CHAPTER 2
MLOps Foundations

By Noah Gift

> Medical school was a woeful experience, an endless litany of fact, whose origins were rarely explained and whose usefulness was infrequently justified. My distaste for rote learning and my questioning attitude were not shared by most of the class of 96 students. This was particularly evident on one occasion when a biochemistry lecturer claimed to be deriving the Nernst equation. The class was faithfully copying what he wrote on the board. Having only a year before taken the Pchem course for chemistry majors at UCLA, I thought he was bluffing.
>
> "Where did you get that value for k?" I asked.
>
> The class shouted me down: "Let him finish! Just copy it."
>
> —Dr. Joseph Bogen

Having a solid foundation to build on is critical to any technical endeavor. In this chapter, several key building blocks set the foundation for the rest of the book. When dealing with students new to data science and machine learning, I have commonly encountered misconceptions about the items covered in this chapter. This chapter aims to build a strong foundation for using MLOps methodologies.

Bash and the Linux Command Line

Most machine learning happens in the cloud, and most cloud platforms assume you will interact with it to some degree with the terminal. As such, it is critical to know the basics of the Linux command line to do MLOps. This section aims to bootstrap you with just enough knowledge to ensure you have success doing MLOps.

There is often a look of both shock and horror when I expose a student to the terminal. The initial reaction is reasonable in most modern computing areas due to the power of GUI interfaces like the MacOS operating system or Windows. However, a better way to think about the terminal is the "advanced settings" of the environment you are working on: the cloud, machine learning, or programming. If you need to do advanced tasks, it is the way to perform them. As a result, competence in the Linux terminal can enormously enhance any skill set. Further, it is a better idea to develop in a cloud shell environment, in most cases, which assumes familiarity with Bash and Linux.

Most servers now run Linux; many new deployment options are using containers that also run Linux. The MacOS operating system terminal is close enough to Linux that most commands are similar, especially if you install third-party tools like Homebrew. You should know the Bash terminal, and this section will give you enough knowledge to be competent with it.

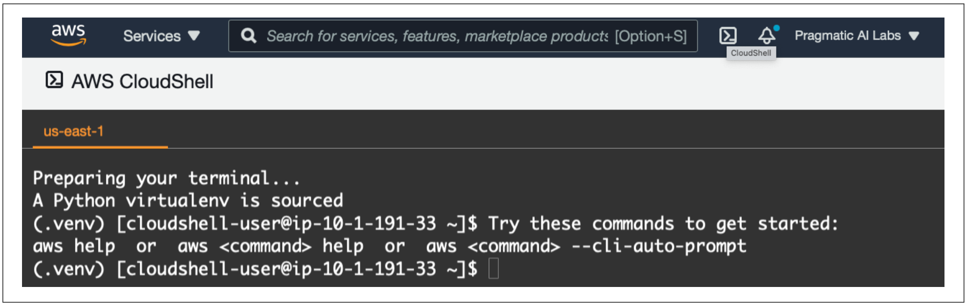
What are the critical and minimalistic components of the terminal to learn? These components include using a cloud-based shell development environment, Bash shell and commands, files and navigation, input/output, configuration, and writing a script. So let's dive into each one of these topics.

Cloud Shell Development Environments

Whether you are new to cloud computing or have decades of experience, it is worth shifting gears from a personal workstation to a web-based cloud shell development environment. A good analogy is a surfer who wants to surf each day at the beach. In theory, they could drive 50 miles each way to the beach each day, but it would be widely inconvenient, inefficient, and expensive. The better strategy, if you could afford it, would be to live at the beach, wake up each morning, walk to the beach, and surf.

Similarly, there are multiple problems a cloud development environment solves: it is more secure since you don't need to pass around developer keys. Many problems are intractable using a local machine since you may not transfer extensive data back and forth from the cloud. The tools available in cloud-based development include deep integration, which makes work more efficient. Unlike moving to the beach, a cloud development environment is free. All major clouds make their cloud development environments available on free tiers. If you are new to these environments, I would recommend starting with the AWS Cloud platform. There are two options to get started on AWS. The first option is the AWS CloudShell shown in Figure 2-1.

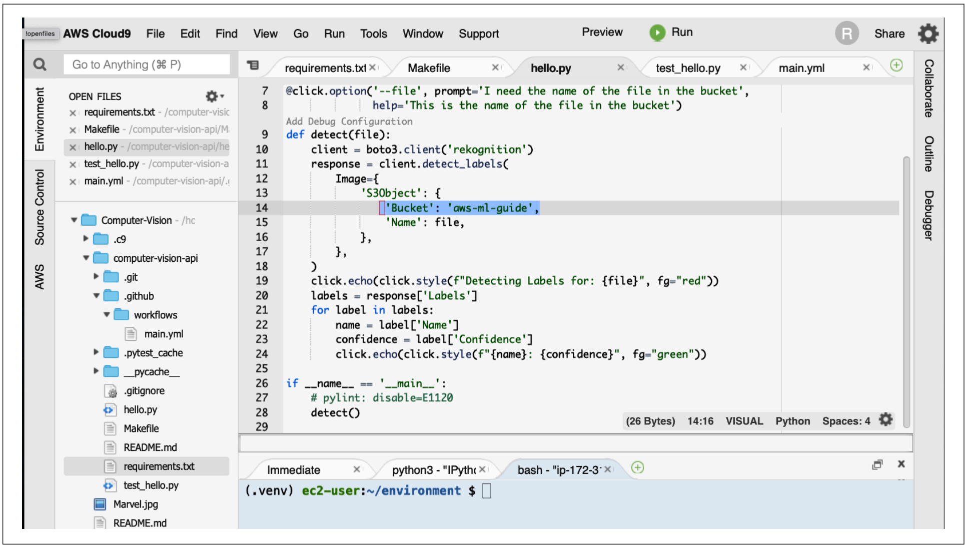
The AWS CloudShell is a Bash shell with unique AWS command completion built in the shell. If you regularly use the AWS CloudShell, it is a good idea to edit the `~/.bashrc` to customize your experience. To do that, you can use the built-in `vim` editor. Many people put off learning `vim`, but they must be proficient in the cloud shell era. You can refer to the official vim FAQ for how to get things done.



```
Preparing your terminal...
A Python virtualenv is sourced
(.venv) [cloudshell-user@ip-10-1-191-33 ~]$ Try these commands to get started:
aws help  or  aws <command> help  or  aws <command> --cli-auto-prompt
(.venv) [cloudshell-user@ip-10-1-191-33 ~]$
```

Figure 2-1. AWS CloudShell

A second option on AWS is the AWS Cloud9 development environment. A critical difference between AWS CloudShell and the AWS Cloud9 environment is that it is a more comprehensive way to develop software solutions. For example, you can see a shell and a GUI editor in Figure 2-2 to do syntax highlighting for multiple languages, including Python, Go, and Node.



```
@click.option('--file', prompt='I need the name of the file in the bucket',
              help='This is the name of the file in the bucket')
def detect(file):
    client = boto3.client('rekognition')
    response = client.detect_labels(
        Image={
            'S3Object': {
                'Bucket': 'aws-ml-guide',
                'Name': file,
            },
        },
    )
    click.echo(click.style(f"Detecting Labels for: {file}", fg="red"))
    labels = response['Labels']
    for label in labels:
        name = label['Name']
        confidence = label['Confidence']
        click.echo(click.style(f"{name}: {confidence}", fg="green"))

if __name__ == '__main__':
    # pylint: disable=E1120
    detect()
```

```
(.venv) ec2-user:~/environment $
```

Figure 2-2. AWS Cloud9 development environment

In particular, when developing machine learning microservices, the Cloud9 environment is ideal since it allows you to make web requests from the console to deployed services and has deep integration with AWS Lambda. On the other hand, suppose you are on another platform, like Microsoft Azure or Google Cloud. In that case, the same concepts apply in that the cloud-based development environments are the ideal location to build machine learning services.

There is an optional video resource I created called "Bash Essentials for Cloud Computing" that walks you through the basics. You can view it on the O'Reilly platform or the Pragmatic AI Labs YouTube Channel.

Bash Shell and Commands

The shell is an interactive environment that contains a prompt and the ability to run commands. Most shells today run either Bash or ZSH.

A couple of immediately valuable things to do in an environment you commonly use for development are installing ZSH and `vim` configurations. For `vim`, one recommended setting is awesome vim, and for ZSH, there is ohmyzsh.

What is "the shell"? Ultimately it is a user interface that controls a computer, just like the MacOS Finder. As MLOps practitioners, it is worth knowing how to use the most robust user interface, the command line, for people who work with data. Here are a few things you can do.

List Files

With the shell, you can list files through the `ls` command. The flag `-l` adds additional listing information:

```
bash-3.2$ ls -l
total 11
drwxrwxr-x 130 root admin 4160 Jan 20 22:00 Applications
drwxr-xr-x  75 root wheel 2400 Dec 14 23:13 Library
drwxr-xr-x@  9 root wheel  288 Jan 1 2020 System
drwxr-xr-x  6 root admin  192 Jan 1 2020 Users
```

Run Commands

In a GUI, you click a button or open an application to do work. In the shell, you run a command. There are many helpful built-in commands in the shell, and they often work well together. For example, an excellent way to figure out the location of shell executables is to use `which`. Here is an example:

```
bash-3.2$ which ls
/bin/ls
```

Notice that the `ls` command is in the */bin* directory. This "hint" shows that I can find other executables in this directory. Here is a count of the executables in */bin/* (the pipe operator `|` will be explained in just a bit, but, in short, it accepts the input from another command):

```
bash-3.2$ ls -l /bin/ | wc -l
37
```

Files and Navigation

In a GUI, you open a folder or a file; you use commands to accomplish the same thing in a shell.

`pwd` shows the full path to where you are:

```
bash-3.2$ pwd
/Users/noahgift
```

`cd` changes into a new directory:

```
bash-3.2$ cd /tmp
```

Input/Output

In a previous example, the output of `ls` redirects to another command. Piping is an example of input and output operations working to accomplish a more sophisticated task. It is common to use the shell to pipe one command into another.

Here is an example that shows a workflow with both a redirect and a pipe. Notice that first, the words "foo bar baz" direct to a file called *out.txt*. Next, the contents of this file print out via `cat`, and then they pipe into the command `wc`, which can count either the number of words via `-w` or characters via `-c`:

```
bash-3.2$ cd /tmp
bash-3.2$ echo "foo bar baz" > out.txt
bash-3.2$ cat out.txt | wc -c
      12
bash-3.2$ cat out.txt | wc -w
       3
```

Here is another example that directs the output of the `shuf` command to a new file. You can download that file from my GitHub repository. The `shuf` executable "shuffles" a file while limiting it to the rows specified. In this case, it takes an almost 1 GB file and takes the first 100,000 rows, and outputs a new file using the `>` operator:

```
bash-3.2$ time shuf -n 100000 en.openfoodfacts.org.products.tsv >\
10k.sample.en.openfoodfacts.org.products.tsv
1.89s user 0.80s system 97% cpu 2.748 total
```

Using a shell technique like this can save the day when working on a CSV file too large to process data science libraries on a laptop.

Configuration

The ZSH and Bash shell configuration files store settings that invoke each new time a terminal opens. As I mentioned earlier, customizing your Bash environment is recommended in a cloud-based development environment. For ZSH, an excellent place to start is *.zshrc*, and for Bash, it is *.bashrc*. Here is an example of something I store in my *.zshrc* configuration on my MacOS laptop. The first item is an alias that allows me to type the command `flask-azure-ml`, `cd` into a directory, and source a Python virtual environment in one fell swoop. The second section is where I export the AWS command line tool variables so I can make API calls:

```
## Flask ML Azure
alias flask-azure-ml="/Users/noahgift/src/flask-ml-azure-serverless &&\
source ~/.flask-ml-azure/bin/activate"

## AWS CLI
export AWS_SECRET_ACCESS_KEY="<key>"
export AWS_ACCESS_KEY_ID="<key>"
export AWS_DEFAULT_REGION="us-east-1"
```

In summary, my recommendation is to customize your shell configuration file for both your laptop and for cloud-based development environments. This small investment pays big dividends as you build in automation to your regular workflows.

Writing a Script

It can be a bit daunting to think about writing your first shell script. The syntax is much scarier than Python with weird characters. Fortunately, in many ways, it is easier to get started. The best way to write a shell script is to put a command into a file, then run it. Here is an excellent example of a "hello world" script.

The first line is called the "shebang" line and tells the script to use Bash. The second line is a Bash command, `echo`. The nice thing about a Bash script is that you can paste any commands you want in it. This fact makes the automation of small tasks straightforward even with little to no knowledge of programming:

```
#!/usr/bin/env bash

echo "hello world"
```

Next, you use the `chmod` command to set the executable flag to make this script executable. Finally, you run it by appending `./`:

```
bash-3.2$ chmod +x hello.sh
bash-3.2$ ./hello.sh
Hello World
```

The main takeaway of the shell is that you must have at least some basic skills to do MLOps. However, it is easy to get started, and before you know it, you can dramatically improve things you do daily via the automation of shell scripts and the use of the Linux command line. Next, let's talk get started with an overview of the essential parts of cloud computing.

Cloud Computing Foundations and Building Blocks

It is safe to say that almost all forms of machine learning require cloud computing in some form. A key concept in cloud computing is the idea of near-infinite resources, as described in "Above the Clouds: A Berkeley View of Cloud Computing". Without the cloud, it simply isn't feasible to do many machine learning models. For example, Stephen Strogatz, the author of *Infinite Powers: How Calculus Reveals the Secrets of the Universe* (Mariner Books), makes the case that "By wielding infinity in just the right way, calculus can unlock the secrets of the universe." For centuries specific problems like finding the shape of a circle were impossible without calculus to deal with infinite numbers. It is the same with cloud computing; many issues in machine learning, especially the operationalizing of models, are not feasible without the cloud. As shown in Figure 2-3, the cloud provides near-infinite compute and storage and works on the data without moving it.

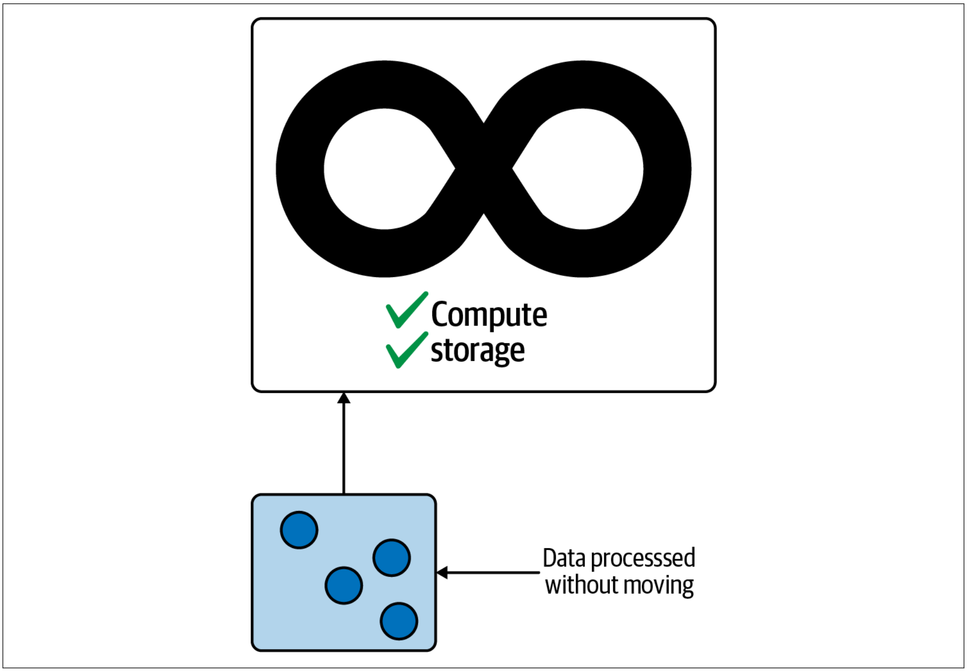


Figure 2-3. Cloud computing harnesses near-infinite compute and data

It turns out that the ability to harness computing power on data without moving it, using near-infinite resources via machine learning platforms like AWS SageMaker or Azure ML Studio, is the killer feature of the cloud not replicable without cloud computing. Coupled with this killer feature is something I call the "Automator's law." Once the general public starts talking about automation of a vertical—self-driving cars, IT, factories, machine learning—it eventually happens.

This concept doesn't mean that some magic unicorn appears that sprinkles fairy dust on projects, and they get more manageable; it is that humans are good at spotting trends as a collective group. For example, when I worked in the television industry as a teenager, there was only the concept of "linear" editing. This workflow meant you needed three different tapes to dissolve to a black screen—the source tape, the edit master tape, and a third tape that contained the black footage.

I remember people talking about how much work it was to keep swapping out new tapes and how it would be fantastic if this workflow became automated. Later it did become fully automated through the introduction of nonlinear editing. This technology allows you to store digital copies of the material and perform digital manipulation of footage versus inserting new material in a linear tape. These nonlinear editing systems cost hundreds of thousands of dollars in the early 1990s. Now I do more complicated editing on my thousand-dollar laptop with enough storage capacity to store thousands of such tapes.

The same scenario occurred with cloud computing in the early 2000s. Many companies I worked at used their own data centers staffed by teams of people that maintained them. When initial components of cloud computing cropped up, many people said, "I bet companies in the future can control an entire data center on their laptop." Many expert data center technicians scoffed at the idea of their job falling victim to automation, yet, the Automator's law struck again. A majority of companies in 2020 and beyond have some form of cloud computing, and newer jobs involve harnessing that power.

Similarly, the automation of machine learning via AutoML is a nontrivial advancement that enables the creation of models more quickly, with higher accuracy and better explainability. As a result, jobs in the data science industry will change, just like editors and data center operator jobs changed.

AutoML is the automation of the modeling aspect of machine learning. A crude and straightforward example of AutoML is an Excel spreadsheet that performs linear regression. You tell Excel which column is the target to predict and then which column is the feature.

More sophisticated AutoML systems work similarly. You select what value you want to predict—for example, image classification, a numerical trend, categorical text classification, or perhaps clustering. Then the AutoML software system performs many of the same techniques that a data scientist would perform, including hyperparameter tuning, algorithm selection, and model explainability.

All major cloud platforms have AutoML tools embedded into MLOps platforms. As a result, AutoML is an option for all cloud-based machine learning projects and is increasingly becoming another productivity enhancement for them.

Tyler Cowen is an economist, author, and columnist for Bloomberg and grew up playing competitive chess. In his book *Average is Over* (Plume), Cowen mentions that chess software eventually beat humans, also proving the Automator's law in action. Surprisingly, though, at the end of Cowen's book, expert humans and chess software won versus chess software alone. Ultimately, this story may occur with machine learning and data science. Automation may replace simple machine learning tasks and make the domain expert humans controlling the ML automation orders of magnitude more effective.

Getting Started with Cloud Computing

A recommended approach to getting started with cloud computing is setting up a multicloud development environment, as shown in the O'Reilly Video Course: Cloud Computing with Python. This video is an excellent companion to this section but isn't required to follow along. The basic structure of a multicloud environment shows that a cloud shell is something all these clouds have in common, as shown in Figure 2-4.

A source control repository using GitHub, or a similar service, is the central location where all three cloud environments initially communicate. Each of the three clouds—AWS, Azure, and GCP—has cloud-based development environments via a cloud shell. In Chapter 1, a necessary Python scaffolding showed the advantages of developing a repeatable and testable structure. A CI/CD (continuous integration/continuous delivery) process via GitHub Actions ensures the Python code is working and of high quality.

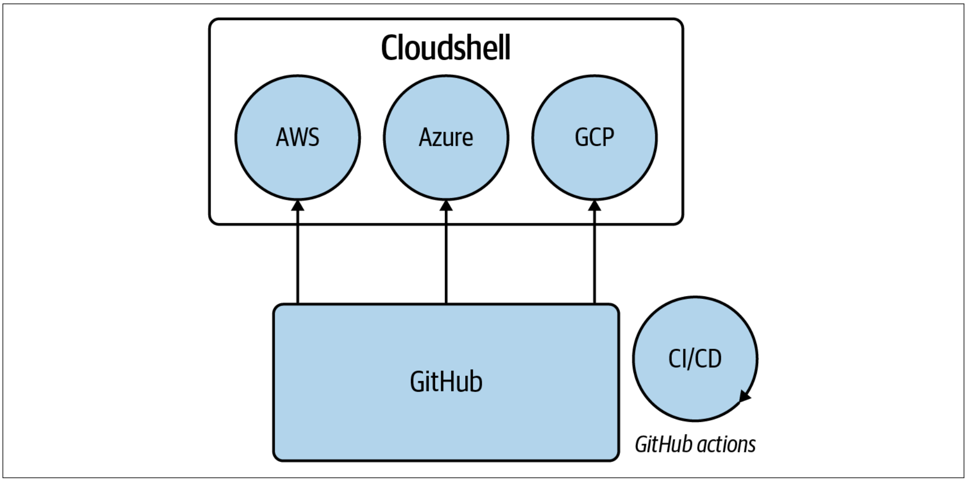


Figure 2-4. Starting cloud computing

Testing and linting Python code is a process that validates the quality of a software project. A developer will run testing and linting locally to assist with keeping software quality high. This process is similar to having a robotic vacuum machine (robovac) you turn on when you want to clean a room in your house. The robovac is a helpful assistant that keeps your room in a known good state, and running lints and testing your code keeps your code in a known good state.

A CI/CD pipeline runs these quality-control checks in an external environment to ensure the application is working before its release into another production. This pipeline allows software deployment to be repeatable and reliable and is a modern software engineering best practice—another word for these software engineering best practices is DevOps.

Using all three major clouds is a great way to familiarize yourself with cloud computing if you are learning the cloud. This cross-cloud workflow helps because it solidifies knowledge. After all, the names of things are different, but the concepts are identical. If you need help with some of the terminologies, refer to Appendix A.

Let's dive into Python next, which is the de facto language of DevOps, cloud computing, data science, and machine learning.

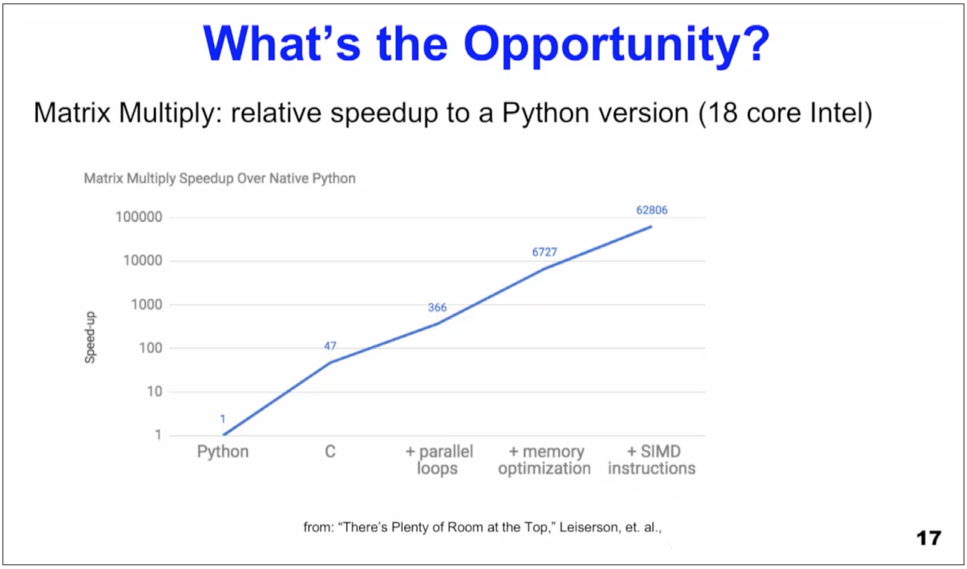
Python Crash Course

A key reason for Python's dominance is that the language is optimized for the developer and not for the computer. Languages like C or C++ have excellent performance because they are "lower-level," meaning the developer must work much harder in solving a problem. For example, a C programmer must allocate memory, declare types, and compile a program. On the other hand, a Python programmer can put in some commands and run them, and there is often significantly less code in Python.

For that convenience, though, Python's performance is much slower than C, C#, Java, and Go. Further, Python has limitations inherent to the language itself, including lack of true threads, lack of a JIT (Just in Time) compiler, and lack of type inference found in languages like C# or F#. With cloud computing, though, language performance does not bind to many problems. So it would be fair to say that Python's performance got accidentally lucky because of two things: cloud computing and containers. With Cloud computing, the design is fully distributed, building on top of it using technologies like AWS Lambda and AWS SQS (Simple Queuing Service). Similarly, containerized technology like Kubernetes does the heavy lifting of building distributed systems, so Python threads become suddenly irrelevant.

> AWS Lambda is a function as a service (FaaS) technology that runs on the AWS platform. It has the name FaaS because an AWS Lambda function can be just a few lines of code—literally a function. These functions can then attach to events like a cloud queuing system, Amazon SQS, or an image uploaded to Amazon S3 object storage.
>
> One way to think about a cloud is that it is an operating system. In the mid-1980s, Sun Computer used the marketing phrase, "The Network is the Computer." This slogan may have been premature in 1980, but in 2021 it is very accurate. For example, instead of spawning threads on a single machine, you could spawn AWS Lambda functions in the cloud, which behaves like an operating system with infinitely scalable resources.

In a talk I attended at Google, Dr. Patterson, a retired Berkeley Computer Science Professor and co-creator of the TPU (TensorFlow Processing Unit), mentioned that Python is 64,000 times slower than equivalent matrix operations in C shown in Figure 2-5. This fact is in addition to not having true threads.



What's the Opportunity?

Matrix Multiply: relative speedup to a Python version (18 core Intel)

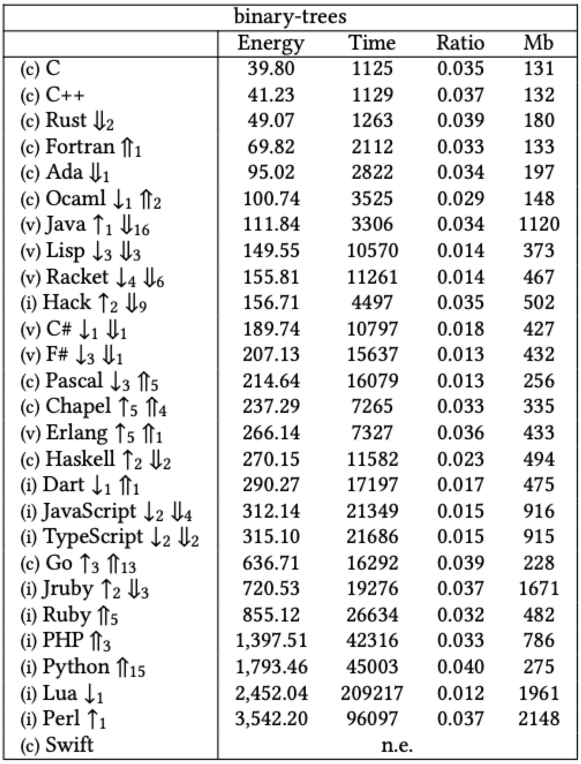
from: "There's Plenty of Room at the Top," Leiserson, et. al.,

17

Figure 2-5. Python performance is 64,000 times worse than matrix operations in C

Simultaneously, a research paper, "Energy Efficiency across programming languages", shows that many operations in Python require 50 times more energy than equivalent operations in C. This research into energy efficiency also pegs Python as one of the worst-performing languages regarding how much power it requires to perform tasks compared to other languages, as shown in Figure 2-6. As Python has surged in popularity as one of the most popular languages on the planet, it does raise concerns on whether it is more like a coal powerplant than a green energy solar system. Ultimately, Python could need rescuing again for energy consumption. One solution could be for a major cloud vendor to actively build a new runtime for Python that takes advantage of modern computer science techniques like a JIT compiler.

One final technical hurdle I have seen with newcomers to data science programming is their adoption of the traditional computer science approach to learning to code. For example, cloud computing and machine learning are very different from conventional software engineering projects like developing user-facing applications via a GUI (graphical user interface). Instead, much of the cloud and ML world involves writing small functions. Most of the time, these other Python parts are not needed, i.e., object-oriented code, and discourage a newcomer from speaking the language.



	binary-trees			
	Energy	Time	Ratio	Mb
(c) C	39.80	1125	0.035	131
(c) C++	41.23	1129	0.037	132
(c) Rust $\Downarrow_2$	49.07	1263	0.039	180
(c) Fortran $\Uparrow_1$	69.82	2112	0.033	133
(c) Ada $\Downarrow_1$	95.02	2822	0.034	197
(c) Ocaml $\downarrow_1$ $\Uparrow_2$	100.74	3525	0.029	148
(v) Java $\uparrow_1$ $\Downarrow_{16}$	111.84	3306	0.034	1120
(v) Lisp $\downarrow_3$ $\Downarrow_3$	149.55	10570	0.014	373
(v) Racket $\downarrow_4$ $\Downarrow_6$	155.81	11261	0.014	467
(i) Hack $\uparrow_2$ $\Downarrow_9$	156.71	4497	0.035	502
(v) C# $\downarrow_1$ $\Downarrow_1$	189.74	10797	0.018	427
(v) F# $\downarrow_3$ $\Downarrow_1$	207.13	15637	0.013	432
(c) Pascal $\downarrow_3$ $\Uparrow_5$	214.64	16079	0.013	256
(c) Chapel $\uparrow_5$ $\Uparrow_4$	237.29	7265	0.033	335
(v) Erlang $\uparrow_5$ $\Uparrow_1$	266.14	7327	0.036	433
(c) Haskell $\uparrow_2$ $\Downarrow_2$	270.15	11582	0.023	494
(i) Dart $\downarrow_1$ $\Uparrow_1$	290.27	17197	0.017	475
(i) JavaScript $\downarrow_2$ $\Downarrow_4$	312.14	21349	0.015	916
(i) TypeScript $\downarrow_2$ $\Downarrow_2$	315.10	21686	0.015	915
(c) Go $\uparrow_3$ $\Uparrow_{13}$	636.71	16292	0.039	228
(i) Jruby $\uparrow_2$ $\Downarrow_3$	720.53	19276	0.037	1671
(i) Ruby $\Uparrow_5$	855.12	26634	0.032	482
(i) PHP $\Uparrow_3$	1,397.51	42316	0.033	786
(i) Python $\Uparrow_{15}$	1,793.46	45003	0.040	275
(i) Lua $\downarrow_1$	2,452.04	209217	0.012	1961
(i) Perl $\uparrow_1$	3,542.20	96097	0.037	2148
(c) Swift	n.e.			

Figure 2-6. The Python language is among the worst offenders in energy consumption

Computer science topics include concurrency, object-oriented programming, metaprogramming, and algorithm theory. Unfortunately, studying these topics is orthogonal to the programming style necessary for most programming in cloud computing and data science. It isn't that these topics are not valuable; they are beneficial to the creators of platforms, libraries, and tools. If you are not initially "creating" libraries and frameworks but "using" libraries and frameworks, then you can safely ignore these advanced topics and stick to functions.

This crash-course approach temporarily ignores the creator of code used by others in favor of the consumer of code and libraries, i.e., the data scientist or MLOps practitioner. This brief crash course is for the consumer, where most people spend their career in data science. After these topics, if you are curious, you will have a solid foundation to move onto more complex computer science–focused topics. These advanced topics are not necessary to be productive immediately in MLOps.

Minimalistic Python Tutorial

If you wanted to learn the smallest amount of Python necessary to get started, what would you need to know? The two most essential components of Python are statements and functions. So let's start with Python statements. A Python statement is an instruction to a computer; i.e., similar to telling a person "hello," you can say to a computer to "print hello." The following example is the Python interpreter. Notice, the "statement" in the example is the phrase `print("Hello World")`:

```
Python 3.9.0 (default, Nov 14 2020, 16:06:43)
[Clang 12.0.0 (clang-1200.0.32.27)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World")
Hello World
```

With Python, you can also chain together two statements with a semicolon. For example, I import the `os` module, which has a function I want to use, `os.listdir`, then I call it to list the contents of a directory I am inside:

```
>>> import os;os.listdir(".")
['chapter11', 'README.md', 'chapter2', '.git', 'chapter3']
```

This pattern is pervasive in data science notebooks, and this is all you need to know to get started with Python. I would recommend trying things out in Python or IPython, or Jupyter REPL as the first way to get familiar with Python.

The second item to know is how to write and use functions in Python. Let's do that in the following example. This example is a two-line function that adds two numbers together, `x` and `y`. The entire point of a Python function is to serve as a "unit of work." For example, a toaster in the kitchen works as a unit of work. It takes bread as input, warms the bread, then returns toast. Similarly, once I write the `add` function, I can use it as many times as possible with new inputs:

```
>>> def add(x,y):
...   return x+y
...
>>> add(1,1)
2
>>> add(3,4)
7
>>>
```

Let's assemble our knowledge and build a Python script. At the beginning of a Python script, there is a shebang line, just like in Bash. Next, the `choices` library is imported. This module is later used in a "loop" to send random numbers to the `add` function:

```
#!/usr/bin/env python
from random import choices

def add(x,y):
    print(f"inside a function and adding {x}, {y}")
    return x+y

#Send random numbers from 1-10, ten times to the add function
numbers = range(1,10)
for num in numbers:
    xx = choices(numbers)[0]
    yy = choices(numbers)[0]
    print(add(xx,yy))
```

The script needs to be made executable by running `chmod +x add.py`, just like a Bash script:

```
bash-3.2$ ./add.py
inside a function and adding 7, 5
12
inside a function and adding 9, 5
14
inside a function and adding 3, 1
4
inside a function and adding 7, 2
9
inside a function and adding 5, 8
13
inside a function and adding 6, 1
7
inside a function and adding 5, 5
10
inside a function and adding 8, 6
14
inside a function and adding 3, 3
6
```

You can learn about Python a lot more, but "toying" around with examples like the one shown here is perhaps the quickest way to go from "zero to one." So let's move on to another topic, math for programmers, and tackle it in a minimalistic fashion as well.

Math for Programmers Crash Course

Math can be both daunting and irritating, but understanding the basics is essential for working with machine learning. So let's tackle a few useful and essential ideas.

Descriptive Statistics and Normal Distributions

Many things in the world are "normally" distributed. A good example is height and weight. If you plot the height of every person in the world, you will get a "bell-shaped" distribution. This distribution is intuitive in that most people you encounter are of average height, and it is unusual to see a seven-foot-tall basketball player. Let's walk through a Jupyter notebook containing 25,000 records of human heights and weights of 19-year-old children:

```
In [0]:
import pandas as pd

In [7]:
df = pd.read_csv("https://raw.githubusercontent.com/noahgift/\
regression-concepts/master/height-weight-25k.csv")

Out[7]:
IndexHeight-InchesWeight-Pounds
01      65.78331        112.9925
12      71.51521        136.4873
23      69.39874        153.0269
34      68.21660        142.3354
45      67.78781        144.2971
```

Next, a plot, shown in Figure 2-7, shows a linear relationship between height and weight, which most of us intuitively know. The taller you are, the more you weigh:

```
In [0]:
import seaborn as sns
import numpy as np

In [9]:
sns.lmplot("Height-Inches", "Weight-Pounds", data=df)
```

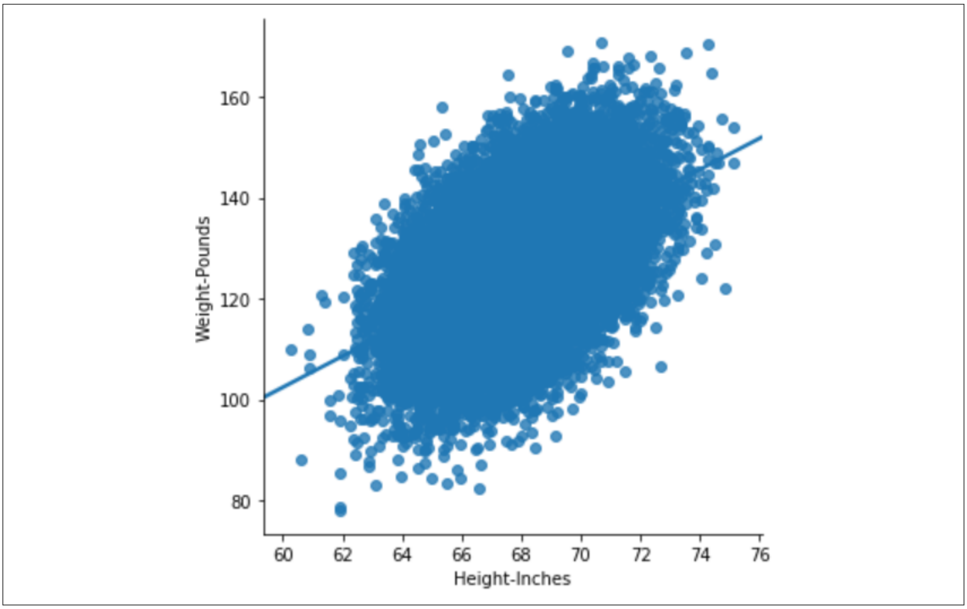
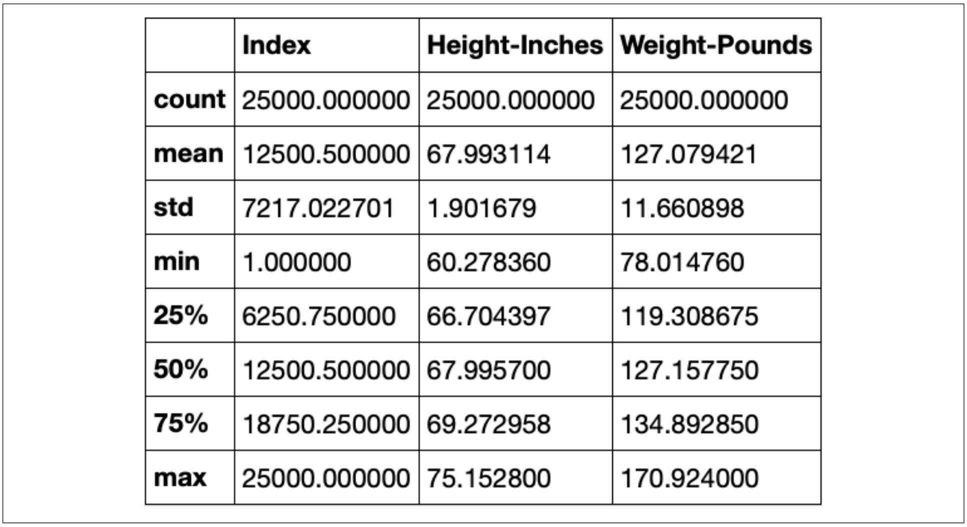


Figure 2-7. Height and weight

The step of visualizing the data in a dataset is called "Exploratory Data Analysis." The general idea is to "look around" using a combination of math and visualization. The next step is to look at the descriptive statistics of this "normal distribution."

In Pandas, you get these descriptive statistics through the use of `df.describe()`. One way to consider descriptive statistics is to view them as a way of "seeing" numerically what the eye sees visually. For example, the 50th percentile, or median, shows the number that represents the exact middle height. This value is about 68 inches. The max statistic in this dataset is 75 inches. Max represents the most extreme observation or the tallest person measured for this dataset. The max observation in a normally distributed dataset is rare, just as the minimum is. You can see this trend in Figure 2-8. The `DataFrame` in Pandas comes with a describe method that when called gives a full range of descriptive statistics :

```
In [10]: df.describe()
```



	Index	Height-Inches	Weight-Pounds
count	25000.000000	25000.000000	25000.000000
mean	12500.500000	67.993114	127.079421
std	7217.022701	1.901679	11.660898
min	1.000000	60.278360	78.014760
25%	6250.750000	66.704397	119.308675
50%	12500.500000	67.995700	127.157750
75%	18750.250000	69.272958	134.892850
max	25000.000000	75.152800	170.924000

Figure 2-8. Height/weight descriptive statistics

One of the best ways to visualize height and weight's normal bell-shaped distribution is to use a Kernel Density plot:

```
In [11]:
sns.jointplot("Height-Inches", "Weight-Pounds", data=df, kind="kde");
```

Both the weight and the height show a "bell distribution." The extreme values are rare, and most of the values are in the middle, as shown in Figure 2-9.

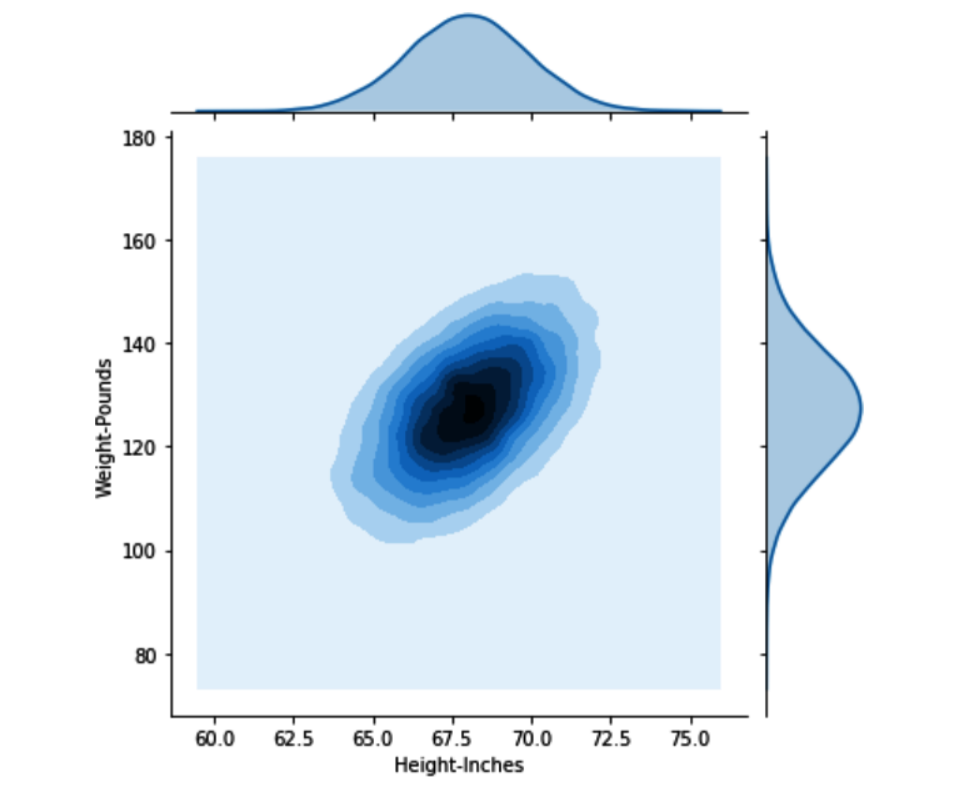


Figure 2-9. Kernel Density plot of weight and height

Machine learning heavily builds on the idea of a normal distribution, and having this intuition goes a long way to building and maintaining machine learning models. However, it is essential to note that other distributions are beyond the normal distribution, making the world much harder to model. An excellent example of this is what the author Nassim Taleb calls "fat tails," i.e., hard-to-predict and rare events that significantly affect the world.

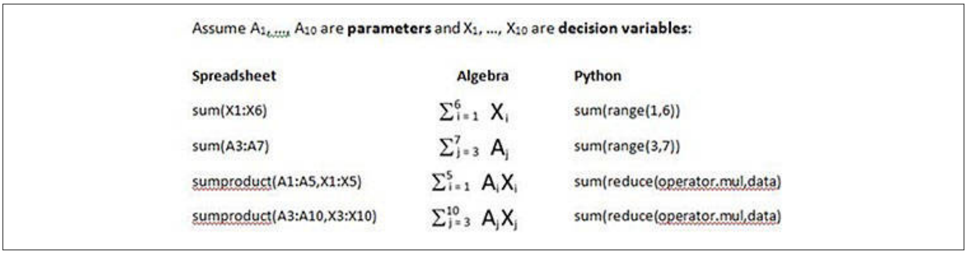
Another example of the danger of being too confident in modeling the world can be found in Dr. Steven Koonin's book, *Unsettled* (BenBella Books). I worked with Dr. Koonin when he was in the administration of Caltech and found him to be an enthusiastic scientist and fun person to have a random conversation with. Here is a quote about modeling from his book:

> Since we have a very solid understanding of the physical laws that govern matter and energy, it's easy to be seduced by the notion that we can just feed the present state of the atmosphere and oceans into a computer, make some assumptions about future human and natural influences, and so accurately predict the climate decades into the future. Unfortunately, that's just a fantasy, as you might infer from weather forecasts, which can be accurate only out to two weeks or so.

Optimization

A fundamental problem in machine learning is the concept of optimization. Optimization is the ability to find the best, or good enough, solution to a problem. Gradient descent is an optimization algorithm at the heart of deep learning. The goal of gradient descent is to convert to the global minimum, i.e., the optimal solution, versus getting stuck in a local minimum. The intuition behind the algorithm is relatively straightforward if you imagine walking down a mountain in the dark. The global minimum solution means you get off the mountain alive at the bottom. The local minimum means you walked accidentally into a lake on the side of the mountain 1,000 feet from the bottom.

Let's walk through an example of optimization problems. An excellent place to start is by observing the types of notation involved with optimization. When creating a model, you need to understand the Python method of notating algebraic expressions. The quick summary in Figure 2-10 compares terms between a spreadsheet, algebra, and Python. A key takeaway is you can do the same thing on a whiteboard, in Excel, or with a bit of code.



Assume $A_1, \ldots, A_{10}$ are **parameters** and $X_1, \ldots, X_{10}$ are **decision variables**:

Spreadsheet	Algebra	Python
sum(X1:X6)	$\sum_{i=1}^{6} X_i$	sum(range(1,6))
sum(A3:A7)	$\sum_{j=3}^{7} A_j$	sum(range(3,7))
sumproduct(A1:A5,X1:X5)	$\sum_{i=1}^{5} A_i X_i$	sum(reduce(operator.mul,data)
sumproduct(A3:A10,X3:X10)	$\sum_{j=3}^{10} A_j X_j$	sum(reduce(operator.mul,data)

Figure 2-10. Notation is relative

Now, let's look at a solution to making the correct change. You can find the code to this solution on GitHub. The general idea with this code example is to pick a *greedy* solution to make the change. Greedy algorithms work by always taking the best option first. They also work well if you don't care about the perfect solution, or it is impossible to find the perfect solution, but you are OK with a "good enough" solution. In this case, it would be the highest value coins and making the change with those, then moving onto the next highest:

```
python change.py --full 1.34

Quarters 5: , Remainder: 9
Dimes 0: , Remainder: 0
Nickles 1: , Remainder: 4
Pennies 4:
```

The following is the core section of the code that does a greedy match. Note that a recursive function solves each iteration of the problem since the large coins eventually run out. The algorithm next finds medium coins that run out; finally, it moves to the smallest coins:

```
    def recursive_change(self, rem):
        """Greedy Coin Match with Recursion
        >>> c = Change(.71)
        >>> c.recursive_change(c.convert)
        2 quarters
        2 dimes
        1 pennies
        [1, 0, 2, 2]

        """
        if len(self.coins) == 0:
            return []
        coin = self.coins.pop()
        num, new_rem = divmod(rem, coin)
        self.printer(num,coin)
        return self.recursive_change(new_rem) + [num]
```

While there are many different ways to express the concept algorithmically, the idea is the same. Without knowing how to find the "perfect" solution, the appropriate answer is always picking the best choice when presented. Intuitively, this is like walking to a city's diagonal destination and going straight or turning right each time the light ahead of you turns red.

Here is a series of tests for this algorithm. They show how the algorithm performs, which is often a good idea when testing a solution involving optimization:

```
#!/usr/bin/env python2.5
#Noah Gift
#Greedy Coin Match Python

import unittest
import change

class TestChange(unittest.TestCase):
    def test_get_quarter(self):
        c = change.Change(.25)
        quarter, qrem, dime, drem, nickel, nrem, penny =\
            c.make_change_conditional()
        self.assertEqual(quarter,1)  #quarters
        self.assertEqual(qrem, 0)   #quarter remainder
    def test_get_dime(self):
        c = change.Change(.20)
        quarter, qrem, dime, drem, nickel, nrem, penny =\
            c.make_change_conditional()
        self.assertEqual(quarter,0)  #quarters
        self.assertEqual(qrem, 20)   #quarter remainder
        self.assertEqual(dime, 2)    #dime
        self.assertEqual(drem, 0)    #dime remainder
    def test_get_nickel(self):
        c = change.Change(.05)
        quarter, qrem, dime, drem, nickel, nrem, penny =\
            c.make_change_conditional()
        self.assertEqual(dime, 0)    #dime
        self.assertEqual(drem, 0)    #dime remainder
        self.assertEqual(nickel, 1)  #nickel
        self.assertEqual(nrem, 0)    #nickel remainder
    def test_get_penny(self):
        c = change.Change(.04)
        quarter, qrem, dime, drem, nickel, nrem, penny =\
            c.make_change_conditional()
        self.assertEqual(penny, 4)  #nickel
    def test_small_number(self):
        c = change.Change(.0001)
        quarter, qrem, dime, drem, nickel, nrem, penny =\
            c.make_change_conditional()
        self.assertEqual(quarter,0)  #quarters
        self.assertEqual(qrem, 0)   #quarter remainder
        self.assertEqual(dime, 0)    #dime
        self.assertEqual(drem, 0)    #dime remainder
        self.assertEqual(nickel, 0)  #nickel
        self.assertEqual(nrem, 0)    #nickel remainder
        self.assertEqual(penny, 0)    #penny
    def test_large_number(self):
        c = change.Change(2.20)
        quarter, qrem, dime, drem, nickel, nrem, penny =\
            c.make_change_conditional()
        self.assertEqual(quarter, 8)  #nickel
        self.assertEqual(qrem, 20)  #nickel
        self.assertEqual(dime, 2)  #nickel
        self.assertEqual(drem, 0)  #nickel
    def test_get_quarter_dime_penny(self):
        c = change.Change(.86)
        quarter, qrem, dime, drem, nickel, nrem, penny =\
            c.make_change_conditional()
        self.assertEqual(quarter,3)  #quarters
        self.assertEqual(qrem, 11)   #quarter remainder
        self.assertEqual(dime, 1)    #dime
        self.assertEqual(drem, 1)    #dime remainder
        self.assertEqual(penny, 1)   #penny
    def test_get_quarter_dime_nickel_penny(self):
        c = change.Change(.91)
        quarter, qrem, dime, drem, nickel, nrem, penny =\
            c.make_change_conditional()
        self.assertEqual(quarter,3)  #quarters
        self.assertEqual(qrem, 16)   #quarter remainder
        self.assertEqual(dime, 1)    #dime
        self.assertEqual(drem, 6)    #dime remainder
        self.assertEqual(nickel, 1)  #nickel
        self.assertEqual(nrem, 1)    #nickel remainder
        self.assertEqual(penny, 1)    #penny

if __name__ == "__main__":
    unittest.main()
```

Next, let's build on greedy algorithms in the following problem. One of the most studied problems in optimization is the traveling salesman problem. You can find the source code on GitHub. This example is a list of routes that are in a *routes.py* file. It shows the distance between different companies in the Bay Area.

It is an excellent example of a perfect solution that doesn't exist, but a good enough one does. The general problem asks the question, "How can you travel to a list of cities and minimize the distance?"

One way to do this is to use a "greedy" algorithm. It will pick the right solution at every choice. Often this can lead to a good enough answer. In this particular example, a "random" city is chosen each time as the starting point. This example adds the ability for simulations to pick the lowest distance. A user of the simulations could simulate as many times as they have time. The smallest total length is the best answer. The following is a sample of what the input looks like before processing into the TSP algorithm:

```
values = [
("AAPL", "CSCO", 14),
("AAPL", "CVX", 44),
("AAPL", "EBAY", 14),
("AAPL", "GOOG", 14),
("AAPL", "GPS", 59),
("AAPL", "HPQ", 14),
("AAPL", "INTC", 8),
("AAPL", "MCK", 60),
("AAPL", "ORCL", 26),
("AAPL", "PCG", 59),
("AAPL", "SFO", 46),
("AAPL", "SWY", 37),
("AAPL", "URS", 60),
("AAPL", "WFC", 60),
```

Let’s run the script. First, note that it takes as an input of the complete simulations to run:

```
#!/usr/bin/env python
"""
Traveling salesman solution with random start and greedy path selection
You can select how many iterations to run by doing the following:

python greedy_random_start.py 20 #runs 20 times


"""

import sys
from random import choice
import numpy as np
from routes import values

dt = np.dtype([("city_start", "S10"), ("city_end", "S10"), ("distance", int)])
data_set = np.array(values, dtype=dt)


def all_cities():
    """Finds unique cities

    array([["A", "A"],
    ["A", "B"]])

    """
    cities = {}
    city_set = set(data_set["city_end"])
    for city in city_set:
        cities[city] = ""
    return cities


def randomize_city_start(cities):
    """Returns a randomized city to start trip"""

    return choice(cities)


def get_shortest_route(routes):
    """Sort the list by distance and return shortest distance route"""

    route = sorted(routes, key=lambda dist: dist[2]).pop(0)
    return route


def greedy_path():
    """Select the next path to travel based on the shortest, nearest path"""

    itinerary = []
    cities = all_cities()
    starting_city = randomize_city_start(list(cities.keys()))
    # print "starting_city: %s" % starting_city
    cities_visited = {}
    # we want to iterate through all cities once
    count = 1
    while True:
        possible_routes = []
        # print "starting city: %s" % starting_city
        for path in data_set:
            if starting_city in path["city_start"]:
                # we can't go to cities we have already visited
                if path["city_end"] in cities_visited:
                    continue
                else:
                    # print "path: ", path
                    possible_routes.append(path)

        if not possible_routes:
            break
        # append this to itinerary
        route = get_shortest_route(possible_routes)
        # print "Route(%s): %s " % (count, route)
        count += 1
        itinerary.append(route)
        # add this city to the visited city list
        cities_visited[route[0]] = count
        # print "cities_visited: %s " % cities_visited
        # reset the starting_city to the next city
        starting_city = route[1]
        # print "itinerary: %s" % itinerary

    return itinerary


def get_total_distance(complete_itinerary):

    distance = sum(z for x, y, z in complete_itinerary)
    return distance


def lowest_simulation(num):

    routes = {}
    for _ in range(num):
        itinerary = greedy_path()
        distance = get_total_distance(itinerary)
        routes[distance] = itinerary
    shortest_distance = min(routes.keys())
```

```
        route = routes[shortest_distance]
        return shortest_distance, route


    def main():
        """runs everything"""

        if len(sys.argv) == 2:
            iterations = int(sys.argv[1])
            print("Running simulation %s times" % iterations)
            distance, route = lowest_simulation(iterations)
            print("Shortest Distance: %s" % distance)
            print("Optimal Route: %s" % route)
        else:
            # print "All Routes: %s" % data_set
            itinerary = greedy_path()
            print("itinerary: %s" % itinerary)
            print("Distance: %s" % get_total_distance(itinerary))


    if __name__ == "__main__":
        main()
```

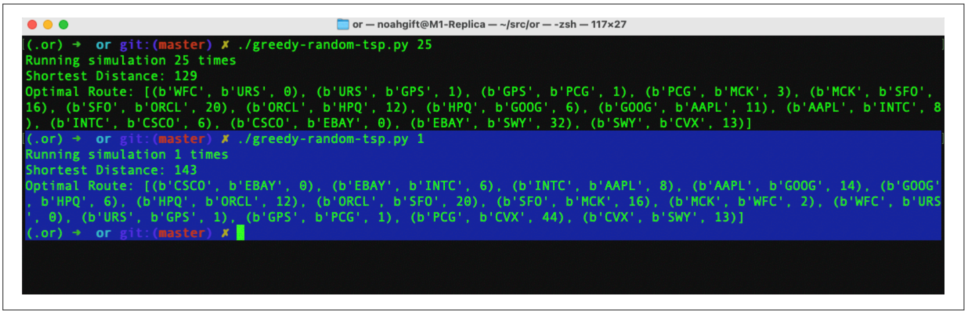
Let's run this "greedy" algorithm 25 times. Notice that it finds a "good" solution of 129. This version may or may not be the optimal solution in a more extensive set of coordinates, but it is good enough to start a road trip for our purposes:

```
> ./greedy-random-tsp.py 25
Running simulation 25 times
Shortest Distance: 129
Optimal Route: [(b'WFC', b'URS', 0), (b'URS', b'GPS', 1),\
(b'GPS', b'PCG', 1), (b'PCG', b'MCK', 3), (b'MCK', b'SFO', 16),\
(b'SFO', b'ORCL', 20), (b'ORCL', b'HPQ', 12), (b'HPQ', b'GOOG', 6),\
(b'GOOG', b'AAPL', 11), (b'AAPL', b'INTC', 8), (b'INTC', b'CSCO', 6),\
(b'CSCO', b'EBAY', 0), (b'EBAY', b'SWY', 32), (b'SWY', b'CVX', 13)]
```

Notice that if I run the simulation just once, it randomly selects a worse distance of 143:

```
> ./greedy-random-tsp.py 1
Running simulation 1 times
Shortest Distance: 143
Optimal Route: [(b'CSCO', b'EBAY', 0), (b'EBAY', b'INTC', 6),\
(b'INTC', b'AAPL', 8), (b'AAPL', b'GOOG', 14), (b'GOOG', b'HPQ', 6),\
(b'HPQ', b'ORCL', 12), (b'ORCL', b'SFO', 20), (b'SFO', b'MCK', 16),\
 (b'MCK', b'WFC', 2), (b'WFC', b'URS', 0), (b'URS', b'GPS', 1),\
 (b'GPS', b'PCG', 1), (b'PCG', b'CVX', 44), (b'CVX', b'SWY', 13)]
```

Notice in Figure 2-11 how I would run multiple iterations of the code in a real-world scenario to "try out ideas." If the dataset was enormous and I was in a hurry, I may do only a few simulations, but if I was leaving for the day, I might let it run 1,000 times and finish by the time I come back in the morning the next day. There could be many local minima in a geographic coordinates dataset—that is, solutions to the problem that don't quite reach the global minima, or the optimal solution.



```
(.or) ➜  or git:(master) ✗ ./greedy-random-tsp.py 25
Running simulation 25 times
Shortest Distance: 129
Optimal Route: [(b'WFC', b'URS', 0), (b'URS', b'GPS', 1), (b'GPS', b'PCG', 1), (b'PCG', b'MCK', 3), (b'MCK', b'SFO',
16), (b'SFO', b'ORCL', 20), (b'ORCL', b'HPQ', 12), (b'HPQ', b'GOOG', 6), (b'GOOG', b'AAPL', 11), (b'AAPL', b'INTC', 8
), (b'INTC', b'CSCO', 6), (b'CSCO', b'EBAY', 0), (b'EBAY', b'SWY', 32), (b'SWY', b'CVX', 13)]
(.or) ➜  or git:(master) ✗ ./greedy-random-tsp.py 1
Running simulation 1 times
Shortest Distance: 143
Optimal Route: [(b'CSCO', b'EBAY', 0), (b'EBAY', b'INTC', 6), (b'INTC', b'AAPL', 8), (b'AAPL', b'GOOG', 14), (b'GOOG'
, b'HPQ', 6), (b'HPQ', b'ORCL', 12), (b'ORCL', b'SFO', 20), (b'SFO', b'MCK', 16), (b'MCK', b'WFC', 2), (b'WFC', b'URS
', 0), (b'URS', b'GPS', 1), (b'GPS', b'PCG', 1), (b'PCG', b'CVX', 44), (b'CVX', b'SWY', 13)]
(.or) ➜  or git:(master) ✗
```

Figure 2-11. TSP simulation

Optimization is part of our lives, and we use greedy algorithms to solve everyday problems because they are intuitive. Optimization is also at the core of how machine learning works using gradient descent. A machine learning problem iteratively walks toward a local or global minimum using the gradient descent algorithm, as shown in Figure 2-12.

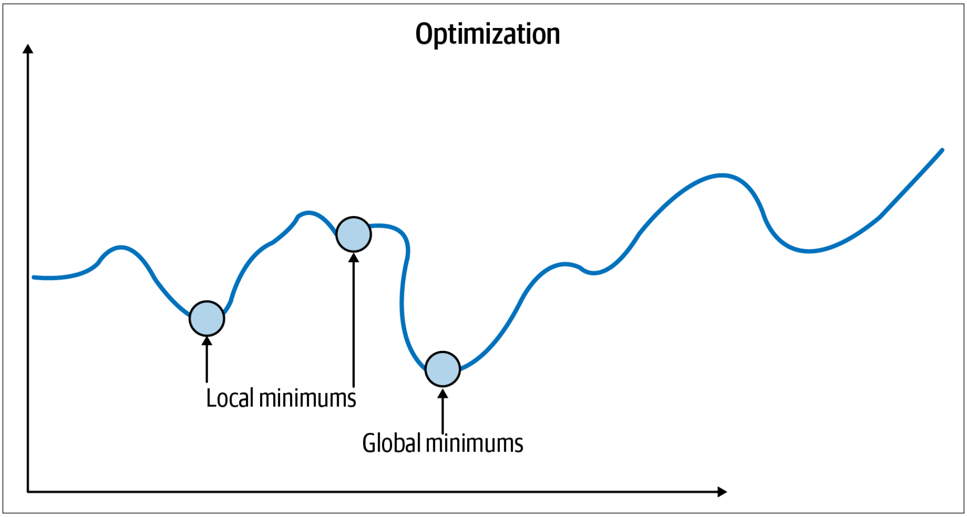
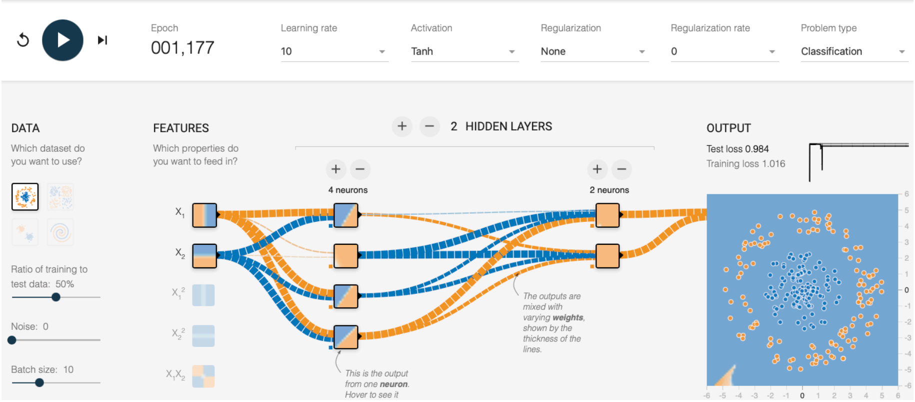


Figure 2-12. Optimization

Notes on Deep Learning Intuition

For MLOps, convergence, i.e., creating a model that finds a solution that won't be improved by adding more data, is an essential operational issue. For example, would a GPU-based training cluster allow faster convergence? Would a CPU-based training cluster offer lower costs? Operating costs could break a company or a project in the real world, and it is essential to both have an intuition for how optimization works and test it out in practice.

A valuable tool to enhance intuition about gradient descent is the TensorFlow Playground. In particular, experimenting with the learning rate shows how too high a rate can lead to oscillation, as shown in Figure 2-13. Notice how the Test loss sticks at 0.984 because the learning rate is too high to use the gradient descent algorithm effectively. Likewise, if you set the learning rate too low, it may not reach the global minimum or will take too long to converge on the right solution.



Epoch 001,177

Learning rate 10

Activation Tanh

Regularization None

Regularization rate 0

Problem type Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

FEATURES

Which properties do you want to feed in?

2 HIDDEN LAYERS

4 neurons

2 neurons

OUTPUT

Test loss 0.984

Training loss 1.016

The outputs are mixed with varying **weights**, shown by the thickness of the lines.

This is the output from one **neuron**. Hover to see it

Figure 2-13. Learning rate too high

Mathematically this set of trade-offs shows up in Figure 2-14. The optimal learning rate converges on the global minimum, but too high leads to thrashing, as shown in the TensorFlow Playground example. Alternatively, too low can lead to getting stuck in a local minimum or taking too long to converge.

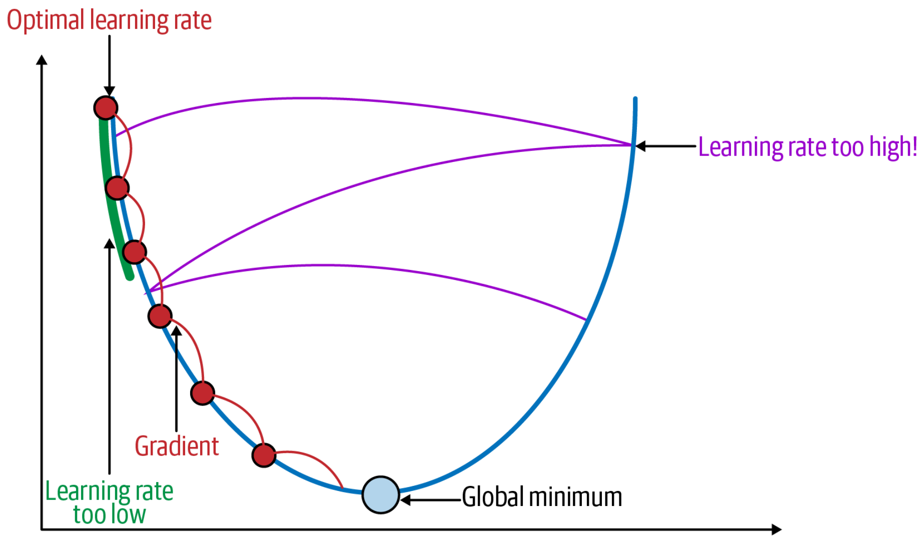


Figure 2-14. Learning rate intuition

Next, let's dive into machine learning core concepts.

Machine Learning Key Concepts

Machine learning is the ability for computers to perform tasks without explicit programming. They do this by "learning" from data. As discussed earlier, a good intuition would be a machine learning model that could predict weight based on height. It could "learn" from 25,000 observations and then give a prediction.

Machine learning involves three categories: supervised, unsupervised, and reinforcement learning. Supervised machine learning is when the "labels" are known, and the model learns from historical data. In the previous example, height and weight are labels. Additionally, the 25,000 observations are an example of historical data. Note that all machine learning requires data to be in a numerical form and requires scaling. Imagine if a friend bragged about running 50. What did they mean? Was it 50 miles or 50 feet? Magnitude is the reason for scaling data before processing a prediction.

Unsupervised machine learning works to "discover" labels. A good intuition of how this works is to consider an NBA season. In the visualization shown in Figure 2-15 from the 2015–2016 NBA season, the computer "learned" how to group the different NBA players. It is up to the domain expert, in this case, me, to select the appropriate labels. The algorithm was able to cluster groups, one of which I labeled the "best" players.

As a domain expert in basketball, I then added a label called "best." Another domain expert may disagree, though, and call these players "elite well-rounded" or some other label. Clustering is both an art and a science. Having a domain expert who understands the trade-offs for what to label a clustered dataset could make or break the usefulness of the unsupervised machine learning prediction.

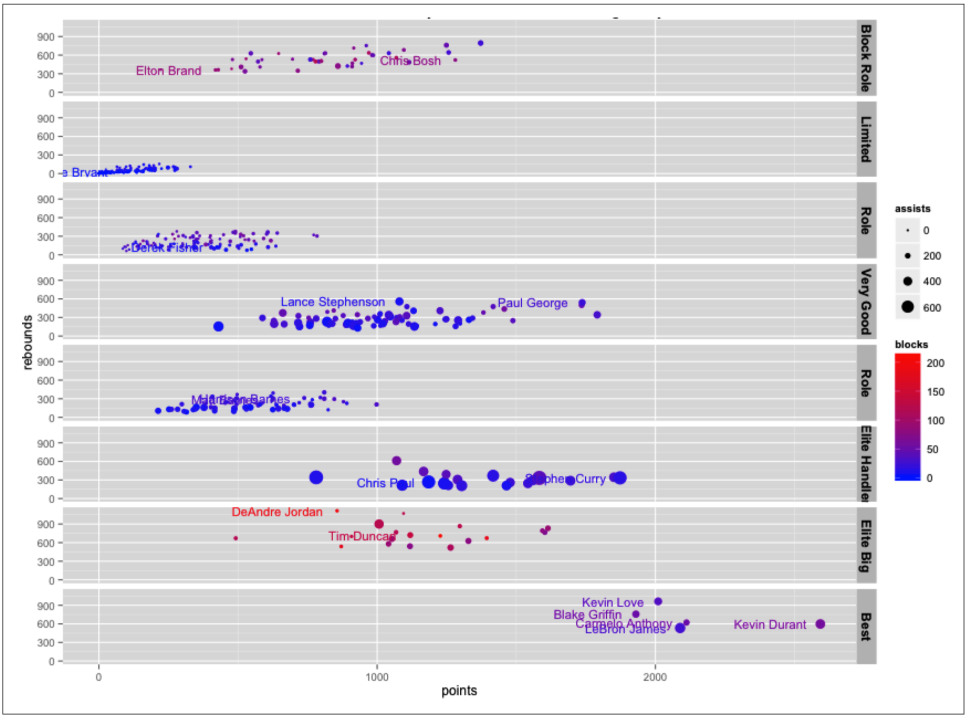


Figure 2-15. Clustered and faceted K-means clustering of NBA players

The computer grouped the data based on a comparison of four attributes: points, rebounds, blocks, and assists. Then, in multidimensional space, players with the lowest total distance from each other were grouped to form a label. This clustering algorithm is why LeBron James and Kevin Durant group together; they have similar metrics. Additionally, Steph Curry and Chris Paul are alike since they score many points and give out many assists.

A common dilemma with K-means clustering is how to select the correct number of clusters. This part of the problem is also an art and science problem as there isn't necessarily a perfect answer. One solution is to use a framework to create elbow plots for you, like Yellowbrick for sklearn.

Another MLOps-style solution is to let the MLOps platform, say AWS Sagemaker, do the K-means cluster assignment through automated hyperparameter tuning.

Finally, with reinforcement learning, an "agent" explores an environment to learn how to perform tasks. For example, consider a pet or a small child. They know how to interact with the world by exploring their environment. A more concrete example is the AWS DeepRacer system that allows you to train a model car to drive around a track, as shown in Figure 2-16.

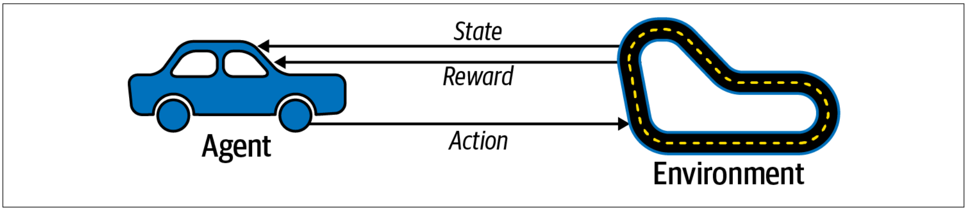


Figure 2-16. AWS DeepRacer

The agent, which is the car, interacts with the track, which is the environment. The vehicle moves through each section of the track, and the platform stores data about where it is on the track. A reward function decides how the agent interacts on each run through the track. Randomness plays a huge role in training this type of model, so different reward function strategies could yield different results.

The following is an example of a reward function in Python for AWS DeepRacer that rewards following the centerline:

```
def reward_function(params):
    '''
    Example of rewarding the agent for following the centerline
    '''

    # Read input parameters
    track_width = params['track_width']
    distance_from_center = params['distance_from_center']

    # Calculate 3 markers that are at varying distances away from the centerline
    marker_1 = 0.1 * track_width
    marker_2 = 0.25 * track_width
    marker_3 = 0.5 * track_width

    # Give higher reward if the car is closer to centerline and vice versa
    if distance_from_center <= marker_1:
        reward = 1.0
    elif distance_from_center <= marker_2:
        reward = 0.5
    elif distance_from_center <= marker_3:
        reward = 0.1
    else:
        reward = 1e-3  # likely crashed/ close to off track

    return float(reward)
```

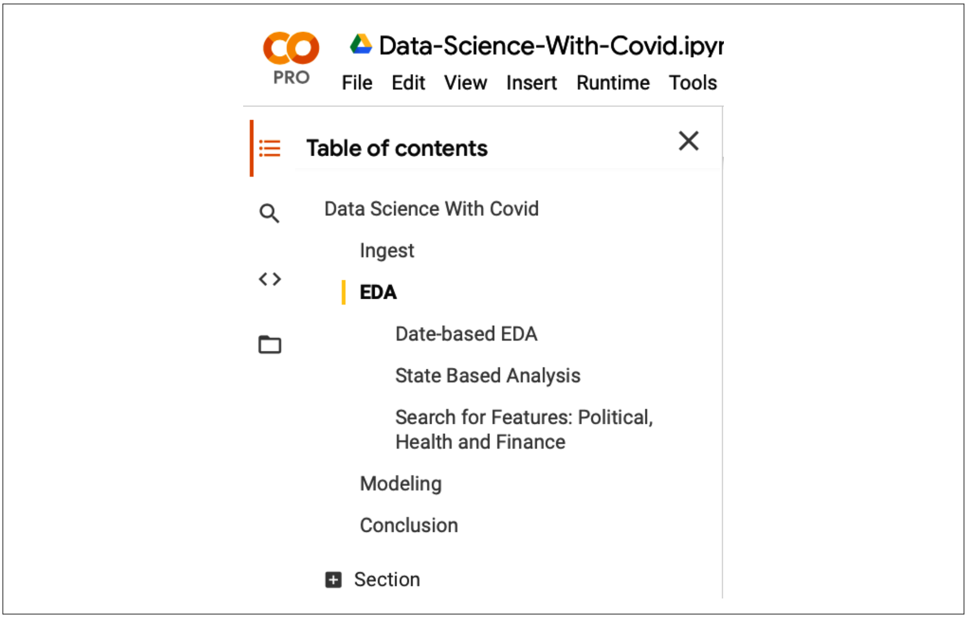
Here is a different reward function that rewards the agent for staying inside the two borders of the track. This approach is similar to the previous reward function, yet it could yield dramatically different results:

```
def reward_function(params):
    '''
    Example of rewarding the agent for staying inside the two borders of the
    track
    '''

    # Read input parameters
    all_wheels_on_track = params['all_wheels_on_track']
    distance_from_center = params['distance_from_center']
    track_width = params['track_width']

    # Give a very low reward by default
    reward = 1e-3

    # Give a high reward if no wheels go off the track and
    # the agent is somewhere in between the track borders
    if all_wheels_on_track and (0.5*track_width - distance_from_center) >= 0.05:
        reward = 1.0

    # Always return a float value
    return float(reward)
```

Doing machine learning in production requires the foundational knowledge covered in this chapter, i.e., knowing which approach to use. For example, discovering labels through unsupervised machine learning can be invaluable to determining who are the best paying customers. Similarly, predicting the number of units that will sell next quarter can be accomplished through a supervised machine learning approach that takes the historical data and creates a prediction. So next, let's dive into the basics of data science.

Doing Data Science

Another foundational skill to master is "the data science way." I recommend creating the following formulaic structure in a notebook in classes I teach: *Ingest*, *EDA*, and *Modeling* and *Conclusion*. This structure allows anyone in a team to quickly toggle through different project sections to get a feel for it. In addition, for a model deployed to production, it is beneficial to have a notebook checked in alongside the code that deploys the model to serve as a *README* for thinking behind the project. You can see an example of this in Figure 2-17.



Data Science With Covid

Ingest

EDA

Date-based EDA

State Based Analysis

Search for Features: Political, Health and Finance

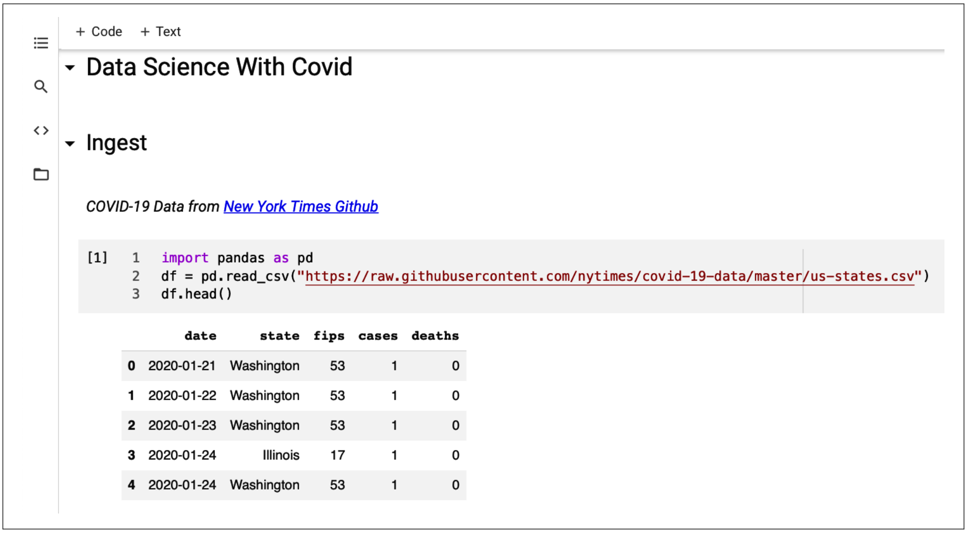
Modeling

Conclusion

Figure 2-17. Colab notebook

You can see an example of this structure in the Colab notebook Data Science with Covid.

This clean breakdown of parts of the notebook means that each section can be a "chapter" in writing a data science book. The Ingest section benefits from data sources that load via a web request, i.e., feed directly to Pandas. The notebook data sources can be replicated by others using this approach, as shown in Figure 2-18.



Data Science With Covid

Ingest

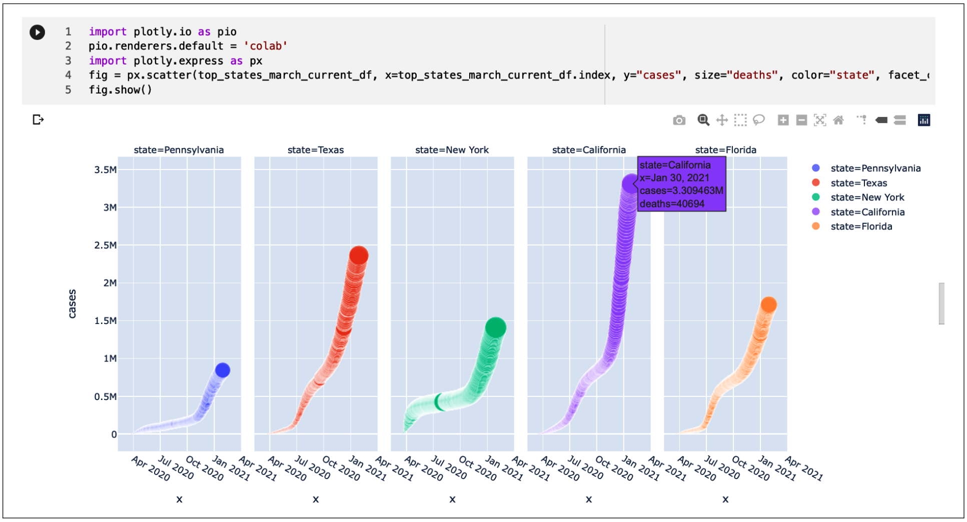
COVID-19 Data from New York Times Github

```
import pandas as pd
df = pd.read_csv("https://raw.githubusercontent.com/nytimes/covid-19-data/master/us-states.csv")
df.head()
```

	date	state	fips	cases	deaths
0	2020-01-21	Washington	53	1	0
1	2020-01-22	Washington	53	1	0
2	2020-01-23	Washington	53	1	0
3	2020-01-24	Illinois	17	1	0
4	2020-01-24	Washington	53	1	0

Figure 2-18. Colab notebook structure

The EDA section is for the exploration of ideas. What is going on with the data? This is the opportunity to find out, as the top Covid states show in the following chart using Plotly in Figure 2-19.



```
import plotly.io as pio
pio.renderers.default = 'colab'
import plotly.express as px
fig = px.scatter(top_states_march_current_df, x=top_states_march_current_df.index, y="cases", size="deaths", color="state", facet_
fig.show()
```

Figure 2-19. Colab notebook EDA

The Modeling section is where the model lives. Later, this repeatability can be critical since the MLOps pipeline may need to reference how the model creation occurred. For example, you can see an excellent example of a serialized sklearn model in this

Boston Housing Pickle Colab notebook. Note that I test out how this model will eventually work in an API or cloud-based system, like this Flask ML deployment project.

The Conclusion section should be a summary for a business leader making the decision. Finally, check your project into GitHub to build your MLOps portfolio. The rigor in adding this additional documentation pays off as an ML project matures. Operations teams, in particular, may find it very valuable to understand the original thinking for why a model is in production and decide to remove the model from production since it no longer makes sense.

Next, let's discuss building an MLOps pipeline step by step.

Build an MLOps Pipeline from Zero

Let's put everything in this chapter together, and let's dive into Deploy Flask Machine Learning Application on Azure App Services. Notice in Figure 2-20 that GitHub events trigger a build from the Azure Pipelines build process, which then deploys the changes to a serverless platform. The names are different on other cloud platforms, but conceptionally things are very similar in both AWS and GCP.

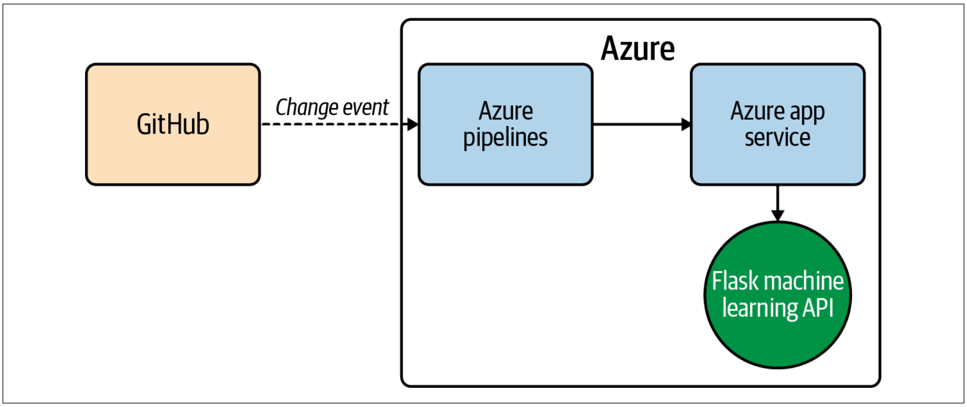


Figure 2-20. MLOps overview

To run it locally, follow these steps:

1. Create virtual environment and source:

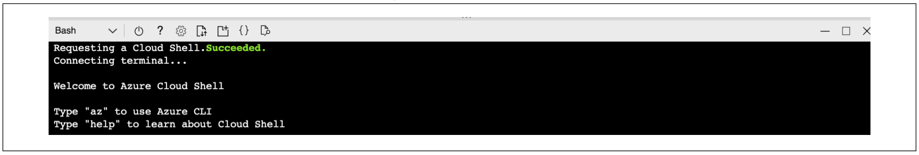
```
python3 -m venv ~/.flask-ml-azure
source ~/.flask-ml-azure/bin/activate
```

2. Run `make install`.

3. Run `python app.py`.
4. In a separate shell, run `./make_prediction.sh`.

To run it in Azure Pipelines (refer to Azure Official Documentation guide throughout):

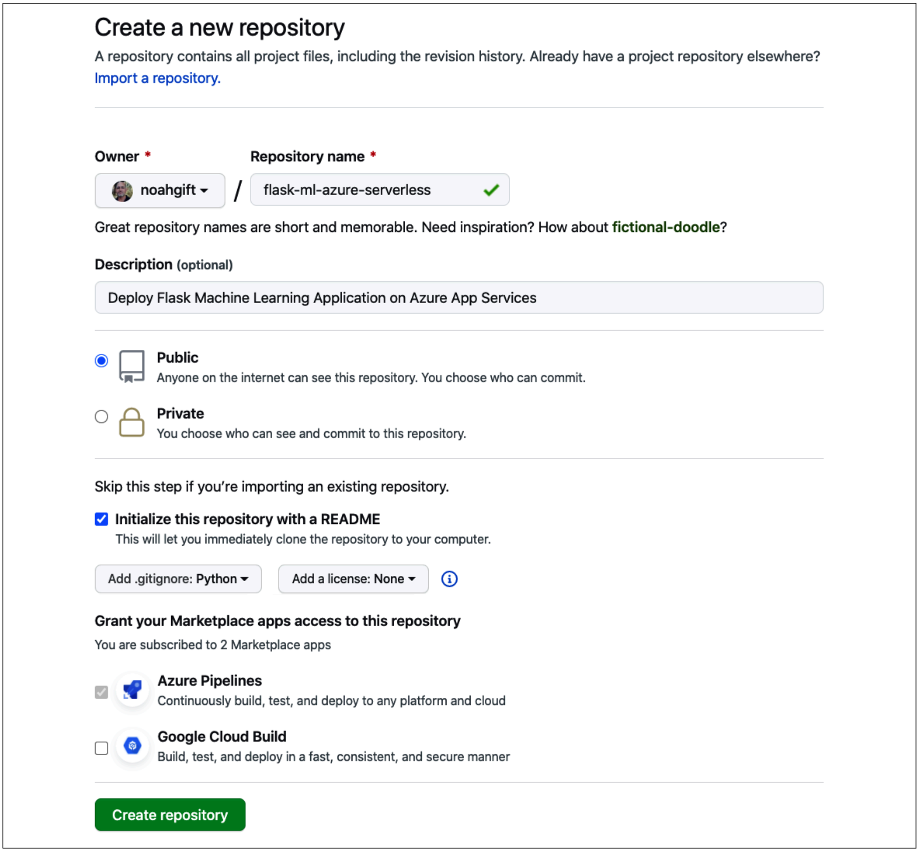
1. Launch Azure Shell as shown in Figure 2-21.



```
Requesting a Cloud Shell.Succeeded.
Connecting terminal...

Welcome to Azure Cloud Shell

Type "az" to use Azure CLI
Type "help" to learn about Cloud Shell
```

Figure 2-21. Launch Azure Cloud Shell

2. Create a GitHub repo with Azure Pipelines enabled (which could be a fork of this repo) as shown in Figure 2-22.



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Owner * noahgift / Repository name * flask-ml-azure-serverless ✓

Great repository names are short and memorable. Need inspiration? How about fictional-doodle?

Description (optional): Deploy Flask Machine Learning Application on Azure App Services

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

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

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

☑ Initialize this repository with a README
This will let you immediately clone the repository to your computer.

Add .gitignore: Python ▾ Add a license: None ▾

Grant your Marketplace apps access to this repository
You are subscribed to 2 Marketplace apps

☑ Azure Pipelines
Continuously build, test, and deploy to any platform and cloud

□ Google Cloud Build
Build, test, and deploy in a fast, consistent, and secure manner

Figure 2-22. Create GitHub repo with Azure Pipelines

3. Clone the repo into Azure Cloud Shell.

   If you need more information on how to setup SSH keys, you can follow this YouTube video guide on how to setup SSH keys and configure cloud shell environment.

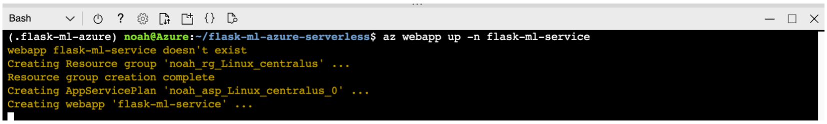
4. Create virtual environment and source:

   ```
   python3 -m venv ~/.flask-ml-azure
   source ~/.flask-ml-azure/bin/activate
   ```

5. Run `make install`.

6. Create an app service and initially deploy your app in Cloud Shell, as shown in Figure 2-23.

   ```
   az webapp up -n <your-appservice>
   ```

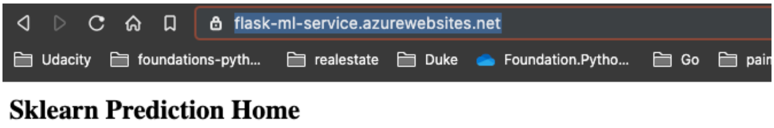


```
(.flask-ml-azure) noah@Azure:~/flask-ml-azure-serverless$ az webapp up -n flask-ml-service
webapp flask-ml-service doesn't exist
Creating Resource group 'noah_rg_Linux_centralus' ...
Resource group creation complete
Creating AppServicePlan 'noah_asp_Linux_centralus_0' ...
Creating webapp 'flask-ml-service' ...
```

Figure 2-23. Flask ML service

7. Verify the deployed application works by browsing to the deployed url: `https://<your-appservice>.azurewebsites.net/`.

   You will see the output as shown in Figure 2-24.

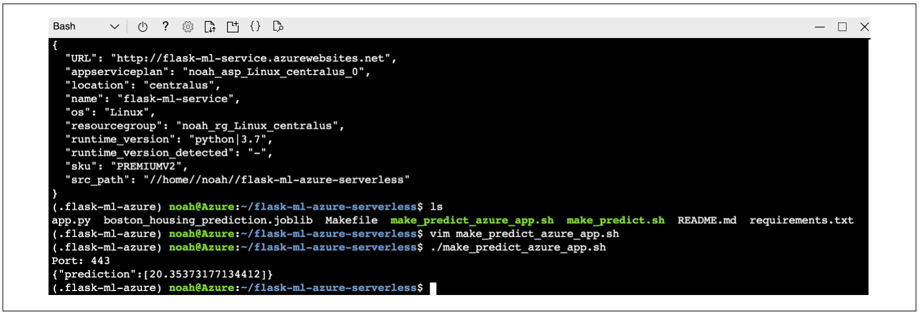


Sklearn Prediction Home

Figure 2-24. Flask deployed app

8. Verify machine learning predictions work, as shown in Figure 2-25.

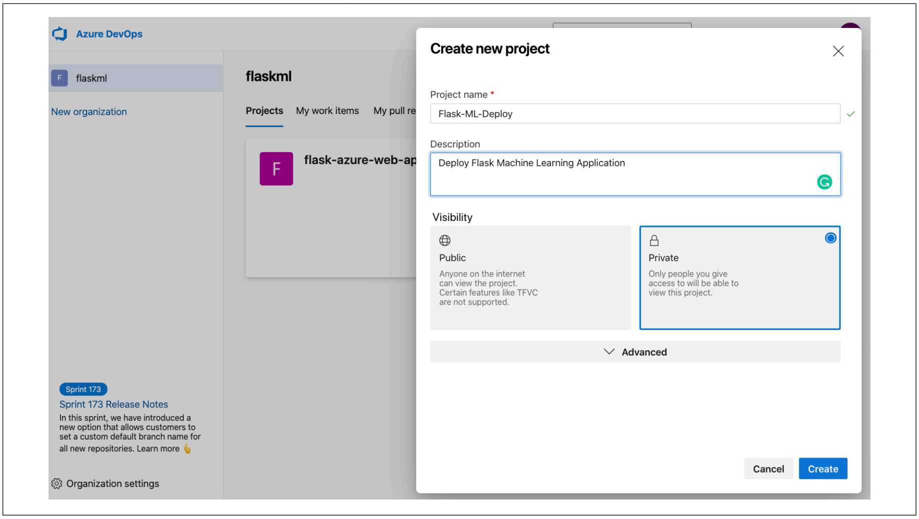
   Change the line in `make_predict_azure_app.sh` to match the deployed prediction `-X POST https://<yourappname>.azurewebsites.net:$PORT/predict`.



```
{
  "URL": "http://flask-ml-service.azurewebsites.net",
  "appserviceplan": "noah_asp_Linux_centralus_0",
  "location": "centralus",
  "name": "flask-ml-service",
  "os": "Linux",
  "resourcegroup": "noah_rg_Linux_centralus",
  "runtime_version": "python|3.7",
  "runtime_version_detected": "-",
  "sku": "PREMIUMV2",
  "src_path": "//home//noah//flask-ml-azure-serverless"
}
(.flask-ml-azure) noah@Azure:~/flask-ml-azure-serverless$ ls
app.py  boston_housing_prediction.joblib  Makefile  make_predict_azure_app.sh  make_predict.sh  README.md  requirements.txt
(.flask-ml-azure) noah@Azure:~/flask-ml-azure-serverless$ vim make_predict_azure_app.sh
(.flask-ml-azure) noah@Azure:~/flask-ml-azure-serverless$ ./make_predict_azure_app.sh
Port: 443
{"prediction":[20.35373177134412]}
(.flask-ml-azure) noah@Azure:~/flask-ml-azure-serverless$
```

Figure 2-25. Successful prediction

9. Create an Azure DevOps project and connect to Azure (as official documentation describes), as shown in Figure 2-26.



Azure DevOps
flaskml
New organization
flaskml
Projects
My work items
My pull re
F
flask-azure-web-ap
Sprint 173
Sprint 173 Release Notes
In this sprint, we have introduced a new option that allows customers to set a custom default branch name for all new repositories. Learn more
Organization settings
Create new project
Public
Anyone on the internet can view the project. Certain features like TFVC are not supported.
Only people you give access to will be able to view this project.
Advanced
Cancel
Create

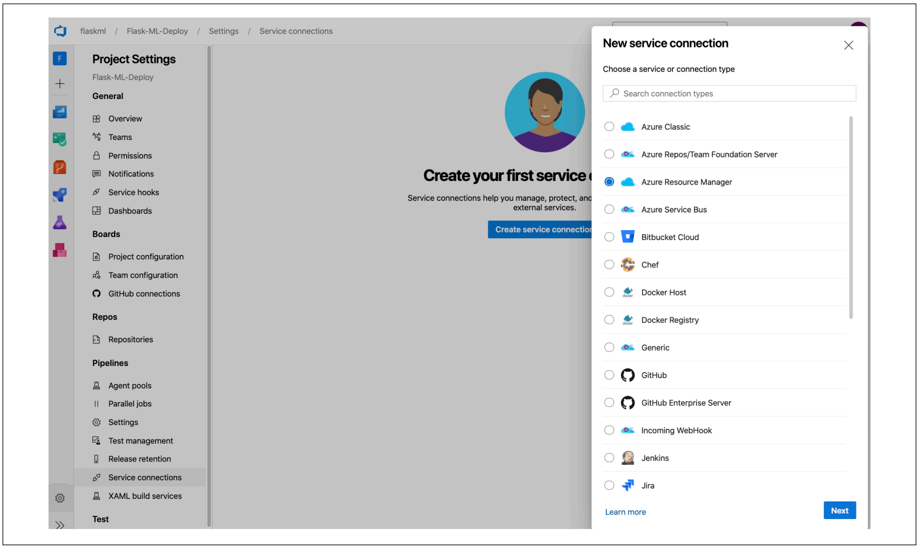
Project name *
Flask-ML-Deploy

Description
Deploy Flask Machine Learning Application

Visibility
○ Public
● Private

Figure 2-26. Azure DevOps connection

10. Connect to Azure Resource Manager as shown in Figure 2-27.



flaskml / Flask-ML-Deploy / Settings / Service connections

Project Settings
Flask-ML-Deploy

General
- Overview
- Teams
- Permissions
- Notifications
- Service hooks
- Dashboards

Boards
- Project configuration
- Team configuration
- GitHub connections

Repos
- Repositories

Pipelines
- Agent pools
- Parallel jobs
- Settings
- Test management
- Release retention
- Service connections
- XAML build services

Test

Create your first service c

Service connections help you manage, protect, and
external services.

Create service connectio

New service connection

Choose a service or connection type

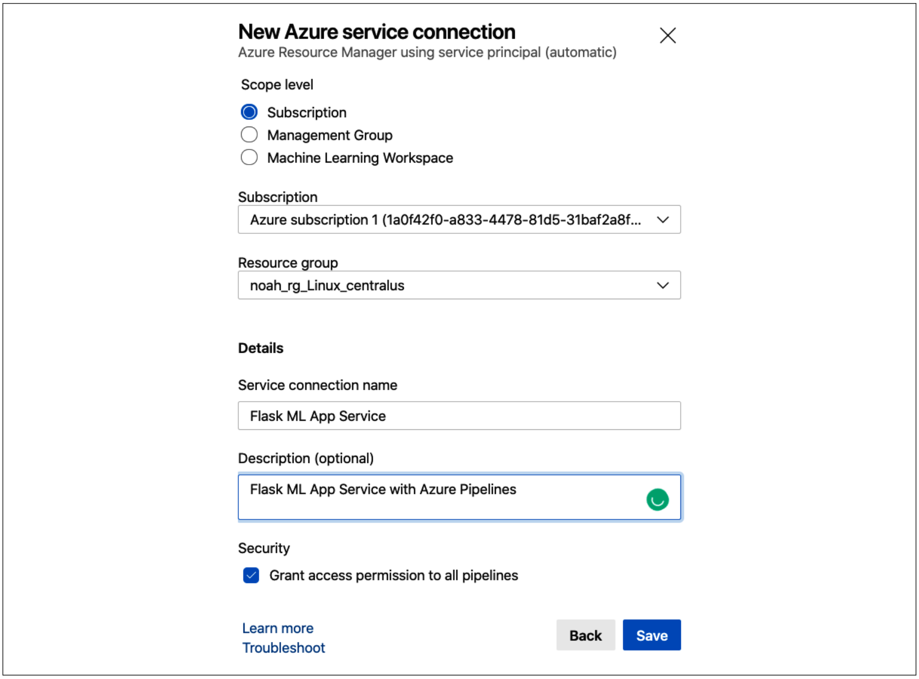
Search connection types

○ Azure Classic
○ Azure Repos/Team Foundation Server
● Azure Resource Manager
○ Azure Service Bus
○ Bitbucket Cloud
○ Chef
○ Docker Host
○ Docker Registry
○ Generic
○ GitHub
○ GitHub Enterprise Server
○ Incoming WebHook
○ Jenkins
○ Jira

Learn more   Next

Figure 2-27. Service connector

11. Configure the connection to the previously deployed resource group as shown in Figure 2-28.



New Azure service connection
Azure Resource Manager using service principal (automatic)

Scope level
● Subscription
○ Management Group
○ Machine Learning Workspace

Subscription
Azure subscription 1 (1a0f42f0-a833-4478-81d5-31baf2a8f...

Resource group
noah_rg_Linux_centralus

Details

Service connection name
Flask ML App Service

Description (optional)
Flask ML App Service with Azure Pipelines

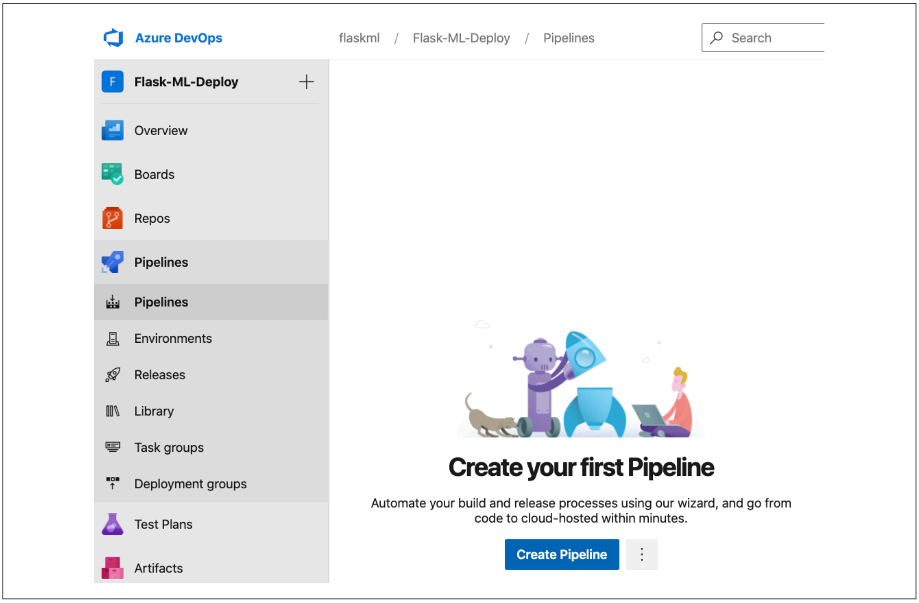
Security
☑ Grant access permission to all pipelines

Learn more
Troubleshoot

Back   Save

Figure 2-28. New service connection

12. Create a new Python pipeline with GitHub integration, as shown in Figure 2-29.


Azure DevOps   flaskml / Flask-ML-Deploy / Pipelines   Search

Flask-ML-Deploy +

- Overview
- Boards
- Repos
- Pipelines
- Pipelines
- Environments
- Releases
- Library
- Task groups
- Deployment groups
- Test Plans
- Artifacts

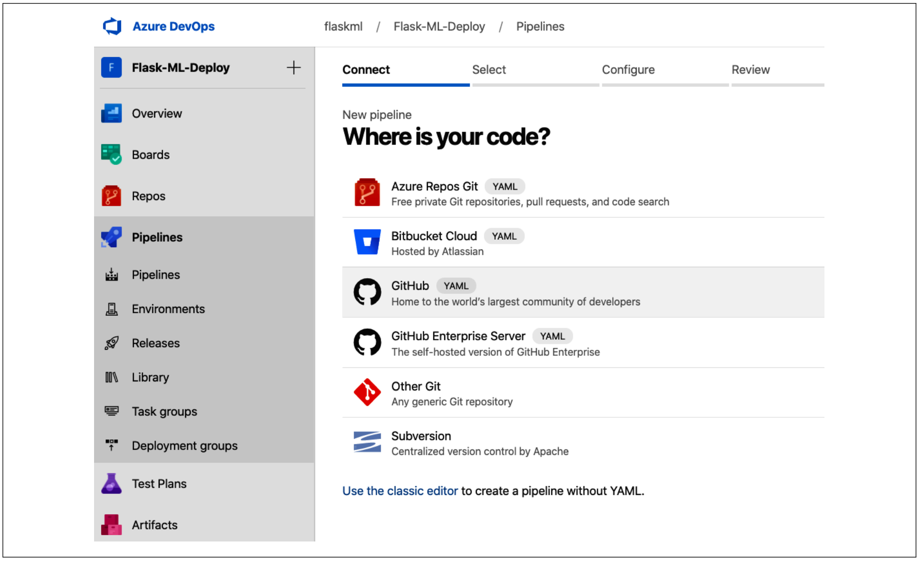
Create your first Pipeline

Automate your build and release processes using our wizard, and go from code to cloud-hosted within minutes.

Create Pipeline ⋮

Figure 2-29. New Pipeline

Finally, set up the GitHub integration as shown in Figure 2-30.


Azure DevOps   flaskml / Flask-ML-Deploy / Pipelines

Flask-ML-Deploy +

- Overview
- Boards
- Repos
- Pipelines
- Pipelines
- Environments
- Releases
- Library
- Task groups
- Deployment groups
- Test Plans
- Artifacts

Connect   Select   Configure   Review

New pipeline

Where is your code?

Azure Repos Git YAML
Free private Git repositories, pull requests, and code search

Bitbucket Cloud YAML
Hosted by Atlassian

GitHub YAML
Home to the world's largest community of developers

GitHub Enterprise Server YAML
The self-hosted version of GitHub Enterprise

Other Git
Any generic Git repository

Subversion
Centralized version control by Apache

Use the classic editor to create a pipeline without YAML.

Figure 2-30. GitHub integration

This process will create a YAML file that looks roughly like the YAML output shown in the following code. Refer to the official Azure Pipeline YAML documentation for more information about it. This is the first part of the machine generated file:

```
# Python to Linux Web App on Azure
# Build your Python project and deploy it to Azure as a Linux Web App.
# Change python version to one thats appropriate for your application.
# https://docs.microsoft.com/azure/devops/pipelines/languages/python

trigger:
- master

variables:
  # Azure Resource Manager connection created during pipeline creation
  azureServiceConnectionId: 'df9170e4-12ed-498f-93e9-79c1e9b9bd59'

  # Web app name
  webAppName: 'flask-ml-service'

  # Agent VM image name
  vmImageName: 'ubuntu-latest'

  # Environment name
  environmentName: 'flask-ml-service'

  # Project root folder. Point to the folder containing manage.py file.
  projectRoot: $(System.DefaultWorkingDirectory)

  # Python version: 3.7
  pythonVersion: '3.7'

stages:
- stage: Build
  displayName: Build stage
  jobs:
  - job: BuildJob
    pool:
      vmImage: $(vmImageName)
    steps:
    - task: UsePythonVersion@0
      inputs:
        versionSpec: '$(pythonVersion)'
      displayName: 'Use Python $(pythonVersion)'

    - script: |
        python -m venv antenv
        source antenv/bin/activate
        python -m pip install --upgrade pip
        pip install setup
        pip install -r requirements.txt
      workingDirectory: $(projectRoot)
```

13. Verify continuous delivery of Azure Pipelines by changing *app.py*.

    You can watch this YouTube Walkthrough of this process.

14. Add a lint step (this gates your code against syntax failure):

```
- script: |
    python -m venv antenv
    source antenv/bin/activate
    make install
    make lint
  workingDirectory: $(projectRoot)
  displayName: 'Run lint tests'
```

For a complete walkthrough of the code, you can watch the following YouTube walkthrough of this MLOps deployment process.

Conclusion

This chapter aimed to give you the foundational knowledge necessary to deploy machine learning into production, i.e., MLOps. One of the challenges of MLOps is how multidisciplinary the field is. When dealing with something inherently complex, a good approach starts small and gets the most basic solution working, then iterates from there.

It is also essential to be aware of foundational skills for an organization wishing to do MLOps. In particular, this means a team must know the basics of cloud computing, including the Linux terminal and how to navigate it. Likewise, a firm understanding of DevOps—i.e., how to set up and use CI/CD—is a required component to do MLOps. This final exercise is an excellent test of your skills before diving into more nuanced topics later in the book and pulls together all of these foundational components into a minimalist MLOps-style project.

In the next chapter, we'll dive into containers and edge devices. These are essential components of most MLOps platforms like AWS SageMaker or Azure ML Studio and build on the knowledge covered in this chapter.

Exercises

- Run a hello world Python GitHub project and check it out and run your tests on all three clouds: AWS, Azure, and GCP.
- Make a new Flask application that serves out a "hello world" type route using AWS Elastic Beanstalk you think other people would find helpful and put the code into a GitHub repo along with a screenshot of it serving out a request in the GitHub *README.md*. Then, create a continuous delivery process to deploy the Flask application using AWS CodeBuild.
- Fork this repository that contains a Flask machine learning application and deploy it with continuous delivery on AWS using Elastic Beanstalk and Code Pipeline.
- Fork this repository that contains a Flask machine learning application and deploy it with continuous delivery on GCP using Google App Engine and Cloud Build or Cloud Run and Cloud Build.
- Fork this repository that contains a Flask machine learning application and deploy it with continuous delivery on Azure using Azure App Services and Azure DevOps Pipelines.
- Use the Traveling Salesman code example and port it to work with coordinates you grab from an API, say all of the best restaurants in a city you want to visit. You will never think about vacation the same again.
- Using the TensorFlow Playground, experiment with changing the hyperparameters across different datasets as well as problem types. Can you identify optimal configurations of hidden layers, learning rate, and regularization rate for different datasets?

Critical Thinking Discussion Questions

- A company specializing in GPU databases has a key technical member advocating they *stop* using the cloud because it would be much more practical to buy their GPU hardware since they run it 24/7. This step would also allow them to get access to specialized GPUs much more quickly than they are available. On the other hand, another critical technical member who has *all* of the AWS certifications has promised to get him fired if he dares to try. He claims that they have already invested too much into AWS. Argue for or against this proposal.
- A "Red Hat Certified Engineer" has built one of the most successful data centers in the Southeast for a company with only 100 employees. Even though the company is an e-commerce company and not a cloud company, he claims this gives the company a huge advantage.

On the other hand, a “Google Certified Architect” and “Duke Data Science Masters” graduate claims the company is in a risky position by using a data center they own. They point out that the company keeps losing data center engineers for Google and has no disaster recovery plan or fault tolerance. Argue for or against this proposal.

- What are the key technical differences between AWS Lambda and AWS Elastic Beanstalk including the pros and cons of each solution?
- Why would a managed file service like EFS on AWS or Google Filestore be helpful in a real-world MLOps workflow in corporate America?
- Kaizen starts with a simple question: can we do better? If so, what should we do to get better this week or today? Finally, how can we apply Kaizen to our machine learning projects?