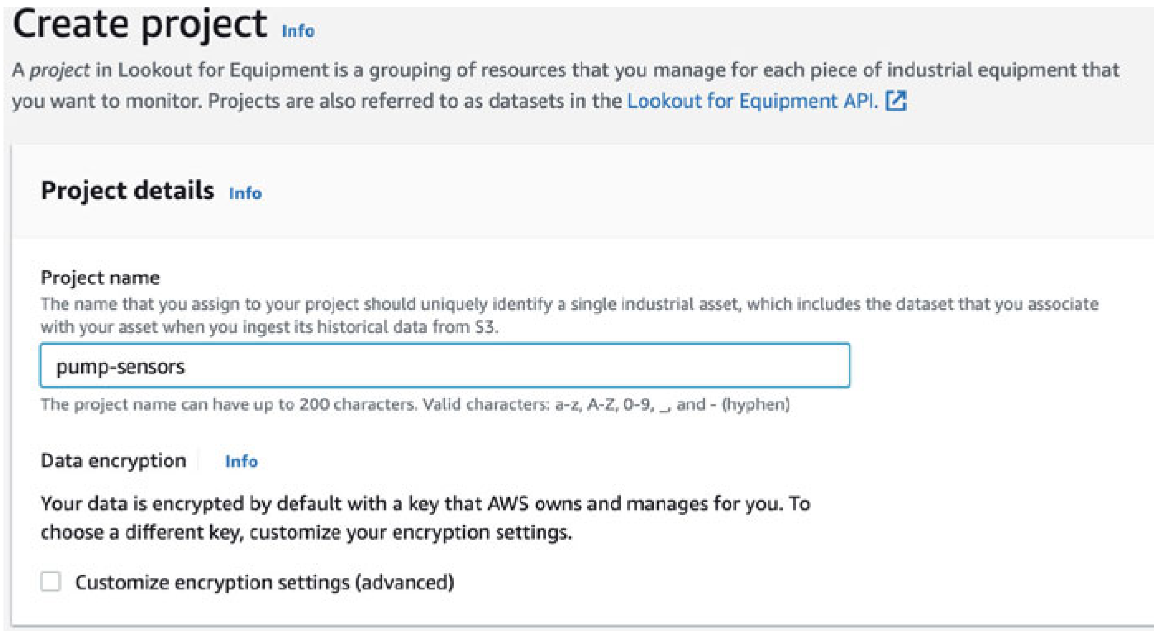
And that brings us to the end of this section on Amazon Lookout for Vision. As you saw, with just a few clicks and page navigations, and a limited dataset, we were able to train a powerful computer vision model for anomaly detection. Considering this model can be used in the cloud or also on edge in factory floors and manufacturing facilities, it can improve the operational efficiency of production/manufacturing for your organization pretty easily (helping save months of time and a significant amount of funds if you have to train and tune your own model from the ground up). In the next section, we will go through a similar service for detecting problematic equipment behaviour.

Predictive Analytics with Amazon Lookout for Equipment

In this section, we will train a model to detect faulty equipment behavior using Amazon Lookout for Equipment. Before we get started, we need to ensure that our pre-requisites are in place. In case you haven't signed up for an AWS account yet, now is the time to do so. Follow the instructions in the Setting up your AWS account section in *Chapter 2, Hydrating Your Data Lake*, to sign up for an AWS account. Once you have signed up, log in to your AWS account using the instructions at **https://docs.aws.amazon.com/IAM/latest/UserGuide/console.html**. If you have not already done so, create an Amazon S3 bucket (**https://docs.aws.amazon.com/AmazonS3/latest/userguide/create-bucket-overview.html**). Execute the following instructions to build the model:

1. First, download the training dataset from our GitHub repository **https://github.com/garchangel/AIMLwithAWS/blob/main/Chapter-11/l4e-train-data.csv**. The original source for this data is **https://www.kaggle.com/datasets/nphantawee/pump-sensor-data**.

2. The CSV file that is provided to you for this solution contains time series data for 3.5 months, with a data point for every minute for 10 sensors.
3. Log in to your AWS Management Console if not already done, and type S3 to bring up the Amazon S3 console. Navigate to the bucket you created, click on **`Create folder`**, and provide the folder name as l4e-training. Click on the **`l4e-training`** folder name to go into the folder, click on the **`Create folder`** button in the top-right corner once again, and create a new folder called **`compressor-room-1`**. Now, click on the **`Upload`** button and load the CSV file you downloaded from our GitHub repository in *Step 1* into the **`compressor-room-1`** folder. Make a note of the S3 URI for your CSV file.
4. Now type equipment in the services search bar and select **`Amazon Lookout for Equipment`** to be navigated to the console.
5. In the **`Lookout for Equipment`** console, click on the **`Create project`** button.
6. In the **`Create project`** screen, in the Project details pane, type **`pump-sensors`** for the **`Project name`** field. Refer to *Figure 11.17*:



Create project Info

A *project* in Lookout for Equipment is a grouping of resources that you manage for each piece of industrial equipment that you want to monitor. Projects are also referred to as datasets in the Lookout for Equipment API.

Project details Info

Project name

The name that you assign to your project should uniquely identify a single industrial asset, which includes the dataset that you associate with your asset when you ingest its historical data from S3.

pump-sensors

The project name can have up to 200 characters. Valid characters: a-z, A-Z, 0-9, _, and - (hyphen)

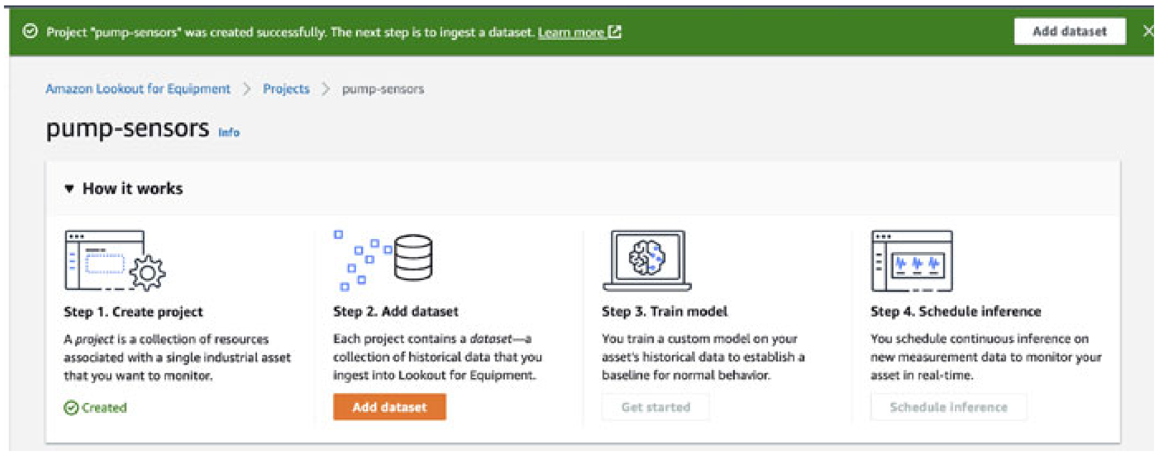
Data encryption Info

Your data is encrypted by default with a key that AWS owns and manages for you. To choose a different key, customize your encryption settings.

☐ Customize encryption settings (advanced)

Figure 11.17: *Create Lookout for Equipment project*

7. Leave the rest of the default values as they are, scroll down, and click on the **`Create project`** button. After the project is created, you will be navigated back to the **`Project details`** page for your project with a status message, and the page will also display a process flow for building your model. Our next step would be to add the dataset for model training. Click on the **`Add dataset`** button. Refer to *Figure 11.18*:


Project "pump-sensors" was created successfully. The next step is to ingest a dataset. Learn more

Add dataset

Amazon Lookout for Equipment > Projects > pump-sensors

pump-sensors Info

How it works

Step 1. Create project

A *project* is a collection of resources associated with a single industrial asset that you want to monitor.

Created

Step 2. Add dataset

Each project contains a *dataset*—a collection of historical data that you ingest into Lookout for Equipment.

Add dataset

Step 3. Train model

You train a custom model on your asset's historical data to establish a baseline for normal behavior.

Get started

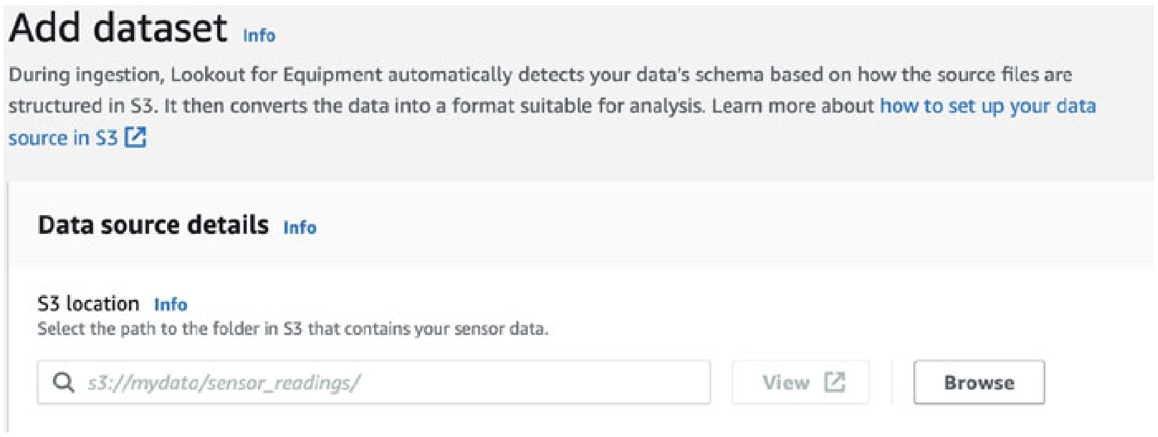
Step 4. Schedule inference

You schedule continuous inference on new measurement data to monitor your asset in real-time.

Schedule inference

Figure 11.18: *Start adding a dataset after project creation*

8. In the **`Add dataset`** page, under **`Data source details`**, click on the **`Browse`** button. Refer to *Figure 11.19*:


Add dataset Info

During ingestion, Lookout for Equipment automatically detects your data's schema based on how the source files are structured in S3. It then converts the data into a format suitable for analysis. Learn more about how to set up your data source in S3

Data source details Info

S3 location Info

Select the path to the folder in S3 that contains your sensor data.

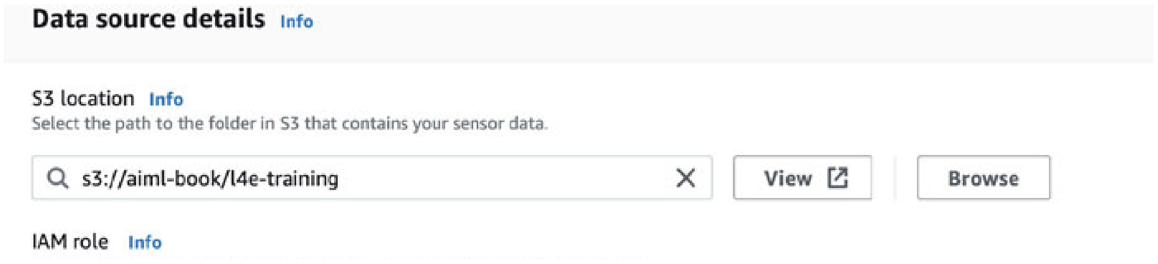
s3://mydata/sensor_readings/

View

Browse

Figure 11.19: *Click the Browse button for S3*

9. In the **`Choose Bucket/Prefix`** popup that appears, select your S3 bucket name, and select the **`l4e-training`** folder (which is the top folder in the hierarchy for our solution) that contains the compressor-room-1 folder, which, in turn, contains the CSV file you uploaded in *Step 2*. Leave the IAM role selection as it is. In the **`Schema detection`** method, select **`By folder name`**. Scroll down and click on the **`Start ingestion`** button. Refer to *Figure 11.20*:


Data source details Info

S3 location Info

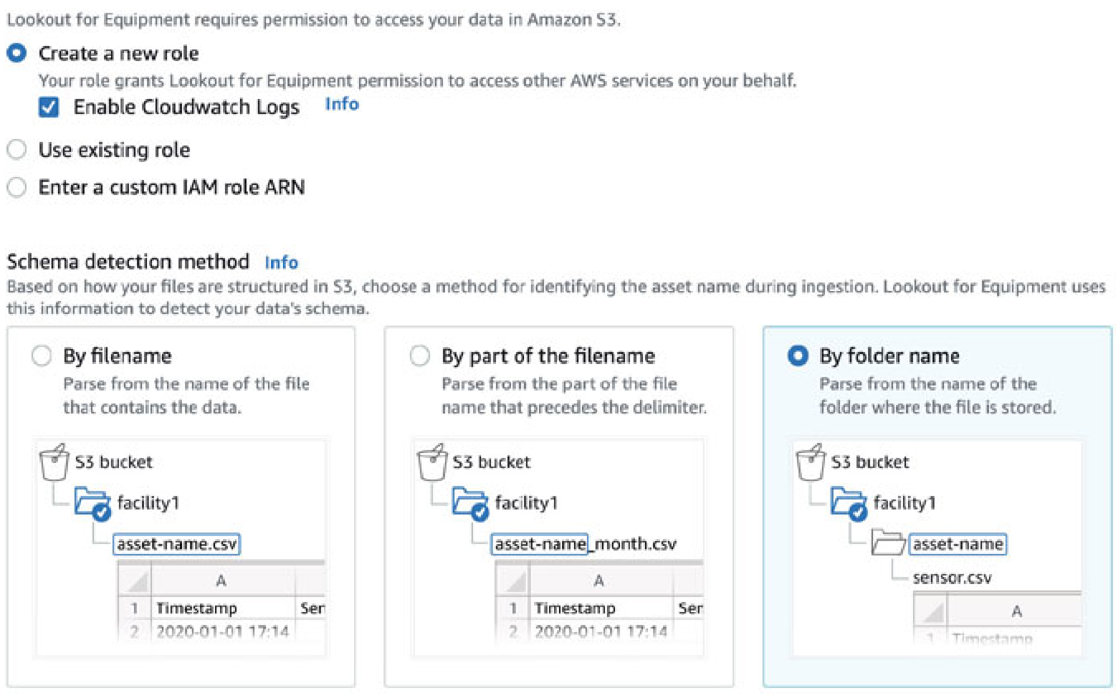
Select the path to the folder in S3 that contains your sensor data.

s3://aiml-book/l4e-training

View

Browse

IAM role Info


Lookout for Equipment requires permission to access your data in Amazon S3.

[✓ Create a new role]
Your role grants Lookout for Equipment permission to access other AWS services on your behalf.
[✓ Enable Cloudwatch Logs] Info

Use existing role

Enter a custom IAM role ARN

Schema detection method Info
Based on how your files are structured in S3, choose a method for identifying the asset name during ingestion. Lookout for Equipment uses this information to detect your data's schema.

By filename
Parse from the name of the file that contains the data.

By part of the filename
Parse from the part of the file name that precedes the delimiter.

[✓ By folder name]
Parse from the name of the folder where the file is stored.

Figure 11.20: *Add dataset for ingestion*

10. Once the IAM role has been propagated, you are navigated back to the `Project details` page with a status on the ingestion process. This will take up to 20 minutes. You get a success message once the ingestion completes. Refer to *Figure 11.21*:

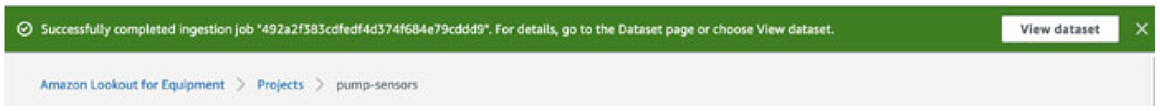
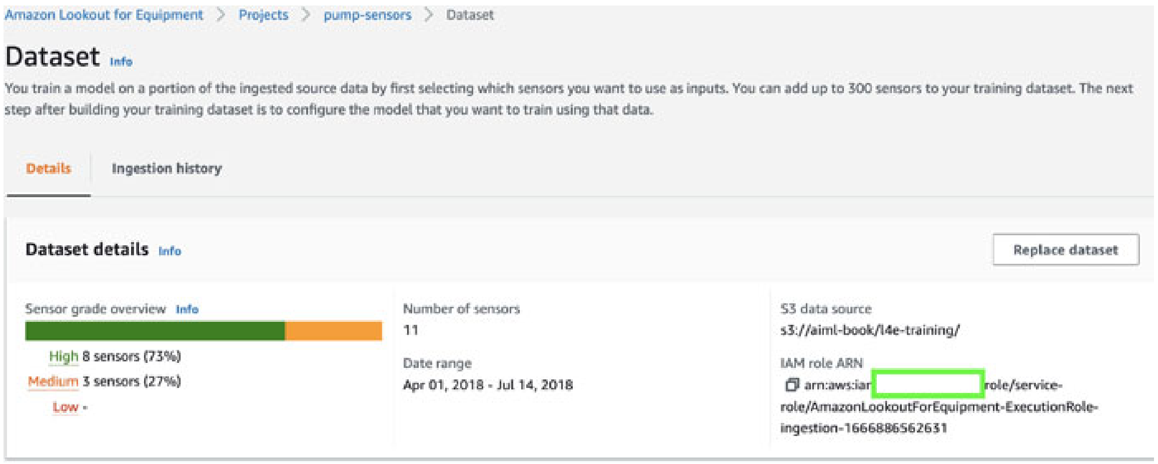

Successfully completed ingestion job "492a2f383cdfedf4d374f684e79cddd9". For details, go to the Dataset page or choose View dataset.

Figure 11.21: *Lookout for Equipment data ingestion successful*

11. Click on the `View dataset` button either in the success message or in the `How it works` section to check the metrics for dataset import. Lookout for Equipment provides a detailed analysis of your dataset and surfaces data quality in the data distributions into High, Medium, and Low categories. Low indicates significant issues with the data and cannot be used for model training, high indicates good quality data for training, and medium indicates outliers in the data. Refer to *Figure 11.22*:



Amazon Lookout for Equipment > Projects > pump-sensors > Dataset

Dataset Info

You train a model on a portion of the ingested source data by first selecting which sensors you want to use as inputs. You can add up to 300 sensors to your training dataset. The next step after building your training dataset is to configure the model that you want to train using that data.

Details | Ingestion history

Dataset details Info

Sensor grade overview Info

High 8 sensors (73%)
Medium 3 sensors (27%)
Low -

Number of sensors
11

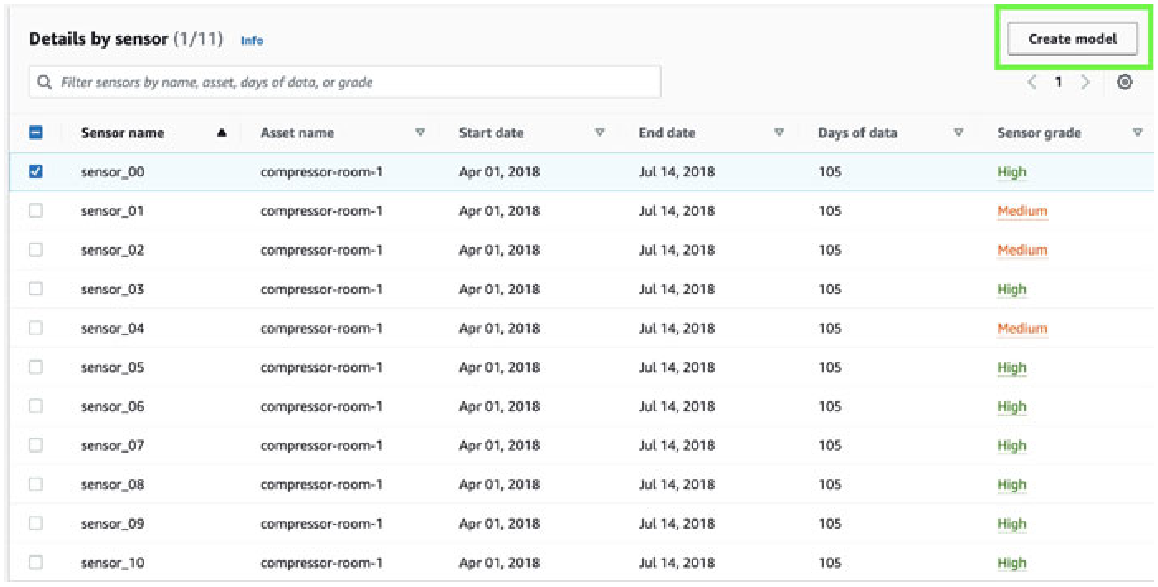
Date range
Apr 01, 2018 - Jul 14, 2018

S3 data source
s3://aiml-book/l4e-training/

IAM role ARN
arn:aws:iar[redacted]role/service-role/AmazonLookoutForEquipment-ExecutionRole-ingestion-1666886562631

Figure 11.22: Data ingestion successful in Lookout for Equipment

12. Scroll down a little to the **`Details by sensor`** pane, select the check box to the left of **`sensor_00`** (the first row), and click **`Create model`** button on the right of the pane. Lookout for Equipment allows you to create one model for all your sensors; you can create a separate model for each sensor, or you can mix and match sensors across models. Depending on your industrial process, you can choose an approach that works best. For our example, we will train a model for **`sensor_00`** and validate the results. Refer to *Figure 11.23*:



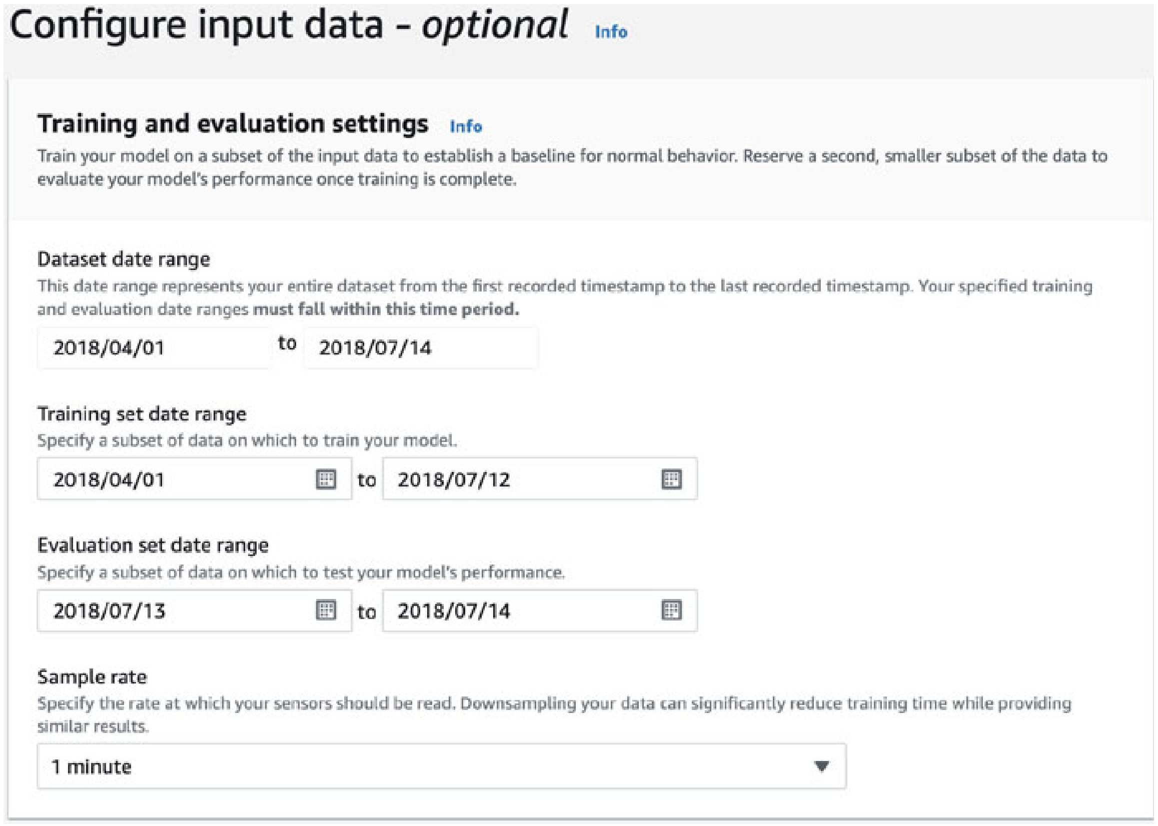
Details by sensor (1/11) Info

	Sensor name	Asset name	Start date	End date	Days of data	Sensor grade
✓	sensor_00	compressor-room-1	Apr 01, 2018	Jul 14, 2018	105	High
	sensor_01	compressor-room-1	Apr 01, 2018	Jul 14, 2018	105	Medium
	sensor_02	compressor-room-1	Apr 01, 2018	Jul 14, 2018	105	Medium
	sensor_03	compressor-room-1	Apr 01, 2018	Jul 14, 2018	105	High
	sensor_04	compressor-room-1	Apr 01, 2018	Jul 14, 2018	105	Medium
	sensor_05	compressor-room-1	Apr 01, 2018	Jul 14, 2018	105	High
	sensor_06	compressor-room-1	Apr 01, 2018	Jul 14, 2018	105	High
	sensor_07	compressor-room-1	Apr 01, 2018	Jul 14, 2018	105	High
	sensor_08	compressor-room-1	Apr 01, 2018	Jul 14, 2018	105	High
	sensor_09	compressor-room-1	Apr 01, 2018	Jul 14, 2018	105	High
	sensor_10	compressor-room-1	Apr 01, 2018	Jul 14, 2018	105	High

Figure 11.23: Create a model for Sensor 00

13. On the **`Model details`** page, specify **`sensor_00_model`** for the **`Model name`**, scroll down, and click on the **`Next`** button.

14. In the **`Configure input data`** page, for **`Training and evaluation settings`**, for the **`Training set date range`**, provide **`2018/04/01 to 2018/07/12`**. For the **`Evaluation set date range`**, provide **`2018/07/13 to 2018/07/14`**. Leave the rest of the default settings as they are, scroll down, and click on the **`Next`** button. Refer to *Figure 11.24*:



Configure input data - *optional* Info

Training and evaluation settings Info

Train your model on a subset of the input data to establish a baseline for normal behavior. Reserve a second, smaller subset of the data to evaluate your model's performance once training is complete.

Dataset date range

This date range represents your entire dataset from the first recorded timestamp to the last recorded timestamp. Your specified training and evaluation date ranges **must fall within this time period.**

2018/04/01 to 2018/07/14

Training set date range

Specify a subset of data on which to train your model.

2018/04/01 to 2018/07/12

Evaluation set date range

Specify a subset of data on which to test your model's performance.

2018/07/13 to 2018/07/14

Sample rate

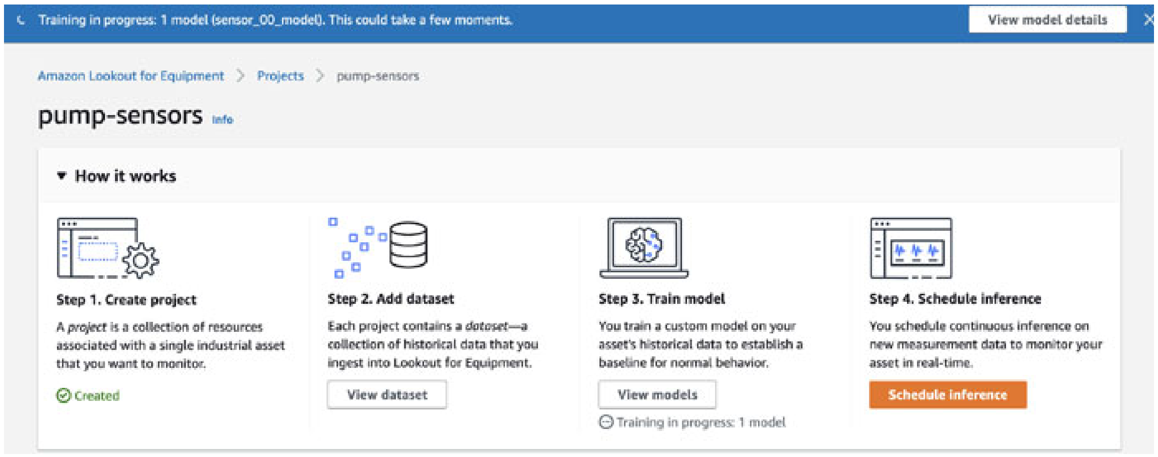
Specify the rate at which your sensors should be read. Downsampling your data can significantly reduce training time while providing similar results.

1 minute

Figure 11.24: *Configure input data for model training*

15. Leave the **`Provide data labels`** page as it is, scroll down and click on the **`Next`** button. Review your inputs on the **`Review and train`** page, scroll down and click on the **`Start training`** button.

16. You are navigated back to the **`Project details`** page, and the **`Training in progress`** message is displayed in the **`How it works`** pane, **`Step 4. Schedule inference`** is highlighted. Once the model training is completed, your model will be ready to receive sensor data in real time to detect anomalies. You can also schedule periodic inference requests. Refer to *Figure 11.25*:


Training in progress: 1 model (sensor_00_model). This could take a few moments.

Step 1. Create project

A *project* is a collection of resources associated with a single industrial asset that you want to monitor.

Created

Step 2. Add dataset

Each project contains a *dataset*—a collection of historical data that you ingest into Lookout for Equipment.

Step 3. Train model

You train a custom model on your asset's historical data to establish a baseline for normal behavior.

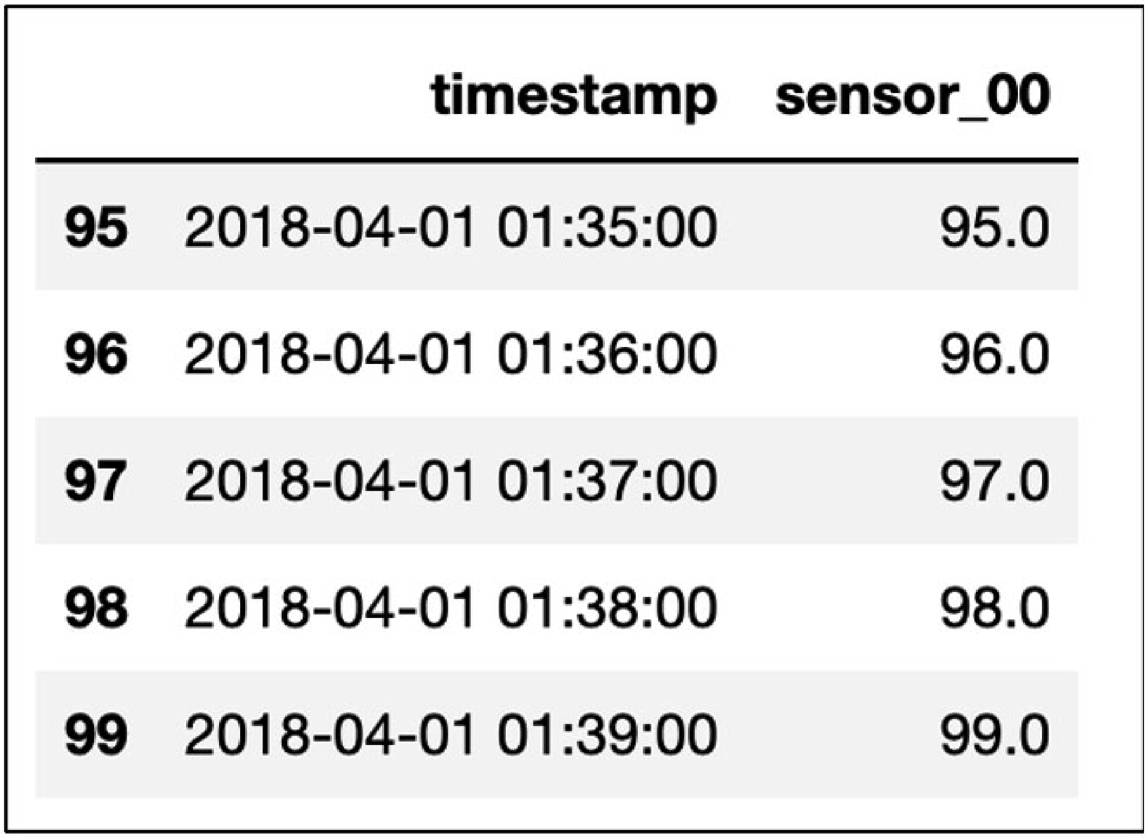
Training in progress: 1 model

Step 4. Schedule inference

You schedule continuous inference on new measurement data to monitor your asset in real-time.

Figure 11.25: *Training in progress*

17. Once the training completes, click on the **`View models`** button in the **`How it works`** pane.
18. Click on the name of your model (**`sensor_00_model`**) to go to the **`Model overview`** page. Scroll down to the **`Evaluation results`** section; you should see that the model trained successfully and did not find any anomalies in the evaluation dataset for the time range we specified in *Step 12*.
19. In a new browser tab, go to the GitHub link **https://github.com/garchangel/AIMLwithAWS/blob/main/Chapter-11/sensor00-anomalous.csv** and download the file to your local computer. We synthetically created this file to test the Lookout for Equipment inference. The file contains 50 normal data points and 50 anomalous data points. Find a snapshot of the anomalous data file. Refer to *Figure 11.26*:



	timestamp	sensor_00
95	2018-04-01 01:35:00	95.0
96	2018-04-01 01:36:00	96.0
97	2018-04-01 01:37:00	97.0
98	2018-04-01 01:38:00	98.0
99	2018-04-01 01:39:00	99.0

Figure 11.26: *Example of an anomalous sensor readings*

20. Lookout for Equipment works with time series data, so the inference runs in real time on durations that we define in the inference scheduler.

    NOTE: To test the inference, you need to update the timestamp of the inference file to reflect the date and time of 5 minutes in the future when you are running the test. You will have to run the Python code provided in Step 25 to do this. Use the SageMaker Studio notebook to paste and execute this code.

21. Once the file is downloaded, go back to your S3 bucket that you used for data ingestion (what you used in Step 3) and create a new folder (outside the **`l4e-training`** folder you created earlier) called **`l4e-inference`**.
22. Now go back to your **`Model overview`** page and click on the **`Schedule inference`** button in the top-right corner.
23. In the **`Schedule inference`** page, provide a value for the **`Inference schedule name`** called **`sensor_00_inference`**.
24. For **`Input data`**, click on the **`Browse`** button and select the **`l4e-inference`** folder from your S3 bucket.

25. For **`Data upload frequency`**, select 5 minutes.
26. Leave the rest of the default selections as they are, and scroll down to **`Output data`**. For the S3 location, provide **`s3://<bucket-name>/l4e-inference-output`**. Scroll down to the bottom of the page and click on the **`Schedule inference`** button.
27. Immediately after you schedule inference, run the code snippet to create a new anomalous dataset to reflect a date and time 5 minutes in your future at 1 minute intervals for Sensor00. The code will also upload your inference CSV file to your S3 bucket into the folder you created for inference. Refer to the following code snippet.

```
import datetime
from datetime import timezone
import pandas as pd
import boto3
bucket = 'aiml-book'
s3 = boto3.client('s3')
starttime = datetime.timedelta(minutes=5)
dated = datetime.datetime.now(timezone.utc)+starttime
cols = ['timestamp','sensor_00']
time_df = pd.DataFrame()
for i in range(50):
 dated += datetime.timedelta(minutes=1)
 new = dated.strftime('%Y%m%d%H%M%S')
 time_df.at[i,'timestamp'] = new
 time_df.at[i,'sensor_00'] = 50+i
time_df.to_csv('inference.csv', index=False)
s3.upload_file('inference.csv', bucket, 'l4e-
inference/inference.csv')
time_df.head()
```

Note: You may encounter errors in your inference scheduler if time elapses between when you execute this code and when the inference scheduler is activated. From the time the scheduler is active, it expects time series datapoints for 5 minutes, in 1-minute intervals. To mitigate, implement this preceding code using an AWS Lambda function that is triggered every minute by an AWS EventBridge trigger, and which updates the inference.csv file continuously. Please refer to https://docs.aws.amazon.com/eventbridge/latest/userguide/eb-run-lambda-schedule.html.

28. After 5 minutes, Lookout for Equipment will automatically pick up the file, run inference using the model and store the results in the output folder. You can open the output folder and go through the file to see how many anomalous data points were detected.

And with that, we conclude this chapter on how you can use AWS AI services for industrial automation, cost reduction, efficiency improvement, and overall profitability enhancement. We covered two major AI services for industrial automation in this chapter, but that is just the tip of the iceberg. For massive-scale computer vision at the edge with large fleets of cameras for industrial monitoring, you can use AWS Panorama (**https://aws.amazon.com/panorama/**), or if you like a turnkey solution for equipment monitoring that includes sensors, gateways, models, and monitoring capabilities, you can use Amazon Monitron (**https://aws.amazon.com/monitron/**). When you look at business automation in addition to industrial automation, you can use services like Lookout for Metrics (**https://aws.amazon.com/lookout-for-metrics/**), Amazon Forecast (**https://aws.amazon.com/forecast/**), and Amazon Fraud Detector (**https://aws.amazon.com/fraud-detector/**).

Conclusion

In this chapter, we reviewed more AWS AI services but for use with industrial automation, predictive analytics and quality control inspections. There are more AWS AI services than we were able to cover in this chapter and the previous one. For a full list of services, you can refer to **https://aws.amazon.com/machine-learning/ai-services/**. In the next chapter (also the final chapter of this book), we will learn how to build operationalized ML workflows built for scale, reliability, and efficiency using AWS MLOps concepts and services such as Amazon SageMaker Pipelines, AWS StepFunctions and more.

Points to Remember

In this chapter, we were introduced to and learned how to build solutions for industrial automation using the AWS AI services layer. Specifically, we covered the following topics:

- We learned about the importance of monitoring machinery, predictive analytics, continuous improvement, and automation in the context of manufacturing industries that typically have factories with equipment and machinery involved in production.

- We learned the significance of quality control and equipment failure prediction in the context of operational costs and understood how it impacts a company's profitability.
- We discussed the cost of poor quality and how AWS AI services for industrial automation, such as Amazon Lookout for Vision and Amazon Lookout for Equipment, can help build powerful ML models for improving product quality and proactively mitigate equipment issues, with no ML expertise or skills required.
- We then selected an example of printed circuit board datasets and built a computer vision-based quality inspection model using Amazon Lookout for Vision to automatically detect anomalous printed circuit boards.
- We completed the end-to-end ML process directly from the AWS console, without writing a single line of code or performing any of the undifferentiated data science tasks.
- Finally, we picked a dataset containing time series data from multiple sensors in a compressor pump, trained a predictive analytics model using Amazon Lookout for Equipment to predict sensor faults, and scheduled real-time inference to validate the model, also using just the AWS console and without writing code or running feature engineering tasks.

Multiple Choice Questions

Use these questions to challenge your knowledge of AWS AI services for industrial automation.

1. **How can you use AI/ML for industrial automation? (Select two correct answers.)**
    a. It can help alert maintenance when a machine breaks down.
    b. It can predict when a machine will break down.
    c. It is an engine that keeps the assembly line moving.
    d. It can automatically inspect faulty products and help mitigate quality issues.
2. **What type of data does Lookout for Equipment need for model training?**
    a. Images
    b. Structured text data
    c. Unstructured text or tabular data

   d. Time series data

3. **What type of machine learning does Lookout for Vision use?**
   a. Reinforcement learning
   b. Supervised learning
   c. Unsupervised learning
   d. Generative learning
4. **Lookout for Equipment can train only a single model across all sensors.**
   a. True
   b. False
5. **What is the formula to calculate Precision in classification models?**
   a. True Positive/(True Positive + True Negative)
   b. True Positive/(True Positive + False Positive)
   c. True Positive/(True Positive + False Negative)
   d. True Positive/False Positive

Answers

1. **b, d**
2. **d**
3. **b**
4. **b**
5. **b**

CHAPTER 12

Operationalized Model Assembly (MLOps and Best Practices)

Introduction

Congratulations, you are now an expert in knowing and applying Artificial Intelligence, Machine Learning using AWS for your enterprise use cases. This means you know how to approach, design, train, build, deploy, and monitor your AI/ML solutions. This means that you are ready for the final step in your end-to-end Machine Learning journey: operationalization. Apart from being a lengthy Scrabble-winning word, it is the difference between a manual solution and an automated well-architected workload. Implementing Machine Learning Operations, also known as MLOps colloquially, prepares your project for production deployment while implementing an automated Continuous Integration (CI) and Continuous Delivery (CD) pipeline. This accelerates your ongoing experiments and time-to-market cycles and empowers the contributions of your data science teams. To try the examples in this section, refer to the *Technical Requirements* section in *Chapter 1, Introducing the ML Workflow*, to sign into the AWS management console, execute the steps to onboard to SageMaker studio, and execute cloning the repository to SageMaker Studio to get started. Click on the folder that corresponds to this chapter number. If you see multiple notebooks, the section title corresponds to the notebook name for easy identification. You can also passively follow the code samples using the GitHub repository provided at the beginning of the book.

Chapter Scenario

The team assigned to implement the various machine learning models has decided to adopt **Site Reliability Engineering** (**SRE**). This means that members of the team rotate out as an operations engineer, assisting the larger production operations team in deploying their models through the software delivery lifecycle environments. This works particularly well because no one knows the minute details about the operations need of a produced model more than the team that created it. Our data scientist turned SRE doesn't have much experience in operations, but they are supported by a very strong, very experienced production operations team, so they feel confident that they have the support they need to implement automated, repeatable, observable workflows for their models.

Structure

In this chapter, we will discuss the following topics:

- What is Machine Learning Operations, commonly called MLOps?
- What is the purpose of an orchestrator in terms of MLOps?
- What are the common options for orchestrators in AWS?
- The individual phases of an MLOps process.
- How to implement an Amazon SageMaker Pipeline
- How to make a Machine Learning workflow Well-Architected.

Objectives

In this chapter, we will use Amazon SageMaker pipelines and the AWS Step Functions Data Science SDK to build end-to-end MLOps pipelines. In the process, we will also learn best practices for machine learning workflows. We will cover the critical phases of a machine learning solution pipeline, starting with iterative model training and moving on to model evaluation, artifact management, and SDLC deployment. We will focus on the methods for implementation, management, observation, and mutability for each of the phases and look at how to ensure that the resultant telemetry is ingestible by both your data science and operations teams. By the end of this chapter, you should have understood how the experimentation phase of your ML tasks should translate to automated implementations and how to decouple the train-to-deploy mechanics to allow repeatability as well as extendibility.

MLOps Defined

Needless to say, the field of machine learning is evolving rapidly. A part of this process is the maturation of processes, tools, and services associated with the technology. In the previous chapters, we reviewed much of this modernization with tools that enable efficient experimentation, training, deployment, and solutioning associated with machine learning projects. The same is true for the operationalization of these projects. When we use the term operationalization, we mean taking a manual process and adding automation, controls, and visibility. This method enables repetition of a process in the absence of human interaction coordinating your chosen deployable package through the respective environments and to a production deployment.

Machine Learning Operations or MLOps is this practice. There are numerous tools to manage, automate, and orchestrate this process for you, but before you choose to use one of these, you should understand the individual phases, their unique needs, the ways in which machine learning diverges from traditional delivery processes, and the distinct monitoring needs.

In order to achieve the automation desired in the process, we can break down the steps to take your raw data and create a a trained model deployment in a production environment into five distinct phases.

The first step is the data transformation needed to take the raw data sources and transform them into the format necessary to effectively train a machine learning model. This phase is separate from the others because the ability to adapt, execute, and monitor it allows agility in cases where changes are needed apart from all others in the pipeline. We also treat the transformed data as a versioned unit and as input to the next phase.

The second phase is the actual training of your machine learning model. Like all the other phases, it takes the output of the previous phase. In this case, we retrieve our transformed data and start training a model; the training is complete when we have a model artifact to store. Like our data, we version the resulting model for the next phase.

The third phase is the evaluation of the model. This requires that we deploy the model and use data that the model has never seen to determine its suitability to address our stated business outcome. This may include accuracy, loss, response time, artifact size, RMSE, or similar metrics. These can be static, requiring you model pass a specified numerical before being saved or they can be dynamic, requiring that your potential model perform better than the model already deployed to production.

The fourth phase is the storage management of the trained models. When just starting the experimentation phase, storing models in s3 is sufficient, but as more models are trained, more teams train models, and different models move along their respective deployment paths, we need more robust solutions. Additionally, we want both ephemeral and long-term storage of models, along with the ability to act on their metadata, including version.

The fifth and final phase is the deployment of the created models. Like the other phases, this phase should be able to run without a human performing the deployment. We can, and often should, add approval steps requiring environment owners to allow deployment to proceed into their respective environments, with the actual implementation occurring automatically once approved.

Each of these phases should have a trigger that begins the automated steps of the phase itself and emits sufficient metrics to allow both the data science and operations teams to understand the status, results, and errors, if any. Triggers differ for each phase, but they can include new training data becoming available, a new training script being checked into source control, updated model evaluation metrics, new deployment resources, a new environment, or simply a time period passing.

Each of these phases, their automation and steps, the scripts needed to complete them, their metrics, and their outputs could be managed separately, but this would involve quite a bit of management between phases. It would also introduce brittleness to your process since any change in one of the component pieces would cause the entire structure to fail. Thankfully, the same method used in traditional DevOps projects can also be used here: orchestrators.

Think of an orchestrator as a specialized application whose purpose is to manage phased workflows and pass the results of one workflow as inputs to the next. It also manages the triggering events of each of the phases, coordinates the resulting metrics, and can be easily mutated and extended.

Since we have chosen AWS as the cloud computing option for our machine learning workloads, we have a choice of orchestrators. We will be focusing on three: AWS Step Functions, AWS SageMaker Pipelines, and Amazon Managed Workflows for Apache Airflow.

Orchestration Options

A workflow orchestrator is an application whose purpose is the management, coordination, parameterization, and invocation of a defined set of other applications. As applications evolve, they tend to move away from monolithic all-in-one types of deployment to distributed microservice-style applications. This brings several advantages but adds the need to coordinate the different smaller applications. One of the ways to do this is via an orchestrator. You are already familiar with orchestration workflows, like the one where a package is ordered, packaged, and delivered depending on shipping options. Refer to *Figure 12.1*:

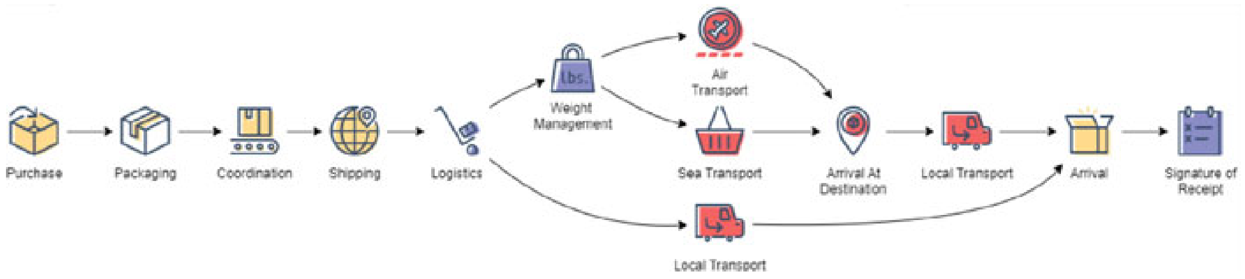


Figure 12.1: *An Example Diagram*

Applications are similar in that we can use an orchestrator to tie a series of applications together, pass the output from one as the input to the next, and add logical conditions to guide the flow depending on parameters, results, or other factors. We can also express this diagram as code, which allows us to check it into our source code repository, test, validate, package, and deploy it in conjunction with our application. In fact, the very deployment pipeline we invoke to accomplish these steps is performed by an orchestrator. Here, you can see an example of such a process triggered by code pushed to a Source Control Repository and orchestrated by AWS CodePipeline. Refer to *Figure 12.2*:

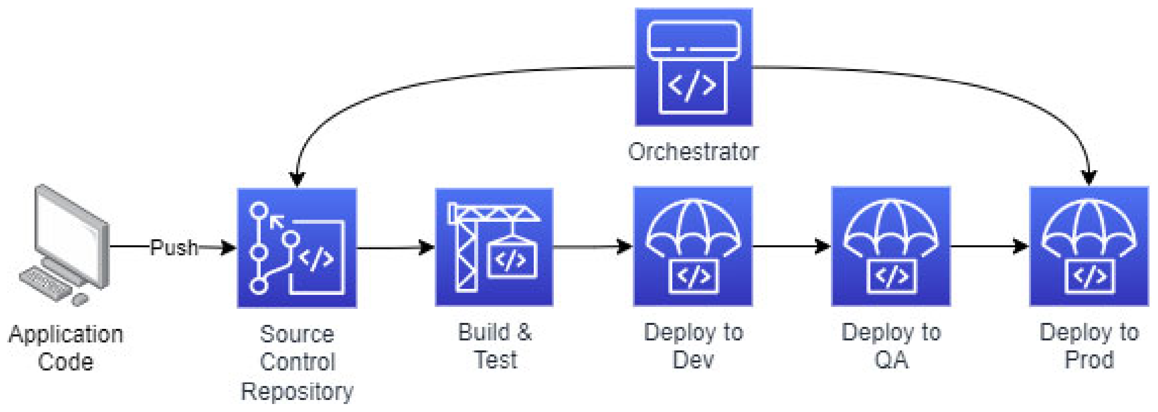


Figure 12.2: *AWS CodePipeline Deployment*

Our machine learning operations (MLOps) workflow has a similar pattern and can be managed by a workflow that makes decisions similar to those we made when we were experimenting with the training, testing, tuning, and deployment options for our models. The result is our machine learning pipeline configured as code and deployed to our choice of orchestrator. Refer to *Figure 12.3*:

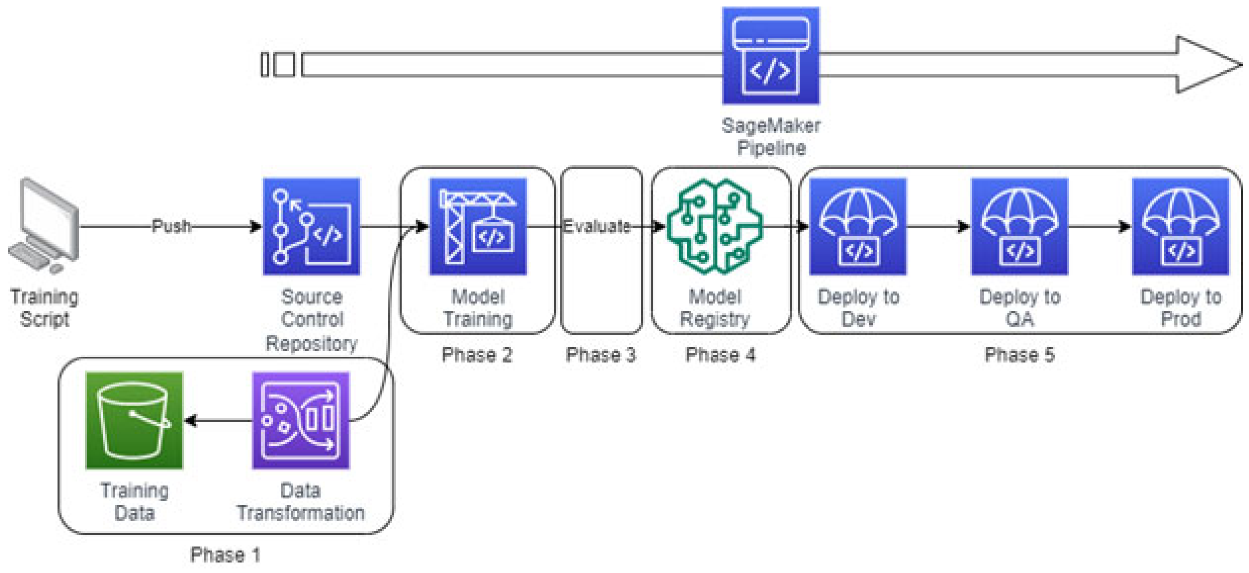


Figure 12.3: *An example machine learning workflow*

The code for each of these phases can be stored alongside the training code or in its own repository. The following is a minor code snippet, which we will expand on later in the chapter, showing how to define our Model Training phase using Amazon SageMaker Pipelines. It is written in Python using the AWS CDK:

```
BuildPipelineConstruct(
    self,
    "ModelTrain",
    project_name,
    project_id,
    s3_model_location,
    s3_pipeline_location,
    model_package_group,
    s3_seed_location,
    kms_key,
)
```

Amazon SageMaker Pipelines

If you have worked with AWS for any amount of time, you might have heard the statement that more than 90% of the services created are based on feedback from the customer. Amazon SageMaker Pipelines are a direct example of this customer obsession. While there are a number of other orchestration options available from AWS, it was clear from machine learning practitioners that they wanted an option that could be created, managed, and observed directly from within their **Integrated Development Environment** (**IDE**). SageMaker Projects and SageMaker Pipelines were created to meet this need.

SageMaker Projects are a logical container for Machine Learning-focused DevOps processes, and SageMaker Pipelines are the connected steps of that DevOps implementation. First, you create the project, and then you define and connect the steps within that project to form your pipeline. Since each of your machine learning workflows have distinct stages, inputs, and results, the project gives you a grouping mechanism for those workflows.

SageMaker Pipelines are under the hood abstractions of other AWS DevOps services, specifically, CodeCommit for the source control repository, CodeBuild for the actual code preparation phase, CodeDeploy for the creation of SageMaker Endpoints, and CodePipeline for the orchestration of all the individual steps. This abstraction serves a specific purpose, allowing the machine learning practitioner to interact with their pipeline from within their chosen IDE and also allowing the operations team to observe and interact with the exact same pipeline from outside SageMaker Studio. This is especially useful because for larger, enterprise-level projects, it is unlikely that the machine learning practitioner will be the one deploying the model to a production environment. This means that the practitioner can create, manage, and observe the pipeline from SageMaker studio, and the operations team can approve deployments to productionand make any changes necessary without getting access to the same Studio environment.

There are several example pipelines pre-loaded in your SageMaker Studio environment; to get started, you can choose Projects from the left-hand menu. From there, you can create a project and choose the appropriate template to start with. The templates themselves are referred to as Infrastructure as Code documents written in the CloudFormation format. These documents are added to the AWS Service Catalog product listing and must contain the **`SageMakerProjectName`** and **`SageMakerProjectId`** keys/values. Finally, the product should have the **`sagemaker:studio-visibility`** key with true value added.

SageMaker Pipelines uses a discrete SDK to describe the steps, requirements, and relationships between steps in a **directed acyclic graph** (**DAG**); it is expressed in the JSON format. The following is an example of a simple model build and training DAG. Refer to *Figure 12.4*:

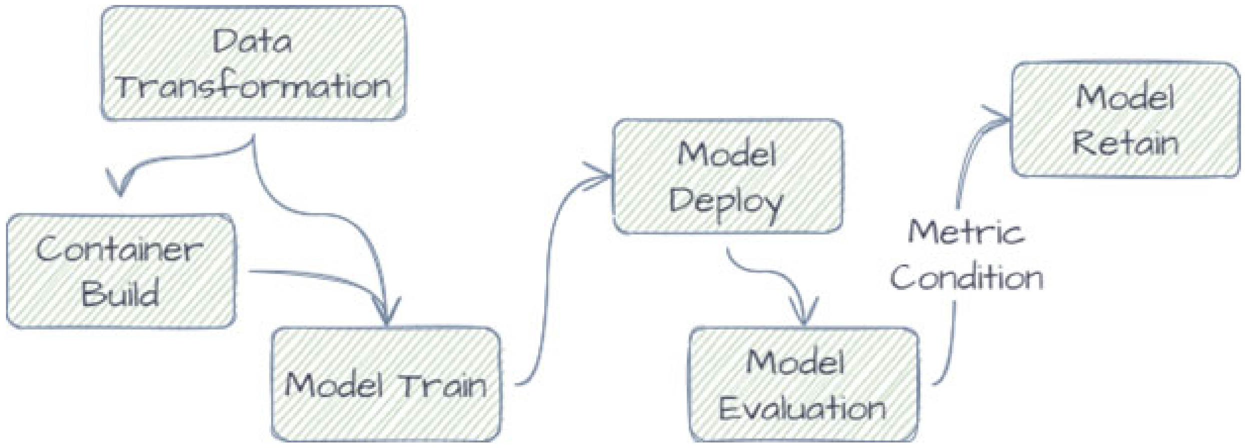


Figure 12.4: *Example DAG flow*

Each pipeline has a discrete number of parameters and steps associated with the machine learning workflow. An example pipeline definition in the AWS SageMaker Pipeline SDK might be as follows:

```
SageMakerPipeline = Pipeline(
   name="RegionallyUniquePipelineName",
   parameters=[
     data_instance_type,
     data_instance_count,
     training_instance_type,
     approval_step,
     training_data
   ],
   steps=[data_transformation, model_train, model_deploy,
   model_evaluation, model_retain],
 )
```

Each of the steps in the steps list will have their own definition block that details the information necessary to complete that step and informs which sections are prerequisites. The following is the **`model_evaluation`** step that determines whether a created model meets the required parameters. There is a branch that gets evaluated if the resulting model does not meet the required parameters and another if it does:

```
model_evaluation = ConditionStep(
  if_steps = [step_register],
  else_steps = [step_fail],
)
```

For the condition step, on the left is the property to use in the comparison and on the right is the value to compare against. In this step, if our model's resultant evaluation score is not greater than 80%, the step evaluates to **`false`**. If it is, the step evaluates to true, and we can move on to registering our model.

As mentioned, one of the key benefits of Amazon SageMaker Pipelines is the ability to view, execute, and monitor your pipeline status from within SageMaker Studio. At any time, you can select SageMaker Projects, then Pipelines, and the pipeline you wish to review. This includes the DAG, a list of executions of the pipeline, and associated data. You can also select one of the pipeline executions to see the details of that specific invocation of your pipeline. If you want to start a pipeline, you can select the Executions tab and click on Start an Execution.

AWS CodePipeline

Amazon SageMaker Pipelines is an abstraction of AWS CodePipeline, a continuous delivery service. This means that your DevOps or operations teams can monitor and interact with your pipeline directly from that service, the AWS SDK, or AWS CLI, without having to access SageMaker Studio.

AWS Step Functions

AWS Step Functions is a serverless orchestrator that allows you to coordinate invocations of various AWS services in order to accomplish a goal. Step Functions can be invoked on a time schedule via Amazon EventBridge, or they can be started as a result of an event emitted by another AWS service. This is particularly useful as a machine learning orchestrator since you can trigger a model training process every time new data arrives in S3 (`PutObject`), every time a new training script is merged into the development branch of a CodeCommit repository (`pullRequestMergeStatusUpdated`), or even when a CloudWatch alarm is triggered on ModelQuality metrics falling below an acceptable level.

Just like Amazon SageMaker Pipelines, Step Functions are defined in discrete stages that can include logical paths and the output of the preceding steps as inputs. Additionally, Step Functions can interact at the API level with many AWS services, eliminating the need to call a separate compute resource (such as AWS Lambda) to trigger the actual service invocation.

With Step Functions, each workflow is a State Machine containing one or more Task states that perform work and Choice states with logic for determining flow, among others.

Step Functions and their respective components, referred to as tasks, can be expressed as JSON using Amazon States Language. An example State Machine that performs several variations on the classic “Hello, World!” program:

```
{"Comment": "This is an example of a State Machine that
completes a Hello, World! scenario",
  "StartAt": "StartState",
  "States": {
  "StartState": {
    "Type": "Task",
    "Resource": "arn:aws :lambda:us-west-
    2:123456789012:function:example_hello_world",
    "Next": "ChoiceState"
  },
  "ChoiceState": {
    "Type" : "Choice",
    "Choices": [
    {
     "Variable": "$.hello_success",
     "NumericEquals": 0,
     "Next": "HelloSuccess"
    },
    {
     "Variable": "$.hello_failure",
     "NumericEquals": 1,
     "Next": "HelloFailure"
    }
    ],
    "Default": "DefaultState"
  },
  "HelloSuccess": {
    "Type" : "Task",
    "Resource": "arn:aws:lambda:us-west-
    2:123456789012:function:double_hello",
    "Next": "EndState"
  },
  "HelloFailure": {
    "Type" : "Task",
    "Resource": "arn:aws:lambda:us-west-
    2:123456789012:function:hello_dlq",
    "Next": "EndState"
  },
  "DefaultState": {
    "Type": "Fail",
    "Error": "StateMachineError**",
    "Cause": "No State Matched"
  },
  "EndState": {
    "Type": "Task",
    "Resource": "arn:aws:lambda:us-
    west=2:123456789012:function:hello_end",
    "End": true
   }
   }
 }
```

The AWS Step Functions console also includes the Step Function Studio, a graphical interface for building State Machines and their connections. This makes it easier to start with the individual components of your flow, then connect them and update the process logic to complete the task.

AWS Step Functions Data Science SDK

The AWS Step Functions Data Science SDK is an open-source library that enables machine learning practitioners to easily create, train, and deploy machine learning models in Amazon Web Services. The SDK is intended to be written using Python and allows pipelines to be designed programmatically. You can use PyPi and Python 3.6 to get started quickly, as follows:

```
pip install stepfunctions
git clone https://github.com/aws/aws-step-functions-data-
science-sdk-python.git
cd aws-step-functions-data-science-sdk-python
pip install.
```

Once you have the SDK installed, you can create the individual steps of your workflow. For example, you can define a step that invokes a Lambda function:

```
LambdaStep = LambdaStep(
  state_id="Print out Hello, World!",
  parameters={
    "FunctionName": "hello_world",
    "Payload": {
    "input": "Function Input"
    }})
```

Once you have the individual steps of your workflow defined, you put them together in a Chain:

```
aiml_chain=Chain([first_task, second_task, third_state])
```

Last, given the workflow tasks are chained together, you can declare your workflow definition:

```
aiml_workflow = Workflow(
  name="AI/ML Workflow v0.0.1",
  definition=aiml_chain,
  role=aiml_iam_role
)
```

Similar to using the AWS CDK for infrastructure, the AWS Step Functions Data Science SDK allows you to manage your workflow programmatically instead of having to learn a semi-structured language like CloudFormation. This is particularly appealing to data scientists since they likely already know and work in Python and would like their tools available in that language as well. You can also use the same SDK to graphically render your workflow:

```
aiml_workflow.render_graph(portrait=False)
```

Similarly, you can create your defined workflow in the AWS Step Functions service by executing the Create function on your workflow object and then executing function to invoke the workflow:

```
aiml_workflow.create()
aiml_execution = aiml_workflow.execute(inputs={
  "times_to_print": "3"
})
```

Apache Airflow Workflows

Apache Airflow is a platform that allows you to create, manage, trigger, and review your created workflows. Similar to SageMaker Pipelines and AWS Step Functions, Apache Airflow allows you to create individual components in your flow and then link them together, including inputs and outputs from the respective tasks. You can create workflows either with Amazon SageMaker operators (v1.10.1 or later) or with Airflow PythonOperator.

Apache Airflow may be an appealing option if you manage workflows in multiple cloud or hybrid environments. It may also be advantageous in situations where your team has a strong understanding of Apache Airflow.

Using the SageMaker operators, you take your created estimator object and pass it to an imported **`sagemaker.workflow.airflow training_config`** object:

```
training_config = training_config(estimator=estimator_object,
inputs=s3_train_data)
```

Once you have that, you can create a similarly imported sagemaker.workflow.airflow **`transform_config_from_estimator`** object:

```
transform_config =
transform_config_from_estimator(estimator=estimator_object,
            task_id='training',
            task_type='training',
            instance_count=1,
            instance_type='ml.c5.large',
            data=s3_transform_data,
            content_type='text/csv')
```

Then you declare the airflow default arguments inside your new DAG (similar to the Step Functions Data Science SDK):

```
workflow = DAG(
  "MLOps Example", default_args={
    "depends_on_past": False,
    "retries": 3,
    "retry_delay": datetime.timedelta(hours=1),
  },
  start_date=pendulum.datetime(2022, 09, 1, tz="UTC"),
  description="Airflow ",
  schedule="@daily",
  catchup=False,
)
```

Phase Discrimination

After careful consideration of the options, our data scientist turned SRE decided to implement SageMaker Pipelines. This is mainly because the team has no existing Step Function or Apache Airflow workloads, so extending their existing knowledge to machine learning isn't applicable. Being able to create, trigger, monitor, and manage the pipelines from SageMaker Studio is a significant advantage. The deciding factor is that other groups in the company use AWS CodePipeline, which means that since SageMaker

Pipelines is an abstraction of AWS CodePipeline, the existing operations teams can continue to manage the production deployments the same way they are today.

The decision has been made; the next step is to get started. Opening SageMaker Studio, select the SageMaker Resources icon on the left sidebar. From there, they can select `Projects` from the drop-down list, as shown in *Figure 12.5*:

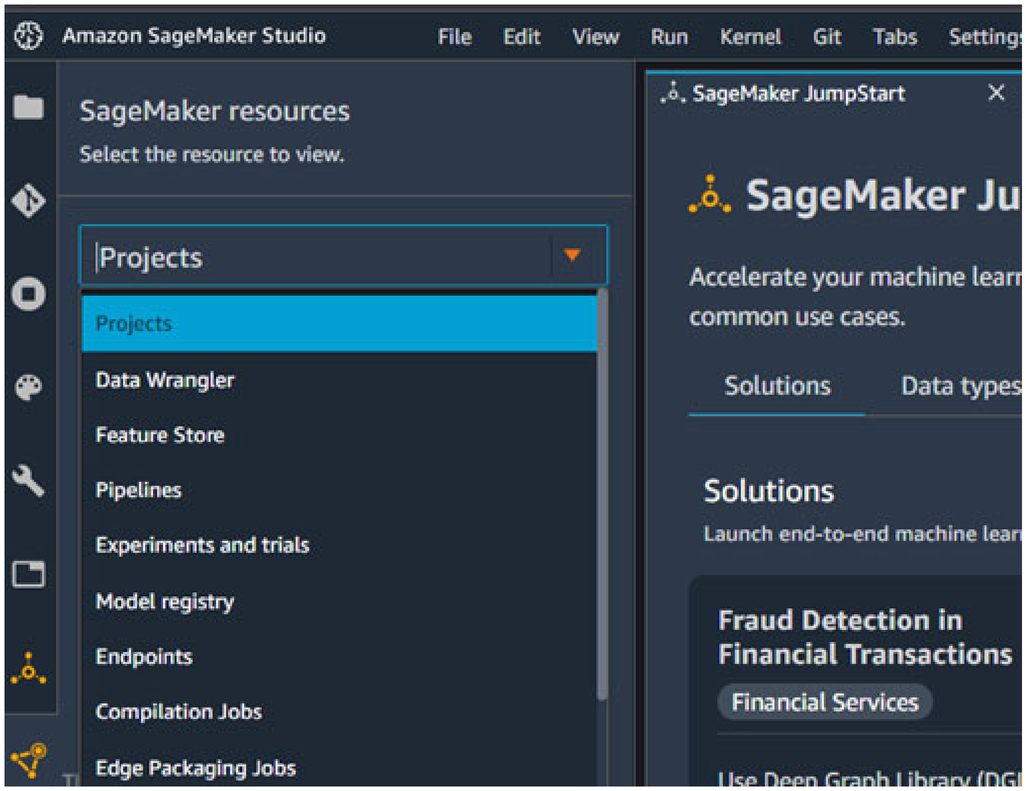
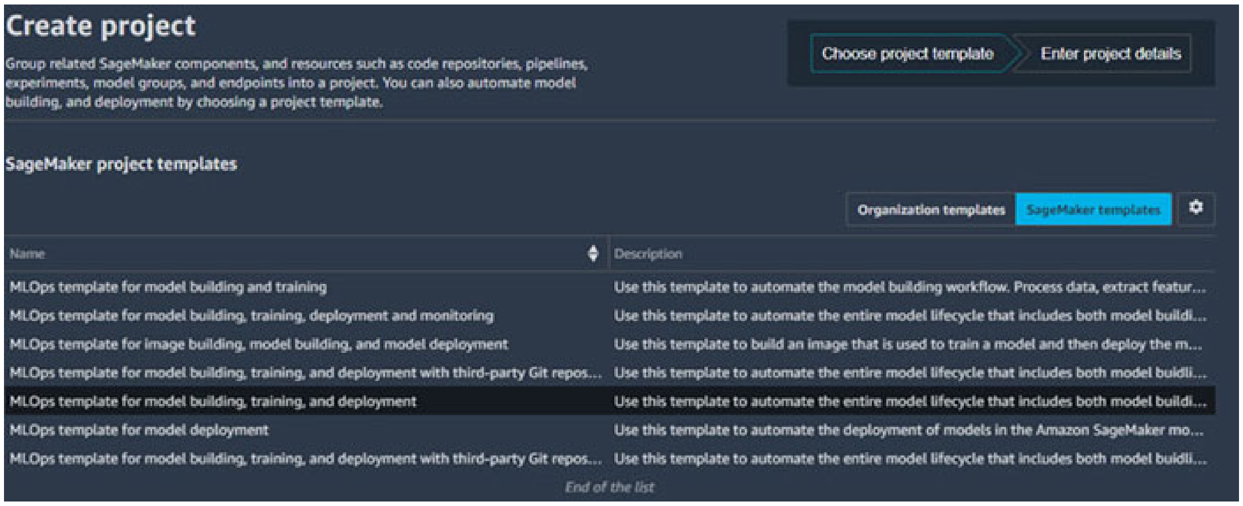


Figure 12.5: *SageMaker Studio Projects*

SageMaker Projects is a logical grouping of source control repositories associated with the individual components of a machine learning workflow. Since the phases are invoked separately, it is common to have one or more of them in separate repositories. SageMaker Projects allow you to group those repositories together in a single object.

SageMaker Projects has several predefined templates that cover many workflows need. Our nascent SRE knows that customized versions can be created but wants to start with one of the provided ones to test how it all works. Their teams already use AWS CodeCommit as a source control repository, committing their data transformation and model training scripts, so they select **`MLOps template for model building, training, and deployment`**. Refer to *Figure 12.6*:



Create project

Group related SageMaker components, and resources such as code repositories, pipelines, experiments, model groups, and endpoints into a project. You can also automate model building, and deployment by choosing a project template.

SageMaker project templates

Name	Description
MLOps template for model building and training	Use this template to automate the model building workflow. Process data, extract featur...
MLOps template for model building, training, deployment and monitoring	Use this template to automate the entire model lifecycle that includes both model buildi...
MLOps template for image building, model building, and model deployment	Use this template to build an image that is used to train a model and then deploy the m...
MLOps template for model building, training, and deployment with third-party Git repos...	Use this template to automate the entire model lifecycle that includes both model buildi...
MLOps template for model building, training, and deployment	Use this template to automate the entire model lifecycle that includes both model buildi...
MLOps template for model deployment	Use this template to automate the deployment of models in the Amazon SageMaker mo...
MLOps template for model building, training, and deployment with third-party Git repos...	Use this template to automate the entire model lifecycle that includes both model buildi...

End of the list

Figure 12.6: *SageMaker Pipeline Template Selection*

Once the template has been selected and the project has been given a name, the associated AWS resources are provisioned via the CloudFormation template linked to the SageMaker Pipeline template. The template provisions a CodeCommit repository to store the training scripts, a CodeDeploy job to train the model, an S3 bucket to store model artifacts, and a CodePipeline job to coordinate it all. These services are separate from the SageMaker Studio, but our SRE will be able to observe their behaviors and control their actions from within.

Once the pipeline has finished creating the associated infrastructure, it will respond with two repositories: one for the build portion of our pipeline and the other for the deployment. SageMaker Studio Pipelines separates the steps into two repositories, but in order to understand the process, we will treat it as five separate sections as detailed below.

Data Transformation

When first building the solution to a Machine Learning problem, we went through the steps of identifying our data sources, integrating the respective data selected from those sources, and the feature engineering of the data set. From that initial work, we can extract an automate-able data transformation script that will run based on a trigger we can define. That trigger can be new data written to one of our data repositories, such as new data in Amazon

Aurora for MySQL, a specific S3 bucket, or a Snowflake Data Warehouse. It can also be a new training script checked into our source code repository.

Once that triggering action takes place, the build pipeline will be triggered, and the script checked in will be downloaded from the repository and run. The behavior of that build function is controlled by one of the files in the repository: **`codebuild-buildspec.yml`**.

This file contains the command that runs the pipeline, as shown below, and can be edited to match the name of your project:

```
run-pipeline --module-name pipelines.abalone.pipeline
```

Replace **`abalone`** with a chosen pipeline name.

The default location created as part of the pipeline template is in **`/pipelines/abalone/preprocess.py`**. You can replace this code (and path, since it is unlikely that you are attempting to predict the size of snails, to match the edits you made to **`codebuild-buildspec.yml`** above) with your feature engineering code or edit the existing file to create your processing code.

Our SRE is implementing a pricing prediction linear regression model, so they choose **`priceIsRight`** as their pipeline name. In the same **`/pipelines/priceIsRight/`** directory, there is also a **`pipelines.py`** file with references to abalone that will need to be updated as well as an **`input_data`** object that should refer to the s3 location where our data transformation step will write the data to be used in the next step.

```
input_data = ParameterString(
  name="InputDataUrl", default_value=f"s3://your-s3-data-
  location/data/priceIsRightData.csv",
  )
```

There is also an **`evaluate.py`** in the same directory, but we will come back to that file in phase 3: model evaluation. Once the changes are made, you can stage, commit, and push them back to the CodeCommit repository. There is a trigger set up for you that invokes your pipeline when any changes are pushed to the repository, and you can track the changes in SageMaker Studio by selecting your project.

If the process works as expected, you should see the results if the data appears in your selected S3 bucket. If there are errors, check the log outputs of the SageMaker processing job.

Model Training

Now that we have input data prepared, we can move to the second phase of the process, though we will still use the first of our two repositories: the build. From that process, we will continue with the `pipeline.py` file, which contains the script used to train our model. The provided script is intended for the SageMaker built-in XGBoost framework, and the file can be replaced with your model training commands created during your experiments.

The steps to train your model are the same as when we were experimenting in earlier chapters. We continue to create an estimator, set hyperparameters, then call `.fit()` on your estimator. Your chosen model may have different steps that can be added to the `pipeline.py` file.

Model Evaluation

Once we observe a successful training job and a model artifact written to the identified s3 bucket, we can consider the method we will use for model evaluation. During the experiments we ran, we immediately deployed a trained model to a SageMaker endpoint to evaluate it against the chosen metric. We will do the same here, but with the data from those experiments, we can also set the minimum threshold for model performance that must be met before we can consider the model viable to address our problem.

The chosen metric will depend on the model but should take the business outcome into consideration. The value can and should be updated as the business needs change. In the early portions of a Machine Learning project, an accuracy of 60% may be acceptable, but as the product and process mature, that may be adjusted to something more accurate. It should be noted that this is not intended to compare the score against the currently deployed model but determine a minimum threshold that any model should pass before being saved.

Note on s3 Lifecycle Policies: The amount of data written to s3 for even a single data scientist can be significant. Training data, feature engineered data, model artifacts, model monitoring data captures, training logs, and other data can clutter your s3 buckets and lead to unnecessary costs. Consider implementing an s3 lifecycle policy that moves any objects not accessed in the last 30 days to a lower cost tier.

You can then set a longer policy to either expire objects or move them to an even lower cost tier.

For your pipeline, you set the evaluation script invocation, including the model deployment, to a short-lived SageMaker endpoint in the model build repository **`pipeline.py`**. The actual evaluation script is in the same location and entitled **`evaluate.py`**.

The specific line in **`pipeline.py`** that determines the threshold for a model retention is the **`ConditionStep`** portion of the pipeline. The left portion is the metric to evaluate, and the right portion is the value reported by the model evaluation. In the following example, we require that the accuracy of the model be greater than 75%.

```
modelMetric= JsonGet(
  step_name=stepProcess.name,
  property_file=evaluationReport,
  json_path="evaluation.accuracy",
)
stepCondition = ConditionStep(
  name="priceIsRightConditionStep",
  conditions=[ConditionGreaterThanOrEqualTo(left=modelMetric,
  right=0.75)],
  if_steps=[stepRegister],
  else_steps=[stepFailure],
)
```

Once a model passes the minimum conditions, the model URL is passed to the next phase: model retention. If your training process is triggered by new training scripts or new data written to their respective repositories, you may have quite a number of viable models to deploy within a given window. In this situation, you can retrieve all the recorded metrics for saved models and select the one with the best possible capability. Choosing this model and deploying it alongside the current production model is often referred to as a champion/challenger approach. The existing model is the champion, and the new model is the challenger. The common approach is to use SageMaker Endpoint production variants between the two models and adjust their model weight to send some fraction of live traffic to the new model to validate its accuracy and overall capability of replacing the champion. The replacement action can be completed by selecting the new challenger model to receive all the production traffic.

Model Artifact Management

Once a model has passed our condition expression, we want to store it somewhere we can retrieve it from later. When we were in the initial experimentation phases with our model building, we often stored our models in s3. Object storage is especially suited to this, since the trained model location in s3 is stored as part of the estimator's **`.fit()`** function. As we move into the operationalization portion of our workflow, we can continue to use s3, but keeping track of those individual models becomes more and more challenging. Additionally, storing metadata alongside the models enables functionality like semantic versioning (outside of the model filename, but this quickly encounters file length issues), model evaluation metrics, references to training data, or even group model versions together.

There are third-party options that can store several different artifacts, such as Artifactory, Nexus, or even AWS CodeArtifact. These will work, but they are not specifically suited for machine learning models.

Taking those options into consideration, our SRE chooses to use the built-in option: SageMaker Model Registry. The decision also has the advantage of being supported by the chosen SageMaker Pipeline template. The specific steps are in the **`pipeline.py`** file and should be updated with specific information about your model:

```
priceIsRightModel = Model(
    image_uri=imageURI,
    model_data=stepTrain.properties.ModelArtifacts.S3ModelArtifac
    ts,
    sagemaker_session=sagemaker_session,
    role=sagemaker_role,
  )
  stepArgs = priceIsRightModel.register(
    content_types=["text/json"],
    response_types=["text/json"],
    inference_instances=["ml.m5.large", "ml.m5.xlarge"],
    transform_instances=["ml.c5.large"],
    model_package_group_name=ModelPackageGroup,
  approval_status=modelApprovalState,
  model_metrics=modelMetric,
 )
 stepRegister = ModelStep(
  name="RegisterPriceIsRightModel",
  step_args=stepArgs,
 )
```

Model Deploy

Here, in the last phase, we finally switch to the other repository created for us: the deploy repo. Similar to the build repo, we can clone (git nomenclature for creating a local copy of a remote repository) the repo and take a look at the contents.

For our build repository, the trigger was a new object being checked into the repository. The deploy repository has a trigger, the act of registering a model in the Model Registry. That registration triggers the pipeline to deploy the model to a staging endpoint. Once that deployment is finished, it will wait for approval to deploy the same model to a production endpoint. Automating this process and separating it from the previous steps means that they can operate independently. Once the machine learning practitioners revise their training process, the automatic training will retrigger. Once a model has been programmatically identified as suitable for deployment by exceeding the identified model metrics, it starts the deployment process.

This results in freedom for your machine learning practitioners to continue with their experiments, identifying improvements to the training process or model behaviors, and then, when they are ready for use, checking them into the repository.

All resultant models that pass the identified metrics will be queued for deployment. Of course, not all of them will be deployed to production, but the ones that are not selected can have their approval status set to Rejected.

The deploy pipeline has two endpoints created: staging and production. You can add as many as are relevant to your software development life cycle needs, including Quality Assurance, Integration, or Beta endpoints. You can also set these endpoints to be deployed in other AWS accounts that your AWS CodeDeploy job is given permission to access.

Within the deploy repository, **`buildspec.yml`** gives instructions for AWS CodeDeploy to take when it is invoked. This file uses the included **`build.py`** to take arguments and write staging and production configuration files that are then used by the Amazon CloudFormation Template to create the actual endpoint infrastructure.

Each environment also has an environmental configuration file where you can set parameters like instance type and number of instances for the endpoint. If you added a QA environment configuration file, it might look as follows:

```
{
  "Parameters": {
    "StageName": "qa",
    "EndpointInstanceCount": "1",
    "EndpointInstanceType": "ml.m5.large",
  }
}
```

You would also need to add an argument in **`build.py`** to import the new **`qa-config.json`**:

```
parser.add_argument("--import-qa-config", type=str,
default="qa-config.json")
```

Finally, a new code block to write out the qa configuration file:

```
with open(args.import_qa_config, "r") as f: qa_config =
extend_config(args, model_package_arn, json.load(f))
logger.debug("QA config: {}".format(json.dumps(qa_config,
indent=4)))
with open(args.export_qa_config, "w") as f:
json.dump(qa_config, f, indent=4)
if (args.export_cfn_params_tags):
create_cfn_params_tags_file(qa_config, args.export_qa_params,
args.export_qa_tags)
```

The approval for a model to be deployed is managed within the SageMaker Model Registry and can be set either in AWS SageMaker Projects or directly in AWS CodePipeline. This means that you can have approvers that do not need access to SageMaker Studio, including programmatic ones with integrations to change management systems.

Best Practices using the AWS Well-Architected Lens for Machine Learning

Once you have operationalized your machine learning workflow, you can also implement monitoring, automatic retraining, dashboards, and additional processes, but often, the issue is knowing the best practices that should be prioritized. Thankfully, Amazon Web Services has provided a service that distils the codified learnings for all their customers into the Well-Architected Tool.

For a workload to be considered well-architected, it should align with the learnings presented in the six categorical pillars of the Well-Architected Framework. The pillars, that is, operational excellence, security, reliability, performance efficiency, cost optimization, and sustainability, form groups of questions that the machine learning practitioner team can use to guide their technical backlog and development path. Thankfully, these phases and steps align well with the ML workflow that we reviewed in *Chapter 1, Introducing the ML Workflow*.

The outcome of answering the questions and identifying the areas that have not yet been addressed is a set of high and medium vulnerabilities that can be used as structured data, identifying the approach to a goal of being well-architected. Applying a specific set of questions to the well-architected tool is referred to as applying a lens.

For the AWS Well-Architected Machine Learning Lens, the workflow is grouped into eight sections, as shown in *Figure 12.7*:

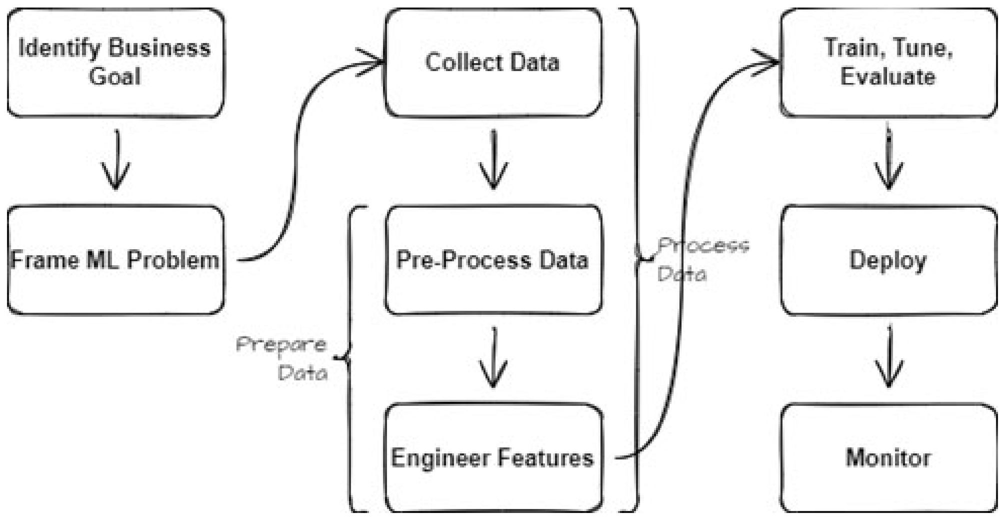


Figure 12.7: *The Well-Architected Machine Learning Lens*

Making the Well-Architected Framework part of your team's planning effectively provides a method for providing structured and actionable metrics from an unstructured discussion.

Some of the best practices to consider including in the process are listed here:

- **Security**: Start with pre-built containers for your workflow. Focus on the training process before taking on the additional development overhead of managing your own DockerFile for training, inference, or both.
- **Reliability**: Agree on a level of explainability. We covered the concepts of model explainability in *Chapter 8, Blue or Green*, but some decisions must be madein coordination with business stakeholders, especiallyabout the level of explainability for the project. No two projects or even industries will be the same, so it would be reconsidered per workload.
- **Performance efficiency**: Quantify the value of machine learning on AWS. Engineering effort on any scale must take into consideration the benefit of the effort. What level of cost reduction, automation, additional experiments, or other key performance indicators will be reported by the project?
- **Cost optimization**: Use managed services; it has been discussed multiple times in this book that these tasks can be accomplished by provisioning one or more EC2 instances and managing the compute resources yourself. The drawbacks of this approach have been discussed, but alongside using managed services like AWS SageMaker, AWS Step Functions, or AWS CodePipeline have advantages that should be quantified and recorded. These cost optimizations can be allocated elsewhere to further accelerate your machine learning goals.
- **Operational Excellence**: Developing the skills of your machine learning practitioners is key, especially in such a fast-paced, rapidly accelerating field. It would be an anti-pattern to assume that the current skill set of your team members will be suitable for everything all the time. Investing in your team's knowledge growth means investing in the quality of your workload.

- **Sustainability**: Despite being the newest pillar, sustainability is quickly gaining importance. The use of AWS services already helps you take steps in the right direction because of the lower carbon footprint AWS can manage as compared to an on-premise data center. Further, using tools like the Customer Carbon Footprint tool can provide quantifiable metrics for environmental impact per workload. Consume these metrics alongside domain specific ones to make directional decisions.

Conclusion

Here, we conclude our learning journey together, and what a fascinating one it has been! You may think that this is too brief, too terse, too fleeting to cover all that can be learned for machine learning on AWS, and you would be absolutely correct. At the same time, the journey continues. This may be your first time working with machine learning concepts, or you may be quite experienced in this field and may be using this book as a method to improve your skills. In either case, the amount of knowledge left to acquire in this field alone is near infinite. Yet, this is a cause of celebration, not regression (pun intended). This is a time of fascinating discovery and amazing enablement for machine learning. Leveraging on-demand near-infinite scales offered by cloud computing means that models with billions of parameters can not only exist but are in use right now. Machine learning is being used to find cures for cancer, put an end to human trafficking, and enable communication between people from all over the world, and your work is part of that.

Even if the project you are working on feels mundane, you work in a field that is expanding moment by moment. Where one journey ends, another begins, and this is the beginning of the things you will accomplisheven if those are a few competitions on Kaggle, an Amazon SageMaker Jumpstart experiment, or the next best style of proto-transformer models, your work matters because you choose to do it. Now, at the end of this book, we are colleagues. So, feel free to reach out and let us know what you are working on and what you think of this book.

Lastly, take a moment and think about your journey and how far you've come. What is next is up to you but consider sharing your journey with others. Social media, video blogging, even a book of your own. We are enriched when we elevate others, and we are in this together; machine learning is no different.

Multiple Choice Questions

Use these questions to challenge your knowledge of MLOps and well-architected Machine Learning workflows.

1. **What is the purpose of SageMaker Projects for MLOps?**
    a. Automatically tracking experiments for a given model
    b. Standardizing developer environments, dependency, and code management
    c. Managing the networking overhead of Machine Learning projects
    d. Managing the planning of machine learning workflows
2. **What details do SageMaker Experiments allow you to track across experiments?**
    a. Parameters, metrics, datasets, and other artifacts related to training jobs
    b. Parameters, metrics, datasets, and associated details of live inference
    c. Request and response payloads
    d. Level of effort put into each training iteration
3. **Which is not an approval status for a model managed by the SageMaker Model Registry?**
    a. PendingManualApproval
    b. Approved
    c. Deferred
    d. Rejected
4. **Which service stores and manages the templates used to create the infrastructure associated with a SageMaker pipeline?**
    a. AWS CloudFormation
    b. Amazon CloudInstigator

    c. AWS Serverless Application Model
    d. AWS Service Catalog

5. **Using the AWS Well-Architected Framework Reliability pillar, who should be involved with determining the level of explainability?**
    a. Business stakeholders
    b. Customers
    c. Data scientists only
    d. Legal team

Answers

1. **b**
2. **a**
3. **c**
4. **d**
5. **a**

Further Reading

Additional reading and information sources that will enrich your knowledge of the concepts covered in this chapter:

- *Integrating Amazon SageMaker Data Wrangler with MLOps workflows:* **https://aws.amazon.com/blogs/machine-learning/integrate-amazon-sagemaker-data-wrangler-with-mlops-workflows/**
- *MLOps at the edge:* **https://aws.amazon.com/blogs/machine-learning/mlops-at-the-edge-with-amazon-sagemaker-edge-manager-and-aws-iot-greengrass/**
- *MLOps for enterprises:* **https://aws.amazon.com/blogs/machine-learning/mlops-foundation-roadmap-for-enterprises-with-amazon-sagemaker/**
- *Orchestrating XGBoost with Apache Airflow:* **https://aws.amazon.com/blogs/machine-learning/orchestrate-**

xgboost-ml-pipelines-with-amazon-managed-workflows-for-apache-airflow/

- *Manage AutoML workflows with AWS step functions:* **https://aws.amazon.com/blogs/machine-learning/manage-automl-workflows-with-aws-step-functions-and-autogluon-on-amazon-sagemaker/**

Index