Chapter 5. AutoML and KaizenML

By Noah Gift

> *Held down too hard by rules, partial thoughts cannot blossom. Rules without ideas is prison. Ideas without rules is chaos. Bonsai teaches us balance. Balancing rules against innovation is a pervasive problem in all of life. I once saw a play entitled "The Game of Life." The message was that one is often asked to play the game for high stakes before anybody has explained the rules. Moreover, it's not so easy to tell if you are winning. It often seems that beginners (and young people generally) need rules or broad theories for guidance. Then, as experience accumulates, the many exceptions and variations gradually invalidate the rules at the same time that the rules become less needed. A great advantage of bonsai over Life is that one can learn from fatal mistakes.*
>
> —Dr. Joseph Bogen

It is an exciting time to be involved in build machine learning systems. Machine learning, i.e., learning from data, has a clear value to humanity in solving problems from autonomous vehicles to more effective cancer screening and treatment. At the same time, automation plays a critical role in enabling this advancement in the automation of model creation, AutoML, and the rest of the tasks surrounding machine learning, something I call KaizenML.

While AutoML is focused strictly on creating a model from clean data, KaizenML is about automating *everything* about the machine learning process and improving it. Let's dive into both topics starting with the reason why AutoML is so essential.

NOTE

Experts in machine learning, like Andrew Ng, now acknowledge that a data-centric approach has merits over a model-centric process. Another way of stating this is that Kaizen, i.e., continuous improvement of the entire system from the data to the software, to the model, to the feedback loop from the customer, is essential. KaizenML, in my mind, means you are improving all aspects of a machine learning system: data quality, software quality, and model quality.

AutoML

The author Upton Sinclair famously said, "it's difficult to get a man to understand something when his salary depends on his not understanding it." An excellent example of the Upton Sinclair quote in action is the misinformation from social media documented in the Netflix documentary, *The Social Dilemma*. Suppose you work at a company that spreads misinformation at scale and gets paid very well. In that case, it is almost impossible to accept that you are an actor in that process, and your company does, in fact, profit handsomely from misinformation. It contributes to your excellent salary and lifestyle.

Similarly, I have come up with something I call the Automator's law. Once the conversation about automating a task begins, then eventually, the automation occurs. Some examples include replacing data centers with cloud computing and replacing telephone switchboard operators with machines. Many companies have held on to their data centers with "white knuckles," saying the cloud was the root of all evil in the world. Yet, eventually, they either switched to the cloud or are in the process of switching to the cloud.

It took almost 100 years, from around 1880 to 1980, to fully automate switching calls by hand to make a machine do them, but it happened. Machines are great at automating labor-intensive manual tasks. If your job involved switching telephone calls in 1970, you might have scoffed at the idea of automating what you did since you understood how difficult the task was. Today, with data science, it may be that we are switchboard operators furiously pushing hyperparameter values into Python func-

tions and sharing our results on Kaggle, unaware that all of this is in the process of being automated away.

In the book, *How We Know What Isn't So*, Thomas Gilovich points out self-handicapping strategies:

> *There really are two classes of self-handicapping strategies, real and feigned. "Real" self-handicapping involves placing visible obstacles to success in one's own path. The obstacles make one less likely to succeed, but they provide a ready excuse for failure. The student who neglects to study before an exam or the aspiring actor who drinks before an audition are good examples. Sometimes failure is all but guaranteed, but at least one will not be thought to be lacking in the relevant ability (or so it is hoped).*
>
> *"Feigned" self-handicapping, on the other hand, is in certain respects a less risky strategy, one in which the person merely claims that there were difficult obstacles in the path to success. This kind of self-handicapping consists simply of making excuses for possible bad performance, either before or after the fact.*

When data science initiatives fail, it is easy to dive into either of these self-handicapping strategies. An example of this in data science could be by not using AutoML when it could help with certain aspects of a project; this is a "real" handicap to its success. However, one of the golden rules of software engineering is to use the best tools you can for the task at hand. The reason to use the best tools available is they reduce the complexity of the software developed. An excellent example of a "best in class" tool that reduces complexity is GitHub Actions because it is simple to create automated testing. Another example is an editor like Visual Studio Code because of its ability to perform code completion, syntax highlighting, and linting with minimal configuration. Both of these tools dramatically increase developer productivity by simplifying the process of creating software.

With data science, this mantra of "use the best tools available" needs evangelism. Alternatively, if a data science project fails, as they often do, a self-handicapping strategy could be to say the problem was too challeng-

ing. In either case, an embrace of automation, when appropriate, is the solution to self-handicapping.

Let's compare food to machine learning in Figure 5-1. Notice that food comes in many forms, from the flour you buy at the store to make your own pizza to the one you have delivered to your house. Just because one is much more complex than another (i.e., making a pizza from scratch versus ordering a ready-made hot pizza) it doesn't mean that the home delivery option isn't also considered food. Difficulty, or lack thereof, does not equate with completeness or realness.

Similarly, not accepting reality doesn't mean that it's not happening. Another way of describing denying reality is to call it "magical thinking." Many magical thinkers said at the start of the COVID-19 pandemic, "This is just like the flu," as a way to reassure themselves (and others) that the danger was not as bad as it appeared to be. The data in 2021 says something entirely different, though. COVID-19 in the USA approached about 75% of the deaths of all combined forms of heart disease, currently the leading cause of death in the USA. Similarly, Justin Fox in a Bloomberg article using the CDC data shows that this pandemic is multiple times more deadly for most age groups than influenza. See Figure 5-2.

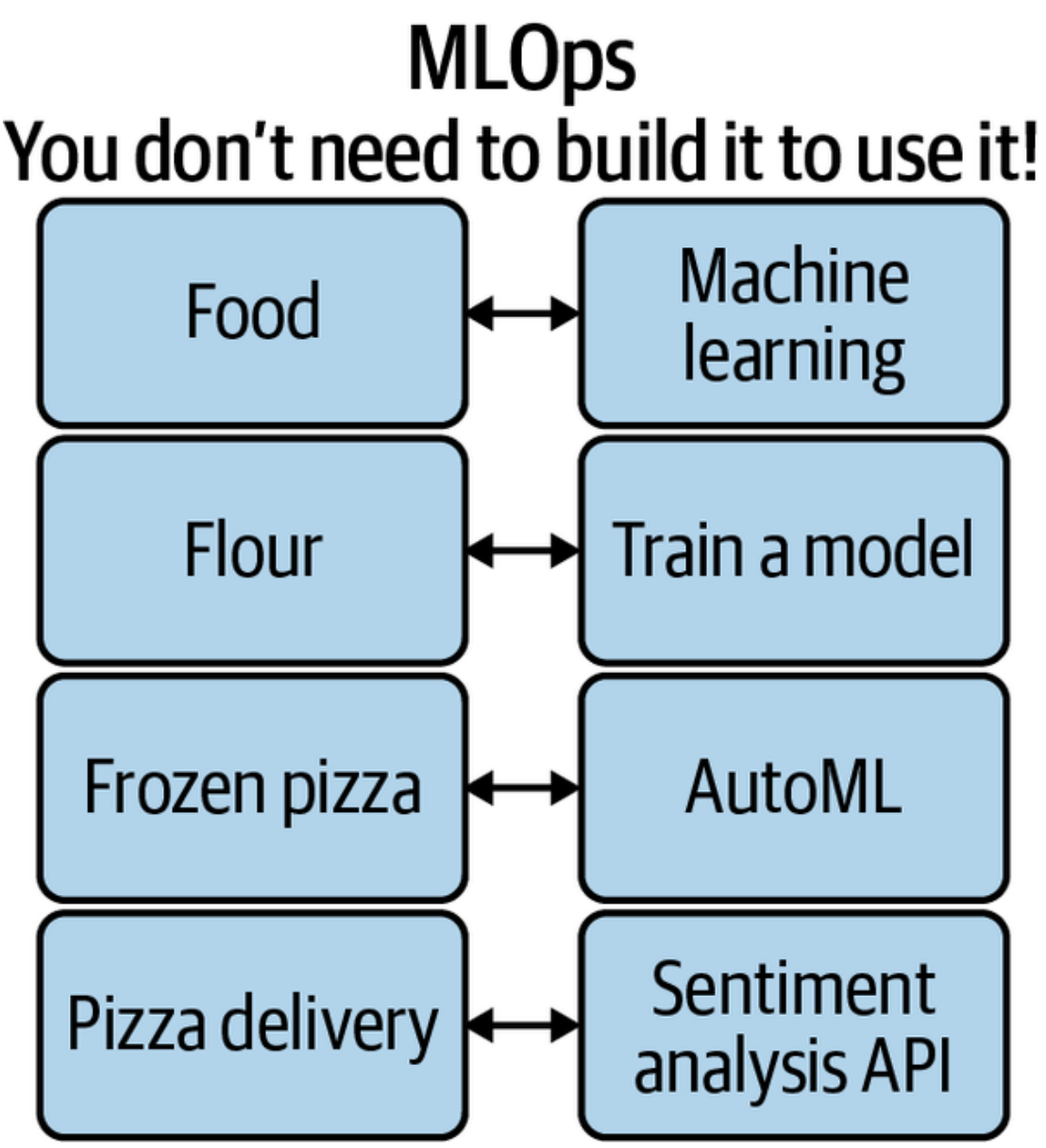

Figure 5-1. Food versus ML

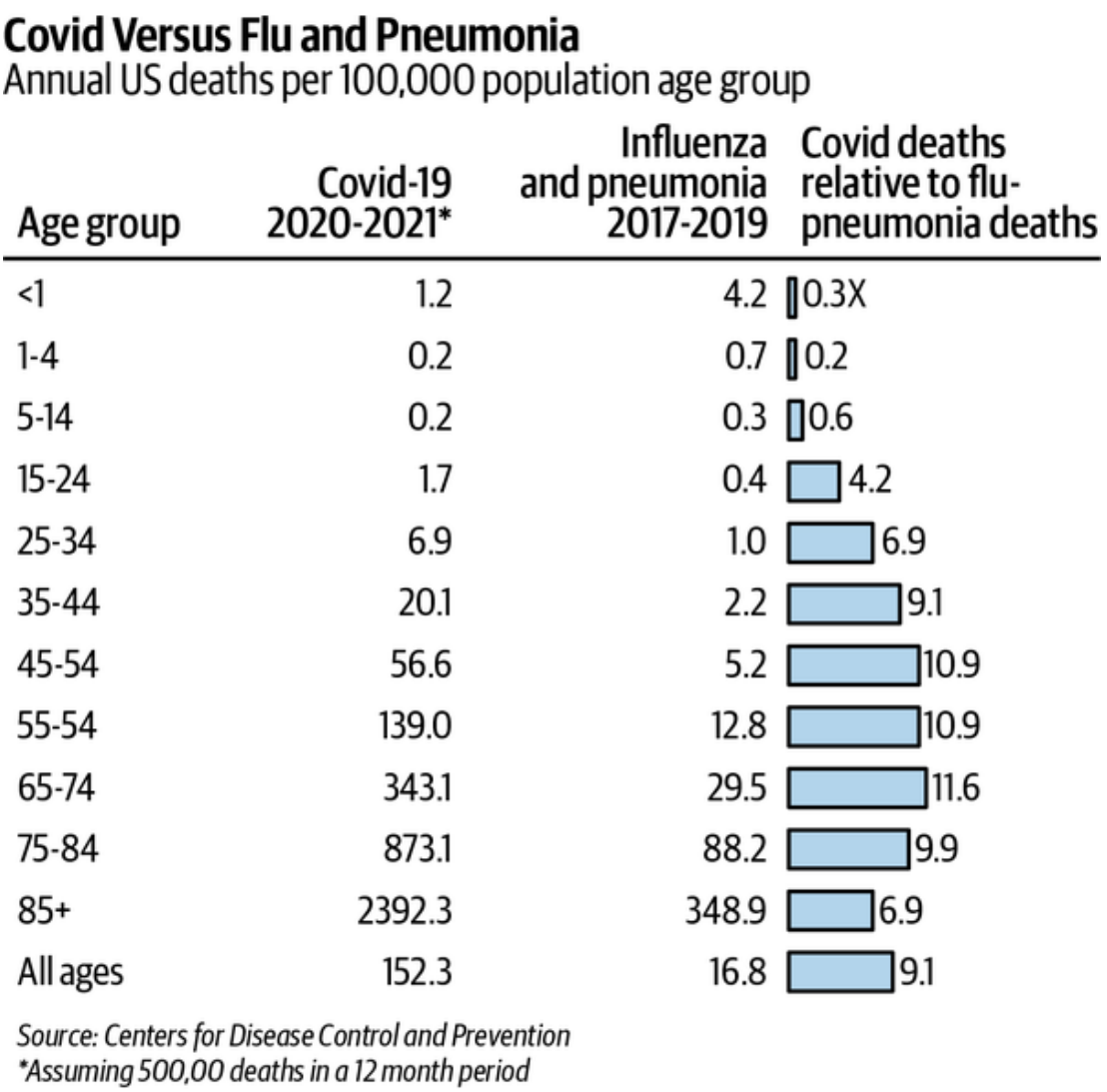

Figure 5-2. Covid versus the flu and pneumonia (source: Bloomberg News)

AutoML is an inflection point for data scientists because it shares similarities to other historical trends: automation and magical thinking. Anything that can automate will automate. Accepting this trend instead of fighting it will lead to massively more productive workflows in machine learning. AutoML, in particular, may be one of the most critical technologies to embrace to implement the MLOps philosophy fully.

Before the COVID-19 outbreak, a research scientist from UC Berkeley, Dr. Jennifer Doudna, and collaborator Dr. Emmanuelle Charpentier worked on the difficult task of researching gene editing. When the COVID-19 pandemic began, Dr. Doudna knew she needed to quickly convert this research into a groundbreaking way to accelerate the creation of a vaccine. As a result, she started work to “save the world.”

NOTE

How is COVID-19 drug discovery related to MLOps? A critical problem in data science is getting a research solution into production. Similarly, a fundamental problem with medical research is getting the discovery into the hands of a patient who benefits from it.

In *The Code Breaker: Jennifer Doudna, Gene Editing, and the Future of the Human Race* (Simon & Schuster Australia), Walter Isaacson describes how Dr. Doudna was "…now a strong believer that basic research should be combined with translational research, moving discoveries from bench to bedside…" The now Nobel Prize–winning scientist Dr. Jennifer Doudna is co-credited, along with Dr. Emmanuelle Charpentier, with creating the research that led to gene editing and commercial application of these CRISPR mechanisms.

One of her rivals, Dr. Feng Zhang, who eventually worked on a competing vaccine, Moderna, mentioned that the UC Berkeley lab wasn't working on applying this in human cells. His critique is that his lab was working on using the CRISPR research by targeting human cells, while Dr. Doudna was focused strictly on the research.

This critique is the heart of a patent dispute about who gets to claim credit for these discoveries, i.e., how much work was research and how much was the application of the research? Doesn't this sound a bit similar to the data science versus software engineering disputes? Ultimately, Dr. Doudna did, in fact, "get it to production" in the form of the Pfizer vaccine. I recently got this vaccine, and like many people, I am thrilled it made it to production.

What else could we collectively accomplish if we had the same sense of urgency as the scientists "operationalizing" the vaccine for COVID-19? When I was an engineering manager at startups, I liked to ask people hypothetical questions. Some variant of "What if you had to save the world on a deadline"? I like this question because it cuts to the heart of things quickly. It cuts to the nature of the problem because if the clock is ticking on saving millions of lives, you work only on the essential components of the problem.

There is an incredible documentary on Netflix entitled *World War II in Colour*. What is impressive about the documentary is that it shows the actual restored and colorized footage of tragic historical events. This really helps you imagine what it might have been like to be present during those events. Along those lines, imagine yourself in a situation where you need to solve a technical problem that would save the world. Of course, if you get it wrong, everyone you know will suffer a horrible fate. However, AutoML or any form of automation coupled with a sense of urgency in solving only the necessary components of a problem can lead to better outcomes globally: i.e., drug discovery and cancer detection.

This form of situational thinking adds a clarifying component to deciding how to solve a problem. Either what you are doing matters, or it doesn't. It is very similar to the way the scientists working on COVID-19 vaccines thought. Either what the scientists did led to a faster COVID-19 vaccine, or it didn't. As a result, every day wasted was a day that more people worldwide succumbed to the virus.

Similarly, I remember going to a fancy startup in the Bay Area around 2016 and remarking on how they received 30M funding from many top-name venture capital firms. The COO then privately told me how he was very concerned that they had no actual product or way to make money. They received even more money many years later, and I am still unsure what their actual product is.

Because this company cannot create revenue, it fundraises. If you cannot fundraise, then you must create revenue. Likewise, if you cannot put your model into production with machine learning, you continue to do "ML research," i.e., working away at tweaking hyperparameters on a Kaggle project. So a good question for Kaggle practitioners to ask is, "Are we sure we aren't just automating our job of tweaking hyperparameters by training Google's AutoML technology?"

We gravitate toward what we excel at doing. There are many beautiful things about focusing on what you do well, such as it leading to a successful career. Still, there are also times to challenge yourself to temporarily think situationally about solving a problem in the most urgent manner possible, much like Drs. Doudna and Zhang did with COVID-19. Does this

change the approach you use? For example, if I had four hours to train a model and put it into production to save the world, I would write as little code as possible and use off-the-shelf automation tools like Azure AutoML, Apple Create ML, Google AutoML, H20, or Ludwig. In that case, the follow-up question becomes, why am I writing *any* code, or at least writing the least amount of code possible for all machine learning engineering projects?

The world needs high-quality machine learning models in production, particularly because there are many urgent problems to solve: finding a cure for cancer, optimizing clean energy, improving drug discovery, and creating safer transportation. One way society can collectively do this is to automate what can be automated now and focus on finding ways to automate what cannot be automated today.

AutoML is the automation of the tasks related to training a model on clean data. Not all problems are so simple in the real world, though, and as a result *everything* related to machine learning needs automation. This gap is where KaizenML steps in. Kaizen, in Japanese, means continuous improvement. With KazienML, you continuously improve and automate as the central way you develop machine learning systems. Let's dive into that concept next.

MLOps Industrial Revolution

Many students and practitioners of machine learning find AutoML to be a polarizing topic. Data science is a behavior; AutoML is a technique—they are a small part of building an ML system, and they are complementary. AutoML is polarizing because data scientists assume it will replace their job, when in fact, AutoML is 5% of a gigantic process of automation and continuous improvement, i.e., MLOps/ML engineering/KaizenML, as described in Figure 5-3.

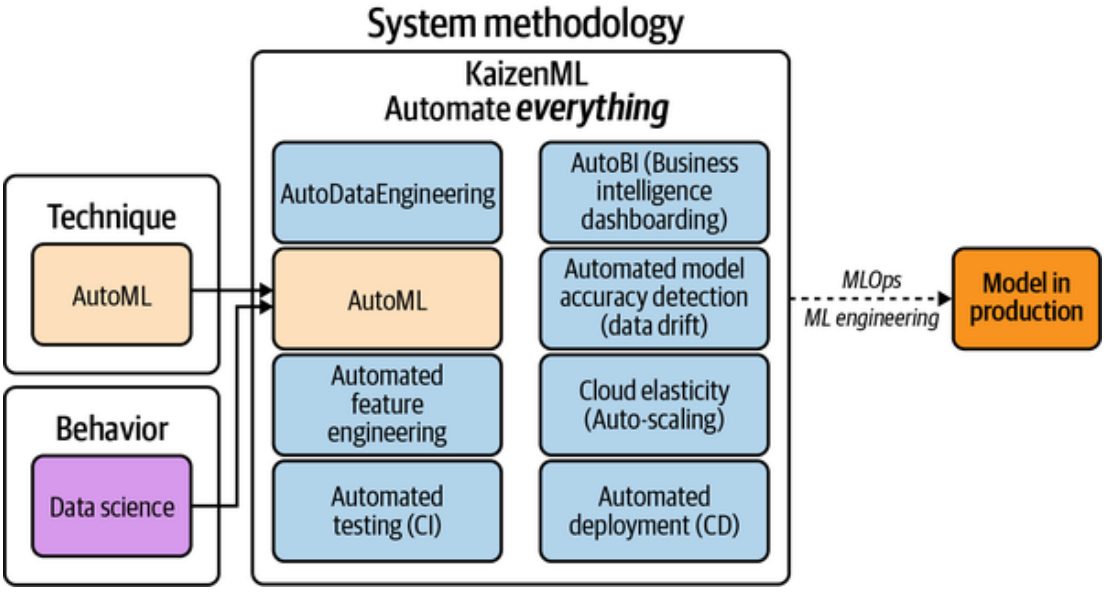

Figure 5-3. AutoML is a tiny part of KaizenML

The industrial revolution from 1760–1840 was a period of dramatic movement of human tasks to automated ones powered by steam and coal machines. This automation led to a rise in population, GDP, and quality of life. Later, around 1870, the second industrial revolution occurred, allowing for mass production and new electrical grid systems.

There is an excellent series on Disney+ called *Made in a Day*. The first episode shows how Tesla uses robots for car development phases. The robots screw in things, bolt things on, and weld parts together. When looking at this factory, I think about how humans are assisting the robots. Essentially, they feed the robots work that they cannot yet fully automate themselves.

Likewise, when looking at traditional data science workflows full of unique snowflake configurations, with humans "bolting on" hyperparameters, it makes me think of an early Ford assembly plant in the second industrial revolution. Eventually, manual human tasks get automated, and the first thing that is automated is the easiest to automate.

Another question people ask is whether many aspects of machine learning techniques are even necessary, like manually adjusting hyperparameters, i.e., picking the number of clusters. Imagine going to a Tesla factory full of advanced robotics and telling the automation engineers that humans can also weld parts together. This statement would be a non sequitur. Of course, we humans can perform tasks that machines do better

than us, but should we? Likewise, with many tedious and manual aspects of machine learning, machines do a better job.

What may occur soon in machine learning and artificial intelligence is that the technique is essentially commoditized. Instead, the automation itself and the ability to execute the automation is the key. The TV show about physical manufacturing has the name "Made in a Day" because cars or guitars manufacture in just one day! Many companies doing machine learning cannot build one software-based model in an entire year, though how could this possibly be the future process?

One possible scenario that I see happening soon is that at least 80% of data science manual training models are replaced with commoditized open source AutoML tools or downloaded pre-built models. This future could happen as both open source projects like Ludwig or commercial projects like Apple CreateML advance in sophistication. Software for training machine learning models could turn into something like the Linux kernel, free and ubiquitous.

If they are in their current form, data science could go bimodal; either you get paid $1M/year, or you're entry-level. Most competitive advantages go into traditional software engineering best practices: data/users, automation, execution, and solid product management and business practices. A data scientist could become a standard skill like accounting, writing, or critical thinking in other cases instead of a job title alone. You could call this the MLOps industrial revolution.

Figure 5-4 is an example of this in practice. Imagine Kaggle as a feedback loop that Google uses to make its AutoML tools much better. Why wouldn't they use the human data scientists' training models to make better AutoML services? In data science 1.0, humans are manually "clicking buttons" just like the switchboard operators of the past. Meanwhile, if they wanted, Google could use these humans to train their AutoML systems to do these manual data science tasks. In data science 2.0, which in many cases is already here, automated tools thoroughly train the previously trained models in Kaggle.

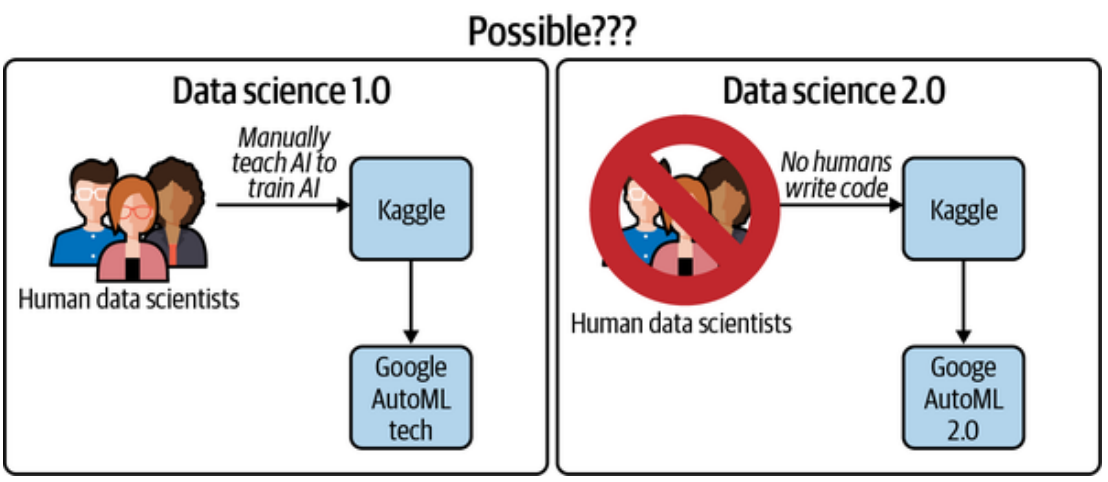

Figure 5-4. Kaggle automation

The MLOps industrial revolution is happening before our eyes as machines play an increasing role in machine learning and data science. What skills do you invest in if these changes are under way? Be world-class at automation and execution both technically and from a business perspective. Also, couple these capabilities with solid domain expertise. In the book *How Innovation Works: And Why It Flourishes in Freedom* (Harper), author Matt Ridley clearly explains how ideas are not the basis of innovation but the combination of the ideas into execution. Essentially, does it work or not, and will someone pay you for it?

Kaizen Versus KaizenML

One problem with talking about data science, AutoML, and MLOps (KaizenML) is that people often misunderstand what each one is. Data science is not a solution any more than statistics is a solution to a problem; it is behavioral. AutoML is just a technique, like continuous integration (CI); it automates trivial tasks. Subsequently, AutoML is not directly competing with data science; cruise control, or semi-autonomous driving for that matter, doesn't compete with a car driver. The driver still must control the vehicle and act as the central arbitrator of what happens. Likewise, even with extensive automation in machine learning, a human must make executive decisions about the bigger picture.

KaizenML/MLOps is a systems methodology that leads to models in production. Machine learning models running in production solving customer problems is the outcome of MLOps. In Figure 5-5, you can see a hypothetical MLOps industrial revolution that could occur in the future.

Data and the expertise in handling data effectively becomes a competitive advantage since it is a scarce resource. As AutoML technology progresses, it is possible many things data scientists do today go away. It is uncommon to find modern vehicles without a form of cruise control or with manual transmission. Likewise, it may be infrequent for data scientists to adjust hyperparameters in the future. What could happen then is that current data scientists turn into ML engineers or domain experts who "do data science" as part of their job.

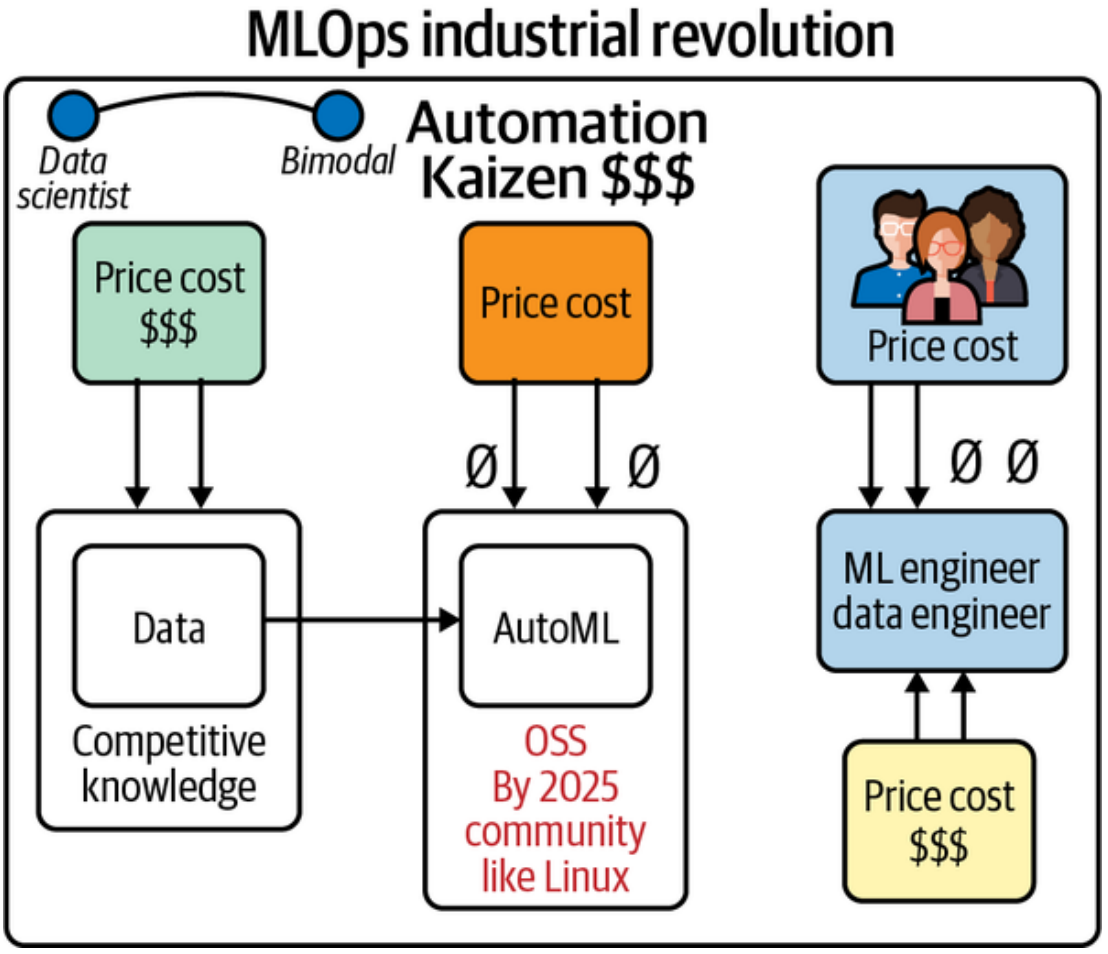

Figure 5-5. MLOps industrial revolution

One issue with talking only about AutoML versus data science is that it trivializes the more significant automation and continuous improvement issues. The automation of machine learning techniques is so polarizing that the core issue disappears: everything should be automated, not just the tedious aspects of ML-like hyperparameter tuning. Automation through continuous improvement allows data scientists, ML engineers, and entire organizations to focus on what matters, i.e., execution. As you can see illustrated in Figure 5-6, Kaizen is a Japanese term for continuous improvement. In post–World War II, Japan builds its automobile industry

around this concept. Essentially, if you find something broken or unoptimized, you fix it. Likewise, with KaizenML, every aspect of machine learning, from feature engineering to AutoML, is improved.

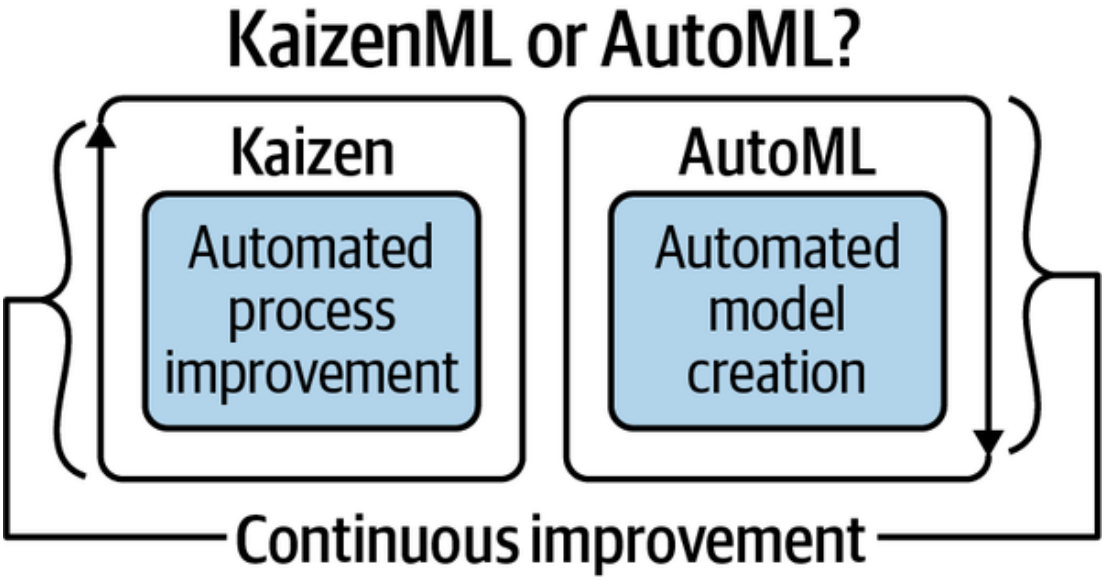

Figure 5-6. Kaizen or AutoML

Every human on earth should do data science and programming because these are forms of critical thinking. The recent pandemic was a big wake-up call about how important to an individual's life understanding data science is. Many people died because they didn't understand the data that showed COVID-19 was not, in fact, just like the flu; it was much, much more deadly. Likewise, stories abound about people refusing to get a vaccine because they incorrectly calculated the risk the vaccine posed to themselves versus the risk COVID-19 posed to themselves or vulnerable members of their community. Understanding data science can save your life, and therefore, anyone should have access to the tools data scientists have.

These tools are "human rights" that don't belong in the hands of an elite priesthood. It is a non sequitur to suggest "elite" people only can write simple programs, understand machine learning, or do data science. Automation will make data science and programming easy enough that every human being will do it, and in many cases, they can do so using even the existing automation available.

KaizenML/MLOps is about a narrow focus on solving problems with machine learning and software engineering (influenced by DevOps) to lead

to business value or improvement of the human condition, e.g., cure cancer.

Feature Stores

All complex software systems require automation and simplification of critical components. DevOps is about automating the testing and deployment of software. MLOps is about doing this and also improving the quality of both data and machine learning models. I have previously called these continuous improvements of both data and machine learning models KaizenML. One way to think about this is that DevOps + KaizenML = MLOps. KaizenML includes building Feature Stores, i.e., a registry of high-quality machine learning inputs and the ability to monitor data for drift and register and serve out ML models.

In Figure 5-7, note that in manual data science, everything is bespoke. As a result, the data is low quality, and it is hard to even get to the point where a working model goes into production and solves a problem. However, as more things get automated from data to features, via a Feature Store, to the actual serving out of the model in production, it leads to a better outcome.

Figure 5-7. Feature Stores as part of KaizenML

Heavily related to KaizenML, i.e., continuously improving machine learning, is the concept of a Feature Store. The Uber Engineering blog has a good breakdown of what problem a Feature Store solves. According to Uber, it does two things:

- Allows users to add features they built into a shared Feature Store.
- Once features are in the Feature Store, they are easy to use in training and prediction.

In Figure 5-8, you can see that data science is a behavior, but AutoML is a technique. AutoML could be only 5% of the entire problem solved by automation. The data itself needs automation through ETL job management. The Feature Store needs automation to improve the ML inputs. Finally, the deployment requires automation through automated deployment

(CD) and the native use of cloud elasticity (autoscaling). Everything requires automation with complex software systems, and Feature Stores are just one of many MLOps components needing continuous improvement, i.e., KaizenML.

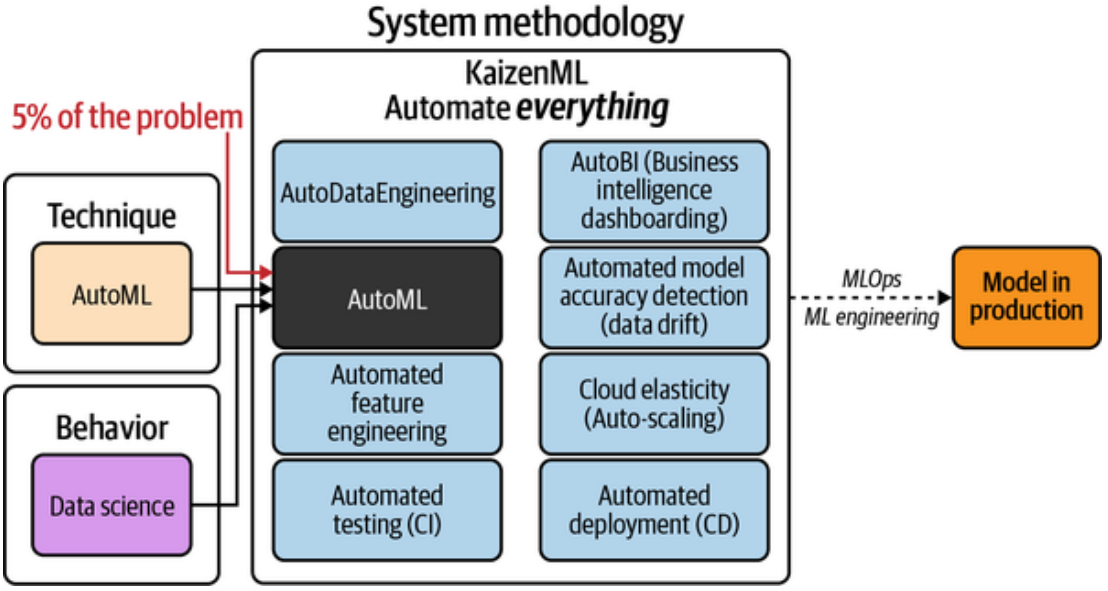

Figure 5-8. Feature Stores are part of a systems methodology of automation

There are many real-world use cases for Feature Stores. For example, Uber explains that it used 10,000 features in the Feature Store to accelerate machine projects and make AutoML solutions on top. In addition, platforms like Databricks have Feature Stores built into their big data system. For example, in Figure 5-9, you can see how raw data is the input that gets transformed into a more refined and specialized feature registry that can solve both batch and online problems.

In Figure 5-10, notice that there are both similarities and differences between a traditional data warehouse and an MLOps Feature Store. A data warehouse feeds into business intelligence systems at a high level, and a Feature Store provides inputs into an ML system. Machine learning data processing is repetitive, including normalizing data, cleaning data, and finding appropriate features that improve an ML model. Creating a Feature Store system is yet one more way of fully embracing automation of the entire process of machine learning from ideation to production.

Figure 5-9. Databricks feature store

Figure 5-10. Data warehouse versus feature store

Next, let's get out of the theory and practice technique using the Apple ML Ecosystem to build machine learning models. We will do this using its high-level AutoML framework, CreateML.

Apple's Ecosystem

Apple may seem like an unlikely candidate to enter the machine learning tools space until you look a bit deeper. Apple has a rich ecosystem around mobile development. According to Statista, the worldwide gross app revenue of the Apple App Store grew from 55.5 billion US Dollars in 2019 to 72.3 billion US Dollars in 2020. Apple benefits from developers creating products that sell in its app store.

I remember talking to a pretty dismissive professor about building "apps for machine learning," presumably because he gravitated toward complexity and discovery in doing research. In a sense, the software industry thinks in an opposite manner of a researcher at a university. Writing academic papers on machine learning is the opposite direction of operationalizing machine learning to "build apps." This ideological difference is the "idea" versus "execution" disconnect discussed earlier.

Apple wants you to build apps in its app store since it takes between 15% and 30% of every transaction. The better Apple makes the developer tools, the more applications live in the app store. There is an expression in business school, "Where do you build a Burger King? Next to McDonald's." This expression is a way of saying that you don't need to spend the money to research where to expand because the top competitor already did the work. You can piggyback on their expertise—likewise, practitioners of machine learning can piggyback on the research Apple has done. They see a future in high-level automated machine learning running on specialized hardware.

Similarly, why do many VC firms fund only companies that top VC firms already funded? Because then they don't need to do any work; they can profit from the expertise of a more knowledgeable firm. Similarly, Apple

has tremendous investments from a hardware perspective into on-device machine learning. In particular, Apple develops chips itself, like the A-series: A12-A14, shown in Figure 5-11, including CPUs, GPUs, and dedicated neural network hardware.

Figure 5-11. Apple's A14 chip

Further, newer chips include the Apple M1 architecture, which Apple uses on mobile devices, laptops, and desktop machines, as shown in Figure 5-12.

Figure 5-12. Apple's M1 chip

The development environment makes use of this technology through Apple's model format Core ML. There is also a Python package to convert models from third-party training libraries like TensorFlow and Keras.

Core ML is optimized for on-device performance and works in tandem with the Apple hardware. There are several non-obvious workflows to consider:

- Use Apple's Create ML framework to make AutoML solutions.
- Download a pretrained model and optionally convert it to the Core ML format. One location to download models from is tfhub.
- Train a model yourself by writing the code in another framework, then converting it to Core ML using coremltools.

Let's dig into Apple's AutoML.

Apple's AutoML: Create ML

One of the core innovations with Apple's ML platform is how it exposes power AutoML technology enclosed in an intuitive GUI. Apple Create ML lets you do the following:

- Create Core ML models
- Preview the model performance

- Train models on the Mac (taking advantage of their M1 chip stack)
- Use training control: i.e., pause, save and resume training
- Use eGPU (external GPUs)

Additionally, it tackles various domains from image, video, motion, sound, text, and tabular. Let's dive into AutoML with Apple's CreateML. Notice the entire list of many automated forms of machine learning in Figure 5-13 and how they ultimately converge to the same Core ML model that runs on iOS.

Figure 5-13. Create ML

To get started with Create ML, do the following:

1. Download XCode.
2. Open up XCode and right-click the icon to launch Create ML (Figure 5-14).

Figure 5-14. Open Create ML

Next, use the Image Classifier template (see Figure 5-15).

Figure 5-15. Image Classifier Template

You can grab a smaller version of the Kaggle dataset "cats and dogs" in the GitHub repository for the book. Drop the `cats-dogs-small` dataset onto the UI for Create ML (see Figure 5-16).

Figure 5-16. Upload data

Also, drop the test data onto the test section of the UI for Create ML.

Next, train the model by clicking the train icon. Note that you can train the model multiple times by right-clicking Model Sources. You may want to experiment with this because it allows you to test with "Augmentations" like Noise, Blur, Crop, Expose, Flip, and Rotate (see

Figure 5-17). These will enable you to create a more robust model that is more generalizable against real-world data.

Figure 5-17. Trained model

This small dataset should only take a few seconds to train the model (especially if you have the newer Apple M1 hardware). You can test it out by finding internet pictures of cats and dogs, downloading them, and dragging them into the preview icon (see Figure 5-18).

Figure 5-18. Preview

A final step is to download the model and use it in an iOS application. Note that in Figure 5-19, I use the OS X Finder menu and name the model and save it to my desktop. This final step may be the terminal step for a hobbyist who wants to build a bespoke iOS application that runs just on their phone. Once you save the model, you could optionally convert it to another format like ONNX and then run it on a cloud platform like Microsoft Azure.

Figure 5-19. Create ML model

Great work! You trained your first model that required zero code. The future will be fantastic as more of these tools evolve and get into the hands of consumers.

Optional next steps:

- You can train a more complex model by downloading a larger Kaggle dataset
- You can try other types of AutoML
- You can experiment with augmentation

Now that you know how to train a model using Create ML, let's go a bit deeper into how you further leverage Apple's Core ML tools.

Apple's Core ML Tools

One of the more exciting workflows available for the Apple ecosystem is downloading models and converting them to the Core ML tools via a Python library. There are many locations to grab pretrained models, including TensorFlow Hub.

In this example, let's walk through the code in this Colab notebook.

First, install the `coremltools` library:

```
!pip install coremltools
import coremltools
```

Next, download the model (based on the official quickstart guide).

Import tensorflow as tf:

```
# Download MobileNetv2 (using tf.keras)
keras_model = tf.keras.applications.MobileNetV2(
    weights="imagenet",
    input_shape=(224, 224, 3,),
    classes=1000,
)
# Download class labels (from a separate file)
import urllib
label_url = 'https://storage.googleapis.com/download.tensorflow.org/\
    data/ImageNetLabels.txt'
class_labels = urllib.request.urlopen(label_url).read().splitlines()
class_labels = class_labels[1:] # remove the first class which is background
assert len(class_labels) == 1000

# make sure entries of class_labels are strings
for i, label in enumerate(class_labels):
  if isinstance(label, bytes):
    class_labels[i] = label.decode("utf8")
```

Convert the model and set the metadata for the model to the correct parameters:

```
import coremltools as ct

# Define the input type as image,
# set preprocessing parameters to normalize the image
# to have its values in the interval [-1,1]
# as expected by the mobilenet model
image_input = ct.ImageType(shape=(1, 224, 224, 3,),
                           bias=[-1,-1,-1], scale=1/127)

# set class labels
classifier_config = ct.ClassifierConfig(class_labels)

# Convert the model using the Unified Conversion API
model = ct.convert(
    keras_model, inputs=[image_input], classifier_config=classifier_config,
)
```

Now update the metadata of the model:

```
# Set feature descriptions (these show up as comments in XCode)
model.input_description["input_1"] = "Input image to be classified"
model.output_description["classLabel"] = "Most likely image category"

# Set model author name
model.author = "" # Set the license of the model

# Set the license of the model
model.license = ""# Set a short description for the Xcode UI

# Set a short description for the Xcode UI
model.short_description = "" # Set a version for the model

# Set a version for the model
model.version = "2.0"
```

Finally, save the model, download it from Colab, and open it in XCode to predict (see Figure 5-20).

Figure 5-20. Download model

```
# Save model
model.save("MobileNetV2.mlmodel")

# Load a saved model
loaded_model = ct.models.MLModel("MobileNetV2.mlmodel")
```

Figure 5-21 shows an example prediction.

Figure 5-21. Stingray predict

The big takeaway is this process is even easier than using AutoML. Therefore, it may make more sense to download a model created by experts with access to expensive compute clusters than train it yourself in many cases. Apple's Core ML framework allows both the use case of bespoke AutoML as well as using pretrained models.

Google's AutoML and Edge Computer Vision

In the last few years, I have taught hundreds of students in a class called "Applied Computer Vision" at top data science universities. The premise of the course is to build solutions quickly using the most high-level tools available, including Google AutoML and edge hardware like the Coral.AI chip that contains a TPU or the Intel Movidius.

Figure 5-22 shows two examples of small form-factor edge machine learning solutions.

Figure 5-22. Edge hardware

One of the surprising things about teaching the class is how quickly students can take "off the shelf" solutions, piece them together, and come up with a solution that solves a problem. I have seen projects including mask detection, license plate detection, and trash sorting applications running

on mobile devices with little to no code. We are in a new era, the MLOps era, and it is easier to put code into working applications.

Like Apple and Google, many companies build a vertically integrated stack that provides a machine learning framework, operating systems, and specialized hardware like an ASIC (application-specific integrated circuit) that performs particular machine learning tasks. For example, the TPU, or TensorFlow Processing Unit, is actively developed with regular updates to the chip design. The edge version is a purpose-built ASIC that runs ML models. This tight integration is essential for organizations looking for the rapid creation of real-world machine learning solutions.

There are several critical approaches to computer vision on the GCP platform (similar to other cloud platforms, the service names are different). These options appear in order of difficulty:

- Write machine learning code that trains a model
- Use Google AutoML Vision
- Download a pretrained model from TensorFlow Hub or another location
- Use the Vision AI API

Let's examine a Google AutoML Vision workflow that ends in a computer vision model deployed to an iOS device. This workflow is essentially the same, whether you use a sample dataset that Google provides or your own:

1. Launch Google Cloud Console and open a cloud shell.
2. Enable the Google AutoML Vision API and give your project permission to it; you would set both the `PROJECT_ID` and `USERNAME`:

```
gcloud projects add-iam-policy-binding $PROJECT_ID \
--member="user:$USERNAME" \
--role="roles/automl.admin"
```

3. Upload training data and labels via a CSV file to Google Cloud Storage. If you set a `${BUCKET} ` variable `export BUCKET=$FOOBAR`, then three commands are all you need to copy Google sample data. Here is

one example for Cloud Classification (cirrus, cumulonimbus, cumulus). You can find a walkthrough on Google Qwiklabs under "Classify Images of Clouds in the Cloud with AutoML Vision". The `gs://spls/gsp223/images/` location holds the data in this example, and the `sed` command swaps out the specific paths:

```
gsutil -m cp -r gs://spls/gsp223/images/* gs://${BUCKET}
gsutil cp gs://spls/gsp223/data.csv .
sed -i -e "s/placeholder/${BUCKET}/g" ./data.csv
```

ADDITIONAL DATASETS IDEAL FOR GOOGLE AUTOML

Two other datasets you may also want to try to include are tf_flowers data and cats and dogs data. Yet another idea is uploading your data.

---

4. Visually inspect the data.
   One of the valuable aspects of Google's Cloud AutoML systems is using high-level tools to inspect the data, add new labels, or fix data quality control issues. Notice in Figure 5-23 that you have the ability to toggle between the different classification categories, which happen to be flowers.

Figure 5-23. Inspect data

5. Train the model and evaluate.
   Training the model is a button click in the console. Google has collected these options into its product Google Vertex AI. Notice in Figure 5-24 that there are a series of actions from Notebooks to Batch Predictions on the left panel. When creating a new training job, AutoML is an option, as well as AutoML Edge.

Figure 5-24. Google Vertex AI

6. Afterward, evaluate the trained model using the built-in tools (see Figure 5-25).

Figure 5-25. Evaluate data

7. Do something with the model: predict online or download.
   With Google AutoML vision, there is the ability to create an online hosted endpoint or download the model and make predictions on an edge device: iOS, Android, Javascript, Coral Hardware, or a container (see Figure 5-26).

Figure 5-26. Download model

The main takeaway is that Google Cloud offers a well-traveled path from uploading training data to training with minimal or no code required to build machine learning solutions that deploy to edge devices. These options are all integrated as part of Google's managed machine learning platform Vertex AI.

Next, let's dive into Azure's AutoML solutions, which like Google's, have a complete story about managing the lifecycle of MLOps.

Azure's AutoML

There are two primary ways to access the Azure AutoML. One is the console, and the other is programmatic access to the AutoML Python SDK. Let's take a look at the console first.

To get started doing AutoML on Azure, you need to launch an instance of Azure ML Studio and select the Automated ML option (see Figure 5-27).

Figure 5-27. Azure AutoML

Next, create a dataset either by uploading it or using an open dataset. In this example, I use the data from the Kaggle Social Power NBA project (see Figure 5-28).

Then, I spin up a classification job to predict which position a player could play based on the features in the dataset. Note that many different

types of machine learning predictions are available, including numerical regression and time-series forecasting. You will need to set up storage and a cluster if you have not already done so (see Figure 5-29) .

Figure 5-28. Azure AutoML create dataset

Figure 5-29. Azure AutoML classify

Once jobs complete, you can also ask Azure ML Studio to "explain" how it got to its predictions. A machine learning system explains how a model comes up with forecasts via "explainability," which is a critical upcoming capability of AutoML systems. You can see these explanatory capabilities in Figure 5-30. Notice how deep integration into the ML Studio solution gives this platform technology an extensive feel.

Figure 5-30. Azure AutoML explain

Let's take a look at the other approach. You can use Python to call the same API available from the Azure ML Studio console. This official Microsoft tutorial explains it in detail, but the critical section is shown here:

```
from azureml.train.automl import AutoMLConfig

automl_config = AutoMLConfig(task='regression',
                             debug_log='automated_ml_errors.log',
                             training_data=x_train,
                             label_column_name="totalAmount",
                             **automl_settings)
```

AWS AutoML

As the largest cloud provider, AWS also has many AutoML solutions. One of the earliest solutions includes a tool with a bad name, "Machine Learning," that is no longer widely available but was an AutoML solution. Now the recommended solution is SageMaker AutoPilot (Figure 5-31). You

can view many examples of SageMaker Autopilot in action from the official documentation.

Figure 5-31. SageMaker Autopilot

Let's walk through how to do an Autopilot experiment with AWS SageMaker. First, as shown in Figure 5-32, open SageMaker Autopilot and select a new task.

Figure 5-32. SageMaker Autopilot task

Next, I upload the "NBA Players Data Kaggle Project" into Amazon S3. Now that I have data to work with, I create an experiment as shown in Figure 5-33. Notice that I select for a target the draft position. This classification is because I want to create a prediction model that shows what draft position an NBA player deserves based on their performance.

Figure 5-33. Create Autopilot experiment

Once I submit the experiment, SageMaker Autopilot goes through a preprocessing stage through Model Tuning, as Figure 5-34 shows.

Now that the AutoML pipeline is running, you can see the resources it uses in the Resources tab, as shown in Figure 5-35.

Figure 5-34. Run Autopilot experiment

Figure 5-35. Autopilot instances

When the training is complete, you can see a list of models and their accuracy, as shown in Figure 5-36. Note that SageMaker was able to create a highly accurate classification model with an accuracy of .999945.

Figure 5-36. Completed Autopilot run

Finally, as shown in Figure 5-37, once the job is complete, you can right-click the model you want to control and either deploy it to production or open it in trail details mode to inspect explainability and/or metrics or charts.

SageMaker Autopilot is a complete solution for AutoML and MLOps, and if your organization is already using AWS, it does seem straightforward to integrate this platform into your existing workflows. It seems especially useful when working with larger datasets and with problems where reproducibility is critically essential.

Figure 5-37. Autopilot model

Next, let's discuss some of the open source AutoML solutions that are emerging.

Open Source AutoML Solutions

I still remember fondly my days of working on Unix clusters while working at Caltech in 2000. This particular time was a transitionary time for Unix, though, because even though in many cases Solaris was superior to Linux, it couldn't compete with the price of the Linux operating system, which was free.

I see a similar thing happening with open source AutoML solutions. The ability to train and run models using high-level tools appears to head toward commodification. So let's take a look at some of the options in open source.

Ludwig

One of the more promising approaches to open source AutoML is Ludwig AutoML. In Figure 5-38, the output from a Ludwig run shows the metrics useful to evaluate the strength of the model. The advantage to open source is that a corporation does not control it! Here is an example project that shows text classification using Ludwig via Colab notebook.

First, install Ludwig and set up a download:

```
!pip install -q ludwig
!wget https://raw.githubusercontent.com/paiml/practical-mlops-book/main/chap05/
    config.yaml
!wget https://raw.githubusercontent.com/paiml/practical-mlops-book/main/chap05/
    reuters-allcats.csv
```

Next, the model is just a command line invocation. This step then trains the model:

```
!ludwig experiment \
 --dataset reuters-allcats.csv \
 --config_file config.yaml
```

Figure 5-38. Ludwig

You can find many additional excellent examples of Ludwig in its official documentation.

One of the more exciting aspects of Ludwig is that it is under active development. As part of the Linux Foundation, they recently released version 4, which you can see in Figure 5-39. It adds many additional features like working with remote filesystems and distributed out-of-memory tools like Dask and Ray. Finally, Ludwig has a deep integration with MLflow. The Ludwig roadmap shows that it will continue to support and enhance this integration.

Figure 5-39. Ludwig version 4

FLAML

Another entrant into open source AutoML is FLAML. It has a design that accounts for cost-effective hyperparameter optimization. You can see the FLAML logo in Figure 5-40.

Figure 5-40. FLAML from Microsoft Research

One of the primary use cases for FLAML is to automate an entire modeling process with as little as three lines of code. You can see this in the following example:

```
from flaml import AutoML
automl = AutoML()
automl.fit(X_train, y_train, task="classification")
```

A more comprehensive example shows that in a Jupyter notebook, you first install the library `!pip install -q flaml`, then configure the AutoML configuration. Then it kicks off a training job to select the optimized classification model:

```
!pip install -q flaml

from flaml import AutoML
from sklearn.datasets import load_iris
# Initialize an AutoML instance
automl = AutoML()
# Specify automl goal and constraint
automl_settings = {
    "time_budget": 10,  # in seconds
    "metric": 'accuracy',
    "task": 'classification',
}
X_train, y_train = load_iris(return_X_y=True)
# Train with labeled input data
automl.fit(X_train=X_train, y_train=y_train,
           **automl_settings)
# Predict
print(automl.predict_proba(X_train))
# Export the best model
print(automl.model)
```

You can see in Figure 5-41 that, after multiple iterations, it selects an XGBClassifier with a set of optimized hyperparameters.

Figure 5-41. FLAML output from model selection

What is exciting about these open source frameworks is their ability to make complicated things possible and easy things automated. Next up, let's take a look at how model explainability works with a project walkthrough.

NOTE

There is no shortage of open source AutoML frameworks. Here are some additional frameworks to look at for AutoML:

- AutoML
  - H2O AutoML
  - Auto-sklearn
  - tpot
  - PyCaret
  - AutoKeras

Model Explainability

An important aspect of automation in machine learning is to automate model explainability. MLOps platforms all can use this capability as yet another dashboard for the team to look at during work. For example, an MLOps team starting at work in the morning may look at the CPU and memory usage of servers *and* the explainability report of the model they trained last night.

Cloud-based MLOps frameworks like AWS SageMaker, Azure ML Studio, and Google Vertex AI have built-in model explainability, but you can implement it yourself with open source software as well. Let's walk through an explainability workflow to see how this works using this model explainability GitHub project.

NOTE

Two popular open source Model Explainability frameworks are ELI5 and SHAP. Here is a bit more information about each one.

ELI5

ELI5 stands for "explain like I am five." It allows you to visualize and debug machine learning models and supports several frameworks, including sklearn.

SHAP

SHAP is a "game-theoretic" approach to explain the output of machine learning models. In particular, it has excellent visualizations as well as explanations.

First, using a Jupyter notebook, let's ingest NBA data from the 2016–2017 season and print out the first few rows using the `head` command. This data contains AGE, POSITION, FG (Field Goals/Game), and social media data like Twitter retweets:

```
import pandas as pd

player_data = "https://raw.githubusercontent.com/noahgift/socialpowernba/\
master/data/nba_2017_players_with_salary_wiki_twitter.csv"
df = pd.read_csv(player_data)
df.head()
```

Next, let's create a new feature called a `winning_season`, which allows us to predict whether a player will be part of a team with a winning season. For example, in Figure 5-42, you can see plotting the NBA player's age versus the wins to discover a potential age-based pattern.

Figure 5-42. Winning season feature

Now, let's move on to modeling and predict wins. But first, let's clean up the data a bit and drop columns that aren't necessary and drop missing values:

```
df2 = df[["AGE", "POINTS", "SALARY_MILLIONS", "PAGEVIEWS",
 "TWITTER_FAVORITE_COUNT","winning_season", "TOV"]]
df = df2.dropna()
target = df["winning_season"]
features = df[["AGE", "POINTS","SALARY_MILLIONS", "PAGEVIEWS",
 "TWITTER_FAVORITE_COUNT", "TOV"]]
classes = ["winning", "losing"]
```

After this cleanup, the `shape` command prints out the number of rows, 239, and columns, 7:

```
df2.shape
(239, 7)
```

Next up, let's train the model by first splitting the data, then using logistic regression:

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(features, target,
 test_size=0.25,
 random_state=0)
from sklearn.linear_model import LogisticRegression
model = LogisticRegression(solver='lbfgs', max_iter=1000)
model.fit(x_train, y_train)
```

You should see an output similar to the following result that shows the model training is successful:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=1000,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

Now, let's move on to the fun part explaining how the model came up with its SHAP framework predictions. But, first, SHAP needs installation:

```
!pip install -q shap
```

Next, let's use `xgboost`, another classification algorithm, to explain to the model since SHAP has outstanding support for it:

```
import xgboost
import shap
model_xgboost = xgboost.train({"learning_rate": 0.01},
                              xgboost.DMatrix(x_train, label=y_train), 100)
# load JS visualization code to notebook
shap.initjs()
# explain the model's predictions using SHAP values
# (same syntax works for LightGBM, CatBoost, and scikit-learn models)
explainer = shap.TreeExplainer(model_xgboost)
shap_values = explainer.shap_values(features)
# visualize the first prediction's explanation
shap.force_plot(explainer.expected_value, shap_values[0,:], features.iloc[0,:])
```

In Figure 5-43, you can see a force plot in SHAP showing the features in red push the prediction higher, and the features in blue push the prediction lower.

Figure 5-43. SHAP output xgboost

```
shap.summary_plot(shap_values, features, plot_type="bar")
```

In Figure 5-44, a summary plot shows the absolute mean value of what features drive the model. So, for example, you can see how "off the court" metrics like Twitter and Salary are essential factors in why the model predicts wins the way it does.

Figure 5-44. SHAP feature importance

Let's see how another open source tool works; this time, let's use ELI5. First, install it with `pip`:

```
!pip install -q eli5
```

Next up, permutation importance performs on the original logistic regression model created earlier. This process works by measuring the amount of accuracy decrease by removing features:

```
import eli5
from eli5.sklearn import PermutationImportance

perm = PermutationImportance(model, random_state=1).fit(x_train, y_train)
eli5.show_weights(perm, feature_names = features.columns.tolist())
```

You can see in Figure 5-45 that the original logistic regression model has a different feature importance than the XGBoost model. In particular, note that the age of a player has a negative correlation with wins.

Figure 5-45. ELI5 permutation importance

Explainability is an essential aspect of MLOps. Just as we have dashboards and metrics for software systems, there should also be explainability for how an AI/ML system came to its prediction. This explainability can lead to healthier outcomes for both stakeholders of the business and the business itself.

Next, let's wrap up everything covered in the chapter.

Conclusion

AutoML is an essential new capability for any team doing MLOps. AutoML improves the ability for a team to push models into production, work on complex problems, and ultimately work on what matters. It is essential to point out that Automated Modeling, i.e., AutoML, is not the only component of KaizenML or continuous improvement. In the oft-cited paper "Hidden Technical Debt in Machine Learning Systems" the authors mention that modeling is a trivial amount of the work in a real-world ML system. Likewise, it should be no surprise that AutoML, i.e., the automation of modeling, is a small part of what needs to be automated. Everything from data ingestion to feature storage to modeling to training to deployment to an evaluation of the model in production is a candidate

for full automation. KaizenML means you are constantly improving every single part of the machine learning system.

Just as automated transmission and cruise-control systems assist an expert driver, automation of the subcomponents of a production machine learning system makes the humans in charge of the ML decisions better. Things can and should be automated, including modeling aspects, software engineering best practices, testing, data engineering, and other essential components. Continuous improvement is a cultural change that doesn't have an end date and fits into any organization wanting to make impactful changes with AI and machine learning.

A final takeaway is that there are many free or nearly free AutoML solutions. Just as developers worldwide use free or roughly free high-level tools like build servers and code editors to improve software, ML practitioners should use automation tools of all types to enhance their productivity.

Next up is the chapter on monitoring and logging. I call this "data science for operations." Before you jump into that topic, take a look at the following exercises and critical thinking questions.

Exercises

- Download XCode and use Apple's Create ML to train a model from a sample dataset you find on Kaggle, or another open dataset location.
- Use Google's AutoML Computer Vision platform to train a model and deploy it to a Coral.AI device.
- Use Azure ML Studio to train a model and explore the explainability features of Azure ML Studio.
- Use ELI5 to explain a machine learning model.
- Use Ludwig to train a machine learning model.
- Select a SageMaker Automatic Model Tuning example from the official SageMaker examples and run through it on your AWS Account.

Critical Thinking Discussion Questions

- Why is AutoML only part of the automation story with modern machine learning?
- How could the NIH (National Institutes of Health) use a Feature Store to increase the speed of medical discoveries?
- By 2025, what parts of machine learning will be fully automated and what aspects will not? By 2035, what parts of machine learning will be fully automated and what factors will not?
- How could vertically integrated AI platforms (chips, frameworks, data, and more) give particular companies a competitive advantage?
- How does the chess software industry provide insights into how AI and humans work together for improved outcomes in solving problems for AutoML?
- How does a data-centric approach differ from a model-centric approach to machine learning? How about a KaizenML approach where data, software, and modeling are all treated as equally important?