Ben G. Weber

Data Science in Production: Building Scalable Model Pipelines with Python

Contents

Preface

This book was developed using the leanpub[1] platform. Please send any feedback or corrections to: `bgweber@gmail.com`

The data science landscape is constantly evolving, because new tools and libraries are enabling smaller teams to deliver more impactful products. In the current state, data scientists are expected to build systems that not only scale to a single product, but a portfolio of products. The goal of this book is to provide data scientists with a set of tools that can be used to build predictive model services for product teams.

This text is meant to be a Data Science 201 course for data science practitioners that want to develop skills for the applied science discipline. The target audience is readers with past experience with Python and scikit-learn than want to learn how to build data products. The goal is to get readers hands-on with a number of tools and cloud environments that they would use in industry settings.

0.1 Prerequisites

This book assumes that readers have prior knowledge of Python and Pandas, as well as some experience with modeling packages such as scikit-learn. This is a book that will focus on breadth rather than depth, where the goal is to get readers hands on with a number of different tools.

Python has a large library of books available, covering the language fundamentals, specific packages, and disciplines such as data

[1]`https://leanpub.com/ProductionDataScience`

science. Here are some of the books I would recommend for readers to build additional knowledge of the Python ecosystem.

- **Python And Pandas**
  - *Data Science from Scratch* (Grus, 2015): Introduces Python from a data science perspective.
  - *Python for Data Analysis* (McKinney, 2017): Provides extensive details on the Pandas library.
- **Machine Learning**
  - *Hands-On Machine Learning* (Géron, 2017): Covers scikit-learn in depth as well as TensorFlow and Keras.
  - *Deep Learning for Python* (Chollet, 2017): Provides an excellent introduction to deep learning concepts using Keras as the core framework.

I will walk through the code samples in this book in detail, but will not cover the fundamentals of Python. Readers may find it useful to first explore these texts before digging into building large scale pipelines in Python.

0.2 Book Contents

The general theme of the book is to take simple machine learning models and to scale them up in different configurations across multiple cloud environments. Here's the topics covered in this book:

1. **Introduction:** This chapter will motivate the use of Python and discuss the discipline of applied data science, present the data sets, models, and cloud environments used throughout the book, and provide an overview of automated feature engineering.

2. **Models as Web Endpoints:** This chapter shows how to use web endpoints for consuming data and hosting machine learning models as endpoints using the Flask and Gunicorn libraries. We'll start with scikit-learn models and also set up a deep learning endpoint with Keras.

3. **Models as Serverless Functions:** This chapter will build upon the previous chapter and show how to set up model endpoints as serverless functions using AWS Lambda and GCP Cloud Functions.

4. **Containers for Reproducible Models:** This chapter will show how to use containers for deploying models with Docker. We'll also explore scaling up with ECS and Kubernetes, and building web applications with Plotly Dash.

5. **Workflow Tools for Model Pipelines:** This chapter focuses on scheduling automated workflows using Apache Airflow. We'll set up a model that pulls data from BigQuery, applies a model, and saves the results.

6. **PySpark for Batch Modeling:** This chapter will introduce readers to PySpark using the community edition of Databricks. We'll build a batch model pipeline that pulls data from a data lake, generates features, applies a model, and stores the results to a No SQL database.

7. **Cloud Dataflow for Batch Modeling:** This chapter will introduce the core components of Cloud Dataflow and implement a batch model pipeline for reading data from BigQuery, applying an ML model, and saving the results to Cloud Datastore.

8. **Streaming Model Workflows:** This chapter will introduce readers to Kafka and PubSub for streaming messages in a cloud environment. After working through this material, readers will learn how to use these message brokers to creating streaming model pipelines with PySpark and Dataflow that provide near real-time predictions.

After working through this material, readers should have hands-on experience with many of the tools needed to build data products, and have a better understanding of how to build scalable machine learning pipelines in a cloud environment.

0.3 Code Examples

Since the focus of this book is to get readers hands on with Python code, I have provided code listings for a subset of the chapters on GitHub. The following URL provides listings for code examples that work well in a Jupyter environment:

- `https://github.com/bgweber/DS_Production`

Due to formatting restrictions, many of the code snippets in this book break commands into multiple lines while omitting the continuation operator (\). To get code blocks to work in Jupyter or another Python coding environment, you may need to remove these line breaks. The code samples in the notebooks listed above do not add these line breaks and can be executed without modification, excluding credential and IP changes. This book uses the terms `scikit-learn` and `sklearn` interchangeably with `sklearn` used explicitly in Section 3.3.3.

0.4 Acknowledgements

I was able to author this book using Yihui Xie's excellent bookdown package (Xie, 2015). For the design, I used Shashi Kumar's template[2] available under the Creative Commons 4.0 license. The book cover uses Cédric Franchetti's image from pxhere[3].

This book was last updated on December 31, 2019.

[2] `https://bit.ly/2MjFDgV`
[3] `https://pxhere.com/en/photo/1417846`

1

Introduction

Putting predictive models into production is one of the most direct ways that data scientists can add value to an organization. By learning how to build and deploy scalable model pipelines, data scientists can own more of the model production process and rapidly deliver data products. Building data products is more than just putting code into production, it also includes DevOps and lifecycle management of live systems.

Throughout this book, we'll cover different cloud environments and tools for building scalable data and model pipelines. The goal is to provide readers with the opportunity to get hands on and start building experience with a number of different tools. While this book is targeted at analytics practitioners with prior Python experience, we'll walk through examples from start to finish, but won't dig into the details of the programming language itself.

The role of data science is constantly transforming and adding new specializations. Data scientists that build production-grade services are often called applied scientists. Their goal is to build systems that are scalable and robust. In order to be scalable, we need to use tools that can parallelize and distribute code. Parallelizing code means that we can perform multiple tasks simultaneously, and distributing code means that we can scale up the number of machines needed in order to accomplish a task. Robust services are systems that are resilient and can recover from failure. While the focus of this book is on scalability rather than robustness, we will cover monitoring systems in production and discuss measuring model performance over time.

During my career as a data scientist, I've worked at a number of video game companies and have had experience putting propensity models, lifetime-value predictions, and recommendation systems into production. Overall, this process has become more streamlined with the development of tools such as PySpark, which enable data scientists to more rapidly build end-to-end products. While many companies now have engineering teams with machine learning focuses, it's valuable for data scientists to have broad expertise in productizing models. Owning more of the process means that a data science team can deliver products quicker and iterate much more rapidly.

Data products are useful for organizations, because they can provide personalization for the user base. For example, the recommendation system that I designed for EverQuest Landmark[1] provided curated content for players from a marketplace with thousands of user-created items. The goal of any data product should be creating value for an organization. The recommendation system accomplished this goal by increasing the revenue generated from user-created content. Propensity models, which predict the likelihood of a user to perform an action, can also have a direct impact on core metrics for an organization, by enabling personalized experiences that increase user engagement.

The process used to productize models is usually unique for each organization, because of different cloud environments, databases, and product organizations. However, many of the same tools are used within these workflows, such as SQL and PySpark. Your organization may not be using the same data ecosystem as these examples, but the methods should transfer to your use cases.

In this chapter, we will introduce the role of applied science and motivate the usage of Python for building data products, discuss different cloud and coding environments for scaling up data science, introduce the data sets and types of models used throughout the book, and introduce automated feature engineering as a step to include in data science workflows.

[1] https://bit.ly/2YFlYPg

1.1 Applied Data Science

Data science is a broad discipline with many different specializations. One distinction that is becoming common is product data science and applied data science. Product data scientists are typically embedded on a product team, such as a game studio, and provide analysis and modeling that helps the team directly improve the product. For example, a product data scientist might find an issue with the first-time user experience in a game, and make recommendations such as which languages to focus on for localization to improve new user retention.

Applied science is at the intersection of machine learning engineering and data science. Applied data scientists focus on building data products that product teams can integrate. For example, an applied scientist at a game publisher might build a recommendation service that different game teams can integrate into their products. Typically, this role is part of a central team that is responsible for owning a data product. A data product is a production system that provides predictive models, such as identifying which items a player is likely to buy.

Applied scientist is a job title that is growing in usage across tech companies including Amazon, Facebook, and Microsoft. The need for this type of role is growing, because a single applied scientist can provide tremendous value to an organization. For example, instead of having product data scientists build bespoke propensity models for individual games, an applied scientist can build a scalable approach that provides a similar service across a portfolio of games. At Zynga, one of the data products that the applied data science team built was a system called AutoModel[2], which provided several propensity models for all of our games, such as the likelihood for a specific player to churn.

There's been a few developments in technology that have made applied science a reality. Tools for automated feature engineering,

[2] https://ubm.io/2KdYRDq

such as deep learning, and scalable computing environments, such as PySpark, have enabled companies to build large scale data products with smaller team sizes. Instead of hiring engineers for data ingestion and warehousing, data scientists for predictive modeling, and additional engineers for building a machine learning infrastructure, you can now use managed services in the cloud to enable applied scientists to take on more of the responsibilities previously designated to engineering teams.

One of the goals of this book is to help data scientists make the transition to applied science, by providing hands-on experience with different tools that can be used for scalable compute and standing up services for predictive models. We will work through different tools and cloud environments to build proof of concepts for data products that can translate to production environments.

1.2 Python for Scalable Compute

Python is quickly becoming the de facto language for data science. In addition to the huge library of packages that provide useful functionality, one of the reasons that the language is becoming so popular is that it can be used for building scalable data and predictive model pipelines. You can use Python on your local machine and build predictive models with scikit-learn, or use environments such as Dataflow and PySpark to build distributed systems. While these different environments use different libraries and programming paradigms, it's all in the same language of Python. It's no longer necessary to translate an R script into a production language such as Java, you can use the same language for both development and production of predictive models.

It took me awhile to adopt Python as my data science language of choice. Java had been my preferred language, regardless of task, since early in my undergraduate career. For data science tasks, I used tools like Weka to train predictive models. I still find Java to be useful when building data pipelines, and it's great to know in order to directly collaborate with engineering teams on projects. I later switched to R while working at Electronic Arts, and found the transition to an interactive coding environment to be quite useful for data science. One of the features I really enjoyed in R is R Markdown, which you can use to write documents with inline code. In fact, this entire book was written using an extension of R Markdown called bookdown (Xie, 2019). I later switched to using R within Jupyter notebooks and even wrote a book on using R and Java for data science at a startup (Weber, 2018).

When I started working at Zynga in 2018, I adopted Python and haven't looked back. It took a bit of time to get used to the new language, but there are a number of reasons that I wanted to learn Python:

- **Momentum**: Many teams are already using Python for production, or portions of their data pipelines. It makes sense to also use Python for performing analysis tasks.
- **PySpark:** R and Java don't provide a good transition to authoring Spark tasks interactively. You can use Java for Spark, but it's not a good fit for exploratory work, and the transition from Python to PySpark seems to be the most approachable way to learn Spark.
- **Deep Learning:** I'm interested in deep learning, and while there are R bindings for libraries such as Keras, it's better to code in the native language of these libraries. I previously used R to author custom loss functions, and debugging errors was problematic.
- **Libraries:** In addition to the deep learning libraries offered for Python, there's a number of other useful tools including Flask and Bokeh. There's also notebook environments that can scale including Google's Colaboratory and AWS SageMaker.

To ease the transition from R to Python, I took the following steps:

- **Focus on outcomes, not semantics:** Instead of learning about all of the fundamentals of the language, I first focused on doing in Python what I already knew how to do in other languages, such as training a logistic regression model.

- **Learn the ecosystem, not the language:** I didn't limit myself to the base language when learning, and instead jumped right in to using Pandas and scikit-learn.
- **Use cross-language libraries:** I was already familiar working with Keras and Plotly in R, and used knowledge of these libraries to bootstrap learning Python.
- **Work with real-world data:** I used the data sets provided by Google's BigQuery to test out my scripts on large-scale data.
- **Start locally if possible:** While one of my goals was to learn PySpark, I first focused on getting things up and running on my local machine before moving to cloud ecosystems.

There are many situations where Python is not the best choice for a specific task, but it does have broad applicability when prototyping models and building scalable model pipelines. Because of Python's rich ecosystem, we will be using it for all of the examples in this book.

1.3 Cloud Environments

In order to build scalable data science pipelines, it's necessary to move beyond single machine scripts and move to clusters of machines. While this is possible to do with an on-premise setup, a common trend is using cloud computing environments in order to achieve large-scale processing. There's a number of different options available, with the top three platforms currently being Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure.

Most cloud platforms offer free credits for getting started. GCP offers a $300 credit for new users to get hands on with their tools, while AWS provides free-tier access for many services. In this book, we'll get hands on with both AWS and GCP, with little to no cost involved.

1.3.1 Amazon Web Services (AWS)

AWS is currently the largest cloud provider, and this dominance has been demonstrated by the number of gaming companies using the platform. I've had experience working with AWS at Electronic Arts, Twitch, and Zynga. This platform has a wide range of tools available, but getting these components to work well together generally takes additional engineering work versus GCP.

With AWS, you can use both self-hosted and managed solutions for building data pipelines. For example, the managed option for messaging on AWS is Kinesis, while the self-hosted option is Kafka. We'll walk through examples with both of these options in Chapter 8. There's typically a tradeoff between cost and DevOps when choosing between self-hosted and managed options.

The default database to use on AWS is Redshift, which is a columnar database. This option works well as a data warehouse, but it doesn't scale well to data lake volumes. It's common for organizations to set up a separate data lake and data warehouse on AWS. For example, it's possible to store data on S3 and use tools such as Athena to provide data lake functionality, while using Redshift as a solution for a data warehouse. This approach has worked well in the past, but it creates issues when building large-scale data science pipelines. Moving data in and out of a relational database can be a bottleneck for these types of workflows. One of the solutions to this bottleneck is to use vendor solutions that separate storage from compute, such as Snowflake or Delta Lake.

The first component we'll work with in AWS is Elastic Compute (EC2) instances. These are individual virtual machines that you can spin up and provision for any necessary task. In section 1.4.1, we'll show how to set up an instance that provides a remote Jupyter environment. EC2 instances are great for getting started with tools such as Flask and Gunicorn, and getting started with Docker. To scale up beyond individual instances, we'll explore Lambda functions and Elastic Container Services.

To build scalable pipelines on AWS, we'll focus on PySpark as the main environment. PySpark enables Python code to be distributed across a cluster of machines, and vendors such as Databricks provide managed environments for Spark. Another option that is available only on AWS is SageMaker, which provides a Jupyter notebook environment for training and deploying models. We are not covering SageMaker in this book, because it is specific to AWS and currently supports only a subset of predictive models. Instead, we'll explore tools such as MLflow.

1.3.2 Google Cloud Platform (GCP)

GCP is currently the third largest cloud platform provider, and offers a wide range of managed tools. It's currently being used by large media companies such as Spotify, and within the games industry being used by King and Niantic. One of the main benefits of using GCP is that many of the components can be wired together using Dataflow, which is a tool for building batch and streaming data pipelines. We'll create a batch model pipeline with Dataflow in Chapter 7 and a streaming pipeline in Chapter 8.

Google Cloud Platform currently offers a smaller set of tools than AWS, but there is feature parity for many common tools, such as PubSub in place of Kinesis, and Cloud Functions in place of AWS Lambda. One area where GCP provides an advantage is BigQuery as a database solution. BigQuery separates storage from compute, and can scale to both data lake and data warehouse use cases.

Dataflow is one of the most powerful tools for data scientists that GCP provides, because it empowers a single data scientist to build large-scale pipelines with much less effort than other platforms. It enables building streaming pipelines that connect PubSub for messaging, BigQuery for analytics data stores, and BigTable for application databases. It's also a managed solution that can autoscale to meet demand. While the original version of Dataflow was specific to GCP, it's now based on the Apache Beam library which is portable to other platforms.

1.4 Coding Environments

There's a variety of options for writing Python code in order to do data science. The best environment to use likely varies based on what you are building, but notebook environments are becoming more and more common as the place to write Python scripts. The three types of coding environments I've worked with for Python are IDEs, text editors, and notebooks.

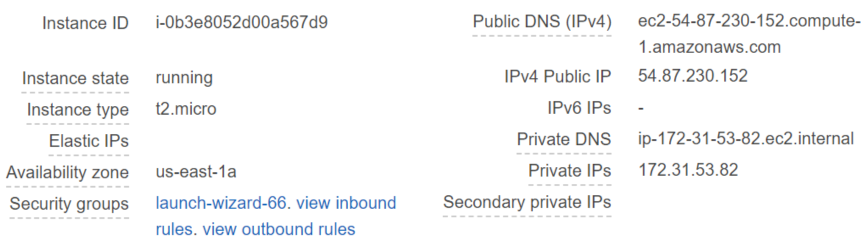
If you're used to working with an IDE, tools like PyCharm and Rodeo are useful editors and provide additional tools for debugging versus other options. It's also possible to write code in text editors such as Sublime and then run scripts via the command line. I find this works well for building web applications with Flask and Dash, where you need to have a long running script that persists beyond the scope of running a cell in a notebook. I now perform the majority of my data science work in notebook environments, and this covers exploratory analysis and productizing models.

I like to work in coding environments that make it trivial to share code and collaborate on projects. Databricks and Google Colab are two coding environments that provide truly collaborative notebooks, where multiple data scientists can simultaneously work on a script. When using Jupyter notebooks, this level of real-time collaboration is not currently supported, but it's good practice to share notebooks in version control systems such as GitHub for sharing work.

In this book, we'll use only the text editor and notebook environments for coding. For learning how to build scalable pipelines, I recommend working on a remote machine, such as EC2, to become more familiar with cloud environments, and to build experience setting up Python environments outside of your local machine.

1.4.1 Jupyter on EC2

To get experience with setting up a remote machine, we'll start by setting up a Jupyter notebook environment on a EC2 instance in



Instance ID	i-0b3e8052d00a567d9	Public DNS (IPv4)	ec2-54-87-230-152.compute-1.amazonaws.com
Instance state	running	IPv4 Public IP	54.87.230.152
Instance type	t2.micro	IPv6 IPs	-
Elastic IPs		Private DNS	ip-172-31-53-82.ec2.internal
Availability zone	us-east-1a	Private IPs	172.31.53.82
Security groups	launch-wizard-66. view inbound rules. view outbound rules	Secondary private IPs	

FIGURE 1.1: Public and Private IPs on EC2.

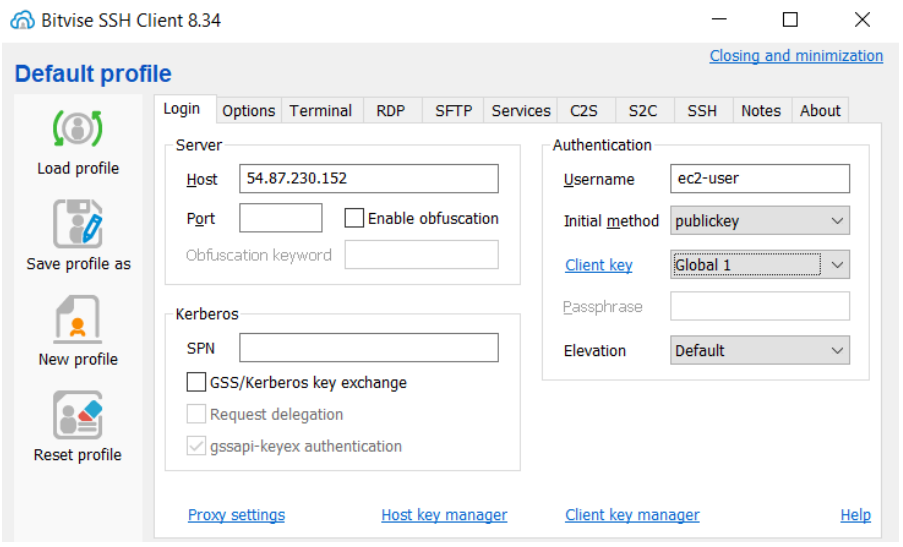
AWS. The result is a remote machine that we can use for Python scripting. Accomplishing this task requires spinning up an EC2 instance, configuring firewall settings for the EC2 instance, connecting to the instance using SSH, and running a few commands to deploy a Jupyter environment on the machine.

The first step is to set up an AWS account and log into the AWS management console. AWS provides a free account option with free-tier access to a number of services including EC2. Next, provision a machine using the following steps:

1. Under "Find Services", search for EC2
2. Click "Launch Instance"
3. Select a free-tier Amazon Linux AMI
4. Click "Review and Launch", and then "Launch"
5. Create a key pair and save to your local machine
6. Click "Launch Instances"
7. Click "View Instances"

The machine may take a few minutes to provision. Once the machine is ready, the instance state will be set to "running". We can now connect to the machine via SSH. One note on the different AMI options is that some of the configurations are set up with Python already installed. However, this book focuses on Python 3 and the included version is often 2.7.

There's two different IPs that you need in order to connect to the machine via SSH and later connect to the machine via web browser. The public and private IPs are listed under the "Description" tab


Bitvise SSH Client 8.34

Closing and minimization

Default profile

Load profile
Save profile as
New profile
Reset profile

Login | Options | Terminal | RDP | SFTP | Services | C2S | S2C | SSH | Notes | About

Server
Host: 54.87.230.152
Port:
☐ Enable obfuscation
Obfuscation keyword:

Kerberos
SPN:
☐ GSS/Kerberos key exchange
☐ Request delegation
✓ gssapi-keyex authentication

Authentication
Username: ec2-user
Initial method: publickey
Client key: Global 1
Passphrase:
Elevation: Default

Proxy settings | Host key manager | Client key manager | Help

FIGURE 1.2: SSH connection settings.

as shown in Figure 1.1. To connect to the machine via SSH we'll use the Public IP (54.87.230.152). For connecting to the machine, you'll need to use an SSH client such as Putty if working in a Windows environment. For Linux and Mac OS, you can use ssh via the command line. To connect to the machine, use the user name "ec2-user" and the key pair generated when launching the instance. An example of connecting to EC2 using the Bitvise client on Windows is shown in Figure 1.2.

Once you connect to the machine, you can check the Python version by running `python --version`. On my machine, the result was `2.7.16`, meaning that additional setup is needed in order to upgrade to Python 3. We can run the following commands to install Python 3, pip, and Jupyter.

```
sudo yum install -y python37
python3 --version
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
sudo python3 get-pip.py
```

```
pip --version
pip install --user jupyter
```

The two version commands are to confirm that the machine is pointing at Python 3 for both Python and pip. Once Jupyter is installed, we'll need to set up a firewall restriction so that we can connect directly to the machine on port 8888, where Jupyter runs by default. This approach is the quickest way to get connected to the machine, but it's advised to use SSH tunneling to connect to the machine rather than a direct connection over the open web. You can open up port 8888 for your local machine by performing the following steps from the EC2 console:

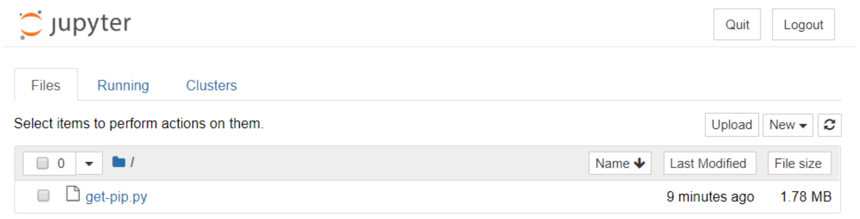
1. Select your EC2 instance
2. Under "Description", select security groups
3. Click "Actions" -> "Edit Inbound Rules"
4. Add a new rule: change the port to 8888, select "My IP"
5. Click "Save"

We can now run and connect to Jupyter on the EC2 machine. To launch Jupyer, run the command shown below while replacing the IP with your EC2 instance's Private IP. It is necessary to specify the `--ip` parameter in order to enable remote connections to Jupyer, as incoming traffic will be routed via the private IP.

```
jupyter notebook --ip 172.31.53.82
```

When you run the `jupyter notebook` command, you'll get a URL with a token that can be used to connect to the machine. Before entering the URL into your browser, you'll need to swap the Private IP output to the console with the Public IP of the EC2 instance, as shown in the snippet below.

```
# Original URL
The Jupyter Notebook is running at:
```



Select items to perform actions on them.

	Name	Last Modified	File size
	get-pip.py	9 minutes ago	1.78 MB

FIGURE 1.3: Jupyter Notebook on EC2.

```
http://172.31.53.82:8888/?token=
    98175f620fd68660d26fa7970509c6c49ec2afc280956a26

# Swap Private IP with Public IP
http://54.87.230.152:8888/?token=
    98175f620fd68660d26fa7970509c6c49ec2afc280956a26
```

You can now paste the updated URL into your browser to connect to Jupyter on the EC2 machine. The result should be a Jupyer notebook fresh install with a single file `get-pip.py` in the base directory, as shown in Figure 1.3. Now that we have a machine set up with Python 3 and Jupyter notebook, we can start exploring different data sets for data science.

1.5 Datasets

To build scalable data pipelines, we'll need to switch from using local files, such as CSVs, to distributed data sources, such as Parquet files on S3. While the tools used across cloud platforms to load data vary significantly, the end result is usually the same, which is a dataframe. In a single machine environment, we can use Pandas to load the dataframe, while distributed environments use different implementations such as Spark dataframes in PySpark.

This section will introduce the data sets that we'll explore throughout the rest of the book. In this chapter we'll focus on loading the data using a single machine, while later chapters will present distributed approaches. While most of the data sets presented here can be downloaded as CSV files as read into Pandas using `read_csv`, it's good practice to develop automated workflows to connect to diverse data sources. We'll explore the following datasets throughout this book:

- **Boston Housing:** Records of sale prices of homes in the Boston housing market back in 1980.
- **Game Purchases:** A synthetic data set representing games purchased by different users on XBox One.
- **Natality:** One of BigQuery's open data sets on birth statistics in the US over multiple decades.
- **Kaggle NHL:** Play-by-play events from professional hockey games and game statistics over the past decade.

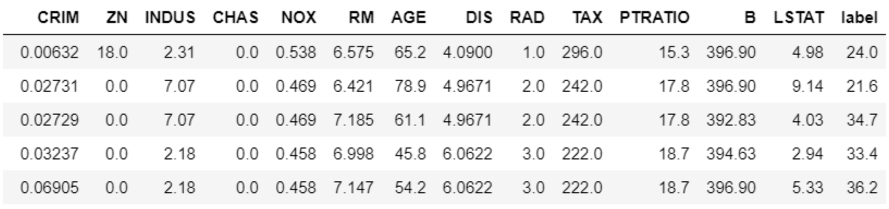
The first two data sets are single commands to load, as long as you have the required libraries installed. The Natality and Kaggle NHL data sets require setting up authentication files before you can programmatically pull the data sources into Pandas.

The first approach we'll use to load a data set is to retrieve it directly from a library. Multiple libraries include the Boston housing data set, because it is a small data set that is useful for testing out regression models. We'll load it from scikit-learn by first running pip from the command line:

```
pip install --user pandas
pip install --user sklearn
```

Once scikit-learn is installed, we can switch back to the Jupyter notebook to explore the data set. The code snippet below shows how to load the scikit-learn and Pandas libraries, load the Boston data set as a Pandas dataframe, and display the first 5 records. The result of running these commands is shown in Figure 1.4.

```
from sklearn.datasets import load_boston
import pandas as pd
```



	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	label
	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2

FIGURE 1.4: Boston Housing data set.

```
data, target = load_boston(True)
bostonDF = pd.DataFrame(data, columns=load_boston().feature_names)
bostonDF['label'] = target
bostonDF.head()
```

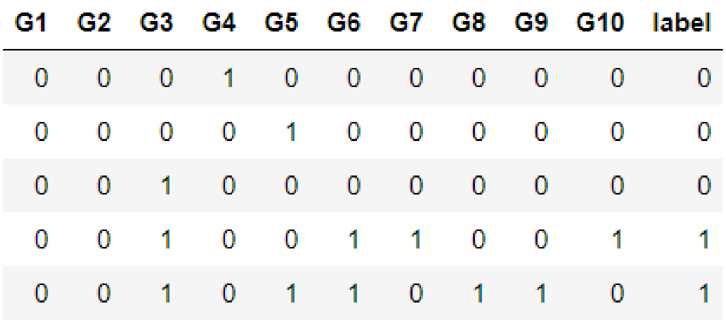
The second approach we'll use to load a data set is to fetch it from the web. The CSV for the Games data set is available as a single file on GitHub. We can fetch it into a Pandas dataframe by using the `read_csv` function and passing the URL of the file as a parameter. The result of reading the data set and printing out the first few records is shown in Figure 1.5.

```
gamesDF = pd.read_csv("https://github.com/bgweber/
  Twitch/raw/master/Recommendations/games-expand.csv")
gamesDF.head()
```

Both of these approaches are similar to downloading CSV files and reading them from a local directory, but by using these methods we can avoid the manual step of downloading files.

1.5.1 BigQuery to Pandas

One of the ways to automate workflows authored in Python is to directly connect to data sources. For databases, you can use connectors based on JDBC or native connectors, such as the `bigquery` module provided by the Google Cloud library. This connector enables Python applications to send queries to BigQuery and load the results as a Pandas dataframe. This process involves setting up



G1	G2	G3	G4	G5	G6	G7	G8	G9	G10	label
0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	1	1	0	0	1	1
0	0	1	0	1	1	0	1	1	0	1

FIGURE 1.5: Game Purchases data set.

a GCP project, installing the prerequisite Python libraries, setting up the Google Cloud command line tools, creating GCP credentials, and finally sending queries to BigQuery programmatically.

If you do not already have a GCP account set up, you'll need to create a new account[3]. Google provides a $300 credit for getting up and running with the platform. The first step is to install the Google Cloud library by running the following steps:

```
pip install --user google-cloud-bigquery
pip install --user matplotlib
```

Next, we'll need to set up the Google Cloud command line tools, in order to set up credentials for connecting to BigQuery. While the files to use will vary based on the current release[4], here are the steps I ran on the command line:

```
curl -O https://dl.google.com/dl/cloudsdk/channels/
        rapid/downloads/google-cloud-sdk-255.0.0-
        linux-x86_64.tar.gz
tar zxvf google-cloud-sdk-255.0.0-linux-x86_64.tar.gz
        google-cloud-sdk
./google-cloud-sdk/install.sh
```

[3] https://cloud.google.com/gcp

[4] https://cloud.google.com/sdk/install

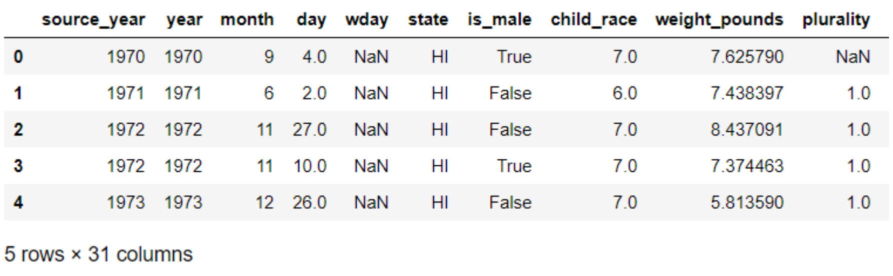
Once the Google Cloud command line tools are installed, we can set up credentials for connecting to BigQuery:

```
gcloud config set project project_name
gcloud auth login
gcloud init
gcloud iam service-accounts create dsdemo
gcloud projects add-iam-policy-binding your_project_id
  --member "serviceAccount:dsdemo@your_project_id.iam.
            gserviceaccount.com" --role "roles/owner"
gcloud iam service-accounts keys
       create dsdemo.json --iam-account
       dsdemo@your_project_id.iam.gserviceaccount.com
export GOOGLE_APPLICATION_CREDENTIALS=
         /home/ec2-user/dsdemo.json
```

You'll need to substitute `project_name` with your project name, `your_project_id` with your project ID, and `dsdemo` with your desired service account name. The result is a json file with credentials for the service account. The export command at the end of this process tells Google Cloud where to find the credentials file.

Setting up credentials for Google Cloud is involved, but generally only needs to be performed once. Now that credentials are configured, it's possible to directly query BigQuery from a Python script. The snippet below shows how to load a BigQuery client, send a SQL query to retrieve 10 rows from the natality data set, and pull the results into a Pandas dataframe. The resulting dataframe is shown in Figure 1.6.

```
from google.cloud import bigquery
client = bigquery.Client()
sql = """
  SELECT *
  FROM  `bigquery-public-data.samples.natality`
  limit 10
"""
```



	source_year	year	month	day	wday	state	is_male	child_race	weight_pounds	plurality
0	1970	1970	9	4.0	NaN	HI	True	7.0	7.625790	NaN
1	1971	1971	6	2.0	NaN	HI	False	6.0	7.438397	1.0
2	1972	1972	11	27.0	NaN	HI	False	7.0	8.437091	1.0
3	1972	1972	11	10.0	NaN	HI	True	7.0	7.374463	1.0
4	1973	1973	12	26.0	NaN	HI	False	7.0	5.813590	1.0

5 rows × 31 columns

FIGURE 1.6: Previewing the BigQuery data set.

```
natalityDF = client.query(sql).to_dataframe()
natalityDF.head()
```

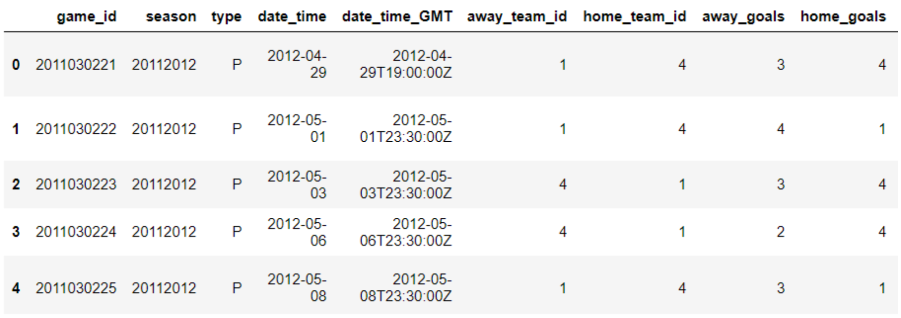
1.5.2 Kaggle to Pandas

Kaggle is a data science website that provides thousands of open data sets to explore. While it is not possible to pull Kaggle data sets directly into Pandas dataframes, we can use the Kaggle library to programmatically download CSV files as part of an automated workflow.

The quickest way to get set up with this approach is to create an account on Kaggle[5]. Next, go to the account tab of your profile and select 'Create API Token', download and open the file, and then run `vi .kaggle/kaggle.json` on your EC2 instance to copy over the contents to your remote machine. The result is a credential file you can use to programmatically download data sets. We'll explore the NHL (Hockey) data set by running the following commands:

```
pip install kaggle --user

kaggle datasets download martinellis/nhl-game-data
unzip nhl-game-data.zip
chmod 0600 *.csv
```

[5]https://www.kaggle.com



	game_id	season	type	date_time	date_time_GMT	away_team_id	home_team_id	away_goals	home_goals
0	2011030221	20112012	P	2012-04-29	2012-04-29T19:00:00Z	1	4	3	4
1	2011030222	20112012	P	2012-05-01	2012-05-01T23:30:00Z	1	4	4	1
2	2011030223	20112012	P	2012-05-03	2012-05-03T23:30:00Z	4	1	3	4
3	2011030224	20112012	P	2012-05-06	2012-05-06T23:30:00Z	4	1	2	4
4	2011030225	20112012	P	2012-05-08	2012-05-08T23:30:00Z	1	4	3	1

FIGURE 1.7: NHL Kaggle data set.

These commands will download the data set, unzip the files into the current directory, and enable read access on the files. Now that the files are downloaded on the EC2 instance, we can load and display the Game data set, as shown in Figure 1.7. This data set includes different files, where the `game` file provides game-level summaries and the `game_plays` file provides play-by-play details.

```
import pandas as pd
nhlDF = pd.read_csv('game.csv')
nhlDF.head()
```

We walked through a few different methods for loading data sets into a Pandas dataframe. The common theme with these different approaches is that we want to avoid manual steps in our workflows, in order to automate pipelines.

1.6 Prototype Models

Machine learning is one of the most important steps in the pipeline of a data product. We can use predictive models to identify which users are most likely to purchase an item, or which users are most likely to stop using a product. The goal of this section is to present simple versions of predictive models that we'll later scale up in more complex pipelines. This book will not focus on state-of-theart models, but instead cover tools that can be applied to a variety of different machine learning algorithms.

The library to use for implementing different models will vary based on the cloud platform and execution environment being used to deploy a model. The regression models presented in this section are built with scikit-learn, while the models we'll build out with PySpark use MLlib.

1.6.1 Linear Regression

Regression is a common task for supervised learning, such as predicting the value of a home, and linear regression is a useful algorithm for making predictions for these types of problems. scikit-learn provides both linear and logistic regression models for making predictions. We'll start by using the `LinearRegression` class in scikit-learn to predict home prices for the Boston housing data set. The code snippet below shows how to split the Boston data set into different training and testing data sets and separate data (`train_x`) and label (`train_y`) objects, create and fit a linear regression model, and calculate error metrics on the test data set.

```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

# See Section 1.4 (Boston Hosing data set)
bostonDF = ...

x_train, x_test, y_train, y_test = train_test_split(
  bostonDF.drop(['label'],axis=1),bostonDF['label'],test_size=0.3)

model = LinearRegression()
model.fit(x_train, y_train)

print("R^2: " + str(model.score(x_test, y_test)))
print("Mean Error: " + str(sum(
      abs(y_test - model.predict(x_test) ))/y_test.count()))
```

The `train_test_split` function is used to split up the data set into 70% train and 30% holdout data sets. The first parameter is the data attributes from the Boston dataframe, with the label dropped, and the second parameter is the labels from the dataframe. The two commands at the end of the script calculate the R-squared value based on Pearson correlation, and the mean error is defined as the mean difference between predicted and actual home prices. The output of this script was an R^2 value of 0.699 and mean error of 3.36. Since house prices in this data set are divided by a thousand, the mean error is \$3.36k.

We now have a simple model that we can productize in a number of different environments. In later sections and chapters, we'll explore methods for scaling features, supporting more complex regression models, and automating feature generation.

1.6.2 Logistic Regression

Logistic regression is a supervised classification algorithm that is useful for predicting which users are likely to perform an action, such as purchasing a product. Using scikit-learn, the process is similar to fitting a linear regression model. The main differences from the prior script are the data set being used, the model object instantiated (`LogisticRegression`), and using the `predict_proba` function to calculate error metrics. This function predicts a probability in the continuous range of [0,1] rather than a specific label. The snippet below predicts which users are likely to purchase a specific game based on prior games already purchased:

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score
import pandas as pd


# Games data set
gamesDF = pd.read_csv("https://github.com/bgweber/Twitch/raw/
                       master/Recommendations/games-expand.csv")
```

```
x_train, x_test, y_train, y_test = train_test_split(
    gamesDF.drop(['label'],axis=1),gamesDF['label'],test_size=0.3)

model = LogisticRegression()
model.fit(x_train, y_train)

print("Accuracy: " + str(model.score(x_test, y_test)))
print("ROC: " + str(roc_auc_score(y_test,
                    model.predict_proba(x_test)[:, 1] )))
```

The output of this script is two metrics that describe the performance of the model on the holdout data set. The accuracy metric describes the number of correct predictions over the total number of predictions, and the ROC metric describes the number of correctly classified outcomes based on different model thresholds. ROC is a useful metric to use when the different classes being predicted are imbalanced, with noticeably different sizes. Since most players are unlikely to buy a specific game, ROC is a good metric to utilize for this use case. When I ran this script, the result was an accuracy of 86.6% and an ROC score of 0.757.

Linear and logistic regression models with scikit-learn are a good starting point for many machine learning projects. We'll explore more complex models in this book, but one of the general strategies I take as a data scientist is to quickly deliver a proof of concept, and then iterate and improve a model once it is shown to provide value to an organization.

1.6.3 Keras Regression

While I generally recommend starting with simple approaches when building model pipelines, deep learning is becoming a popular tool for data scientists to apply to new problems. It's great to explore this capability when tackling new problems, but scaling up deep learning in data science pipelines presents a new set of challenges. For example, PySpark does not currently have a native way of distributing the model application phase to big data.

There's plenty of books for getting started with deep learning in Python, such as (Chollet, 2017).

In this section, we'll repeat the same task from the prior section, which is predicting which users are likely to buy a game based on their prior purchases. Instead of using a shallow learning approach to predict propensity scores, we'll use the Keras framework to build a neural network for predicting this outcome. Keras is a general framework for working with deep learning implementations. We can install these dependencies from the command line:

```
pip install --user tensorflow==1.14.0
pip install --user keras==2.2.4
```

This process can take awhile to complete, and based on your environment may run into installation issues. It's recommended to verify that the installation worked by checking your Keras version in a Jupyter notebook:

```
import tensorflow as tf
import keras
from keras import models, layers
import matplotlib.pyplot as plt
keras.__version__
```

The general process for building models with Keras is to set up the structure of the model, compile the model, fit the model, and evaluate the model. We'll start with a simple model to provide a baseline, which is shown in the snippet below. This code creates a network with an input layer, a dropout layer, a hidden layer, and an output layer. The input to the model is 10 binary variables that describe prior games purchased, and the output is a prediction of the likelihood to purchase a specified game.

```
x_train, x_test, y_train, y_test = train_test_split(
  gamesDF.drop(['label'], axis=1),gamesDF['label'],test_size=0.3)
```

```
# define the network structure
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10,)))
model.add(layers.Dropout(0.1))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

# define ROC AUC as a metric
def auc(y_true, y_pred):
    auc = tf.metrics.auc(y_true, y_pred)[1]
    keras.backend.get_session().run(
                  tf.local_variables_initializer())
    return auc

# compile and fit the model
model.compile(optimizer='rmsprop',
                 loss='binary_crossentropy', metrics=[auc])
history = model.fit(x_train, y_train, epochs=100, batch_size=100,
                 validation_split = .2, verbose=0)
```

Since the goal is to identify the likelihood of a player to purchase a game, ROC is a good metric to use to evaluate the performance of a model. Keras does not support this directly, but we can define a custom metrics function that wraps the `auc` functionality provided by TensorFlow.

Next, we specify how to optimize the model. We'll use `rmsprop` for the optimizer and `binary_crossentropy` for the loss function.The last step is to train the model. The code snippet shows how to fit the model using the training data set, 100 training epochs with a batch size of 100, and a cross validation split of 20%. This process can take awhile to run if you increase the number of epochs or decrease the batch size. The validation set is sampled from only the training data set.

The result of this process is a history object that tracks the loss and metrics on the training and validation data sets. The code

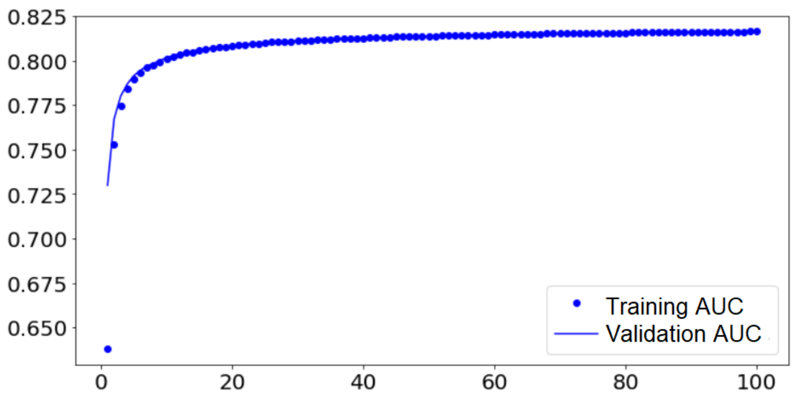


FIGURE 1.8: ROC AUC metrics for the training process.

snippet below shows how to plot these values using Matplotlib. The output of this step is shown in Figure 1.8. The plot shows that both the training and validation data sets leveled off at around a 0.82 AUC metric during the model training process. To compare this approach with the logistic regression results, we'll evaluate the performance of the model on the holdout data set.

```
loss = history.history['auc']
val_loss = history.history['val_auc']
epochs = range(1, len(loss) + 1)

plt.figure(figsize=(10,6) )
plt.plot(epochs, loss, 'bo', label='Training AUC')
plt.plot(epochs, val_loss, 'b', label='Validation AUC')
plt.legend()
plt.show()
```

To measure the performance of the model on the test data set, we can use the `evaluate` function to measure the ROC metric. The code snippet below shows how to perform this task on our training data set, which results in an ROC AUC value of 0.816. This is noticeably better than the performance of the logistic regression model, with an AUC value of 0.757, but using other shallow learning methods such as random forests or XGBoost would likely perform much better on this task.

```
results = model.evaluate(x_test, y_test, verbose = 0)
print("ROC: " + str(results[1]))
```

1.7 Automated Feature Engineering

Automated feature engineering is a powerful tool for reducing the amount of manual work needed in order to build predictive models. Instead of a data scientist spending days or weeks coming up with the best features to describe a data set, we can use tools that approximate this process. One library I've been working with to implement this step is FeatureTools. It takes inspiration from the automated feature engineering process in deep learning, but is meant for shallow learning problems where you already have structured data, but need to translate multiple tables into a single record per user. The library can be installed as follows:

```
sudo yum install gcc
sudo yum install python3-devel
pip install --user framequery
pip install --user fsspec
pip install --user featuretools
```

In addition to this library, I loaded the framequery library, which enables writing SQL queries against dataframes. Using SQL to work with dataframes versus specific interfaces, such as Pandas, is useful when translating between different execution environments.

The task we'll apply the FeatureTools library to is predicting which games in the Kaggle NHL data set are postseason games. We'll make this prediction based on summarizations of the play events that are recorded for each game. Since there can be hundreds of play events per game, we need a process for aggregating these into a single summary per game. Once we aggregate these events into a single game record, we can apply a logistic regression model to predict whether the game is regular or postseason.

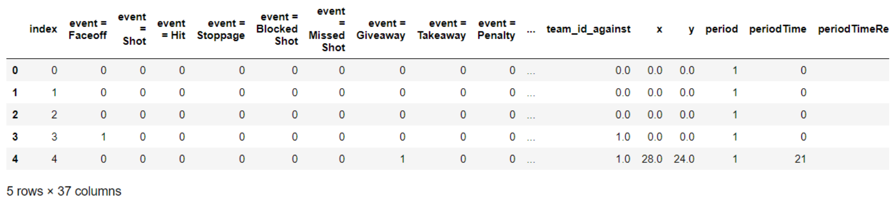
The first step we'll perform is loading the data sets and performing some data preparation, as shown below. After loading the data sets as Pandas dataframes, we drop a few attributes from the plays object, and fill any missing attributes with 0.

```
import pandas as pd

game_df = pd.read_csv("game.csv")
plays_df = pd.read_csv("game_plays.csv")

plays_df = plays_df.drop(['secondaryType', 'periodType',
                 'dateTime', 'rink_side'], axis=1).fillna(0)
```

To translate the play events into a game summary, we'll first 1-hot econde two of the attributes in the plays dataframe, and then perform deep feature synthesis. The code snippet below shows how to perform the first step, and uses FeatureTools to accomplish this task. The result is quite similar to using the `get_dummies` function in Pandas, but this approach requires some additional steps.

The base representation in FeatureTools is an EntitySet, which describes a set of tables and the relationships between them, which is similar to defining foreign key constraints. To use the `encode_features` function, we need to first translate the plays dataframe into an entity. We can create an EntitySet directly from the `plays_df` object, but we also need to specify which attributes should be handled as categorical, using the `variable_types` dictionary parameter.

```
import featuretools as ft
from featuretools import Feature

es = ft.EntitySet(id="plays")
```



	index	event = Faceoff	event = Shot	event = Hit	event = Stoppage	event = Blocked Shot	event = Missed Shot	event = Giveaway	event = Takeaway	event = Penalty	...	team_id_against	x	y	period	periodTime	periodTimeRe
0	0	0	0	0	0	0	0	0	0	0	...	0.0	0.0	0.0	1	0	
1	1	0	0	0	0	0	0	0	0	0	...	0.0	0.0	0.0	1	0	
2	2	0	0	0	0	0	0	0	0	0	...	0.0	0.0	0.0	1	0	
3	3	1	0	0	0	0	0	0	0	0	...	1.0	0.0	0.0	1	0	
4	4	0	0	0	0	0	0	1	0	0	...	1.0	28.0	24.0	1	21	

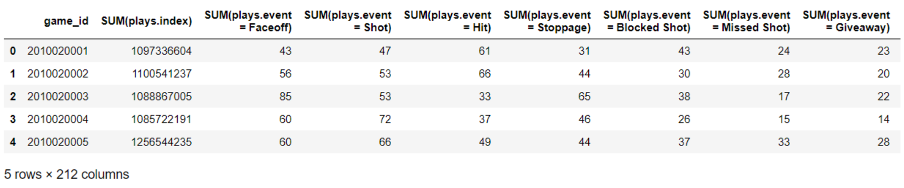
5 rows × 37 columns

FIGURE 1.9: The 1-hot encoded Plays dataframe.

```
es = es.entity_from_dataframe(entity_id="plays",dataframe=plays_df
           ,index="play_id", variable_types = {
              "event": ft.variable_types.Categorical,
               "description": ft.variable_types.Categorical })

f1 = Feature(es["plays"]["event"])
f2 = Feature(es["plays"]["description"])

encoded, defs = ft.encode_features(plays_df, [f1, f2], top_n=10)
encoded.reset_index(inplace=True)
encoded.head()
```

Next, we pass a list of features to the `encode_features` function, which returns a new dataframe with the dummy variables and a `defs` object that describes how to translate an input dataframe into the 1-hot encoded format. For pipelines later on in this book, where we need to apply transformations to new data sets, we'll store a copy of the `defs` object for later use. The result of applying this transformation to the plays dataframe is shown in Figure 1.9.

The next step is to aggregate the hundreds of play events per game into single game summaries, where the resulting dataframe has a single row per game.To accomplish this task, we'll recreate the EntitySet from the prior step, but use the 1-hot encoded dataframe as the input. Next, we use the `normalize_entity` function to describe games as a parent object to plays events, where all plays with the same `game_id` are grouped together. The last step is to use the `dfs` function to perform deep feature synthesis. DFS applies aggregate calculations, such as `SUM` and `MAX`, across the different features in



	game_id	SUM(plays.index)	SUM(plays.event = Faceoff)	SUM(plays.event = Shot)	SUM(plays.event = Hit)	SUM(plays.event = Stoppage)	SUM(plays.event = Blocked Shot)	SUM(plays.event = Missed Shot)	SUM(plays.event = Giveaway)
0	2010020001	1097336604	43	47	61	31	43	24	23
1	2010020002	1100541237	56	53	66	44	30	28	20
2	2010020003	1088867005	85	53	33	65	38	17	22
3	2010020004	1085722191	60	72	37	46	26	15	14
4	2010020005	1256544235	60	66	49	44	37	33	28

5 rows × 212 columns

FIGURE 1.10: Generated features for the NHL data set.

the child dataframe, in order to collapse hundreds of records into a single row.

```
es = ft.EntitySet(id="plays")
es = es.entity_from_dataframe(entity_id="plays",
                      dataframe=encoded, index="play_id")
es = es.normalize_entity(base_entity_id="plays",
                      new_entity_id="games", index="game_id")

features,transform=ft.dfs(entityset=es,
                        target_entity="games",max_depth=2)
features.reset_index(inplace=True)
features.head()
```

The result of this process is shown in Figure 1.10. The shape of the sampled dataframe, 5 rows by 212 columns, indicates that we have generated hundreds of features to describe each game using deep feature synthesis. Instead of hand coding this translation, we utilized the FeatureTools library to automate this process.

Now that we have hundreds of features for describing a game, we can use logistic regression to make predictions about the games. For this task, we want to predict whether a game is regular season or postseason, where `type = 'P'`. The code snippet below shows how to use the framequery library to combine the generated features with the initially loaded games dataframe using a SQL join. We use the `type` attribute to assign a label, and then return all of the generated features and the label. The result is a dataframe that we can pass to scikit-learn.

```
import framequery as fq

# assign labels to the generated features
features = fq.execute("""
  SELECT f.*
    ,case when g.type = 'P' then 1 else 0 end as label
  FROM features f
  JOIN game_df g
    on f.game_id = g.game_id
""")
```

We can re-use the logistic regression code from above to build a model that predicts whether an NHL game is a regular or postseason game. The updated code snippet to build a logistic regression model with scikit-learn is shown below. We drop the `game_id` column before fitting the model to avoid training the model on this attribute, which typically results in overfitting.

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score

# create inputs for sklearn
y = features['label']
X = features.drop(['label', 'game_id'], axis=1).fillna(0)

# train a classifier
lr = LogisticRegression()
model = lr.fit(X, y)

# Results
print("Accuracy: " + str(model.score(X, y)))
print("ROC" + str(roc_auc_score(y,model.predict_proba(X)[:,1])))
```

The result of this model was an accuracy of 94.7% and an ROC measure of 0.923. While we likely could have created a better performing model by manually specifying how to aggregate play events into a game summary, we were able to build a model with good accuracy while automating much of this process.

1.8 Conclusion

Building data products is becoming an essential competency for applied data scientists. The Python ecosystem provides useful tools for taking prototype models and scaling them up to production-quality systems. In this chapter, we laid the groundwork for the rest of this book by introducing the data sets, coding tools, cloud environments, and predictive models that we'll use to build scalable model pipelines. We also explored a recent Python library called FeatureTools, which enables automating much of the feature engineering steps in a model pipeline.

In our current setup, we built a simple batch model on a single machine in the cloud. In the next chapter, we'll explore how to share our models with the world, by exposing them as endpoints on the web.

2

Models as Web Endpoints

In order for a machine learning model to be useful, you need a way of sharing the results with other services and applications within your organization. While you can precompute results and save them to a database using a batch pipeline approach, it's often necessary to respond to requests in real-time with up-to-date information. One way of achieving this goal is by setting up a predictive model as a web endpoint that can be invoked from other services. This chapter shows how to set up this functionality for both scikit-learn and Keras models, and introduces Python tools that can help scale up this functionality.

It's good to build experience both hosting and consuming web endpoints when building out model pipelines with Python. In some cases, a predictive model will need to pull data points from other services before making a prediction, such as needing to pull additional attributes about a user's history as input to feature engineering. In this chapter, we'll focus on JSON based services, because it is a popular data format and works well with Python's data types.

A model as an endpoint is a system that provides a prediction in response to a passed in set of parameters. These parameters can be a feature vector, image, or other type of data that is used as input to a predictive model. The endpoint then makes a prediction and returns the results, typically as a JSON payload. The benefits of setting up a model this way are that other systems can use the predictive model, it provides a real-time result, and can be used within a broader data pipeline.

In this chapter, we'll cover calling web services using Python, setting up endpoints, saving models so that they can be used in production environments, hosting scikit-learn and Keras predictive models, scaling up a service with Gunicorn and Heroku, and building an interactive web application with Plotly Dash.

2.1 Web Services

Before we host a predictive model, we'll use Python to call a web service and to process the result. After showing how to process a web response, we'll set up our own service that echoes the passed in message back to the caller. There's a few different libraries we'll need to install for the examples in this chapter:

```
pip install --user requests
pip install --user flask
pip install --user gunicorn
pip install --user mlflow
pip install --user pillow
pip install --user dash
```

These libraries provide the following functionality:

- **requests:** Provides functions for GET and POST commands.
- **flask:** Enables functions to be exposed as HTTP locations.
- **gunicorn:** A WSGI server that enables hosting Flask apps in production environments.
- **mlflow:** A model library that provides model persistence.
- **pillow:** A fork of the Python Imaging Library.
- **dash:** Enables writing interactive web apps in Python.

Many of the tools for building web services in the Python ecosystem work well with Flask. For example, Gunicorn can be used to host Flask applications at production scale, and the Dash library builds on top of Flask.

To get started with making web requests in Python, we'll use the Cat Facts Heroku app[1]. Heroku is a cloud platform that works

[1] https://cat-fact.herokuapp.com/#/

well for hosting Python applications that we'll explore later in this chapter. The Cat Facts service provides a simple API that provides a JSON response containing interesting tidbits about felines. We can use the `/facts/random` endpoint to retrieve a random fact using the requests library:

```
import requests

result = requests.get("http://cat-fact.herokuapp.com/facts/random")
print(result)
print(result.json())
print(result.json()['text'])
```

This snippet loads the requests library and then uses the `get` function to perform an HTTP get for the passed in URL. The result is a response object that provides a response code and payload if available. In this case, the payload can be processed using the `json` function, which returns the payload as a Python dictionary. The three print statements show the response code, the full payload, and the value for the `text` key in the returned dictionary object. The output for a run of this script is shown below.

```
<Response [200]>

{'used': False, 'source': 'api', 'type': 'cat', 'deleted': False
,'_id': '591f98c5d1f17a153828aa0b', '__v': 0, 'text':
'Domestic cats purr both when inhaling and when exhaling.',
'updatedAt': '2019-05-19T20:22:45.768Z',
'createdAt': '2018-01-04T01:10:54.673Z'}

Domestic cats purr both when inhaling and when exhaling.
```

2.1.1 Echo Service

Before we set up a complicated environment for hosting a predictive model, we'll start with a simple example. The first service we'll set up is an echo application that returns the passed in message parameter as part of the response payload. To implement this functionality, we'll use Flask to build a web service hosted on an EC2 instance. This service can be called on the open web, using the public IP of the EC2 instance. You can also run this service on your local machine, but it won't we accessible over the web. In order to access the function, you'll need to enable access on port 5000, which is covered in Section 1.4.1. The complete code for the echo web service is shown below:

```
import flask
app = flask.Flask(__name__)

@app.route("/", methods=["GET","POST"])
def predict():
    data = {"success": False}

    # check for passed in parameters
    params = flask.request.json
    if params is None:
        params = flask.request.args

    # if parameters are found, echo the msg parameter
    if "msg" in params.keys():
        data["response"] = params.get("msg")
        data["success"] = True

    return flask.jsonify(data)

if __name__ == '__main__':
    app.run(host='0.0.0.0')
```

The first step is loading the Flask library and creating a Flask object using the `name` special variable. Next, we define a `predict` function with a Flask annotation that specifies that the function should be hosted at "/" and accessible by HTTP `GET` and `POST` commands. The last step specifies that the application should run using `0.0.0.0` as the host, which enables remote machines to access the application. By default, the application will run on port 5000, but it's possible to override this setting with the `port` parameter. When running Flask directly, we need to call the `run` function, but we do not want to call the command when running as a module within another application, such as Gunicorn.

The predict function returns a JSON response based on the passed in parameters. In Python, you can think of a JSON response as a dictionary, because the `jsonify` function in Flask makes the translation between these data formats seamless. The function first defines a dictionary with the success key set to `False`. Next, the function checks if the `request.json` or `request.args` values are set, which indicates that the caller passed in arguments to the function, which we'll cover in the next code snippet. If the user has passed in a `msg` parameter, the `success` key is set to `True` and a `response` key is set to the `msg` parameter in the dictionary. The result is then returned as a JSON payload.

Instead of running this in a Jupyter notebook, we'll save the script as a file called `echo.py`. To launch the Flask application, run `python3 echo.py` on the command line. The result of running this command is shown below:

```
python3 echo.py
 * Serving Flask app "echo" (lazy loading)
 * Environment: production
   WARNING: This is a development server.
   Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

The output indicates that the service is running on port 5000. If you launched the service on your local machine, you can browse to `http://localhost:5000` to call the application, and if using EC2 you'll need to use the public IP, such as `http://52.90.199.190:5000`. The result will be `{"response":null,"success":false}`, which indicates that the service was called but that no message was provided to the echo service.

We can pass parameters to the web service using a few different approaches. The parameters can be appended to the URL, specified using the `params` object when using a GET command, or passed in using the `json` parameter when using a POST command. The snippet below shows how to perform these types of requests. For small sets of parameters, the GET approach works fine, but for larger parameters, such as sending images to a server, the POST approach is preferred.

```
import requests

result = requests.get("http://52.90.199.190:5000/?msg=HelloWorld!")
print(result.json())

result = requests.get("http://52.90.199.190:5000/",
                        params = { 'msg': 'Hello from params' })
print(result.json())

result = requests.post("http://52.90.199.190:5000/",
                        json = { 'msg': 'Hello from data' })
print(result.json())
```

The output of the code snippet is shown below. There are 3 JSON responses showing that the service successfully received the message parameter and echoed the response:

```
{'response': 'HelloWorld!', 'success': True}
{'response': 'Hello from params', 'success': True}
{'response': 'Hello from data', 'success': True}
```

In addition to passing values to a service, it can be useful to pass larger payloads, such as images when hosting deep learning models. One way of achieving this task is by encoding images as strings, which will work with our existing echo service. The code snippet

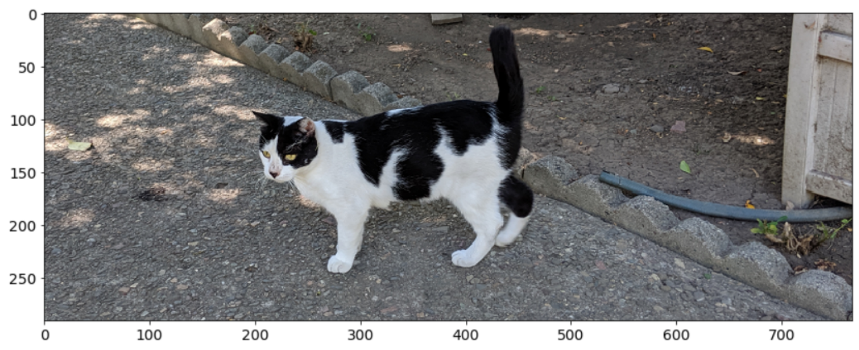


FIGURE 2.1: Passing an image to the echo web service.

below shows how to read in an image and perform base64 encoding on the image before adding it to the request object. The echo service responds with the image payload and we can use the `PIL` library to render the image as a plot.

```
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image
import io
import base64

image = open("luna.png", "rb").read()
encoded = base64.b64encode(image)
result = requests.get("http://52.90.199.190:5000/",
                       json = {'msg': encoded})
encoded = result.json()['response']
imgData = base64.b64decode(encoded)
plt.imshow( np.array(Image.open(io.BytesIO(imgData))))
```

We can run the script within a Jupyter notebook. The script will load the image and send it to the server, and then render the result as a plot. The output of this script, which uses an image of my in-laws' cat, is shown in Figure 2.1. We won't work much with image data in this book, but I did want to cover how to use more complex objects with web endpoints.

2.2 Model Persistence

To host a model as a web service, we need to provide a model object for the predict function. We can train the model within the web service application, or we can use a pre-trained model. Model persistence is a term used for saving and loading models within a predictive model pipeline. It's common to train models in a separate workflow than the pipeline used to serve the model, such as a Flask application. In this section, we'll save and load both scikit-learn and Keras models, with both direct serialization and the MLFlow library. The goal of saving and loading these models is to make the logistic regression and deep learning models we built in Chapter 1 available as web endpoints.

2.2.1 Scikit-Learn

We'll start with scikit-learn, which we previously used to build a propensity model for identifying which players were most likely to purchase a game. A simple `LogisticRegression` model object can be created using the following script:

```
import pandas as pd
from sklearn.linear_model import LogisticRegression

df = pd.read_csv("https://github.com/bgweber/Twitch/
                  raw/master/Recommendations/games-expand.csv")
x = df.drop(['label'], axis=1)
y = df['label']

model = LogisticRegression()
model.fit(x, y)
```

The default way of saving scikit-learn models is by using pickle, which provides serialization of Python objects. You can save a model using `dump` and load a model using the `load` function, as shown below. Once you have loaded a model, you can use the prediction functions, such as `predict_proba`.

```
import pickle
pickle.dump(model, open("logit.pkl", 'wb'))

model = pickle.load(open("logit.pkl", 'rb'))
model.predict_proba(x)
```

Pickle is great for simple workflows, but can run into serialization issues when your execution environment is different from your production environment. For example, you might train models on your local machine using Python 3.7 but need to host the models on an EC2 instance running Python 3.6 with different library versions installed.

MLflow is a broad project focused on improving the lifecycle of machine learning projects. The `Models` component of this platform focuses on making models deployable across a diverse range of execution environments. A key goal is to make models more portable, so that your training environment does not need to match your deployment environment. In the current version of MLflow, many of the save and load functions wrap direct serialization calls, but future versions will be focused on using generalized model formats.

We can use MLflow to save a model using `sklearn.save_model` and load a model using `sklearn.load_model`. The script below shows how to perform the same task as the prior code example, but uses MLflow in place of pickle. The file is saved at the `model_path` location, which is a relative path. There's also a commented out command, which needs to be uncommented if the code is executed multiple times. MLflow currently throws an exception if a model is already saved at the current location, and the `rmtee` command can be used to overwrite the existing model.

```
import mlflow
import mlflow.sklearn
import shutil

model_path = "models/logit_games_v1"
#shutil.rmtree(model_path)
mlflow.sklearn.save_model(model, model_path)

loaded = mlflow.sklearn.load_model(model_path)
loaded.predict_proba(x)
```

2.2.2 Keras

Keras provides built-in functionality for saving and loading deep learning models. We covered building a Keras model for the games data set in Section 1.6.3. The key steps in this process are shown in the following snippet:

```
import tensorflow as tf
import keras
from keras import models, layers

# define the network structure
model = models.Sequential()
model.add(layers.Dense(64,activation='relu',input_shape=(10,)))
model.add(layers.Dropout(0.1))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

def auc(y_true, y_pred):
    auc = tf.metrics.auc(y_true, y_pred)[1]
    keras.backend.get_session().run(
                  tf.local_variables_initializer())
    return auc

model.compile(optimizer='rmsprop',
                 loss='binary_crossentropy', metrics=[auc])
```

```
history = model.fit(x, y, epochs=100, batch_size=100,
                  validation_split = .2, verbose=0)
```

Once we have trained a Keras model, we can use the `save` and `load_model` functions to persist and reload the model using the h5 file format. One additional step here is that we need to pass the custom `auc` function we defined as a metric to the load function in order to reload the model. Once the model is loaded, we can call the prediction functions, such as `evaluate`.

```
from keras.models import load_model
model.save("games.h5")

model = load_model('games.h5', custom_objects={'auc': auc})
model.evaluate(x, y, verbose = 0)
```

We can also use MLflow for Keras. The `save_model` and `load_model` functions can be used to persist Keras models. As before, we need to provide the custom-defined `auc` function to load the model.

```
import mlflow.keras

model_path = "models/keras_games_v1"
mlflow.keras.save_model(model, model_path)

loaded = mlflow.keras.load_model(model_path,
                      custom_objects={'auc': auc})
loaded.evaluate(x, y, verbose = 0)
```

2.3 Model Endpoints

Now that we know how to set up a web service and load pre-trained predictive models, we can set up a web service that provides a prediction result in response to a passed-in instance. We'll deploy models for the games data set using scikit-learn and Keras.

2.3.1 Scikit-Learn

To use scikit-learn to host a predictive model, we'll modify our echo service built with Flask. The main changes to make are loading a scikit-learn model using MLflow, parsing out the feature vector to pass to the model from the input parameters, and adding the model result to the response payload. The updated Flask application for using scikit-learn is shown in the following snippet:

```
import pandas as pd
from sklearn.linear_model import LogisticRegression
import mlflow
import mlflow.sklearn
import flask

model_path = "models/logit_games_v1"
model   = mlflow.sklearn.load_model(model_path)

app = flask.Flask(__name__)

@app.route("/", methods=["GET","POST"])
def predict():
    data = {"success": False}
    params = flask.request.args

    if "G1" in params.keys():
        new_row = { "G1": params.get("G1"),"G2": params.get("G2"),
                    "G3": params.get("G3"),"G4": params.get("G4"),
                    "G5": params.get("G5"),"G6": params.get("G6"),
                    "G7": params.get("G7"),"G8": params.get("G8"),
                    "G9": params.get("G9"),"G10":params.get("G10")}

        new_x = pd.DataFrame.from_dict(new_row,
                                        orient = "index").transpose()
        data["response"] = str(model.predict_proba(new_x)[0][1])
        data["success"] = True

    return flask.jsonify(data)

if __name__ == '__main__':
    app.run(host='0.0.0.0')
```

After loading the required libraries, we use `load_model` to load the scikit-learn model object using MLflow. In this setup, the model is loaded only once, and will not be updated unless we relaunch the application. The main change from the echo service is creating the feature vector that we need to pass as input to the model's prediction functions. The `new_row` object creates a dictionary using the passed in parameters. To provide the Pandas row format needed by scikit-learn, we can create a Pandas dataframe based on the dictionary and then transpose the result, which creates a dataframe with a single row. The resulting dataframe is then passed to `predict_proba` to make a propensity prediction for the passed in user. The model output is added to the JSON payload under the `response` key.

Similar to the echo service, we'll need to save the app as a Python file rather than running the code directly in Jupyter. I saved the code as predict.py and launched the endpoint by running `python3 predict.py`, which runs the service on port 5000.

To test the service, we can use Python to pass in a record representing an individual user. The dictionary defines the list of games that the user has previously purchased, and the GET command is used to call the service. For the example below, the response key returned a value of `0.3812`. If you are running this script in a Jupyer notebook on an EC2 instance, you'll need to enable remote access for the machine on port 5000. Even though the web service and notebook are running on the same machine, we are using the public IP to reference the model endpoint.

```
import requests

new_row = { "G1": 0, "G2": 0, "G3": 0, "G4": 0, "G5": 0,
            "G6": 0, "G7": 0, "G8": 0, "G9": 0, "G10": 1 }

result = requests.get("http://52.90.199.190:5000/", params=new_row)
print(result.json()['response'])
```

2.3.2 Keras

The setup for Keras is similar to scikit-learn, but there are a few additions that need to be made to handle the TensorFlow graph context. We also need to redefine the `auc` function prior to loading the model using MLflow. The snippet below shows the complete code for a Flask app that serves a Keras model for the game purchases data set.

The main thing to note in this script is the use of the `graph` object. Because Flask uses multiple threads, we need to define the graph used by Keras as a global object, and grab a reference to the graph using the `with` statement when serving requests.

```
import pandas as pd
import mlflow
import mlflow.keras
import flask
import tensorflow as tf
import keras as k

def auc(y_true, y_pred):
    auc = tf.metrics.auc(y_true, y_pred)[1]
    k.backend.get_session().run(
                  tf.local_variables_initializer())
    return auc

global graph
```

```
graph = tf.get_default_graph()
model_path = "models/keras_games_v1"
model = mlflow.keras.load_model(model_path,
                                custom_objects={'auc': auc})

app = flask.Flask(__name__)

@app.route("/", methods=["GET","POST"])
def predict():
    data = {"success": False}
    params = flask.request.args

    if "G1" in params.keys():
        new_row = { "G1": params.get("G1"), "G2": params.get("G2"),
                    "G3": params.get("G3"), "G4": params.get("G4"),
                    "G5": params.get("G5"), "G6": params.get("G6"),
                    "G7": params.get("G7"), "G8": params.get("G8"),
                 "G9": params.get("G9"), "G10": params.get("G10") }

        new_x = pd.DataFrame.from_dict(new_row,
                                      orient = "index").transpose()

        with graph.as_default():
            data["response"] = str(model.predict(new_x)[0][0])
            data["success"] = True

    return flask.jsonify(data)

if __name__ == '__main__':
    app.run(host='0.0.0.0')
```

I saved the script as `keras_predict.py` and then launched the Flask app using `python3 keras_predict.py`. The result is a Keras model running as a web service on port 5000. To test the script, we can run the same script from the following section where we tested a scikit-learn model.

2.4 Deploying a Web Endpoint

Flask is great for prototyping models as web services, but it's not intended to be used directly in a production environment. For a proper deployment of a web application, you'll want to use a WSGI Server, which provides scaling, routing, and load balancing. If you're looking to host a web service that needs to handle a large workload, then Gunicorn provides a great solution. If instead you'd like to use a hosted solution, then Heroku provides a platform for hosting web services written in Python. Heroku is useful for hosting a data science portfolio, but is limited in terms of components when building data and model pipelines.

2.4.1 Gunicorn

We can use Gunicorn to provide a WSGI server for our echo Flask application. Using gunicorn helps separate the functionality of an application, which we implemented in Flask, with the deployment of an application. Gunicorn is a lightweight WSGI implementation that works well with Flask apps.

It's straightforward to switch form using Flask directly to using Gunicorn to run the web service. The new command for running the application is shown below. Note that we are passing in a bind parameter to enable remote connections to the service.

```
gunicorn --bind 0.0.0.0 echo:app
```

The result on the command line is shown below. The main difference from before is that we now interface with the service on port 8000 rather than on port 5000. If you want to test out the service, you'll need to enable remote access on port 8000.

```
gunicorn --bind 0.0.0.0 echo:app
 [INFO] Starting gunicorn 19.9.0
 [INFO] Listening at: http://0.0.0.0:8000 (9509)
```

```
[INFO]  Using worker: sync
[INFO] Booting worker with pid: 9512
```

To test the service using Python, we can run the following snippet. You'll need to make sure that access to port 8000 is enabled, as discussed in Section 1.4.1.

```
result = requests.get("http://52.90.199.190:8000/",
                       params = { 'msg': 'Hello from Gunicorn' })
print(result.json())
```

The result is a JSON response with the passed in message. The main distinction from our prior setup is that we are now using Gunicorn, which can use multiple threads to handle load balancing, and can perform additional server configuration that is not available when using only Flask. Configuring Gunicorn to serve production workloads is outside the scope of this book, because it is a hosted solution where a team needs to manage DevOps of the system. Instead, we'll focus on managed solutions, including AWS Lambda and Cloud Functions in Chapter 3, where minimal overhead is needed to keep systems operational.

2.4.2 Heroku

Now that we have a Gunicorn application, we can host it in the cloud using Heroku. Python is one of the core languages supported by this cloud environment. The great thing about using Heroku is that you can host apps for free, which is great for showcasing data science projects. The first step is to set up an account on the web site: `https://www.heroku.com/`

Next, we'll set up the command line tools for Heroku, by running the commands shown below. There can be some complications when setting up Heroku on an AMI EC2 instance, but downloading and unzipping the binaries directly works around these problems. The steps shown below download a release, extract it, and install an additional dependency. The last step outputs the version of

Heroku installed. I got the following output: `heroku/7.29.0 linux-x64 node-v11.14.0`.

```
wget https://cli-assets.heroku.com/heroku-linux-x64.tar.gz
unzip heroku-linux-x64.tar.gz
tar xf heroku-linux-x64.tar
sudo yum -y install glibc.i686
/home/ec2-user/heroku/bin/heroku --version
```

Once Heroku is installed, we need to set up a project for where we will deploy projects. We can use the CLI to create a new Heroku project by running the following commands:

```
/home/ec2-user/heroku/bin/heroku login
/home/ec2-user/heroku/bin/heroku create
```

This will create a unique app name, such as `obscure-coast-69593`. It's good to test the setup locally before deploying to production. In order to test the setup, you'll need to install the `django` and `django-heroku` packages. Heroku has some dependencies on Postgres, which is why additional `install` and `easy_install` commands are included when installing these libraries.

```
pip install --user django
sudo yum install gcc python-setuptools postgresql-devel
sudo easy_install psycopg2
pip install --user django-heroku
```

To get started with building a Heroku application, we'll first download the sample application from GitHub and then modify the project to include our echo application.

```
sudo yum install git
git clone https://github.com/heroku/python-getting-started.git
cd python-getting-started
```

Next, we'll make our changes to the project. We copy our `echo.py` file into the directory, add Flask to the list of dependencies in the `requirements.txt` file, override the command to run in the `Procfile`, and then call `heroku local` to test the configuration locally.

```
cp ../echo.py echo.py
echo 'flask' >> requirements.txt
echo "web: gunicorn echo:app" > Procfile
/home/ec2-user/heroku/bin/heroku local
```

You should see a result that looks like this:

```
/home/ec2-user/heroku/bin/heroku local
[OKAY] Loaded ENV .env File as KEY=VALUE Format
[INFO] Starting gunicorn 19.9.0
[INFO] Listening at: http://0.0.0.0:5000 (10485)
[INFO] Using worker: sync
[INFO] Booting worker with pid: 10488
```

As before, we can test the endpoint using a browser or a Python call, as shown below. In the Heroku local test configuration, port 5000 is used by default.

```
result = requests.get("http://localhost:5000/",
                      params = { 'msg': 'Hello from Heroku Local'})
print(result.json())
```

The final step is to deploy the service to production. The git commands are used to push the results to Heroku, which automatically releases a new version of the application. The last command tells Heroku to scale up to a single worker, which is free.

```
git add echo.py
git commit .
git push heroku master
/home/ec2-user/heroku/bin/heroku ps:scale web=1
```

After these steps run, there should be a message that the application has been deployed to Heroku. Now we can call the endpoint, which has a proper URL, is secured, and can be used to publicly share data science projects.

```
result = requests.get("https://obscure-coast-69593.herokuapp.com",
                      params = { 'msg': 'Hello from Heroku Prod' })
print(result.json())
```

There's many languages and tools supported by Heroku, and it's useful for hosting small-scale data science projects.

2.5 Interactive Web Services

While the standard deployment of a model as a web service is an API that you can call programmatically, it's often useful to expose models as interactive web applications. For example, we might want to build an application where there is a UI for specifying different inputs to a model, and the UI reacts to changes made by the user. While Flask can be used to build web pages that react to user input, there are libraries built on top of Flask that provide higher-level abstractions for building web applications with the Python language.

2.5.1 Dash

Dash is a Python library written by the Plotly team than enables building interactive web applications with Python. You specify an application layout and a set of callbacks that respond to user input. If you've used Shiny in the past, Dash shares many similarities, but is built on Python rather than R. With Dash, you can create simple applications as we'll show here, or complex dashboards that interact with machine learning models.

We'll create a simple Dash application that provides a UI for interacting with a model. The application layout will contain three text boxes, where two of these are for user inputs and the third one shows the output of the model. We'll create a file called `dash_app.py` and start by specifying the libraries to import.

```
import dash
import dash_html_components as html
import dash_core_components as dcc
from dash.dependencies import Input, Output
import pandas as pd
import mlflow.sklearn
```

Next, we'll define the layout of our application. We create a Dash object and then set the `layout` field to include a title and three text boxes with labels. We'll include only 2 of the 10 games from the games data set, to keep the sample short. The last step in the script launches the web service and enables connections from remote machines.

```
app = dash.Dash(__name__)

app.layout = html.Div(children=[
    html.H1(children='Model UI'),
    html.P([
        html.Label('Game 1 '),
        dcc.Input(value='1', type='text', id='g1'),
    ]),
    html.Div([
        html.Label('Game 2 '),
        dcc.Input(value='0', type='text', id='g2'),
    ]),
    html.P([
        html.Label('Prediction '),
        dcc.Input(value='0', type='text', id='pred')
    ]),
])
```

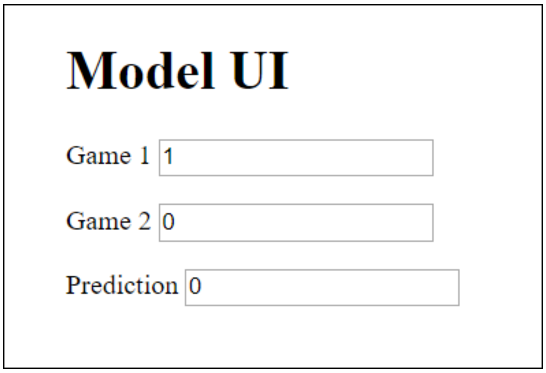


FIGURE 2.2: The initial Dash application.

```
if __name__ == '__main__':
    app.run_server(host='0.0.0.0')
```

Before writing the callbacks, we can test out the layout of the application by running `python3 dash_app.py`, which will run on port 8050 by default. You can browse to your public IP on port 8050 to see the resulting application. The initial application layout is shown in Figure 2.2. Before any callbacks are added, the result of the Prediction text box will always be 0.

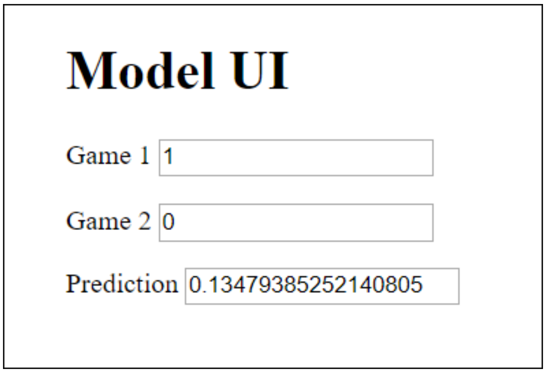
The next step is to add a callback to the application so that the Prediction text box is updated whenever the user changes one of the Game 1 or Game 2 values. To perform this task, we define a callback shown in the snippet below. The callback is defined after the application layout, but before the `run_server` command. We also load the logistic regression model for the games data set using MLflow. The callback uses an annotation to define the inputs to the function, the output, and any additional state that needs to be provided. The way that the annotation is defined here, the function will be called whenever the value of Game 1 or Game 2 is modified by the user, and the value returned by this function will be set as the value of the Prediction text box.

```
model_path = "models/logit_games_v1"
model   = mlflow.sklearn.load_model(model_path)

@app.callback(
    Output(component_id='pred', component_property='value'),
    [Input(component_id='g1', component_property='value'),
     Input(component_id='g2', component_property='value')]
)
def update_prediction(game1, game2):

    new_row = { "G1": float(game1),
                "G2": float(game2),
                "G3": 0, "G4": 0,
                "G5": 0, "G6": 0,
                "G7": 0, "G8": 0,
                "G9": 0, "G10":0 }

    new_x = pd.DataFrame.from_dict(new_row,
                                   orient = "index").transpose()
    return str(model.predict_proba(new_x)[0][1])
```

The function takes the two values provided by the user, and creates a Pandas dataframe. As before, we transpose the dataframe to provide a single row that we'll pass as input to the loaded model. The value predicted by the model is then returned and set as the value of the Prediction text box.

The updated application with the callback function included is shown in Figure 2.3. The prediction value now dynamically changes in response to changes in the other text fields, and provides a way of introspecting the model.

Dash is great for building web applications, because it eliminates the need to write JavaScript code. It's also possible to stylize Dash application using CSS to add some polish to your tools.


Model UI

Game 1 1

Game 2 0

Prediction 0.13479385252140805

FIGURE 2.3: The resulting model prediction.

2.6 Conclusion

The Python ecosystem has a great suite of tools for building web applications. Using only Python, you can write scalable APIs deployed to the open web or custom UI applications that interact with backend Python code. This chapter focused on Flask, which can be extended with other libraries and hosted in a wide range of environments. One of the important concepts we touched on in this chapter is model persistence, which will be useful in other contexts when building scalable model pipelines. We also deployed a simple application to Heroku, which is a separate cloud platform from AWS and GCP.

This chapter is only an introduction to the many different web tools within the Python ecosystem, and the topic of scaling these types of tools is outside the scope of this book. Instead, we'll focus on managed solutions for models on the web, which significantly reduces the DevOps overhead of deploying models as web services. The next chapter will cover two systems for serverless functions in managed environments.

3

Models as Serverless Functions

Serverless technologies enable developers to write and deploy code without needing to worry about provisioning and maintaining servers. One of the most common uses of this technology is serverless functions, which makes it much easier to author code that can scale to match variable workloads. With serverless function environments, you write a function that the runtime supports, specify a list of dependencies, and then deploy the function to production. The cloud platform is responsible for provisioning servers, scaling up more machines to match demand, managing load balancers, and handling versioning. Since we've already explored hosting models as web endpoints, serverless functions are an excellent tool to utilize when you want to rapidly move from prototype to production for your predictive models.

Serverless functions were first introduced on AWS in 2015 and GCP in 2016. Both of these systems provide a variety of triggers that can invoke functions, and a number of outputs that the functions can trigger in response. While it's possible to use serverless functions to avoid writing complex code for glueing different components together in a cloud platform, we'll explore a much narrower use case in this chapter. We'll write serverless functions that are triggered by an HTTP request, calculate a propensity score for the passed in feature vector, and return the prediction as JSON. For this specific use case, GCP's Cloud Functions are much easier to get up and running, but we'll explore both AWS and GCP solutions.

In this chapter, we'll introduce the concept of managed services, where the cloud platform is responsible for provisioning servers. Next, we'll cover hosting sklearn and Keras models with Cloud

Functions. To conclude, we'll show how to achieve the same result for sklearn models with Lambda functions in AWS. We'll also touch on model updates and access control.

3.1 Managed Services

Since 2015, there's been a movement in cloud computing to transition developers away from manually provisioning servers to using managed services that abstract away the concept of servers. The main benefit of this new paradigm is that developers can write code in a staging environment and then push code to production with minimal concerns about operational overhead, and the infrastructure required to match the required workload can be automatically scaled as needed. This enables both engineers and data scientists to be more active in DevOps, because much of the operational concerns of the infrastructure are managed by the cloud provider.

Manually provisioning servers, where you `ssh` into the machines to set up libraries and code, is often referred to as *hosted* deployments, versus *managed* solutions where the cloud platform is responsible for abstracting away this concern from the user. In this book, we'll cover examples in both of these categories. Here are some of the different use cases we'll cover:

- **Web Endpoints:** Single EC2 instance (hosted) vs AWS Lambda (managed).
- **Docker:** Single EC2 instance (hosted) vs ECS (managed).
- **Messaging:** Kafka (hosted) vs PubSub (managed).

This chapter will walk through the first use case, migrating web endpoints from a single machine to an elastic environment. We'll also work through examples that thread this distinction, such as deploying Spark environments with specific machine configurations and manual cluster management.

Serverless technologies and managed services are a powerful tool for data scientists, because they enable a single developer to build data pipelines that can scale to massive workloads. It's a powerful tool for data scientists to wield, but there are a few trade-offs to consider when using managed services. Here are some of the main issues to consider when deciding between hosted and managed solutions:

- **Iteration:** Are you rapidly prototyping on a product or iterating on a system in production?
- **Latency:** Is a multi-second latency acceptable for your SLAs?
- **Scale:** Can your system scale to match peak workload demands?
- **Cost:** Are you willing to pay more for serverless cloud costs?

At a startup, serverless technologies are great because you have low-volume traffic and have the ability to quickly iterate and try out new architectures. At a certain scale, the dynamics change and the cost of using serverless technologies may be less appealing when you already have in-house expertise for provisioning cloud services. In my past projects, the top issue that was a concern was latency, because it can impact customer experiences. In chapter 8, we'll touch on this topic, because managed solutions often do not scale well to large streaming workloads.

Even if your organization does not use managed services in daily operations, it's a useful skill set to get hands on with as a data scientist, because it means that you can separate model training from model deployment issues. One of the themes in this book is that models do not need to be complex, but it can be complex to deploy models. Serverless functions are a great approach for demonstrating the ability to serve models at scale, and we'll walk through two cloud platforms that provide this capability.

3.2 Cloud Functions (GCP)

Google Cloud Platform provides an environment for serverless functions called Cloud Functions. The general concept with this tool is that you can write code targeted for Flask, but leverage the managed services in GCP to provide elastic computing for your

Python code. GCP is a great environment to get started with serverless functions, because it closely matches standard Python development ecosystems, where you specify a requirements file and application code.

We'll build scalable endpoints that serve both sklearn and Keras models with Cloud Functions. There are a few issues to be aware of when writing functions in this environment:

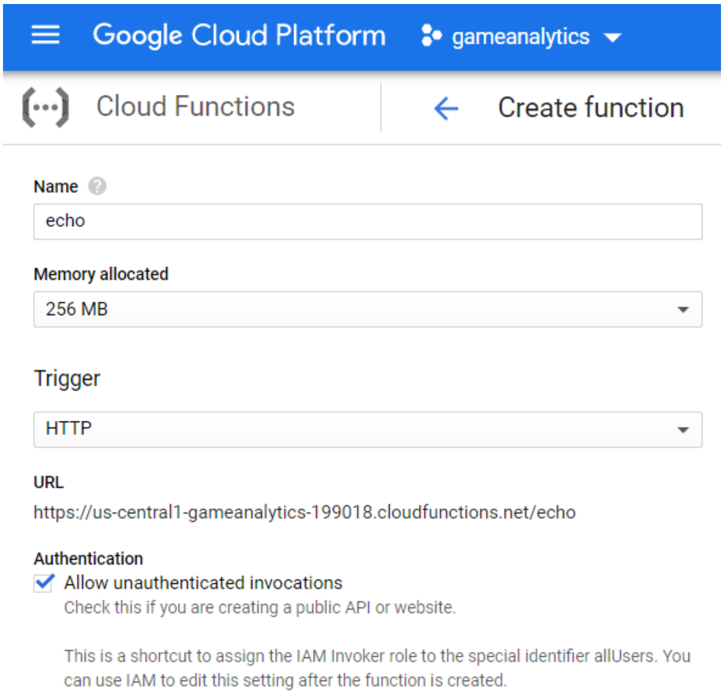
- **Storage:** Cloud Functions run in a read-only environment, but you can write to the `/tmp` directory.
- **Tabs:** Spaces versus tabs can cause issues in Cloud Functions, and if you are working in the web editor versus familiar tools like Sublime Text, these can be difficult to spot.
- **sklearn:** When using a requirements file, it's important to differentiate between `sklearn` and `scikit-learn` based on your imports. We'll use sklearn in this chapter.

Cloud platforms are always changing, so the specific steps outlined in this chapter may change based on the evolution of these platforms, but the general approach for deploying functions should apply throughout these updates. As always, the approach I advocate for is starting with a simple example, and then scaling to more complex solutions as needed. In this section, we'll first build an echo service and then explore sklearn and Keras models.

3.2.1 Echo Service

GCP provides a web interface for authoring Cloud Functions. This UI provides options for setting up the triggers for a function, specifying the requirements file for a Python function, and authoring the implementation of the Flask function that serves the request. To start, we'll set up a simple echo service that reads in a parameter from an HTTP request and returns the passed in parameter as the result.

In GCP, you can directly set up a Cloud Function as an HTTP endpoint without needing to configure additional triggers. To get started with setting up an echo service, perform the following actions in the GCP console:



Name
echo

Memory allocated
256 MB

Trigger
HTTP

URL
https://us-central1-gameanalytics-199018.cloudfunctions.net/echo

Authentication
✓ Allow unauthenticated invocations
Check this if you are creating a public API or website.

This is a shortcut to assign the IAM Invoker role to the special identifier allUsers. You can use IAM to edit this setting after the function is created.

FIGURE 3.1: Creating a Cloud Function.

1. Search for "Cloud Function"
2. Click on "Create Function"
3. Select "HTTP" as the trigger
4. Select "Allow unauthenticated invocations"
5. Select "Inline Editor" for source code
6. Select Python 3.7 as the runtime

An example of this process is shown in Figure 3.1. After performing these steps, the UI will provide tabs for the main.py and requirements.txt files. The requirements file is where we will specify libraries, such as `flask >= 1.1.1`, and the main file is where we'll implement our function behavior.

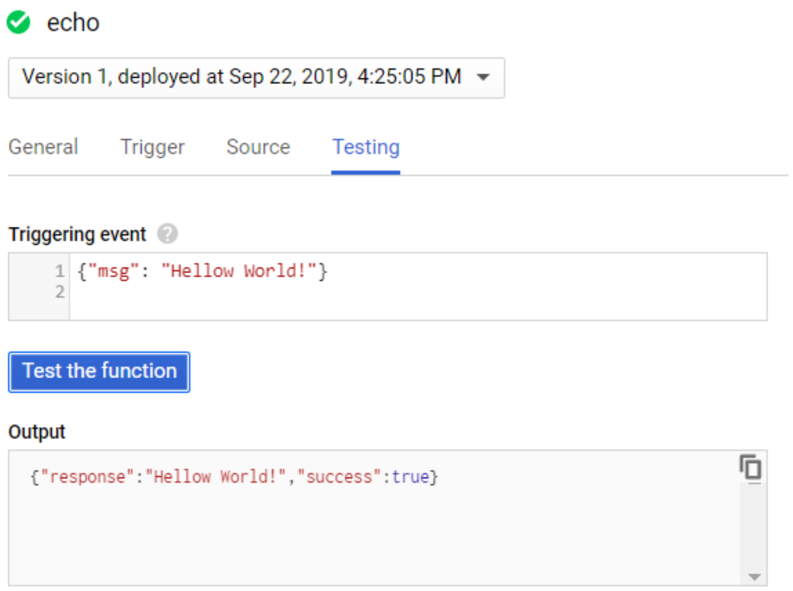
We'll start by creating a simple echo service that parses out the `msg` parameter from the passed in request and returns this parameter as a JSON response. In order to use the `jsonify` function we need to include the `flask` library in the requirements file. The `requirements.txt` file and `main.py` files for the simple echo service are shown in the snippet below. The echo function here is similar to the echo service we coded in Section 2.1.1, the main distinction here is that we are no longer using annotations to specify the endpoints and allowed methods. Instead, these settings are now being specified using the Cloud Functions UI.

```
# requirements.txt
flask

#main.py
def echo(request):
    from flask import jsonify

    data = {"success": False}
    params = request.get_json()

    if "msg" in params:
        data["response"] = str(params['msg'])
        data["success"] = True

    return jsonify(data)
```

We can deploy the function to production by performing the following steps:

1. Update “Function to execute” to “echo”
2. Click “Create” to deploy

Once the function has been deployed, you can click on the “Testing” tab to check if the deployment of the function worked as intended. You can specify a JSON object to pass to the function, and invoke the function by clicking “Test the function”, as shown in Figure 3.2. The result of running this test case is the JSON object returned in the `Output` dialog, which shows that invoking the echo function worked correctly.



```
{"msg": "Hellow World!"}
```

```
{"response":"Hellow World!","success":true}
```

FIGURE 3.2: Testing a Cloud Function.

Now that the function is deployed and we enabled unauthenticated access to the function, we can call the function over the web using Python. To get the URL of the function, click on the "trigger" tab. We can use the `requests` library to pass a JSON object to the serverless function, as shown in the snippet below.

```
import requests

result = requests.post(
        "https://us-central1-gameanalytics.cloudfunctions.net/echo"
                    ,json = { 'msg': 'Hello from Cloud Function' })
print(result.json())
```

The result of running this script is that a JSON payload is returned from the serverless function. The output from the call is the JSON shown below.

```
{
    'response': 'Hello from Cloud Function',
    'success': True
}
```

We now have a serverless function that provides an echo service. In order to serve a model using Cloud Functions, we'll need to persist the model specification somewhere that the serverless function can access. To accomplish this, we'll use Cloud Storage to store the model in a distributed storage layer.

3.2.2 Cloud Storage (GCS)

GCP provides an elastic storage layer called Google Cloud Storage (GCS) that can be used for distributed file storage and can also scale to other uses such as data lakes. In this section, we'll explore the first use case of utilizing this service to store and retrieve files for use in a serverless function. GCS is similar to AWS's offering called S3, which is leveraged extensively in the gaming industry to build data platforms.

While GCP does provide a UI for interacting with GCS, we'll explore the command line interface in this section, since this approach is useful for building automated workflows. GCP requires authentication for interacting with this service, please revisit section 1.5.1 if you have not yet set up a JSON credentials file. In order to interact with Cloud Storage using Python, we'll also need to install the GCS library, using the command shown below:

```
pip install --user google-cloud-storage
export GOOGLE_APPLICATION_CREDENTIALS=/home/ec2-user/dsdemo.json
```

Now that we have the prerequisite libraries installed and credentials set up, we can interact with GCS programmatically using Python. Before we can store a file, we need to set up a bucket on GCS. A bucket is a prefix assigned to all files stored on GCS, and each bucket name must be globally unique. We'll create a bucket name called `dsp_model_store` where we'll store model objects. The script below shows how to create a new bucket using the `create_bucket` function and then iterate through all of the available buckets using the `list_buckets` function. You'll need to change the `bucket_name` variable to something unique before running this script.

```
from google.cloud import storage
bucket_name = "dsp_model_store"

storage_client = storage.Client()
storage_client.create_bucket(bucket_name)

for bucket in storage_client.list_buckets():
    print(bucket.name)
```

After running this code, the output of the script should be a single bucket, with the name assigned to the `bucket_name` variable. We now have a path on GCS that we can use for saving files: `gs://dsp_model_storage`.

We'll reuse the model we trained in Section 2.2.1 to deploy a logistic regression model with Cloud Functions. To save the file to GCS, we need to assign a path to the destination, shown by the `bucket.blob` command below and select a local file to upload, which is passed to the upload function.

```
from google.cloud import storage

bucket_name = "dsp_model_store"
storage_client = storage.Client()
bucket = storage_client.get_bucket(bucket_name)

blob = bucket.blob("serverless/logit/v1")
blob.upload_from_filename("logit.pkl")
```

After running this script, the local file `logit.pkl` will now be available on GCS at the following location:

```
gs://dsp_model_storage/serverless/logit/v1/logit.pkl
```

While it's possible to use URIs such as this directly to access files, as we'll explore with Spark in Chapter 6, in this section we'll retrieve the file using the bucket name and blob path. The code snippet below shows how to download the model file from GCS to local storage. We download the model file to the local path of `local_logit.pkl` and then load the model by calling `pickle.load` with this path.

```
import pickle
from google.cloud import storage

bucket_name = "dsp_model_store"
storage_client = storage.Client()
bucket = storage_client.get_bucket(bucket_name)

blob = bucket.blob("serverless/logit/v1")
blob.download_to_filename("local_logit.pkl")
model = pickle.load(open("local_logit.pkl", 'rb'))
model
```

We can now programmatically store model files to GCS using Python and also retrieve them, enabling us to load model files in Cloud Functions. We'll combine this with the Flask examples from the previous chapter to serve sklearn and Keras models as Cloud Functions.

3.2.3 Model Function

We can now set up a Cloud Function that serves logistic regression model predictions over the web. We'll build on the Flask example that we explored in Section 2.3.1 and make a few modifications for the service to run on GCP. The first step is to specify the required Python libraries that we'll need to serve requests in the `requirements.txt` file, as shown below. We'll also need Pandas to set up a dataframe for making the prediction, sklearn for applying the model, and cloud storage for retrieving the model object from GCS.

```
google-cloud-storage
sklearn
pandas
flask
```

The next step is to implement our model function in the `main.py` file. A small change from before is that the `params` object is now fetched using `request.get_json()` rather than `flask.request.args`. The main change is that we are now downloading the model file from GCS rather than retrieving the file directly from local storage, because local files are not available when writing Cloud Functions with the UI tool. An additional change from the prior function is that we are now reloading the model for every request, rather than loading the model file once at startup. In a later code snippet, we'll show how to use global objects to cache the loaded model.

```
def pred(request):
    from google.cloud import storage
    import pickle as pk
    import sklearn
    import pandas as pd
    from flask import jsonify

    data = {"success": False}
    params = request.get_json()

    if "G1" in params:

        new_row = { "G1": params.get("G1"),"G2": params.get("G2"),
                    "G3": params.get("G3"),"G4": params.get("G4"),
                    "G5": params.get("G5"),"G6": params.get("G6"),
                    "G7": params.get("G7"),"G8": params.get("G8"),
                    "G9": params.get("G9"),"G10":params.get("G10")}
        new_x = pd.DataFrame.from_dict(new_row,
                              orient = "index").transpose()

        # set up access to the GCS bucket
        bucket_name = "dsp_model_store"
        storage_client = storage.Client()
        bucket = storage_client.get_bucket(bucket_name)

        # download and load the model
        blob = bucket.blob("serverless/logit/v1")
        blob.download_to_filename("/tmp/local_logit.pkl")
        model = pk.load(open("/tmp/local_logit.pkl", 'rb'))

        data["response"] = str(model.predict_proba(new_x)[0][1])
        data["success"] = True

    return jsonify(data)
```

One note in the code snippet above is that the `/tmp` directory is used to store the downloaded model file. In Cloud Functions, you are unable to write to the local disk, with the exception of this directory. Generally it's best to read objects directly into memory rather than pulling objects to local storage, but the Python library for reading objects from GCS currently requires this approach.

For this function, we created a new Cloud Function named `pred`, set the function to execute to `pred`, and deployed the function to production. We can now call the function from Python, using the same approach from 2.3.1 with a URL that now points to the Cloud Function, as shown below:

```
import requests

result = requests.post(
        "https://us-central1-gameanalytics.cloudfunctions.net/pred"
         ,json = { 'G1':'1', 'G2':'0', 'G3':'0', 'G4':'0', 'G5':'0'
               ,'G6':'0', 'G7':'0', 'G8':'0', 'G9':'0', 'G10':'0'})
print(result.json())
```

The result of the Python web request to the function is a JSON response with a response value and model prediction, shown below:

```
{
  'response': '0.06745113592634559',
  'success': True
}
```

In order to improve the performance of the function, so that it takes milliseconds to respond rather than seconds, we'll need to cache the model object between runs. It's best to avoid defining variables outside of the scope of the function, because the server hosting the function may be terminated due to inactivity. Global variables are an execution to this rule, when used for caching objects between function invocations. This code snippet below shows how a global model object can be defined within the scope of the pred function to provide a persistent object across calls. During the first function invocation, the model file will be retrieved from GCS and loaded via pickle. During following runs, the model object will already be loaded into memory, providing a much faster response time.

```
model = None

def pred(request):
    global model

    if not model:

        # download model from GCS
        model = pk.load(open("/tmp/local_logit.pkl", 'rb'))
    # apply model

    return jsonify(data)
```

Caching objects is important for authoring responsive models that lazily load objects as needed. It's also useful for more complex models, such as Keras which requires persisting a TensorFlow graph between invocations.

3.2.4 Keras Model

Since Cloud Functions provide a requirements file that can be used to add additional dependencies to a function, it's also possible to serve Keras models with this approach. We'll be able to reuse most of the code from the past section, and we'll also use the Keras and Flask approach introduced in Section 2.3.2. Given the size of the Keras libraries and dependencies, we'll need to upgrade the memory available for the Function from 256 MB to 1GB. We also need to update the requirements file to include Keras:

```
google-cloud-storage
tensorflow
keras
pandas
flask
```

The full implementation for the Keras model as a Cloud Function is shown in the code snippet below. In order to make sure that the TensorFlow graph used to load the model is available for future invocations of the model, we use global variables to cache both the model and graph objects. To load the Keras model, we need to redefine the `auc` function that was used during model training, which we include within the scope of the `predict` function. We reuse the same approach from the prior section to download the model file from GCS, but now use `load_model` from Keras to read the model file into memory from the temporary disk location. The result is a Keras predictive model that lazily fetches the model file and can scale to meet variable workloads as a serverless function.

```
model = None
graph = None

def predict(request):
    global model
    global graph

    from google.cloud import storage
    import pandas as pd
    import flask
    import tensorflow as tf
    import keras as k
    from keras.models import load_model
    from flask import jsonify

    def auc(y_true, y_pred):
        auc = tf.metrics.auc(y_true, y_pred)[1]
        k.backend.get_session().run(
                      tf.local_variables_initializer())
        return auc

    data = {"success": False}
    params = request.get_json()

    # download model if not cached
    if not model:
        graph = tf.get_default_graph()

        bucket_name = "dsp_model_store_1"
        storage_client = storage.Client()
        bucket = storage_client.get_bucket(bucket_name)

        blob = bucket.blob("serverless/keras/v1")
        blob.download_to_filename("/tmp/games.h5")
        model = load_model('/tmp/games.h5',
                          custom_objects={'auc':auc})

    # apply the model
    if "G1" in params:
        new_row = { "G1": params.get("G1"),"G2": params.get("G2"),
                    "G3": params.get("G3"),"G4": params.get("G4"),
                    "G5": params.get("G5"),"G6": params.get("G6"),
                    "G7": params.get("G7"),"G8": params.get("G8"),
                    "G9": params.get("G9"),"G10":params.get("G10")}

        new_x = pd.DataFrame.from_dict(new_row,
                                       orient = "index").transpose()

        with graph.as_default():
            data["response"]= str(model.predict_proba(new_x)[0][0])
            data["success"] = True

    return jsonify(data)
```

To test the deployed model, we can reuse the Python web request script from the prior section and replace `pred` with `predict` in the request URL. We have now deployed a deep learning model to production.

3.2.5 Access Control

The Cloud Functions we introduced in this chapter are open to the web, which means that anyone can access them and potentially abuse the endpoints. In general, it's best not to enable unauthenticated access and instead lock down the function so that only authenticated users and services can access them. This recommendation also applies to the Flask apps that we deployed in the last chapter, where it's a best practice to restrict access to services that can reach the endpoint using AWS private IPs.

There are a few different approaches for locking down Cloud Functions to ensure that only authenticated users have access to the functions. The easiest approach is to disable "Allow unauthenticated invocations" in the function setup to prevent hosting the function on the open web. To use the function, you'll need to set up IAM roles and credentials for the function. This process involves a number of steps and may change over time as GCP evolves. Instead of walking through this process, it's best to refer to the GCP documentation[1].

Another approach for setting up functions that enforce authentication is by using other services within GCP. We'll explore this approach in Chapter 8, which introduces GCP's PubSub system for producing and consuming messages within GCP's ecosystem.

3.2.6 Model Refreshes

We've deployed sklearn and Keras models to production using Cloud Functions, but the current implementations of these functions use static model files that will not change over time. It's usually necessary to make changes to models over time to ensure that the accuracy of the models do not drift too far from expected performance. There's a few different approaches that we can take to update the model specification that a Cloud Function is using:

1. **Redeploy:** Overwriting the model file on GCS and redeploying the function will result in the function loading the updated file.
2. **Timeout:** We can add a timeout to the function, where the model is re-downloaded after a certain threshold of time passes, such as 30 minutes.
3. **New Function:** We can deploy a new function, such as `pred_v2` and update the URL used by systems calling the service, or use a load balancer to automate this process.
4. **Model Trigger:** We can add additional triggers to the function to force the function to manually reload the model.

[1] https://cloud.google.com/functions/docs/securing/authenticating

While the first approach is the easiest to implement and can work well for small-scale deployments, the third approach, where a load balancer is used to direct calls to the newest function available is probably the most robust approach for production systems. A best practice is to add logging to your function, in order to track predictions over time so that you can log the performance of the model and identify potential drift.

3.3 Lambda Functions (AWS)

AWS also provides an ecosystem for serverless functions called Lambda. AWS Lambda is useful for glueing different components within an AWS deployment together, since it supports a rich set of triggers for function inputs and outputs. While Lambda does provide a powerful tool for building data pipelines, the current Python development environment is a bit clunkier than GCP.

In this section we'll walk through setting up an echo service and an sklearn model endpoint with Lambda. We won't cover Keras, because the size of the library causes problems when deploying a function with AWS. Unlike the past section where we used a UI to define functions, we'll use command line tools for providing our function definition to Lambda.

3.3.1 Echo Function

For a simple function, you can use the inline code editor that Lambda provides for authoring functions. You can create a new function by performing the following steps in the AWS console:

1. Under "Find Services", select "Lambda"
2. Select "Create Function"
3. Use "Author from scratch"
4. Assign a name (e.g. echo)
5. Select a Python runtime
6. Click "Create Function"

After running these steps, Lambda will generate a file called `lambda_function.py`. The file defines a function called `lambda_handler` which we'll use to implement the echo service. We'll make a small modification to the file, as shown below, which echoes the `msg` parameter as the body of the response object.

```
def lambda_handler(event, context):

    return {
        'statusCode': 200,
        'body': event['msg']
    }
```

Click "Save" to deploy the function and then "Test" to test the file. If you use the default test parameters, then an error will be returned when running the function, because no `msg` key is available in the event object. Click on "Configure test event", and define use the following configuration:

```
{
  "msg": "Hello from Lambda!"
}
```

After clicking on "Test", you should see the execution results. The response should be the echoed message with a status code of 200 returned. There's also details about how long the function took to execute (25.8ms), the billing duration (100ms), and the maximum memory used (56 MB).

We have now a simple function running on AWS Lambda. For this function to be exposed to external systems, we'll need to set up an API Gateway, which is covered in Section 3.3.3. This function will scale up to meet demand if needed, and requires no server monitoring once deployed. To setup a function that deploys a model, we'll need to use a different workflow for authoring and publishing the function, because AWS Lambda does not currently support a `requirements.txt` file for defining dependencies when writing functions with the inline code editor. To store the model file that we want to serve with a Lambda function, we'll use S3 as a storage layer for model artifacts.

3.3.2 Simple Storage Service (S3)

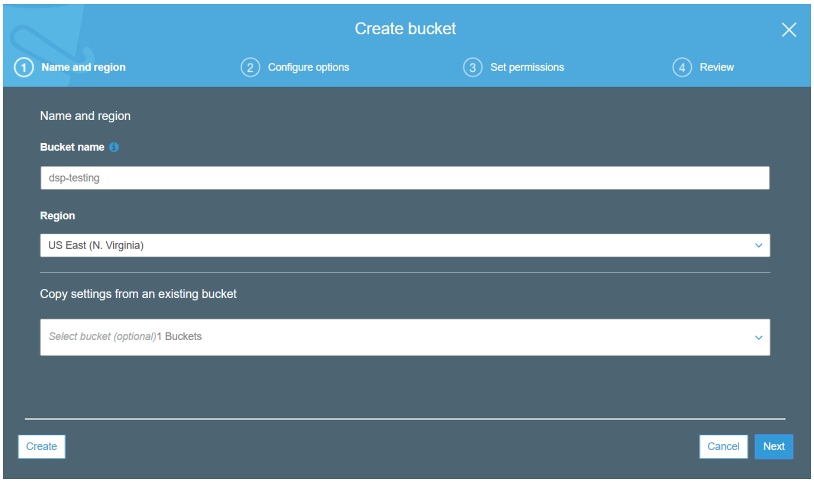
AWS provides a highly-performant storage layer called S3, which can be used to host individual files for web sites, store large files for data processing, and even host thousands or millions of files for building data lakes. For now, our use case will be storing an individual `zip` file, which we'll use to deploy new Lambda functions. However, there are many broader use cases and many companies use S3 as their initial endpoint for data ingestion in data platforms.

In order to use S3 to store our function to deploy, we'll need to set up a new S3 bucket, define a policy for accessing the bucket, and configure credentials for setting up command line access to S3. Buckets on S3 are analogous to GCS buckets in GCP.

To set up a bucket, browse to the AWS console and select "S3" under find services. Next, select "Create Bucket" to set up a location for storing files on S3. Create a unique name for the S3 bucket, as shown in Figure 3.3, and click "Next" and then "Create Bucket" to finalize setting up the bucket.

We now have a location to store objects on S3, but we still need to set up a user before we can use the command line tools to write and read from the bucket. Browse to the AWS console and select "IAM" under "Find Services". Next, click "Users" and then "Add user" to set up a new user. Create a user name, and select "Programmatic access" as shown in Figure 3.4.

The next step is to provide the user with full access to S3. Use the attach existing policies option and search for S3 policies in order to find and select the `AmazonS3FullAccess` policy, as shown in Figure 3.5. Click "Next" to continue the process until a new user is defined. At the end of this process, a set of credentials will be displayed, including an access key ID and secret access key. Store these values in a safe location.



Name and region

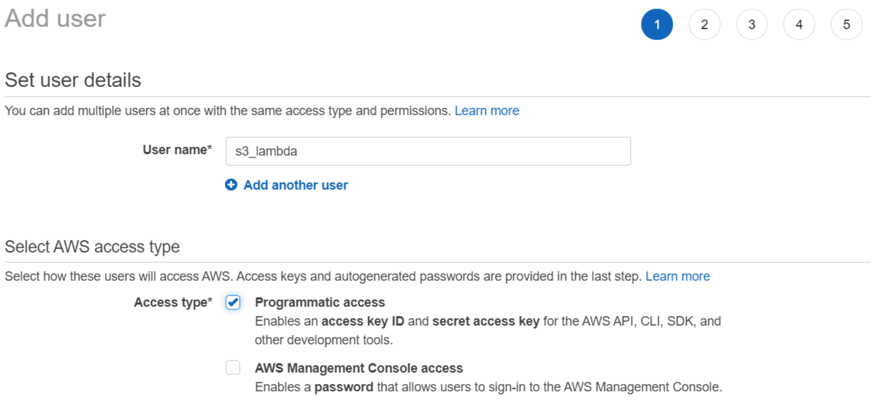
Bucket name: dsp-testing

Region: US East (N. Virginia)

Copy settings from an existing bucket

Select bucket (optional)1 Buckets

FIGURE 3.3: Creating an S3 bucket on AWS.



Set user details

You can add multiple users at once with the same access type and permissions. Learn more

User name*: s3_lambda

Select AWS access type

Select how these users will access AWS. Access keys and autogenerated passwords are provided in the last step. Learn more

Access type*

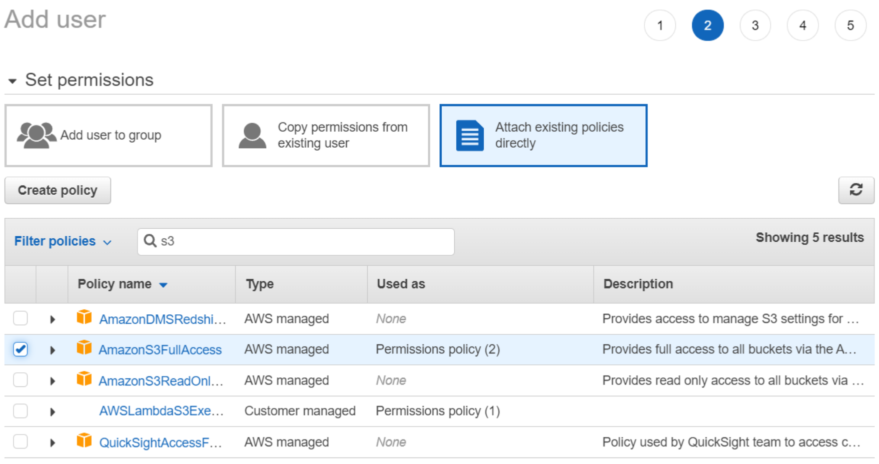
[✓] **Programmatic access**
Enables an **access key ID** and **secret access key** for the AWS API, CLI, SDK, and other development tools.

[ ] **AWS Management Console access**
Enables a **password** that allows users to sign-in to the AWS Management Console.

FIGURE 3.4: Setting up a user with S3 access.

The last step needed for setting up command line access to S3 is running the `aws configure` command from your EC2 instance. You'll be asked to provide the access and secret keys from the user we just set up. In order to test that the credentials are properly configured, you can run the following commands:

```
aws configure
aws s3 ls
```



		Policy name	Type	Used as	Description
☐	▸	AmazonDMSRedshi...	AWS managed	None	Provides access to manage S3 settings for ...
✓	▸	AmazonS3FullAccess	AWS managed	Permissions policy (2)	Provides full access to all buckets via the A...
☐	▸	AmazonS3ReadOnl...	AWS managed	None	Provides read only access to all buckets via ...
☐	▸	AWSLambdaS3Exe...	Customer managed	Permissions policy (1)	
☐	▸	QuickSightAccessF...	AWS managed	None	Policy used by QuickSight team to access c...

FIGURE 3.5: Selecting a policy for full S3 access.

The results should include the name of the S3 bucket we set up at the beginning of this section. Now that we have an S3 bucket set up with command line access, we can begin writing Lambda functions that use additional libraries such as Pandas and sklearn.

3.3.3 Model Function

In order to author a Lambda function that uses libraries outside of the base Python distribution, you'll need to set up a local environment that defines the function and includes all of the dependencies. Once your function is defined, you can upload the function by creating a zip file of the local environment, uploading the resulting file to S3, and configuring a Lambda function from the file uploaded to S3.

The first step in this process is to create a directory with all of the dependencies installed locally. While it's possible to perform this process on a local machine, I used an EC2 instance to provide a clean Python environment. The next step is to install the libraries needed for the function, which are Pandas and sklearn. These libraries are already installed on the EC2 instance, but need to be reinstalled in the current directory in order to be included in the zip file that we'll upload to S3. To accomplish this, we can append `-t .` to the end of the pip command in order to install the libraries into the current directory. The last steps to run on the command line are copying our logistic regression model into the current directory, and creating a new file that will implement the Lambda function.

```
mkdir lambda
cd lambda
pip install pandas -t .
pip install sklearn -t .
cp ../logit.pkl logit.pkl
vi logit.py
```

The full source code for the Lambda function that serves our logistic regression model is shown in the code snippet below. The structure of the file should look familiar, we first globally define a model object and then implement a function that services model requests. This function first parses the response to extract the inputs to the model, and then calls `predict_proba` on the resulting dataframe to get a model prediction. The result is then returned as a dictionary object containing a `body` key. It's important to define the function response within the `body` key, otherwise Lambda will throw an exception when invoking the function over the web.

```
from sklearn.externals import joblib
import pandas as pd
import json
model = joblib.load('logit.pkl')

def lambda_handler(event, context):

    # read in the request body as the event dict
    if "body" in event:
        event = event["body"]
        if event is not None:
            event = json.loads(event)
        else:
            event = {}

    if "G1" in event:
        new_row = { "G1": event["G1"],"G2": event["G2"],
                    "G3": event["G3"],"G4": event["G4"],
                    "G5": event["G5"],"G6": event["G6"],
                    "G7": event["G7"],"G8": event["G8"],
                    "G9": event["G9"],"G10":event["G10"]}

        new_x = pd.DataFrame.from_dict(new_row,
                             orient = "index").transpose()
        prediction = str(model.predict_proba(new_x)[0][1])

        return { "body": "Prediction " + prediction }

    return { "body": "No parameters" }
```

Unlike Cloud Functions, Lambda functions authored in Python are not built on top of the Flask library. Instead of requiring a single parameter (`request`), a Lambda function requires `event` and `context` objects to be passed in as function parameters. The event includes the parameters of the request, and the context provides information about the execution environment of the function. When testing a Lambda function using the "Test" functionality in the Lambda console, the test configuration is passed directly to the function as a dictionary in the `event` object. However, when the function is called from the web, the event object is a dictionary that describes the web request, and the request parameters are stored in the `body` key in this dict. The first step in the Lambda function above checks if the function is being called directly from the console, or via the web. If the function is being called from the web, then the function overrides the event dictionary with the content in the body of the request.

One of the main differences from this approach with the GCP Cloud Function is that we did not need to explicitly define global variables that are lazily defined. With Lambda functions, you can define variables outside the scope of the function that are persisted before the function is invoked. It's important to load model objects outside of the model service function, because reloading the model each time a request is made can become expensive when handling large workloads.

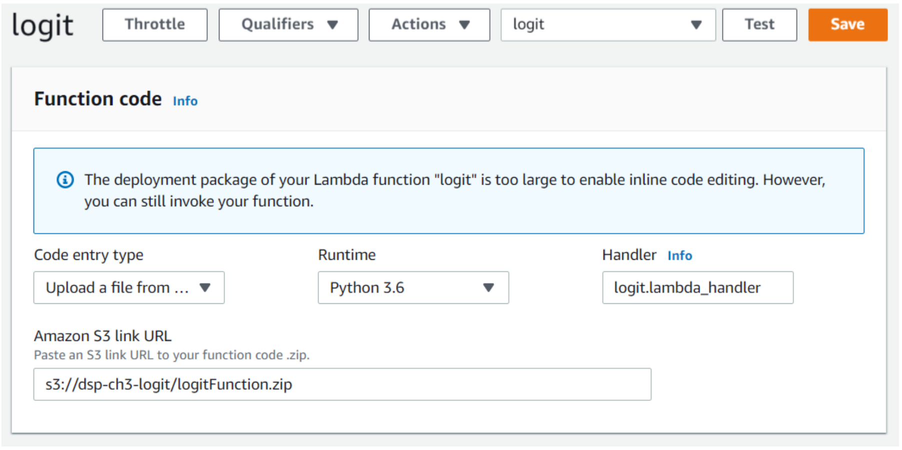
To deploy the model, we need to create a zip file of the current directory, and upload the file to a location on S3. The snippet below shows how to perform these steps and then confirm that the upload succeeded using the `s3 ls` command. You'll need to modify the paths to use the S3 bucket name that you defined in the previous section.

```
zip -r logitFunction.zip .
aws s3 cp logitFunction.zip s3://dsp-ch3-logit/logitFunction.zip
aws s3 ls s3://dsp-ch3-logit/
```

Once your function is uploaded as a zip file to S3, you can return to the AWS console and set up a new Lambda function. Select "Author from scratch" as before, and under "Code entry type" select the option to upload from S3, specifying the location from the `cp` command above. You'll also need to define the `Handler`, which is a combination of the Python file name and the Lambda function name. An example configuration for the logit function is shown in Figure 3.6.

Make sure to select the Python runtime as the same version of Python that was used to run the pip commands on the EC2 instance. Once the function is deployed by pressing "Save", we can test the function using the following definition for the test event.

```
{
  "G1": "1", "G2": "1", "G3": "1",
  "G4": "1", "G5": "1",
```



Function code Info

The deployment package of your Lambda function "logit" is too large to enable inline code editing. However, you can still invoke your function.

Code entry type: Upload a file from ...
Runtime: Python 3.6
Handler Info: logit.lambda_handler

Amazon S3 link URL
Paste an S3 link URL to your function code .zip.
s3://dsp-ch3-logit/logitFunction.zip

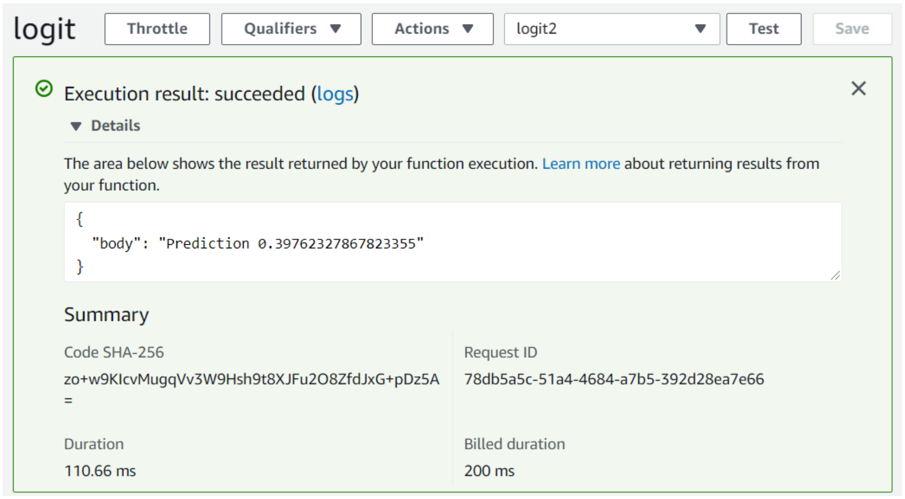
FIGURE 3.6: Defining the logit function on AWS Lambda.

```
  "G6": "1", "G7": "1", "G8": "1",
  "G9": "1", "G10": "1"
}
```

Since the model is loaded when the function is deployed, the response time for testing the function should be relatively fast. An example output of testing the function is shown in Figure 3.7. The output of the function is a dictionary that includes a body key and the output of the model as the value. The function took 110 ms to execute and was billed for a duration of 200 ms.

So far, we've invoked the function only using the built-in test functionality of Lambda. In order to host the function so that other services can interact with the function, we'll need to define an API Gateway. Under the "Designer" tab, click "Add Trigger" and select "API Gateway". Next, select "Create a new API" and choose "Open" as the security setting. After setting up the trigger, an API Gateway should be visible in the Designer layout, as shown in Figure 3.8.

Before calling the function from Python code, we can use the API Gateway testing functionality to make sure that the function is set up properly. One of the challenges I ran into when testing this



Execution result: succeeded (logs)

▼ Details

The area below shows the result returned by your function execution. Learn more about returning results from your function.

```
{
  "body": "Prediction 0.39762327867823355"
}
```

Summary

Code SHA-256	Request ID
zo+w9KIcvMugqVv3W9Hsh9t8XJFu2O8ZfdJxG+pDz5A=	78db5a5c-51a4-4684-a7b5-392d28ea7e66
Duration	Billed duration
110.66 ms	200 ms

FIGURE 3.7: Testing the logit function on AWS Lambda.

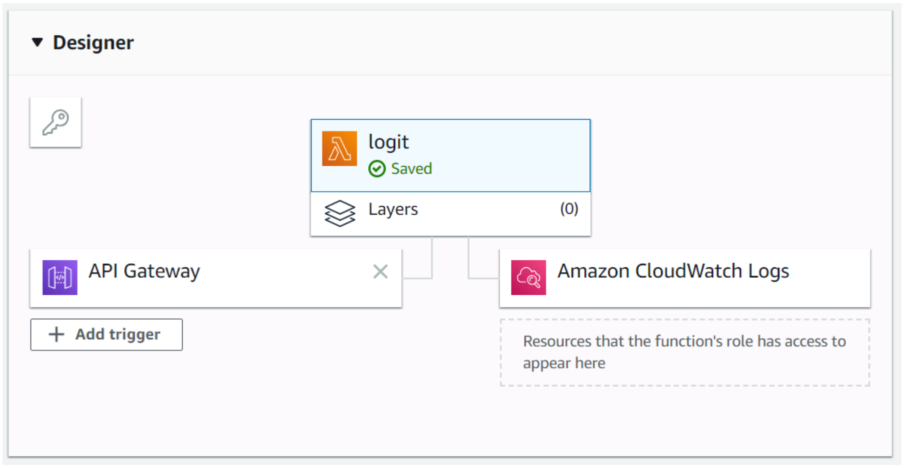
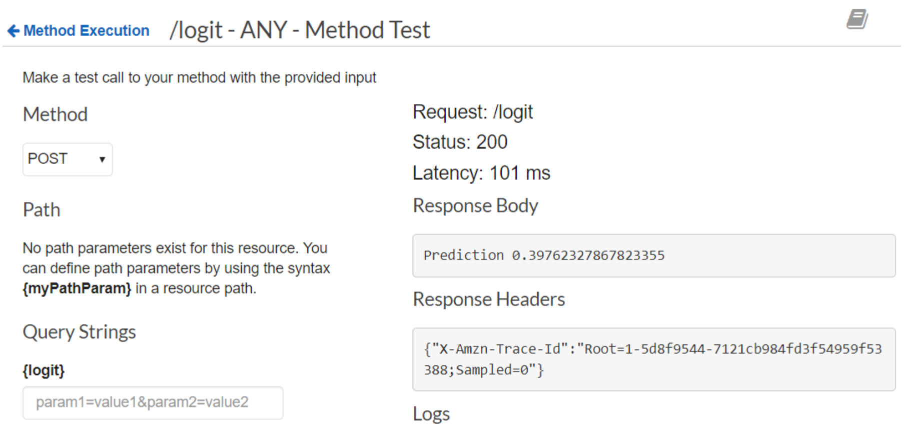


FIGURE 3.8: Setting up an API Gateway for the function.



Make a test call to your method with the provided input

Method

POST

Path

No path parameters exist for this resource. You can define path parameters by using the syntax **{myPathParam}** in a resource path.

Query Strings

{logit}

param1=value1&param2=value2

Request: /logit

Status: 200

Latency: 101 ms

Response Body

```
Prediction 0.39762327867823355
```

Response Headers

```
{"X-Amzn-Trace-Id":"Root=1-5d8f9544-7121cb984fd3f54959f53
388;Sampled=0"}
```

Logs

FIGURE 3.9: Testing post commands on the Lambda function.

Lambda function was that the structure of the request varies when the function is invoked from the web versus the console. This is why the function first checks if the `event` object is a web request or dictionary with parameters. When you use the API Gateway to test the function, the resulting call will emulate calling the function as a web request. An example test of the logit function is shown in Figure 3.9.

Now that the gateway is set up, we can call the function from a remote host using Python. The code snippet below shows how to use a POST command to call the function and display the result. Since the function returns a string for the response, we use the `text` attribute rather than the json function to display the result.

```
import requests

result = requests.post("https://3z5btf0ucb.execute-api.us-east-1.
                                  amazonaws.com/default/logit",
     json = { 'G1':'1', 'G2':'0', 'G3':'0', 'G4':'0', 'G5':'0',
              'G6':'0', 'G7':'0', 'G8':'0', 'G9':'0', 'G10':'0' })

print(result.text)
```

We now have a predictive model deployed to AWS Lambda that will autoscale as necessary to match workloads, and which requires minimal overhead to maintain.

Similar to Cloud Functions, there are a few different approaches that can be used to update the deployed models. However, for the approach we used in this section, updating the model requires updating the model file in the development environment, rebuilding the zip file and uploading it to S3, and then deploying a new version of the model. This is a manual process and if you expect frequent model updates, then it's better to rewrite the function so that it fetches the model definition from S3 directly rather than expecting the file to already be available in the local context. The most scalable approach is setting up additional triggers for the function, to notify the function that it's time to load a new model.

3.4 Conclusion

Serverless functions are a type of managed service that enable developers to deploy production-scale systems without needing to worry about infrastructure. To provide this abstraction, different cloud platforms do place constraints on how functions must be implemented, but the trade-off is generally worth the improvement in DevOps that these tools enable. While serverless technologies like Cloud Functions and Lambda can be operationally expensive, they provide flexibility that can offset these costs.

In this chapter, we implemented echo services and sklearn model endpoints using both GCP's Cloud Functions and AWS's Lambda offerings. With AWS, we created a local Python environment with all dependencies and then uploaded the resulting files to S3 to deploy functions, while in GCP we authored functions directly using the online code editor. The best system to use will likely depend on which cloud provider your organization is already using, but when prototyping new systems, it's useful to have hands on experience using more than one serverless function ecosystem.