O'REILLY®

Data Science on AWS

Implementing End-to-End, Continuous AI and Machine Learning Pipelines

Early Release

RAW & UNEDITED

Chris Fregly &
Antje Barth

Data Science on AWS

Implementing End-to-End, Continuous AI and Machine Learning Pipelines

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Chris Fregly and Antje Barth

Data Science on AWS

by Chris Fregly and Antje Barth

Copyright © 2021 Antje Barth and Flux Capacitor, LLC. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: Jessica Haberman

Development Editor: Gary O'Brien

Production Editor: Katherine Tozer

Interior Designer: David Futato

Cover Designer: Karen Montgomery

July 2021: First Edition

Revision History for the Early Release

- 2020-05-05: First Release
- 2020-06-10: Second Release
- 2020-07-17: Third Release
- 2020-08-03: Fourth Release
- 2020-08-26: Fifth Release

- 2020-10-02: Sixth Release
- 2020-11-30: Seventh Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492079392 for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Data Science on AWS*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-07932-3

Chapter 1. Automated Machine Learning

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *support@pipeline.ai*.

In this chapter, we will show how to use the fully-managed Amazon AI and machine learning services to offload the undifferentiated heavy lifting of building AI pipelines. We dive deep into two Amazon services for automated machine learning, Amazon SageMaker Autopilot and Amazon Comprehend, designed for users who want to build powerful predictive models from their datasets with just a few clicks.

What is Automated Machine Learning?

Automated machine learning (AutoML) commonly refers to the effort of automating the typical steps of a machine learning pipeline shown in Figure 1-1.

Figure 1-1. Typical machine learning pipeline.

Machine learning practitioners spend a lot of time building and managing such pipelines. They need to prepare the data and decide on the framework and algorithm to use. Seasoned data scientists use years of experience and intuition to choose the best algorithm for a given dataset. In an iterative process, ML practitioners try to find the best performing model configuration called "hyper-parameters". Unfortunately, there is no cheat sheet either for choosing any of these parameters. We still need experience, intuition, and patience to run many experiments and find the best hyper-parameters for our algorithm and dataset.

What if we could just use a service that automatically finds and trains the best model for our dataset and deploys the model to production with a single click? Amazon SageMaker Autopilot offers us exactly this functionality. Autopilot simplifies the model training and tuning processing by handling many aspects of the model development life cycle (MDLC) including feature transformation, algorithm selection, model training, tuning, and deployment.

Simply point Autopilot to your dataset - and out comes a set of fully-trained and optimized predictive models. Autopilot explores many algorithms and configurations based on many years of AI and machine learning experience at Amazon. The model candidates are summarized by Autopilot through a set of generated Jupyter notebooks and Python scripts. You have full control over these generated notebooks and scripts. You can modify them, automate them, and share them with colleagues. You can select the top model candidate based on your desired balance of model accuracy, model size, and prediction latency. Let's dive deeper into the process of automated machine learning with Autopilot.

Automated Machine Learning with Autopilot

Autopilot is the name of Amazon SageMaker's AutoML service. You simply provide your raw data in a S3 bucket, for example in the form of a tabular CSV file, and tell Autopilot which column you want to predict. As the name implies, Autopilot then does the rest automatically.

> **NOTE**
>
> S3 is Amazon's Simple Storage Service. S3 provides a simple web service interface that you can use to store and retrieve any amount of data. We will discuss this service in more detail in the next chapter.

Autopilot uses automated machine learning to analyze the data and identifies the best algorithm and model configuration for your data as shown in Figure 1-2.

Figure 1-2. Amazon SageMaker Autopilot.

You can tell Autopilot how many model candidates to explore. In the process of building those candidates, Autopilot tries different algorithms and algorithm settings. Autopilot also applies all needed data transformations to your data to optimize the input for each algorithm. The algorithm, configuration, and data transformation code are then combined into a single ML pipeline definition. The most promising pipelines are selected by Autopilot and used to find the best performing model. Lastly, Autopilot shares the results in a model leaderboard. You can use the best performing model as a baseline and optimize the model even further. A second option is to simply deploy the model and start predicting.

Another highlight of Autopilot is the fact that it provides full visibility into each of those steps and shares all code needed to reproduce the results.

AWS calls this a “white-box” approach. This white-box approach to AutoML is very unique. Let’s explore the white-box vs. black-box approach to AutoML a bit further.

Understand Autopilot’s White-Box Approach to AutoML

In a black-box approach as shown in Figure 1-3, you don’t have control or visibility into the chosen algorithms, applied data transformations, or hyper-parameter choices. You point the AutoML service to your data and receive a trained model.

Figure 1-3. Other AutoML products use a black-box approach.

This makes it hard to understand and explain the model, and to manually reproduce the model. Most AutoML solutions implement this kind of black-box approach. In contrast, Autopilot documents and shares its findings throughout the data analysis, feature engineering, and model tuning steps, as shown in Figure 1-4.

Figure 1-4. Autopilot's white-box approach to AutoML.

Autopilot not only shares the models, Autopilot also logs all observed metrics and generates Jupyter notebooks which contain the code to reproduce the model pipelines. The Data Exploration notebook contains the data analysis results and identifies potential data quality issues such as missing values that might impact model performance if not addressed. The Candidate Definition notebook highlights the best algorithms to learn our given dataset, as well as the code and configuration needed to use your dataset with each algorithm.

> **NOTE**
>
> The Jupyter notebooks are available after the first data analysis step. You can configure Autopilot to stop after this step if you want to iterate quickly over your data before starting the actual model tuning step.

Use SageMaker Experiments in Autopilot

Autopilot uses SageMaker Experiments to keep track of all data analysis, feature engineering and model training/tuning jobs. This feature of the

broader Amazon SageMaker family of ML services helps you organize, track, compare and evaluate machine learning experiments. SageMaker Experiments enables model versioning and lineage tracking across all phases of the ML lifecycle.

An experiment consists of trials and training jobs as shown in Figure 1-5. A trial is a collection of training steps and metadata for those steps. The training steps typically include data preprocessing, model training, and model tuning. Metadata includes dataset locations, algorithm hyper-parameters, output artifacts, and performance metrics.

Figure 1-5. SageMaker Experiments.

You can explore and manage Autopilot experiments and trials either through the UI or using the AWS Python SDK boto3

[https://github.com/boto/boto3]. Let's have a look at both options and start an Autopilot experiment to build a custom classifier model.

As input data, we leverage samples from the Amazon Customer Reviews Dataset [https://s3.amazonaws.com/amazon-reviews-pds/readme.html]. This dataset is a collection of over 150 million product reviews on Amazon.com from 1995 to 2015. Those product reviews and star ratings are a popular customer feature of Amazon.com. Star rating 5 is the best and 1 is the worst. We will describe and explore this dataset in much more detail in the next chapters.

Train and Deploy with Autopilot UI

The Autopilot UI is integrated into SageMaker Studio, an IDE which provides a single, web-based visual interface where you can perform your machine learning development. Simply navigate to Amazon SageMaker in your AWS Console and click on SageMaker Studio as shown in Figure 1-6.

aws Services Resource Groups N. Virginia Support

Amazon SageMaker X

Amazon SageMaker Studio

Dashboard

Search

▼ Ground Truth

Labeling jobs

Labeling datasets

Labeling workforces

▼ Notebook

Notebook instances

Lifecycle configurations

Git repositories

▼ Training

Algorithms

Training jobs

Hyperparameter tuning jobs

Amazon SageMaker 〉 Dashboard

Overview

Hide

Ground Truth

Set up and manage labeling jobs for highly accurate training datasets using active learning and human labeling.

Labeling jobs

Notebook

Availability of AWS and SageMaker SDKs and sample notebooks to create training Jobs and deploy models.

Notebook instances

Training

Train and tune models at any scale. Leverage high performance AWS algorithms or bring your own.

Training jobs

Hyperparameter tuning jobs

Inference

Create models from training jobs or import external models for hosting to run inferences on new data.

Models

Endpoints

Batch transform jobs

AWS Marketplace

Find, buy, and deploy ready to use model packages, algorithms, and data products in AWS Marketplace

Browse Catalog

Figure 1-6. AWS Console > Amazon SageMaker > Amazon SageMaker Studio

Then, follow the instructions to set up SageMaker Studio and click on Open Studio once completed as shown in Figure 1-7.

aws Services ▾ Resource Groups ▾ N. Virginia ▾ Support ▾

Amazon SageMaker X

Amazon SageMaker Studio

Dashboard

Search

▼ Ground Truth

Labeling jobs

Labeling datasets

Labeling workforces

▼ Notebook

Notebook instances

Lifecycle configurations

Git repositories

▼ Training

Algorithms

Training jobs

Hyperparameter tuning jobs

Amazon SageMaker 〉 Amazon SageMaker Studio 〉 Control Panel

Amazon SageMaker Studio Control Panel

Choose your user name, then choose Open Studio to get started

Add user

Search users

〈 1 〉

User name	Last modified	Created	
antje	Apr 29, 2020 14:37 UTC	Apr 29, 2020 14:37 UTC	Open Studio

▼ Studio Summary

How to delete Studio | Delete Studio

Status	Studio ID	Execution role	Authentication method
Ready	d-ntcp5dhsicjk	arn:aws:iam:: [redacted] :role/service-role/AmazonSageMaker-ExecutionRole-20191201T115647	AWS Identity and Access Management (IAM)

Use the Studio ID for troubleshooting and tracking usage.

The status shown is for the Amazon SageMaker Studio service, and is not the status of compute resources such as EC2 instances to execute notebooks.

Figure 1-7. Setup SageMaker Studio

This will take you to the SageMaker Studio IDE as shown in Figure 1-8.

Amazon SageMaker Studio File Edit View Run Kernel Git Tabs Settings Help Feedback

/

Name Last Modified

Launcher

Launch a new activity

Depending on the SageMaker image that you choose, your activity will start in a ml.t3.medium instance (for CPU optimized images) or a ml.g4dn.xlarge instance (for GPU optimized images). You can change your instance at any time. Learn more about instances and pricing

Select a SageMaker image to launch your activity

Data Science

Create a new notebook, or launch an interactive shell or terminal within the selected SageMaker image

Learn more about image and system terminals

Notebook

Python 3

Interactive Shell

Python 3

Terminal

Image Terminal

0 0 Launcher

Figure 1-8. SageMaker Studio IDE

You will find the Autopilot UI when you click the laboratory flask icon in the left-side menu as shown in Figure 1-9. Once there, click on Create experiment to create and configure our first Autopilot experiment.

Figure 1-9. Create an Autopilot experiment.

In preparation for our Autopilot experiment, we use a subset of the Amazon Customer Reviews Dataset to train our model. We want to train a classifier to predict the star rating for a given review, therefore we only use the `star_rating` and `review_body` columns:

```
star_rating,review_body
5,"GOOD, GREAT, WONDERFUL, MORE THAN ADEQUATE AND EXACTLY WHAT I  NEED FOR MY
COMPUTER. NONE BETTER OR GOOD TO USE."
2,"Even though it does the same job as TurboTax, it isn't as user friendly as
TurboTax. I guess you get what you paid for."
4,"Pretty easy to use. No issues."
…
```

> **NOTE**
>
> In other scenarios, you will likely want to use more columns from your dataset and let Autopilot choose the most important ones. In our example, however, we want to keep things simple and just use the star_rating and review_body columns to focus on the technical aspects of Autopilot.

Next, we configure the Autopilot experiment as shown in Figure 1-10.

Create experiment

Create Amazon SageMaker Autopilot Experiment

JOB SETTINGS

Experiment Name

amazon-customer-reviews

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

Input data location (S3 bucket)
Enter the location in S3 where your training data is stored. You can point to a single data file, an S3 object key prefix that contains only data files, or a manifest file that contains the location of your input data. See more in the AWS Docs

Find S3 bucket Enter S3 bucket location

Note: The S3 bucket must be in the same AWS Region where you're running SageMaker Studio because SageMaker doesn't allow cross-region requests.

S3 bucket address

s3://dsoaws-amazon-reviews/data/amazon_reviews_us_Digital_Software_v1_00_header.cs

Is your S3 input a manifest file?
For more information on the format of a manifest file, please see the AWS Docs

Target attribute name
The target attribute is the attribute in your dataset that you want Amazon SageMaker Autopilot to make predictions for.

star_rating

The attribute name is case-sensitive and must match exactly the name in your input dataset

Output data location (S3 bucket)
Enter the location in S3 where you want to store the output.

Find S3 bucket Enter S3 bucket location

Note: The S3 bucket must be in the same AWS Region where you're running SageMaker Studio because SageMaker doesn't allow cross-region requests.

S3 bucket address

s3://dsoaws-amazon-reviews/autopilot/output

Select the machine learning problem type

- Auto
- Binary classification
- Regression
- Multiclass classification

Do you want to run a complete experiment?

- Yes
- No, run a pilot to create a notebook with candidate definitions

ADVANCED SETTINGS - *Optional*

Create Experiment

Figure 1-10. Configure the Autopilot Experiment.

We just need to provide a few input parameters:

- *Experiment name*: A name to identify the experiment, e.g. `amazon-customer-reviews`
- *Input data location*: The S3 path to our training data, e.g. `s3://dsoaws-amazon-reviews/data/amazon_reviews_us_Digital_Software_v1_00_header.csv`
- *Target attribute name*: The attribute (column) we want to predict, i.e. `star_rating`
- *Output data location*: The S3 path for storing the generated output, such as models and other artifacts, e.g. `s3://dsoaws-amazon-reviews/autopilot/output`
- *Problem type*: The machine learning problem type such as Binary classification, Regression, and Multiclass classification. The default, "Auto", allows Autopilot to choose for itself based on the given input data.
- *Run complete experiment*: We can choose to run a complete experiment or just generate the Data Exploration and Candidate Definition notebooks as part of the Data Analysis phase.

Let's click `Create Experiment` and start our first Autopilot job. You can observe the progress of the job in the UI as shown in Figure 1-11.

Figure 1-11. Progress of a running Autopilot job.

The UI shows the 3 stages of the Autopilot job: Analyzing Data, Feature Engineering, and Model Tuning. Once Autopilot completes the Analyzing Data stage, you can see the links to the two generated notebooks appearing in the UI as shown in Figure 1-12.

Figure 1-12. Autopilot > Analyzing Data > Generated notebooks

If you have a look at the S3 output bucket, you can find the generated notebooks, code, and transformed data in the following structure:

```
amazon-customer-reviews/
        data-processor-models/
                amazon-cus-dpp0-1-xxx/
output/model.tar.gz
                amazon-cus-dpp1-1-xxx/
output/model.tar.gz
```

```
...
        preprocessed-data/
                header/
headers.csv
tuning_data/
                train/
                                chunk_20.csv
                                chunk_21.csv
                                ...
                        validation/
                                chunk_0.csv
                                chunk_1.csv
                                ...
        sagemaker-automl-candidates/
pr-1-a1cd024ace184d2fa646384b014a98f3a13ef4fa0cd24f4ebe555ea005/
                generated_module/
                candidate_data_processors/
                                        dpp0.py
                                        dpp1.py
                                        ...
                        notebooks/
SageMakerAutopilotCandidateDefinitionNotebook.ipynb
                        SageMakerAutopilotDataExplorationNotebook.ipynb
                        ...
```

When the Feature Engineering stage starts, you will see SageMaker training jobs appearing in the AWS Console as shown in Figure 1-13.

Amazon SageMaker 〉 Training jobs

Training jobs

Actions ▼ Create training job

Search training jobs

〈 1 2 〉

Name	Creation time	Duration	Status
amazon-cus-dpp9-1-38f61e39ce5e4cc9a58b77b91b3d44e7059b9d934f814	Apr 30, 2020 15:41 UTC	3 minutes	Completed
amazon-cus-dpp8-1-61ef71e5fb8b4677bb760bfd47ba796015c1b2971c004	Apr 30, 2020 15:41 UTC	4 minutes	Completed
amazon-cus-dpp7-1-7c42dfaf045b4b3e8b14c794b466a84ee1dcb6f7d0864	Apr 30, 2020 15:41 UTC	3 minutes	Completed
amazon-cus-dpp6-1-63c01f386b8d4947aba70cd4ad32ae6580b47b8720454	Apr 30, 2020 15:41 UTC	3 minutes	Completed
amazon-cus-dpp5-1-8e2671a9f87e4a94b957ee6f1b8396d6e81a2add6b034	Apr 30, 2020 15:40 UTC	3 minutes	Completed
amazon-cus-dpp0-1-b94488732c254dc99134dbbcc46a74e7f4fd401acd0b4	Apr 30, 2020 15:34 UTC	3 minutes	Completed
amazon-cus-dpp4-1-48649a0ac66e4cf8bf6e78a1864981103da5239501e64	Apr 30, 2020 15:34 UTC	4 minutes	Completed
amazon-cus-dpp2-1-da406d02a9564fef82f059a729dc41c3c2c670e76c0d4	Apr 30, 2020 15:34 UTC	3 minutes	Completed
amazon-cus-dpp1-1-ca932ea1c4084f1fa2f8524e0d7cc3c17b884416ccc74	Apr 30, 2020 15:34 UTC	3 minutes	Completed
amazon-cus-dpp3-1-25d6b6f6908647de83aef3a3e59a651b4a753bc4a1334	Apr 30, 2020 15:34 UTC	3 minutes	Completed

Figure 1-13. Autopilot Feature Engineering > Performed By SageMaker Training Jobs

Each training job is a combination of a model candidate and the data preprocessor (dpp) code, named dpp0 through dpp9. You can think of those training jobs as the ten machine learning pipelines Autopilot built to find the best performing model. You can select any of those training jobs to view the job status, configuration, parameters, and log files.

Once the Feature Engineering stage has completed, we can view the transformed data directly in S3 grouped by pipeline. The data has been divided into smaller chunks and split into separate train and validation datasets which we will cover in more detail in the data preparation chapter.

```
transformed-data/
        dpp0/
                rpb/
                        train/
                                chunk_20.csv_out
                                chunk_21.csv_out
                                ...
                        validation/
                                chunk_0.csv_out
                                chunk_1.csv_out
                                ...
        dpp1/
                csv/
                        train/
                                chunk_20.csv_out
                                chunk_21.csv_out
                                ...
                        validation/
                                chunk_0.csv_out
                                chunk_1.csv_out
                                ...
        ..
        dpp9/
```

Finally, Autopilot runs the Model Tuning stage as shown in Figure 1-14. Now, we can actually see the trials appearing in the Autopilot UI.

amazon-customer-reviews X

less than 20 seconds ago

EXPERIMENT: AMAZON-CUSTOMER-REVIEWS

Open candidate generation notebook | Open data exploration notebook

✓ Analyzing Data → ✓ Feature Engineering → Model Tuning → Completed

Amazon SageMaker Autopilot is tuning the model.

If experiment is taking too long to run, you can stop the experiment

You can always return to this page later by choosing this experiment on the Experiments tab in the navigation panel.

Trials | Job profile

TRIALS

0 row selected

Deploy model

Trial name	Status	Start time	Objective
tuning-job-1-8fc3bb8155c645f282-010...	In Progress	1 minute ago	
tuning-job-1-8fc3bb8155c645f282-002...	In Progress	1 minute ago	
tuning-job-1-8fc3bb8155c645f282-006...	In Progress	1 minute ago	
tuning-job-1-8fc3bb8155c645f282-007...	In Progress	1 minute ago	
tuning-job-1-8fc3bb8155c645f282-001...	In Progress	1 minute ago	
tuning-job-1-8fc3bb8155c645f282-008...	In Progress	1 minute ago	
tuning-job-1-8fc3bb8155c645f282-004...	In Progress	1 minute ago	
tuning-job-1-8fc3bb8155c645f282-005...	In Progress	1 minute ago	
tuning-job-1-8fc3bb8155c645f282-003...	In Progress	2 minutes ago	
tuning-job-1-8fc3bb8155c645f282-009...	In Progress	2 minutes ago	

Figure 1-14. Autopilot Model Tuning Stage.

The Model Tuning creates a SageMaker Hyperparameter tuning job as shown in Figure 1-15. Amazon SageMaker automatic model tuning, also known as hyperparameter tuning (HPT), is another functionality of the SageMaker service.

Figure 1-15. SageMaker Hyperparameter Tuning Job

HPT finds the best version of a model by running many training jobs on your dataset using the algorithm and ranges of hyperparameters that you specify. You can find the corresponding training jobs listed in the SageMaker training jobs UI as shown in Figure 1-16.

Amazon SageMaker > Training jobs

Training jobs

Actions | Create training job

Search training jobs

1 2

Name	Creation time	Duration	Status
tuning-job-1-8fc3bb8155c645f282-020-e3d28caf	Apr 30, 2020 15:53 UTC	-	InProgress
tuning-job-1-8fc3bb8155c645f282-019-930c1c90	Apr 30, 2020 15:53 UTC	-	InProgress
tuning-job-1-8fc3bb8155c645f282-018-1052417e	Apr 30, 2020 15:53 UTC	-	InProgress
tuning-job-1-8fc3bb8155c645f282-017-22bd21d6	Apr 30, 2020 15:52 UTC	-	InProgress
tuning-job-1-8fc3bb8155c645f282-016-0e79f26a	Apr 30, 2020 15:52 UTC	-	InProgress
tuning-job-1-8fc3bb8155c645f282-015-f6347d06	Apr 30, 2020 15:52 UTC	-	InProgress
tuning-job-1-8fc3bb8155c645f282-014-21941557	Apr 30, 2020 15:52 UTC	-	InProgress
tuning-job-1-8fc3bb8155c645f282-013-7235ea51	Apr 30, 2020 15:52 UTC	-	InProgress
tuning-job-1-8fc3bb8155c645f282-012-1ea8b599	Apr 30, 2020 15:52 UTC	-	InProgress
tuning-job-1-8fc3bb8155c645f282-011-8a56aa9d	Apr 30, 2020 15:52 UTC	-	InProgress

Figure 1-16. SageMaker Training Jobs UI

Again, you can click and inspect any of these jobs to view the job status, configuration, parameters, and log files. Back in the Autopilot UI, we can inspect the trials.

The four Autopilot trial components make up a pipeline of the following jobs:

Processing Job:

Splits the data into train and validation data and separates the header data.

Training Job:

Trains a batch transformation model using the previously-split training and validation data with the data preprocessor code (dpp[0-9].py) for each model candidate.

Batch Transform Job:

Transforms the raw data into algorithm-specific features using the previously-generated batch transformation model.

Tuning Job:

Finds the best-performing model candidates using the previously-transformed algorithm-specific features by optimizing the algorithm configuration and parameters.

Those four components document all parameters as well as the input and output artifacts used, which ensures data lineage. After the model tuning step is completed, you can find the final outputs and model candidates in the S3 bucket organized per model candidate pipeline.

```
tuning/
        amazon-cus-dpp0-xgb/
                tuning-job-1-8fc3bb8155c645f282-001-da2b6b8b/
                        output/model.tar.gz
```

```
                tuning-job-1-8fc3bb8155c645f282-004-911d2130/
                        output/model.tar.gz
                tuning-job-1-8fc3bb8155c645f282-012-1ea8b599/
                        output/model.tar.gz
        …
        amazon-cus-dpp3-ll/
        …
amazon-cus-dpp-9-xgb/
```

Note that the pipeline name (i.e. `amazon-cus-dpp0-xgb`) conveniently contains the information of settings used (dpp0 = data preprocessor pipeline dpp0, xgb = chosen algorithm XGBoost). Autopilot also highlights the best performing model for you as shown in Figure 1-17.

Figure 1-17. Autopilot > Best Performing Model

Deploying this model into production is now as easy as a right click on the name and selecting `Deploy model` as shown in Figure 1-18.

amazon-customer-reviews

half a minute ago

EXPERIMENT: AMAZON-CUSTOMER-REVIEWS

Trials Job profile

TRIALS

1 row selected

Trial name	Status
Best: tuning-job-1-8fc3bb8155c645f	Completed
tuning-job-1-8fc3bb8155c645f2	
tuning-job-1-8fc3bb8155c645f2	
tuning-job-1-8fc3bb8155c645f2	
tuning-job-1-8fc3bb8155c645f2	
tuning-job-1-8fc3bb8155c645f282-249...	Completed

Open in trial component list

Open in trial details

Deploy model

Copy cell contents

Shift+Right Click for Browser Menu

Figure 1-18. Autopilot > Deploy model

We just need to give our endpoint a name, select the AWS instance type we want to deploy the model on, e.g. `ml.m5.xlarge` and define the number of instances to start with as shown in Figure 1-19.

amazon-customer-reviews X | Deploy model X

Deploy model

REQUIRED SETTINGS

Endpoint name

reviews-endpoint

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

Instance type	Instance count
ml.m5.xlarge	1

Data capture

SageMaker Studio will save prediction requests and responses from the endpoint to an Amazon S3 location specified below

- [ ] Save prediction requests
- [ ] Save prediction responses

ADVANCED SETTINGS - *Optional*

Deploy model

Figure 1-19. Autopilot > Deploy Model > Configuration

> **NOTE**
>
> You can find an overview of all AWS instance types supported by Amazon SageMaker and their performance characteristics here: https://aws.amazon.com/sagemaker/pricing/instance-types/. Note that those instances start with ml. in their name.

Optionally, you can enable data capture of all prediction requests and responses for your deployed model. We can now click on `Deploy model` and watch our model endpoint being created. Once the endpoint shows up as `In Service` as shown in Figure 1-20, we can invoke the endpoint to serve predictions.

Figure 1-20. Autopilot > Endpoints

Here is a simple Python code snippet to invoke the endpoint. We pass a sample review ("I loved it!") and see which star rating our model chooses. Remember, star rating 1 is the worst and star rating 5 is the best.

```
import boto3
sagemaker_runtime = boto3.client('sagemaker-runtime')
csv_line_predict = """I loved it!"""
ep_name = 'reviews-endpoint'
response = sagemaker_runtime.invoke_endpoint(EndpointName=ep_name,
ContentType='text/csv', Accept='text/csv', Body=csv_line_predict)
response_body = response['Body'].read().decode('utf-8').strip()
Let's print the response and see which star rating our model predicts:
response_body
"5"
```

Good job. Our model classifies the review as a 5-star rating.

Train and Deploy with Autopilot Python SDK

If you prefer to interact with AWS services in a programmatic way, you can use the AWS SDK for Python boto3 [https://boto3.amazonaws.com/v1/documentation/api/latest/index.html], to interact with AWS services from your Python development environment. In addition, we are going to use the SageMaker Python SDK [https://sagemaker.readthedocs.io/en/stable/] to complement the boto3 Python SDK. Here is a sample setup:

```
import boto3
import sagemaker
session = sagemaker.Session(default_bucket="dsoaws-amazon-reviews")
bucket = session.default_bucket()
role = sagemaker.get_execution_role()
region = boto3.Session().region_name
sm = boto3.Session().client(service_name='sagemaker',
 region_name=region)
```

One advantage of using the SDK is that we have access to more Autopilot configuration options. For example, we can specify the number of model candidates to explore. We can also set a maximum runtime in seconds for each training job and the overall Autopilot job.

```
max_candidates = 3
job_config = {
```

```
    'CompletionCriteria': {
      'MaxRuntimePerTrainingJobInSeconds': 600,
      'MaxCandidates': max_candidates,
      'MaxAutoMLJobRuntimeInSeconds': 3600
    },
}
```

Similar to the Autopilot UI configuration, we provide a S3 input and output location and define the target attribute for predictions.

```
input_data_config = [{
      'DataSource': {
        'S3DataSource': {
          'S3DataType': 'S3Prefix',
          'S3Uri': 's3://dsoaws-amazon-
reviews/data/amazon_reviews_us_Digital_Software_v1_00_header.csv'
        }
      },
      'TargetAttributeName': 'star_rating'
    }
]
output_data_config = {
    'S3OutputPath': 's3://dsoaws-amazon-reviews/autopilot/output/'
}
```

Next, we create our Autopilot job. Note that we add a timestamp to the Autopilot job name which helps to keep the jobs unique and easy to track. We pass the job name, input/output configuration, job configuration and execution role. The execution role is part of the AWS Identity and Access Management (IAM) service and manages service access permissions.

```
from time import gmtime, strftime, sleep
timestamp_suffix = strftime('%d-%H-%M-%S', gmtime())
auto_ml_job_name = 'automl-dm-' + timestamp_suffix
sm.create_auto_ml_job(AutoMLJobName=auto_ml_job_name,
                      InputDataConfig=input_data_config,
                      OutputDataConfig=output_data_config,
                      AutoMLJobConfig=job_config,
                      RoleArn=role)
```

This should create an output similar to this:

```
{'AutoMLJobArn': 'arn:aws:sagemaker:<region>:<account_id>:automl-job/automl-dm-
19-13-07-41',
 'ResponseMetadata': {'RequestId': '8508d54f-1910-481b-ab42-9c46d110959e',
  'HTTPStatusCode': 200,
```

```
  'HTTPHeaders': {'x-amzn-requestid': '8508d54f-1910-481b-ab42-9c46d110959e',
   'content-type': 'application/x-amz-json-1.1',
   'content-length': '92',
   'date': 'Tue, 19 May 2020 13:07:40 GMT'},
  'RetryAttempts': 0}}
```

The Autopilot job has been created with a unique identifier described as `AutoMLJobArn` above. ARN stands for "Amazon Resource Name" and is often encoded in the form of `arn:partition:service:region:account-id:resource-id`. ARNs are used across all AWS services to specify resources unambiguously.

We can poll the Autopilot job status and check if the data analysis step has completed:

```
job = sm.describe_auto_ml_job(AutoMLJobName=auto_ml_job_name)
job_status = job['AutoMLJobStatus']
job_sec_status = job['AutoMLJobSecondaryStatus']
if job_status not in ('Stopped', 'Failed'):
    while job_status in ('InProgress') and job_sec_status in ('AnalyzingData'):
        job = sm.describe_auto_ml_job(AutoMLJobName=auto_ml_job_name)
        job_status = job['AutoMLJobStatus']
        job_sec_status = job['AutoMLJobSecondaryStatus']
        print(job_status, job_sec_status)
        sleep(30)
    print("Data analysis complete")

print(job)
```

The code will return the following output (shortened):

```
InProgress AnalyzingData
InProgress AnalyzingData
InProgress AnalyzingData
...
Data analysis complete
```

Similarly, you can query for `job_sec_status in ('FeatureEngineering')` and `job_sec_status in ('ModelTuning')` for the two following Autopilot steps.

Once the Autopilot job has finished, you can list all model candidates:

```
candidates = sm.list_candidates_for_auto_ml_job(AutoMLJobName=auto_ml_job_name,
SortBy='FinalObjectiveMetricValue')['Candidates']
for index, candidate in enumerate(candidates):
```

```
    print(str(index) + "  "
        + candidate['CandidateName'] + "  "
        + str(candidate['FinalAutoMLJobObjectiveMetric']['Value']))
```

This will generate output similar to this:

```
0  tuning-job-1-655f4ef810d441d4a8-003-b80f5233  0.4437510073184967
1  tuning-job-1-655f4ef810d441d4a8-001-7c10cb15  0.29365700483322144
2  tuning-job-1-655f4ef810d441d4a8-002-31088991  0.2874149978160858
```

You can also retrieve the best candidate:

```
best_candidate = sm.describe_auto_ml_job(AutoMLJobName=auto_ml_job_name)
['BestCandidate']
best_candidate_identifier = best_candidate['CandidateName']
print("Candidate name: " + best_candidate_identifier)
print("Metric name: " + best_candidate['FinalAutoMLJobObjectiveMetric']
['MetricName'])
print("Metric value: " + str(best_candidate['FinalAutoMLJobObjectiveMetric']
['Value']))
```

This will generate output similar to this:

```
Candidate name: tuning-job-1-655f4ef810d441d4a8-003-b80f5233
Metric name: validation:accuracy
Metric value: 0.4437510073184967
```

Now, let's deploy the best model as a REST endpoint. First, we need to create a model object:

```
model_name = 'automl-dm-model-' + timestamp_suffix
model_arn = sm.create_model(Containers=best_candidate['InferenceContainers'],
                            ModelName=model_name,
                            ExecutionRoleArn=role)
print('Best candidate model ARN: ', model_arn['ModelArn'])
```

The output should look similar to this:

```
Best candidate model ARN:  arn:aws:sagemaker:<region>:<account_id>:model/automl-
dm-model-01-16-34-00
```

The code above reveals another detail that has been hidden in the UI. When we deploy our model as a REST endpoint, we actually deploy a whole inference pipeline. The inference pipeline consists of three containers.

Data Transformation Container:

This is essentially the "request handler" that converts the application inputs (e.g. `review_body`) into a format recognized by the model (ie. numpy arrays or tensors). The container hosts the model which Autopilot trained for the feature engineering step.

Algorithm Container:

This container hosts the actual model that serves the predictions.

Inverse Label Transformer Container: T

his is the "response handler" that converts the algorithm-specific output (ie. numpy arrays or tensors) back into a format recognized by the invoker (e.g. `star_rating`.)

Figure 1-21 shows an example of the inference pipeline.

Figure 1-21. Autopilot deploys a model as an inference pipeline.

We pass our reviews as raw text and the data transformation container converts the text into TF/IDF embeddings. TF/IDF stands for term frequency - inverse document frequency. TF/IDF encodes the relevance of a word to a document in a collection of documents. The algorithm container processes the input and predicts the star rating. Note that the algorithm in our example returns the prediction results as a 0-based index value. The task of the inverse label transformer container is to map the index (0,1,2,3,4) to the correct star rating label (1,2,3,4,5).

To deploy the inference pipeline, we need to create an endpoint configuration:

```
# EndpointConfig name
timestamp_suffix = strftime('%d-%H-%M-%S', gmtime())
epc_name = 'automl-dm-epc-' + timestamp_suffix
# Endpoint name
ep_name = 'automl-dm-ep-' + timestamp_suffix
variant_name = 'automl-dm-variant-' + timestamp_suffix
ep_config = sm.create_endpoint_config(
EndpointConfigName = epc_name,
        ProductionVariants=[{'InstanceType':'ml.c5.2xlarge',
'InitialInstanceCount': 1,
'ModelName': model_name,
'VariantName': variant_name}])
create_endpoint_response = sm.create_endpoint(
EndpointName=ep_name,
EndpointConfigName=epc_name)
```

Autopilot is now deploying the inference pipeline. Let's query for the endpoint status to see when the pipeline is successfully in service:

```
response = sm.describe_endpoint(EndpointName=autopilot_endpoint_name)
status = response['EndpointStatus']
print("Arn: " + response['EndpointArn'])
print("Status: " + status)
```

After a couple of minutes, the output should look similar to this:

```
Arn: arn:aws:sagemaker:<region>:<account_id>:endpoint/automl-dm-ep-19-13-29-52
Status: InService
```

We can now invoke the endpoint and run a sample prediction. We pass the review "It's OK." to see which star rating class the model predicts:

```
sagemaker_runtime = boto3.client('sagemaker-runtime')
csv_line_predict = """It's OK."""
response = sagemaker_runtime.invoke_endpoint(EndpointName=ep_name,
ContentType='text/csv', Accept='text/csv', Body=csv_line_predict)
response_body = response['Body'].read().decode('utf-8').strip()
```

Let's print the response:

```
response_body
'3'
```

Our endpoint has successfully classified this sample review as a 3-star rating.

> **NOTE**
>
> You can check the S3 output location again for all generated models, code and other artifacts, including the Data Exploration notebook and Candidate Definition notebook.

Invoking our models using the SageMaker SDK is just one option. There are many more service integrations available in AWS. In the next section, we describe how you can run real-time predictions from within a SQL query using Amazon Athena.

Predict with Amazon Athena

Amazon Athena is an interactive query service which lets you analyze data stored in S3 using standard SQL. To call SageMaker from Athena, we need to define a function with the `USING FUNCTION` clause as shown in Figure 1-22. Any subsequent `SELECT` statement can then reference the function to invoke a model prediction.

Figure 1-22. Amazon Athena ML Integrations.

Here's a simple SQL query which selects product reviews stored in an Athena table called `dsaws.product_reviews`. The function

`predict_star_rating` then calls a SageMaker endpoint with the name reviews to serve the prediction.

```
USING FUNCTION predict_star_rating(review_body VARCHAR)
    RETURNS VARCHAR TYPE
    SAGEMAKER_INVOKE_ENDPOINT WITH (sagemaker_endpoint = 'reviews')
SELECT review_id, review_body,
                predict_star_rating(REPLACE(review_body, ',', ' '))
                AS predicted_star_rating
FROM dsoaws.product_reviews
```

The result should look similar to this (shortened):

review_id	review_body	predicted_star_rating
R23CFDQ6SLMET	The photographs of this book is a let down. I ...	1
R1301KYAYKX8FU	This is my go-to commentary for all of my semi...	5
R1CKM3AKI920D7	I can't believe I get to be the first to tell ...	5
RA6CYWHAHSR9H	There's Something About Christmas / Debbie Mac...	5
R1T1CCMH2N9LBJ	This revised edition by Murray Straus is an ex...	1
...	...	...

This example shows how easy it is to enrich your S3 data with machine learning prediction results using a simple SQL query.

Automated Machine Learning with Comprehend

Amazon Comprehend is a fully-managed AI service for natural language processing (NLP) tasks. Comprehend takes text documents as input and recognizes entities, key phrases, language, and sentiment. Comprehend also offers built-in functionality to create custom models using automated machine learning.

Predict with Comprehend Built-In Model

Sentiment analysis is a text classification task which predicts positive, negative, or neutral sentiment of a given input text. This is extremely helpful if you want to analyze product reviews and identify quality issues.

Let's implement this text classifier with Amazon Comprehend. As input data, we leverage a subset of Amazon's public customer reviews dataset. We want Amazon Comprehend to classify the sentiment of a provided review. The Comprehend UI is the easiest way to get started. You can paste in any text and Comprehend will analyze the input in real-time using the built-in model. Let's test this with a sample product review such as "I loved it! I will recommend this to everyone." as shown in Figure 1-23.

aws Services Resource Groups

Amazon Comprehend

Real-time analysis

Analysis jobs

Customization

Custom classification

Custom entity recognition

Amazon Comprehend Medical

Real-time analysis

Analysis jobs

Real-time analysis Info

Amazon Comprehend enables real-time analysis using either built-in or custom models. With built-in models, you can recognize entities, extract key phrases, detect dominant languages, analyze syntax, or determine sentiment. With custom models, create an endpoint to classify documents using custom categories or labels. For documents relating to healthcare, use Amazon Comprehend Medical.

Overview

Real-time analysis with built-in models

Use Amazon Comprehend built-in models to analyze content in real-time.

Custom real-time analysis with endpoint

Alternatively, create an endpoint for your custom model to provide Custom Classification insights in real-time.

Input text

Supported languages

Analysis type

- (●) Built-in
  View real-time insights based on AWS built-in models
- ( ) Custom
  View real-time insights based on custom models from an endpoint you've created

Input text

I loved it! I will recommend this to everyone.

48 of 5000 characters used.

Clear text | Analyze

Figure 1-23. Amazon Comprehend > Using the built-in model for sentiment analysis.

After clicking `Analyze` we see the results under `Insights` as shown in Figure 2.24.

Insights Info

Entities | Key phrases | Language | Sentiment | Syntax

Analyzed text

I loved it! I will recommend this to everyone.

Results

Sentiment

Neutral	Positive	Negative	Mixed
0.00 confidence	0.99 confidence	0.00 confidence	0.00 confidence

Application integration

Figure 1-24. Amazon Comprehend > Sentiment analysis results shown in the console.

The score tells us that Comprehend is 99% confident that our sample review has a positive sentiment. Now let's implement a custom model which classifies our product reviews into star ratings again.

Train and Deploy Comprehend Custom Model with Comprehend UI

Comprehend Custom is another example of automated machine learning that enables the practitioner to fine-tune Comprehend's built-in model to a specific dataset. Let's re-use the Amazon Customer Reviews Dataset file from the previous Autopilot example as our training data:

```
star_rating,review_body
5,"GOOD, GREAT, WONDERFUL, MORE THAN ADEQUATE AND EXACTLY WHAT I  NEED FOR MY
COMPUTER. NONE BETTER OR GOOD TO USE."
2,"Even though it does the same job as TurboTax, it isn't as user friendly as
TurboTax. I guess you get what you paid for."
4,"Pretty easy to use. No issues."
…
```

In the Comprehend UI, click on `Custom classification` and then on `Train classifier` which opens a dialog as shown in Figure 1-25.

aws Services Resource Groups

Amazon Comprehend

- Real-time analysis
- Analysis jobs
- Customization
  - Custom classification
  - Custom entity recognition
- Amazon Comprehend Medical
  - Real-time analysis
  - Analysis jobs

Amazon Comprehend > Custom classification > Train classifier

Train classifier Info

Classifier settings

Name

Amazon-Customer-Reviews-Classifier

The name can be from 1 to 64 characters. Valid characters are a-z, A-Z, 0-9 and hyphen (-).

Language

English

Classifier encryption Info

Training data Info

Classifier mode

Using multi-class mode

Training document file must have one class and document per line.

Example

COMEDY	document text 1
COMEDY	document text 2
DRAMA	document text 3

File format: csv. Must be at least 10 documents per class.

Using multi-label mode

Training document file must have one or more classes per line, and one document per line.

Example

COMEDY	document text 1
DRAMA	document text 2
COMEDY | DRAMA	document text 3

File format: csv. Must be at least 10 documents per class.

S3 location Info

Paste the URL of an input data file in S3, or select a bucket or folder location in S3

s3:// /data/amazon_reviews_us_Digital_Software

Browse S3

Output data - *optional* Info

S3 location

Paste the URL of a bucket or folder location in S3, or select a bucket or folder location in S3

s3:// /models/comprehend/output/

Browse S3

Encryption Info

Access permissions Info

IAM role

Use an existing IAM role

Create an IAM role

Role name

A role that grants access to the S3 input and output locations.

DSOAWS_Comprehend

VPC settings - *optional*

Use a VPC to restrict the data that can be uploaded to, or downloaded from, an S3 bucket that you use with Amazon Comprehend.

Tags - *optional* Info

A tag is a label that you can add to a resource as metadata to help you organize, search, or filter your data. Each tag consists of a key and an optional value.

Cancel

Train classifier

Figure 1-25. Amazon Comprehend > Custom Classification > Train Classifier

First, we provide a name for the custom classifier, select multi-class mode, and put in the path to the training data. Next, we define the S3 location to store the trained model outputs and select an IAM role with permissions to access that S3 location. Then, we click `Train classifier` to start the training process.

You can now see the custom classifier showing up in the UI with the status `Submitted` and shortly after with the status `Training` as shown in Figure 1-26.

Figure 1-26. Amazon Comprehend > Custom Classifier submitted.

Once the classifier shows up as `Trained`, we can deploy it as a Comprehend Endpoint to serve predictions. Simply select the trained model and click on `Actions`, followed by `Create Endpoint`. Give the endpoint a name and click on `Create Endpoint`. In the Comprehend UI, navigate to `Real-time analysis` and select the analysis type `Custom`. Select the custom endpoint from the endpoint drop-down list. Comprehend can now analyze input text using the custom text classifier model. Let's paste in the review "Really bad. I hope they don't make this anymore." and click `Analyze`.

You can see in the results that our custom model now classifies input text into the star ratings from 1 to 5 (with 5 being the best rating). In this example, the model is 76% confident that the review classifies as star rating 2 as shown in Figure 1-27.

Insights Info

Analyzed text

Really bad. I hope they don't make this anymore.

Results

Classes

2	1	3
0.76 confidence	0.13 confidence	0.09 confidence

Application integration

Figure 1-27. Amazon Comprehend > Custom classifier predicts star rating.

In just a few clicks, we trained a Comprehend Custom model on the Amazon Customer Reviews Dataset to predict star rating from review text. That is the power of Amazon AI services.

Train and Deploy Comprehend Custom Model with the Comprehend Python SDK

You can also interact programmatically with Comprehend. Let's use the Comprehend Python SDK to train and deploy the custom classifier:

```
import boto3
comprehend = boto3.client('comprehend')
```

Now we can simply run the following lines of code to start the training of our custom classifier:

```
# Create a unique timestamp ID to attach to our training job name
import datetime
id = str(datetime.datetime.now().strftime("%s"))
# Start training job
training_job = comprehend.create_document_classifier(
    DocumentClassifierName='Amazon-Customer-Reviews-Classifier-'+ id,
    DataAccessRoleArn=iam_role_comprehend_arn,
    InputDataConfig={
        'S3Uri': 's3://<your_bucket>/<path>/amazon_reviews.csv'
    },
    OutputDataConfig={
        'S3Uri': 's3://<your_bucket>/<path>/model/outputs'
    },
    LanguageCode='en'
)
```

The input parameters are as follows:

`DocumentClassifierName`:

The name of the custom model.

`DataAccessRoleArn`:

The Amazon Resource Name (ARN) of the IAM role that grants Amazon Comprehend read access to your input data.

`InputDataConfig`:

Specifies the format and location of the training data.

`S3Uri`: S3 path to the training data.

`OutputDataConfig`:

Specifies the location of the model outputs.

`S3Uri`: S3 path for model outputs.

`LanguageCode`:

The language of the training data.

The training job will now run for some time depending on the amount of training data to process. Once it is finished, we can deploy an endpoint with our custom classifier to serve predictions.

To deploy the custom model, let's first find out the ARN of the model which we need to reference:

```
model_arn = training_job['DocumentClassifierArn']
```

With the `model_arn`, we can now create a model endpoint:

```
inference_endpoint_response = comprehend.create_endpoint(
    EndpointName='comprehend-inference-endpoint',
    ModelArn = model_arn,
    DesiredInferenceUnits = 1
)
```

The input parameters are as follows:

`EndpointName`:

A name for our endpoint.

`ModelArn`:

The Amazon Resource Number (ARN) of the model to which the endpoint will be attached.

`DesiredInferenceUnits`:

The desired number of inference units to be used by the model attached to this endpoint. Each inference unit represents a throughput of 100 characters per second.

Once the model endpoint is successfully created and In Service, we can invoke it for a sample prediction. To invoke the custom model, let's find out the ARN of our endpoint:

```
endpoint_arn = inference_endpoint_response["EndpointArn"]
```

We can now run a prediction using `comprehend.classify_document()` along with text we want to classify and the ARN of our endpoint:

```
# Sample text to classify
txt = """It's OK."""
response = comprehend.classify_document(
    Text= txt,
    EndpointArn = endpoint_arn
)
```

The JSON-formatted response will look similar to this:


```
{
  "Classes": [
    {
      "Name": "3",
      "Score": 0.977475643157959
    },
    {
      "Name": "4",
      "Score": 0.021228035911917686
    },
    {
      "Name": "2",
      "Score": 0.001270478474907577
    }
  ],
...
}
```

Our custom classifier is 97% confident that our sample review deserves a 3-star rating.

With just a few lines of Python code, we trained a Comprehend Custom model on the Amazon Customer Reviews Dataset to predict star rating from review text.

Summary

In this chapter, we discussed the concept of automated machine learning (AutoML). We introduced you to SageMaker Autopilot's unique white-box approach to AutoML. Autopilot offloads the heavy lifting of building ML pipelines while providing full visibility into the automated process. You learned how to invoke machine learning models from SQL queries using Amazon Athena. We also showed how Amazon Comprehend uses AutoML to train and deploy a custom text classification model based on our dataset in just a few clicks or lines of Python code.

In the next chapter, we will discuss the concept of Augmented AI and active learning. You will learn how to integrate human-in-the-loop workflows into your AI applications to automate data labeling and improve model accuracy.

Chapter 2. Ingesting Data Into The Cloud

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *support@pipeline.ai*.

In this chapter, we will show you how to ingest data into the cloud. For that purpose, we will look at a typical scenario in which an application writes files into an Amazon S3 Data Lake which in turn need to be accessed by both the ML Engineering / Data Science team, as well as the Business Intelligence / Data Analyst team as shown in Figure 3-1 below.

Figure 2-1. Chapter Overview > Data is being written to S3, and accessed by different teams.

Amazon Simple Storage Service, short Amazon S3, is object storage to store and retrieve any amount of data. It offers you extreme durability, high availability, and infinitely scalable data storage infrastructure at very low costs. Hence, it is the perfect foundation for data lakes, and for storing all of

your Machine Learning data - whether it's training and test datasets, model code, or even the model files themselves. You will learn more about the advantages of building data lakes on Amazon S3 in the next section.

Let's assume our application continually captures some form of log data (i.e. customer interactions on our website) and writes it in tabular-separated file format (TSV) into S3.

As a data scientist or machine learning engineer, you want to have access to all of the raw data, and be able to quickly explore it. We will introduce you to Amazon Athena and show you how to leverage Athena as an interactive query service to analyze data in S3 using standard SQL, without moving the data. In the first step, we will register the TSV data in our S3 bucket with Athena, and then run some ad-hoc queries on the dataset. We will also show how you can easily convert the TSV data into the more query-optimized, columnar file format Apache Parquet.

Your business intelligence team and data analysts might also want to have a subset of the data in a data warehouse which they can then transform, and query with their standard SQL clients to create reports and visualize trends. We will introduce you to Amazon Redshift, a fully managed data warehouse service, and show you how to insert TSV data into Redshift, as well as combine the data warehouse queries with the less frequently accessed data that's still in our S3 data lake via Amazon Redshift Spectrum. Your Business Intelligence team can also use Amazon Redshift's data lake export functionality to unload (transformed, enriched) data back into our S3 data lake in Parquet file format.

We will also discuss key services and functionalities that can help you secure your data, in particular using the AWS Identity & Access Management (IAM) service, the Key Management System (KMS) for encryption, and leveraging S3 Access Points to control access to data in shared S3 buckets.

We will conclude this chapter with some tips and tricks on how you can increase performance using compression algorithms, and reduce cost by leveraging S3 Intelligent Tiering.

NOTE

In order for you to run the code examples in your own environment, make sure you have access to an AWS account with corresponding privileges. Please see Appendix C on how to setup a new AWS account, and the AWS services discussed in this book.

Data Lakes

In the previous chapter, we discussed the democratization of Artificial Intelligence and Data Science over the last few years, the explosion of data, and how cloud services provide the infrastructure agility to store and process data of any amount.

Yet, in order to use all this data efficiently, companies are tasked to break down existing data silos, find ways to analyze very diverse datasets, dealing with both structured and unstructured data, while ensuring highest standards of data governance, data security and compliance with privacy regulations. These (big) data challenges set the stage for data lakes.

As illustrated in Figure 3-2, a data lake is a centralized and secure repository that enables you to store, govern, discover and share data at any scale. One of the biggest advantages of data lakes is that you don't need to pre-define any schemas. You can store your raw data at scale and then decide later in which ways you need to process and analyze it. Data Lakes may contain structured relational data, files, and any form of semi-structured and unstructured data. You can also ingest data in real time.

Figure 2-2. Data Lake.

Data lakes provide a perfect base for Data Science and Machine Learning, as they give you access to large and diverse datasets to train and deploy more accurate models.

Building a data lake typically consists of the following (high-level) steps as shown in Figure 3-3.

1. Setup storage.
2. Move your data.

3. Cleanse, prepare and catalog the data.

4. Configure and enforce security and compliance policies.

5. Make data available for analytics.

Figure 2-3. Typical steps of building a data lake.

Each of those steps involves a range of tools and technologies, and while you can build a data lake manually from the ground up, there are cloud services available to help you streamline this process, i.e. AWS Lake Formation.

Lake Formation helps you to collect and catalog data from databases and object storage, move the data into your Amazon S3 data lake, clean and classify your data using machine learning algorithms, and secure access to your sensitive data.

> **NOTE**
>
> You can find more information on AWS Lake Formation here: https://aws.amazon.com/lake-formation

From a data analysis perspective, another key benefit of storing your data in Amazon S3 is, that it shortens the "time to insight' dramatically, as you can run ad-hoc queries directly on the data in S3, and you don't have to go through complex ETL (Extract-Transform-Load) processes and data pipelines to get your data into traditional enterprise data warehouses, as we will see in the next sections of this chapter.

Import Your Data into the S3 Data Lake

We are now ready to import our data into S3. We have chosen the Amazon Customer Reviews Dataset as the primary dataset for this book.

The Amazon Customer Reviews Dataset consists of more than 130+ million of customer reviews of products across 43 different product categories on the Amazon.com website from 1995 until 2015. It is a great resource for demonstrating Machine Learning concepts in general, and Natural Language Processing (NLP) techniques in particular.

You have likely seen these customer reviews on the Amazon.com website before when contemplating whether to purchase a certain product via the Amazon.com marketplace. Figure 3-4 shows the product reviews section on the Amazon.com website for an Amazon Echo Dot device.

amazon

All

EN

Hello, Sign in
Account & Lists

Returns
& Orders

0
Cart

Hello
Select your address

Today's Deals | Customer Service | Registry | Gift Cards | Sell

Amazon Devices | Echo & Alexa | Fire Tablets | Fire TV | Kindle | Home Security | Smart Home | Device Deals | Accessories | Pre-Owned | Device Support | Manage Your Content and Devices

Echo Dot (3rd Gen) - Smart speaker with Alexa - Charcoal

by Amazon

4.7 | 327,547 ratings | 1000+ answered questions

Amazon's Choice for "echo dot"

List Price: ~~$49.99~~

Price: $29.99 & FREE Shipping. Details

You Save: $20.00 (40%)

In Stock.

Want it Thursday, April 16? Choose **Two-Day Shipping** at checkout. Details

Ships from and sold by Amazon.com Services LLC.

Style: **Echo Dot**

Echo Dot | Echo Dot Kids Edition

Echo Dot with clock (Sandstone only)

Color: **Charcoal**

Configuration: **Device only**

Device only | with $10 Smart Plug | with $5 smart bulb

Share

Buy new: $29.99

Upgrade and save with Trade-In

Qty: 1

Add to Cart

Buy Now

Secure transaction

This is a gift

Link to my Amazon account to simplify setup. Why is this important?

Select delivery location

Buy used: $26.39

Add to List

Figure 2-4. Product reviews section on the Amazon.com website.

Let's have a closer look at our dataset and its schema.

Describe the Dataset

Customer reviews are one of Amazon's most valuable tools they offer customers for making informed purchase decisions.

In Amazon's annual shareholder letters, Jeff Bezos (Amazon Founder and CEO) regularly elaborates on the importance of "word of mouth" as a customer acquisition tool, and his love for "customers' constant discontent" as he calls it:

> *"We now offer customers … vastly more reviews, content, browsing options, and recommendation features. … Word of mouth remains the most powerful customer acquisition tool we have, and we are grateful for the trust our customers have placed in us. Repeat purchases and word of mouth have combined to make Amazon.com the market leader in online bookselling."*
>
> —Jeff Bezos, 1997 Shareholder ("Share Owner") Letter

Here is the schema for the dataset:

marketplace

2-letter country code (in this case all "US").

customer_id

Random identifier that can be used to aggregate reviews written by a single author.

review_id

A unique ID for the review.

product_id

The Amazon Standard Identification Number (ASIN).

http://www.amazon.com/dp/<ASIN> links to the product's detail page.

product_parent

he parent of that ASIN. Multiple ASINs (color or format variations of the same product) can roll up into a single product parent.

product_title

Title description of the product.

product_category

Broad product category that can be used to group reviews.

star_rating

The review's rating (1 to 5 stars).

helpful_votes

Number of helpful votes for the review.

total_votes

Number of total votes the review received.

vine

Was the review written as part of the Vine program?

verified_purchase

Was the review from a verified purchase?

review_headline

The title of the review itself.

review_body

The text of the review.

review_date

The date the review was written.

The dataset is shared in a public Amazon S3 bucket, and is available in two file formats:

- Tab separated value (TSV), a text format - s3://amazon-reviews-pds/tsv/
- Parquet, an optimized columnar binary format - s3://amazon-reviews-pds/parquet/

The Parquet dataset is partitioned (divided into subfolders) by the column product_category to further improve query performance. With this, you can use a WHERE clause on product_category in your SQL queries to only read data specific to that category.

We can use the AWS Command Line Interface (CLI) to list the S3 bucket content using the following CLI commands:

- aws s3 ls s3://amazon-reviews-pds/tsv/
- aws s3 ls s3://amazon-reviews-pds/parquet/

NOTE

The AWS CLI tool provides a unified command line interface to Amazon Web Services. You can find more information on how to install and configure the tool here: https://aws.amazon.com/cli/.

The listings below show us the available dataset files in TSV format, and the Parquet partitioning folder structure.

Dataset Files in TSV Format

```
2017-11-24 13:48:03  241896005 amazon_reviews_multilingual_DE_v1_00.tsv.gz
2017-11-24 13:48:17   70583516 amazon_reviews_multilingual_FR_v1_00.tsv.gz
2017-11-24 13:48:34   94688992 amazon_reviews_multilingual_JP_v1_00.tsv.gz
2017-11-24 13:49:14  349370868 amazon_reviews_multilingual_UK_v1_00.tsv.gz
2017-11-24 13:48:47 1466965039 amazon_reviews_multilingual_US_v1_00.tsv.gz
2017-11-24 13:49:53  648641286 amazon_reviews_us_Apparel_v1_00.tsv.gz
2017-11-24 13:56:36  582145299 amazon_reviews_us_Automotive_v1_00.tsv.gz
2017-11-24 14:04:02  357392893 amazon_reviews_us_Baby_v1_00.tsv.gz
2017-11-24 14:08:11  914070021 amazon_reviews_us_Beauty_v1_00.tsv.gz
2017-11-24 14:17:41 2740337188 amazon_reviews_us_Books_v1_00.tsv.gz
2017-11-24 14:45:50 2692708591 amazon_reviews_us_Books_v1_01.tsv.gz
2017-11-24 15:10:21 1329539135 amazon_reviews_us_Books_v1_02.tsv.gz
2017-11-24 15:22:13  442653086 amazon_reviews_us_Camera_v1_00.tsv.gz
2017-11-24 15:27:13 2689739299
amazon_reviews_us_Digital_Ebook_Purchase_v1_00.tsv.gz
2017-11-24 15:49:13 1294879074
amazon_reviews_us_Digital_Ebook_Purchase_v1_01.tsv.gz
2017-11-24 15:59:24  253570168
amazon_reviews_us_Digital_Music_Purchase_v1_00.tsv.gz
2017-11-24 16:01:47   18997559 amazon_reviews_us_Digital_Software_v1_00.tsv.gz
2017-11-24 16:01:53  506979922
amazon_reviews_us_Digital_Video_Download_v1_00.tsv.gz
2017-11-24 16:06:31   27442648 amazon_reviews_us_Digital_Video_Games_v1_00.tsv.gz
2017-11-24 16:06:42  698828243 amazon_reviews_us_Electronics_v1_00.tsv.gz
2017-11-24 16:12:44  148982796 amazon_reviews_us_Furniture_v1_00.tsv.gz
2017-11-24 16:13:52   12134676 amazon_reviews_us_Gift_Card_v1_00.tsv.gz
2017-11-24 16:13:59  401337166 amazon_reviews_us_Grocery_v1_00.tsv.gz
2017-11-24 19:55:29 1011180212
amazon_reviews_us_Health_Personal_Care_v1_00.tsv.gz
2017-11-24 20:30:55  193168458 amazon_reviews_us_Home_Entertainment_v1_00.tsv.gz
2017-11-24 20:37:56  503339178 amazon_reviews_us_Home_Improvement_v1_00.tsv.gz
2017-11-24 20:55:43 1081002012 amazon_reviews_us_Home_v1_00.tsv.gz
2017-11-24 21:47:51  247022254 amazon_reviews_us_Jewelry_v1_00.tsv.gz
2017-11-24 21:59:56  930744854 amazon_reviews_us_Kitchen_v1_00.tsv.gz
2017-11-24 23:41:48  486772662 amazon_reviews_us_Lawn_and_Garden_v1_00.tsv.gz
2017-11-24 23:59:42   60320191 amazon_reviews_us_Luggage_v1_00.tsv.gz
2017-11-25 00:01:59   24359816 amazon_reviews_us_Major_Appliances_v1_00.tsv.gz
2017-11-25 00:02:45  557959415 amazon_reviews_us_Mobile_Apps_v1_00.tsv.gz
2017-11-25 00:22:19   22870508 amazon_reviews_us_Mobile_Electronics_v1_00.tsv.gz
2017-11-25 00:23:06 1521994296 amazon_reviews_us_Music_v1_00.tsv.gz
2017-11-25 00:58:36  193389086 amazon_reviews_us_Musical_Instruments_v1_00.tsv.gz
2017-11-25 01:03:14  512323500 amazon_reviews_us_Office_Products_v1_00.tsv.gz
2017-11-25 07:21:21  448963100 amazon_reviews_us_Outdoors_v1_00.tsv.gz
2017-11-25 07:32:46 1512903923 amazon_reviews_us_PC_v1_00.tsv.gz
2017-11-25 08:10:33   17634794
amazon_reviews_us_Personal_Care_Appliances_v1_00.tsv.gz
```

```
2017-11-25 08:11:02  515815253 amazon_reviews_us_Pet_Products_v1_00.tsv.gz
2017-11-25 08:22:26  642255314 amazon_reviews_us_Shoes_v1_00.tsv.gz
2017-11-25 08:39:15   94010685 amazon_reviews_us_Software_v1_00.tsv.gz
2017-11-27 10:36:58  872478735 amazon_reviews_us_Sports_v1_00.tsv.gz
2017-11-25 08:52:11  333782939 amazon_reviews_us_Tools_v1_00.tsv.gz
2017-11-25 09:06:08  838451398 amazon_reviews_us_Toys_v1_00.tsv.gz
2017-11-25 09:42:13 1512355451 amazon_reviews_us_Video_DVD_v1_00.tsv.gz
2017-11-25 10:50:22  475199894 amazon_reviews_us_Video_Games_v1_00.tsv.gz
2017-11-25 11:07:59  138929896 amazon_reviews_us_Video_v1_00.tsv.gz
2017-11-25 11:14:07  162973819 amazon_reviews_us_Watches_v1_00.tsv.gz
2017-11-26 15:24:07 1704713674 amazon_reviews_us_Wireless_v1_00.tsv.gz
2017-11-28 15:55:13       6162 index.txt
2017-11-27 11:08:16      17553 sample_fr.tsv
2017-11-27 11:08:17      15906 sample_us.tsv
```

Dataset Files in Parquet Format

```
PRE product_category=Apparel/
                           PRE product_category=Automotive/
                           PRE product_category=Baby/
                           PRE product_category=Beauty/
                           PRE product_category=Books/
                           PRE product_category=Camera/
                           PRE product_category=Digital_Ebook_Purchase/
                           PRE product_category=Digital_Music_Purchase/
                           PRE product_category=Digital_Software/
                           PRE product_category=Digital_Video_Download/
                           PRE product_category=Digital_Video_Games/
                           PRE product_category=Electronics/
                           PRE product_category=Furniture/
                           PRE product_category=Gift_Card/
                           PRE product_category=Grocery/
                           PRE product_category=Health_&_Personal_Care/
                           PRE product_category=Home/
                           PRE product_category=Home_Entertainment/
                           PRE product_category=Home_Improvement/
                           PRE product_category=Jewelry/
                           PRE product_category=Kitchen/
                           PRE product_category=Lawn_and_Garden/
                           PRE product_category=Luggage/
                           PRE product_category=Major_Appliances/
                           PRE product_category=Mobile_Apps/
                           PRE product_category=Mobile_Electronics/
                           PRE product_category=Music/
                           PRE product_category=Musical_Instruments/
                           PRE product_category=Office_Products/
                           PRE product_category=Outdoors/
                           PRE product_category=PC/
```

```
PRE product_category=Personal_Care_Appliances/
PRE product_category=Pet_Products/
PRE product_category=Shoes/
PRE product_category=Software/
PRE product_category=Sports/
PRE product_category=Tools/
PRE product_category=Toys/
PRE product_category=Video/
PRE product_category=Video_DVD/
PRE product_category=Video_Games/
PRE product_category=Watches/
PRE product_category=Wireless/
```

Note that `PRE` stands for `prefix`. For now, you can think of prefixes as folders in S3.

> **NOTE**
>
> At any time, you can use EXPLAIN on your queries to make sure the partitions are being utilized. If your query patterns change over time, you may want to revisit updating the existing partitions - or even adding new partitions to match your business needs.

So which data format should you choose? The query-optimized Parquet columnar file format is definitely preferred when running analytics queries as it improves your query performance. On the other hand, many applications write out data in simple comma-separated (CSV) or tab-separated (TSV) files, i.e. application log files. So let's actually assume we don't have the Parquet files ready-to-use as this allows us to show you how you can easily get there from CSV or TSV files.

In a first step, let's copy the TSV data from Amazon's public S3 bucket into a privately hosted S3 bucket, to simulate that process, as shown in figure 3-5.

Figure 2-5. Copy our dataset from the public S3 bucket to a privately hosted S3 bucket.

You can use the AWS CLI tool again to perform the following steps.

1. Create a new private S3 bucket:

```
aws s3 mb s3://data-science-on-aws
```

1. Copy the content of the public S3 bucket to our newly created private S3 bucket like follows (only include the files starting with `amazon_reviews_us_`, i.e. skipping any index, multilingual and sample data files in that directory).

```
aws s3 cp --recursive s3://amazon-reviews-pds/tsv/ s3://data-science-on-
aws/amazon-reviews-pds/tsv/ --exclude "*" --include "amazon_reviews_us_*"
```

We are now ready to use Amazon Athena to register and query the data, and transform the TSV files into Parquet.

Query the S3 Data Lake with Amazon Athena

Amazon Athena is an interactive query service that makes it easy to analyze data in Amazon S3 using standard SQL. Athena is serverless, so you don't need to manage any infrastructure, and you only pay for the queries you run.

With Athena, you can query data wherever it is stored (S3 in our case) without needing to move the data to a relational database. Athena separates

compute from storage -- lowering the overall Time-To-Insight (TTI) from when the data lands in S3 from your application to when the user seeks an insight for your business. When you register an Athena table with your S3 data, Athena stores the Table-to-S3 mapping in a data catalog. Athena is using the AWS Glue Data Catalog service for that purpose. You can think of the Glue Data Catalog as a persistent metadata store in your AWS account which other AWS services such as Athena and Redshift Spectrum can use to locate and query data.

See the AWS Glue Data Catalog in Figure 3-6.

Figure 2-6. AWS Glue Data Catalog.

Athena queries run in parallel over a dynamic, serverless cluster which makes Athena extremely fast -- even on large datasets. Athena will automatically scale the cluster depending on the query and dataset -- freeing the user from worrying about these details.

In addition Athena supports the Parquet columnar file format with 10's of millions of partitions (ie. by `product_category`, `year`, or `marketplace`) to improve the performance of your queries. For example, if you plan to run

frequent queries that group the results by `product_category`, then you should create a partition in Athena for `product_category`. Upon creation, Athena will update the Glue Catalog accordingly so that future queries will inherit the performance benefits of this new partition.

Athena is based on Presto, an open source, distributed SQL query engine designed for fast, ad-hoc data analytics on large datasets. Similar to Apache Spark, Presto uses high RAM clusters to perform its queries. However, Presto does not require a large amount of disk as it is designed for ad-hoc queries (vs. automated, repeatable queries) and therefore does not perform the checkpointing required for fault-tolerance.

> **NOTE**
>
> You can find more information on Presto here: https://aws.amazon.com/big-data/what-is-presto/

While Athena is very fast for ad-hoc queries, it is perhaps not the best option for mission-critical queries that need to recover from partial failures. This is the classic trade-off between performance and fault-tolerance per the CAP Theorem as shown in Figure 3-7 below.

CAP Theorem

Choose Only 2 of 3 Below

Figure 2-7. CAP Theorem.

If you truly need fault-tolerant queries, you will need something like Apache Spark with Elastic Map Reduce (EMR) or SageMaker Processing Jobs. Apache Spark, while known for its efficient cluster-memory utilization, actually uses a combination of checkpointing and disk-spilling to recover from cluster-node failures. The drawback is performance: Apache Spark is slower than Athena for many ad-hoc queries.

For longer-running Athena jobs, you can listen for query-completion events using CloudWatch Events. When the query completes, all listeners are notified with the event details including query success status, total execution time, and total bytes scanned.

With a functionality called Athena Federated Query, you can also run SQL queries across data stored in relational databases (such as Amazon RDS and Amazon Aurora), non-relational databases (such as Amazon DynamoDB), object storage (Amazon S3), and custom data sources. This gives you a unified analytics view across data stored in your data warehouse, data lake and operational databases without the need to actually move the data.

You can access Athena via the AWS Management Console, an API, or an ODBC or JDBC driver for programmatic access. Let's have a look at how to use Amazon Athena via the AWS Management Console.

Access Athena from the AWS Console

In order for us to use Amazon Athena, we first need to setup the service as follows:

1. Click on Amazon Athena in the AWS Management Console.

   If this is your first time visiting the Athena console, you'll see a "Getting Started" page as shown in Figure 3-8. Choose "Get Started" to open the Query Editor.

aws Services ▾ Resource Groups ▾ N. Virginia ▾ Support ▾

Amazon Athena

Amazon Athena is a fast, cost-effective, interactive query service that makes it easy to analyze petabytes of data in S3 with no data warehouses or clusters to manage.

Get Started

Getting started guide

Select a data set

Identify where your data is located in S3. Athena allows you to query data in CSV, TSV, JSON, Parquet, and ORC formats.

Create a table

Use the Create Table Wizard or write your own DDL (Data Definition Language) statements using Hive. Learn more

Query data

Run queries on your data. Amazon Athena supports ANSI SQL queries. Learn more

Figure 2-8. Amazon Athena Console > click the "Getting Started" page.

You might also get asked to set up a query result location in Amazon S3 if you are using Amazon Athena for the first time, click on the link shown in Figure 3-9 and define a S3 bucket, e.g. data-science-on-aws/athena/staging

Services
Resource Groups
N. Virginia
Support
Athena
Query Editor
Saved Queries
History
AWS Glue Data Catalog
Workgroup : primary
Settings
Tutorial
Help
What's new
Database
sampledb
Filter tables and views...
Tables (1)
Create table
elb_logs
Views (0)
Create view
You have not created any views. To create a view, run a query and click "Create view from query"
Before you run your first query, you need to set up a query result location in Amazon S3. Learn more
New query 1
Run query
Save as
Create
Format query
Clear
Use Ctrl + Enter to run query, Ctrl + Space to autocomplete

Figure 2-9. Amazon Athena Console > define a S3 bucket to store results.

1. Create a Database.

   In the Athena Query Editor, you see a query pane with an example query. Start typing your query anywhere in the query pane as shown in Figure 3-10.

Services
Resource Groups
N. Virginia
Support
Athena
Query Editor
Saved Queries
History
AWS Glue Data Catalog
Workgroup : primary
Settings
Tutorial
Help
What's new
10+
Database
sampledb
Filter tables and views...
Tables (1)
Create table
elb_logs
Views (0)
Create view
You have not created any views. To create a view, run a query and click "Create view from query"
Before you run your first query, you need to set up a query result location in Amazon S3. Learn more
New query 1
1
Run query
Save as
Create
Format query
Clear
Use Ctrl + Enter to run query, Ctrl + Space to autocomplete

Figure 2-10. Amazon Athena Query Editor.

To create our database, enter the following CREATE DATABASE statement, and then choose “Run Query”:

```
CREATE DATABASE dsoaws;
```

Confirm that the catalog display refreshes and dsoaws appears in the DATABASE list in the Catalog dashboard on the left side as shown in Figure 3-11.

Services ▾ Resource Groups ▾ N. Virginia ▾ Support ▾

Athena Query Editor Saved Queries History AWS Glue Data Catalog Workgroup : primary Settings Tutorial Help What's new 10+

Database

dsoaws

Filter tables and views...

▾ Tables (0) Create table

The selected database has no tables.

▾ Views (0) Create view

You have not created any views. To create a view, run a query and click "Create view from query"

New query 1

```
1 CREATE DATABASE dsoaws;
```

Run query Save as Create (Run time: 0.36 seconds, Data scanned: 0 KB) Format query Clear

Use Ctrl + Enter to run query, Ctrl + Space to autocomplete

Results

Query successful.

Figure 2-11. Amazon Athena Console. Database created successfully.

When we run `CREATE DATABASE` and `CREATE TABLE` queries in Athena with the AWS Glue Data Catalog as our source, we automatically see the database and table metadata entries being created in the AWS Glue Data Catalog as shown in Figure 3-12.

Figure 2-12. Amazon Glue Console. Database entry created successfully.

Register Your S3 Data as an Athena Table

Now that we have a database, we're ready to create a table that's based on the Amazon Customer Reviews Dataset. We define the columns that map to the data, specify how the data is delimited, and provide the location in Amazon S3 for the file(s). We're basically defining a "schema-on-read" here.

In the Athena Console, make sure that dsoaws is selected for DATABASE and then choose "New Query" and run the following SQL statement:

```
CREATE EXTERNAL TABLE IF NOT EXISTS dsoaws.amazon_reviews_tsv(
         marketplace string,
         customer_id string,
         review_id string,
         product_id string,
         product_parent string,
         product_title string,
         product_category string,
         star_rating int,
         helpful_votes int,
         total_votes int,
         vine string,
         verified_purchase string,
         review_headline string,
         review_body string,
         review_date string
) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' LINES TERMINATED BY '\n'
LOCATION 's3://data-science-on-aws/amazon-reviews-pds/tsv' TBLPROPERTIES
('compressionType'='gzip', 'skip.header.line.count'='1')
```

You should see the newly created table amazon_reviews_tsv appear on the left under "Tables" as shown in Figure 3-13.

Services ▾ Resource Groups ▾ N. Virginia ▾ Support ▾

Athena Query Editor Saved Queries History Data sources Workgroup : primary Settings Tutorial Help What's new 10+

Data source Connect data source

AwsDataCatalog

Database

dsoaws

Filter tables and views...

▾ Tables (1) Create table

amazon_reviews_tsv

▾ Views (0) Create view

You have not created any views. To create a view, run a query and click "Create view from query"

New query 1

```
CREATE EXTERNAL TABLE IF NOT EXISTS dsoaws.amazon_reviews_tsv(
        marketplace string,
        customer_id string,
        review_id string,
        product_id string,
        product_parent string,
        product_title string,
        product_category string,
        star_rating int,
        helpful_votes int,
        total_votes int,
        vine string,
        verified_purchase string,
        review_headline string,
        review_body string,
        review_date string
) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' LINES TERMINATED BY '\n' LOCATION 's3://data-science-on-aws/amazon-reviews-pds/tsv'
TBLPROPERTIES ('compressionType'='gzip', 'skip.header.line.count'='1')
```

Run query Save as Create (Run time: 0.47 seconds, Data scanned: 0 KB) Format query Clear

Use Ctrl + Enter to run query, Ctrl + Space to autocomplete

Results

Query successful.

Figure 2-13. Amazon Athena Console. TSV table created successfully.

You can run a sample query like this to check if everything works correctly:

```
                SELECT *
FROM dsoaws.amazon_reviews_parquet
WHERE product_category = 'Digital_Video_Download' LIMIT 10
```

You should see a result similar to what is shown in Figure 3-14.

aws Services ▾ Resource Groups ▾ N. Virginia ▾ Support ▾

Athena Query Editor Saved Queries History Data sources Workgroup : primary Settings Tutorial Help What's new 10+

Data source Connect data source

AwsDataCatalog

Database

dsoaws

Filter tables and views...

▾ Tables (1) Create table

amazon_reviews_tsv

▾ Views (0) Create view

You have not created any views. To create a view, run a query and click "Create view from query"

New query 1

```
SELECT *
FROM dsoaws.amazon_reviews_tsv
WHERE product_category = 'Digital_Video_Download' LIMIT 10
```

Run query Save as Create (Run time: 1 minute 26 seconds, Data scanned: 29.71 GB) Format query Clear

Use Ctrl + Enter to run query, Ctrl + Space to autocomplete

Results

	marketplace	customer_id	review_id	product_id	product_parent	product_title	product_category	star_rating	helpf
1	US	12190288	R3FU16928EP5TC	B00AYB1482	668895143	Enlightened: Season 1	Digital_Video_Download	5	0
2	US	30549954	R1IZHHS1MH3AQ4	B00KQD28OM	246219280	Vicious	Digital_Video_Download	5	0
3	US	52895410	R52R85WC6TIAH	B01489L5LQ	534732318	After Words	Digital_Video_Download	4	17
4	US	27072354	R7HOOYTVIB0DS	B008LOVIIK	239012694	Masterpiece: Inspector Lewis Season 5	Digital_Video_Download	5	0
5	US	26939022	R1XQ2N5CDOZGNX	B0094LZMT0	535858974	On The Waterfront	Digital_Video_Download	5	0
6	US	4772040	R1HCST57W334KN	B0112OSOQE	38517795	Rick and Morty Season 2	Digital_Video_Download	5	5
7	US	12910040	R32BUTYQS1ZJBQ	B000NPE55A	373323715	Africa Screams	Digital_Video_Download	4	1
8	US	38805573	RH4SXPL4L9QU	B00XWV4QXG	633842417	Entourage: Season 7	Digital_Video_Download	3	0
9	US	37100714	R37INWIQA5YW8N	B00X8UKOUK	666093513	Catastrophe - Season 1	Digital_Video_Download	2	0
10	US	41234409	R18GSVAAS3N8GX	B00OOKXTFU	801680808	The Worricker Trilogy Season 1	Digital_Video_Download	3	0

Figure 2-14. Amazon Athena showing query result.

> **NOTE**
>
> When using LIMIT, you can better-sample the rows by adding `TABLESAMPLE BERNOULLI(10)` after the FROM. Otherwise, you will always return the data in the same order that it was ingested into S3 which could be skewed towards a single `product_category`, for example. To reduce code clutter, we will just use LIMIT without TABLESAMPLE.

Create a Parquet-based Table in Athena

In a next step, we will show you how you can easily convert that data now into the Apache Parquet columnar file format to improve the query performance. Parquet is optimized for columnar-based queries such as counts, sums, averages, and other aggregation-based summary statistics that focus on the column values vs. row information.

> **NOTE**
>
> While we already have the data in Parquet format from the public dataset, we feel that creating a Parquet table is an important enough topic to demonstrate in this book. If you are doing any type of large scale data analytics, you should be using a columnar file format like Parquet. We discuss the benefits of Parquet in the performance section.

- Again, make sure that dsoaws is selected for DATABASE and then choose “New Query” and run the following “CREATE TABLE AS” (short CTAS) SQL statement:

```
CREATE TABLE IF NOT EXISTS dsoaws.amazon_reviews_parquet
WITH (format = 'PARQUET', external_location = 's3://data-science-on-aws/amazon-
reviews-pds/parquet', partitioned_by = ARRAY['product_category']) AS
SELECT marketplace,
        customer_id,
        review_id,
        product_id,
        product_parent,
        product_title,
```

```
        star_rating,
        helpful_votes,
        total_votes,
        vine,
        verified_purchase,
        review_headline,
        review_body,
        CAST(YEAR(DATE(review_date)) AS INTEGER) AS year,
        DATE(review_date) AS review_date,
        product_category
FROM dsoaws.amazon_reviews_tsv
```

As you can see from the query, we're also adding a new `year` column to our dataset by converting the `review_date` string to a date format, and then cast the year out of the date. Let's store the year value as an integer.

- You should now see the newly created table amazon_reviews_parquet appear as well on the left under "Tables" as shown in Figure 3-15.

Services ▾ Resource Groups ▾ N. Virginia ▾ Support ▾

Athena Query Editor Saved Queries History Data sources Workgroup : primary Settings Tutorial Help What's new 10+

Data source Connect data source

AwsDataCatalog

Database

dsoaws

Filter tables and views...

Tables (2) Create table

amazon_reviews_parquet (Partitioned)

amazon_reviews_tsv

Views (0) Create view

You have not created any views. To create a view, run a query and click "Create view from query"

New query 1

```
CREATE TABLE IF NOT EXISTS dsoaws.amazon_reviews_parquet
WITH (format = 'PARQUET', external_location = 's3://data-science-on-aws/amazon-reviews-pds/parquet', partitioned_by =
ARRAY['product_category']) AS
SELECT marketplace,
       customer_id,
       review_id,
       product_id,
       product_parent,
       product_title,
       star_rating,
       helpful_votes,
       total_votes,
       vine,
       verified_purchase,
       review_headline,
       review_body,
       CAST(YEAR(DATE(review_date)) AS INTEGER) AS year,
       DATE(review_date) AS review_date,
       product_category
FROM dsoaws.amazon_reviews_tsv
```

Run query Save as Create (Run time: 4 minutes 10 seconds, Data scanned: 30.17 GB) Format query Clear

Use Ctrl + Enter to run query, Ctrl + Space to autocomplete

Results

Query successful.

Figure 2-15. Amazon Athena. Parquet table created successfully.

- As a last step, we need to load the Parquet partitions. To do so, just issue the following SQL command:

MSCK REPAIR TABLE amazon_reviews_parquet;

> **NOTE**
>
> You can automate the `MSCK REPAIR TABLE` command to load the partitions after data ingest from any workflow manager (or using an AWS Lambda function that gets triggered by uploading data to the S3 bucket) by executing the following AWS CLI command `aws athena start-query-execution --query-string "MSCK REPAIR TABLE <database>.<table>" --result-configuration "OutputLocation=s3://<bucket>"`

- You can run our sample query again to check if everything works correctly:

```
SELECT *
FROM dsoaws.amazon_reviews_parquet
WHERE product_category = 'Digital_Video_Download' LIMIT 10
```

- We can also double check that both tables' metadata has been stored in the AWS Glue Data Catalog as shown in Figure 3-16.

Figure 2-16. AWS Glue console. Both tables' metadata information was created.

Done! In just a few steps we have set up Amazon Athena to connect to our Amazon Customer Reviews TSV files, and transformed them into Apache Parquet file format.

You might have noticed that our second sample query finished in a fraction of the time compared to the one before we ran on the TSV table. We accelerated our query response time by leveraging our data stored in Parquet and partitioned by `product_category`.

Now, for the next section, let's assume our business intelligence team wants to load the last 2 years of customer review data into our data warehouse to analyze customer behavior and buying trends.

Load Dataset into Redshift Data Warehouse

One of the fundamental differences between data lakes and data warehouses is that while you ingest and store huge amounts of raw, unprocessed data in your data lake, you normally only load some fraction of your recent data into your data warehouse. Depending on your business and analytics use case, this might be data from the past couple of months, a year, or maybe the past 2 years. Let's assume we want to have the past 2 years of our Amazon Customer Reviews Dataset in a data warehouse to analyze year-over-year customer behavior and review trends. We will use Amazon Redshift as our data warehouse for this.

Amazon Redshift is a fully managed data warehouse which allows you to run complex analytic queries against petabytes of structured data. Your queries are distributed and parallelized across multiple nodes. In contrast to relational databases which are optimized to store data in rows and mostly serve transactional applications, Redshift implements columnar data storage which is optimized for analytical applications where you are mostly interested in the data within the individual columns.

Amazon Redshift also includes Amazon Redshift Spectrum, which allows you to directly execute SQL queries from Redshift against exabytes of unstructured data in your Amazon S3 data lake without the need to physically move the data. Amazon Redshift Spectrum automatically scales the compute resources needed based on how much data is being received, so queries against Amazon S3 run fast, regardless of the size of your data. We will use Amazon Redshift Spectrum to access our data in S3, and then show you how you can combine data that is stored in Redshift with data that is still in S3.

This might sound similar to the approach we showed earlier with Amazon Athena, but note that in this case we show how your Business Intelligence

team can enrich their queries with data that is not stored in the data warehouse itself.

> **NOTE**
>
> Before you can run the queries shown next, you need to have an Amazon Redshift cluster set up in your AWS account. Please refer to Appendix C - Setup Amazon Redshift for detailed instructions.

Query the Data Lake and Data Warehouse with Redshift Spectrum

Once you have your Redshift cluster set up and configured, you can navigate to the AWS Console, Amazon Redshift, and click on Query Editor to put in SQL commands to execute as shown in Figure 3-17.

Services ▾ Resource Groups ▾ N. Virginia ▾ Support ▾

Redshift dashboard

Clusters

Query editor

Saved queries

Snapshots

Security

Workload management New

Reserved nodes

Advisor

Events

Connect client

What's new

Cluster dscaws

Database dscaws

Database user dscaws

Schema

public

Tables Showing 0 of 0 table(s)

Filter tables

No tables exist for the selected schema

New Query 1

1

Use Ctrl + Enter to run query, Ctrl + Space to autocomplete

Run query Save as Save Clear

Send feedback

Figure 2-17. AWS Console > Amazon Redshift > Query Editor.

We can leverage our previously created table in Amazon Athena with its metadata and schema information stored in the AWS Glue Data Catalog to access our data in S3 through Redshift Spectrum. All we need to do is create an external schema in Redshift, point it to our AWS Glue Data Catalog, and point Redshift to the database we've created.

In the Redshift Query Editor (or via any other ODBC/JDBC SQL client that you might prefer to use), execute the following command:

```
CREATE EXTERNAL SCHEMA IF NOT EXISTS athena FROM DATA CATALOG
DATABASE 'dsoaws'
IAM_ROLE '<YOUR-IAM-ROLE>'
CREATE EXTERNAL DATABASE IF NOT EXISTS
```

With this command, we are creating a new schema in Redshift called `athena` to highlight the data access we set up through our tables in Amazon Athena.

`FROM DATA CATALOG` indicates that the external database is defined in the Amazon Athena (AWS Glue) Data Catalog.

`DATABASE`refers to our previously created database in Athena.

`IAM_ROLE` needs to point to an Amazon Resource Name (ARN) for an IAM role that your cluster uses for authentication and authorization

NOTE

`IAM` is AWS's Identity and Access Management service which enables you to manage and control access to AWS services and resources in your account. With an IAM role you can specify the permissions a user or service is granted. In this example, the IAM role must have at a minimum permission to perform a LIST operation on the Amazon S3 bucket to be accessed and a GET operation on the Amazon S3 objects the bucket contains. If the external database is defined in an Amazon Athena data catalog, the IAM role must have permission to access Athena unless CATALOG_ROLE is specified. We will go into more details on IAM in a later section of this chapter when we discuss how you can secure your data.

If you now select `athena` in the Schema dropdown menu in the Redshift Query Editor view as shown in Figure 3-18, you can see our two tables `amazon_reviews_tsv` and `amazon_reviews_parquet` appear which we created with Amazon Athena.

Services
Resource Groups
N. Virginia
Support
Redshift dashboard
Clusters
Query editor
Saved queries
Snapshots
Security
Workload management
New
Reserved nodes
Advisor
Events
Connect client
What's new
Cluster dsoaws
Database dsoaws
Database user dsoaws
Schema
athena
Tables
Showing 2 of 2 table(s)
Filter tables
amazon_reviews_parquet
amazon_reviews_tsv
New Query 1
Use Ctrl + Enter to run query, Ctrl + Space to autocomplete
Run query
Save as
Save
Clear
Send feedback

Figure 2-18. External schema `athena` shows our two tables previously created in Athena.

Let's go ahead and run a sample query again to make sure everything works. In the Query Editor, run the following command:

```
SELECT
  product_category,
COUNT(star_rating) AS count_star_rating
FROM
  athena.amazon_reviews_tsv
GROUP BY
 product_category
ORDER BY
 count_star_rating DESC
```

You should see a similar response to the one shown in figure 3-19 below.

Services ▾ Resource Groups ▾ N. Virginia ▾ Support ▾

Redshift dashboard
Clusters
Query editor
Saved queries
Snapshots
Security
Workload management New
Reserved nodes
Advisor
Events
Connect client
What's new

Cluster dsoaws
Database dsoaws
Database user dsoaws
Schema
athena
Tables Showing 2 of 2 table(s)
Filter tables
amazon_reviews_parquet
amazon_reviews_tsv

New Query 1

```
SELECT
    product_category,
    COUNT(star_rating) AS count_star_rating
FROM
    athena.amazon_reviews_tsv
GROUP BY
    product_category
ORDER BY
    count_star_rating DESC
```

Use Ctrl + Enter to run query, Ctrl + Space to autocomplete

Run query | Save as | Save | Clear | Send feedback

Query results Query completed in 1.608 seconds

Download CSV Showing row(s) 1 - 43 View execution

	product_category	count_star_rating
1	Books	19531329
2	Digital_Ebook_Purchase	17622415
3	Wireless	9002021
4	PC	6908554
5	Home	6221559
6	Apparel	5906333
7	Health & Personal Care	5331449
8	Beauty	5115666
9	Video DVD	5069140
10	Mobile_Apps	5033376
11	Kitchen	4880466
12	Toys	4864249

Figure 2-19. Redshift query response reading from Amazon S3.

So with just one command, we now have access and can query our S3 data lake from Amazon Redshift without moving any data into our data warehouse. This is the power of Redshift Spectrum.

But now, let's actually copy some data from S3 into Amazon Redshift. Let's pull in customer reviews data from the year 2015.

First, create another Redshift schema called `redshift` with the following SQL command

CREATE SCHEMA IF NOT EXISTS redshift

Next, we will create a new table that represents our Customer Reviews data. We will also add a new column and add `year` to our table.

```
CREATE TABLE IF NOT EXISTS redshift.amazon_reviews_tsv_2015(
 marketplace varchar(2) ENCODE zstd,
  customer_id varchar(8) ENCODE zstd,
review_id varchar(14) ENCODE zstd,
 product_id varchar(10) ENCODE zstd DISTKEY,
 product_parent varchar(10) ENCODE zstd,
 product_title varchar(400) ENCODE zstd,
 product_category varchar(24) ENCODE raw,
 star_rating int ENCODE az64,
 helpful_votes int ENCODE zstd,
 total_votes int ENCODE zstd,
 vine varchar(1) ENCODE zstd,
 verified_purchase varchar(1) ENCODE zstd,
 review_headline varchar(128) ENCODE zstd,
 review_body varchar(65535) ENCODE zstd,
 review_date varchar(10) ENCODE bytedict,
 year int ENCODE az64) SORTKEY (product_category)
```

In the performance section, we will dive deep into the SORTKEY, DISTKEY, and ENCODE attributes above. For now, let's copy the data from S3 into our new Redshift table and run some sample queries.

For such bulk inserts, you can either use a `COPY` command, or an `INSERT INTO` command. In general, the `COPY` command is preferred, as it loads data

in parallel and more efficiently from Amazon S3, or other supported data sources.

If you are loading data or a subset of data from one table into another, you can use the `INSERT INTO` command with a `SELECT` clause for high-performance data insertion. As we're loading our data from the `athena.amazon_reviews_tsv` table, let's choose this option.

To actually copy the data from S3 into our new Redshift table, we will run the following `INSERT INTO` command:

```
INSERT
INTO
 redshift.amazon_reviews_tsv_2015
 SELECT
 marketplace,
 customer_id,
 review_id,
 product_id,
 product_parent,
 product_title,
 product_category,
 star_rating,
 helpful_votes,
 total_votes,
 vine,
 verified_purchase,
 review_headline,
 review_body,
 review_date,
 CAST(DATE_PART_YEAR(TO_DATE(review_date,
 'YYYY-MM-DD')) AS INTEGER) AS year
 FROM
 athena.amazon_reviews_tsv
 WHERE
 year = 2015
```

You might notice that we use a date conversion to parse the year out of our `review_date` column and store it in a separate `year` column which we then use to filter on all records from 2015. This is an example on how you can simplify ETL tasks, as we're putting our data transformation logic directly in a `SELECT` query and ingest the result into Redshift.

Another way to optimize our tables would be to create them as a sequence of time-series tables, especially when our data has a fixed retention period. Let's say we want to store data of the last 2 years (24 months) in our data warehouse, and update with new data once a month.

If you create one table per month, you can easily remove old data simply by running a `DROP TABLE` command on the corresponding table. This approach is much faster than running a large-scale DELETE process and also saves you from having to run a subsequent VACUUM process to reclaim space and re-sort the rows.

To combine query results across tables, we can use a UNION ALL view. Similarly, when we need to delete old data, we remove the dropped table from the UNION ALL view.

Here's an example of a `UNION ALL` view across two tables with customer reviews from year 2014 and 2015 (assuming we have one table with data for 2014, and one with data for 2015):

```
SELECT
 product_category,
 COUNT(star_rating) AS count_star_rating,
 year
FROM
 redshift.amazon_reviews_tsv_2014
GROUP BY
 redshift.amazon_reviews_tsv_2014.product_category,
 year
UNION
ALL SELECT
 product_category,
 COUNT(star_rating) AS count_star_rating,
 year
FROM
 redshift.amazon_reviews_tsv_2015
GROUP BY
 redshift.amazon_reviews_tsv_2015.product_category,
 year
ORDER BY
 count_star_rating DESC,
 year ASC
```

The query result should look similar to this:

product_category	count_star_rating	year
Digital_Ebook_Purchase	6615914	2014
Digital_Ebook_Purchase	4533519	2015
Books	3472631	2014
Wireless	2998518	2015
Wireless	2830482	2014
Books	2808751	2015
Apparel	2369754	2015
Home	2172297	2015
Apparel	2122455	2014
Home	1999452	2014

Now, let's actually run a query and combine data from Redshift with data which is still in S3. Let's take the data from the previous query for years 2015 and 2014, and query Athena/S3 for the years 2013-1995 by running this command:

```
SELECT
 year,
 product_category,
 COUNT(star_rating) AS count_star_rating
FROM
 redshift.amazon_reviews_tsv_2015
GROUP BY
 redshift.amazon_reviews_tsv_2015.product_category,
 year
UNION
ALL SELECT
 year,
 product_category,
 COUNT(star_rating) AS count_star_rating
FROM
 redshift.amazon_reviews_tsv_2014
GROUP BY
 redshift.amazon_reviews_tsv_2014.product_category,
```

```
 year
UNION
ALL SELECT
 CAST(DATE_PART_YEAR(TO_DATE(review_date,
 'YYYY-MM-DD')) AS INTEGER) AS year,
 product_category,
 COUNT(star_rating) AS count_star_rating
FROM
 athena.amazon_reviews_tsv
WHERE
 year <= 2013
GROUP BY
 athena.amazon_reviews_tsv.product_category,
 year
ORDER BY
 product_category ASC,
 year DESC
```

The query result should look similar to this:

year	product_category	count_star_rating

2015	Apparel	4739508
2014	Apparel	4244910
2013	Apparel	854813
2012	Apparel	273694
2011	Apparel	109323
2010	Apparel	57332
2009	Apparel	42967
2008	Apparel	33761
2007	Apparel	25986
2006	Apparel	7293
2005	Apparel	3533
2004	Apparel	2357
2003	Apparel	2147
2002	Apparel	907
2001	Apparel	5
2000	Apparel	6
2015	Automotive	2609750
2014	Automotive	2350246

Export Redshift Data to S3 Data Lake as Parquet

Redshift Data Lake Export gives you the ability to unload the result of a Redshift query to your S3 data lake in the optimized Apache Parquet columnar file format. This enables you to share any data transformation and enrichment you have done in Redshift back into your S3 data lake in an open format.

You can specify one or more partition columns so that unloaded data is automatically partitioned into folders in your Amazon S3 bucket. For example, you can choose to unload our customer reviews data and partition it by `product_category`.

Simply run the following SQL command to unload our 2015 customer reviews data in Parquet file format into S3, partitioned by

`product_category`:

```
          UNLOAD (
           'SELECT marketplace, customer_id, review_id, product_id,
product_parent,
           product_title, product_category, star_rating, helpful_votes,
total_votes,
           vine, verified_purchase, review_headline, review_body, review_date,
year
           FROM redshift.amazon_reviews_tsv_2015')
          TO 's3://data-science-on-aws/amazon-reviews-pds/parquet-from-
redshift/2015/'
          IAM_ROLE '<IAM_ROLE>'
          PARQUET PARALLEL ON
          PARTITION BY (product_category)
```

You can use the AWS CLI tool again to list the S3 folder and see our unloaded data from 2015 in Parquet format:

aws s3 ls s3://data-science-on-aws/amazon-reviews-pds/parquet-from-redshift/2015/

This concludes the section on how to load data from our S3 data lake into our Redshift data warehouse, how to query data with Redshift, query data across both Redshift and S3, and how to export data from Redshift back to S3 in the optimized Parquer columnar file format.

You might ask yourself now, when should I use Athena, and when should I use Redshift? Let's discuss.

Choosing Athena vs. Redshift

Amazon Athena should be your preferred choice when running ad-hoc SQL queries on data that is stored in Amazon S3. It doesn't require you to set up or manage any infrastructure resources, and you don't need to move any data. It supports structured, unstructured, and semi-structured data. With Athena, you are defining a "schema on read" -- you basically just log in, create a table and you are good to go.

Amazon Redshift is targeted for modern data analytics on large, peta-byte scale, sets of structured data. Here, you need to have a predefined “schema on write”. Unlike serverless Athena, Redshift requires you to create a cluster (compute and storage resources), ingest the data and build tables before you can start to query, but caters to performance and scale. So for any highly-relational data with a transactional nature (data gets updated), workloads which involve complex joins, and latency requirements to be sub-second, Redshift is the right choice.

Athena and Redshift are optimized for read-heavy analytics workloads - and therefore not replacements for write-heavy, relational databases such as Amazon Relational Database System (RDS) and Aurora. At a high level, use Athena for exploratory analytics and operational debugging; use Redshift for business-critical reports and dashboards.

NOTE

Some ultra-write-heavy workloads may require a “non-relational” or NoSQL database such as Amazon DynamoDB or Managed Apache Cassandra. These databases offer reduced referential integrity for increased performance. The referential integrity is usually managed at the application layer - perhaps as a “reaper” background process that periodically fixes or identifies dangling references. NoSQL databases also allow flexible schemas for an ever-changing business domain. Similar to referential integrity, schema validation usually occurs at the application level for NoSQL databases. These are the trade-offs you sometimes need to make when designing a data solution at scale.

Secure Your Dataset

In today’s world, keeping data secure and safe is a top priority. AWS is designed to help you build secure, high-performing, resilient, and efficient infrastructure for your applications. AWS implements the shared responsibility model through which they provide a global secure infrastructure and foundational compute, storage, networking and database services, as well as a range of security services which you can use to secure anything you build and run on top of these services.

Looking at S3, AWS provides a number of security controls and guidelines you can leverage when defining your own security policies:

- Ensure that your Amazon S3 buckets use the correct policies and are not publicly accessible
- Implement least privilege access
- Use IAM roles for applications and AWS services that require Amazon S3 access
- Enable multi-factor authentication (MFA) Delete
- Consider encryption of data at rest
- Enforce encryption of data in transit
- Consider Amazon S3 Object Lock
- Enable versioning
- Consider Amazon S3 cross-region replication
- Consider VPC endpoints and S3 Access Points for Amazon S3 access

Let's briefly discuss how you can use AWS Identity Access Management (IAM) Roles, KMS encryption, and S3 Access Points in this context.

Authenticating and Authorizing with Identity and Access Management (IAM) Roles

AWS Identity and Access Management (IAM) is a service that helps you to manage access to AWS resources. IAM controls who is authenticated and authorized to use resources.

You can create individual IAM users for people accessing your AWS account. Each user will have unique security credentials. You can also assign IAM users to IAM groups with defined access permissions (i.e. for specific job functions) and the IAM users inherit those permissions.

A more preferred way to delegate access permissions is via IAM roles. In contrast to an IAM user which is uniquely associated with one person, a role can be assumed by anyone who needs it, and provides you with only temporary security credentials for the duration of the role session. AWS Service Roles control which actions a service can perform on your behalf.

Access permissions are defined using IAM policies. It's a standard security best practice to only grant least privilege, in other words -- only grant the permissions required to perform a task.

By default, all Amazon S3 resources are private: only the resource owner, an AWS account that created it, can access the resource. The resource owner can optionally grant access permissions to others by writing an access policy.

Here's a sample policy that shows how you can grant a user additional access (add, update, and delete objects) to our sample S3 bucket called `data-science-on-aws` by attaching this policy to the corresponding IAM role:

```
{
 "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:::data-science-on-aws"
 },
 {
 "Effect":"Allow",
 "Action":[
```

```
            "s3:PutObject",
            "s3:PutObjectAcl",
            "s3:GetObject",
            "s3:GetObjectAcl",
            "s3:DeleteObject"
            ],
            "Resource":"arn:aws:s3:::data-science-on-aws/*"
            }
            ]
          }
```

In addition to users, you also need to manage the AWS Service Role privileges. Please see Appendix C with more details on which privileges Amazon Athena and Amazon Redshift require, and how you can customize them.

Storing Keys with Key Management Service

AWS Key Management Service (KMS) is a managed service that enables you to easily create and control the keys used for cryptographic operations. There are two ways to use AWS KMS with Amazon S3 to implement data-at-rest encryption. You can use server-side encryption to protect your data with a master key or you can use an AWS KMS customer master key (CMK) with the Amazon S3 Encryption Client to protect your data on the client side.

If you select Server-Side Encryption, you can choose between the following options:

SSE-S3

Requires that Amazon S3 manage the data and master encryption keys

SSE-C

Requires that you manage the encryption key.

SSE-KMS

Requires that AWS manage the data key but you manage the customer master key (CMK) in AWS KMS.

To require server-side encryption of all objects in a particular Amazon S3 bucket (enforcing data-at-rest encryption), use a bucket policy. For example, the following bucket policy denies upload object (s3:PutObject) permission to everyone if the request does not include the `x-amz-server-side-encryption` header requesting server-side encryption with SSE-KMS:

```
{
 "Version":"2012-10-17",
 "Id":"PutObjPolicy"
 "Statement":[{
 "Sid":"DenyUnEncryptedObjectUploads",
 "Effect":"Deny",
 "Principal":"*",
 "Action":"s3:PutObject",
 "Resource":"arn:aws:s3:::data-science-on-aws/*",
 "Condition":{
 "StringNotEquals":{
 "s3:x-amz-server-side-encryption":"aws:kms"
 }
 }
 }
 ]
}
```

When you upload an object, you can specify the AWS KMS CMK using the `x-amz-server-side-encryption-aws-kms-key-id header`. If the header is not present in the request, Amazon S3 assumes the AWS managed CMK.

Figure 3-20 and 3-21 show how to enable data-at-rest encryption with default encryption at bucket creation time via the AWS Console, and the two default options of server-side encryption SSE-S3 and SSE-KMS.

Create bucket

X

1 Name and region | 2 Configure options | 3 Set permissions | 4 Review

Object-level logging

☐ Record object-level API activity using AWS CloudTrail for an additional cost. See CloudTrail pricing or learn more

Default encryption

☑ Automatically encrypt objects when they are stored in S3. Learn more

○ AES-256
Use Server-Side Encryption with Amazon S3-Managed Keys (SSE-S3)

○ AWS-KMS
Use Server-Side Encryption with AWS KMS-Managed Keys (SSE-KMS)

▾ Advanced settings

Object lock

☐ Permanently allow objects in this bucket to be locked. Learn more

Object lock requires bucket versioning to be enabled.

Management

CloudWatch request metrics

☐ Monitor requests in your bucket for an additional cost. See CloudWatch pricing or learn more

Previous | Next

Figure 2-20. AWS Console > S3 > Create Bucket > Default Encryption > Use SSE-S3.

Create bucket

X

Name and region | 2 Configure options | 3 Set permissions | 4 Review

Object-level logging

Record object-level API activity using AWS CloudTrail for an additional cost. See CloudTrail pricing or learn more

Default encryption

Automatically encrypt objects when they are stored in S3. Learn more

AES-256

Use Server-Side Encryption with Amazon S3-Managed Keys (SSE-S3)

AWS-KMS

Use Server-Side Encryption with AWS KMS-Managed Keys (SSE-KMS)

aws/s3

Type to search

aws/s3

Custom KMS ARN

Advanc

Object lock

Permanen

Object lock

Management

CloudWatch request metrics

Previous | Next

Figure 2-21. AWS Console > S3 > Create Bucket > Default Encryption > Use SSE-KMS

Securing Buckets with S3 Access Points

Amazon S3 Access Points simplify access control for large, shared buckets such as data lakes. Up until recently, you accessed your S3 buckets through a unique bucket host name, and defined access control with a combination of IAM policies and a single bucket policy. You can imagine that for shared data sets and a growing number of users, teams, and applications which needed access this could end up fast in a complex environment for you to maintain.

Amazon S3 Access Points now simplify managing data access by providing you with a customized path into a bucket, each with a unique hostname and IAM access policy that enforces the specific permissions and network controls for any request made through the access point. This is particularly useful if you need to manage access to shared data sets.

You can also require that all access points be restricted to a Virtual Private Cloud (VPC), providing an extra level of security by basically firewalling your data to within your private networks.

Let's assume we have our sample S3 bucket called `data-science-on-aws` with prefixes (subfolders) called `feature-store` and `data-warehouse`. Our data science team needs read/write access to the feature store data, and our Business Intelligence team needs read access to the data-warehouse data stored in that bucket.

Figure 3-22 shows how that scenario would look without the use of S3 Access Points.

Figure 2-22. Accessing objects in Amazon S3 previously.

A single S3 bucket policy would have maybe looked like this:

```
"Sid":”PrefixBasedAccessDataScience",
"Effect":"Allow",
"Principal":{"AWS":”arn:aws:iam::123456789012:group/ds},
"Action":["s3:GetObject","s3:PutObject"],
"Resource":"arn:aws:s3:::data-science-on-aws/feature-store/*"
...
"Sid":”TagBasedAccessBusinessIntelligence",
"Effect":"Allow",
"Principal":{"AWS":”arn:aws:iam::123456789012:group/bi},
"Action":["s3:GetObject"],
"Resource":"arn:aws:s3:::data-science-on-aws/data-warehouse/*”
...
```

Now let’s see how we can simplify this with the use of S3 Access Points. The following sample command shows how to create Access Points called `ap1-ds` and `ap2-bi` via the AWS CLI command on our sample bucket called `data-science-on-aws`:

aws s3control create-access-point --name ap1-ds --account-id 123456789012 --bucket data-science-on-aws

aws s3control create-access-point --name ap2-bi --account-id 123456789012 --bucket data-science-on-aws

In an access point policy we then grant the IAM group for our Data Scientist team ('ds') in account 123456789012 permissions to GET and PUT objects with the prefix `feature-store/` through access point `ap1-ds`, and the IAM group for our Business Intelligence team ('bi') permissions to GET objects with the prefix `data-warehouse/` through access point `ap2-bi`:

```
{
 "Version":"2012-10-17",
 "Statement": [
 {
 "Effect": "Allow",
 "Principal": {
 "AWS": "arn:aws:iam::123456789012:group/ds"
 },
 "Action": ["s3:GetObject", "s3:PutObject"],
 "Resource": "arn:aws:s3:us-east-1:123456789012:accesspoint/ap1-
ds/object/feature-store/*"
 }]
}
{
 "Version":"2012-10-17",
 "Statement": [
 {
 "Effect": "Allow",
 "Principal": {
 "AWS": "arn:aws:iam::123456789012:group/bi"
 },
 "Action": ["s3:GetObject"],
 "Resource": "arn:aws:s3:us-east-1:123456789012:accesspoint/ap2-
bi/object/data-warehouse/*"
 }]
}
```

Figure 3-23 shows you can now manage access to your S3 objects with S3 Access Points.

Figure 2-23. Accessing objects in Amazon S3 with Access Points.

An AWS CLI request to an object in that bucket through the S3 Access Point would then look like this (assuming we are in the us-east-1 region and have the access permissions):

```
aws s3api get-object --key sample_us.tsv --bucket arn:aws:s3:us-east-1:123456789012:accesspoint/ap1-ds feature-store/raw/sample_us.tsv
```

You can also access the objects in an Amazon S3 bucket with an access point using the AWS Management Console, AWS SDKs, or the S3 REST APIs. For an application or user to be able to access objects through an access point, both the access point and the underlying bucket must permit the request.

Increase Performance and Reduce Cost

In this section, we want to provide some tips & tricks to increase performance and reduce cost during data ingestion including file formats, partitions, compression, and sort/distribution keys.

We will also demonstrate how to use Amazon S3 Intelligent Tiering to lower your storage bill.

Parquet Partitions and Compression

Athena supports the Parquet columnar format for large-scale analytics workloads. Parquet enables the following performance optimizations for your queries:

Partitions and Pushdowns

> Partitions are physical groupings of data on disk to match your query patterns (ie. SELECT * FROM reviews WHERE product_category='Books'). Modern query engines like Athena, Redshift, and Apache Spark will "pushdown" the WHERE into the physical storage system to allow the disk controller to seek once and read all relevant data in 1 scan without randomly skipping to different areas of the disk. This improves query performance even with Solid State Device (SSD) disks which have a lower seek time than traditional, media-based disk.

Dictionary encoding/compression

> When a small number of categorical values are stored together on disk (ie. `product_category` which has 43 total values in our dataset), the

values can be compressed into a small number of bits to represent each value (ie. `Books`, `Lawn_and_Garden`, `Software`, etc.) vs. storing the entire string.

Type compression

When values of a similar type (ie. String, Date, Integer) are stored together on disk, the values can be compressed together: (String, String), (Date, Date), (Integer, Integer). This compression is more efficient than if the values were stored separately on disk in a row-wise manner: (String, Date, Integer), (String, Date, Integer)

Vectorized aggregations

Because column values are stored together on disk, the disk controller needs to only perform one disk seek to find the beginning of the data. From that point, it will scan the data to perform the aggregation. Additionally, modern chips / processors offer high performance vectorization instructions to perform calculations on large amounts of data vs. flushing data in and out of the various data caches (L1, L2) or main memory.

See an example of row vs. columnar storage in figure 3-24.

Row Storage

product_category	review_date	review_length
Books	2015-04-20	511
Digital_Software	2014-09-13	278
Lawn_and_Garden	2011-07-04	401
Books	2010-04-15	726
Books	1999-10-05	140

Partitions and Pushdowns Improve Seek Time for WHERE & GROUP BY

Columnar Formats Improve Seek and Scan Time For SUM, AVG, STDDEV

Columnar Storage

product_category
Books
Books
Books
Digital_Software
Lawn_and_Garden
review_date
2015-04-20
2010-04-15
1999-10-05
2014-09-13
2011-07-04
review_length
511
726
140
278
401

Dictionary Encoding

Type Compression

Vectorized Aggregations

Figure 2-24. Row Storage (CSV/TSV File) vs. Columnar Storage (Parquet).

Redshift Table Design and Compression

Here is the CREATE TABLE script we used to create the Redshift tables:

```
CREATE TABLE IF NOT EXISTS redshift.amazon_reviews_tsv_2015(
 marketplace varchar(2) ENCODE zstd,
 customer_id varchar(8) ENCODE zstd,
 review_id varchar(14) ENCODE zstd,
 product_id varchar(10) ENCODE zstd DISTKEY,
 product_parent varchar(9) ENCODE zstd,
 product_title varchar(400) ENCODE zstd,
 product_category varchar(24) ENCODE raw,
 star_rating int ENCODE az64,
 helpful_votes int ENCODE zstd,
 total_votes int ENCODE zstd,
 vine varchar(1) ENCODE zstd,
 verified_purchase varchar(1) ENCODE zstd,
 review_headline varchar(128) ENCODE zstd,
 review_body varchar(65535) ENCODE zstd,
 review_date varchar(10) ENCODE bytedict,
 year int ENCODE az64) SORTKEY (product_category)
```

When you create a table, you can specify one or more columns as the SORT KEY. Amazon Redshift stores your data on disk in sorted order according to the SORT KEY. This means, you can optimize your table by choosing a SORT KEY that reflects your most frequently used query types. If you query a lot of recent data, you can specify a timestamp column as the sort key. If you frequently query based on range or equality filtering on one column, you should choose that column as the sort key. As we are going to run a lot of queries in the next chapter filtering on `product_category`, let's choose that one as our sort key.

> **NOTE**
>
> Redshift Advisor continuously recommends SORT KEYs for frequently-queries tables. Advisor will generate an ALTER TABLE command which you run without having to recreate the tables - and without impacting concurrent read and write queries. Note that Advisor does not provide recommendations if it doesn't see enough data (queries) or if the benefits are relatively small.

You can also define a distribution style for every table. When you load data into a table, Redshift distributes the rows of the table among your cluster nodes according to the table's distribution style. When you run a query, the query optimizer redistributes the rows to the cluster nodes as needed to perform any joins and aggregations. So our goal should be to optimize the rows distribution to minimize data movements. There are three distribution styles from which you can choose from:

KEY distribution

Distribute the rows according to the values in one column

ALL distribution

Distribute a copy of the entire table to every node

EVEN distribution

The rows are distributed across all nodes in a round-robin-fashion which is the default distribution style

For our table, we've chosen KEY distribution based on `product_id` as this column has a high cardinality, shows an even distribution and can be used to join with other tables.

> **NOTE**
>
> At any time, you can use EXPLAIN on your queries to make sure the DIST KEY and SORT KEY are being utilized. If your query patterns change over time, you may want to revisit changing these keys.

In addition, we are using compression for most columns to reduce the overall storage footprint and reduce our cost. Table 3-1 analyzes the compression used for each Redshift column in our schema.

Table 2-1. Compression used for each Redshift column in our schema.

Column	Data Type	Encoding	Explanation
marketplace	varchar(2)	zstd	Low cardinality, too small for higher compression overhead
customer_id	varchar(8)	zstd	High cardinality, relatively few repeat values
review_id	varchar(14)	zstd	Unique, unbounded cardinality, no repeat values
product_id	varchar(10)	zstd	Unbounded cardinality, relatively low number of repeat values
product_parent	varchar(10)	zstd	Unbounded cardinality, relatively low number of repeat words
product_title	varchar(400)	zstd	Unbounded cardinality, relatively low number of repeat words
product_category	varchar(24)	raw	Low cardinality, many repeat values, but first SORT key is raw
star_rating	int	az64	Low cardinality, many repeat values
helpful_votes	int	zstd	Relatively high cardinality
total_votes	int	zstd	Relatively high cardinality
vine	varchar(1)	zstd	Low cardinality, too small to incur higher compression overhead
verified_purchase	varchar(1)	zstd	Low cardinality, too small to incur higher compression overhead
review_headline	varchar(128)	zstd	Varying length text, high cardinality, low repeat words
review_body	varchar(65535)	zstd	Varying length text, High cardinality, low repeat words
review_date	varchar(10)	bytedict	Fixed length, relatively low cardinality, many repeat values
year	int	az64	Low cardinality, many repeat values

> **NOTE**
>
> While our CEO Andy Jassy maintains, “There is no compression algorithm for experience,”, there is a compression algorithm for data. Compression is a powerful tool for the ever-growing world of big data. All modern big data processing tools are compression-friendly including Amazon Athena, Redshift, Parquet, Pandas, and Apache Spark. Using compression on small values such as varchar(1) may not improve performance. However, due to native hardware support, there are almost no drawbacks to using compression.

zstd is a generic compression algorithm that works across many different data types and column sizes. `star_rating` and `year` fields are set to the default az64 encoding applied to most numeric and date fields. For most columns, we gain a quick win by using the default az64 encoding for integers and overriding the default lzo encoding in favor of the flexible zstd encoding for everything else including text.

We are using bytedict for `review_date` to perform dictionary encoding on the string-based dates (`YYYY-MM-DD`). While it seemingly has a large number of unique values, `review_date` actually contains a small number of unique values as there are only ~7,300 (365 days per year * 20 years) days in a 20 year span. This cardinality is low enough to capture all possible dates in just a few bits versus using a full `varchar(10)` for each date.

While `product_category` is a great candidate for bytedict dictionary encoding, `product_category` is our first (and only, in this case) SORTKEY. As a performance best practice, the first SORTKEY should not be compressed.

While `marketplace`, `product_category`, `vine`, and `verified_purchase` seem to be good candidates for bytedict, they are too small to benefit from the extra overhead. For now, we leave them as zstd.

If you have an existing Redshift table that you would like to optimize, you can run the ANALYZE COMPRESSION command in Redshift to generate a report of suggestion compression encodings as follows:

ANALYZE COMPRESSION redshift.customer_reviews_tsv_2015

The result will be a table like the following showing the % improvement in compression if you switch to another encoding:

Column	Encoding	Estimated Reduction (%)
marketplace	zstd	90.84
customer_id	zstd	38.88
review_id	zstd	36.56
product_id	zstd	44.15
product_parent	zstd	44.03
product_title	zstd	30.72
product_category	zstd	99.95
star_rating	az64	0
helpful_votes	zstd	47.58
total_votes	zstd	39.75
vine	zstd	85.03
verified_purchase	zstd	73.09
review_headline	zstd	30.55
review_body	zstd	32.19
review_date	bytedict	64.1
year	az64	0

NOTE

We performed the following analysis on a version of the CREATE TABLE that did not specify any ENCODE attributes. By default, Redshift will use az64 for numerics/dates and lzo for everything else (hence the 0% gain for the az64 suggestions below.)

Keep in mind that these are just suggestions and not always appropriate for your specific environment. You should try different encodings for your dataset and query the STV_BLOCKLIST table to compare the % reduction in physical number of blocks. For example, the analyzer recommends using zstd for our SORT KEY, `product_category`, but our experience shows

that query performance suffers when we compress the SORT KEY. We are using the extra disk space to improve our query performance.

S3 Intelligent Tiering

We introduced Amazon Simple Storage Service (Amazon S3) in this chapter as a scalable, durable storage service for building shared datasets such as data lakes in the cloud. And while we keep the S3 usage fairly simple in this book, the service actually offers you a variety of options to optimize your storage cost as your data grows.

Depending on your data's access frequency patterns and SLA needs, you can choose from various Amazon S3 storage classes. At the time of writing, these include S3 Standard for general-purpose storage of frequently accessed data (which is the default storage class); S3 Intelligent-Tiering for data with unknown or changing access patterns; S3 Standard-Infrequent Access (S3 Standard-IA) and S3 One Zone-Infrequent Access (S3 One Zone-IA) for long-lived, but less frequently accessed data; and Amazon S3 Glacier (S3 Glacier) and Amazon S3 Glacier Deep Archive (S3 Glacier Deep Archive) for long-term digital archival.

Figure 3-25 compares the Amazon S3 storage classes.

Figure 2-25. Amazon S3 storage classes.

NOTE

Amazon S3 Pricing might have changed since the time of writing. Please refer to https://aws.amazon.com/s3/pricing/ for up-to-date information.

But how do you know which objects to move? Imagine your S3 data lake has grown over time, and you might have billions of objects across several S3 buckets in S3 Standard storage class. Some of those objects are extremely important, while you haven't accessed others maybe in months or even years. This is where S3 Intelligent-Tiering comes into play.

Amazon S3 Intelligent-Tiering, automatically optimizes your storage cost for data with changing access patterns by moving objects between the frequent-access tier optimized for frequent use of data, and the lower-cost infrequent-access tier optimized for less-accessed data. Intelligent-Tiering monitors your access patterns and auto-tiers on a granular object level without performance impact, or any operational overhead.

Summary

In this chapter, we provided you with an overview on how you can load your data into Amazon S3, discussed the value of a S3 data lake, and showed you how you can leverage services like Amazon Athena to run ad-hoc SQL queries across the data in S3 without the need to physically move the data. We also introduced our dataset, the Amazon Customer Reviews Dataset, which we will be using through the rest of this book.

As different use cases require data in different formats, we elaborated on how you can use Athena to convert tab-separated data into query-optimized, columnar Parquet data.

As data in your S3 data lake often needs to be accessed not only by the Data Science / Machine Learning teams, but also from your Business Intelligence teams, we introduced you to Amazon Redshift, AWS' peta-byte scale cloud data warehouse. We transformed and enriched the data in Redshift, and showed how you can use Redshift Spectrum to combine queries across the data you keep in Redshift as well as data you keep in your S3 data lake.

We highlighted some important concepts you should leverage to secure your data. As part of this discussion, we touched on AWS Identity and Access Management Roles, implementing data-at-rest encryption with

AWS Key Management Service (KMS), and how the use of S3 Access Points can help manage access to data in shared S3 buckets (aka. your S3 data lake).

To conclude this chapter, we've discussed the various data compression formats and how they can improve your query results, and how you can reduce your AWS costs with Amazon S3 Intelligent Tiering.

In the next chapter we will explore the dataset in more detail. We will run queries to understand our data better, and visualize the results. We will also show you how you can detect data anomalies with Apache Spark and Amazon SageMaker Processing Jobs.

Chapter 3. Exploring the Dataset

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *support@pipeline.ai*.

In the previous chapter, we've talked about how you can ingest your data into the cloud, and run some ad-hoc queries with Athena as well as Redshift.

We also had a first peak into our dataset. As we've learned, the Amazon Customer Reviews dataset consists of more than 130+ million of those customer reviews of products across 43 different product categories on the Amazon.com website from 1995 until 2015.

The dataset contains the actual customer reviews text together with additional metadata, and is shared in both a tab separated value (TSV) text format, as well as in the optimized columnar binary format Parquet.

Here is the schema for the dataset again:

marketplace

2-letter country code (in this case all "US").

customer_id

Random identifier that can be used to aggregate reviews written by a single author.

review_id

A unique ID for the review.

product_id

The Amazon Standard Identification Number (ASIN). http://www.amazon.com/dp/<ASIN> links to the product's detail page.

product_parent

The parent of that ASIN. Multiple ASINs (color or format variations of the same product) can roll up into a single parent parent.

product_title

Title description of the product.

product_category

Broad product category that can be used to group reviews (in this case digital videos).

star_rating

The review's rating (1 to 5 stars).

helpful_votes

Number of helpful votes for the review.

total_votes

Number of total votes the review received.

vine

Was the review written as part of the Vine program?

verified_purchase

Was the review from a verified purchase?

review_headline

The title of the review itself.

review_body

The text of the review.

review_date

The date the review was written.

In the next sections of this chapter, we will explore and visualize the dataset in more depth, and introduce some tools and services that assist you in this task.

Overview

Query services like Athena, data warehouses like Redshift, Jupyter notebooks like SageMaker Notebooks, data processing engines like EMR, and dashboard and visualization tools like QuickSight, all address different use cases. Of course, you should choose the right tool for the right purpose. This chapter describes the breadth and depth of tools available within AWS, and maps these tools to common use cases seen across our customer base.

In the previous chapter, we already introduced you to Athena and Redshift.

- Amazon Athena offers ad-hoc, serverless SQL queries for data in S3 without needing to setup, scale, and manage any clusters.
- Amazon Redshift provides the fastest query performance for enterprise reporting and business intelligence workloads,

particularly those involving extremely complex SQL with multiple joins and subqueries across many data sources including relational databases and flat files.

In this chapter, we will also start to use Jupyter notebooks as our main workspace and development environment.

- Amazon SageMaker Notebooks are provided via fully managed compute instances which run the Jupyter Notebook App. With just a couple of clicks, you can provision such a notebook instance and start using Jupyter notebooks to run ad-hoc data analysis, preparation and processing, as well as model development as we will see here in a bit.

To interact with AWS resources from within a Python Jupyter notebook, we leverage the AWS Python SDK boto3, the Python DB client PyAthena to connect to Athena, and SQLAlchemy) as a Python SQL toolkit to connect to Redshift.

NOTE

AWS Data Wrangler is an open source toolkit from Amazon to facilitate data movement between Pandas and AWS services such as S3, Athena, Redshift, Glue, EMR, etc. Many AWS best practices are built into this library.

In addition to those services, we also have Amazon Elastic MapReduce (EMR) and Amazon QuickSight available.

- Amazon EMR supports flexible, highly-distributed, data-processing and analytics frameworks such as Hadoop and Spark when compared to on-premises deployments. Amazon EMR is flexible so you can run custom jobs with specific compute, memory, and storage parameters to optimize your analytics requirements.

- Amazon QuickSight is a fast, easy-to-use business intelligence service to build visualizations, perform ad-hoc analysis, and build dashboards from many data sources - and across many devices.

Let's explore our dataset in more depth now.

Visualize our Data Lake with Athena

At this point, we will shift over to use SageMaker Notebooks based on Jupyter to query and visualize our data in S3.

Amazon SageMaker is a fully managed service that provides developers and data scientists with a platform to build, train, and deploy their machine learning models. SageMaker aims at removing the heavy lifting from each step of the machine learning process to make it easier and faster to develop high quality models.

In this section, we will start working with Amazon SageMaker which provides managed Jupyter notebooks that you can start working with in minutes.

NOTE

For more detailed instructions on how to setup Amazon SageMaker Notebooks in your AWS account, please refer to Appendix C - Setup Amazon SageMaker.

Prepare SageMaker Notebook for Athena

For our exploratory data analysis in this Jupyter notebook, we will use Pandas, NumPy, Matplotlib and Seaborn, which are probably the most commonly used libraries for data analysis and data visualization in Python. Seaborn is built on top of matplotlib and adds support for pandas.

We will also introduce you to PyAthena, the Python DB Client for Amazon Athena, that enables us to run Athena queries right from our notebook. So let's get started by installing the mentioned libraries as follows:

```
%%bash
pip install -q pandas==0.23.0
pip install -q numpy==1.14.3
pip install -q matplotlib==3.0.3
pip install -q seaborn==0.8.1
pip install -q PyAthena==1.8.0
```

We also need to import a couple more libraries, one of them is boto3. Boto is the Amazon Web Services (AWS) SDK for Python. It enables Python developers to create, configure, and manage AWS services. Boto provides an easy to use, object-oriented API, as well as low-level access to AWS services.

We use boto3 to identify the AWS region we're currently working in, and use the pre-installed Amazon SageMaker Python SDK (sagemaker) to configure and interact with the service. Let's also define the database and table holding our Amazon Customer Reviews dataset information in Amazon Athena.

```
import boto3
import sagemaker
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
%config InlineBackend.figure_format='retina'
session = boto3.session.Session()
region_name = session.region_name
sagemaker_session = sagemaker.Session()
bucket = sagemaker_session.default_bucket()
```

```
database_name = 'dsoaws'
table_name = 'amazon_reviews_parquet'
```

With that, we are now ready to run our first SQL queries right from the notebook.

NOTE

If you are on a Mac, make sure you specify the `retina` setting above for retina-resolution images in Matplotlib. The difference is stunning!

Run a Sample Athena Query in the SageMaker Notebook

In the first example shown below, we will query our dataset to give us a list of the distinct product categories. PyAthena uses a connection cursor to connect to the data source, execute the query and store the query results in S3.

- connect returns a cursor configured with our region and a S3 bucket to save query results to, our s3_staging_bucket
- cursor.execute will execute our SQL query and store the results in CSV file format in the defined s3_staging_bucket

> **NOTE**
>
> There are different cursor implementations that you can use. While the standard cursor fetches the query result row by row, the PandasCursor will first save the CSV query results in the S3 staging directory, then read the CSV from S3 in parallel down to your Pandas DataFrame. This leads to better performance than fetching data with the standard cursor implementation. You can use the PandasCursor by specifying the cursor_class with the connect method or connection object.

```
# PyAthena imports
from pyathena import connect
from pyathena.pandas_cursor import PandasCursor
from pyathena.util import as_pandas
# Set S3 staging directory
# This is a temporary directory used for Athena queries
s3_staging_dir = 's3://{0}/athena/staging'.format(bucket)
# Execute query using connection cursor
cursor = connect(region_name=region_name, s3_staging_dir=s3_staging_dir).cursor()
cursor.execute("""
SELECT DISTINCT product_category from {0}.{1}
ORDER BY product_category
""".format(database_name, table_name))
```

We can now have a look at the results by reading the query results stored in a CSV file in our S3 staging bucket into a pandas dataframe as shown below:

```
# Load query results into Pandas DataFrame and show results
df_categories = as_pandas(cursor)
df_categories
```

The output should look similar to this:

product_category	product_category (continued)
Apparel	Luggage
Automotive	Major Appliances
Baby	Mobile_Apps
Beauty	Mobile_Electronics
Books	Music
Camera	Musical Instruments
Digital_Ebook_Purchase	Office Products
Digital_Music_Purchase	Outdoors
Digital_Software	PC
Digital_Video_Download	Personal_Care_Appliances
Digital_Video_Games	Pet Products
Electronics	Shoes
Furniture	Software
Gift Card	Sports
Grocery	Tools
Health & Personal Care	Toys
Home	Video
Home Entertainment	Video DVD
Home Improvement	Video Games
Jewelry	Watches
Kitchen	Wireless
Lawn and Garden	

> **NOTE**
>
> As common with Pandas DataFrames, PandasCursor handles the data on memory. Pay attention to the file size when reading data into the DataFrame. You can easily exceed your available memory when working with large datasets.

Dive Deep into the Dataset with Athena and SageMaker

Let's use Athena, SageMaker Notebooks, Matplotlib, and Seaborn to track down answers to the following questions over the entire dataset:

1. Which product categories are the highest rated by average rating?
2. Which product categories have the most reviews?
3. When did each product category become available in the Amazon catalog based on the date of the first review?
4. What is the breakdown of star ratings (1-5) per product category?
5. How have the star ratings changed over time? Is there a drop-off point for certain product categories throughout the year?
6. Which star ratings (1-5) are the most helpful?

> **NOTE**
>
> From this point, we will only show the Athena query and the results. The full source code to execute and render the results is available in the accompanying GitHub repo.

7. Which product categories are the highest rated by average rating?

```
SELECT product_category, AVG(star_rating) AS avg_star_rating
FROM dsoaws.amazon_reviews_parquet
GROUP BY product_category
ORDER BY avg_star_rating DESC
```

Let’s plot this in a horizontal bar chart using Seaborn and Matplotlib.

Average Rating by Product Category
Gift Card 4.73
_Music_Purchase 4.64
Music 4.44
Books 4.34
Grocery 4.31
_Ebook_Purchase 4.31
Video DVD 4.3
Tools 4.26
sical Instruments 4.25
Automotive 4.25
Shoes 4.24
Outdoors 4.24
Sports 4.23
Luggage 4.22
Toys 4.21
Kitchen 4.21
Video_Download 4.2
Video 4.19
Beauty 4.19
ne Improvement 4.18
Home 4.18
Baby 4.16
& Personal Care 4.16
Jewelry 4.14
Pet Products 4.14
Watches 4.14
Camera 4.13
Apparel 4.11
awn and Garden 4.09
PC 4.09
Furniture 4.08
Office Products 4.07
Video Games 4.06
Electronics 4.04
Mobile_Apps 3.98
Care_Appliances 3.98
e Entertainment 3.9
Wireless 3.89
al_Video_Games 3.85
obile_Electronics 3.76
Major Appliances 3.72
Software 3.57
Digital_Software 3.54
1-Star
2-Star
3-Star
4-Star
5-Star
Average Rating

Figure 3-1. Horizontal bar chart showing average rating per category

8. As we can see in Figure 4-1, Amazon Gift Cards are the highest rated product category with an average star rating of 4.73, followed by Music Purchase with an average of 4.64 and Music with an average of 4.44.

9. Which product categories have the most reviews?

```
SELECT product_category, COUNT(star_rating) AS count_star_rating FROM
dsoaws.amazon_reviews_parquet
GROUP BY product_category
ORDER BY count_star_rating DESC
```

10. Let's plot this again in a horizontal bar chart using Seaborn and Matplotlib.

Number of Ratings per Product Category
Product Category
Books
Digital_Ebook_Purchase
Wireless
PC
Home
Apparel
Health & Personal Care
Beauty
Video DVD
Mobile_Apps
Kitchen
Toys
Sports
Music
Shoes
Digital_Video_Download
Automotive
Electronics
Pet Products
Office Products
Home Improvement
Lawn and Garden
Grocery
Outdoors
Camera
Video Games
Jewelry
Baby
Tools
Digital_Music_Purchase
Watches
Musical Instruments
Furniture
Home Entertainment
Video
Luggage
Software
Gift Card
Digital_Video_Games
Mobile_Electronics
Digital_Software
Major Appliances
Personal_Care_Appliances
100K
1m
5m
10m
15m
20m
Number of Ratings

Figure 3-2. Horizontal bar chart showing number of reviews per category.

11. We can see in Figure 4-2 that the “Books” product category has the most reviews with close to 20 million. This makes sense as Amazon.com initially launched as the “Earth’s Biggest Bookstore” back in 1995 as shown in a restored picture of Amazon.com’s first website in Figure 3-3.

Welcome to Amazon.com Books!

One million titles, consistently low prices.

(If you explore just one thing, make it our personal notification service. We think it's very cool!)

Spotlight! -- August 16th

These are the books we love, offered at Amazon.com low prices. The spotlight moves **EVERY** day so please come often.

One Million Titles

Search Amazon.com's million title catalog by author, subject, title, keyword, and more... Or take a look at the books we recommend in over 20 categories... Check out our customer reviews and the award winners from the Hugo and Nebula to the Pulitzer and Nobel... and bestsellers are 30% off the publishers list...

Eyes & Editors, a personal notification service

Like to know when that book you want comes out in paperback or when your favorite author releases a new title? Eyes, our tireless, automated search agent, will send you mail. Meanwhile, our human editors are busy previewing galleys and reading advance reviews. They can let you know when especially wonderful works are published in particular genres or subject areas. Come in, meet Eyes, and have it all explained.

Your Account

Check the status of your orders or change the email address and password you have on file with us. Please note that you **do not** need an account to use the store. The first time you place an order, you will be given the opportunity to create an account.

Figure 3-3. Amazon.com Website, 1995 (Source: Version Museum)

12. The second most reviewed category is “Digital_Ebook_Purchase” representing Kindle book reviews. So we notice that book reviews - whether printed or as ebooks - still count the most reviews.

13. “Personal_Care_Appliances” has the least number of reviews. This could potentially be due to the fact that the product category got added more recently.

14. Let’s check this out by querying for the first review in each category which will give us a rough timeline of product category introductions.

15. When did each product category become available in the Amazon catalog?

16. The initial review date is a strong indicator of when each product category went live on Amazon.com.

```
SELECT product_category, MIN(DATE_FORMAT(DATE_ADD('day', review_date,
DATE_PARSE('1970-01-01','%Y-%m-%d')), '%Y-%m-%d')) AS first_review_date
FROM amazon_reviews_tsv
GROUP BY product_category
ORDER BY first_review_date
```

17. The result should look similar to this:

product_category	first_review_date
Books	1995-06-24
Music	1995-11-11
Video	1995-11-11
Video DVD	1996-07-08
Toys	1997-01-05
Sports	1997-10-09
Video Games	1997-11-06
Home	1998-05-29
Office Products	1998-07-15
Pet Products	1998-08-23
Software	1998-09-21
Home Entertainment	1998-10-15
Camera	1998-11-20
Wireless	1998-12-04
Health & Personal Care	1999-02-06
Outdoors	1999-03-24
Electronics	1999-06-09
PC	1999-07-01
Baby	1999-07-13
Digital_Ebook_Purchase	1999-08-28
Grocery	1999-09-05
Automotive	1999-10-24
Shoes	1999-11-08

Home Improvement	1999-11-08
Tools	1999-11-09
Lawn and Garden	1999-11-27
Musical Instruments	1999-12-13
Kitchen	2000-01-20
Furniture	2000-03-17
Digital_Music_Purchase	2000-06-28
Major Appliances	2000-08-26
Apparel	2000-09-06
Digital_Video_Download	2000-10-04
Personal_Care_Appliances	2000-10-29
Beauty	2000-10-31
Watches	2001-04-05
Jewelry	2001-11-10
Mobile_Electronics	2001-12-22
Luggage	2002-11-05
Gift Card	2004-10-14
Digital_Video_Games	2006-08-08
Digital_Software	2008-01-26
Mobile_Apps	2010-11-04

18. We can see that personal care appliances were indeed added somewhat later to the Amazon.com catalog, but that doesn't seem to be the only reason for the low number of reviews. Mobile apps appear to have been added around 2010.

19. Let's visualize the number of first reviews per category per year, shown in Figure 4-4

Figure 3-4. Number of first product category reviews per year.

20. We notice that a lot of our first product category reviews (13) happened in 1999. Whether this is really related to the introduction of those product categories around this time, or just a coincidence created by the available data in our dataset, we can't tell for sure.

21. What is the breakdown of ratings (1-5) per product category?

```
SELECT product_category,
        star_rating,
        COUNT(*) AS count_reviews
FROM dsoaws.amazon_reviews_parquet
GROUP BY  product_category, star_rating
ORDER BY  product_category ASC, star_rating DESC,
    count_reviews
```

22. The result should look similar to this (shortened):

product_category	star_rating	count_reviews
Apparel	5	3320566
Apparel	4	1147237
Apparel	3	623471
Apparel	2	369601
Apparel	1	445458
Automotive	5	2300757
Automotive	4	526665
Automotive	3	239886
Automotive	2	147767
Automotive	1	299867
...	...	...

23. With this information, we can also quickly group by star ratings and count the reviews for each rating (5, 4, 3, 2, 1):

```
        SELECT star_rating,
                  COUNT(*) AS count_reviews
FROM dsoaws.amazon_reviews_parquet
GROUP BY  star_rating
ORDER BY  star_rating DESC, count_reviews
```

24. The result should look similar to this:

star_rating	count_reviews
5	93200812
4	26223470
3	12133927
2	7304430
1	12099639

25. Approximately 62% of all reviews have a 5 star rating. We will come back to this relative imbalance of star ratings when we perform feature engineering to prepare for model training.

26. We can now calculate and visualize a stacked percentage horizontal bar plot, showing the proportion of each star rating per product category.

```
SELECT product_category,
         star_rating,
         COUNT(*) AS count_reviews
FROM {}.{}
GROUP BY  product_category, star_rating
ORDER BY  product_category ASC, star_rating DESC, count_reviews
```

27. In Figure 4-5 you see the visualization of the breakdown by star rating:

Distribution of Reviews Per Rating Per Category
5-Star Ratings
4-Star Ratings
3-Star Ratings
2-Star Ratings
1-Star Ratings
Gift Card
Digital_Music_Purchase
Music
Books
Grocery
Digital_Ebook_Purchase
Video DVD
Tools
Musical Instruments
Automotive
Shoes
Outdoors
Sports
Luggage
Toys
Kitchen
Digital_Video_Download
Video
Beauty
Home Improvement
Home
Baby
Health & Personal Care
Jewelry
Pet Products
Watches
Camera
Apparel
Lawn and Garden
PC
Furniture
Office Products
Video Games
Electronics
Mobile_Apps
Personal_Care_Appliances
Home Entertainment
Wireless
Digital_Video_Games
Mobile_Electronics
Major Appliances
Software
Digital_Software
0
20
40
60
80
100
% Breakdown of Star Ratings

Figure 3-5. Distribution of reviews per star rating (5, 4, 3, 2, 1) per product category.

28. We see that 5 and 4 star ratings make up the largest proportion within each product category. But let's see if we can spot differences in product satisfaction over time.

29. How have the star ratings changed over time? Is there a drop-off point for certain product categories throughout the year?

30. Let's first have a look at the average star rating across all product categories over the years:

```
SELECT year, ROUND(AVG(star_rating), 4) AS avg_rating
FROM dsoaws.amazon_reviews_parquet
GROUP BY year
ORDER BY year;
```

31. The result should look similar to this:

year	avg_rating
1995	4.6169
1996	4.6003
1997	4.4344
1998	4.3607
1999	4.2819
2000	4.2569
2001	4.1954
2002	4.1568
2003	4.1104
2004	4.057
2005	4.0614
2006	4.0962
2007	4.1673
2008	4.1294
2009	4.1107
2010	4.069
2011	4.0516
2012	4.1193
2013	4.1977
2014	4.2286
2015	4.2495

32. If we plot this as shown in Figure 4-6, we notice the general upwards trend with two lows in 2004 and 2011.

Average Star Rating Over Time (Across All Product Categories)

Figure 3-6. Average Star Rating across all product categories over the years

33. Let's have a look now at our top 5 product categories by number of ratings ('Books', 'Digital_Ebook_Purchase', 'Wireless', 'PC' and 'Home').

```
SELECT product_category,
         year,
         ROUND(AVG(star_rating),
        4) AS avg_rating_category
FROM dsoaws.amazon_reviews_parquet
WHERE product_category IN ('Books', 'Digital_Ebook_Purchase',
'Wireless', 'PC', 'Home')
GROUP BY  product_category, year
ORDER BY  year
```

34. The result should look similar to this (shortened):

product_category	year	avg_rating_category
Books	1995	4.6111
Books	1996	4.6024
Books	1997	4.4339
Home	1998	4.4
Wireless	1998	4.5
Books	1998	4.3045
Home	1999	4.1429
Digital_Ebook_Purchase	1999	5.0
PC	1999	3.7917
Wireless	1999	4.1471
...	...	...

35. If we plot this now as shown in Figure 3-7, we can see something interesting:

Average Star Rating Over Time Per Category (Top 5)
Books
Digital_Ebook_Purchase
Wireless
PC
Home
5
4
Average Star Rating
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
Years

Figure 3-7. Average Star Rating Over Time Per Category

36. While books have been relatively consistent in star ratings, and stick between 4.1 and 4.6, the other categories are more affected by customer satisfaction. The Digital_Ebooks_Purchase (Kindle books) seem to spike a lot with dropping as low as 3.5 in 2005, while at 5.0 in 2003. This would definitely require a closer look into our dataset, to decide whether this is due to limited reviews in that time or some sort of skewed data, or really reflecting the voice of our customers.

37. Which star ratings (1-5) are the most helpful?

```
SELECT star_rating,
         AVG(helpful_votes) AS avg_helpful_votes
FROM dsoaws.amazon_reviews_parquet
GROUP BY  star_rating
ORDER BY  star_rating DESC
```

38. The result should look similar to this:

star_rating	avg_helpful_votes
5	1.672697561905362
4	1.6786973653753678
3	2.048089542651773
2	2.5066350146417995
1	3.6846412525200134

39. We see that customers find negative reviews more helpful than positive reviews, also visualized in Figure 3-8 below.

Helpfulness Of Star Ratings
Average Helpful Votes
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
1
2
3
4
5
Star Rating

Figure 3-8. Helpfulness Of Star Ratings.

In the next section, we will show you how to query our data warehouse from Jupyter notebooks.

Query our Data Warehouse

At this point, we will use Redshift to query and visualize some use cases. Similar to the previous example with Athena, we first need to prepare our SageMaker Notebook environment again.

Prepare SageMaker Notebook for Redshift

We need to install SQLAlchemy, define our Redshift connection parameters, query the Redshift secret credentials from AWS Secret Manager, and obtain our Redshift Endpoint address. Finally, create the Redshift Query Engine.

```
# Install SQLAlchemy
!pip install -q SQLAlchemy==1.3.13
# Redshift Connection Parameters
redshift_schema = 'redshift'
redshift_cluster_identifier = 'dsoaws'
redshift_host = 'dsoaws'
redshift_database = 'dsoaws'
redshift_port = '5439'
redshift_table_2015 = 'amazon_reviews_tsv_2015'
redshift_table_2014 = 'amazon_reviews_tsv_2014'
# Load the Redshift Secrets from Secrets Manager
import json
import boto3
secretsmanager = boto3.client('secretsmanager')
secret = secretsmanager.get_secret_value(SecretId='dsoaws_redshift_login')
cred = json.loads(secret['SecretString'])
redshift_username = cred[0]['username']
redshift_pw = cred[1]['password']
redshift = boto3.client('redshift')
response =
redshift.describe_clusters(ClusterIdentifier=redshift_cluster_identifier)
redshift_endpoint_address = response['Clusters'][0]['Endpoint']['Address']
# Create Redshift Query Engine
from sqlalchemy import create_engine
engine = create_engine('postgresql://{}:{}@{}:{}/{}'.format(redshift_username,
redshift_pw, redshift_endpoint_address, redshift_port, redshift_database))
```

We are now ready to fire up our first Redshift query from the notebook.

Run a Sample Redshift Query from the SageMaker Notebook

In the example shown below, we will query our dataset to give us the number of unique customers per product category. We can use the Pandas `read_sql_query` function to run our SQLAlchemy query and store the query result in a Pandas dataframe.

```
df = pd.read_sql_query("""
SELECT product_category, COUNT(DISTINCT customer_id) as num_customers
FROM {}.{}
GROUP BY product_category
ORDER BY num_customers DESC
""".format(redshift_schema, redshift_table_2015), engine)
```

You can now simply print the first ten results with `df`:

```
df.head(10)
```

The output should look similar to this:

product_category	num_customers
Wireless	1979435
Digital_Ebook_Purchase	1857681
Books	1507711
Apparel	1424202
Home	1352314
PC	1283463
Health & Personal Care	1238075
Beauty	1110828
Shoes	1083406
Sports	1024591

We can see that the Wireless product category has the most unique customers providing reviews, followed by the Digital_Ebook_Purchase and

Books category. We are all set now to query Redshift for some deeper customer insights.

Dive Deep into the Dataset with Redshift and SageMaker

Now, let's query our data from 2015 in Redshift for deeper insight into our customer , and find answers to the following questions:

1. Which product categories had the most reviews in 2015?
2. Which products had the most helpful reviews in 2015? How long are those reviews?
3. What is the breakdown of star ratings (1-5) per product category in 2015?
4. How did the star ratings change during 2015? Is there a drop-off point for certain product categories throughout the year?
5. Which customers have written the most helpful reviews in 2015? And how many reviews have they written? Across how many categories? What is their average star rating?
6. Which customers are abusing the review system in 2015 by repeatedly reviewing the same product more than once? What was their average star rating for each product?

> **NOTE**
>
> Similar to the Athena examples before, we will only show the Redshift SQL query and the results. The full source code to execute and render the results is available in the accompanying GitHub repo.

Let's run the queries and find out the answers!

7. Which product categories had the most reviews in 2015?

```
SELECT
    year,
```

```
    product_category,
    COUNT(star_rating) AS count_star_rating
FROM
    redshift.amazon_reviews_tsv_2015
GROUP BY
    product_category,
    year
ORDER BY
    count_star_rating DESC,
    year DESC
```

8. The result should look similar to this (shortened):

year	product_category	count_star_rating
2015	Digital_Ebook_Purchase	4533519
2015	Wireless	2998518
2015	Books	2808751
2015	Apparel	2369754
2015	Home	2172297
2015	Health & Personal Care	1877971
2015	PC	1877971
2015	Beauty	1816302
2015	Digital_Video_Download	1593521
2015	Sports	1571181
...	...	...

9. We notice that books are still the most reviewed product categories, but it's actually the eBooks (Kindle books) now.
10. Let's visualize this in a horizontal barplot as shown in Figure 3-9.

Number of Ratings Per Product Category (2015)
Digital_Ebook_Purchase
Wireless
Books
Apparel
Home
Health & Personal Care
PC
Beauty
Digital_Video_Download
Sports
Kitchen
Shoes
Toys
Mobile_Apps
Automotive
Home Improvement
Lawn and Garden
Pet Products
Office Products
Video DVD
Electronics
Outdoors
Grocery
Jewelry
Tools
Baby
Music
Camera
Digital_Music_Purchase
Video Games
Watches
Musical Instruments
Furniture
Home Entertainment
Luggage
Software
Gift Card
Digital_Software
Digital_Video_Games
Major Appliances
Mobile_Electronics
Personal_Care_Appliances
Video
Product Category
1m
2m
3m
4m
5m
Number of Ratings

Figure 3-9. Number of Ratings Per Product Category (2015)

11. Which products had the most helpful reviews in 2015? How long are those reviews?

```
SELECT
    product_title,
    helpful_votes,
    LENGTH(review_body) AS review_body_length,
    SUBSTRING(review_body, 1, 100) AS review_body_substring
FROM
    redshift.amazon_reviews_tsv_2015
ORDER BY
    helpful_votes DESC LIMIT 10
```

12. The result should look similar to this:

product_title	helpful_votes	review_body_length	review_body_substring
Fitbit Charge HR Wireless Activity Wristband	16401	2824	Full disclosure, I ordered the Fitbit Charge HR only after I gave up on Jawbone fulfilling my preord
Kindle Paperwhite	10801	16338	Review updated September 17, 2015 As a background, I am a retired Information Systems pro
Kindle Paperwhite	8542	9411	[[VIDEOID:755c0182976ece27e407ad23676f3ae8]]If you’re reading reviews of the new 3rd generation Pape
Weslo Cadence G 5.9 Treadmill	6246	4295	I got the Weslo treadmill and absolutely dig it. I’m 6’2", 230 lbs. and like running outside (ab
Haribo Gummi Candy Gold-Bears	6201	4736	It was my last class of the semester, and the final exam was worth 30% of our grade. After a la
FlipBelt - World’s Best Running Belt & Fitness Workout Belt	6195	211	The size chart is wrong on the selection. I’ve attached a photo so you can see what you really need.
Amazon.com eGift Cards	5987	3498	I think I am just wasting time writing this, but this must have happened to someone else. Something
Melanie’s Marvelous Measles	5491	8028	If you enjoyed this book, check out these other fine titles from the same author: Abby’s
Tuft & Needle	5404	4993	tl;dr: Great mattress, after some

Mattress			hurdles that were deftly cleared by stellar customer service. The
Ring Wi-Fi Enabled Video Doorbell	5399	3984	First off, the Ring is very cool. I really like it and many people that come to my front door (sale

13. We see that the ‘Fitbit Charge HR Wireless Activity Wristband’ had the most helpful review in 2015 with a total of 16401 votes, and a decent, yet not too long review text of 2824 characters. It’s followed by two reviews for the ‘Kindle Paperwhite’ where people were writing longer reviews of 16338 and 9411 characters respectively.

14. What is the breakdown of star ratings (1-5) per product category in 2015?

```
        SELECT
                product_category,
                star_rating,
                COUNT(DISTINCT review_id) AS count_reviews
FROM
                redshift.amazon_reviews_tsv_2015
GROUP BY
                product_category,
                star_rating
ORDER BY
                product_category ASC,
                star_rating DESC,
                count_reviews
```

15. The result should look like this (shortened):

product_category	star_rating	count_reviews
Apparel	5	1350943
Apparel	4	445152
Apparel	3	245475
Apparel	2	143557
Apparel	1	184627
Automotive	5	889297
Automotive	4	177546
Automotive	3	81205
Automotive	2	49870
Automotive	1	106957
...	...	...

16. Let's visualize this again in a stacked percentage horizontal bar chart as shown in figure 4-10 below:

Distribution of Reviews Per Rating Per Category (2015)
5-Star Ratings
4-Star Ratings
3-Star Ratings
2-Star Ratings
1-Star Ratings
Gift Card
Digital_Music_Purchase
Music
Video DVD
Video
Books
Grocery
Digital_Ebook_Purchase
Tools
Kitchen
Luggage
Automotive
Sports
Musical Instruments
Toys
Outdoors
Personal_Care_Appliances
Home Improvement
Shoes
Home
Baby
Health & Personal Care
Beauty
Video Games
Office Products
Digital_Video_Download
Camera
Jewelry
Pet Products
Watches
Lawn and Garden
Furniture
PC
Apparel
Electronics
Mobile_Apps
Digital_Video_Games
Home Entertainment
Wireless
Major Appliances
Mobile_Electronics
Software
Digital_Software
0
20
40
60
80
100
% Breakdown of Star Ratings

Figure 3-10. Distribution of star ratings (1-5) per product category in 2015.

17. How did the star ratings change during 2015? Is there a drop-off point for certain product categories throughout the year?

```
SELECT
    CAST(DATE_PART('month', TO_DATE(review_date, 'YYYY-MM-DD')) AS
INTEGER) AS month,
    AVG(star_rating::FLOAT) AS avg_rating
FROM
    redshift.amazon_reviews_tsv_2015
GROUP BY
    month
ORDER BY
    month
```

18. The result should look similar to this:

month	avg_rating
1	4.277998926134835
2	4.267851231035101
3	4.261042822856084
4	4.247727865199895
5	4.239633709986397
6	4.235766635971452
7	4.230284081689972
8	4.231862792031927

19. We notice that we only have data until August 2015. We can also spot that the average rating is slowly declining, as you can easily spot in the visualization in figure 4-11 below. Assuming this data was current, the business should look into the root cause of this decline.

Average Star Rating Throughout 2015 (Across All Product Categories)

Figure 3-11. Average Star Rating Throughout 2015 (August) Across All Product Categories.

20. Now let's explore if this is due to a certain product category dropping drastically in customer satisfaction, or more of a trend across all categories.

```
SELECT
    product_category,
    CAST(DATE_PART('month', TO_DATE(review_date, 'YYYY-MM-DD')) AS
INTEGER) AS month,
    AVG(star_rating::FLOAT) AS avg_rating
FROM
    redshift.amazon_reviews_tsv_2015
GROUP BY
    product_category, month
ORDER BY
    product_category, month
```

21. The result should look similar to this (shortened)

product_category	month	avg_rating
Apparel	1	4.159321618698804
Apparel	2	4.123969612021801
Apparel	3	4.109944336469443
Apparel	4	4.094360325567125
Apparel	5	4.0894595692213125
Apparel	6	4.09617799917213
Apparel	7	4.097665115845663
Apparel	8	4.112790034578352
Automotive	1	4.325502388403887
Automotive	2	4.318120214368761
...	...	...

22. Let's visualize this to see the trends easier as shown in Figure 3-12 below:

Average Star Rating Over Time Per Category In 2015

Figure 3-12. Average Star Rating over time per product category in 2015.

23. We can indeed see that most categories follow the same average star rating over the months, with just 3 categories fluctuating more (but actually improving over the year) - it's Digital Software, Software, and Mobile Electronics.

24. Which customers have written the most helpful reviews in 2015? And how many reviews have they written? Across how many categories? What is their average star rating?

```
SELECT
    customer_id,
    AVG(helpful_votes) AS avg_helpful_votes,
    COUNT(*) AS review_count,
    COUNT(DISTINCT product_category) AS product_category_count,
    ROUND(AVG(star_rating::FLOAT), 1) AS avg_star_rating
FROM
    redshift.amazon_reviews_tsv_2015
GROUP BY
    customer_id
HAVING
    count(*) > 100
ORDER BY
    avg_helpful_votes DESC LIMIT 10;
```

25. The result should look similar to this:

customer_id	avg_helpful_votes	review_count	product_category_count	avg_star_rating

35360512	48	168	26	4.5
52403783	44	274	25	4.9
28259076	40	123	7	4.3
15365576	37	569	30	4.9
14177091	29	187	15	4.4
28021023	28	103	18	4.5
20956632	25	120	23	4.8
53017806	25	549	30	4.9
23942749	25	110	22	4.5
44834233	24	514	32	4.4

26. We can see that our customers which provided the most helpful votes (with more than 100 reviews provided) review across many different categories and are generally rating with a positive sentiment.

27. Which customers are abusing the review system in 2015 by repeatedly reviewing the same product more than once? What was their average star rating for each product?

```
SELECT
    customer_id,
    product_category,
    product_title,
    ROUND(AVG(star_rating::FLOAT), 4) AS avg_star_rating,
    COUNT(*) AS review_count
FROM
    redshift.amazon_reviews_tsv_2015
GROUP BY
    customer_id,
    product_category,
    product_title
HAVING
    COUNT(*) > 1
ORDER BY
    review_count DESC LIMIT 5
```

28. The result should look similar to this:

customer_id	product_category	product_title	avg_star_rating	review_count
28401 68	Camera	(Create a generic Title per Amazons guidelines)	5.0	45
9016330	Video Games	Disney INFINITY Disney Infinity: Marvel Super Heroes (2.0 Edition) Characters	5.0	23
10075230	Video Games	Skylanders Spyro's Adventure: Character Pack	5.0	23
50600878	Digital_Ebook_Purchase	The Art of War	2.35	20
10075230	Video Games	Activision Skylanders Giants Single Character Pack Core Series 2	4.85	20

29. Note where the avg_star_rating is not always a whole number. This means some customers rated the product differently over time.
30. It's good to know that customer 9016330 finds the Disney Infinity: Marvel Super Heroes Video Game to be 5 stars - even after playing it 23 times! While customer 50600878 has been fascinated by reading The Art of War 20 times, but is still struggling to find a positive sentiment there.

Create Dashboards with QuickSight

Amazon QuickSight is an easy-to-use business analytics service which you can leverage to quickly build powerful visualizations. QuickSight automatically discovers the data sources in your AWS account including MySQL, Salesforce, Amazon Redshift, Athena, S3, Aurora, RDS, and many more as shown in Figure 3-13.

QuickSight

N. Virg... Admin...

Data Sets

SPICE capacity for this region: 14.5MB of 1GB

Create a Data Set

FROM NEW DATA SOURCES

- Upload a file (.csv, .tsv, .clf, .elf, .xlsx, .json)
- Salesforce — Connect to Salesforce
- S3 Analytics
- S3
- Athena
- RDS
- Redshift — Auto-discovered
- Redshift — Manual connect
- MySQL
- PostgreSQL
- SQL Server
- Aurora
- MariaDB
- Presto
- Spark
- Teradata — Provided by Teradata
- Snowflake
- AWS IoT Analytics
- GitHub
- Twitter
- Jira
- ServiceNow
- Adobe Analytics

Figure 3-13. Amazon QuickSight > Supported Data Sources.

Here is the high-level QuickSight Workflow overview shown in Figure 3-14.

Figure 3-14. High-Level QuickSight Workflow.

Let's use Amazon QuickSight to create a dashboard with our Amazon Customer Reviews Dataset.

Setup the Datasource

To add our dataset,

1. Click on “Manage data” on the upper right corner as shown in Figure 3-15, and then on “New data set”.

Figure 3-15. Amazon QuickSight > Manage data.

2. Click on “Athena” and provide a Data source name, for example amazon_reviews_parquet. Click on “Create data source”.

3. Select the previously generated dsoaws database from the drop-down list, and our amazon_reviews_parquet table. Click on

“Select”.

4. Choose the “Directly query your data” option, and click on “Visualize”.
5. You should now see an empty graph and our dataset columns listed on the left as shown in Figure 3-16.

Add
Undo
Redo
amazon_reviews_parquet analysis
Autosave ON
Print
Capture
Share
N. Virg...
Admin...
Visualize
Filter
Insights
Story
Parameters
Actions
Themes
Data set
amazon_reviews_parquet
Fields list
customer_id
helpful_votes
marketplace
product_category
product_id
product_parent
product_title
review_body
Visual types
Field wells
Sheet 1
AutoGraph
Choose 1 or more fields and let QuickSight choose the most appropriate chart

Figure 3-16. Amazon QuickSight after connecting our dataset

Query and Visualize the Dataset Within QuickSight

So let's start visualizing.

1. Select a Horizontal chart bar graph under Visual types in the lower left corner as shown in Figure 4.17.

Add
Undo
Redo
amazon_reviews_parquet analysis
Autosave ON
Print
Capture
Share
N. Virg...
Admin...
Visualize
Filter
Insights
Story
Parameters
Actions
Themes
Data set
amazon_reviews_parquet
Fields list
customer_id
helpful_votes
marketplace
product_category
product_id
product_parent
product_title
review_body
Visual types
Field wells
Y axis
Add a dimension here
Value
Add measures here
Group/Color
Add a dimension here
Sheet 1
You need to add or remove fields
Horizontal bar chart requires
1 dimension in Y axis

Figure 3-17. Amazon QuickSight, choose Horizontal bar chart.

2. Then drag the product_category field to the y-axis and value boxes. The visualization should now look like shown in Figure 3-18.

Add
Undo
Redo
amazon_reviews_parquet analysis
Autosave ON
Print
Capture
Share
N. Virg...
Admin...
Visualize
Filter
Insights
Story
Parameters
Actions
Themes
Data set
amazon_reviews_parquet
Fields list
customer_id
helpful_votes
marketplace
product_category
product_id
product_parent
product_title
review_body
Visual types
Field wells
Y axis
product_category
Value
product_category (Count)
Group/Color
Add a dimension here
Sheet 1
Count of Product_category by Product_category
Books
Digital_Ebook_Purchase
Wireless
Video_DVD
PC
Mobile_Apps
Home
Music
Apparel
Health_&_Personal_Care
0
5M
10M
15M
20M
25M

Figure 3-18. Amazon QuickSight, visualizing review counts per product category.

In just a few clicks, we have a visualization of review counts per product category - accessible by any device, anywhere.

Figure 3-19 shows how you can access the dashboard from a mobile device with the Amazon QuickSight Mobile App.

Figure 3-19. Using the Amazon QuickSight Mobile App to access dashboards.

Using this visualization, we quickly see the imbalance in our dataset, matching the results of our earlier Athena query. The two product categories of "Books" and "Digital_Ebook_Purchase" have by far the most reviews. We are going to address this in the next chapter where we prepare the data for model training.

Another important step in our early data exploration is to look for any data quality issues in our dataset. Let's do this next.

Detect Data Quality Issues with Apache Spark

Data is never perfect - especially in a dataset with 130+ million rows spread across 20 years! Additionally, data quality may actually degrade over time as new application features are introduced and others are retired. Schemas evolve, code gets crusty, and queries get slow.

Since data quality is not always a priority for the upstream application teams, the downstream data engineering and data science teams need to handle bad or missing data. We want to make sure that our data is high quality for our downstream consumers including Business Intelligence, ML Engineering, and Data Science teams.

Figure 3-20 shows how applications generate data for engineers, scientists and analysts to consume, and which tools and services the various teams are likely to interact with to access that data.

Figure 3-20. Applications generate data for engineers, scientists and analysts to consume.

Data quality can halt a data processing pipeline in its tracks. If these issues are not caught early, they can lead to misleading reports (ie. double-counted revenue), biased AI/ML models (skewed towards/against a single gender or race), and other unintended data products.

To catch these data issues early, we use Deequ, an open source library from Amazon that uses Apache Spark to analyze data quality, detect anomalies,

and even “notify the Data Scientist at 3am” about a data issue. Deequ continuously analyzes data throughout the complete, end-to-end lifetime of the model from feature engineering to model training to model serving in production.

You can see a high-level overview of the Deequ architecture and components in Figure 3-21 below.

Figure 3-21. Overview of Deequ components: constraints, metrics, and suggestions.

Learning from run to run, Deequ will suggest new rules to apply during the next pass through the dataset. Deequ learns the baseline statistics of our dataset at model training time, for example - then detects anomalies as new data arrives for model prediction. This problem is classically called “training-serving skew”. Essentially, a model is trained with one set of learned constraints, then the model sees new data that does not fit those existing constraints. This is a sign that the data has shifted - or skewed - from the original distribution.

Since we have 130+ million reviews, we need to run Deequ on a cluster vs. inside our notebook. This is the trade-off of working with data at scale. Notebooks work fine for exploratory analytics on small data sets, but not suitable to process large data sets or train large models. We will use a notebook to kick off a Deequ Spark job on a cluster using SageMaker Processing Jobs.

SageMaker Processing Jobs

Processing Jobs as shown in Figure 3-22 can run any Python script - or custom Docker image - on the fully managed, pay-as-you-go AWS infrastructure using familiar open source tools such as Scikit-Learn and Apache Spark.

Figure 3-22. Container for a Processing Job.

Fortunately, Deequ is a high-level API on top of Apache Spark, so we use Processing Jobs to run our large-scale analysis job.

> **NOTE**
>
> Deequ is similar to TensorFlow Extended (TFX) in concept - specifically, the TensorFlow Data Validation TFDV) component. However, Deequ builds upon popular open source Apache Spark to increase usability, debug-ability, and scalability. Additionally, Apache Spark and Deequ natively support the Parquet format - our preferred file format for analytics.

Analyze Our Dataset with Deequ and Apache Spark

Table 4-1 shows just a few of the metrics that Deequ supports:

Metric	Description	Usage Example
ApproxCountDistinct	Approximate number of distinct values using HLL++	ApproxCountDistinct(“review_id”)
ApproxQuantiles	Approximate quantiles of a distribution.	ApproxQuantiles(“star_rating”, quantiles = Seq(0.1, 0.5, 0.9))
Completeness	Fraction of non-null values in a column.	Completeness(“review_id”)
Compliance	Fraction of rows that comply with the given column constraint.	Compliance(“top star_rating”, “star_rating >= 4.0”)
Correlation	Pearson correlation coefficient	Correlation(“total_votes”, “star_rating”)
Maximum	Maximum value.	Maximum(“star_rating”)
Mean	Mean value; null values are excluded.	Mean(“star_rating”)
Minimum	Minimum value.	Minimum(“star_rating”)
MutualInformation	How much information about one column can be inferred from another column	MutualInformation(Seq(“total_votes”, “star_rating”))
Size	Number of rows in a DataFrame.	Size()
Sum	Sum of all values of a column.	Sum(“total_votes”)
Uniqueness	Fraction of unique values	Uniqueness(“review_id”)

Table 4-1. Sample Deequ metrics.

Let's kick off the Apache Spark-based Deequ analyzer job by invoking the ScriptProcessor and launching a 10-node Apache Spark Cluster right from our notebook. Note: We need to build and push a custom Docker image with Apache Spark installed. This is the `image_uri` variable in this code sample. See the GitHub repository for more details on building this Docker image.

```
from sagemaker.processing import ScriptProcessor
processor =
        ScriptProcessor(base_job_name='SparkAmazonReviewsAnalyzer',
                    image_uri=image_uri,
                    command=['/opt/program/submit'],
                    role=role,
                    instance_count=10,
                    instance_type='ml.r5.8xlarge',
                    max_runtime_in_seconds=600,
                    env={
                        'mode': 'jar',
                        'main_class': 'Main'
                    })
processor.run(code='preprocess-deequ.py',
              arguments=['s3_input_data',
                                                s3_input_data,
                         'S3_output_analyze_data',
                                                s3_output_analyze_data,
              ]
)
```

> **NOTE**
>
> The code above calls a simple Python script that invokes the Apache Spark Scala code. Native PySpark is coming soon for Deequ to avoid this shim.

Below is our Deequ code specifying the various constraints we wish to apply to our TSV dataset. We are using TSV in this example to maintain consistency with the rest of the examples, but we could easily switch to Parquet with just a 1 line code change.

```
  val dataset = spark.read.option("sep", "\t")
                        .option("header", "true")
                        .option("quote", "")
```

```
                        .schema(schema)
                        .csv("s3://amazon-reviews-pds/tsv")
```

Define the analyzers

```
  .addAnalyzer(Size())
  .addAnalyzer(Completeness("review_id"))
  .addAnalyzer(ApproxCountDistinct("review_id"))
  .addAnalyzer(Mean("star_rating"))
  .addAnalyzer(Compliance("top star_rating", "star_rating >= 4.0"))
  .addAnalyzer(Correlation("total_votes", "star_rating"))
  .addAnalyzer(Correlation("total_votes", "helpful_votes"))
Define the checks, compute the metrics, and verify check conditions
  .addCheck(
    Check(CheckLevel.Error, "Review Check")
    .hasSize(_ >= 130000000) // at least 130 million rows
    .hasMin("star_rating", _ == 1.0) // min is 1.0
    .hasMax("star_rating", _ == 5.0) // max is 5.0
    .isComplete("review_id") // should never be NULL
    .isUnique("review_id") // should not contain duplicates
    .isComplete("marketplace") // should never be NULL
        .isContainedIn("marketplace", Array("US", "UK", "DE", "JP", "FR"))
    .isNonNegative("year")) // should not contain negative values
  )
```

We have defined our set of constraints and assertions to apply to our dataset. Let's run the job and ensure that our data is what we expect.

Below in Table 4-2 are the results from our Deequ job summarizing the results of the constraints and checks that we specified:

Table 3-1. Deequ job results.

check_name	columns	value
ApproxCountDistinct	review_id	149075190
Compleness	review_id	1.00
Compliance	Marketplace contained in US,UK, DE,JP,FR	1.00
Compliance	top star_rating	0.79
Correlation	helpful_votes,total_votes	0.99
Correlation	total_votes,star_rating	-0.03
Maximum	star_rating	5.00
Mean	star_rating	4.20
Minimum	star_rating	1.00
Size	*	150962278
Uniqueness	review_id	1.00

We learned the following:

- review_id has no missing values and approximately (within 2% accuracy) 149,075,190 unique values.
- 79% of reviews have a “top” star_rating of 4 or higher.
- total_votes and star_rating are weakly correlated.
- helpful_votes and total_votes are strongly correlated.
- The average star_rating is 4.20.
- The dataset contains exactly 150,962,278 reviews (1.27% different than ApproxCountDistinct).

Deequ also suggests useful constraints based on the current characteristics of our dataset. This is useful when we have new data entering the system that may differ statistically from the original dataset. In the real world, this is very common as new data is coming in all the time.

In Table 4-3 are the checks - and the accompanying Scala code - that Deequ suggests we add to detect anomalies as new data comes arrives into the system:

Table 3-2. Deequ suggestions for checks to be added.

column	check	deequ_code
customer_id	‘customer_id’ has type Integral	.hasDataType(\customer_id\”, ConstrainableDataTypes.Integral)”
helpful_votes	‘helpful_votes’ has no negative values	.isNonNegative(\helpful_votes\”)”
review_he	‘review_headline’ has less than 1% missing values	.hasCompleteness(\review_headline\”, _ >= 0.99, Some(\"It should be above 0.99!\”))”

a d l i n e		
product_category	‘product_category’ has value range ‘Books', ‘Digital_Ebook_Purchase', ‘Wireless', ‘PC', ‘Home', ‘Apparel', ‘Health & Personal Care', ‘Beauty', ‘Video DVD', ‘Mobile_Apps', ‘Kitchen', ‘Toys', ‘Sports', ‘Music', ‘Shoes', ‘Digital_Video_Download', ‘Automotive', ‘Electronics', ‘Pet Products', ‘Office Products', ‘Home Improvement', ‘Lawn and Garden', ‘Grocery', ‘Outdoors', ‘Camera', ‘Video Games', ‘Jewelry', ‘Baby', ‘Tools', ‘Digital_Music_Purchase', ‘Watches', ‘Musical Instruments', ‘Furniture', ‘Home Entertainment', ‘Video', ‘Luggage', ‘Software', ‘Gift Card', ‘Digital_Video_Games', ‘Mobile_Electronics', ‘Digital_Software', ‘Major Appliances', ‘Personal_Care_Appliances’	.isContainedIn(\product_category\”, Array(\"Books\”, \"Digital_Ebook_Purchase\”, \"Wireless\”, \"PC\”, \"Home\”, \"Apparel\”, \"Health & Personal Care\”, \"Beauty\”, \"Video DVD\”, \"Mobile_Apps\”, \"Kitchen\”, \"Toys\”, \"Sports\”, \"Music\”, \"Shoes\”, \"Digital_Video_Download\”, \"Automotive\”, \"Electronics\”, \"Pet Products\”, \"Office Products\”, \"Home Improvement\”, \"Lawn and Garden\”, \"Grocery\”, \"Outdoors\”, \"Camera\”, \"Video Games\”, \"Jewelry\”, \"Baby\”, \"Tools\”, \"Digital_Music_Purchase\”, \"Watches\”, \"Musical Instruments\”, \"Furniture\”, \"Home Entertainment\”, \"Video\”, \"Luggage\”, \"Software\”, \"Gift Card\”, \"Digital_Video_Games\”, \"Mobile_Electronics\”, \"Digital_Software\”, \"Major Appliances\”, \"Personal_Care_Appliances\”))”
vine	‘vine’ has value range ‘N’ for at least 99.0% of values	.isContainedIn(\vine\”, Array(\"N\”), _ >= 0.99, Some(\"It should be above 0.99!\”))”

In addition to the Integral type and not-negative checks, Deequ also suggests that we constrain `product_category` to the 43 currently-known values including `Books`, `Software`, etc. Deequ also recognized that at least 99% of the `vine` values are `N` and < 1% of `review_headline` values are empty, so it recommends that we add checks for these conditions moving forward.

Increase Performance and Reduce Cost

In this section, we want to provide some tips & tricks to increase performance and reduce cost during data exploration. You can optimize expensive SQL `COUNT` queries across large datasets by using approximate counts. Leveraging Redshift AQUA you can reduce network I/O and increase query performance. And if you feel your QuickSight dashboards could benefit from a performance increase, have a look at QuickSight SPICE.

Approximate Counts with HyperLogLog

Counting is a big deal in analytics. We always need to count users (daily active users), orders, returns, support calls, etc. Maintaining super-fast counts in an ever-growing dataset can be a critical advantage over competitors.

Both Redshift and Athena support HyperLogLog (HLL), a type of "cardinality-estimation" or COUNT DISTINCT algorithm designed to provide highly accurate counts (<2% error) in a small fraction of the time (seconds) requiring a tiny fraction of the storage (1.2KB) to store 130+ million separate counts.

The way this works is that Redshift and Athena update a tiny HLL data structure when inserting new data into the database (think of a tiny hash table). The next time a count query arrives from a user, Redshift and Athena simply look up the value in the HLL data structure and quickly returns the value - without having to physically scan all the disks in the cluster to perform the count.

NOTE

Other forms of HLL include HyperLogLog++, Streaming HyperLogLog, and HLL-TailCut+ among others. Count-Min Sketch and Bloom Filters are similar algorithms for approximating counts and set membership, respectively.

Let's compare the execution times of both `SELECT APPROXIMATE COUNT()` and `SELECT COUNT()` in Redshift as follows:

```
        %%time
df = pd.read_sql_query("""
SELECT APPROXIMATE COUNT(DISTINCT customer_id)
FROM {}.{}
GROUP BY product_category
""".format(redshift_schema, redshift_table_2015), engine)
        -------------
%%time
df = pd.read_sql_query("""
SELECT COUNT(DISTINCT customer_id)
FROM {}.{}
GROUP BY product_category
""".format(redshift_schema, redshift_table_2015), engine)
```

You should see outputs similar to this:

For the query which used `APPROXIMATE COUNT()`:

```
CPU times: user 3.66 ms, sys: 3.88 ms, total: 7.55 ms
Wall time: 18.3 s
```

For the query which used `COUNT()`:

```
CPU times: user 2.24 ms, sys: 973 µs, total: 3.21 ms
Wall time: 47.9 s
```

NOTE

Note that we run the APPROXIMATE COUNT first to factor out the performance boost of the query cache. The COUNT is much slower. If you re-run, both queries will be very fast due to the query cache.

We see that APPROXIMATE COUNT DISTINCT is 160% faster than regular COUNT DISTINCT in this case. The results were approximately 1.2% different - satisfying the <2% error guaranteed by HyperLogLog.

> **NOTE**
>
> Reminder that HyperLogLog is an approximation. Approximations may not be suitable for use cases that require exact numbers such as financial reporting, etc.

Dynamically Scale Your Data Warehouse with Redshift AQUA

Existing data warehouses move data from storage nodes to compute nodes during query execution. This requires high network I/O between the nodes - and reduces query performance.

Figure 3-23 below shows a traditional data warehouse architecture with shared, centralized storage.

Figure 3-23. Traditional Data Warehouse with Shared, Centralized Storage.

Redshift Advanced Query Accelerator (AQUA) is a hardware-accelerated, distributed cache on top of your Redshift data warehouse. AQUA uses custom, AWS-designed chips to perform computations directly in the cache. This reduces the need to move data from storage nodes to compute nodes - and therefore reducing network I/O and increasing query performance.

These AWS-designed chips are implemented in FPGAs and help speed up data encryption and compression for maximum security of your data. AQUA dynamically scales out more capacity, as well.

Figure 3-24 shows the AQUA architecture.

Figure 3-24. Redshift Advanced Query Accelerator (AQUA) Dynamically Scales Out As Needed.

Improve Dashboard Performance with QuickSight SPICE

Amazon QuickSight is built with the "Super-fast, Parallel, In-memory Calculation Engine" or SPICE. SPICE uses a combination of columnar storage, in-memory storage, and machine code generation to run low-latency queries on large datasets. Amazon QuickSight updates its cache as data changes in the underlying data sources including S3 and Redshift.

Summary

In this chapter, we answered various questions about our data using various tools from the AWS Analytics Stack including Athena and Redshift. We created a Business Intelligence (BI) dashboard using QuickSight and deployed a SageMaker Processing Job using Deequ and Apache Spark to continuously monitor data quality and detect anomalies as new data arrives into the system. This continuous monitoring creates confidence in our data pipelines - and allows downstream teams including Data Scientists and AI/ML Engineers to develop highly-accurate and relevant models for our applications to consume.

In the next chapter, we will engineer our dataset and prepare "features" to use in our model training, optimization, and deployment efforts going forward.

Chapter 4. Preparing the Dataset for Model Training

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *support@pipeline.ai*.

In the previous chapter, we explored our dataset using SageMaker Notebooks and various Python-based visualization libraries. We gained some key insights using customer feedback on many Amazon.com products. In addition, we analyzed the quality of our dataset using managed SageMaker Processing Jobs and open source libraries including AWS Deequ and Apache Spark. In this chapter, we discuss how to transform human-readable data into machine-readable vectors in a process called "feature engineering." Specifically, we will convert the raw `review_body` text from the Amazon Customer Reviews Dataset into numeric vectors to train and tune our predictive models in the next chapters.

Perform Feature Engineering

AI and machine learning algorithms are mathematics and statistics based - and therefore require numbers and vectors instead of raw text and images. These "feature vectors", often called "embeddings" in the NLP domain, are projected into a high-dimensional vector space called a "learned representation." These learned representations provide better separability and facilitate data clustering and classification as shown in Figure 6-1.

Figure 4-1. Transforming Data into an Alternate, Learned Representation for Better Separability
https://en.wikipedia.org/wiki/Feature_learning

NOTE

One of the premier AI and machine learning conferences in the world is called International Conference of *Learned Representations* (ICLR).

One of the most important aspects of AI and machine learning, feature engineering usually requires more time than any other phase in the typical machine learning pipeline. Feature engineering helps to reduce data dimensionality, prevent certain features from statistically dominating the algorithm, speed up model-training time, and improve overall model-prediction accuracy.

Careless feature engineering can lead to disastrous results. At worst, poor feature engineering can lead to socially-destructive models that propagate racial, gender, and age bias. At best, poor feature engineering produces suboptimal models that make poor movie recommendations, overstated revenue forecasts, or excess inventory.

While domain experts can certainly help evaluate which features to include and how they should be engineered, there are certain "latent" features hidden in our data sets and not immediately-recognizable by a human. Netflix's recommendation system is famous for discovering new movie genres beyond the usual drama, horror, and romantic comedy. For example, they discovered very specific genres such as "Gory Canadian Revenge Movies," "Sentimental Movies about Horses for Ages 11-12," "Romantic Crime Movies based on Classic Literature," and "Raunchy Mad Scientist Comedies." Figure 6-2 shows more "secret" genres discovered by Netflix's Viewing History Service - code named, "VHS," like the popular video tape format from the 80's and 90's.

Action & Adventure (1365)

Asian Action Movies (77232)
Classic Action & Adventure (46576)
Action Comedies (43040)
Action Thrillers (43048)
Adventures (7442)
Comic Book and Superhero Movies (10118)
Westerns (7700)
Spy Action & Adventure (10702)
Crime Action & Adventure (9584)
Foreign Action & Adventure (11828)
Martial Arts Movies (8985)
Military Action & Adventure (2125)

Anime (7424)

Adult Animation (11881)
Anime Action (2653)
Anime Comedies (9302)
Anime Dramas (452)
Anime Features (3063)
Anime Sci-Fi (2729)
Anime Horror (10695)
Anime Fantasy (11146)
Anime Series (6721)

Children & Family Movies (783)

Movies for ages 0 to 2 (6796)
Movies for ages 2 to 4 (6218)
Movies for ages 5 to 7 (5455)
Movies for ages 8 to 10 (561)
Movies for ages 11 to 12 (6962)
Education for Kids (10659)
Disney (67673)
Movies based on children's books (10056)
Family Features (51056)
TV Cartoons (11177)
Kids' TV (27346)
Kids Music (52843)
Animal Tales (5507)

Classic Movies (31574)

Classic Comedies (31694)
Classic Dramas (29809)
Classic Sci-Fi & Fantasy (47147)
Classic Thrillers (46588)
Film Noir (7687)

Dramas (5763)

Biographical Dramas (3179)
Classic Dramas (29809)
Courtroom Dramas (528582748)
Crime Dramas (6889)
Dramas based on Books (4961)
Dramas based on real life (3653)
Tearjerkers (6384)
Foreign Dramas (2150)
Sports Dramas (7243)
Gay & Lesbian Dramas (500)
Independent Dramas (384)
Teen Dramas (9299)
Military Dramas (11)
Period Pieces (12123)
Political Dramas (6616)
Romantic Dramas (1255)
Showbiz Dramas (5012)
Social Issue Dramas (3947)

Faith & Spirituality (26835)

Faith & Spirituality Movies (52804)
Spiritual Documentaries (2760)
Kids Faith & Spirituality (751423)

Foreign Movies (7462)

Art House Movies (29764)
Foreign Action & Adventure (11828)
Classic Foreign Movies (32473)
Foreign Comedies (4426)
Foreign Documentaries (5161)
Foreign Dramas a(2150)
Foreign Gay & Lesbian Movies (8243)
Foreign Horror Movies (8654)
Foreign Sci-Fi & Fantasy (6485)
Foreign Thrillers (10306)
Romantic Foreign Movies (7153)
African Movies (3761)
Australian Movies (5230)
Belgian Movies (262)
Korean Movies (5685)
Latin American Movies (1613)
Middle Eastern Movies (5875)
New Zealand Movies (63782)
Russian (11567)
Scandinavian Movies (9292)
Southeast Asian Movies (9196)
Spanish Movies (58741)

Independent Movies (7077)

Experimental Movies (11079)
Independent Action & Adventure (11804)
Independent Thrillers (3269)
Romantic Independent Movies (9916)
Independent Comedies (4195)
Independent Dramas (384)

Music (1701)

Kids Music (52843)
Country & Western/Folk (1105)
Jazz & Easy Listening (10271)
Latin Music (10741)
Urban & Dance Concerts (9472)
World Music Concerts (2856)
Rock & Pop Concerts (3278)

Musicals (13335)

Classic Musicals (32392)
Disney Musicals (59433)
Showbiz Musicals (13573)
Stage Musicals (55774)

Romantic Movies (8883)

Romantic Favorites (502675)
Quirky Romance (36103)
Romantic Independent Movies (9916)
Romantic Foreign Movies (7153)
Romantic Dramas (1255)
Steamy Romantic Movies (35800)
Classic Romantic Movies (31273)
Romantic Comedies (5475)

Sci-Fi & Fantasy (1492)

Action Sci-Fi & Fantasy (1568)
Alien Sci-Fi (3327)
Classic Sci-Fi & Fantasy (47147)
Cult Sci-Fi & Fantasy (4734)
Fantasy Movies (9744)
Sci-Fi Adventure (6926)
Sci-Fi Dramas (3916)
Sci-Fi Horror Movies (1694)
Sci-Fi Thrillers (11014)
Foreign Sci-Fi & Fantasy (6485)

Figure 4-2. "Secret" Netflix genres derived from the internal Viewing History Service (VHS)

At a high-level, feature engineering is divided into 3 logical types: selection, creation, and transformation.

Feature selection identifies the data points that best-represent your dataset - and filters out irrelevant or redundant features using statistical methods. If two data points are highly correlated such as `total_votes` and `helpful_votes`, then perhaps only one is needed to train our model. Selecting only one of these data points helps to reduce feature dimensionality and potentially leads to faster model training while preserving model accuracy.

Feature creation combines existing data points into new features that help improve the predictive power of your model. For example, combining `review_headline` and `review_body` into a single feature may lead to more-accurate predictions than using them separately.

Feature transformation converts data from one representation to another to facilitate machine learning. Transforming continuous values such as a timestamp into categorical "bins" such as hourly, daily, or monthly helps to reduce dimensionality. Two common statistical feature transformations are normalization and standardization. Normalization scales all values of a particular data point between 0 and 1, while standardization transforms the values to a mean of 0 and standard deviation of 1. These techniques help reduce the impact of large-valued data points such as number of reviews (represented in 1,000's) over small-valued data points such as helpful_votes (represented in 10's.) Without these techniques, the model could potentially favor the number of reviews over helpful_votes given the order of magnitude difference in values.

Let's walk through a typical feature engineering pipeline from feature selection to feature transformation as shown in Figure 6-2.

Figure 4-3. Steps in a Typical Feature Engineering Pipeline

Select Training Features and Labels

While every data point in the Amazon Customer Reviews Dataset should be considered, we have decided to predict the `star_rating` from the `review_body`. We can use this model to predict the `star_rating` of Twitter comments outside of our review system, for example.

The `star_rating` is the "label" as it's commonly called in AI and machine learning. And `review_body` is the basis of our "features". The label is the actual observed value of 1 through 5 in the dataset. This is ultimately the value that we will predict once our model is trained. During training, we use both the features and the label to train our model to predict a `star_rating` label from `review_body` text in the wild. Let's create a dataframe that contains both the `star_rating` and `review_body` for each review.

import csv

```
df = pd.read_csv('./data/amazon_reviews_us_Digital_Software_v1_00.tsv.gz',
                 delimiter='\t',
                 quoting=csv.QUOTE_NONE,
                 compression='gzip')
df.head(5)
```

star_rating	review_body
1	Poor business decision to strip away user abil...
5	Avast is an easy to use and download. I feel i...
2	Problems from the start. It has been 30 days,...
4	Works well.
3	Hard to use

Since the `star_rating` label is discrete and categorical (1, 2, 3, 4, 5), we will use a "classification" algorithm as suggested by the Scikit-Learn Algorithm "Cheat Sheet" as shown in Figure 6-4.

Figure 4-4. Scikit-Learn "Cheat Sheet" for Algorithm Selection

> **NOTE**
>
> In our case, `star_rating` is considered categorical - and not continuous - as there are only 5 possible values: 1, 2, 3, 4, 5. If `star_rating` contained continuous values such as 3.14, 4.20, or 4.91, then we would define `star_rating` as continuous and likely choose a regression model instead of classification.

Instead of the traditional machine learning classification algorithms listed in the Cheat Sheet, we will use a neural-network based classification model using Keras and TensorFlow. We will dive deep into model training in the next chapter.

Let's prepare our Amazon Customer Reviews Dataset to train a model that predicts `star_rating` (1-5) from `review_body` text.

Balance the Dataset to Improve Your Model

In the previous chapter, we showed the breakdown of `star_rating` for all reviews in our dataset as shown in Figure 6-5.

Figure 4-5. Unbalanced Number of Reviews by Star Rating

We saw that approximately 62% of all reviews have a star rating of 5. If we naively train on this unbalanced dataset, our classifier will simply learn to predict 5 every time for the star_rating since 62% is better than 20% if it randomly guessed 1, 2, 3, 4, or 5. In other words, an unbalanced training dataset may create a model with artificially-high accuracy when, in reality, it learned to just predict 5 every time. This model would not do well in a production setting.

> **NOTE**
>
> Some algorithms like XGBoost support a scaling factor to counteract the problem of unbalanced classes. However, in general, it's a good idea to handle class imbalance during the feature engineering process to avoid misusing these features later.

There are two common ways to balance your dataset: undersampling the majority classes (ie. star_rating 5, in this case) and oversampling the minority classes (ie. star_rating 2 and 3.) We see examples of these in Figure 6-6 and 6-7.

Figure 4-6. Undersampling the majority class down to the minority class

Figure 4-7. Oversampling the minority class up to the majority class

The idea is to evenly distribute data along a particular label or "class" as it's commonly called. In our case, the class is our categorical `star_rating` field. Therefore, we want our training dataset to contain a consistent number of reviews for each star_rating: 1, 2, 3, 4, and 5. The original dataset is imbalanced as shown in Figure 6-8.

Figure 4-8. Unbalanced Dataset for `star_rating` Category

Here is the code to undersample the original dataset using `star_rating`:

```
five_star_df = resample(five_star_df,
                        replace = False,
                        n_samples = len(two_star_df),
                        random_state = 27)
four_star_df = resample(four_star_df,
                        replace = False,
                        n_samples = len(two_star_df),
                        random_state = 27)
three_star_df = resample(three_star_df,
                         replace = False,
                         n_samples = len(two_star_df),
                         random_state = 27)
two_star_df = resample(two_star_df,
                       replace = False,
                       n_samples = len(two_star_df),
                       random_state = 27)
one_star_df = resample(one_star_df,
                       replace = False,
                       n_samples = len(two_star_df),
                       random_state = 27)
df_balanced = pd.concat([five_star_df,
```

```
    four_star_df,
    three_star_df,
    two_star_df,
    one_star_df])
df_balanced = df_balanced.reset_index(drop=True)
```

We now have a balanced dataset as shown in Figure 6-9.

Figure 4-9. Balanced Dataset for `star_rating` class

One drawback to undersampling is that your training dataset size is sampled down to the size of the smallest category. This can reduce the predictive power of your trained models. In this example, we reduced the number of reviews by 65% from approximately 100,000 to 35,000.

Oversampling will artificially create new data for the under-represented class. In our case, star_rating 2 and 3 are under-represented. One common oversampling technique is called Synthetic Minority Oversampling Technique (SMOTE). Oversampling techniques use statistical methods such as interpolation to generate new data from your current data. They tend to work better when you have a larger data set, so be careful when using oversampling on small datasets with a low number of minority class examples. Figure 6-10 shows SMOTE generating new examples for the minority class to improve the imbalance.

Figure 4-10. Using Synthetic Minority Oversample Technique (SMOTE) to generate new data for the minority class (Reference: https://imbalanced-learn.readthedocs.io/en/stable/over_sampling.html)

Split the Dataset into Train, Validation, and Test

Model development typically follows three phases: model *training*, model *validating*, and model *testing* phases as shown in Figure 6-11.

Figure 4-11. Phases of a Typical Model Development Lifecycle

To align with these three phases, we split the data into separate train, validation, and test datasets. The train dataset is used for model training. The validation dataset is used to validate the model training configuration called the "hyper-parameters". And the test dataset is used to test the chosen hyper-parameters. For our model, we chose 90% train, 5% validation and 5% test as this breakdown, shown in Figure 6-12, works well for our dataset and model.

Figure 4-12. Dataset Splits for the Typical Phases of the Model Development Lifecycle

Let's split the data using Scikit-Learn's `train_test_split` function with the `stratify` parameter set to the `star_rating` field to preserve our balance efforts from above. If we don't specify the stratify parameter, the split function is free to choose any data in the given dataset causing the splits to become unbalanced.

from sklearn.model_selection import train_test_split

```
        df_all_data = ... # <== all data
# Split all data into 90% train and 10% holdout
df_train, df_holdout = train_test_split(df_all_data,
test_size=0.10,
                                                                stratify=df['star_rating'])

# Split holdout data into 50% validation and 50% test
df_validation, df_test = train_test_split(df_holdout,
test_size=0.50,
        stratify=df_holdout['star_rating'])
```

We now have three evenly balanced splits for the train, validation, and test phases.

Each of the three phases should use a separate and independent dataset - otherwise "leakage" may occur. Leakage happens when data is leaked from one phase of modeling into another through the splits. Leakage can artificially inflate the accuracy of your model.

Time-series data is often prone to leakage across splits. Companies often want to validate a new model using “back-in-time” historical information before pushing the model to production. When working with time-series data, make sure your model does not peak into the future accidentally. Otherwise, these models may appear more accurate than they really are.

Additionally, you may want to keep all data for the same customer in the same split. Otherwise, an individual’s customer data is spread across multiple splits which could cause problems. In this case, you would group the data by customer_id before creating the splits. Our model does not require us to group our data by `customer_id`, so we will skip this step.

Transform Raw Text into BERT Embeddings

We will use TensorFlow and a state-of-the-art Natural Language Processing (NLP) and Natural Language Understanding (NLU) neural network architecture called BERT. Unlike previous generations of NLP models such as Word2Vec, BERT captures the bi-directional (left-to-right and right-to-left) context of each word in a sentence. This allows BERT to learn different meanings of the same word across different sentences. For example, the meaning of the word “bank” is different between these two sentences: “A thief stole money from the *bank* vault” and “Later, he was arrested while fishing on a river *bank*.”

For each `review_body`, we use BERT to create a feature vector within a previously-learned, high-dimensional vector space of 30,000 words or “tokens.” BERT learned these tokens by training on millions of documents including Wikipedia and Google Books.

Let’s use a variant of BERT called DistilBert. DistilBert is a light-weight version of BERT that is 60% faster, 40% smaller, and preserves 97% of BERT’s language understanding capabilities. We use a popular Python library called Transformers to perform the transformation.

from transformers import DistilBertTokenizer

```
tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-uncased')
```

tokens = tokenizer.tokenize(“"""I needed an antivirus application and know the quality of Norton products. This was a no brainer for me and I am glad it was so simple to get."""”)

The tokenizer performs lower-casing and parses the text into a set of words contained in the pre-trained DistilBert vocabulary. The Transformers library uses another popular library called WordPieces to parse the text into word tokens.

print(tokens)

```
['i', 'needed', 'an', 'anti', '##virus', 'application', 'and', 'know', 'the', 'quality', 'of',
'norton', 'products', '.', 'this', 'was', 'a', 'no', 'brain', '##er', 'for', 'me', 'and', 'i',
'am', 'glad', 'it', 'was', 'so', 'simple', 'to', 'get', '.']
```

Most BERT variants, including DistilBert, have a concept of a "maximum sequence length" that defines the maximum number of tokens used to represent each text input. In our case, any reviews that end up with 128 tokens (after tokenization) will be truncated down to 128. Reviews that end up with less than 128 tokens will be padded to a length of 128. Empirically, we have chosen 128 for the maximum sequence length as this works well for our dataset. In this example, our review_body ends up with less than 128 tokens, so we don't need to truncate in this case.

You must use this same maximum sequence length during feature engineering and model training. So if you want to try a different value, you will need to re-generate the BERT embeddings with the updated value. If you aren't sure which value to choose, you may want to generate multiple versions of the embeddings using 128, 256, and 512 as the maximum sequence length. These seem to work well for most BERT tasks. Larger values will likely increase model training time due to higher dimensionality.

We still have some more processing to do, however, as our DistilBert model requires the following three numeric arrays of length 128 derived from the text-based tokens above:

input_ids

The numeric id of the token from the BERT vocabulary

input_mask

Specifies which tokens BERT should pay attention to (0 or 1)

segment_ids

Always 0 in our case since we are doing a single-sequence NLP task (1 if we were doing a two-sequence NLP task such as next-sentence prediction)

Fortunately, the Transformers tokenizer creates two of the three arrays for us - and even pads and truncates the arrays as needed based on the maximum sequence length!

```
MAX_SEQ_LENGTH = 128
        encode_plus_tokens = tokenizer.encode_plus(text_input.text,
                                                   pad_to_max_length=True,
                                                   max_length=MAX_SEQ_LENGTH)
        # Convert the text-based tokens to ids from the pre-trained BERT vocabulary
        input_ids = encode_plus_tokens['input_ids']
        print(input_ids)
```

Output:

```
[101, 1045, 2734, 2019, 3424, 23350, 4646, 1998, 2113, 1996, 3737, 1997, 10770, 3688, 1012,
2023, 2001, 1037, 2053, 4167, 2121, 2005, 2033, 1998, 1045, 2572, 5580, 2009, 2001, 2061, 3722,
2000, 2131, 1012, 102, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0]
        # Specifies which tokens BERT should pay attention to (0 or 1)
        input_mask = encode_plus_tokens['attention_mask']
        print(input_mask)
```

Output:

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0]
```

The third array, `segment_ids`, is easy to generate on our own since it contains all 0's in our case.

```
        # Segment Ids are always 0 for single-sequence tasks (or 1 if two-sequence tasks)
```

segment_ids = [0] * MAX_SEQ_LENGTH

print(segment_ids)

Output:

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0]
```

Convert Features to TFRecord File Format

The last step in our feature engineering journey is to store our newly-engineering features in the TFRecord file format (.tfrecord file extension). TFRecord is a binary file format optimized for TensorFlow data processing and based on protocol buffers ("protobufs"). TFRecords are cross-platform and cross-language - as well as highly-efficient for data processing workloads. Similar to Parquet, TFRecords are encoded and optimized for sequential access patterns such as model training. For random access, you should not use the TFRecord file format.

While TFRecord is the file format, `tf.train.Example` and `tf.train.Feature` are the primary APIs to create TFRecords with TensorFlow. `tf.train.Feature` stores lists of either `bytes`, `floats`, or `int64`s using `tf.train.BytesList`, `FloatList`, and `Int64List`, respectively.

> **NOTE**
>
> The term "example" in the machine learning context means a row of data used for model training (including the label) or predicting (predicting the label).

Here is the code to convert our features into TFRecords using the TensorFlow API:

```
import tensorflow as tf
import collections

        bert_features = ... # <= features from Transformers library above
        tfrecord_writer = tf.io.TFRecordWriter(output_file)
        tfrecord_features = collections.OrderedDict()
        tfrecord_features['input_ids'] =
tf.train.Feature(int64_list=tf.train.Int64List(
value=bert_features.input_ids))
        tfrecord_features['input_mask'] =
tf.train.Feature(int64_list=tf.train.Int64List(
value=bert_features.input_mask))
        tfrecord_features['segment_ids'] =
tf.train.Feature(int64_list=tf.train.Int64List(
value=bert_features.segment_ids))
        tfrecord_features['label_ids'] =
tf.train.Feature(int64_list=tf.train.Int64List(
value=[bert_features.label_id]))
        tfrecord = tf.train.Example(
features=tf.train.Features(feature=tfrecord_features))
        tfrecord_writer.write(tfrecord.SerializeToString())
        tfrecord_writer.close()
```

`tf.train.Example.SerializeToString()` produces a serialized, binary, and human-unreadable string as follows.


```
[b'\n\xfe\x03\n\x96\x01\n\x0bsegment_ids\x12\x86\x01\x1a\x83\x01\n\x80\x01\n\xb6\x01\n\tinpu
t_ids\x12\xa8\x01\x1a\xa5\x01\n\xa2\x01e\x95\x08\x8d\x10\x8a\x1d\xd0\x0f\xd3\x10\xf4\x07f\n\x9
5\x01\n\ninput_mask\x12\x01\x01\x01\x01\x01\x01\x01\n\tlabel_ids\x12\x05\x1a\x03\n\x01\x04'
]
```


Scale Feature Engineering with SageMaker Processing Jobs

Up until now, we've been working in a SageMaker Notebook on a sample of the dataset. Let's move this code into a SageMaker Processing Job and scale our feature engineering to all 150 million reviews in our dataset. SageMaker Processing Jobs let us parallelize the processing over many nodes in a cluster. In a later chapter, we will automate this step into an end-to-end pipeline. For now, let's just focus on scaling our feature engineering step.

Transform with Scikit-Learn and TensorFlow

Let's balance, split, and transform the entire dataset across a cluster using TensorFlow, Scikit-Learn, BERT, and SageMaker Processing Jobs as shown in Figure 6-13.

Figure 4-13. Transform Raw Text into BERT Embeddings with Scikit-Learn and SageMaker Processing Jobs

First we configure the Scikit-Learn Processing Job with the version of Scikit-Learn, the instance type, and number of instances for our cluster. Since the transformations are stateless, the more instances we use, the faster the processing will happen.

> **NOTE**
>
> Note that we are only using Scikit-Learn to balance and split the data. The heavy lifting is done using TensorFlow and Transformers.

```
from sagemaker.sklearn.processing import SKLearnProcessor
from sagemaker.processing import ProcessingInput, ProcessingOutput
processor = SKLearnProcessor(framework_version='0.20.0',
                             role=role,
                             instance_type='ml.c5.4xlarge',
                             instance_count=2)
```

Next, we start the Processing Job by specifying the location of the transformed features.

```
processor.run(code='preprocess-scikit-text-to-bert.py',
              inputs=[ProcessingInput(source=s3_input_data,
   destination='/opt/ml/processing/input/data/',
   s3_data_distribution_type='ShardedByS3Key')],
              outputs=[
ProcessingOutput(s3_upload_mode='EndOfJob',
output_name='bert-train',
                                source='/opt/ml/processing/output/bert/train'),
ProcessingOutput(s3_upload_mode='EndOfJob',
output_name='bert-validation',
source='/opt/ml/processing/output/bert/validation'),
ProcessingOutput(s3_upload_mode='EndOfJob',
        output_name='bert-test',
        source='/opt/ml/processing/output/bert/test'),
],
                logs=True,
                wait=False)
We can retrieve the S3 output location as follows:
output_config = processing_job_description['ProcessingOutputConfig']
for output in output_config['Outputs']:
    if output['OutputName'] == 'bert-train':
        processed_train_data_s3_uri = output['S3Output']['S3Uri']
    if output['OutputName'] == 'bert-validation':
        processed_validation_data_s3_uri = output['S3Output']['S3Uri']
    if output['OutputName'] == 'bert-test':
        processed_test_data_s3_uri = output['S3Output']['S3Uri']
```

Transform with Apache Spark and TensorFlow

Apache Spark is a powerful data processing and feature transformation engine supported by SageMaker Processing Jobs. While Apache Spark does not support BERT natively, we can use the Python-based BERT Transformers library within a PySpark application to

scale our BERT Transformations across a distributed Spark cluster. In this case, we are using Spark as just a distributed processing engine and Transformers as just another Python library installed in the cluster as shown in Figures 6-14 and 6-15.

Figure 4-14. Apache Spark cluster with multiple libraries installed including TensorFlow and BERT

NOTE

Apache Spark has an implementation of Term Frequency / Inverse Document Frequency (TF/IDF) for feature engineering as part of the Spark ML library. TF/IDF, dating back to the 1980's, requires a stateful pre-training step which limits its scalability and effectiveness for feature engineering. BERT has been pre-trained on millions of documents and generally performs better on our dataset than TF/IDF, so we will use BERT for the feature engineering tasks presented here.

Figure 4-15. Transform Raw Text into BERT Embeddings with Apache Spark

We can spin-up a cloud-native, pay-for-what-you-use Apache Spark cluster to create the BERT vectors from the raw `review_body` data using SageMaker Processing Jobs. We specify the `instance_type` and the `instance_count`. Since Spark performs better with more RAM, we use a high-RAM `r5` instance type. The `image_uri` is a Docker image

that contains Apache Spark. You can build this image yourself using the Dockerfile found in the GitHub repo for this book. For now, let's assume the image is built and in your account's Elastic Container Registry (ECR) Docker repo.

```
from sagemaker.processing import ScriptProcessor
processor =
 ScriptProcessor(base_job_name='spark-amazon-reviews-processor',
image_uri=image_uri,
                 command=['/opt/program/submit'],
                            role=role,
                            instance_count=2,
                            instance_type='ml.r5.8xlarge',
                            env={'mode': 'python'})
```

Let's run the Processing Job. Since Apache Spark efficiently reads and writes to S3 directly, we don't need to specify the usual `ProcessingInput` and `ProcessingOutput` parameters to our `run()` function. Instead, we use the `arguments` parameter to pass in four S3 locations: one for the raw TSV files and the three S3 locations for the generated BERT vectors for the train, validation, and test splits.

```
train_data_bert_output = 's3://{}/{}/output/bert-train'.format(bucket, output_prefix)
validation_data_bert_output = 's3://{}/{}/output/bert-validation'.format(bucket, output_prefix)
test_data_bert_output = 's3://{}/{}/output/bert-test'.format(bucket, output_prefix)
processor.run(code='preprocess-spark-text-to-bert.py',
              arguments=['s3_input_data', s3_input_data,
                         's3_output_train_data', train_data_bert_output,
                         's3_output_validation_data', validation_data_bert_output,
                         's3_output_test_data', test_data_bert_output,
              ],
              logs=True,
              wait=False
)
```

The above code runs in the notebook and launches the `preprocess-spark-text-to-bert.py` script on the SageMaker Processing Job Cluster running Apache Spark. The code snippet below is from the `preprocess-spark-text-to-bert.py` PySpark script:

```
def transform(spark, s3_input_data, s3_output_train_data,
   s3_output_validation_data, s3_output_test_data):

    schema = StructType([
        StructField('marketplace', StringType(), True),
        StructField('customer_id', StringType(), True),
        StructField('review_id', StringType(), True),
        StructField('product_id', StringType(), True),
        StructField('product_parent', StringType(), True),
        StructField('product_title', StringType(), True),
        StructField('product_category', StringType(), True),
        StructField('star_rating', IntegerType(), True),
        StructField('helpful_votes', IntegerType(), True),
        StructField('total_votes', IntegerType(), True),
        StructField('vine', StringType(), True),
        StructField('verified_purchase', StringType(), True),
```

```
        StructField('review_headline', StringType(), True),
        StructField('review_body', StringType(), True),
        StructField('review_date', StringType(), True)
    ])

    df_csv = spark.read.csv(path=s3_input_data,
                            sep='\t',
                            schema=schema,
                            header=True,
                            quote=None)
```

Here is the Spark User-Defined Function (UDF) to transform the raw text into BERT embeddings using the Transformers Python library:

```
MAX_SEQ_LENGTH = 128
DATA_COLUMN = 'review_body'
LABEL_COLUMN = 'star_rating'
LABEL_VALUES = [1, 2, 3, 4, 5]
label_map = {}
for (i, label) in enumerate(LABEL_VALUES):
    label_map[label] = i
def convert_input(label, text):
    encode_plus_tokens = tokenizer.encode_plus(text,
                                           pad_to_max_length=True,
                                               max_length=MAX_SEQ_LENGTH)
    # Convert the text-based tokens to ids from the pre-trained BERT vocabulary
    input_ids = encode_plus_tokens['input_ids']
    # Specifies which tokens BERT should pay attention to (0 or 1)
    input_mask = encode_plus_tokens['attention_mask']
    # Segment Ids are always 0 for single-sequence tasks (or 1 if two-sequence tasks)
    segment_ids = [0] * MAX_SEQ_LENGTH
    # Label for our training data (star_rating 1 through 5)
    label_id = label_map[label]
    return {'input_ids': input_ids, 'input_mask': input_mask, 'segment_ids': segment_ids,
'label_ids': [label_id]}
```

Here is the Spark code that invokes the UDF on each worker in the cluster. Note that we are preparing to write a `TFRecord`, so we are setting up a PySpark schema that matches the desired TFRecord format.

```
    tfrecord_schema = StructType([
      StructField("input_ids", ArrayType(IntegerType(), False)),
      StructField("input_mask", ArrayType(IntegerType(), False)),
      StructField("segment_ids", ArrayType(IntegerType(), False)),
      StructField("label_ids", ArrayType(IntegerType(), False))
    ])
    bert_transformer = udf(lambda text, label: convert_input(text, label), tfrecord_schema)
```

Split the data into train, validation, and test:

```
train_df, validation_df, test_df = features_df.randomSplit(
[0.9, 0.05, 0.05]
 )
```

Save the splits to S3 as TFRecords:

```
train_df.write.format('tfrecord').option('recordType',
'Example').save(path=s3_output_train_data)

validation_df.write.format('tfrecord').option('recordType',
'Example').save(path=s3_output_validation_data)
test_df.write.format('tfrecord').option('recordType', 'Example').save(path=s3_output_test_data)
```

> **NOTE**
>
> We are using `format('tfrecord')` from an open source library that implements the Apache Spark DataFrameReader and DataFrameWriter interfaces for TFRecord. You can find references to this library in our GitHub repository.

Automate Feature Engineering with Amazon Step Functions

Feature engineering is the core of every AI and machine learning pipeline. As new data arrives into the system, we need to continuously transform the raw data into machine-readable feature vectors. While it's perfectly acceptable to manually run this step every week, day, or even hour, you can easily automate this process as new data lands into S3 from an upstream application, for example. The sooner we can convert the raw data into features, the sooner we can kick off another training job to improve our model with the latest data. Let's demonstrate how to implement continuous feature engineering on AWS using Step Functions, EventBridge, S3 Triggers, and Processing Jobs as shown in Figure 6-16.

Figure 4-16. Trigger a Pipeline When New Data Arrives

As new data arrives in S3 - either directly from an application or through data streaming services like Kinesis and Managed Kafka - we want to trigger a pipeline using AWS Step Functions, Airflow DAGs, or Kubeflow Pipelines. We will discuss pipelines in greater detail in a later chapter, but, for now, let's focus on how to configure S3 to invoke a Step Function workflow when new data arrives in S3.

Invoke a Pipeline with S3 Triggers

There are two main ways to trigger a Step Function workflow: time-based and event-based. Time-based is set up to run the workflow every week, 2 days, 4 hours, 15 minutes, etc.

Figure 4-17. Render a Pipeline using the Step Functions Data Science SDK

Similar to configuring a cron job, you can specify the frequency of runs using familiar cron syntax. Event-based triggers will fire when a particular event occurs. Let's explore how to set up an S3 Trigger to start a workflow when new data is written to a specific S3 location. Figure 6-18 shows what we're building.

Figure 4-18. Flow of data landing in S3 to invoking a Step Functions pipeline

First, we need to create a CloudTrail Log and start the logging process on our `watched_bucket`. Let's do this using the AWS command-line interface (CLI) where a separate `cloudtrail_bucket` has been previously created to store the CloudTrail Logs.

We create the CloudTrail as follows:

```
aws cloudtrail create-trail --name dsoaws \
--s3-bucket-name $cloudtrail_bucket \
--is-multi-region-trail
aws cloudtrail start-logging --name dsoaws
aws cloudtrail get-trail-status --name dsoaws
```

```
Output:
{
    "IsLogging": true,
    "LatestDeliveryTime": 1589322691.606,
    "StartLoggingTime": 1589140543.171,
    "LatestDeliveryAttemptTime": "2020-05-12T22:31:31Z",
    "LatestNotificationAttemptTime": "",
    "LatestNotificationAttemptSucceeded": "",
    "LatestDeliveryAttemptSucceeded": "2020-05-12T22:31:31Z",
    "TimeLoggingStarted": "2020-05-10T19:55:43Z",
    "TimeLoggingStopped": ""
}
```

Next, we need to enable Object-level logging on our application S3 `watched_bucket` by selecting the bucket and clicking Properties as shown in Figures 6-19 and 6-20.

Figure 4-19. Select Object-level logging from the Properties tab of our application S3 bucket

Figure 4-20. Enable Object-Level Logging using our newly-created CloudTrail

Now we need to create an EventBus within Amazon EventBridge to capture the S3 events from CloudTrail.

```
events = boto3.client('events')
try:
```

```
    response = events.create_event_bus(Name='dsoaws')
    print(response)
except ClientError as e:
    if e.response['Error']['Code'] == 'ResourceAlreadyExistsException':
        print("You're good. Event Bus already exists.")
    else:
        print("Unexpected error: %s" % e)
response = events.describe_event_bus(Name='dsoaws')
eventbus_arn = response['Arn']
print(eventbus_arn)
We then create an EventBridge Rule to capture S3's write API called PutObject

pattern = {
  "source": [
    "aws.s3"
  ],
  "detail-type": [
    "AWS API Call via CloudTrail"
  ],
  "detail": {
    "eventSource": [
      "s3.amazonaws.com"
    ],
    "eventName": [
      "PutObject"
    ],
    "requestParameters": {
      "bucketName": [
        "{}".format(watched_bucket)
      ]
    }
  }
}
print(pattern)
Lastly, we associate the rule with the Step Function as follows:
response = events.put_targets(
    Rule='S3-Trigger',
    EventBusName='dsoaws',
    Targets=[
        {
            'Id': '1',
            'Arn': stepfunctions_arn,
            'RoleArn': iam_role_eventbridge_arn,
        }
    ]
)
```

At this point, any write (PutObject API) to S3 will trigger the EventBridge rule created above. Trigger this rule will start the Step Function pipeline that transforms our features from raw text to BERT embeddings. We will discuss pipelines in much more detail later in the book. For now, we want to show how to invoke a feature engineering pipeline when new data arrives in S3.

Share Features Through a Feature Store

Feature engineering requires intuition, patience, trial, and error. As more teams utilize AI and machine learning to solve business use cases, the need arises for a centralized, discoverable, and re-usable repository of features. The term for this is, “feature store.” Feature stores are, essentially, a data lake for machine-readable features. Since these features require processing as we demonstrated earlier with the SageMaker Processing Jobs, we would like to store and re-use these features throughout the organization. The following figure shows the relationship between a feature store, data lake, and data warehouse as shown in Figure 6-21.

Figure 4-21. Relationship Between Feature Store, Data Lake, and Data Warehouse

Since we spent a fair amount of time generating our BERT features, we'd like to share them with other teams in our organization. Perhaps these other teams can discover new and improved combinations of our features that we never explored. We'd like to use our feature store to help us safely "travel in time" and avoid leakage. Feature stores can

cache “hot features” into memory to reduce model-training times. A feature store can provide governance and access control to regulate and audit our features. Lastly, a feature store can provide consistency between model training and model predicting by ensuring the same features for both batch training and real-time predicting.

Customers have implemented feature stores using a combination of DynamoDB, ElasticSearch, and S3. DynamoDB and ElasticSearch track metadata such as file format (ie. csv, parquet), BERT-specific data (ie. maximum sequence length), and other summary statistics (ie. min, max, standard deviation). S3 stores the underlying features such as our generated BERT embeddings. This feature store reference architecture is shown in Figure 6-22.

Figure 4-22. Feature Store Reference Architecture on AWS

Summary

In this chapter, we explored the art of feature engineering using a real-world example of transforming raw Amazon Customer Reviews into machine learning features using BERT and TensorFlow. We demonstrated how to automate the feature engineering process by triggering on new data as it arrives in S3. We can now use these features to train a classification model to predict the `star_rating` from raw text in the wild.

Chapter 5. Training Your First Model with SageMaker

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 7th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *support@pipeline.ai*.

In the previous chapter, we used SageMaker Processing Jobs to transform a raw dataset into machine-usable features through the "feature engineering" process. In this chapter, we use these features to train a custom text classifier using TensorFlow, BERT, and SageMaker. We explain the training options provided by SageMaker. Next, we dive deep into the SageMaker infrastructure including containers, networking, and security. We then monitor our training job using CloudWatch and SageMaker Debugger. Lastly, we provide tips to increase performance and reduce costs when developing models with SageMaker.

Understand the SageMaker Infrastructure

Largely container-based, SageMaker manages the infrastructure and helps you focus on your specific machine learning task. Out of the box, you can use pre-built, open source SageMaker containers for many popular AI and machine learning frameworks such as TensorFlow, PyTorch, MXNet, and Scikit-Learn. You can also provide your own Docker containers with the libraries and frameworks of your choice. In this section, we go into more detail about the SageMaker infrastructure including environment variables, S3 locations, security, and encryption.

You can choose to train on a single instance or on a distributed cluster of instances. Amazon SageMaker removes the burden of managing the underlying infrastructure and handles the undifferentiated heavy lifting for you.

SageMaker Container Environment Variables and S3 Locations

Much like a software framework, the SageMaker managed infrastructure provides multiple "hot spots" for your training script to leverage. There are two(2) hot spots that stand out: environment variables and S3 locations.

SageMaker provides your container with many predefined environment variables such as the number of GPUs available to the container (SM_NUM_GPUS) and the log level (SM_LOG_LEVEL). Your training script can use these variables to configure your training job accordingly. These variables are summarized here: https://github.com/aws/sagemaker-containers#sagemaker-containers

SageMaker also provides your container with S3 storage for your training files. For example, a typical training job reads in data files, trains the model, and writes out a model file. Some AI and machine learning frameworks may even save a checkpoint file just in case the training job stalls or fails. In this case, the job can restart from where it left off. These input, output, and checkpoint files must move in and out of the ephemeral Docker container from/to more durable storage like S3. Otherwise, when the training job ends and the Docker container goes away, the data is lost.

While seemingly simple, this mapping is a very critical piece in the training performance puzzle. If this layer mapping is not optimized, your training times will suffer greatly. Later, we will discuss a SageMaker feature called Pipe Mode that specifically optimizes the movement of data at this layer. Figure 7-1 shows the mapping of the file location inside the Docker container to the S3 location outside the container.

Figure 5-1. Mapping of the file location inside the Docker container to the S3 location outside the container

Compute and Network Isolation

As a security best practice, all SageMaker components should be used in a private VPC with no internet connectivity. This requires that you carefully configure IAM roles, VPC endpoints, subnets, and security groups for least-privilege access to Amazon S3, CloudWatch, SageMaker, Redshift, and any other AWS service needed by your SageMaker notebooks, training jobs, etc.

We discussed the concept of IAM roles and how to secure your dataset in the previous chapter on data ingest. Here is a sample S3 Bucket Policy that shows how you can limit access to a specific S3 bucket from a single VPC only:

```
{
   "Version": "2012-10-17",
   "Id": "Policy1415115909152",
   "Statement": [
     {
       "Sid": "Access-to-specific-VPC-only",
       "Principal": "*",
       "Action": "s3:*",
       "Effect": "Deny",
       "Resource": ["arn:aws:s3:::our-sagemaker-bucket",
                    "arn:aws:s3:::our-sagemaker-bucket/*"],
       "Condition": {
         "StringNotEquals": {
           "aws:SourceVpc": "vpc-1a2b3c4d"
         }
       }
     }
   ]
}
```

Network isolation is also important from a high-availability standpoint. While we usually discuss high-availability in terms of microservices and real-time systems, we should also strive to increase the availability of our training jobs.

Our training scripts almost always include pip installing Python libraries from PyPi or downloading pre-trained models from third-party model repositories (or "model zoo's") on the internet. By creating dependencies on external resources, your training job is now at the mercy of these third-party services. If one of these services is temporarily down, your training job may not start.

To improve availability, it is recommended that we reduce as many external dependencies as possible by copying these resources into your Docker images - or into your own S3 bucket. This has the added benefit of reducing network utilization and starting our training jobs faster.

> **NOTE**
>
> At Netflix, we "burned" all dependencies into our Docker images and Amazon Machine Images (AMIs) to remove all external dependencies and achieve higher availability. This was absolutely critical during rapid scale-out and failover events.

Data Encryption at Rest and In-Transit

For data at rest, SageMaker supports both AWS and Customer Managed Keys (CMKs) using the Amazon Key Management Service (KMS). This includes data in S3, local instance storage, network-attached EBS storage, and distributed EFS storage. We are encouraged to enable default encryption for all storage volumes and use deny policies to prevent unencrypted uploads.

For data in-transit, SageMaker supports inter-container encryption for all SageMaker infrastructure containers. This includes traffic between distributed training containers in a cluster.

Develop a SageMaker Model

Just as Amazon.com provides many options to customers through the Amazon.com Marketplace, Amazon SageMaker provides many options for building, training, optimizing, and deploying models. These options offer different levels of customization and complexity. Figure 7-2 shows the three(3) main options.

Figure 5-2. SageMaker Options to Build, Train, Optimize, and Deploy Your Model

Built-In Algorithms

SageMaker provides built-in algorithms that are ready to use out of the box across a number of different domains such as natural language processing, computer vision, anomaly detection, and recommendations. Simply point these highly-optimized algorithms at your data and you will get a fully-trained, easily deployed machine learning model to integrate into your application. These algorithms, shown below, are targeted towards those of us who don't want to manage a lot of

infrastructure, but rather want to re-use battle-tested algorithms used by tens of thousands of customers.

Classification Linear Learner XGBoost KNN	Computer Vision Image Classification Object Detection Semantic Segmentation	Working with Text BlazingText Supervised Unsupervised
Regression Linear Learner XGBoost KNN	Anomaly Detection Random Cut Forests IP Insights	Topic Modeling LDA NTM
Sequence Translation Seq2Seq	Recommendation Factorization Machines	Clustering KMeans
Feature Reduction PCA Object2Vec	Forecasting DeepAR	

Bring Your Own Script or Script Mode

SageMaker offers a more customizable option to let you “bring your own script” often called “script mode”. Script mode lets you focus on your training script while SageMaker provides highly-optimized Docker containers for each of the familiar open source frameworks such as TensorFlow, PyTorch, MXNet, XGBoost, and Scikit-Learn as shown in Figure 7-3.

Figure 5-3. Popular AI and Machine Learning Frameworks

This option is a good balance of high customization and low maintenance. Most of the remaining SageMaker examples in this book will utilize script mode with TensorFlow and BERT for natural language processing (NLP) and natural language understanding (NLU) use cases as shown in Figure 7-4.

Figure 5-4. Script mode with BERT and TensorFlow is a good balance of high customization and low maintenance

Bring Your Own Container

The most customizable option is "bring your own container" (BYOC). This option lets you build and deploy your own Docker container to SageMaker. This Docker container can contain any library or framework. While we maintain complete control over the details of the training script and its dependencies, SageMaker manages the low-level infrastructure for logging, monitoring,

environment variables, S3 locations, etc. This option is targeted at more specialized or systems-focused machine learning folks.

A Brief History of Natural Language Processing

In the previous chapter, we transformed raw Amazon Customer Reviews into BERT feature vectors to ultimately build a review classifier model to predict the `star_rating` from `review_body` text. Before we build our natural language model, we want to provide some background on natural language processing and understanding.

In 1935, a famous British linguist, J.R. Firth, said the following: *"The complete meaning of a word is always contextual, and no study of meaning apart from context can be taken seriously."* Fast forward 80 years to 2013 when word vectors or "word embeddings" began to dominate language representations as shown in Figure 7-5. These word embeddings capture the contextual relationships between words in a set of documents or "corpus", as it is commonly called.

Figure 5-5. A Brief History of Natural Language Processing and Understanding

Contextual Algorithms

Word2Vec, one of the most popular implementations of word embeddings, uses a co-occurrence matrix to represent these relationships. Co-occurrence is the number of times that a pair of words appear together in a given "context window". For example, a context window of 3 means that we look 1 word before and 1 word after - centered around each word - to calculate the number of times that a pair of words appear in the same window. Let's apply the context window of 3 to the following text:

"Books are great. Writers are awesome. Readers are smart."

In this case, the words “Books” and “great” appear together 1 time, “Writers” and “great” appear together 2 times, “are” and “Readers” appear together 3 times as shown in Figure 7-6.

Co-Occurrences with Context Window Size of 3

empty **Books** are great. Writers are awesome. Readers are smart.

Books **are** great. Writers are awesome. Readers are smart.

Books are **great**. Writers are awesome. Readers are smart.

Books are great. **Writers** are awesome. Readers are smart.

Books are great. Writers **are** awesome. Readers are smart.

Books are great. Writers are **awesome**. Readers are smart.

Books are great. Writers are awesome. **Readers** are smart.

Books are great. Writers are awesome. Readers **are** smart.

Books are great. Writers are awesome. Readers are **smart**. *empty*

Figure 5-6. Co-Occurrences with Context Window Size of 3 Centered Around Each Word

We can represent these co-occurrences through a matrix shown in Figure 7-7. Note that the words “Books” and “awesome” never appear together in the same window, so the value is empty (or 0) at that matrix location.

Co-Occurrence Matrix with Context Window of Size 3

	Books	are	great	Writers	awesome	Readers	smart
Books		1	1				
are	1		3	2	2	3	2
great	1	3		2			
Writers		2	2		1		
awesome		2		1		2	
Readers		3			2		1
smart		2				1	

Figure 5-7. Word Co-Occurrence Matrix for Context Window 2

We have now learned a numeric representation of our corpus in a new vector space defined by our co-occurrence matrix. As mentioned in the previous chapter, this learned representation lets us perform mathematical computations such as word similarity and word differences.

GloVe goes one step further and uses recurrent neural networks (RNNs) to encode the *global* co-occurrence of words vs. Word2Vec's *local* co-occurence of words. An RNN is a special type of

neutral network that learns and remembers longer-form inputs such as text sequences and time-series data.

FastText continues the innovation and builds word embeddings using combinations of lower-level character embeddings using character-level RNNs. This character-level focus allows FastText to learn non-English language models with relatively small amounts of data compared to other models. Amazon SageMaker offers a built-in, pay-as-you-go SageMaker algorithm called BlazingText which is an implementation of FastText optimized for AWS. This algorithm was shown in the Built-In Algorithms section above.

There are some drawbacks to this generation of NLP models, however, as they are all forms of static word and character embeddings. While static embeddings capture the semantic meanings of words, they don't actually understand high-level language concepts. In fact, once the embeddings are created, the actual model is often discarded after training (ie. word2vec, GloVe) and simply preserve the word embeddings to use as features for classical machine learning algorithms such as Logistic Regression and XGBoost.

ELMo preserves the trained model and uses two separate Long-Short Term Memory (LSTM) networks: one to learn from left-to-right and one to learn from right-to-left. Neither LSTM uses both the previous and next words at the same time, however. Therefore ELMo does not learn a true bidirectional contextual representation of the words and phrases in the corpus, but it performs very well nonetheless.

NOTE

An LSTM is a special type of Recurrent Neural Network (RNN) that selectively chooses which information to remember and which information to forget. This keeps the LSTM relatively memory and compute efficient, yet maintains very good predictive power.

GPT and the newer GPT-2 and GPT-3 models (GPT-n) preserve the trained model and use a relatively simple neural network architecture called the "Transformer" to learn the contextual word representations. Transformers were popularized along with their attention-mechanism counterpart in the 2017 paper titled, "Attention is All You Need." The GPT-n transformer uses a directional, left-to-right "masked self-attention" mechanism to learn the left-to-right contextual representation as shown in Figure 7-8.

Figure 5-8. GPT-n's Masked Self Attention Mechanism

This prevents the model from peeking ahead to see the next words in the sentence. Even with this limitation, GPT-n performs very well on text generation tasks.

Attention-Based Algorithms

In 2018, a new neural network-based algorithm for NLP was released called Bidirectional Encoder Representations from Transformers (BERT). BERT has revolutionized the field of NLP and NLU and is now widely used throughout the industry at Facebook, LinkedIn, Netflix, and many other AI-first companies. BERT uses a forward-and-backward looking self-attention mechanism that improves upon the GPT-n backward-looking, masked self-attention mechanism.

At its core, BERT's Transformer architecture uses an attention mechanism to "pay attention" to specific and interesting words or phrases as it traverses the corpus. Specifically, the BERT transformer uses "self-attention" to attend every token in the data to all other tokens in the input sequence. Additionally, BERT uses "multi-headed attention" to handle ambiguity in the meanings of words, also called "polysemy" (Greek "poly" = many, "sema" = sign). An example of attention is shown in Figure 7-9 where the word "it" attends highly to the word "movie" as well as the words "funny" and "great", though to a lesser degree than the word "movie."

Figure 5-9. Visualizing Attention

> **NOTE**
>
> For more information on attention, see the paper "Attention Is All You Need" published by Google in 2017. (https://arxiv.org/pdf/1706.03762.pdf)

Without this bi-directional attention, an algorithm would potentially create the same embedding for the word *bank* for the following two(2) sentences: "A thief stole money from the *bank* vault" and "Later, he was arrested while fishing on a river *bank*." Note that the word *bank* has a

different meaning in each sentence. This is easy for humans to distinguish because of our life-long, natural "pre-training", but this is not easy for a machine without similar pre-training.

Training BERT from Scratch

While we can use BERT as-is without training from scratch, it's useful to know how BERT uses word masking and next sentence prediction - in parallel - to learn and understand language.

Masked Language Model (Masked LM)

As BERT sees new text, it masks 15% of the words in each sentence or "sequence" in BERT terminology. BERT then predicts the masked words and corrects itself (aka. "updates the model weights") when it predicts incorrectly. This is called Masked Language Model or Masked LM. Masking forces the model to learn the surrounding words for each sequence as shown in Figure 7-10.

Figure 5-10. BERT masks 15% of input tokens and learns by predicting the masked tokens - correcting itself when it predicts the wrong word.

To be more concrete, BERT is trained by forcing it to predict masked words in a sentence. For example, if we feed in the contents of this book, we can ask BERT to predict the missing word in

the following sentence: "*This book is called Data ____ on AWS*." Obviously, the missing word is "*Science*." This is easy for a human who has been pre-trained on millions of documents since birth, but not easy for a machine - not without training, anyway.

Next Sentence Prediction

At the same time BERT is masking and predicting input tokens, it is also performing Next Sentence Prediction (NSP) on pairs of input sequences. Both of these training tasks are optimized together to create a single accuracy score for the combined training efforts. This results in a more robust model capable of performing word and sentence level predictive tasks.

To perform NSP, BERT randomly chooses 50% of the sentence pairs and replaces one of the two sentences with a random sentence from another part of the document. BERT then predicts if the two sentences are a valid sentence pair or not as shown in Figure 7-11. BERT will correct itself when it predicts incorrectly.

Figure 5-11. BERT Pre-Training: Next Sentence Prediction (NSP)

For more details on BERT, check out the 2018 paper "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." (https://arxiv.org/pdf/1810.04805.pdf).

In most cases, we don't need to train BERT from scratch. Neural networks are designed to be re-used and continuously trained as new data arrives into the system. Since BERT has already been pre-trained on millions of public documents from Wikipedia and the Google Books Corpus, the

vocabulary and learned representations are transferable to a large number of NLP and NLU tasks across a wide variety of domains.

Training BERT from scratch requires a lot of data and compute, it allows BERT to learn a representation of the custom dataset using a highly-specialized vocabulary. Companies like LinkedIn have pre-trained BERT from scratch to learn language representations specific to their domain including job titles, resumes, companies, and business news. The default pre-trained BERT models were not good enough for NLP/NLU tasks. Fortunately, LinkedIn has plenty of data and compute!

Use Pre-Trained BERT Models

ELMo, GPT/GPT-2, and BERT preserve their trained models called, "pre-trained models." Pre-trained on millions of documents across many different domains, these models are not only good at predicting missing words, but also very good at learning the meaning of words, sentence structure, and sentence correlations. Their ability to generate meaningful, relevant, and realistic text is phenomenal and scary. Let's dive deeper into BERT's pretrained models.

BERT's pre-trained models are, like most neural network models, just point-in-time snapshots of the model weights learned from the data seen to date. And like most models, BERT becomes even more valuable with more data.

The core BERT pre-trained models come in "base" and "large" variants that differ by number of layers, attention heads, hidden units, and parameters. We see very good performance with the smaller model with only 12 attention heads and 110 million parameters as shown in Figure 7-12.

	Layers	Hidden Units	Parameters
BERT base	12	768	110M
BERT large	24	1024	340M

Figure 5-12. The core BERT pre-trained models comes in "base" and "large" variants

Additionally, the community has created many pre-trained versions of BERT using domain and language-specific datasets including PatentBERT (U.S. Patent data), ClinicalBERT (healthcare data), CamemBERT (French language), German BERT (German language), and BERTje (Dutch language) as shown in Figure 7-13.

Figure 5-13. Many Community-Driven, Pre-Trained BERT Models

These BERT variants were pre-trained from scratch because the default BERT models, trained on English versions of Wikipedia and Google Books, do not share the same vocabulary as the custom datasets - ie. French CamemBERT and healthcare ClinicalBERT. When training from scratch, we can re-use BERT's neural network transformer architecture, but throw out the pre-trained base model weights learned from Wikipedia and Google Books.

For our Amazon Customer Reviews Dataset, we can safely re-use the default BERT models as they share a similar vocabulary and language representation as our Amazon Customer Reviews Dataset. There is no doubt that training BERT from scratch to learn the specific Amazon.com product catalog would improve accuracy on some tasks such as entity recognition. However, the default BERT models perform very well on our review text, so we will keep things simple and "fine-tune" a default BERT model to create a custom text classifier using our Amazon Customer Reviews Dataset.

Let's re-use the language understanding and semantics learned by the pre-trained BERT model to learn a new, domain-specific NLP task using the Amazon Customer Reviews Dataset. This process, called "fine-tuning," is shown in Figure 7-14.

Figure 5-14. Fine-tuning a pre-trained BERT model for a domain-specific task using a custom dataset

The simplicity and bidirectional nature of the BERT self-attention mechanism allow us to fine-tune the base BERT models to a wide range of out-of-the-box, "downstream" NLP/NLU tasks including text-classification to analyze sentiment, entity-recognition to detect a product name, and next-sentence-prediction to provide answers to natural language questions as shown in Figure 7-15.

Figure 5-15. We can fine-tune the default BERT models to many "downstream" NLP and NLU tasks

Since fine-tuning is a supervised training process (versus pre-training which is unsupervised), masking and next sentence prediction do not happen during fine-tuning - only during pre-training. As a result, fine-tuning is very fast and requires a relatively small number of samples, or reviews in our case. This translates to lower processing power, lower cost, and faster training/tuning iterations.

Fine Tune the BERT Model to Create a Custom Classifier

Since we already generated the BERT embeddings from the raw `review_body` text in the previous chapter, we are ready to go! Let's fine tune BERT to create a custom text classifier that predicts `star_rating` from `review_body` using our dataset as shown in Figure 7-16.

Figure 5-16. Fine-tuning a BERT model to create a custom text classifier with our reviews dataset

We can use this classifier to predict the sentiment of an incoming customer service email or Twitter comment, for example. So when a new email or comment enters the system, we first classify the email as negative (`star_rating` 1), neutral (`star_rating` 3), or positive (`star_rating` 5). This can help us determine the urgency of the response - or help us route the message to the right person as shown in Figure 7-17.

Figure 5-17. Fine-tuning BERT to classify review text into star_rating categories of 1 (worst) through 5 (best)

Create the Training Script

Let's create a training script called `tf_bert_reviews.py` that creates our classifier using TensorFlow and Keras. We will then pass the features generated from the previous chapter into our classifier for model training.

Setup the Train, Validation, and Test Datasets

In the last chapter, we used SageMaker Processing Jobs to transform raw Amazon Customer Reviews into BERT embeddings as shown in Figure 7-18.

Figure 5-18. BERT Embeddings are Inputs to our Model Training

In this section, we load the train, validation, and test datasets to feed into our model for training. We will use TensorFlow's `TFRecordDataset` implementation to load the `TFRecords` in parallel

and shuffle the data to prevent the model from learning the pattern in which the data is presented to the model.

> **NOTE**
>
> In AI and machine learning, randomness is celebrated. Proper shuffling of the training data will provide enough randomness to prevent the model from learning any patterns about how the data is stored on disk and/or presented to the model.

Let's create a helper function to load, parse, and shuffle the `TFRecord`s. You should recognize the `input_ids`, `input_mask`, `segment_ids`, and `label_ids` field names from the previous chapter.

```
def file_based_input_dataset_builder(channel,
                                     input_filenames,
                                     pipe_mode,
                                     is_training,
                                     drop_remainder,
                                     batch_size,
                                     epochs,
                                     steps_per_epoch,
                                     max_seq_length):
    dataset = tf.data.TFRecordDataset(input_filenames)
    dataset = dataset.repeat(epochs * steps_per_epoch * 100)
    dataset = dataset.prefetch(tf.data.experimental.AUTOTUNE)
    name_to_features = {
      "input_ids": tf.io.FixedLenFeature([max_seq_length], tf.int64),
      "input_mask": tf.io.FixedLenFeature([max_seq_length], tf.int64),
      "segment_ids": tf.io.FixedLenFeature([max_seq_length], tf.int64),
      "label_ids": tf.io.FixedLenFeature([], tf.int64),
    }
    def _decode_record(record, name_to_features):
        """Decodes a record to a TensorFlow example."""
        record = tf.io.parse_single_example(record, name_to_features)
        return record
    dataset = dataset.apply(
        tf.data.experimental.map_and_batch(
          lambda record: _decode_record(record, name_to_features),
          batch_size=batch_size,
          drop_remainder=drop_remainder,
          num_parallel_calls=tf.data.experimental.AUTOTUNE))
    dataset.cache()
    if is_training:
        dataset = dataset.shuffle(seed=42,
                                  buffer_size=steps_per_epoch * batch_size,
                                  reshuffle_each_iteration=True)
    return dataset
```

If `is_training` is true, we are in the training phase. During the training phase, we want to shuffle the data between iterations. Otherwise, the model may pick up on patterns about how the data is stored on disk and presented to the model - ie. first all the 5's, then all the 4's, 3's, 2's, 1's, etc. To discourage the model from learning this pattern, we shuffle the data. If `is_training` is false, then we are in either the validation or test phase and we can avoid the shuffle overhead and iterate sequentially.

Let's read in the training, validation, and test datasets using the helper function created above.

```
        # Training Dataset
        train_data_filenames = glob(os.path.join(train_data, '*.tfrecord'))

        train_dataset = file_based_input_dataset_builder(
channel='train',
           input_filenames=train_data_filenames,
                                 pipe_mode=pipe_mode,
is_training=True,
drop_remainder=False,            batch_size=train_batch_size,
epochs=epochs,
steps_per_epoch=train_steps_per_epoch,
max_seq_length=max_seq_length).map(select_data_and_label_from_record)
        # Validation Dataset
validation_data_filenames = glob(os.path.join(validation_data, '*.tfrecord'))
validation_dataset = file_based_input_dataset_builder(
                        channel='validation',
                        input_filenames=validation_data_filenames,
                pipe_mode=pipe_mode,
                        is_training=False,
                        drop_remainder=False,
                        batch_size=validation_batch_size,
                epochs=epochs,
                        steps_per_epoch=validation_steps,
max_seq_length=max_seq_length).map(select_data_and_label_from_record)
```

We will soon pass these train, validation, and test datasets to our model training process. But first, let's set up the custom reviews classifier using TensorFlow, Keras, BERT, and Hugging Face.

Set Up the Custom Classifier Model

Soon, we will feed the `review_body` embeddings and `star_rating` labels into a neural network to fine-tune the BERT model and train the custom review classifier as shown in Figure 7-19.

Figure 5-19. Classify Reviews into Star Rating 1 (Worst) through 5 (Best) Using Our Custom Classifier

For this, we use TensorFlow and Keras to add a neural classifier layer on top of the pre-trained BERT model to learn the `star_rating` (1-5). Reminder that we are using a variant of BERT called DistilBert which requires less memory and compute, but maintains very good accuracy on our dataset.

```
config = DistilBertConfig.from_pretrained('distilbert-base-uncased',
                                          num_labels=len(CLASSES))
model = TFDistilBertForSequenceClassification.from_pretrained('distilbert-base-uncased',
```

```
                                                                    config=config)
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
metric = tf.keras.metrics.SparseCategoricalAccuracy('accuracy')
optimizer = tf.keras.optimizers.Adam(learning_rate=3e-5, epsilon=1e-08)
model.compile(optimizer=optimizer, loss=loss, metrics=[metric])
Model.layers[0].trainable = False
model.summary()
```

NOTE

TIP: It's important to highlight that the `from_pretrained()` function calls will download a large model from the Hugging Face service. You should consider downloading this model to your own S3 bucket and pass the S3 URI to the `from_pretrained()` function calls. This small change will decouple you from the Hugging Face service, remove a potential single-point-of-failure, enable network isolation, and reduce the start time of your model training jobs.

We have chosen not to train the 0th layer, the BERT layer by specifying `trainable=False`. We do this on purpose to keep the underlying BERT model from changing - and focusing only on training our custom classifier. Training the BERT layer will likely improve our accuracy, but the training job will take longer. Since our accuracy is pretty good without training the underlying BERT model, we focus only on training the classifier layer. Here is the output of the model summary showing the breakdown of trainable and non-trainable parameters:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
distilbert (TFDistilBertMain multiple                  66362880
_________________________________________________________________
pre_classifier (Dense)       multiple                  590592
_________________________________________________________________
classifier (Dense)           multiple                  3845
_________________________________________________________________
dropout_19 (Dropout)         multiple                  0
=================================================================
Total params: 66,957,317
Trainable params: 594,437
Non-trainable params: 66,362,880
_________________________________________________________________
```

Train and Validate the Model

At this point, we have prepared our train, validation, and test datasets as input data and defined our custom classifier `model`. Let's pull everything together and invoke the `fit()` function on our model using the `train_dataset` and `validation_dataset`. By passing `validation_dataset`, we are instructing TensorFlow and Keras to perform both training and validation simultaneously.

```
train_and_validation_history = model.fit(train_dataset,
shuffle=True,
                                epochs=5,
                                        ...
                                        validation_data=validation_dataset)
```

We set `shuffle=True` to shuffle our dataset and `epochs=5` to run through our dataset 5 times. The number of `epochs` (pronounced "eh-puhks") is configurable and tunable. We will explore model tuning in the next chapter.

Save the Model

Now, let's save the model in two(2) formats: 1) raw "pre-trained" model in the Keras .h5 file format to be used later for additional training and 2) inference-ready model in the TensorFlow SavedModel format to be used by applications.

```
model.save_pretrained('./transformers/')
model.save('./tensorflow/', save_format='tf')
```

In a bit, we will use the model saved in `./transformers/` with a Transformers component called `InferencePipeline` to perform some ad hoc predictions in our notebook. In a future chapter, we will use the model saved in `./tensorflow/` with TensorFlow Serving to serve our review classification predictions at scale using SageMaker Batch Transformations (offline, batch) and SageMaker Endpoints (online, real-time).

Launch the Script from a SageMaker Notebook

Let's walk through the steps needed to run our training script from a SageMaker Notebook. Later, we will run this same script from an automated pipeline. For now, we run the script from the notebook. First, we will set up the metrics needed to monitor the training job. We'll then configure our algorithm-specific hyper-parameters. Next, we'll select the instance type and number of instances in our cluster. And finally, we will launch our training job.

Define the Metrics to Capture and Monitor

We can create a metric from anything that your training script prints or logs to the console. Let's assume that our TensorFlow model emits the following log lines with the training loss and training accuracy (`loss`, `accuracy`) as well as the validation loss and validation accuracy (`val_loss, val_accuracy`):

```
5000/10000 [>.............................] - loss: 0.1420 - accuracy: 0.800103
6000/10000 [>.............................] - loss: 0.1081 - accuracy: 0.939455
...
10000/10000 [>.............................] - val_loss: 0.1420 - val_accuracy: 0.952193
```

We can define the following four(4) regular expressions to populate four(4) metrics by parsing the values from the log lines:

```
metrics_definitions = [
     {'Name': 'train:loss', 'Regex': 'loss: ([0-9\\.]+)'},
     {'Name': 'train:accuracy', 'Regex': 'accuracy: ([0-9\\.]+)'},
     {'Name': 'validation:loss', 'Regex': 'val_loss: ([0-9\\.]+)'},
     {'Name': 'validation:accuracy', 'Regex': 'val_accuracy: ([0-9\\.]+)'},
]
```

Configure the Hyper-Parameters for Our Algorithm

Every algorithm supports a set of hyper-parameters that alter the algorithm's behavior while learning the dataset. Hyper-parameters can be anything from the depth of a decision tree to the number of layers in your neural network. Hyper-parameter selection involves the usual trade-offs between latency and accuracy. For example, a deeper neural network with lots of layers may provide better accuracy than a shallow neural network, but the deeper network may lead to higher latency during inference as prediction time increases with each layer in the network.

While most hyper-parameters have suitable defaults based on empirical testing, they are highly tunable. In fact, there's an entire subfield within machine learning dedicated to "hyper-parameter tuning" (HPT) or "hyper-parameter optimization" (HPO).

We will dive deep into the art and science hyper-parameter selection and optimization in the next chapter. For now, we set these hyper-parameters based on our experience and intuition - as well as empirical testing with our specific dataset and algorithm.

```
epochs=10
learning_rate=0.00001
epsilon=0.00000001
train_batch_size=128
validation_batch_size=128
train_steps_per_epoch=10000
validation_steps=1000
train_volume_size=1024
use_xla=True
use_amp=True
freeze_bert_layer=True
```

When evaluating an algorithm, you should seek to understand all of the available hyper-parameters. Setting these hyper-parameters to sub-optimal values can make or break a data science project. This is why the subfield of hyper-parameter optimization is so important.

Select Instance Type and Instance Count

The choice of instance type and instance count depends on your workload and budget. Fortunately AWS offers many different instance types including AI/ML-optimized instances with terabytes of RAM and gigabits of network bandwidth. In the cloud, we can easily scale our training job to tens, hundreds, or even thousands of instances with just one line of code.

Let's select 3 instances of the powerful `p3.2xlarge` - each with 8 CPUs, 61GB of CPU RAM, 1 Nvidia Volta V100's GPU processor, and 16GB of GPU RAM. Empirically, we found this combination to perform well with our specific training script and dataset - and within our budget for this task.

```
instance_type='ml.p3.2xlarge'
instance_count=3
```

> **NOTE**
>
> TIP: You can specify `instance_type='local'` to run the script either inside your notebook or on your local laptop. In both cases, your script will execute inside of the same open source SageMaker Docker container that runs in the managed SageMaker service. This lets you test locally before incurring any cloud cost.

To save cost, you should start small and slowly ramp up the compute resources needed for your specific workload. This is commonly called "right sizing your cluster". Also, it's important to choose parallelizable algorithms that benefit from multiple cluster instances. If your algorithm is not parallelizable, you should not add more instances as they will not be used. And adding too many instances may actually slow down your training job by creating too much communication overhead between the instances. Most neural network-based algorithms like BERT are parallelizable and benefit from a distributed cluster.

> **NOTE**
>
> Most, if not all, of the Apache Spark ML algorithms are parallelizable as Spark is natively distributed. Spark ML contributors are encouraged to test their algorithms on a cluster before submitting their code to the Spark ML project.

Putting it All Together in the Notebook

Here is our Jupyter Notebook that sets up and invokes the TensorFlow training job using SageMaker Script Mode:

```
from sagemaker.tensorflow import TensorFlow
estimator = TensorFlow(entry_point='tf_bert_reviews.py',
                       train_instance_count=train_instance_count,
                       train_instance_type=train_instance_type,
                       train_volume_size=train_volume_size,
                       py_version='py3',
                       framework_version='2.1.0',
                       hyperparameters={'epochs': epochs,
                                        'learning_rate': learning_rate,
                                        'epsilon': epsilon,
                                        'train_batch_size': train_batch_size,
                                        'validation_batch_size': validation_batch_size,
                                        'train_steps_per_epoch': train_steps_per_epoch,
                                        'validation_steps': validation_steps,
                                        'use_xla': use_xla,
                                        'use_amp': use_amp,
                                        'max_seq_length': max_seq_length,
                                        'freeze_bert_layer': freeze_bert_layer,
                                        'run_validation': run_validation,
                                        'run_sample_predictions': run_sample_predictions},
                       metric_definitions=metrics_definitions,
                       train_max_run=7200, # max 2 hours * 60 minutes seconds per hour * 60 seconds per
minute

)
```

Lastly, we call the `estimator.fit()` method to start the training job from the notebook as follows:

```
estimator.fit(inputs={'train': s3_input_train_data,
                      'validation': s3_input_validation_data,
                     },
                     wait=False)
```

Evaluate Model Training

After we have trained and validated our model, we can use the remaining holdout dataset - the test dataset - to perform our own predictions and measure the model's performance. Testing the model with the test dataset helps us evaluate how well the model generalizes on unseen data. Therefore, you should never use the holdout test dataset for training or validation. Based on the test results, you may need to modify your algorithm, hyper-parameters, or training data. Adding more training data will almost always improve your model.

Below is the code to evaluate the model using TensorFlow and Keras similar to how we trained and validated the model in the previous section.

```
        test_batch_size = 128
        test_steps = 1000
        test_data_filenames = glob(os.path.join(test_data, '*.tfrecord'))

        test_dataset = file_based_input_dataset_builder(                channel='test',
                        input_filenames=test_data_filenames,
                        pipe_mode=pipe_mode,
                        is_training=False,
                        drop_remainder=False,
                        batch_size=test_batch_size,
                        epochs=epochs,
steps_per_epoch=test_steps,
max_seq_length=max_seq_length).map(select_data_and_label_from_record)
test_history = model.evaluate(test_dataset,
steps=test_steps,
callbacks=callbacks)
        print(test_history)
```

The `test_history` contains the `test_loss` and `test_accuracy` respectively:

```
[0.17315794393, 0.93945542373]
```

Run Some AdHoc Predictions from the Notebook

We can also run some cursory predictions from the notebook to quickly satisfy our curiosity about the model's health. Here's a snippet of relevant code where we load our trained classifier model as a "pre-trained" model and make some predictions with `TextClassificationPipeline` from the Transformers library. `TextClassificationPipeline` is an implementation of a Transformers `InferencePipeline` designed specifically to handle inference for our text classification task. `TextClassificationPipeline` handles the tokenizing to make your inference code simpler and easier to understand.

```
from transformers import TFDistilBertForSequenceClassification
from transformers import TextClassificationPipeline
```

```
      loaded_model = TFDistilBertForSequenceClassification.from_pretrained('./transformers/')
tokenizer =
   DistilBertTokenizer.from_pretrained('distilbert-base-uncased')
inference_pipeline = TextClassificationPipeline(model=loaded_model,
tokenizer=tokenizer,
framework='tf')
inference_pipeline("""I loved it!  I will recommend this to everyone.""")
        inference_pipeline("""It's OK.""")
inference_pipeline("""Really bad.  I hope they don't make this anymore.""")
```

The output shows the predicted label (1-5) as well as the confidence for each prediction (0.92 is 92% confident, for example):

```
[{'label': 5, 'score': 0.92150515}]
[{'label': 3, 'score': 0.2807838}]
[{'label': 1, 'score': 0.48466408}]
```

Confusion Matrix

Let's create a confusion matrix to visually inspect the test results. We start by reading in the holdout test dataset that includes the raw `review_body` text.

```
import csv
df_test_reviews = pd.read_csv('./data/holdout_test.tsv.gz',
        delimiter='\t',
quoting=csv.QUOTE_NONE,
                        compression='gzip')[['review_body', 'star_rating']]
df_test_reviews = df_test_reviews.sample(n=100)
```

Next, we define a `predict` function to call the same `inference_pipeline` function that we created previously.

```
import pandas as pd
def predict(review_body):
    prediction_map = inference_pipeline(review_body)
    return prediction_map[0]['label']
```

Next, we use the `predict` function to calculate the predicted `y_test` dataset. We'll compare this to the observed values, `y_actual` using the following code

```
y_test = df_test_reviews['review_body'].map(predict)
y_actual = df_test_reviews['star_rating']
```

which results in the confusion matrix shown in Figure 7-20.

Figure 5-20. Confusion Matrix Showing the True (Actual) Labels and Predicted Labels

TensorBoard

Since we are using TensorFlow, we can visualize the training and validation metrics using TensorBoard as shown in Figure 7-21 and 7-22.

Figure 5-21. TensorBoard Showing the Loss and Accuracy Over Time

Figure 5-22. TensorBoard Showing the TensorFlow Graph for BERT

CloudWatch

After training, we should evaluate a model's performance using metrics such as accuracy to determine if the model achieves our business objective. In our example, we want to measure if our model correctly predicts the `star_rating` from the `review_body`. In our example, we actually performed training and validation in the same Keras step.

Let’s chart the train and validation metrics using CloudWatch as shown in Figure 7-23. We see that accuracy is going up and to the right while loss is down and to the right - exactly what we expect. We include the GPU utilization metrics for fun in Figure 7-24.

train:accuracy

train:loss

Figure 5-23. Dashboard of Training and Validation Metrics

Figure 5-24. Dashboard of GPU System Metrics (300% is 3 GPUs @ 100%)

Debug Model Training with SageMaker Debugger

During training, we can use SageMaker Debugger to provide full insight into model training by monitoring, recording, and analyzing the state of each model training job - without any code changes. With Amazon SageMaker Debugger, we can interactively and visually explore the data captured during training - even across the workers in a distributed training cluster.

Combined with CloudWatch Events, Debugger can trigger an alert if a particular rule condition is met such as bad training data, vanishing gradients, and exploding gradients. Bad data includes NaNs and nulls. Vanishing gradients occur when really small values are multiplied by other

really small values - and the result is too small for our float data type to store. Exploding gradients are the opposite of vanishing gradients. They occur when really large values are multiplied by other really large values - and the result can't be represented by the 32 bits of our float data type. Both vanishing and exploding gradients can occur in deep neural networks as shown in Figure 7-25.

Figure 5-25. Vanishing gradients occur when small numbers are multiplied by other small numbers like in a neural network

If an alert is triggered at 3am, we can page the on-call data scientist instead of the DevOps team. The data scientist can then use Debugger to analyze the training run and determine the root cause of the job failure. Meanwhile, the on-call DevOps person sleeps peacefully through the night. Figure 7-26 shows how SageMaker Debugger analyzes and monitors your training job.

Figure 5-26. Analyzing and Monitoring a Training Job with SageMaker Debugger

We capture debugger data by adding rules to our training job definition. Out of the box, Debugger supports many rules for common debugging scenarios such as overfitting, vanishing/exploding tensors, and class imbalances. SageMaker launches one evaluation job for each SageMaker rule specified. Below is the code to create two rules to detect: 1) when the loss stops decreasing and 2) when we are overtraining. These are both signals to consider "early stopping" the training job and reduce the cost of training.

```
rules=[
        Rule.sagemaker(
            rule_configs.loss_not_decreasing(),
            rule_parameters={
                'collection_names': 'losses,metrics',
                'use_losses_collection': 'true',
                'num_steps': '5',
                'diff_percent': '5'
            },
            collections_to_save=[
                CollectionConfig(name='losses',
                                 parameters={
                                     'save_interval': '100',
                                 }),
                CollectionConfig(name='metrics',
                                 parameters={
                                     'save_interval': '100',
                                 })
            ]
        ),
        Rule.sagemaker(
            rule_configs.overtraining(),
            rule_parameters={
                'collection_names': 'losses,metrics',
                'patience_train': '5',
                'patience_validation': '10',
                'delta': '0.1'
            },
            collections_to_save=[
                CollectionConfig(name='losses',
                                 parameters={
                                     'save_interval': '100',
                                 }),
                CollectionConfig(name='metrics',
                                 parameters={
                                     'save_interval': '100',
                                 })
            ]
        )
    ]
```

We also need to create a Debugger Hook to use with TensorFlow and Keras as follows:

```
hook_config = DebuggerHookConfig(
    hook_parameters={
        'save_interval': '10', # number of steps
        'export_tensorboard': 'true',
        'tensorboard_dir': 'hook_tensorboard/',
    })
```

Then we need to set the rules and debugger hook in our Estimator as follows:

```
from sagemaker.tensorflow import TensorFlow
estimator = TensorFlow(entry_point='tf_bert_reviews.py',
                                 ...
                       rules=rules,
                       debugger_hook_config=hook_config,
                                 ...
)
```

Here is the rule summary from our training run:

```
estimator.latest_training_job.rule_job_summary()
```

Here is the output:

```
[{'RuleConfigurationName': 'LossNotDecreasing',
  'RuleEvaluationJobArn': 'arn:aws:sagemaker:us-east-1:835319576252:processing-job/tensorflow-training-
2020-0-lossnotdecreasing-cad4baba',
  'RuleEvaluationStatus': 'NoIssuesFound',
  'LastModifiedTime': datetime.datetime(2020, 5, 12, 16, 17, 14, 545000, tzinfo=tzlocal())},
 {'RuleConfigurationName': 'Overtraining',
  'RuleEvaluationJobArn': 'arn:aws:sagemaker:us-east-1:835319576252:processing-job/tensorflow-training-
2020-0-overtraining-74d9dac1',
  'RuleEvaluationStatus': 'IssuesFound',
  'StatusDetails': 'RuleEvaluationConditionMet: Evaluation of the rule Overtraining at step 1100
resulted in the condition being met\n',
  'LastModifiedTime': datetime.datetime(2020, 5, 12, 16, 17, 14, 545000, tzinfo=tzlocal())}]
```

Looks like we are overtraining at step 1100.

> **NOTE**
>
> TIP: You can also create custom rules by bringing your own container (BYOC).

Let's inspect some of our tensors using the SageMaker Debugger API.

```
from smdebug.trials import create_trial
trial = create_trial(estimator.latest_job_debugger_artifacts_path())
for i in trial.tensor_names():
    print(i)
```

Here are some of the tensors tracked by SageMaker Debugger:

```
accuracy
batch
loss
size
val_accuracy
val_loss
```

Let's plot the `accuracy` during the beginning part of our training job using pandas and matplotlib as shown in Figure 7-27.

```
     steps  accuracy
0     2000  0.450938
1     3000  0.527415
2     4000  0.760599
3     5000  0.800103
4     6000  0.939455
```

Figure 5-27. Accuracy vs. Number of Steps

Increase Performance and Reduce Costs

In this section, we provide tips on how to improve performance and reduce costs by using hardware and infrastructure level optimizations such as reduced precision and spot instances. Additionally, we describe how you can stop your training jobs early when they stop improving.

Reduced 16-bit Half Precision

Most models store parameters and perform calculations using full 32-bit numerical precision. Intuitively, if we reduce the precision to 16-bit or “reduced” or “half” precision, we would not only reduce the footprint of the stored parameters by half, but also increases computation performance by 2x as the chip can perform two(2) 16-bit calculations on the same 32-bit hardware. This speedup is so noticeable and desirable that modern hardware - including the Nvidia Volta V100 GPU - provides heavily-optimized 16-bit cores built right into the silicon. Figure 7-28 shows the difference between 32-bit floats and 16-bit floats in terms of the number of bits used to capture exponent and fraction portions of each float.

Figure 5-28. 32-bit Floats vs. 16-bit Floats

There are downsides to reduced precision, however. While the training times go towards zero in this perfect world, so can accuracy and numeric instability. By reducing the numerical precision to 16 bits, our model may not learn as well as a 32-bit model. Additionally, you may experience more frequent vanishing gradients as the model only has 16 bits to represent the parameters and gradients. So the chance of the value going to zero(0) is much higher than using 32 bits.

> **NOTE**
>
> If you plan to deploy your model to tiny devices with limited memory, you may need to reduce precision to 8-bit, 4-bit, or even 1-bit for your floats. The challenge is preserving accuracy at this lower precision.

Mixed 32-bit Full and 16-bit Half Precision

The choice of 32-bit or 16-bit is yet another hyper-parameter to optimize. Some algorithms and datasets may be more sensitive to reduced precision than others. However, there is a middle ground called “mixed” precision that stores the parameters in 32-bits to maintain numerical stability, but performs the calculations using 16-bit operations.

TensorFlow and Keras offer native mixed precision support at the network-layer level. Here, we set the global policy for all layers using a mixed precision “policy”.

```
import tf.keras.mixed_precision.Policy
policy = mixed_precision.Policy('mixed_float16')
mixed_precision.set_policy(policy)
```

This is effectively a “turbo-button” for your model training, however, you should treat this like any other hyper-parameter and tune it for your specific dataset and algorithm.

Quantization

In the future chapter on model deployment, we will describe how to reduce the precision of a model from 32-bit to 16-bit *after training* to reduce the size of the model and speed up the computations. The quantization process uses statistical methods - rooted in audio signal processing - to preserve the dynamic range of the parameter values. While not required, you can modify your training script to be “quantization-aware” and prepare the model for the post-training quantization. This helps to preserve model accuracy after quantizing.

Spot Instances and Checkpoints

If you are using an algorithm that supports checkpointing such as TensorFlow, you can use spot instances with SageMaker training to save cost. Spot instances are much cheaper than on-demand instances. However, spot instances may be replaced by on-demand instances while the training job is running. Any number of spot instances may be replaced at any time, but our checkpoints will allow us to continue training - even if all instances are replaced simultaneously. While Figure 7-29 shows an example of one(1) spot instance replaced at time 0, and 3 spot instances replaced at time 1. However, the replacement cadence is driven by supply-demand of instances - and somewhat difficult to predict.

Figure 5-29. Continuing to Train When Spot Instances are Replaced with On-Demand Instances

The new on-demand instances use the latest checkpoint and pick up where the spot instances left off using the following checkpointing code:

```
import uuid
checkpoint_s3_prefix = 'checkpoints/{}'.format(str(uuid.uuid4()))
checkpoint_s3_uri = 's3://{}/{}/'.format(bucket,
   checkpoint_s3_prefix)
estimator = TensorFlow(entry_point='tf_bert_reviews.py',
                       source_dir='src'
```

```
 ...
                      checkpoint_s3_uri=checkpoint_s3_uri,
```

...

Our script then leverages the provided checkpoint location to save a checkpoint using Keras `ModelCheckpoint` as follows:

```
checkpoint_path = '/opt/ml/checkpoints'
checkpoint_callback = ModelCheckpoint(
            filepath=os.path.join(checkpoint_path, 'tf_model_{epoch:05d}.h5'),
            save_weights_only=False,
            verbose=1,
            monitor='val_accuracy')
callbacks.append(checkpoint_callback)
```

To load the model, our script uses the checkpoint location to load the model as follows

```
def load_checkpoint_model(checkpoint_path):
    import glob
    import os

    glob_pattern = os.path.join(checkpoint_path, '*.h5')
    print('glob pattern {}'.format(glob_pattern))
    list_of_checkpoint_files = glob.glob(glob_pattern)
    print('List of checkpoint files {}'.format(list_of_checkpoint_files))

    latest_checkpoint_file = max(list_of_checkpoint_files)
    print('Latest checkpoint file {}'.format(latest_checkpoint_file))

    loaded_model = TFDistilBertForSequenceClassification.from_pretrained(
                      latest_checkpoint_file,
 config=config)
if os.listdir(checkpoint_path):
    model = load_checkpoint_model(checkpoint_path)
```

Early Stopping

Early stopping is a powerful mechanism to reduce cost when a training job stops improving on the objective (ie. maximize accuracy) and will no longer benefit from additional training. The key, of course, is to accurately measure the objective such as accuracy and clearly define the stopping rule. Often, we define the stopping rule in terms of relative change over a given time or number of steps. For example, if our accuracy does not improve by 1% over 1000 steps, we may want to stop the training job and save some money.

When the rule fires, we can invoke a number of different targets. For this example, let's create a Lambda function to stop the training job. Here, the lambda event contains `TrainingJobName`, the name of the training job to stop.

```
import json
import boto3
import logging
def lambda_handler(event, context):
    training_job_name = event.get("detail").get("TrainingJobName")
    eval_statuses = event.get("detail").get("DebugRuleEvaluationStatuses", None)
    if eval_statuses is None or len(eval_statuses) == 0:
        logging.info("Couldn't find any debug rule statuses, skipping...")
```

```
        return {
            'statusCode': 200,
            'body': json.dumps('Nothing to do')
        }
    client = boto3.client('sagemaker')
    for status in eval_statuses:
        if status.get("RuleEvaluationStatus") == "IssuesFound":
            logging.info(
                'Evaluation of rule configuration {} resulted in "IssuesFound". '
                'Attempting to stop training job {}'.format(
                    status.get("RuleConfigurationName"), training_job_name
                )
            )
            try:
                client.stop_training_job(
                    TrainingJobName=training_job_name
                )
            except Exception as e:
                logging.error(
                    "Encountered error while trying to "
                    "stop training job {}: {}".format(
                        training_job_name, str(e)
                    )
                )
                raise e
    return None
```

Summary

Congratulations! You trained your first SageMaker model using TensorFlow, Keras, BERT, and Amazon SageMaker!!

In this chapter, we dove deep into the SageMaker infrastructure, model development SDKs, and SageMaker training jobs. We trained a model using SageMaker, described the security best practices, and explored some cost-saving and performance-improvement tips.

We also learned how BERT's Transformer neural-network architecture has revolutionized the field of natural language processing (NLP) and natural language understanding (NLU) by using a bidirectional approach to learn a contextual representation of the words in a corpus. We demonstrated how to fine tune a pre-trained BERT model to build a domain-specific text classifier. This is in contrast to the previous generation of NLP models such as word2vec, GloVe, and ELMo that either 1) learn only in one direction at a time, 2) throw away the original model and preserve only the learned embeddings, or 3) use complex recurrent neural network (RNNs) architectures that require a large amount of memory and compute.

In the next chapter we will re-train our model using different configurations and hyper-parameters in a process called hyper-parameter optimization (HPO) or hyper-parameter tuning (HPT). Through this process, we will find the best model and hyper-parameter combination that provides the highest accuracy. We will further optimize the best model even further to take advantage of hardware optimizations provided by our target deployment hardware such as Nvidia GPUs and Amazon Inferentia chips. In subsequent chapters, we will optimize, deploy, and test our model in production.

Chapter 6. Training and Optimizing Models at Scale

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 8th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *support@pipeline.ai*.

In the previous chapter, we trained a single model - with a single set of hyper-parameters - using Amazon SageMaker. We also demonstrated how to fine tune a pre-trained BERT model to build a NLP domain-specific text classifier.

In this chapter, we will use SageMaker Experiments to measure, track, compare, and improve our models at scale. We also use SageMaker Hyper-Parameter Optimizer (HPO) to choose the best hyper-parameters for our specific algorithm and dataset. We show how to perform distributed training using various communication strategies and distributed file systems. We finish with tips on how to reduce cost and increase performance with refined hyper-parameter selection features of Autopilot, optimized data loading features of SageMaker, and enhanced networking features of AWS.

NOTE

Peter Drucker, one of Jeff Bezos' favorite business strategists, once said, "If you can't measure it, you can't improve it." This quote captures the essence of this chapter which focuses on measuring and optimizing our predictive models.

Compare Training Runs with SageMaker Experiments

In the previous chapter, we trained a BERT-based, natural-language classifier model using a fixed set of hyper-parameters. In this section, we provide ranges of hyper-parameters to SageMaker to create a Hyper-Parameter Tuning Job and find the best configuration for our model and dataset.

Amazon Autopilot, presented in the Automated Machine Learning chapter, automatically chooses the hyper-parameter ranges, spawns the SageMaker Hyper-Parameter Tuning Jobs, tracks the results with SageMaker Experiments, and chooses the best candidates. While the Automated Machine Learning chapter is relevant for many uses cases, this chapter is meant for those of us who want a little more control over the hyper-parameter optimization process - at the expense of adding a bit more complexity. Either way, SageMaker Experiments is a valuable tool in our data science toolkit by giving us deep insight into the model training and tuning process.

SageMaker Experiments helps you track, organize, visualize, and compare your AI and machine learning models across feature engineering, model training, model tuning, and model deploying. Experiments are seamlessly integrated with SageMaker Studio as shown in Figure 6-1.

Figure 6-1. : Amazon SageMaker Studio compares the loss across multiple training runs within an experiment

Trace and Audit Model Lineage

Humans are naturally curious. When presented with an object, people will likely want to know how that object was created. Now consider an object as powerful and mysterious as a predictive model learned by a machine. We naturally want to know how this model was created. Which dataset was used? Which hyper-parameters were chosen? Which other hyper-parameters were explored? How does this version of the model compare to the previous version? All of these questions can be answered by SageMaker Experiments.

The SageMaker Experiments feature brings traceability, lineage tracking, and auditability to your AI and machine learning workflows. Users can quickly traceback the entire model lineage using the Experiments API. In addition, all SageMaker and S3 API calls are logged as private CloudTrail and CloudWatch logs for fine-grained and secure auditing.

Reproduce a Model

Reproducibility is critical to understanding how a model was trained including the data sets, algorithm, and hyper-parameters involved. Along with traceability and auditability, SageMaker Experiments provide full model reproducibility by allowing practitioners to restore everything including the train/validation/test datasets, algorithm, hyper-parameters, and trained model.

However, remember that AI and machine learning models are stochastic vs. deterministic and therefore difficult to reproduce model results exactly. Still, reproducibility is critical to understanding how a model was trained including the data sets, algorithm, and hyper-parameters involved.

Manage Artifacts and Dependencies

We can use the `log_artifact()` Experiment API to save, version, and restore all model artifacts in either S3 or Git. Users should enable S3 versioning and object locking to maintain artifacts for reproducibility and auditability.

Dependencies such as Python libraries and Docker images should always be retrieved from private, internal artifact repositories such as AWS CodeCommit and Elastic Container Registry (ECR) to reduce security risks and increase performance. Any dependency pulled from the Internet may slow down the training job and potentially introduce security risks. If a dependency is not available due to a temporary outage or throttling event with external dependency providers like PyPi, Maven, GitHub, or DockerHub, your training cluster will fail.

Track Our Model Lifecycle with the Experiments API

The SageMaker Experiments API is made up of a few key abstractions as follows:

Experiment

A collection of related Trials. Add Trials to an Experiment that you wish to compare together.

Trial

A description of a multi-step machine learning workflow. Each step in the workflow is described by a Trial Component.

Trial Component

A description of a single step in a machine learning workflow. For example data cleaning, feature extraction, model training, model evaluation, etc.

Tracker

A logger of information about a single TrialComponent.

You can use SageMaker Experiments from any Jupyter notebook or Python script by using the SageMaker Experiments Python SDK and adding a few lines of code to setup and track the experiment.

Let's say we want to improve upon our model from the training run in the previous chapter. We will call this Model A and place it into Trial A within Experiment A as shown in Figure 6-2. Next, we create two(2) additional trials, Trials B and C, which use different sets of hyper-parameters to train two(2) separate models, Models B and C, respectively.

Figure 6-2. : Compare training runs with different hyper-parameters using SageMaker Experiments

Using the SageMaker Experiments API, we create a complete record of every step and hyper-parameter used to re-create Models A, B, and C. Later, we will extend this lineage to include model deployment.

At any given point in time, we should be able to trace a model back to its origins including the exact dataset used to train the model. This traceability is essential for auditing, explaining, and improving your models.

Set Up the Experiment

Let's modify our training script to use the SageMaker Experiment API to track the lineage of our BERT-based review classifier model that we built in the previous chapter. Let's first create our `Experiment`.

```
import time
from smexperiments.experiment import Experiment
timestamp = '{}'.format(int(time.time()))
experiment = Experiment.create(
                experiment_name='Amazon-Customer-Reviews-BERT-Experiment-{}'.format(timestamp),
                description='Amazon Customer Reviews BERT Experiment',
                sagemaker_boto_client=sm)
experiment_name = experiment.experiment_name
print('Experiment name: {}'.format(experiment_name))
```

Next, we add a `Trial` to the experiment and `Tracker` the feature engineering configuration including the S3 locations of the raw_input and train/validation/test datasets - as well as the BERT `max_seq_length` used during tokenization.

```
import time
from smexperiments.trial import Trial
timestamp = '{}'.format(int(time.time()))
trial = Trial.create(trial_name='trial-{}'.format(timestamp),
                     experiment_name=experiment_name,
                     sagemaker_boto_client=sm)
trial_name = trial.trial_name
print('Trial name: {}'.format(trial_name))
from smexperiments.tracker import Tracker
tracker_prepare = Tracker.create(display_name='prepare',
                                 sagemaker_boto_client=sm)
```

```
prepare_trial_component_name = tracker_prepare.trial_component.trial_component_name
print('Prepare trial component name {}'.format(prepare_trial_component_name))
trial.add_trial_component(tracker_prepare.trial_component)
tracker_prepare.log_input(name='raw_data_s3_uri',
                          media_type='s3/uri',
                          value=s3_raw_input_data)
tracker_prepare.log_parameters({
    'max_seq_length': max_seq_length,
    'train_split_percentage': train_split_percentage,
    'validation_split_percentage': validation_split_percentage,
    'test_split_percentage': test_split_percentage,
})
tracker_prepare.log_output(name='train_data_s3_uri',
                           media_type='s3/uri',
                           value=processed_train_data_s3_uri)
tracker_prepare.log_output(name='validation_data_s3_uri',
                           media_type='s3/uri',
                           value=processed_validation_data_s3_uri)
tracker_prepare.log_output(name='test_data_s3_uri',
                           media_type='s3/uri',
                           value=processed_test_data_s3_uri)
tracker_prepare.trial_component.save()
```

Next, let's configure the `experiment_config` parameter that we will pass to the `TensorFlow.fit()` function when we train our model. This `experiment_confi` is used by SageMaker to update the new `TrialComponen` and `Tracker` created during training called `train`.

```
experiment_config = {
    'ExperimentName': experiment_name,
    'TrialName': trial.trial_name,
    'TrialComponentDisplayName': 'train'
}
```

Here is the modified call to the `TensorFlow.fit()` function to track the `train` trial:

```
estimator.fit(inputs={'train': s3_input_train_data,
                              'validation': s3_input_validation_data,
                        'test': s3_input_test_data
                        },
                    experiment_config=experiment_config,
                    wait=False)
```

Here is how we compare the experiment trials:

```
from sagemaker.analytics import ExperimentAnalytics
lineage_table = ExperimentAnalytics(
    sagemaker_session=sess,
    experiment_name=experiment_name,
    metric_names=['validation:accuracy'],
    sort_by="CreationTime",
    sort_order="Ascending",
)
lineage_df = lineage_table.dataframe()
lineage_df
```

Below are the values logged by our `prepare` tracker as well as the `train` tracker created by SageMaker during training.

TrialComponentName	DisplayName	max_seq_length	learning_rate	...
TrialComponent-2020-06-13-062410-pxuy	prepare	128.0	NaN	...
tensorflow-training-2020-06-13-06-24-12-989	train	128.0	0.00001	...

Automatically Find the Best Model Hyper-Parameters

Now that we understand how to track and compare model training runs, we can automatically find the best hyper-parameters at scale using a process called Hyper-Parameter Tuning (HPT) or Hyper-Parameter Optimization (HPO). We have already learned that hyper-parameters are the settings which control how our machine learning algorithm learns our dataset to create the best predictive model possible.

We need to find the set of hyper-parameters that meet or exceed our predefined model objective metric such as accuracy. After each hyper-parameter tuning run, we evaluate the model performance and adjust the hyper-parameters until we reach our goal. Doing this manually is very time-consuming as model tuning often requires training hundreds of models. Luckily, we can leverage SageMaker's hyper-parameter tuning feature to speed up and scale out this process as shown in Figure 6-3.

Figure 6-3. SageMaker Hyper-Parameter Optimization including common tuning strategies.

Common hyper-parameter optimization strategies include the following:

Random search

With random search, we randomly keep picking combinations of hyper-parameters until we find a well performing combination. This approach is very fast, but we might miss the best set of hyper-parameters as we are picking them randomly every time.

Grid search

With grid search, we create a grid of hyper-parameters we want to explore and evaluate the model performance against each combination. This approach is more accurate, but takes more time. We can optimize grid search by applying it to a range of parameters that performed well in random search.

Bayesian optimization

With Bayesian optimization, we treat the task as a regression problem. Similar to how our actual model learns the model weights that minimize a loss function, Bayesian optimization iterates to find the best hyper-parameters using a surrogate model and acquisition function. Bayesian optimization is usually more efficient than manual, random or grid search.

Let's use SageMaker hyper-parameter optimizer to find the best hyper-parameters for our BERT-based review classifier from the previous chapter.

Set Up the Hyper-Parameter Ranges

First, let's create an `optimize` tracker and associate it with our experiment.

```
from smexperiments.tracker import Tracker
tracker_optimize = Tracker.create(display_name='optimize-1',
                                  sagemaker_boto_client=sm)
optimize_trial_component_name =
tracker_optimize.trial_component.trial_component_name
trial.add_trial_component(tracker_optimize.trial_component)
```

To keep this example simple and avoid a combinatorial explosion of trial runs, we will freeze most hyper-parameters and explore only a limited set for this particular optimization run. In a perfect world with unlimited resources and budget, we would explore every combination of hyper-parameters. For now, we will freeze the following:

```
epochs=1
epsilon=0.00000001
train_batch_size=128
validation_batch_size=128
test_batch_size=128
train_steps_per_epoch=50
validation_steps=50
test_steps=50
use_xla=True
use_amp=True
freeze_bert_layer=True
```

Next, let's set up the hyper-parameter ranges that we wish to explore. We are choosing these hyper-parameters based on intuition, domain knowledge, and algorithm documentation. You may also find research papers useful - or other prior work from others in the community. At this point in the life cycle of machine learning and predictive analytics, you can almost-always find relevant information on the problem you are trying to solve.

If you still can't find a suitable starting point, you should explore ranges logarithmically (vs. linearly) to help gain a sense of the scale of the hyper-parameter. There is no point in exploring the set [1, 2, 3, 4] if your best hyper-parameter is orders of magnitude away in the 1,000's, for example.

SageMaker Autopilot is another way to determine a baseline set of hyper-parameters for your algorithm and dataset. Autopilot's hyper-parameter selection process has been refined on many thousands of hours of training jobs across a wide range of data sets, algorithms, and use cases within Amazon.

SageMaker hyper-parameter tuning supports three types of parameter ranges:

Categorical(list)

Hyper-Parameters that have a discrete list of possible values.

`ContinuousParameter(min, max)`: Hyper-Parameters that have a continuous range of possible values.

IntegerParameter (min, max)

Hyper-Parameters that have an integer range of possible values.

```
Here we are specifying defining learning_rate as a ContinuousParameter:
from sagemaker.tuner import IntegerParameter
from sagemaker.tuner import ContinuousParameter
from sagemaker.tuner import CategoricalParameter
from sagemaker.tuner import HyperparameterTuner

hyperparameter_ranges = {
    'learning_rate': ContinuousParameter(0.00001, 0.00005, scaling_type='Linear'),
}
```

You can also specify the scaling type for each type of hyper-parameter ranges. The scaling type can be set to Linear, Logarithmic, ReverseLogarithmic, or Auto to allow SageMaker to decide.

Finally, we need to define the objective metric that the hyper-parameter tuning job is trying to optimize - in our case, validation accuracy. Remember that we need to provide the regular expression (regex) to extract the metric from the SageMaker container logs. We chose to also collect the training loss, training accuracy, and validation loss for informational purposes.

```
objective_metric_name = 'validation:accuracy'
metrics_definitions = [
     {'Name': 'train:loss', 'Regex': 'loss: ([0-9\\.]+)'},
     {'Name': 'train:accuracy', 'Regex': 'accuracy: ([0-9\\.]+)'},
     {'Name': 'validation:loss', 'Regex': 'val_loss: ([0-9\\.]+)'},
     {'Name': 'validation:accuracy', 'Regex': 'val_accuracy: ([0-9\\.]+)'}]
```

Run the Hyper-Parameter Tuning Job

We start by creating our TensorFlow estimator similar to the previous chapter. Note that we are not specifying the `learning_rate` hyper-parameter in this case. We will pass this as a hyper-parameter range to the `HyperparameterTuner` in a moment.

```
from sagemaker.tensorflow import TensorFlow
estimator = TensorFlow(entry_point='tf_bert_reviews.py',
                       source_dir='src',
                       role=role,
                       train_instance_count=train_instance_count,
                       train_instance_type=train_instance_type,
                       py_version='py3',
                       framework_version='2.1.0',
                       hyperparameters={'epochs': epochs,
                                        'epsilon': epsilon,
                                        'train_batch_size': train_batch_size,
                                        'validation_batch_size': validation_batch_size,
                                        'test_batch_size': test_batch_size,
                                        'train_steps_per_epoch': train_steps_per_epoch,
                                        'validation_steps': validation_steps,
                                        'test_steps': test_steps,
                                        'use_xla': use_xla,
                                        'use_amp': use_amp,
                                        'max_seq_length': max_seq_length,
                                        'freeze_bert_layer': freeze_bert_layer,
                       metric_definitions=metrics_definitions,
                      )
```

Next, we can create our Hyper-Parameter Tuning Job by passing the TensorFlow estimator, hyper-parameter range, objective metric, tuning strategy, number of jobs to run in parallel/total, as well as an early stopping strategy. SageMaker will use the given optimization strategy (ie. “Bayesian” or “Random Search”) to explore the values within the given ranges.

```
objective_metric_name = 'validation:accuracy'
tuner = HyperparameterTuner(
    estimator=estimator,
    objective_type='Maximize',
    objective_metric_name=objective_metric_name,
    hyperparameter_ranges=hyperparameter_ranges,
    metric_definitions=metrics_definitions,
    max_jobs=100,
    max_parallel_jobs=10,
    strategy='Bayesian',
    early_stopping_type='Auto'
)
```

In this example, we are using the `Bayesian` optimization strategy and with 10 jobs in parallel and 100 total. By only doing 10 at a time, we give the `Bayesian` strategy a chance to learn from previous runs. In other words, if we did all 100 in parallel, the Bayesian strategy could not use prior information to choose better values within the ranges provided.

```
By setting early_stopping_type to 'Auto', SageMaker will stop the tuning job if the tuning job is not going to improve
upon the objective metric.  This helps save time and reduce the overall cost of your tuning job.
Let’s start the tuning job by calling tuner.fit() and pointing to our training data input channels:
tuner.fit(inputs={'train': s3_input_train_data,
                  'validation': s3_input_validation_data,
                  'test': s3_input_test_data
          },
          include_cls_metadata=False)
```

Using our experiment tracker, let’s update the experiment lineage to include the best hyper-parameters and objective metrics found by our hyper-parameter tuning job.

```
tracker_optimize.log_parameters({
'learning_rate': best_learning_rate
})
tracker_optimize.log_metrics({
'validation:accuracy': best_validation_accuracy
})
tracker_optimize.trial_component.save()
```

Analyze the Tuning Job Results

Once the hyper-parameter tuning job has finished, we can analyze the results directly in our notebook or through SageMaker Studio.

```
from sagemaker.analytics import ExperimentAnalytics
lineage_table = ExperimentAnalytics(
    sagemaker_session=sess,
    experiment_name=experiment_name,
    metric_names=['validation:accuracy'],
    sort_by="CreationTime",
    sort_order="Ascending",
)
lineage_df = lineage_table.dataframe()
lineage_df
```

The output should look similar to this:

TrialComponentName	DisplayName	max_seq_length	learning_rate	validation_accuracy	...
TrialComponent-2020-06-13-062410-pxuy	prepare	128.0	NaN	NaN	...
tensorflow-training-2020-06-13-06-24-12-989	train	128.0	0.00001	0.7420	...
TrialComponent-2020-06-12-193933-bowu	optimize-1	128.0	0.000017	0.7824	...

In this case, the `learning_rate` value of 0.000017 gives the highest validation accuracy of 0.7824.

We can also view details of the best candidate directly in the SageMaker UI as shown in Figure 6-4.

Figure 6-4. Details of the best candidate.

The CloudWatch system and algorithms metrics for one of the tuning jobs is shown in Figure 6-5.

Figure 6-5. CloudWatch System and Algorithm Performance Metrics for a Tuning Job

We can also view the complete list of all parallel tuning jobs as shown in Figure 6-6. Note that some completed successfully and some were stopped early to save time and reduce cost.

Figure 6-6. List of Tuning Jobs.

In this example, we have optimized the hyper-parameters of our TensorFlow BERT classifier layer. SageMaker hyper-parameter tuning also supports automatic hyper-parameter tuning across multiple algorithms by adding a list of algorithms to the tuning job definition. You can specify different hyper-parameters and ranges for each algorithm. SageMaker Autopilot automatically uses multi-algorithm tuning to find the best model across many different algorithms based on your dataset and objective function.

Warm Start Additional Hyper-Parameter Tuning Jobs

Once we have our best candidate, we can choose to perform yet another round of hyper-parameter optimization using a technique called “warm start”. Warm starting re-uses the prior results from a previous hyper-parameter tuning job - or set of jobs - to speed up the optimization process and reduce overall cost. Warm start creates a many-to-many parent-child relationship. In our example, we perform a warm start with a single parent, the previous tuning job, as shown in Figure 6-7.

Figure 6-7. Warm Start an Additional Hyper-Parameter Tuning Job from a Previous Tuning Job

Warm start is particularly useful when you want to change the hyper-parameter tuning ranges from the previous job or add new hyper-parameters. Both scenarios benefit from the knowledge of the previous tuning job to find the best model faster. The two scenarios are implemented with two warm start types, IDENTICAL_DATA_AND_ALGORITHM and TRANSFER_LEARNING.

If you choose IDENTICAL_DATA_AND_ALGORITHM, the new tuning job uses the same input data and training image as the parent job. You are allowed to update the hyper-parameter tuning ranges and the maximum number of training jobs. You can also add previously static hyper-parameters to the list of dynamic, tunable hyper-parameters and vice versa - as long as the overall number of static plus tunable hyper-parameters remains the same. Upon completion, a tuning job with this strategy will return an additional field, OverallBestTrainingJob containing the best model candidate including this tuning job as well as the completed parent tuning jobs. If you choose TRANSFER_LEARNING, you can use updated training data, and a different version of the training algorithm. Perhaps you collected more training data since the last optimization run - and now you want to re-run the tuning job with the updated dataset. Or perhaps a newer version of the algorithm has been released and you want to re-run the optimization process.

Run Hyper-Parameter Tuning Job using Warm Start

We need to configure the tuning job with the WarmStartConfig using one or more of the previous hyper-parameter tuning jobs as parents. The parent HPT jobs must have finished either with one of the following success or failure states: Completed, Stopped, or Failed. Recursive parent-child relationships are not supported. We also need to specify the WarmStartType. In our example, we will use the IDENTICAL_DATA_AND_ALGORITHM as we plan to only modify the hyper-parameter ranges and not use an updated dataset or algorithm version.

Let’s start with the setup of the WarmStartConfig:

```
from sagemaker.tuner import WarmStartConfig
from sagemaker.tuner import WarmStartTypes
warm_start_config =
WarmStartConfig(warm_start_type=WarmStartTypes.IDENTICAL_DATA_AND_ALGORITHM,
 parents={tuning_job_name})
```

Let's define the static hyper-parameters that we are not planning to explore.

```
epochs=1
epsilon=0.00000001
train_batch_size=128
validation_batch_size=128
test_batch_size=128
train_steps_per_epoch=50
validation_steps=50
test_steps=50
use_xla=True
use_amp=True
freeze_bert_layer=True
from sagemaker.tensorflow import TensorFlow
estimator = TensorFlow(entry_point='tf_bert_reviews.py',
                       source_dir='src',
                       role=role,
                       train_instance_count=train_instance_count,
                       train_instance_type=train_instance_type,
                       train_volume_size=train_volume_size,
                       py_version='py3',
                       framework_version='2.1.0',
                       hyperparameters={'epochs': epochs,
                                        'epsilon': epsilon,
                                        'train_batch_size': train_batch_size,
                                        'validation_batch_size': validation_batch_size,
                                        'test_batch_size': test_batch_size,
                                        'train_steps_per_epoch': train_steps_per_epoch,
                                        'validation_steps': validation_steps,
                                        'test_steps': test_steps,
                                        'use_xla': use_xla,
                                        'use_amp': use_amp,
                                        'max_seq_length': max_seq_length,
                                        'freeze_bert_layer': freeze_bert_layer

}

                       metric_definitions=metrics_definitions,
                      )
```

While we can choose to explore more hyper-parameters in this warm start tuning job, we will simply modify the range of our `learning_rate` to narrow in on the best value found in the parent tuning job.

```
from sagemaker.tuner import IntegerParameter
from sagemaker.tuner import ContinuousParameter
from sagemaker.tuner import CategoricalParameter

hyperparameter_ranges = {
    'learning_rate': ContinuousParameter(0.00015, 0.00020,
                                                          scaling_type='Linear')}
```

Now let's define the objective metric, create the `HyperparameterTuner` with the `warm_start_config` from above, and start the tuning job.

```
objective_metric_name = 'validation:accuracy'
tuner = HyperparameterTuner(
    estimator=estimator,
    objective_type='Maximize',
    objective_metric_name=objective_metric_name,
    hyperparameter_ranges=hyperparameter_ranges,
    metric_definitions=metrics_definitions,
    max_jobs=2,
    max_parallel_jobs=1,
    strategy='Bayesian',
    early_stopping_type='Auto',
    warm_start_config=warm_start_config
```

```
)
tuner.fit({'train': s3_input_train_data,
           'validation': s3_input_validation_data,
           'test': s3_input_test_data},
          include_cls_metadata=False)
```

Figure 6-8 shows the results of the warm start hyper parameter tuning job in the SageMaker UI. We see that 7 out of 8 tuning jobs stopped early because they could not beat the accuracy of the parent's best candidate.

Figure 6-8. Result of the Warm Start Hyper-Parameter Tuning Job in the SageMaker UI.

And the tuning job that completed successfully did not beat the accuracy of the parent's best candidate, either. Therefore, the hyper-parameter found from the parent tuning is still the best candidate as shown in Figure 6-9.

Figure 6-9. The Warm Start Tuning Job Does Not Beat the Accuracy of the Parent Tuning Job

Train and Tune Models at Scale

Most modern AI and machine learning frameworks support some form of distributed processing to scale out the computation. Without distributed processing, the training job is limited to the resources of a single instance. While individual instance types are constantly growing in capabilities (RAM, CPU, and GPU), our modern world of big data requires a cluster to power continuous data ingestion, real-time analytics, and data-hungry machine learning models.

In a previous chapter, we used an Apache Spark cluster to perform distributed feature engineering on the raw Amazon Customer Review Dataset at scale. In this section, we perform distributed training and tuning our reviews classifier using TensorFlow, Keras, and BERT.

Increase Cluster Instance Count

SageMaker, following cloud-native principles, is inherently distributed and scalable in nature. In the previous chapter, we were using a single instance by specifying `train_instance_count=1`. Simply by increasing the `train_instance_count > 1`, we will enable SageMaker distributed training as shown below.

```
train_instance_count=3
train_instance_type='ml.p3dn.24xlarge'
from sagemaker.tensorflow import TensorFlow
estimator = TensorFlow(entry_point='tf_bert_reviews.py',
                       source_dir='src',
train_instance_count=train_instance_count,
train_instance_type=train_instance_type,
...
                       py_version='py3',
                       framework_version='2.1.0')
```

SageMaker automatically passes the relevant cluster-environment information to TensorFlow to enable distributed TensorFlow. Our TensorFlow code does not change at all. SageMaker passes the same cluster-environment information to other distributed machine learning frameworks such as PyTorch and MXNet.

> **NOTE**
>
> Scikit-Learn does not natively support distributed computation. Other cluster-coordination frameworks like Dask are required to scale out Scikit-Learn to a cluster.

Choose an Appropriate Cluster Communication Strategy

The instances communicate and share their learnings throughout the training process. This communication benefits from a high-bandwidth connection between the cluster instances. And the instances should be physically close to each other in the cloud data center. Fortunately, SageMaker handles all of the heavy lifting for you so you can stay focused on the business problem you're trying to solve.

"Parameter server" is a primitive distributed training strategy supported by most distributed machine learning frameworks. Reminder that parameters are what the algorithm is learning. Hyper-parameters control how the algorithm learns these parameters. When performing distributed training across many cluster instances, we need a central place to store the parameters learned by all the nodes in the cluster as shown in Figure 6-10.

Figure 6-10. : Parameter Server Cluster Communication Strategy

Because they perform relatively lightweight computations, parameter server clusters are typically CPU-based instances while the worker cluster uses GPUs to perform the heavyweight, algorithm-specific computations and numerical optimizations. It's also important to note that parameter server clusters are stateful while worker clusters are stateless. This is an important distinction when evaluating failover-recovery situations and spot-instance configurations.

Another common distributed communication strategy rooted in parallel computing and message-passing interfaces (MPI) is "all reduce." All reduce uses a ring-like communication pattern as shown in Figure 6-11 which helps to reduce the single-point-of-failure of the parameter server and increase overall training efficiency.

Figure 6-11. : AllReduce Cluster Communication Strategy

Continuing with our TensorFlow example, we will leverage the `MultiWorkerMirroredStrategy` which uses the all reduce distributed communication strategy across many instances in our cluster.

```
distributed_strategy =
tf.distribute.MultiWorkerMirroredStrategy()
with distributed_strategy.scope():
        loss =
tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
     metric =
tf.keras.metrics.SparseCategoricalAccuracy('accuracy')
     model.compile(optimizer=optimizer,
  loss=loss,
  metrics=[metric])
               ...
```

Train with Distributed File Systems

Ideally, our distributed training clusters read and write directly to S3. However, some frameworks and tools are not optimized for S3 natively - or only support POSIX-compatible file systems. For these scenarios, we can use FSx for Lustre (Linux) or FSx for Windows to expose a POSIX-compatible file system on top of S3. This extra layer provides an additional caching benefit, as well.

Another option for distributed training is Amazon Elastic File System (EFS). EFS is compatible with industry-standard Network File System (NFS) protocols, but optimized for the AWS environment including networking, access control, encryption, and availability. In this section, we adapt our distributed training job to use both FSx for Lustre (Linux) and Amazon EFS.

Cache S3 Data using FSx for Lustre

Amazon FSx for Lustre is a high-performance, POSIX-compatible file system that natively integrates with S3. FSx for Lustre is based on the open source Lustre file system designed for highly scalable, highly distributed, and highly parallel training jobs with petabytes of data, terabytes per second of aggregate I/O throughput, and consistent low-latency.

There is also FSx for Windows which provides a Windows-compatible file system that natively integrates with S3, as well. However, we have chosen to focus on FSx for Lustre as our examples are Linux-based. Both file systems are optimized for machine learning, analytics, and high performance computing workloads using S3. And both file systems offer similar features.

FSx for Lustre is a fully-managed service that simplifies the complexity of setting up and managing the Lustre File System. Mounting an S3 bucket as a file system in minutes, FSx for Lustre lets you access data from any number of instances concurrently. FSx for Lustre caches S3 objects to improve performance of iterative machine learning workloads that pass over the dataset many times to fit a high-accuracy model. Figure 8-12 shows how SageMaker uses FSx for Lustre to provide fast, shared access to your S3 data and accelerate your training and tuning jobs. Your SageMaker training cluster instances access a file in FSx for Lustre using /mnt/data/file1.txt. FSx for Lustre translates this request and issues a GetObject request to S3. The file is cached and returned to the cluster instance. If the file has not changed, subsequent requests will return from FSx for Lustre's cache. Since training data does not typically change during a training job run, we see huge performance gains as we iterate through our dataset over many training epochs.

Figure 6-12. SageMaker Uses FSx for Lustre to Increase Training and Tuning Job Performance

Once you have set up the FSx for Lustre file system, you can pass the location of the FSx for Lustre file system into the training job as follows:

```
estimator = TensorFlow(entry_point='tf_bert_reviews.py',
                       source_dir='src',
                       train_instance_count=train_instance_count,
                       train_instance_type=train_instance_type,
                       subnets=['subnet-1', 'subnet-2']
                       security_group_ids=['sg-1'])
fsx_data = FileSystemInput(file_system_id='fs-1',
                           file_system_type='FSxLustre',
                           directory_path='/mnt/data,
                           file_system_access_mode='ro')
estimator.fit(inputs=fsx_data)
```

Share Data using Elastic File System

While many SageMaker optimizations exist for reading training data from S3, there are also optimizations for reading data from Amazon's Elastic File System (EFS), an implementation of the standard Network File System (NFS) optimized for AWS's cloud-native and elastic environment. EFS provides centralized, shared access to your training data sets across thousands of instances in a distributed training cluster as shown in Figure 6-13.

Figure 6-13. Amazon Elastic File System (EFS) with SageMaker

> **NOTE**
>
> SageMaker Studio uses EFS to provide centralized, shared access to code and notebooks across all team members.

Data stored in EFS is replicated across multiple availability zones which provides higher availability and read/write throughput. The EFS file system will scale out automatically as new data is ingested.

Assuming you have mounted and populated the EFS file system with training data, you can pass the EFS mount into the training job using two(2) different implementations: `FileSystemInput` and `FileSystemRecordSet`.

This example shows how to use the `FileSystemInput` implementation:

```
estimator = TensorFlow(entry_point='tf_bert_reviews.py',
                       source_dir='src',
                       train_instance_count=train_instance_count,
                       train_instance_type=train_instance_type,
                       subnets=['subnet-1', 'subnet-2']
                       security_group_ids=['sg-1'])
efs_data = FileSystemInput(file_system_id='fs-1',
                                   file_system_type='EFS',
                           directory_path='/mnt/data,
                           file_system_access_mode='ro')
estimator.fit(inputs=efs_data)
```

Reduce Costs and Increase Performance

In this section, we discuss various ways to increase cost-effectiveness and performance using some advanced SageMaker features including Autopilot for baseline hyper-parameter selection, ShardedByS3Key to distribute input files across all training instances, and PipeMode to improve I/O throughput. We also highlight AWS's enhanced networking capabilities including Enhanced Network Adapter (ENA) and Elastic Fabric Adapter (EFA) to optimize network performance between instances in your training and tuning cluster.

Shard the Data with ShardedByS3Key

When training at scale, we need to consider how each instance in the cluster will read the large training datasets. We can use a brute-force approach and copy all of the data to all of the instances. However, with larger datasets, this may take a long time and potentially dominate the overall training time. For example, after performing feature engineering, our tokenized training dataset has approximately 45 TFRecord "part" files as shown below.

```
part-algo-1-amazon_reviews_us_Apparel_v1_00.tfrecord
...
part-algo-2-amazon_reviews_us_Digital_Software_v1_00.tfrecord
part-algo-4-amazon_reviews_us_Digital_Video_Games_v1_00.tfrecord
...
part-algo-9-amazon_reviews_us_Sports_v1_00.tfrecord
```

Rather than load all 45 parts files onto all instances in the cluster, we can improve startup performance by placing only 15 part files onto each of the 3 cluster instances for a total of 45 part files spread across the cluster. This is called "sharding." We will use a SageMaker feature called `ShardedByS3Key` which evenly distributes the part files across the cluster as shown in Figure 6-14.

Figure 6-14. : Using ShardedByS3Key Distribution Strategy to Distribute the Input Files Across the Cluster Instances

Here we set up the `ShardedByS3Key` distribution strategy for our S3 input data including the train, validation, and test datasets:

```
s3_input_train_data =
sagemaker.s3_input(s3_data=processed_train_data_s3_uri,
                                 distribution='ShardedByS3Key')
s3_input_validation_data =
sagemaker.s3_input(s3_data=processed_validation_data_s3_uri,
   distribution='ShardedByS3Key')
s3_input_test_data =
sagemaker.s3_input(s3_data=processed_test_data_s3_uri,
   distribution='ShardedByS3Key')
```

Next, we call `fit()` with the input map for each of our dataset splits including train, validation, and test.

```
estimator.fit(inputs={'train': s3_input_train_data,
                      'validation': s3_input_validation_data,
                      'test': s3_input_test_data
              })
```

In this case, each instance in our cluster will receive approximately 15 files for each of the dataset splits.

Stream Data On-the-Fly with PipeMode

In addition to sharding, you can also use a SageMaker feature called `PipeMode` to load the data on-the-fly and as needed. Up until now, we've been using the default `input_mode` of `File` which copies all of the data to all the instances when the training job starts. This creates a long pause at the start of the training job as the data is copied.

`PipeMode` streams data in parallel from S3 directly into the training processes running on each instance which provides significantly-higher I/O throughput than `File` mode. All of this allows your training jobs to start quicker, complete faster, and use less disk space - lowering overall cost of your training and tuning jobs.

PipeMode works with S3 to pull the rows of training data as needed. Under the hood, PipeMode is using Unix first-in, first-out (FIFO) files to read data from S3 and cache it locally on the instance shortly before the data is needed by the training job. These FIFO files are one-way readable. In other words, you can't back up or skip ahead randomly.

Here is how we configure our training job to use `PipeMode`:

```
input_mode='Pipe'
estimator = TensorFlow(entry_point='tf_bert_reviews.py',
                       source_dir='src',
                       train_instance_count=train_instance_count,
                       train_instance_type=train_instance_type,
```

...

```
                       input_mode=input_mode)
```

Since `PipeMode` wraps our TensorFlow Dataset Reader, we need to change our TensorFlow code slightly to detect `PipeMode` and use the `PipeModeDataset` wrapper:

```
if input_mode == 'Pipe':
    from sagemaker_tensorflow import PipeModeDataset
    dataset = PipeModeDataset(channel=channel,
                              record_format='TFRecord')
else:
    dataset = tf.data.TFRecordDataset(input_filenames)
```

Enable Enhanced Networking

Training at scale requires super-fast communication between instances in the cluster. Be sure to select an instance type that utilizes the enhanced networking adapter (ENA) and Elastic Fabric Adapter (EFA) to provide high network bandwidth and consistent network latency between the cluster instances.

The Enhanced Network Adapter (ENA) works well with the AWS deep learning instances types including the C, M, P, and X series. These instance types offer a large number of CPUs, so they benefit greatly from efficient sharing of the network adapter. By performing various network-level optimizations such as hardware-based checksum generation and software-based routing, ENA reduces overhead, improves scalability, and maximizes consistency. All of these optimizations are designed to reduce bottlenecks, offload work from the CPUs, and create an efficient path for the network packets.

The Elastic Fabric Adapter (EFA) uses custom-built, OS-level bypass techniques to improve network performance between instances in a cluster. EFA natively supports Message Passing Interface (MPI) which is critical to scaling high-performance computing (HPC) applications that scale to thousands of CPUs. EFA is supported by many of the compute-optimized instance types including the C and P series.

Summary

In this chapter, we used SageMaker Experiments and Hyper-Parameter Optimizer (HPO) to track, compare, and choose the best hyper-parameters for our specific algorithm and dataset. We explored various distributed communication strategies such as Parameter Server and AllReduce. We demonstrated how to use FSx for Lustre to increase S3 performance and how to configure our training job to use the Elastic File System (EFS). Next, we explored a few ways to reduce cost and increase performance using Autopilot's hyper-parameter selection feature and SageMaker's optimized data loading strategies like `ShardedByS3Key` and `PipeMode`. Last, we discussed the enhanced networking features for compute-optimized instance types including Elastic Network Adapter (ENA) and Elastic Fabric Adapter (EFA).

In the next chapter, we will deploy our models into production using various rollout and A/B testing strategies. We will discuss how to integrate model predictions into applications using real-time REST endpoints and offline batch jobs.

Chapter 7. Deploying Models to Production with SageMaker

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 9th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *support@pipeline.ai*.

In the previous chapters, you learned how to train and optimize models. In this chapter, we shift focus from model development in the research lab to model deployment in production. We demonstrate how to deploy, optimize, scale, and monitor models to serve our applications and business use cases. We will deploy our model as both a REST endpoint as well as a batch transformation job.

Collaborating with Multiple Teams

Model deployments require close collaboration between the application, data science, and devops teams to successfully productionize our models as shown in Figure 7-1.

Figure 7-1. Productionizing machine learning applications requires collaboration between teams.

The data scientist delivers the trained model, the DevOps engineer provisions the needed infrastructure to host the model API, and the

application developer integrates the model API calls into their applications. Each team must understand the needs and requirements of the other teams in order to implement an efficient workflow and smooth hand-off process.

Choose Level Of Customization

Similar to model training, you can choose between three(3) deployment options depending on the level of customization you desire as shown in Figure 7-2.

SageMaker Options

Amazon SageMaker

Built-In Models

Amazon SageMaker

Bring Your Own Script

Amazon SageMaker

Bring Your Own Container

More Customizable

Less Code

Figure 7-2. SageMaker Options to Deploy Your Model

Built-In Algorithms

Using a built-in SageMaker algorithm, you leverage both the pre-built inference code and model server container. These algorithms, shown in the table below, are targeted towards practitioners who don't want to manage a lot of infrastructure and don't need a lot of algorithm customizations.

Classification		Computer Vision		Working with Text	
Learner	Linear		Image Classification		BlazingText
	XGBoost		Object Detection		Supervised
	KNN	Segmentation	Semantic	Unsupervised	
Regression		Anomaly Detection		Topic Modeling	
Learner	Linear		Random Cut Forests		LDA
	XGBoost		IP Insights		NTM
	KNN				
Sequence Translation		Recommendation		Clustering	
	Seq2Seq	Machines	Factorization		KMeans
Feature Reduction		Forecasting			
	PCA		DeepAR		
	Object2Vec				

Bring Your Own Script

Similar to "bring your own script" or "script mode" for model training, SageMaker provides highly-optimized, open source inference containers for each of the familiar open source frameworks such as TensorFlow, PyTorch, MXNet, XGBoost, and Scikit-Learn as shown in Figure 5-3.

Figure 7-3. Popular AI and Machine Learning Frameworks

Script mode is a good balance of high customization and low maintenance, so we will choose this option for the examples in this chapter as shown in Figure 7-4.

SageMaker Bring Your Own Script

Figure 7-4. Script model with BERT and TensorFlow is a good balance of customization and maintenance

Bring Your Own Container

The most customizable inference option is to "bring your own container" (BYOC). This option lets you build and deploy your own inference container to SageMaker. This container can contain any library, framework, or model server. SageMaker will manage the low-level infrastructure for logging, monitoring, environment variables, S3 locations, etc. This option is targeted at systems-focused machine learning practitioners with custom inference needs.

Choose Real-Time or Batch Predictions

We need to understand the application and business context to choose between real-time and batch predictions. Are we trying to optimize for latency or throughput? Does the application require our models to scale automatically throughout the day to handle cyclic traffic requirements? Do we plan to compare models in production through A/B tests?

If our application requires low latency, then we should deploy the model as a real-time API to provide super-fast predictions on single prediction requests over HTTPS, for example. We can deploy, scale, and compare our model prediction servers with SageMaker Endpoints as shown in Figure 7-5.

Figure 7-5. Real-time predictions as a REST API

For applications that require high throughput, we should deploy our model as a batch job to perform batch predictions on large amounts of data in S3, for example. We will use SageMaker Batch Transformations to perform the batch predictions along with a data store like RDS or DynamoDB to productionize the predictions as shown in Figure 7-6.

Batch Predictions Require a Data Store to Productionize the Predictions

Figure 7-6. Batch predictions with SageMaker.

Real-Time Predictions with SageMaker Endpoints

In 2002, Jeff Bezos, CEO of Amazon, wrote a memo to his employees later called the "Bezos API Mandate". The mandate dictated that all teams must expose their services through APIs - and communicate with each other through these APIs. This mandate addressed the "dead-lock" situation that Amazon faced back in the early 2000's in which everybody wanted to build and use APIs, but nobody wanted to spend the time refactoring their monolithic code to support this idealistic best practice. The mandate released the deadlock and required all teams to build and use APIs within Amazon.

> **NOTE**
>
> Seen as the cornerstone of Amazon's success early on, the Bezos API Mandate is the foundation of Amazon Web Services as we know it today. APIs helped Amazon re-use their internal ecosystem as scalable managed services for other organizations to build upon.

Following the Bezos API Mandate, we will deploy our model as a REST API using SageMaker Endpoints. SageMaker Endpoints are, by default, distributed containers. Applications invoke our models through a simple RESTful interface as shown in Figure 7-7 which shows the model deployed across multiple cluster instances and availability zones for higher availability.

Application Invoking the SageMaker Endpoint REST API

Figure 7-7. Application invoking our highly-available model as a REST endpoint

Deploy Model using SageMaker Python SDK

There are two ways to deploy the model using the SageMaker Python SDK. We can call `deploy()` on a model object, or we can call `deploy()` on a SageMaker estimator object that we used to train the model. Creating a model object also allows us to deploy a model that has been trained outside of the Amazon SageMaker platform.

Here is sample code for calling `deploy()` on a TensorFlow model:

```
from sagemaker.tensorflow.serving import Model
# Create the model object from a saved model
tf_model = Model(model_data=saved_model,
                        role=role,
                        framework_verison='2.1')
# Deploy the model
predictor = tf_model.deploy(initial_instance_count=1,
                                instance_type = 'ml.m4.xlarge')
# Run a sample prediction
prediction = predictor.predict({'inputs': payload})
```

Endpoint Configuration

When we call `deploy()` SageMaker automatically creates a default `EndpointConfig` for the model endpoint. The `EndpointConfig` defines parameters such as whether to enable data capture for the model, the list of production variants and traffic split weights etc. We will see how to create a custom `EndpointConfig` in a later section.

Here is sample code for calling `deploy()` on a SageMaker estimator object after the training has completed:

```
from sagemaker.tensorflow.serving import TensorFlow
# Create the estimator
estimator = TensorFlow(entry_point='train.py',
                                role=role,
                                train_instance_count=2,
                                train_instance_type='ml.p3.2xlarge',
```

```
                                framework_verison='2.1',
py_version='py3')
# Train the model
estimator.fit(training_data_uri)
# Deploy the model from the estimator
predictor = estimator.deploy(initial_instance_count=1,
                                        instance_type='ml.m4.xlarge')
# Run a sample prediction
prediction = predictor.predict({'inputs': payload})
```

Note that in both scenarios we can simply call `predict()` on the deployed model to run sample predictions. Coming back to our fine-tuned BERT model, let's choose the first option to deploy the model into production:

```
from sagemaker.tensorflow.serving import Model
model =
Model(model_data='s3://{}/{}/output/model.tar.gz'.format(bucket,
training_job_name),
                                                  role=role,
framework_version='2.1.0')
deployed_model = model.deploy(initial_instance_count=1,
        instance_type='ml.m4.xlarge',
                wait=False)
endpoint_name = deployed_model.endpoint
```

Track Model Deployment in our Experiment

We also want to track the deployment within our experiment for data lineage.

```
from smexperiments.trial import Trial
trial = Trial.load(trial_name=trial_name)
from smexperiments.tracker import Tracker
tracker_deploy = Tracker.create(display_name='deploy',
                sagemaker_boto_client=sm)
deploy_trial_component_name = tracker_deploy.trial_component.trial_component_name
# Attach the 'deploy' Trial Component and Tracker to the Trial
trial.add_trial_component(tracker_deploy.trial_component)
# Track the Endpoint Name
tracker_deploy.log_parameters({
  'endpoint_name': endpoint_name,
})
# Must save after logging
tracker_deploy.trial_component.save()
```

Analyze Model Deployment Lineage

Let's use the Experiment Analytics to show us model lineage information:

```
from sagemaker.analytics import ExperimentAnalytics
lineage_table = ExperimentAnalytics(
  sagemaker_session=sess,
  experiment_name=experiment_name,
  metric_names=['validation:accuracy'],
  sort_by="CreationTime",
  sort_order="Ascending",
)
lineage_df = lineage_table.dataframe()
print(lineage_df)
```

If we print `lineage_df` the output should look similar to this (shortened):

	TrialComponentName	DisplayName	max_seq_length	test_split_percentage	train_split_percentage	validation_split_percentage	endpoint_name	SourceArn
0	TrialComponent-2020-06-06-...	prepare	128.0	0.05	0.9	0.05	NaN	NaN
1	tensorflow-training-2020-06-06-...	train	128.0	NaN	NaN	NaN	NaN	arn:aws:sagemaker:us-east-1:...
2	TrialComponent-2020-06-06-...	deploy	NaN	NaN	NaN	NaN	tensorflow-training-2020-06-06-...	NaN

Invoking Predictions using the SageMaker Python SDK

Here is some simple application code to query our deployed endpoint for predictions. We define a `RequestHandler` class which transforms the raw prediction input data (our customer reviews text) into BERT tokens required by the model.

```
class RequestHandler(object):
  import json

  def __init__(self, tokenizer, max_seq_length):
```

```
    self.tokenizer = tokenizer
    self.max_seq_length = max_seq_length
  def __call__(self, instances):
    transformed_instances = []
    for instance in instances:
      encode_plus_tokens = tokenizer.encode_plus(instance,
                           pad_to_max_length=True,
                           max_length=self.max_seq_length)
      input_ids = encode_plus_tokens['input_ids']
      input_mask = encode_plus_tokens['attention_mask']
      segment_ids = [0] * self.max_seq_length
      transformed_instance = {"input_ids": input_ids,
                  "input_mask": input_mask,
                  "segment_ids": segment_ids}
      transformed_instances.append(transformed_instance)
    transformed_data = {"instances": transformed_instances}
    return json.dumps(transformed_data)
```

Similarly, we implement a `ResponseHandler` class which transforms the BERT model output into star rating classes.

```
class ResponseHandler(object):
                import json
  import tensorflow as tf

  def __init__(self, classes):
    self.classes = classes

  def __call__(self, response, accept_header):
    import tensorflow as tf
    response_body = response.read().decode('utf-8')
    response_json = json.loads(response_body)
    log_probabilities = response_json["predictions"]
    predicted_classes = []
    # Convert log_probabilities => softmax (all probabilities add up to 1) =>
argmax (final prediction)
    for log_probability in log_probabilities:
      softmax = tf.nn.softmax(log_probability)
      predicted_class_idx = tf.argmax(softmax, axis=-1, output_type=tf.int32)
      predicted_class = self.classes[predicted_class_idx]
      predicted_classes.append(predicted_class)
    return predicted_classes
```

The SageMaker Predictor class conveniently offers us a serializer and deserializer input parameter which we can use to pass our `RequestHandler` and `ResponseHandler` objects. The final prediction query looks like this:

```
import json
from sagemaker.tensorflow.serving import Predictor
from transformers import DistilBertTokenizer
tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-uncased')
request_handler = RequestHandler(tokenizer=tokenizer,
                 max_seq_length=128)
response_handler = ResponseHandler(classes=[1, 2, 3, 4, 5])
predictor = Predictor(endpoint_name=endpoint_name,
           sagemaker_session=sess,
           serializer=request_handler,
           deserializer=response_handler,
           content_type='application/json',
           model_name='saved_model',
           model_version=0)
```

Now, let's test our model endpoint with two sample reviews:

```
import tensorflow as tf
import json

reviews = ["This is great!",
      "The worst product."]
predicted_classes = predictor.predict(reviews)
for predicted_class, review in zip(predicted_classes, reviews):
  print('[Predicted Star Rating: {}]'.format(predicted_class), review)
```

The output should look similar to this:

```
[Predicted Star Rating: 5] This is great!
[Predicted Star Rating: 1] The worst product.
```

Invoke Predictions from Using HTTP POST

In this example, we invoked the model endpoint directly using the Python SDK. When you productionize your machine learning applications you

need to decide how to make your runtime inference endpoints available to client applications. You can call the SageMaker model endpoint API directly following this HTTP request/response syntax:

```
Request syntax:
POST /endpoints/<EndpointName>/invocations HTTP/1.1
Content-Type: ContentType
Accept: Accept
X-Amzn-SageMaker-Custom-Attributes: <CustomAttributes>
X-Amzn-SageMaker-Target-Model: <TargetModel>
Body
Response syntax:
HTTP/1.1 200
Content-Type: ContentType
x-Amzn-Invoked-Production-Variant: <InvokedProductionVariant>
X-Amzn-SageMaker-Custom-Attributes: <CustomAttributes>
Body
```

In this example, we implemented the data pre- and post-processing in our client application. The RequestHandler transformed our input review text into BERT tokens, and the ResponseHandler transformed the BERT model's output into star rating classes. If you prefer to implement the pre- and post-processing steps server-side, you either include them in `train.py` as `input_handler()` and `output_handler()` functions. For more complex request and response handlers, you can deploy each step in their own container using SageMaker Inference Pipelines.

Creating Inference Pipelines

An inference pipeline is a sequence of containers deployed on a single endpoint. Following our example, we could deploy the RequestHandler as its own model in a separate Scikit-Learn container(Model 1), followed by the TensorFlow/BERT model in its own TensorFlow Serving container (Model 2), and succeeded by the ResponseHandler as its own model in a separate Scikit-Learn container (Model 3) as shown in Figure 7-8.

Inference Pipeline with 3 Models
Pipeline Request
Application
SageMaker Pipeline Controller
Model 1 Request (Pipeline Request)
Model 1 Response
Model 1
Model 1 Response
Model 2 Response
Model 2
Model 2 Response
Model 3 Response (Pipeline Response)
Model 3
Model 3 Response

Figure 7-8. Inference Pipeline with Three(3) Models

You can also deploy completely unique models across different AI and machine learning frameworks including TensorFlow, PyTorch, Scikit-Learn, Apache Spark ML, etc. The model invocations are handled as a sequence of HTTPS requests between the containers controlled by SageMaker. One model's response is used as the prediction request for the next model and so on. The last model returns the final response back to the SageMaker controller which returns the response back to the calling application. The inference pipeline is fully-managed by SageMaker and can be used for real-time predictions as well as batch transforms.

To deploy an inference pipeline, we need to create a `PipelineModel` comprising a list of model objects which represent the sequence of containers. We can then call `deploy()` on the `PipelineModel` which deploys the inference pipeline and returns the endpoint API:

```
# Define model name and endpoint name
model_name = 'inference-pipeline-model'
endpoint_name = 'inference-pipeline-endpoint'
# Create a PipelineModel with a list of models to deploy in sequence
pipeline_model = PipelineModel(name=model_name,
  role=sagemaker_role,
  models=[model1, model2, model3])
# Deploy the PipelineModel
pipeline_model.deploy(initial_instance_count=1,
instance_type='ml.c5.xlarge',
endpoint_name=endpoint_name)
```

The `pipeline_model.deploy()` returns a predictor as seen in the single model example. Whenever you make an inference request to this predictor, make sure you pass the data that the first model container expects. The predictor returns the output from the last model container..

If you want to run batch transform job with the pipeline model, just follow the steps of creating `pipeline_model.transformer()` object and call `transform()`:

```
transformer = pipeline_model.transformer(instance_type='ml.c5.xlarge',
                  instance_count=1,
                  strategy='MultiRecord',
                  max_payload=6,
                  max_concurrent_transforms=8,
                  accept='text/csv',
                  assemble_with='Line',

output_path='s3://my_bucket/path/')
transformer.transform(data='s3://my_bucket/path/to/my/csv/data',
           content_type='text/csv',
           split_type='Line')
```

Deploying New Models

You can test and deploy new models behind a single SageMaker Endpoint with a concept called "production variants." These variants can differ by hardware (CPU/GPU), by data (comedy/drama movies), or by region (US West or Germany North). You can shift traffic between the models in your endpoint for canary rollouts, blue/green deployments, A/B tests and multi-armed bandit (MAB) tests.

Split Traffic for Canary Rollouts

Since our data is continuously changing, our models need to evolve to capture this change. When we update our models, we may choose to do this slowly using a "canary rollout" named after an antiquated and morbid process of using a canary to early-detect if a human could breathe in a coal mine. If the canary survives the coal mine, then the conditions are good and we can proceed. If the canary does not survive, then we should make adjustments and try again later with a different canary. Similarly, we can point a small percentage of traffic to our "canary" model and test if the model services. Perhaps there is a memory leak or other production-specific issue that we didn't catch in the research lab.

The combination of the cloud instance providing compute, memory and storage, and the model container application is called a "production

variant." The production variant defines the instance type, instance count, and model. By default, the SageMaker endpoint is configured with a single production variant, but you can add multiple variants as needed.

Here is the code to setup a single variant, VariantA, at a single endpoint receiving 100% of the traffic across 20 instances:

```
endpoint_config = sm.create_endpoint_config(
    EndpointConfigName='my_endpoint_config_name',
    ProductionVariants=[
        {
         'VariantName': 'VariantA',
         'ModelName': 'ModelA',
         'InstanceType':'ml.m5.large',
         'InitialInstanceCount': 20,
         'InitialVariantWeight': 100,
        }
    ])
```

After creating a new production variant for our canary, we can create a new endpoint and point a small amount of traffic (5%) to the canary and point the rest of the traffic (95%) our existing variant as shown in Figure 7-9.

Add Model Variant B for Canary Release
SageMaker Endpoint Configuration
Model A
Model B
SageMaker Endpoint Configuration
Model A
Model B
100%
0%
Traffic distribution
95%
5%

Figure 7-9. Splitting 5% traffic to a new model for a canary rollout

Below is the code to create a new endpoint including the new canary VariantB accepting 5% of the traffic to the canary. Note that we only specify `'InitialInstanceCount': 1` for the new canary, `VariantB`. Assuming 20 instance handles 100% of the traffic, then 1 instance can handle 5%.

```
updated_endpoint_config=[
    {
     'VariantName': 'VariantA',
     'ModelName': 'ModelA',
     'InstanceType':'ml.m5.large',
     'InitialInstanceCount': 20,
     'InitialVariantWeight': 95,
    },
    {
     'VariantName': 'VariantB',
     'ModelName': 'ModelB',
     'InstanceType':'ml.m5.large',
     'InitialInstanceCount': 1,
     'InitialVariantWeight': 5,
    }
])
sm.update_endpoint(
    EndpointName='my_endpoint_name',
    EndpointConfigName='my_endpoint_config_name'
)
```

Canary rollouts release new models safely to a small percentage of users for initial production testing in the wild. They are useful if you want to test in live production without affecting the entire user base. Since the majority of traffic goes to the existing model, the cluster size of the canary model can be relatively small since it's only receiving 5% traffic. In the example above, we are only using a single instance for the canary variant.

Shift Traffic for Blue/Green Deployments

If the new model performs well, we can process with a blue/green deployment to shift all traffic to the new model as shown in Figure 7-10. Blue/green deployments help to reduce downtime in the case of failure. You

spin up a full copy of the existing cluster using the canary model instead of the existing model. You then shift all the traffic from the old cluster (blue) over to the new cluster (green).

Shift Traffic 100% to Model Variant B for Blue/Green Deployment
SageMaker Endpoint Configuration
SageMaker Endpoint Configuration
Model A
Model B
Model A
Model B
100%
0%
Traffic distribution
0%
100%

Figure 7-10. Shift traffic for blue/green deployments.

Below is the code to update our endpoint and shift 100% of the traffic to the successful canary model, VariantB. Note that we have also increased the size of the new cluster to match the existing cluster since the new cluster is now handling all of the traffic.

```
updated_endpoint_config=[
    {
     'VariantName': 'VariantA',
     'ModelName': 'ModelA',
     'InstanceType':'ml.m5.large',
     'InitialInstanceCount': 20,
     'InitialVariantWeight': 0,
    },
    {
     'VariantName': 'VariantB',
     'ModelName': 'ModelB',
     'InstanceType':'ml.m5.large',
     'InitialInstanceCount': 20,
     'InitialVariantWeight': 100,
    }
])
sm.update_endpoint_weights_and_capacities(
    EndpointName='my_endpoint_name',
    DesiredWeightsAndCapacities=updated_endpoint_config
)
```

We will keeping the old cluster with VariantA idle for 24 hours, let's say, in case our canary fails unexpectedly and we need to rollback quickly to the old cluster. After 24 hours, we can remove the old environment and complete the blue/green deployment. Below is the code to remove the old model, VariantA, by removing VariantA from the endpoint configuration and updating the endpoint.

```
updated_endpoint_config=[
    {
     'VariantName': 'VariantB',
     'ModelName': 'ModelB',
     'InstanceType':'ml.m5.large',
     'InitialInstanceCount': 2,
     'InitialVariantWeight': 100,
```

```
    }
])
sm.update_endpoint(
    EndpointName='my_endpoint_name',
    EndpointConfigName='my_endpoint_config_name'
)
```

While keeping the old cluster idle for a period of time - 24 hours, in our example - may seem wasteful, consider the cost of an outage during the time needed to rollback and scale out the previous model, VariantA. Sometimes the new model cluster works fine for the first few hours, then degrades or crashes unexpectedly after a night-time cron job, early-morning product catalog refresh, or other untested scenario. In these cases, we were able to immediately switch traffic back to the old cluster and conduct business as usual.

Testing and Comparing New Models

We can test new models behind a single SageMaker Endpoint using the same "production variant" concept described in the previous section on model deployment. In this section, we will configure our SageMaker Endpoint to shift traffic between the models in your endpoint to compare model performance in production using A/B and multi-armed bandit (MAB) tests.

When testing our models in production, we need to define and track the business metrics that we wish to optimize. The business metric is usually tied to revenue or user engagement such as orders purchased, movies watched, or ads clicked. We can store the metrics in any database such as DynamoDB as shown in Figure 7-11. Analysts and scientists will use this data to determine the winning model from our tests.

Determine the Best Model Variant through Live Production Testing

Figure 7-11. : Tracking business metrics to determine the best model variant

Continuing with our text-classifier example, we will create a test to maximize the number of successfully-labeled customer service messages. As customer service receives new messages, our application will predict the message's `star_rating` (1-5) and route 1's and 2's to a high-priority customer service queue. If the representative agrees with the predicted `star_rating`, they will mark our prediction as successful (positive feedback), otherwise they will mark the prediction as unsuccessful (negative feedback.) Unsuccessful predictions will be routed to a human-in-the-loop workflow using Amazon Augmented AI and SageMaker Ground Truth as shown in the Augmented AI chapter. We will then choose the model variant with the most successful number of `star_rating` predictions - and start shifting traffic to this winning variant. Let's dive deeper into managing the experiments and shifting the traffic.

Perform A/B Tests to Compare Model Variants

Similar to canary rollouts, we can use traffic splitting to direct subsets of users to different model variants for the purpose of comparing and testing different models in live production. The goal is to see which variants perform better. Often, these tests need to run for a long period of time (weeks) to be statistically significant. Figure 7-12 shows 2 different recommendation models deployed using a random 50-50 traffic split between the 2 variants.

Add Model Variant B for 50/50 A/B Test
SageMaker Endpoint Configuration
Model A
Model B
SageMaker Endpoint Configuration
Model A
Model B
100%
0%
Traffic distribution
50%
50%

Figure 7-12. A/B Testing with two(2) model variants by splitting traffic 50-50

While A/B testing seems similar to canary rollouts, they are focused on gathering data about different variants of a model. A/B tests are targeted to larger user groups, take more traffic, and run for longer periods of time. Canary rollouts are focused more on risk mitigation and smooth upgrades.

One example for a model A/B test could be streaming music recommendations. Let's assume we are recommending a playlist for Sunday mornings. You might want to test if you can identify specific user groups which are more likely to listen to powerful wake-up beats (model A) or that prefer smooth lounge music (model B). Let's implement this A/B test using Python. We start with creating a SageMaker endpoint configuration which defines two production variants: one for model A, and one for model B. We initialize both production variants with the identical instance types and instance counts.

```
import time
timestamp = '{}'.format(int(time.time()))
endpoint_config_name = '{}-{}'.format(training_job_name, timestamp)
variantA = production_variant(model_name='ModelA',
                              instance_type="ml.m5.large",
                              initial_instance_count=1,
                              variant_name='VariantA',
                              initial_weight=50)
variantB = production_variant(model_name='ModelB',
                              instance_type="ml.m5.large",
                              initial_instance_count=1,
                              variant_name='VariantB',
                              initial_weight=50)
endpoint_config = sm.create_endpoint_config(
    EndpointConfigName=endpoint_config_name,
    ProductionVariants=[variantA, variantB]
)
endpoint_name = '{}-{}'.format(training_job_name, timestamp)
endpoint_response = sm.create_endpoint(
  EndpointName=endpoint_name,
  EndpointConfigName=endpoint_config_name)
```

After we have monitored the performance of both models for a period of time, we can shift 100% of the traffic to the better performing model,

Model B in our case. Let's shift our traffic from a 50/50 split to a 0/100 split as shown in Figure 7-13.

Shift from 50/50 Split to 0/100
SageMaker Endpoint Configuration
Model A
Model B
SageMaker Endpoint Configuration
Model A
Model B
50%
50%
Traffic distribution
0%
100%

Figure 7-13. A/B testing traffic shift from 50/50 to 0/100.

Below is the code to shift all traffic to VariantB and ultimately remove VariantA we are confident that VariantB is working correctly.

```
updated_endpoint_config = [
  {
    'VariantName': 'VariantA',
    'DesiredWeight': 0,
  },
  {
    'VariantName': 'VariantB',
    'DesiredWeight': 100,
  }
]
sm.update_endpoint_weights_and_capacities(
  EndpointName='my_endpoint_name',
  DesiredWeightsAndCapacities=updated_endpoint_config
)
updated_endpoint_config=[
    {
     'VariantName': 'VariantB',
     'ModelName': 'ModelB',
     'InstanceType':'ml.m5.large',
     'InitialInstanceCount': 2,
     'InitialVariantWeight': 100,
    }
])
sm.update_endpoint(
    EndpointName='my_endpoint_name',
    EndpointConfigName='my_endpoint_config_name'
)
```

Reinforcement Learning with Multi-Armed Bandit Testing

A/B tests must run for a period of time - sometimes weeks or months - before they are considered statistically significant. During this time, we may have deployed a bad model variant that is negatively affecting revenue. However, if we stop the test, we can't rule out that the poor-performing model is caused merely by random chance.

A more dynamic method for testing different model variants is called Multi-Armed Bandits (MABs). Named after a slot machine which can quickly

take your money, these mischievous bandits can actually earn you quite a bit of money by dynamically shifting traffic to the winning model variants much sooner than with an A/B test. This is the “exploit” part of the MAB. At the same time, the MAB continues to “explore” the non-winning model variants just in case the early winners were not the overall best model variants. This dynamic pull between “exploit and explore” are what give MABs their power.

Based on reinforcement learning, MAB’s rely on the feedback positive-negative mechanism described earlier in the A/B testing section. The MAB acts as the primary SageMaker Endpoint and dynamically shifts the prediction traffic as shown in Figure 7-14.

Find the Best Model Variant with Multi-Armed Bandits

Figure 7-14. : Find the best model variant using multi-armed bandits

The multi-armed bandit chooses the model variant based on the current success metrics and the chosen exploit-explore strategy. There are two(2) popular MAB strategies: Epsilon Greedy and Thompson's Sampling. Epsilon Greedy uses a fixed exploit-explore threshold while Thompson's Sampling uses a more sophisticated and dynamic threshold based on prior information - a technique rooted in Bayesian statistics. The MAB is aware of the latest success metrics and will adjust the prediction traffic accordingly.

Auto-Scale SageMaker Endpoints using CloudWatch

While we can manually scale using the `InstanceCount` parameter in `EndpointConfig`, we can configure our endpoint to automatically scale out (more instances) or in (less instances) based on a given metric like requests per second. As more requests come in, SageMaker will automatically scale your model cluster to meet the demand.

In the cloud, we talk about "scaling in" and "scaling out" in addition to the typical "scaling down" and "scaling up." Scaling in and out refers to removing and adding instances of the same type, respectively. Scaling down and up refers to using smaller or bigger instance types, respectively. Larger instances have more CPUs, GPUs, memory, and network bandwidth, typically.

It's best to use homogenous instance types when defining your cluster. If we mix instance types, we may have difficulty tuning the cluster and defining scaling policies that apply consistently to every instance in the heterogeneous cluster. When trying new instance types, we recommend creating a new cluster with only that instance type and comparing each cluster as a single unit.

Define a Scaling Policy with Custom Metrics

Netflix is known to use a custom auto-scaling metric called “Starts per Second” or SPS. A start is recorded every time a user clicks “play” to watch a movie or tv show. This was a key metric for auto scaling as the more “Starts per Second”, the more traffic we would start receiving on our streaming control plane.

Assuming we are publishing the “StartsPerSecond” metric, we can use this custom metric to scale out our cluster as more movies are started. This metric is called a “target tracking” metric and we need to define the metric name, target value, model name, variant name, and summary statistic. The scaling policy below will begin scaling out the cluster if the aggregate StartsPerSecond metric exceeds an average of 50% across all instances in your model-serving cluster.

```
{  "TargetValue": 50,
  "CustomizedMetricSpecification":
  {
    "MetricName": "StartsPerSecond",
    "Namespace": "/aws/sagemaker/Endpoints",
    "Dimensions": [
      {"Name": "EndpointName", "Value": "ModelA" },
      {"Name": "VariantName","Value": "VariantA"}
    ],
    "Statistic": "Average",
    "Unit": "Percent"
  }
}
```

When using custom metrics for your scaling policy, you should pick a metric that measures instance utilization, decreases as more instances are added, and increases as instances are removed.

Using Pre-Defined Metrics

For our example, we will use an equivalent, pre-defined metric exposed by SageMaker Endpoints called “InvocationsPerInstance” as follows. The scaling policy below will begin scaling out the cluster if the aggregate InvocationsPerSecond metric exceeds an average of 60% across all instances in your model-serving cluster.

```
{  "TargetValue": 60.0,
  "PredefinedMetricSpecification":
  {
    "PredefinedMetricType":
"SageMakerVariantInvocationsPerInstance"
  }
}
```

Tuning Responsiveness Using a Cool Down Period

When your endpoint is autoscaling in or out, you likely want to specify a "cool down" period in seconds. A cool down period essentially reduces the responsiveness of the scaling policy by defining the number of seconds between iterations of the scale events. While you likely want to scale out quickly when a spike of traffic comes in, you may want to slowly scale in to make sure you're handling any temporary dips in traffic during these scale out events. The following scaling policy will take twice as long to scale in as it does to scale out as shown in the `ScaleInColldown` and `ScaleOutCooldown` attributes below.

```
{
  "TargetValue": 60.0,
  "PredefinedMetricSpecification":
  {
    "PredefinedMetricType":
"SageMakerVariantInvocationsPerInstance"
  },
  "ScaleInCooldown": 600,
  "ScaleOutCooldown": 300}
```

Monitor Predictions and Detect Drift

The world continues to change around us. Customer behavior changes relatively quickly. The application team is releasing new features. The Netflix catalog is swelling with new content. Fraudsters are finding clever ways to hack our credit cards. A continuously-changing world requires continuously-changing predictive models that can detect and capture this "drift" and trigger a re-train to increase model performance. By

automatically recording SageMaker Endpoint predictions (and their inputs), SageMaker Model Monitor automatically detects and measures this drift. Model Monitor will notify you when the drift reaches a certain threshold from a baseline specified during model deployment.

To enable Model Monitor, we need to perform the following steps:

Enable Data Capture

Enable the endpoint to capture data from incoming inference requests to a trained model and the resulting model predictions.

Create a Baseline

Create a baseline from the dataset that was used to train the model. Model monitor uses AWS Deequ Open Source to identify the baseline schema constraints and statistics for each feature.

Schedule Monitoring Jobs

Create a monitoring schedule specifying what data to collect, how often to collect it, how to analyze it, and which reports to produce.

Interpret Results

Inspect the reports, which compare the latest data with the baseline, and check for any violations reported and for metrics and notifications from Amazon CloudWatch.

Let's dive deeper into each of these steps to monitor our predictions in production.

Enable Data Capture

SagMaker Model Monitor analyzes your model predictions (and their inputs) to detect drift. So the first step to configuring Model Monitor is to enable data capture for a given endpoint as shown in Figure 7-15.

Figure 7-15. Enable model endpoint data capture.

Below is the code to enable data capture. You can define all configuration options in the `DataCaptureConfig` object. You can choose to capture the request payload, the response payload or both with this configuration. The capture config applies to all model production variants of the endpoint.

```
from sagemaker.model_monitor import DataCaptureConfig
data_capture_config = DataCaptureConfig(
            enable_capture=True,
            sampling_percentage=100,
            destination_s3_uri='s3://my_bucket/path/')
Next, you pass the DataCaptureConfig in the model.deploy() call:
predictor = model.deploy(initial_instance_count=1,
        instance_type='ml.m4.xlarge',
        endpoint_name=endpoint_name,
        data_capture_config=data_capture_config)
```

We are now capturing all inference requests and prediction results in the specified S3 destination. In order to detect any model or data drift, we need to create a baseline to compare the data against.


```
{"captureData":{"endpointInput":
{"observedContentType":"application/json","mode":"INPUT","data":"{\"instances\":
[{\"input_ids\": [101, 2023, 2003, 2307, 999, 102, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], \"input_mask\": [1,
1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0], \"segment_ids\": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]},
{\"input_ids\": [101, 2023, 2003, 6659, 1012, 102, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], \"input_mask\": [1,
1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0], \"segment_ids\": [0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0]}]}","encoding":"JSON"},"endpointOutput":{"observedContentType":
"application/json","mode":"OUTPUT","data":"{\n  \"predictions\": [[-1.5448699,
-2.04928493, -1.71272779, 0.172660828, 4.16521502], [3.47782397, -0.372900903,
-1.70296526, -1.61752951, -0.441117555]\n
]\n}","encoding":"JSON"}},"eventMetadata":{"eventId":"e2a31d52-5685-4105-9ee1-
cc28d4be7c91","customAttributes":["tfs-model-
name=saved_model"],"inferenceTime":"2020-05-17T03:18:26Z"},"eventVersion":"0"}
```


Create Baseline Statistics and Constraints for Features

A baseline helps us detect drift of input features relative to the features used to train the model. We typically use our training dataset to create the first baseline as shown in Figure 7-16.

Figure 7-16. Create a baseline

The training dataset schema and the inference dataset schema must match exactly including the number of features and the order in which they are

passed in for inference. We can now start a SageMaker ProcessingJob to suggest a set of baseline constraints and generate statistics of the data with

```
DefaultModelMonitor.suggest_baseline():
from sagemaker.model_monitor import DefaultModelMonitor
from sagemaker.model_monitor.dataset_format import DatasetFormat
my_default_monitor = DefaultModelMonitor(
  role=role,
  instance_count=1,
  instance_type='ml.m5.xlarge',
  volume_size_in_gb=20,
  max_runtime_in_seconds=3600,
)
my_default_monitor.suggest_baseline(
baseline_dataset='s3://my_bucket/path/some.csv',
        dataset_format=DatasetFormat.csv(header=True),
        output_s3_uri='s3://my_bucket/output_path/',
        wait=True
)
```

After the baseline job has finished, you can explore the generated statistics as follows:

```
import pandas as pd
baseline_job = my_default_monitor.latest_baselining_job
statistics = pd.io.json.json_normalize(
baseline_job.baseline_statistics().body_dict["features"])
```

Here is an example set of statistics for our `review_body` prediction inputs:

```
"name" : "Review Body",
  "inferred_type" : "String",
  "numerical_statistics" : {
   "common" : {
    "num_present" : 1420,
    "num_missing" : 0
   }, "data" : [ [ "I love this item.", "This item is OK", … ] ]
```

You can explore the generated constraints as follows:

```
constraints = pd.io.json.json_normalize(
baseline_job.suggested_constraints().body_dict["features"])
```

Here is an example of the constraints defined for our `review_body` prediction inputs:

```
{
  "name" : "Review Body",
  "inferred_type" : "String",
  "completeness" : 1.0
}
```

> **NOTE**
>
> Under the hood, Model Monitor runs a SageMaker Processing Job on your behalf similar to the Processing Job that we used to analyze the statistics of the Amazon Customer Reviews Dataset.

We now have the foundation to schedule monitoring jobs that continuously check our real-time model performance against the created baseline.

Schedule Monitoring Jobs

SageMaker Model Monitor gives us the ability to continuously monitor the data collected from the endpoints on a schedule. You can create the schedule with the `CreateMonitoringSchedule` API defining a periodic interval. Similar to the baseline job, Model Monitor starts a SageMaker ProcessingJob which compares the dataset for the current analysis with the baseline statistics and constraints. The result is a violation report. In addition, Model Monitor sends metrics for each feature to Amazon CloudWatch as shown in Figure 9.17.

Training Data
Amazon SageMaker Training Job
Model
Amazon SageMaker Endpoint
Applications
Create baseline processing job
Baseline statistics and constraints
Schedule monitoring job
Inference requests, prediction results
Results: Statistics and violations
Amazon CloudWatch metrics
Schedule monitoring jobs

Figure 7-17. Schedule model monitoring jobs

You can create a model monitoring schedule for an endpoint with `my_default_monitor.create_monitoring_schedule()`. In the configuration of the monitoring schedule, you point to the baseline statistics and constraints and define a cron schedule.

```
from sagemaker.model_monitor import DefaultModelMonitor
from sagemaker.model_monitor import CronExpressionGenerator
mon_schedule_name = 'my-model-monitor-schedule'
my_default_monitor.create_monitoring_schedule(
  monitor_schedule_name=mon_schedule_name,
  endpoint_input=predictor.endpoint,
  output_s3_uri=s3_report_path,
  statistics=my_default_monitor.baseline_statistics(),
  constraints=my_default_monitor.suggested_constraints(),
  schedule_cron_expression=CronExpressionGenerator.hourly(),
  enable_cloudwatch_metrics=True,
)
```

Model Monitor now runs at the scheduled intervals and analyzes the captured data against the baseline. The job creates a violation report and stores the report in Amazon S3, along with a statistics report for the collected data.

Once the monitoring job has started its executions, you can use `list_executions()` to view them:

```
executions = my_monitor.list_executions()
```

You can also view the status of a specific execution, e.g. the latest monitoring job:

```
latest_execution = executions[-1]
latest_execution.describe()['ProcessingJobStatus']
latest_execution.describe()['ExitMessage']
```

The Model Monitor jobs should exit with one of the following status:

Completed

This means the monitoring execution completed and no issues were found in the violations report.

CompletedWithViolations

This means the execution completed, but constraint violations were detected.

Failed

The monitoring execution failed, maybe due to client error (such as incorrect role permissions) or infrastructure issues.

Stopped

The job exceeded the max runtime or was manually stopped. To troubleshoot further, you can check `FailureReason` and `ExitMessage`.

> **NOTE**
>
> You can create your own custom monitoring schedules and procedures using preprocessing and postprocessing scripts. You can also build your own analysis container.

Interpret Results

With the monitoring data collected and continuously compared against the model baseline, we are now in a much better position to make decisions about how to improve the model. Depending on the model monitoring results, you might decide to retrain and redeploy the model. In this final step, we visualize and interpret the results as shown in Figure 7-18.

Training Data
Amazon SageMaker Training Job
Model
Amazon SageMaker Endpoint
Applications
Create baseline processing job
Baseline statistics and constraints
Schedule monitoring job
Inference requests, prediction results
Results: Statistics and violations
Amazon CloudWatch metrics
Notifications
Analysis of results
Interpret results

Figure 7-18. Interpret results.

Let's query for the location for the generated reports.

```
report_uri=latest_execution.output.destination
print('Report Uri: {}'.format(report_uri))
```

Next, we can list the generated reports:

```
from urllib.parse import urlparse
s3uri = urlparse(report_uri)
report_bucket = s3uri.netloc
report_key = s3uri.path.lstrip('/')
print('Report bucket: {}'.format(report_bucket))
print('Report key: {}'.format(report_key))
s3_client = boto3.Session().client('s3')
result = s3_client.list_objects(Bucket=report_bucket,
   Prefix=report_key)
report_files = [report_file.get("Key") for report_file in
        result.get('Contents')]
print("Found Report Files:")
print("\n ".join(report_files))
```

Output:

```
s3://<bucket>/<prefix>/constraint_violations.json
s3://<bucket>/<prefix>/constraints.json
s3://<bucket>/<prefix>/statistics.json
```

We already looked at `constraints.json` and `statistics.json`, so let's analyze the violations:

```
violations =
my_default_monitor.latest_monitoring_constraint_violations()
violations = pd.io.json.json_normalize(
violations.body_dict["violations"])
```

Here are example violations for our `review_body` inputs:

```
 {
  "feature_name" : "Review Body",
  "constraint_check_type" : "data_type_check",
  "description" : "Value: 1.0 meets the constraint requirement"
```

```
 }, {
  "feature_name" : "Review Body",
  "constraint_check_type" : "baseline_drift_check",
  "description" : "Numerical distance: 0.2711598746081505 exceeds numerical
threshold: 0"
 }
```


To find the root cause of this drift, we want to examine the prediction inputs and examine any upstream application bugs (or features) that may have been recently introduced. Perhaps the application team started to feature emojis as the primary review mechanism - and our language model was not trained on emojis. In this case, we want to quickly retrain our model with the latest reviews that include the emojis.

Visualize Results in Amazon SageMaker Studio

Amazon SageMaker Studio has built-in capabilities to visualize our model monitoring results. Simply select the endpoint you are monitoring, and click on the Monitoring results tab as shown in Figure 7-19.

Figure 7-19. Visualize model monitoring results in SageMaker Studio.

Now that we have detailed monitoring of our models in place, we can build additional automation. We could leverage the Model Monitor integration into CloudWatch to trigger actions on baseline drift alarms, such as model updates, training data updates or an automated re-training of our model.

Perform Batch Predictions with SageMaker Batch Transform

Amazon SageMaker Batch Transform allows you to make predictions on batches of data in S3 without setting up a REST endpoint. Batch predictions are also called “offline” predictions since they do not require an online REST endpoint. Typically meant for higher-throughput workloads that can tolerate higher latency and lower freshness, batch prediction servers typically do not run 24 hours per day like real-time prediction servers. They run for a few hours on a batch of data, then shut down - hence the term, “batch.” Batch Transform manages all of the resources needed to perform the inferences including the launch and termination of the cluster after the job completes.

For example, if your movie catalog only changes a few times a day, you can likely just run one(1) batch prediction job each night that uses a new recommendation model trained with the days’ new movies and user activity. Since you are only updating the recommendations once in the evening, your recommendations will be a bit stale throughout the day. However, your overall cost is minimized and, even more importantly, stays predictable.

The alternative is to continuously re-train and re-deploy new recommendation models throughout the day with every new movie that joins or leaves your movie catalog. This could lead to excessive model training and deployment costs that are difficult to control and predict. These types of continuous updates typically fall under the “trending now” category of popular websites like Facebook and Netflix that offer real-time content recommendations. We explore these types of continuous models when we discuss streaming data analytics.

Selecting an Instance Type

Similar to model training, the choice of instance type often involves a balance between latency, throughput, and cost. Always start with a small instance type and then increase only as needed. Batch predictions may benefit from GPUs more so than real-time endpoint predictions since GPUs perform much better with large batches of data. However, it is recommended that you first try CPU instances to set the baseline for

latency, throughput, and cost. Here, we are using a cluster of high-CPU instances.

```
instance_type='ml.c5.18xlarge'
instance_count=5
```

Setup the Input Data

Let's specify the input data. In our case, we are using the original TSV's that are stored as gzip compressed text files.

```
# Specify the input data
input_csv_s3_uri =
's3://{}/amazon-reviews-pds/tsv/'.format(bucket)
```

We specify `MultiRecord` for our strategy to take advantage of our multiple CPUs. We specify `Gzip` as the compression type since our input data is compressed using gzip. We're using TSV's, so `text/csv` is a suitable `accept_type` and `content_type`. And since our rows are separated by line breaks, we use `Line` for `assemble_with` and `split_type`.

```
strategy='MultiRecord'
compression_type='Gzip'
accept_type='text/csv'
content_type='text/csv'
assemble_with='Line'split_type='Line'
```

Tune the Batch Transformation Configuration

When we start the batch transformation job, your code runs in an HTTP server inside the TensorFlow Serving inference container. Note that TensorFlow Serving natively supports batches of data on a single request.

Let's leverage TensorFlow Serving's built-in batching feature to batch multiple records to increase prediction throughput - especially on GPU instances that perform well on batches of data. Set the following environment variables to enable batching:

```
batch_env = {
 # Configures whether to enable record batching.
 'SAGEMAKER_TFS_ENABLE_BATCHING': 'true',
 # Name of the model - this is important in multi-model deployments
 'SAGEMAKER_TFS_DEFAULT_MODEL_NAME': 'saved_model',
 # Configures how long to wait for a full batch, in microseconds.
 'SAGEMAKER_TFS_BATCH_TIMEOUT_MICROS': '50000', # microseconds
 # Corresponds to "max_batch_size" in TensorFlow Serving.
 'SAGEMAKER_TFS_MAX_BATCH_SIZE': '10000',
 # Number of seconds for the SageMaker web server timeout
 'SAGEMAKER_MODEL_SERVER_TIMEOUT': '3600', # Seconds
 # Configures number of batches that can be enqueued.
 'SAGEMAKER_TFS_MAX_ENQUEUED_BATCHES': '10000'
}
```

Prepare the Batch Transformation Job

We can inject pre-processing and post-processing code directly into the Batch Transform Container to customize the prediction flow. The pre-processing code is specified in `inference.py` and will transform the request from raw data (i.e., `review_body` text) machine-readable features (i.e., BERT tokens). These features are then fed to the model for inference. The model prediction results are then passed through the post-processing code from `inference.py` to convert the model prediction into human-readable responses before saving to S3. Figure 7-20 shows how SageMaker Batch Transform works in detail.

Figure 7-20. Offline predictions with SageMaker Batch Transform (Source: Amazon SageMaker Developer Guide).

Let's set up the batch transformer to use our `inference.py` script that we will show in a bit. We are specifying the S3 location of the classifier model that we trained in a previous chapter.

```
model_s3_uri = 's3://{}/{}/output/model.tar.gz'.format(bucket,
training_job_name)
batch_model = Model(entry_point='inference.py',
          source_dir='src_tsv',
          model_data=model_s3_uri,
          role=role,
          framework_version='2.1.0',
          env=batch_env)
batch_predictor = batch_model.transformer(
strategy=strategy,
instance_type=instance_type,
instance_count=instance_count,
accept=accept_type,
assemble_with=assemble_with,
max_concurrent_transforms=max_concurrent_transforms,
max_payload=max_payload, # This is in Megabytes
env=batch_env)
```

Below is the `inference.py` script used by the Batch Transformation Job defined above. This script has an `input_handler` for request processing and `output_handler` for response processing as shown in Figure 7-21.

Figure 7-21. Pre-Processing Request Handler and Post-Processing Response Handler

We start with the `input_handler` function. Note that the code looks very similar to the code we used in the feature engineering chapter. The `input_handler` function converts batches of raw text into BERT tokens using the Transformer library.

```
import json
import tensorflow as tf
from transformers import DistilBertTokenizer
review_body_column_idx_tsv = 13
classes=[1, 2, 3, 4, 5]
max_seq_length=128
tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-uncased')
def input_handler(data, context):
  transformed_instances = []
  for instance in data:
    data_str = instance.decode('utf-8')
    data_str_split = data_str.split('\t')
    print(len(data_str_split))
    if (len(data_str_split) >= review_body_column_idx_tsv):
      print(data_str_split[review_body_column_idx_tsv])
    text_input = data_str_split[review_body_column_idx_tsv]
    tokens = tokenizer.tokenize(text_input)
    encode_plus_tokens = tokenizer.encode_plus(text_input,
                         pad_to_max_length=True,
                         max_length=max_seq_length)
    # Convert the text-based tokens to ids from the pre-trained BERT vocabulary
    input_ids = encode_plus_tokens['input_ids']
    # Specifies which tokens BERT should pay attention to (0 or 1)
    input_mask = encode_plus_tokens['attention_mask']
    # Segment Ids are always 0 for single-sequence tasks (or 1 if two-sequence
tasks)
    segment_ids = [0] * max_seq_length

    transformed_instance = {
                "input_ids": input_ids,
                "input_mask": input_mask,
                "segment_ids": segment_ids
               }

    transformed_instances.append(transformed_instance)
  transformed_data = {"instances": transformed_instances}  return
json.dumps(transformed_data)
```

SageMaker then passes this batched output from the `input_handler` into our model which produces batches of predictions. The predictions are passed through the `output_handler` function which converts the prediction into a json response. SageMaker then joins each prediction within a batch to its specific line of input. This produces a single, coherent line of output for each row that was passed in.

```
def output_handler(response, context):
  response_json = response.json()
  log_probabilities = response_json["predictions"]
  predicted_classes = []
  for log_probability in log_probabilities:
    softmax = tf.nn.softmax(log_probability)
    predicted_class_idx = tf.argmax(softmax, axis=-1, output_type=tf.int32)
    predicted_class = classes[predicted_class_idx]
    predicted_classes.append(predicted_class)
  predicted_classes_json = json.dumps(predicted_classes)
  response_content_type = context.accept_header  return predicted_classes_json,
response_content_type
```

Run the Batch Transformation Job

Next we will specify the input data and start the actual batch transformation job. Note that our input data is compressed using Gzip as Batch Transformation Jobs support many types of compression.

```
batch_predictor.transform(data=input_csv_s3_uri,
             split_type=split_type,
             compression_type=compression_type,
             content_type=content_type,
                                 join_source='Input',
             experiment_config=None,
             wait=False)
```

We specify `join_source='Input'` to force SageMaker to join our prediction with the original input before writing to S3. And while not shown here, SageMaker lets you specify the exact input features to pass into this batch transformation process using `InputFilter` and the exact data to write to S3 using `OutputFilter`. This helps to reduce overhead, reduce cost, and improve batch prediction performance.

If you are using `join_source='Input'` and `InputFilter` together, SageMaker will join the original inputs - including the filtered-out inputs - with the predictions to keep all of the data together. You can also filter the outputs to reduce the size of the prediction files written to S3. The whole flow is shown in Figure 7-22.

Input data
Features
ID
{"features":[1,2,3], "ID":1}
InputFilter
InputFilter = "$.features"
[1,2,3]
Model
{"labels":"abc"}
JoinSource
"JoinSource":"Input"
OutputFilter
OutputFilter =
"${'ID', 'SageMakerOutput'}"
Joined data
Features
ID
Labels
SageMakerOutput
{
"features":[1,2,3],
"ID":1,
"SageMakerOutput":{"labels":"abc"}
}
Output data
ID
Labels
SageMakerOutput
{
"ID":1,
"SageMakerOutput":{"labels":"abc"}
}

Figure 7-22. : Filtering and Joining Inputs to Reduce Overhead and Improve Performance

Review the Batch Predictions

Once the Batch Transform Job completes, we can review the generated comma-separated `.out` files that contain our `review_body` inputs and `star_rating` predictions as shown here:

```
amazon_reviews_us_Digital_Software_v1_00.tsv.gz.out
amazon_reviews_us_Digital_Video_Games_v1_00.tsv.gz.out
```

Here are a few sample predictions:

```
'This is the best movie I have ever seen', 5, 'Star Wars'
'This is an ok, decently-funny movie.', 3, 'Spaceballs'
'This is the worst movie I have ever seen', 1, 'Star Trek'
```

At this point, we have performed a large number of predictions and generated comma-separated output files. With a little bit of application code (SQL, Python, Java, etc), we can use these predictions to power natural-language-based applications to improve the customer service experience, for example.

Lambda Functions and API Gateway

You can also deploy your models as serverless APIs with AWS Lambda. When a prediction request arrives, the Lambda function loads the model from S3 and executes the inference function code. You can use the "provisioned concurrency" feature of Lambda to pre-load the model into the function and greatly improve prediction latency. Amazon API Gateway provides additional support for application authentication, authorization, caching, rate-limiting, and web application firewall (WAF) rules. Figure 7-23 shows how you implement serverless inference with AWS Lambda and API Gateway

Figure 7-23. Serverless inference with AWS Lambda and API Gateway

> **NOTE**
>
> At the time of this writing, AWS Lambda functions had some limitations including no GPU support and limited model size. If your model is not affected by those restrictions, AWS Lambda is a viable model deployment option.

Reduce Costs and Increase Performance

In this section, we describe multiple ways to reduce cost and increase performance by packing multiple models into a single SageMaker deployment container, utilizing GPU-based Elastic Inference Adapters, optimizing your trained model for specific hardware, and utilizing inference-optimized hardware such as the Amazon Inferentia chip.

Deploy Multiple Models in One Container

If you have a large number of similar models that you can serve through a shared serving container - and don't need to access all the models at the same time - you can use SageMaker's multi-model jendpoint (MME) capability. When there is a long tail of ML models that are infrequently accessed, using one multi-model endpoint can efficiently serve inference traffic and enable significant cost savings.

Multi-model endpoints can automatically load and unload models based on traffic and resource utilization. For example, if traffic to model1 goes to zero and model2 traffic spikes, SageMaker will dynamically unload model1 and load another instance of model2.

While MME lets you deploy multiple models to a single endpoint and serve them using a single container, you can invoke a specific model by specifying the target model name as a parameter in your prediction request as shown in Figure 7-24.

Figure 7-24. Invoke a specific model with multi-model endpoints

Here is the code to deploy two(2) TensorFlow models behind a single endpoint. The most human-grokkable use case here is when a practitioner has trained multiple for slightly differently audiences (ie. Audi lovers model1, Porsche lovers model2), but wants to deploy them to the same endpoint for convenience and cost purposes.

For TensorFlow, we need to package the models as follows:

```
└── multi
  ├── model1
```

```
  |   └── <version number>
  |     ├── saved_model.pb
  |     └── variables
  |       └── ...
  └── model2
    └── <version number>
      ├── saved_model.pb
      └── variables
        └── ...
from sagemaker.tensorflow.serving import Model, Predictor
# For multi-model endpoints, you should set the default
#  model name in this environment variable.
# If it isn't set, the endpoint will work, but the model
#  it will select as default is unpredictable.
env = {
 'SAGEMAKER_TFS_DEFAULT_MODEL_NAME': 'model1' # <== This must match the directory
}
model_data = '{}/multi.tar.gz'.format(multi_model_s3_uri)
model = Model(model_data=model_data,
       role=role,
       framework_version='2.1.0',        env=env)
```

Attach a GPU-based Elastic Inference Accelerator

Elastic Inference Accelerator (EIA) is a low-cost, dynamically-attached, GPU-powered add-on for SageMaker instances. While standalone GPU instances are a good fit for model training on large datasets, they are typically oversized for smaller-batch inference requests which consume small amounts of GPU resources.

While AWS offers a wide range of instance types with different GPU, CPU, network bandwidth, and memory combinations, your model may use a custom combination. With EIA's, you can start by choosing a base CPU instance and add GPUs until you find the right balance for your model inference needs. Otherwise, you may be forced to optimize one set of resources like CPU and RAM, but underutilize other resources like GPU and network bandwidth.

Here is the code to deploy our same model but with an Elastic Inference Accelerator (EIA).

```
import time
```

```
timestamp = '{}'.format(int(time.time()))
endpoint_config_name = '{}-{}'.format(training_job_name, timestamp)
variantA = production_variant(model_name='ModelA',
instance_type="ml.m5.large",
                              initial_instance_count=1,
                              variant_name='VariantA',
                              initial_weight=50,
   accelerator_type='ml.eia2.medium')
variantB = production_variant(model_name='ModelB',
                              instance_type="ml.m5.large",
                              initial_instance_count=1,
                              variant_name='VariantB',
                              initial_weight=50)
endpoint_config = sm.create_endpoint_config(
  EndpointConfigName=endpoint_config_name,
  ProductionVariants=[variantA, variantB]
)
endpoint_name = '{}-{}'.format(training_job_name, timestamp)
endpoint_response = sm.create_endpoint(
  EndpointName=endpoint_name,
  EndpointConfigName=endpoint_config_name)
```

Optimize a Trained Model with SageMaker Neo and TensorFlow Lite

SageMaker Neo takes a trained model and performs a series of hardware-specific optimizations such as 16-bit quantization, graph pruning, layer fusing, and constant folding for instant 2x model-prediction speedups with minimal accuracy loss. Neo works across popular AI and machine learning frameworks including TensorFlow, PyTorch, MXNet, and XGBoost.

Neo parses the model, optimizes the graph, quantizes tensors, and generates hardware-specific code for a variety of target environments including Intel x86 CPUs, Nvidia GPUs, and Amazon Inferentia as shown in Figures 9-25.

Figure 7-25. Parsing Models, Optimizing Graphs and Tensors, and Generating Hardware-Specific Code

SageMaker Neo supports TensorFlow Lite (TF Lite), a highly-optimized TensorFlow code generator. Neo uses the TF Lite converter to perform hardware-specific optimizations for the The TensorFlow Lite interpreter as shown in Figure 7-26.

File format
Infrastructure
Data Type
Backend
TensorFlow APIs
tf.Keras
Low level API
tf.Keras model
SavedModel
Concrete Function(s)
TFLite Converter
TFLite Flatbuffer
TFLite interpreter
...
NN API
GPU
CPU
server
client

Figure 7-26. TF Lite interpreter (Source: https://www.tensorflow.org/lite/convert)

Here is the TF Lite code that performs 16-bit quantization on a TensorFlow model.

```
import tensorflow as tf
converter = tf.lite.TocoConverter.from_saved_model('./tensorflow/')
converter.post_training_quantize = True
tflite_model = converter.convert()
tflite_model_path = './tflite/tflite_optimized_model.tflite'
model_size = open(tflite_model_path, "wb").write(tflite_model)
```

Here's the prediction code that leads to an order-of-magnitude speed up in prediction time due to the quantization:

```
import numpy as np
import tensorflow as tf
# Load TFLite model and allocate tensors.
interpreter = tf.lite.Interpreter(model_path=tflite_model_path)
interpreter.allocate_tensors()
# Get input and output tensors.
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
# Test model on random input data.
input_shape = input_details[0]['shape']
input_data = np.array(np.random.random_sample(input_shape),
dtype=np.float32)
interpreter.set_tensor(input_details[0]['index'], input_data)
interpreter.invoke()
output_data = interpreter.get_tensor(output_details[0]['index'])
print('Prediction: %s' % output_data)
```

Use Inference-Optimized Hardware

AWS Inferentia is an inference-optimized chip designed to accelerate 16-bit and 8-bit floating point operations generated by SageMaker Neo's Inferentia-specific optimizer called Neuron as shown in Figure 7-27.

Figure 7-27. SageMaker Neo's Neuron Optimizer for AWS Inferentia Chip

Summary

In this chapter, we moved our models out of the research lab and into the end-user application domain. We can now measure, and improve, and deploy our models using real-world, production-ready fundamentals such as canary rollouts, blue/green deployments, A/B tests, drift detection, and lineage tracking. In addition, we performed batch transformations to improve throughput for offline model predictions. We closed out with tips on how to reduce cost and improve performance using SageMaker Neo, TensorFlow Lite, SageMaker Multi-Model Endpoints and inference-optimized hardware such as Elastic Inference Accelerator (EIA) and Amazon Inferentia.

Chapter 8. Human-in-the-Loop Workflows

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 12th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *support@pipeline.ai*.

In this chapter, we discuss the concept of Augmented AI and active learning. We describe two services, Amazon Augmented AI (A2I) and Amazon SakeMaker Ground Truth that demonstrate how humans and AI can work successfully together. A2I enables machine learning practitioners to integrate humans review workflows into their applications. SageMaker Ground Truth leverages human workforces combined with an active learning approach to create accurate training datasets.

Building Human-in-the-Loop Workflows with Amazon Augmented AI

While AI and machine learning services make our lives easier, humans are far from being obsolete. In fact, the concept of "human-in-the-loop" has emerged as an important cornerstone in many AI/ML workflows. Humans

provide necessary quality assurances before pushing sensitive or regulated models into production. You can also leverage human intelligence by "crowd sourcing" the data labels used by your model training workflows.

Amazon Augmented AI (A2I) is a fully-managed service to develop human-in-the-loop workflows that include a clean user interface, role-based access control with IAM, and scalable data storage with S3. A2I is integrated with many Amazon services including Rekognition for content moderation and Amazon Textract for form-data extraction. You can also use A2I with Amazon SageMaker and any of your custom ML models.

A2I introduces a few new terms: Worker Task Template, Flow Definition and Human Loop. The Worker Task Template defines the Human Task UI for the worker. This UI displays input data, such as documents or images, and instructions to workers. The Flow Definition defines the human review workflow. The definition contains the chosen workforce and provides information about how to accomplish the review task. The Human Loop represents the actual human review workflow. Once the human loop is triggered, A2I sends the human review tasks to the workers as specified in the flow definition. Let's start with the creation of a private workforce.

Amazon Comprehend is a fully-managed natural language processing (NLP) service that uses machine learning to find insights and relationships in text. In the following example we combine a Comprehend Custom text classifier model with an Augmented AI workflow to review low-confidence predictions. We use the Amazon customer reviews dataset as our sample data.

We will show how to work with custom machine learning models in the model training chapter. in Figure 8-1 illustrates the general A2I workflow to review model predictions with Amazon Comprehend.

Figure 8-1. Amazon Augmented AI (A2I) workflow.

Amazon Comprehend receives input data in a prediction request. You define a confidence threshold which decides when to involve the human reviewers. If the model's prediction meets the confidence threshold, A2I will send the prediction result directly to the client application. In case the model is unable to make a high-confidence prediction, A2I sends the task to human reviewers. You can also set up the workflow for human reviewers to audit the predictions on an ongoing basis. A2I consolidates the human reviewer results and sends the final prediction response to your client application. A2I can also store the human review results as in S3 which you can leverage to re-train any custom ML model.

With Augmented AI, you can choose between a public or private workforce. Public workforces integrate with the Amazon Mechanical Turk service with hundreds of thousands of human labelers that have been pre-screened by Amazon. You can also use third-party, pre-screened workforce providers listed on the AWS Marketplace. You can also create private workforces and invite your co-workers or employees to join the workforce team.

In the next sections, we show you how to create a private workforce and develop A2I human-in-the-loop workflows.

Create a Private Workforce

You can create a private workforce easily in the AWS Console. Simply navigate to Amazon SageMaker Ground Truth and click on `Labeling workforces`. Select `Private` and `Create privat team` as shown in Figure 8-2.

Figure 8-2. Amazon SakeMaker Ground Truth > Labeling workforces > Private

The work *team* refers to a group of workers from your private workforce that are assigned to Amazon Augmented AI human review tasks. You can have multiple work teams and can assign one or more work teams to each job.

In the dialog that opens, provide a team name, select `Invite new workers by email` and specify the email addresses of workers you want to add to this team, as shown in Figure 8-3. You can also set up an Amazon Simple Notification Service (SNS) topic which ensures our workers receive an email for every assigned review task.

Amazon SageMaker > Labeling workforces > Create private team

Create private team

Team details

Team name
Give your work team descriptive name. This name can't be changed later.

dsoaws

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

Add workers Info

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

- (selected) Invite new workers by email
- Import workers from existing Amazon Cognito user groups

Email addresses

Preview invitation

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

antje.barth@[redacted]com

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

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

Data Science on AWS

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

antje.barth@[redacted]com

Enable Notifications

SNS topic - *optional*
Configuring SNS topic enables your work team to receive notifications on available work. Learn more

AmazonSageMaker-A2I-DSOAWS

Cancel | Create private team

Figure 8-3. Create private team.

You can preview the worker invitation email by clicking on `Preview invitation`. Figure 8-4 shows a sample invitation.

Figure 8-4. Sample worker invitation email.

Once done, click `Create private team`. In the UI, you should see the private workforce and team successfully created as shown in Figure 8-5.

Private team 'dsoaws' was successfully created.

Amazon SageMaker > Labeling workforces

Amazon Mechanical Turk | Private | Vendor

Private workforce summary Info

Your workforce is powered by an Amazon Cognito user pool. With a user pool, your workers can sign in to the labeling portal through Amazon Cognito.

Amazon Cognito user pool

us-west-2_pnVZgLFNc

App client

2iu6hgt8lgapfng0mm0800h3k2

Labeling portal sign-in URL

https://dbz39ez1vg.labeling.us-west-2.sagemaker.aws

Private teams (1) Info

Delete | Create private team

A team of workers from your private workforce. You can create different private teams for your labeling jobs.

Search private teams by name

1

Name	ARN	Creation time
dsoaws	arn:aws:sagemaker:us- :workteam/private-crowd/dsoaws	May 20, 2020, 4:15 PM UTC

Workers Info

Enable | Disable | Delete | Invite new workers

All workers in your private workforce

Search workers by email

1 ...

Email	Status	Cognito status	Enabled	Username
antje.barth@	Invitation sent	Force_change_password	Yes	antje.barth@

Figure 8-5. Private workforce and team have been created.

Make note of the `Private team ARN` which uniquely identifies our work team. We need this resource name in the workflow setup.

In the meantime, the workers should have received the invitation email to join the work team as shown in Figure 8-6.

You're invited by Data Science on AWS to work on a labeling project. Inbox x

no-reply@verificationemail.com

to antje.barth ▾

You're invited to work on a labeling project.

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

User name: antje.barth@

Temporary password: **c@UVPo42**

Open the link below to log in:

https://dbz39ez1vg.labeling.us-west-2.sagemaker.aws

After you log in with your temporary password, you are required to create a new one. If you have any questions, please contact antje.barth@

Figure 8-6. Invitation email to join a labeling project.

The worker needs to follow the instructions to accept the invitation, activate the account and reset the temporary password. Next, we choose the Human Task UI.

Choose a Human Task UI

The Human Task UI is the HTML page our workers see when they log into the review system to perform their task. The UI presents the input data and instructions for the workers. You can completely customize the page to your needs, or grab one of the pre-built Task UIs [https://github.com/aws-samples/amazon-a2i-sample-task-uis].

For our text classification example, we use the following HTML template which is compatible with our Comprehend Custom model inputs/outputs. The model has been trained on the Amazon Customer Reviews Data which we introduced in the previous chapter and predicts the star rating (1-5) given a sample review.

```
template = r"""
<script src="https://assets.crowd.aws/crowd-html-elements.js"></script>
<crowd-form>
    <crowd-classifier
      name="star_rating"
      categories="['1', '2', '3', '4', '5']"
      initial-value="{{ task.input.initialValue }}"
      header="Classify Reviews into Star Ratings Between 1 (Worst) and 5 (Best)"
    >
      <classification-target>
        {{ task.input.taskObject }}
      </classification-target>

      <full-instructions header="Instructions to Classify Reviews into Star
Ratings Between 1 (Worst) and 5 (Best)">
        <p><strong>5</strong>: joy, excitement, delight</p>
        <p><strong>4</strong>: neutral-to-good</p>
        <p><strong>3</strong>: neither positive or negative, such as stating a
fact</p>
        <p><strong>2</strong>: neutral-to-bad</p>
        <p><strong>1</strong>: anger, sarcasm, anxiety</p>
      </full-instructions>
```

```
      <short-instructions>
       Classify Reviews into Star Ratings Between 1 (Worst) and 5 (Best)
      </short-instructions>
    </crowd-classifier>
</crowd-form>
"""
```

Next, we create a Task UI based on the HTML template.

```
# Task UI name - this value is unique per account and region.
taskUIName = 'ui-comprehend-' + str(uuid.uuid4())
# Create a Human Task UI resource.
humanTaskUiResponse = sagemaker.create_human_task_ui(
        HumanTaskUiName=taskUIName,
        UiTemplate={'Content': template})
humanTaskUiArn = humanTaskUiResponse['HumanTaskUiArn']
print(humanTaskUiArn)
```

Again, note down the ARN for the Task UI.

Create a Flow Definition

The Flow Definition defines the actual human review workflow. The definition points to a specific workforce team and Human Task UI. You can also specify the number of workers that receive a task and the S3 location to store the output data.

```
import uuid
# Flow definition name - this value is unique per account and region.
flowDefinitionName = 'fd-dsoaws-comprehend-' + str(uuid.uuid4())
create_workflow_definition_response = sagemaker.create_flow_definition(
        FlowDefinitionName= flowDefinitionName,
        RoleArn= role,
        HumanLoopConfig= {
            'WorkteamArn': WORKTEAM_ARN,
            'HumanTaskUiArn': humanTaskUiArn,
            'TaskCount': 1,
            'TaskDescription': 'Classify Reviews into Star Ratings Between 1
(Worst) and 5 (Best)',
            'TaskTitle': 'Classify Reviews into Star Ratings Between 1 (Worst)
and 5 (Best)'
        },
        OutputConfig={
            'S3OutputPath' : output_path
        }
```

```
    )
flowDefinitionArn = create_workflow_definition_response['FlowDefinitionArn']
```

We are now ready to start our human loop.

Start a Human Loop for Low-Confidence Predictions

The human loop is the actual human review workflow. Let's define some sample product reviews which we send to Amazon Comprehend:

```
sample_reviews = [
                  'I enjoy this product',
                  'I am unhappy with this product',
                  'It is okay',
                  'sometimes it works'
                 ]
```

We also define a prediction confidence score threshold of 70%. If our model returns a prediction with a lower confidence score, A2I will trigger the human loop and our workforce team receives a task. You can define any confidence score threshold value that makes sense for your use case.

```
human_loops_started = []
CONFIDENCE_SCORE_THRESHOLD = 0.70
for sample_review in sample_reviews:
    # Call the Comprehend Custom model
    response = comprehend.classify_document(
Text=sample_review,
EndpointArn=comprehend_endpoint_arn)
    star_rating = response['Classes'][0]['Name']
    confidence_score = response['Classes'][0]['Score']

    print(f'Processing sample_review: \"{sample_review}\"')

    # Our condition for when we want to engage a human for review
    if (confidence_score < CONFIDENCE_SCORE_THRESHOLD):

        humanLoopName = str(uuid.uuid4())
        inputContent = {
            'initialValue': star_rating,
            'taskObject': sample_review
        }
        start_loop_response = a2i.start_human_loop(
            HumanLoopName=humanLoopName,
            FlowDefinitionArn=flowDefinitionArn,
```

```
            HumanLoopInput={
                'InputContent': json.dumps(inputContent)
            }
        )
        human_loops_started.append(humanLoopName)
        print(f'Confidence score of {confidence_score} for star rating of
{star_rating} is less than the threshold of {CONFIDENCE_SCORE_THRESHOLD}')
        print(f'*** ==> Starting human loop with name: {humanLoopName}  \n')
    else:
        print(f'Confidence score of {confidence_score} for star rating of
{star_rating} is above threshold of {CONFIDENCE_SCORE_THRESHOLD}')
        print('No human loop created. \n')
```

If we run this code, we will see the following responses:

```
Processing sample_review: "I enjoy this product"
Confidence score of 0.8727718591690063 for star rating of 3 is above threshold of
0.7
No human loop created.
Processing sample_review: "I am unhappy with this product"
Confidence score of 0.5123735070228577 for star rating of 2 is less than the
threshold of 0.7
*** ==> Starting human loop with name: 72331439-0df9-4190-a42b-3e4048efb0a9
Processing sample_review: "It is okay"
Confidence score of 0.9679936170578003 for star rating of 4 is above threshold of
0.7
No human loop created.
Processing sample_review: "sometimes it works"
Confidence score of 0.6361567974090576 for star rating of 3 is less than the
threshold of 0.7
*** ==> Starting human loop with name: e7994a4c-57bf-4578-aa27-dc5fb8c11d36
```

We see that two predictions didn’t meet our confidence threshold and started human loops. Let’s check the status of our human loops:

```
completed_human_loops = []
for human_loop_name in human_loops_started:
    resp = a2i.describe_human_loop(HumanLoopName=human_loop_name)
    print(f'HumanLoop Name: {human_loop_name}')
    print(f'HumanLoop Status: {resp["HumanLoopStatus"]}')
    print(f'HumanLoop Output Destination: {resp["HumanLoopOutput"]}')
    print('')

    if resp['HumanLoopStatus'] == 'Completed':
        completed_human_loops.append(resp)
```

The output should look similar to this:

```
HumanLoop Name: 72331439-0df9-4190-a42b-3e4048efb0a9
HumanLoop Status: InProgress
HumanLoop Output Destination: {'OutputS3Uri': 's3://sagemaker-<region>-
<account_id>/a2i-comprehend-star-rating-results/fd-dsoaws-comprehend-fb987de9-
90f9-4304-b777-9bd49c41a78d/2020/05/20/19/13/37/72331439-0df9-4190-a42b-
3e4048efb0a9/output.json'}
HumanLoop Name: e7994a4c-57bf-4578-aa27-dc5fb8c11d36
HumanLoop Status: InProgress
HumanLoop Output Destination: {'OutputS3Uri': 's3://sagemaker-<region>-
<account_id>/a2i-comprehend-star-rating-results/fd-dsoaws-comprehend-fb987de9-
90f9-4304-b777-9bd49c41a78d/2020/05/20/19/13/38/e7994a4c-57bf-4578-aa27-
dc5fb8c11d36/output.json'}
```

Two human loops are currently `In Progress.` When the assigned worker logs into the review system, the worker sees the submitted review task as shown in Figure 8-7.

Hello, chris@fregly.com

Log out

Show instructions

Instructions

Get Started

Select a job and click the **Start working** button on the Jobs list.

Read Instructions

- Read the instructions carefully before you start to work on the task.
- More information is available in the full instructions and the tool guide.

Work on labeling tasks

Follow the instructions to complete your tasks. Submit the task before you see the next one.

Take actions

- **Submit** a task when you complete it.
- **Stop working** when you want to exit the annotation task and return here.

Jobs

Start working

1

	Description	Customer ID	Creation time
	Classify Reviews into Star Ratings Between 1 (Worst) and 5 (Best)	835319576252	May 18, 2020 23:43:21 UTC

Figure 8-7. A2I > Worker UI shows the review task.

Once the worker clicks on `Start working`, A2I presents the task as shown in Figure 8-8. This UI is defined by the Human Task UI we created earlier.

Hello, antje.barth@ | Customer ID: | Task description: Classify Reviews into Star Ratings Bet... | Task time: 1:18 of 60 Min | Stop working | Log out

Instructions

X

View full instructions

View tool guide

Classify Reviews into Star Ratings Between 1 (Worst) and 5 (Best)

Classify Reviews into Star Ratings Between 1 (Worst) and 5 (Best)

sometimes it works

Select an option

1	1
2	2
3	3
4	4
5	5

Submit

Figure 8-8. A2I > Worker performs review task.

The instructions are simple: “Classify Reviews into Star Ratings Between 1 (Worst) and 5 (Best)”. The worker sees the input data “sometimes it works” and might classify this as a 3-star rating. Note that you can assign a single task to more than one human reviewer to mitigate human bias. A2I consolidates multiple responses per task using weighted reviewer scores.

Once all review tasks are completed, the UI clears the task from the worker’s UI as shown in Figure 3.9.

Hello, chris@fregly.com

Log out

You're finished with the available tasks.
Refresh the page to see if there are new jobs. Select a job and choose **Start working** to work on new tasks.

Show instructions

Instructions

Get Started

Select a job and click the **Start working** button on the Jobs list.

Read Instructions

- Read the instructions carefully before you start to work on the task.
- More information is available in the full instructions and the tool guide.

Work on labeling tasks

Follow the instructions to complete your tasks. Submit the task before you see the next one.

Take actions

- **Submit** a task when you complete it.
- **Stop working** when you want to exit the annotation task and return here.

Jobs

Start working

Description	Customer ID	Creation time

Figure 8-9. A2I > Worker finished the review task.

We can also query the human loop status programmatically again:

```
completed_human_loops = []
for human_loop_name in human_loops_started:
    resp = a2i.describe_human_loop(HumanLoopName=human_loop_name)
    print(f'HumanLoop Name: {human_loop_name}')
    print(f'HumanLoop Status: {resp["HumanLoopStatus"]}')
    print(f'HumanLoop Output Destination: {resp["HumanLoopOutput"]}')
    print('')
    if resp["HumanLoopStatus"] == "Completed":
        completed_human_loops.append(resp)
```

The output should look similar to this:

```
HumanLoop Name: 72331439-0df9-4190-a42b-3e4048efb0a9
HumanLoop Status: Completed
HumanLoop Output Destination: {'OutputS3Uri': 's3://sagemaker-<region>-
<account_id>/a2i-comprehend-star-rating-results/fd-dsoaws-comprehend-fb987de9-
90f9-4304-b777-9bd49c41a78d/2020/05/20/19/13/37/72331439-0df9-4190-a42b-
3e4048efb0a9/output.json'}
HumanLoop Name: e7994a4c-57bf-4578-aa27-dc5fb8c11d36
HumanLoop Status: Completed
HumanLoop Output Destination: {'OutputS3Uri': 's3://sagemaker-<region>-
<account_id>/a2i-comprehend-star-rating-results/fd-dsoaws-comprehend-fb987de9-
90f9-4304-b777-9bd49c41a78d/2020/05/20/19/13/38/e7994a4c-57bf-4578-aa27-
dc5fb8c11d36/output.json'}
```

A2I stores the human review results in the S3. Let's take a look at the data.

```
import re
import pprint
pp = pprint.PrettyPrinter(indent=4)
for resp in completed_human_loops:
    splitted_string = re.split('s3://' +  BUCKET + '/', resp['HumanLoopOutput']
['OutputS3Uri'])
    output_bucket_key = splitted_string[1]
    response = s3.get_object(Bucket=BUCKET, Key=output_bucket_key)
    content = response["Body"].read()
    json_output = json.loads(content)
    pp.pprint(json_output)
    print('\n')
```

The output should look similar to this:

```
{
        'flowDefinitionArn': 'arn:aws:sagemaker:<region>-<account_id>:flow-
```

```
definition/fd-dsoaws-comprehend-fb987de9-90f9-4304-b777-9bd49c41a78d',
        'humanAnswers': [{
                'answerContent': {
                        'star_rating': {
                                'label': '2'
                        }
                },
                'submissionTime': '2020-05-20T19:18:39.460Z',
                'workerId': '7d504268314dca28'
        }],
        'humanLoopName': '72331439-0df9-4190-a42b-3e4048efb0a9',
        'inputContent': {
                'initialValue': '2',
                'taskObject': 'I am unhappy with this product'
        }
}
{
        'flowDefinitionArn': 'arn:aws:sagemaker:<region>-<account_id>:flow-
definition/fd-dsoaws-comprehend-fb987de9-90f9-4304-b777-9bd49c41a78d',
        'humanAnswers': [{
                'answerContent': {
                        'star_rating': {
                                'label': '3'
                        }
                },
                'submissionTime': '2020-05-20T19:18:35.731Z',
                'workerId': '7d504268314dca28'
        }],
        'humanLoopName': 'e7994a4c-57bf-4578-aa27-dc5fb8c11d36',
        'inputContent': {
                'initialValue': '3',
                'taskObject': 'sometimes it works'
        }
}
```


We can use this newly-labeled data in S3 to build a continuous pipeline for training and improving our Comprehend Custom model as shown in Figure 8-10.

Augmented AI Workflow for Improving Model Predictions

Humans Fix Predictions
If Model Confidence < THRESHOLD

Figure 8-10. Continuous training pipeline to improve model predictions.

The more accurate our model gets, the fewer reviews have to be sent to our workers. This allows us humans to focus on more complex tasks. This concept is also called "Active Learning". Active learning is the automated version of the continuous learning pipeline and is implemented in another service, Amazon SageMaker Ground Truth.

Creating Active Learning Feedback Loops

Active learning starts with a human labeling workflow and then transitions to self-labeling after enough samples have been seen. The active learning feedback loop is used to continuously re-train the model and improve the confidence of future label predictions. Active learning helps to scale the data labeling process, to handle the simple labeling tasks, and to allow the workforce to focus on more-challenging tasks that require specialized human intelligence.

Active Learning with Ground Truth

Amazon SageMaker Ground Truth is an Augmented AI workflow implementation for automatic data labeling. Ground Truth combines human review workflows with active learning. Let's discuss the concept of active learning in more detail.

As the human workforce labels more and more data, Ground Truth proactively trains a model to join the workforce and perform automated labeling of new data as it arrives. If the model is not confident, the data is sent to the human workforce for review and approval/modification. Figure 8-11 illustrates this SageMaker Ground Truth workflow and transition from manual to automated labeling.

Ground Truth Uses Active Learning to Augment Human Data Labeling

Figure 8-11. SageMaker Ground Truth uses active learning to augment human data labeling.

SageMaker Ground Truth offers pre-built labeling workflows and task templates to process images, text and video. You can also define a custom workflow. In the following example, we will create an active learning pipeline for images. Ground Truth will actively create a new object detection model beneath the covers as it sees more and more human labels. Ground Truth uses this new model to automatically detect objects in the images with increasing accuracy. This allows humans to focus on labeling fewer, more difficult images.

Our sample data is an image collection of Finn as shown in Figure 8-12. Finn is an English Retriever, family dog, and happy demo volunteer.

Figure 8-12. Meet Finn, family dog and our Ground Truth demo volunteer.

In case you don't have a family dog on hands, you can simply use the Stanford Dogs Dataset [http://vision.stanford.edu/aditya86/ImageNetDogs/] which contains more than 20,000 images of dogs across 120 dog breeds.

The task for Ground Truth is to detect the objects in those images and label the images with `human`, `animal` and/or `object`. Note that some of the

images will have one label, others might have all 3 labels attached. This is called multi-label image classification.

Augmented AI and SageMaker Ground Truth can both access the private workforce teams created in a specific AWS account and region. This means we can leverage our workforce team from the previous example. Ground Truth also offers built-in Human Task UIs to present input data and instructions to the human labeler. We simply need to create the Flow Definition, called `Labeling Job` in Ground Truth.

Create Labeling Job

Ground Truth expects the unlabeled input data in Amazon S3 together with an input manifest file. S3 manifest files are commonly used across AWS services to describe a set of S3 objects. Each of our input images represents one line in that manifest file as follows:

```
{"source-ref":"s3://dsoaws-groundtruth/finn/finn01.jpeg"}
{"source-ref":"s3://dsoaws-groundtruth/finn/finn02.jpeg"}
{"source-ref":"s3://dsoaws-groundtruth/finn/finn03.jpeg"}
...
{"source-ref":"s3://dsoaws-groundtruth/finn/finn10.jpeg"}
```

Ground Truth can automatically create this input manifest file for you. Let's choose this option when we click on `Create Labeling Job` in the SageMaker Ground Truth UI. This opens a dialog box as shown in Figure 3.13.

Create manifest file

X

Input dataset location

Provide a path to the S3 location where your dataset objects are stored.

s3://dsoaws-groundtruth/finn/

Data type

- (●) Images
  Supported formats are jpg, jpeg and png
- ( ) Text
  Supported formats are txt and csv
- ( ) Videos
  Supported formats are mp4, ogg, and webm

11 objects found

Use this manifest

Use this manifest as the input dataset for the labeling job. When you choose this, we will update the input dataset location with the path to the new manifest file

Cancel

Create

Figure 8-13. Create S3 manifest file.

Provide the S3 path to the sample data, select the data type and click `Create`. Then simply click on `Use this manifest`.

The labeling job also requires a name, the dataset output location in S3 and an IAM role as shown in Figure 3.14. Note that the input dataset location has been updated to the S3 manifest file we just created.

Amazon SageMaker > Labeling jobs > Create labeling job

Step 1
Specify job details

Step 2
Select workers and configure tool

Specify job details

Job overview

Job name

detect-objects

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

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

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

Input dataset location Info

Provide a path to the S3 location where your manifest file is stored. To find a path, go to Amazon S3

Create manifest file, if you don't have one

s3://dsoaws-groundtruth/finn/dataset-20200522T153962.manifest

The bucket and dataset objects must be in the us-east-1 region

Output dataset location Info

Provide a path to the S3 location where you want your labeled dataset to be stored. To find a path, go to Amazon S3

s3://dsoaws-groundtruth/finn/detect-objects/output

The bucket and dataset objects must be in the us-east-1 region

IAM Role Info

Amazon SageMaker requires permissions to call other services on your behalf. Choose a role or let us create a role with the AmazonSageMakerFullAccess IAM policy attached.

AmazonSageMaker-ExecutionRole-20200516T190597 ▼

▸ **Additional configuration** - *optional*

Dataset object selection, encryption

Figure 8-14. SageMaker Ground Truth > Create Labeling Job

The output of the Ground Truth labeling job is an output manifest file. Each line in the output manifest file matches the input manifest file with the addition of an attribute and value for the assigned label. You can specify the label attribute in the labeling job. By default, Ground Truth uses the labeling job name as the label attribute.

We also need to specify the labeling job task type as shown in Figure 8-15. We choose `Image` and `Image Classification (Multi-Label)`.

Task type Info

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

Image

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

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

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

Bounding box
Get workers to draw bounding boxes around specified objects in your images. Info

Semantic segmentation
Get workers to draw pixel level labels around specific objects and segments in your images. Info

Label verification
Get workers to verify existing labels in your dataset. Info

▸ **Tags** - *optional*

Cancel Next

Figure 8-15. Specify labeling job task type.

As mentioned earlier, Amazon Ground Truth already implements the Human Task UI which supports these labeling tasks.

Next, we select the workforce. Similar to Augmented AI, we can choose between public workforces such as Amazon Mechanical Turk and third party vendors, or a private workforce. For our example, we select the previously created private workforce as shown in Figure 8-16.

Amazon SageMaker > Labeling jobs > Create labeling job

Step 1
Specify job details

Step 2
Select workers and configure tool

Select workers and configure tool

Workers Info

Worker types

- Amazon Mechanical Turk
  An on-demand 24/7 workforce of over 500,000 independent contractors worldwide powered by Amazon Mechanical Turk.
- Private
  A team of workers that you have sourced yourself, including you own employees or contractors for handling data that needs to stay within your organization.
- Vendor managed
  A curated list of third party vendors that specialize in providing data labeling services, available via the AWS Marketplace.

Private teams

Choose from the teams you created in the private workforce or if you need to create a new team, save your progress and go to Labeling workforces to create a new one.

dsoaws

Task Timeout

Maximum time a worker can work in a single task

0 hours 5 mins

Additional configuration - *optional*

Workers per dataset object

Number of workers per dataset object

The number of distinct workers you want to perform the same task on a dataset object. This can help increase the accuracy of the data labels.

1 workers

Figure 8-16. SageMaker Ground Truth > Select workers

Lastly, we can put in the instructions for our workers as shown in Figure 8-17. In this case, we ask the workforce to "Please detect all objects in this image and assign the corresponding label of human, animal or object. You can assign multiple labels per image."

Image classification (Multi-label) labeling tool

Preview

Provide labeling instructions with examples below for workers. Workers will be viewing these instructions when they perform your task. You can add up to 50 labels for workers to choose from. See guidelines for creating high-quality instructions

H1 H2 B I A

Good example

Enter description to explain the correct label to the workers

Add image here

Bad example

Enter description of an incorrect label

Add image here

Please detect all objects in this image and assign the corresponding label of human, animal or object. You can assign multiple labels per image.

Select an option

Add up to 50 labels

human X

animal X

object X

Add label

Additional instructions - *optional*

Cancel Previous Create

Figure 8-17. SageMaker Ground Truth > Labeling Job Instructions

If you want to provide more detailed instructions, you can add sample images to show correct and incorrect labeling. Click on `Create` to start the labeling job.

Human Worker Performs Task

The assigned workers now receive an email that new labeling jobs are available. The email contains the job details and the link to the labeling portal UI as shown in Figure 8-18.

New labeling jobs are available Inbox x

AWS Notifications <no-reply@sns.amazonaws.com> 4:57 PM (0 minutes ago)

to me

{"eventVersion":"1.0","eventSource":"aws:sagemaker:labelingjob","eventName":"LabelingJob:WorkAvailable","workTeamArn":"arn:aws:sagemaker:us-east-1: :workteam/private-crowd/dsoaws","taskTitle":"Bounding boxes: Draw a tight bounding box around Finn.","labelingPortalUrl":"https://etu6nojf26.labeling.us-east-1.sagemaker.aws","labelingJobName":"find-the-dog ","jobReferenceCode":"806570384721x7a0rZtqHdP9xczb"}

--

If you wish to stop receiving notifications from this topic, please click or visit the link below to unsubscribe:

https://sns.us-east-1.amazonaws.com/unsubscribe.html?SubscriptionArn=arn:aws:sns:us-east-1: :AmazonSageMaker-dsoaws-groundtruth-sns:a986f2c8-bee7-4dba-816e-9ebc32741f00&Endpoint=antje.barth@

Please do not reply directly to this email. If you have any questions or comments regarding this email, please contact us at

https://aws.amazon.com/support

Figure 8-18. Workers receive an email that new labeling jobs are available.

The worker logs into the labeling portal and sees the available job as shown in Figure 8-19.

Figure 8-19. SageMaker Ground Truth > Worker task list.

Once the worker clicks on `Start working`, they are presented with the images from our dataset that need to be labeled, together with the instructions as shown in Figure 8-20.

Hello, antje.barth@ | Customer ID... | Task description: Categori... | Task time: 0:25 of 5 Min | Stop working | Log out

Instructions

X

View full instructions

View tool guide

Good example

Enter description to explain the correct label to the workers

Add image here

Bad example

Enter description of an incorrect label

Please detect all objects in this image and assign the corresponding label of human, animal or object. You can assign multiple labels per image.

Select appropriate categories

- human 1
- animal 2
- object 3

Zoom in | Zoom out | Move | Fit image

Submit

Figure 8-20. SageMaker Ground Truth > Labeling Job

While the workers are completing the labeling job, Ground Truth creates several directories in the S3 output location:

```
s3://dsoaws-groundtruth/find-the-dog/activelearning
s3://dsoaws-groundtruth/find-the-dog/annotations
s3://dsoaws-groundtruth/find-the-dog/inference
s3://dsoaws-groundtruth/find-the-dog/manifests
s3://dsoaws-groundtruth/find-the-dog/training
```

The `Active Learning` directory, the `Inference` directory and the `Training` directory are only present if you enable automated data labeling. They contain all input/output files needed to train and validate the custom SageMaker model, as well as the model's automatic labeling results.

The `Annotations` directory contains the individual worker responses. In case more workers have provided labels for the same input, you can also find information on how to consolidate the labels in this folder. Here is a sample (shortened) worker response:

```
{
        "answers": [{
                "answerContent": {
                        "crowd-image-classifier-multi-select": {
                                "labels": ["animal", "object"]
                        }
                },
                "submissionTime": "2020-05-22T14:21:04.367Z",
                "workerId": "...",
                "workerMetadata": {
                        "identityData": {
                                "identityProviderType": "...",
                                "issuer": "...",
                                "sub": ".."
                        }
                }
        }]
}
```

The `Manifest` directory contains the final output data file for this labeling job called `output.manifest`. Here is a sample (shortened) output.manifest file:

```
{
        "source-ref": "s3://dsoaws-groundtruth/finn/finn01.jpeg",
        "detect-objects": [1, 2],
        "detect-objects-metadata": {
                "job-name": "labeling-job/detect-objects",
                "class-map": {
                        "1": "animal",
                        "2": "object"
                },
                "human-annotated": "yes",
                "creation-date": "2020-05-22T14:22:05.132702",
                "confidence-map": {
                        "1": 0.63,
                        "2": 0.63
                },
                "type": "groundtruth/image-classification-multilabel"
        }
} {
        ...
}
```


The task in this example was simple. The workers received an image and attached labels for human, animal and/or object. Many AI use cases require more complex labeling though. What if we need to further distinguish humans into adult and child, animals could require a classification into cat and dog, and so on. Figure 8-21 illustrates this challenge of building "hierarchical taxonomies" of our data labels.

Figure 8-21. Hierarchical taxonomies of data labels.

One "brute force" solution to this problem is to manually create a separate labeled dataset for each level of the hierarchy. This means, you create one labeled dataset to distinguish between human, animal, and object. You

create another dataset to sub-classify humans, another dataset to sub-classify animals, and yet another dataset to sub-classify objects. This approach not only creates a large number of datasets to manage, but also requires new ML tools to work efficiently with multi-layer labels. Amazon SageMaker Ground Truth offers a built-in solution.

Chain Ground Truth Jobs

With Ground Truth, you can leverage the output manifest file from one labeling job and use the file as the input manifest for the next labeling job. This chaining of labeling jobs provides the basis for an automated pipeline to create the needed hierarchies as shown in Figure 8-22.

Figure 8-22. Chaining of Ground Truth Labeling Jobs.

When you define the second labeling job, you can apply a filter on the input data. In our example, let's say we want to further classify `human` in an

image into `adult` and `child`. Using a S3 SQL query, we filter for images which have the label "`human`“. Here is a sample SQL query for our example:

```
select * from s3Object s where s."label-human-animal-object-metadata"."class-map"."0" = 'human';
```

The second labeling job then only presents this subset of images to the workers. The output of the second labeling job is an augmented manifest file. Ground Truth “augments” the labels for the images inline with the additional label. We can use augmented manifest files to train a custom machine learning model on this labeled data. Amazon SageMaker supports augmented manifest files natively as an input data channel for model training data.

Let’s chain a second labeling job to our example to sub-classify the images which are labeled with a human into adult and child. You can simply select the first labeling job in the Ground Truth UI and click on `Actions` > `Chain` as shown in Figure 8-23.

Amazon SageMaker 〉 Labeling jobs 〉 detect-objects

detect-objects

Actions ▲

Clone

Chain

Stop

Labeling job summary

View label

Status Complete	Input dataset location s3://dsoaws-groundtruth/finn/dataset-20200522T153962.manifest	ARN arn:aws:sagemaker:us-east-1: :labeling-job/detect-objects
Task type Image Classification (Multi-label)	Labeled / total dataset objects 11 / 11	Worker type Private
Creation time May 22, 2020, 2:10 PM UTC	Output dataset location s3://dsoaws-groundtruth/finn/detect-objects/output/detect-objects/	Workteam ARN arn:aws:sagemaker:us-east-1: :workteam/private-crowd/dsoaws

Figure 8-23. Chain a labeling job.

You will notice that Ground Truth automatically selects the output manifest file from the first job as input manifest file for the new labeling job as shown in Figure 8-24.

Specify job details

Job overview

Job name

label-human-adult-child

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

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

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

Input dataset location Info

Provide a path to the S3 location where your manifest file is stored. To find a path, go to Amazon S3

Create manifest file, if you don't have one

s3://dsoaws-groundtruth/finn/detect-objects/output/detect-objects/manifests/o

The bucket and dataset objects must be in the us-east-1 region

Output dataset location Info

Provide a path to the S3 location where you want your labeled dataset to be stored. To find a path, go to Amazon S3

s3://dsoaws-groundtruth/finn/detect-objects/label-human-adult-child/output

The bucket and dataset objects must be in the us-east-1 region

Figure 8-24. Ground Truth uses output manifest file as input manifest file for new job.

For this second hierarchy of data labels, we want to classify humans into adults and children. Therefore, we filter our input dataset to only show images with humans in it. You can select this option in the Additional configuration of the labeling job as shown in Figure 8-25.

▼ **Additional configuration** - *optional*

Dataset object selection, encryption

Dataset object selection Info

You can use the full dataset or create a subset of your data.

- ○ **Full dataset** — Use all the dataset objects
- ○ **Random sample** — Create a subset by specifying a sample size
- ◉ **Filtered subset** — Create a subset by specifying a query

Filtered subset

Use SQL expressions to query your dataset and create a filtered subset

▶ **File preview** - *optional*

Query

Choose which objects are in the subset by adding a WHERE clause to the query. Learn more

```
1 select * from s3Object s where s."detect-objects-metadata"
    ."class-map"."0" = 'human';
```

Create subset

4 objects selected)}

Use this subset

Use this subset as the input dataset for the labeling job. When you choose this, we will update the input dataset location with the path to the new manifest file

Figure 8-25. Define a filtered subset of the input data.

The following query scans all S3 objects listed in our input manifest file and selects the ones that have a label with metadata for `human` attached:

```
select * from s3Object s where s."detect-objects-
metadata"."class-map"."0" = 'human';
```

You can test the query in the UI and see how many files match this query. In our example, this query correctly finds 4 images with humans. Click on `Use this subset`.

Next, we select `Image Classification (Single Label)` as our Task type as shown in Figure 8-26.

Task type Info

Task category

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

Image

Task selection

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

Image Classification (Single Label)

Get workers to categorize images into individual classes. Info

Image Classification (Multi-label)

Get workers to categorize images into one or more classes. Info

Figure 8-26. Select Image Classification (Single Label) as task type.

Provide worker instructions such as "`Classify whether the human in the image is an adult or child. "`, add the label options and click `Create`. Ground Truth now runs our second labeling job. The assigned workers receive an email and can start classifying the images with the second tier of hierarchical labels.

As mentioned earlier, the output of this second labeling job is an augmented manifest file. We can find the augmented (added) labels from the second labeling job in there:


```
{
        "source-ref": "s3://dsoaws-groundtruth/finn/finn03.jpeg",
        "detect-objects": [0, 1, 2],
        "detect-objects-metadata": {
                "job-name": "labeling-job/detect-objects",
                "class-map": {
                        "1": "animal",
                        "0": "human",
                        "2": "object"
                },
                "human-annotated": "yes",
                "creation-date": "2020-05-22T14:23:10.974325",
                "confidence-map": {
                        "1": 0.58,
                        "0": 0.58,
                        "2": 0.58
                },
                "type": "groundtruth/image-classification-multilabel"
        },
        "label-human-adult-child": 1,
        "label-human-adult-child-metadata": {
                "confidence": 0.84,
                "job-name": "labeling-job/label-human-adult-child",
                "class-name": "child",
                "human-annotated": "yes",
                "creation-date": "2020-05-22T14:50:02.931996",
                "type": "groundtruth/image-classification"
        }
} {
        ...
}
```

This example showed how to chain Ground Truth labeling jobs to build more complex label hierarchies.

Summary

In this chapter, we discussed the concept of augmented AI and active learning. You learned how to integrate human review workflows into your AI applications with A2I. You met Finn, and learned how SageMaker Ground Truth combines human workforces with an active learning model to create accurate training datasets.

About the Authors

Chris Fregly is a Developer Advocate for AI and Machine Learning at AWS based in San Francisco, California. He is also the founder of the Advanced Spark, TensorFlow, and KubeFlow Meetup Series based in San Francisco. Chris regularly speaks at AI and Machine Learning conferences across the world including the O'Reilly AI, Strata, and Velocity Conferences. Previously, Chris was Founder at PipelineAI where he worked with many AI-first startups and enterprises to continuously deploy ML/AI Pipelines using Apache Spark ML, Kubernetes, TensorFlow, Kubeflow, Amazon EKS, and Amazon SageMaker. He is also the author of the O'Reilly Online Training Series "High Performance TensorFlow in Production with GPUs."

Antje Barth is a Developer Advocate for AI and Machine Learning at AWS based in Düsseldorf, Germany. She is also co-founder of the Düsseldorf chapter of Women in Big Data Meetup. Antje frequently speaks at AI and Machine Learning conferences and meetups around the world, including the O'Reilly AI and Strata conferences. Besides ML/AI, Antje is passionate about helping developers leverage Big Data, container and Kubernetes platforms in the context of AI and Machine Learning. Prior to joining AWS, Antje worked in technical evangelist and solutions engineering roles at MapR and Cisco.

1. 1. Automated Machine Learning
    a. What is Automated Machine Learning?
    b. Automated Machine Learning with Autopilot
        i. Understand Autopilot's White-Box Approach to AutoML
        ii. Use SageMaker Experiments in Autopilot
        iii. Train and Deploy with Autopilot UI
        iv. Train and Deploy with Autopilot Python SDK
        v. Predict with Amazon Athena
    c. Automated Machine Learning with Comprehend
        i. Predict with Comprehend Built-In Model
        ii. Train and Deploy Comprehend Custom Model with Comprehend UI
        iii. Train and Deploy Comprehend Custom Model with the Comprehend Python SDK
    d. Summary
2. 2. Ingesting Data Into The Cloud
    a. Data Lakes
        i. Import Your Data into the S3 Data Lake
        ii. Describe the Dataset
    b. Query the S3 Data Lake with Amazon Athena
        i. Access Athena from the AWS Console
        ii. Register Your S3 Data as an Athena Table

iii. Create a Parquet-based Table in Athena

c. Load Dataset into Redshift Data Warehouse

i. Query the Data Lake and Data Warehouse with Redshift Spectrum

ii. Export Redshift Data to S3 Data Lake as Parquet

d. Choosing Athena vs. Redshift

e. Secure Your Dataset

i. Authenticating and Authorizing with Identity and Access Management (IAM) Roles

ii. Storing Keys with Key Management Service

iii. Securing Buckets with S3 Access Points

f. Increase Performance and Reduce Cost

i. Parquet Partitions and Compression

ii. Redshift Table Design and Compression

iii. S3 Intelligent Tiering

g. Summary

3. 3. Exploring the Dataset

a. Overview

b. Visualize our Data Lake with Athena

i. Prepare SageMaker Notebook for Athena

ii. Run a Sample Athena Query in the SageMaker Notebook

iii. Dive Deep into the Dataset with Athena and SageMaker

c. Query our Data Warehouse

i. Prepare SageMaker Notebook for Redshift

ii. Run a Sample Redshift Query from the SageMaker Notebook

iii. Dive Deep into the Dataset with Redshift and SageMaker

d. Create Dashboards with QuickSight

i. Setup the Datasource

ii. Query and Visualize the Dataset Within QuickSight

e. Detect Data Quality Issues with Apache Spark

i. SageMaker Processing Jobs

ii. Analyze Our Dataset with Deequ and Apache Spark

f. Increase Performance and Reduce Cost

i. Approximate Counts with HyperLogLog

ii. Dynamically Scale Your Data Warehouse with Redshift AQUA

iii. Improve Dashboard Performance with QuickSight SPICE

g. Summary

4. 4. Preparing the Dataset for Model Training

a. Perform Feature Engineering

i. Select Training Features and Labels

ii. Balance the Dataset to Improve Your Model

iii. Split the Dataset into Train, Validation, and Test

iv. Transform Raw Text into BERT Embeddings

v. Convert Features to TFRecord File Format

b. Scale Feature Engineering with SageMaker Processing Jobs

i. Transform with Scikit-Learn and TensorFlow

ii. Transform with Apache Spark and TensorFlow

c. Automate Feature Engineering with Amazon Step Functions

i. Invoke a Pipeline with S3 Triggers

d. Share Features Through a Feature Store

e. Summary

5. 5. Training Your First Model with SageMaker

a. Understand the SageMaker Infrastructure

i. SageMaker Container Environment Variables and S3 Locations

ii. Compute and Network Isolation

iii. Data Encryption at Rest and In-Transit

b. Develop a SageMaker Model

i. Built-In Algorithms

ii. Bring Your Own Script or Script Mode

iii. Bring Your Own Container

c. A Brief History of Natural Language Processing

i. Contextual Algorithms

ii. Attention-Based Algorithms

d. Training BERT from Scratch

i. Masked Language Model (Masked LM)

ii. Next Sentence Prediction

e. Use Pre-Trained BERT Models

i. Fine Tune the BERT Model to Create a Custom Classifier

f. Create the Training Script

i. Setup the Train, Validation, and Test Datasets

ii. Set Up the Custom Classifier Model

iii. Train and Validate the Model

iv. Save the Model

g. Launch the Script from a SageMaker Notebook

i. Define the Metrics to Capture and Monitor

ii. Configure the Hyper-Parameters for Our Algorithm

iii. Select Instance Type and Instance Count

iv. Putting it All Together in the Notebook

h. Evaluate Model Training

i. Run Some AdHoc Predictions from the Notebook

ii. Confusion Matrix

iii. TensorBoard

iv. CloudWatch

i. Debug Model Training with SageMaker Debugger

j. Increase Performance and Reduce Costs

i. Reduced 16-bit Half Precision

ii. Mixed 32-bit Full and 16-bit Half Precision

iii. Quantization

iv. Spot Instances and Checkpoints

v. Early Stopping

k. Summary

6. 6. Training and Optimizing Models at Scale

a. Compare Training Runs with SageMaker Experiments

i. Trace and Audit Model Lineage

ii. Reproduce a Model

iii. Manage Artifacts and Dependencies

iv. Track Our Model Lifecycle with the Experiments API

v. Set Up the Experiment

b. Automatically Find the Best Model Hyper-Parameters

i. Set Up the Hyper-Parameter Ranges

ii. Run the Hyper-Parameter Tuning Job

iii. Analyze the Tuning Job Results

c. Warm Start Additional Hyper-Parameter Tuning Jobs

i. Run Hyper-Parameter Tuning Job using Warm Start

d. Train and Tune Models at Scale

i. Increase Cluster Instance Count

ii. Choose an Appropriate Cluster Communication Strategy

e. Train with Distributed File Systems

i. Cache S3 Data using FSx for Lustre

ii. Share Data using Elastic File System

f. Reduce Costs and Increase Performance

i. Shard the Data with ShardedByS3Key

ii. Stream Data On-the-Fly with PipeMode

iii. Enable Enhanced Networking

g. Summary

7. 7. Deploying Models to Production with SageMaker

a. Collaborating with Multiple Teams

b. Choose Level Of Customization

i. Built-In Algorithms

ii. Bring Your Own Script

iii. Bring Your Own Container

c. Choose Real-Time or Batch Predictions

d. Real-Time Predictions with SageMaker Endpoints

    i. Deploy Model using SageMaker Python SDK

    ii. Endpoint Configuration

    iii. Track Model Deployment in our Experiment

    iv. Analyze Model Deployment Lineage

    v. Invoking Predictions using the SageMaker Python SDK

    vi. Invoke Predictions from Using HTTP POST

e. Creating Inference Pipelines

f. Deploying New Models

    i. Split Traffic for Canary Rollouts

    ii. Shift Traffic for Blue/Green Deployments

g. Testing and Comparing New Models

    i. Perform A/B Tests to Compare Model Variants

    ii. Reinforcement Learning with Multi-Armed Bandit Testing

h. Auto-Scale SageMaker Endpoints using CloudWatch

    i. Define a Scaling Policy with Custom Metrics

    ii. Using Pre-Defined Metrics

    iii. Tuning Responsiveness Using a Cool Down Period

i. Monitor Predictions and Detect Drift

    i. Enable Data Capture

ii. Create Baseline Statistics and Constraints for Features

iii. Schedule Monitoring Jobs

iv. Interpret Results

v. Visualize Results in Amazon SageMaker Studio

j. Perform Batch Predictions with SageMaker Batch Transform

i. Selecting an Instance Type

ii. Setup the Input Data

iii. Tune the Batch Transformation Configuration

iv. Prepare the Batch Transformation Job

v. Run the Batch Transformation Job

vi. Review the Batch Predictions

k. Lambda Functions and API Gateway

l. Reduce Costs and Increase Performance

i. Deploy Multiple Models in One Container

ii. Attach a GPU-based Elastic Inference Accelerator

iii. Optimize a Trained Model with SageMaker Neo and TensorFlow Lite

iv. Use Inference-Optimized Hardware

m. Summary

8. 8. Human-in-the-Loop Workflows

a. Building Human-in-the-Loop Workflows with Amazon Augmented AI

    i. Create a Private Workforce

    ii. Choose a Human Task UI

    iii. Create a Flow Definition

    iv. Start a Human Loop for Low-Confidence Predictions

b. Creating Active Learning Feedback Loops

    i. Active Learning with Ground Truth

    ii. Create Labeling Job

    iii. Human Worker Performs Task

    iv. Chain Ground Truth Jobs

c. Summary