Chapter 6. Monitoring and Logging

By Alfredo Deza

> *Not only is the cerebral anatomy double, and not only is it unarguable that one hemisphere is enough for consciousness; beyond that, two hemispheres following callosotomy have been shown to be conscious simultaneously and independently. As Nagel said of the split-brain, "What the right hemisphere can do on its own is too elaborate, too intentionally directed, and too psychologically intelligible to be regarded merely as a collection of unconscious automatic responses.*
>
> —Dr. Joseph Bogen

Both logging and monitoring are core pillars of DevOps principles that are crucial to robust ML practices. Useful logging and monitoring are hard to get right, and although you can leverage cloud services that take care of the heavy lifting, it is up to you to decide and come up with a sound strategy that makes sense. Most software engineers tend to prefer writing code and leave behind other tasks like testing, documentation, and very often logging and monitoring.

Don't be surprised to hear suggestions about automated solutions that can "solve the logging problem." A solid foundation is possible by thinking thoroughly about the problem at hand so that the information produced is usable. The hard work and solid foundation ideals I describe become crystal clear when you face useless information (it doesn't help narrate a story) or cryptic (too hard to understand). A perfect example of this situation is a software issue I opened back in 2014 that captured the following question from an online chat about the product:

"Can anyone help me interpret this line:

```
7fede0763700  0 -- :/1040921 >> 172.16.17.55:6789/0 pipe(0x7feddc022470 \
sd=3 :0 s=1 pgs=0 cs=0 l=1 c=0x7feddc0226e0).fault
```

I’d been working with this software product for almost two years at the time, and I had no idea what that meant. Can you think of a possible answer? A knowledgeable engineer had the perfect translation: “The machine you are on is unable to contact the monitor at 172.16.17.55.” I was baffled by what the log statement meant. Why can’t we make a change to say that instead? The ticket capturing this issue from 2014, as of this writing, is still open. Even more troubling is that engineering replied in that ticket that “The log message is fine.”

Logging and monitoring is hard work because it takes effort to produce meaningful output that helps us understand a program’s state.

I’ve mentioned that it is crucial to have information that helps us narrate a story. This is true of both monitoring and logging. A few years ago, I worked in a large engineering group that delivered one of the world’s largest Python-based CMSs (Content Management System). After proposing adding metrics to the application, the general sentiment was that the CMS didn’t need it. Monitoring was already in place, and the Ops team had all kinds of utilities tied to thresholds for alerts. The engineering managers rewarded excellence by giving an engineer time to work on any relevant project (not just 20% like some famous tech company). Before working on a relevant project, one had to pitch the whole management team’s idea to get buy-in. When my time came, I, of course, chose to add metric facilities to the application.

“Alfredo, we already have metrics, we know the disk usage, and we have memory alerts for every server. We don’t get what we gain by this initiative.” It is hard to be standing in front of a large senior management group, and trying to convince them of something they don’t believe in. My explanation started with the most important button on the website: the subscribe button. That subscribe button was the one in charge of producing paying users and crucial to the business. I explained that, “if we deploy a new version with a JavaScript issue that makes this button unusable, what metric or alert can tell us this is a problem?” Of course, disk usage would be the same, and memory usage would probably not change at all. And yet, the most important button in the application would go unnoticed in an unusable state. In this particular case, metrics can capture the click-through rate of every hour, day, and week for that button. And most importantly, it can help tell a story about how today the website generates more (or perhaps less!) revenue than last year on this same month. Disk usage and memory consumption of servers are worth keeping an eye on, but it isn't the ultimate goal.

These stories aren't explicitly tied to machine learning or delivering trained models to production. Still, as you will see in this chapter, it will help you and your company tell a story and increase confidence in your process by surfacing important issues and pointing at the reason why a model probably needs better data before hitting production. Identifying data drift and accuracy over time is critical in ML operations. Deploying a model into production with a significant change in accuracy should never happen and needs prevention. The earlier these problems are detected, the cheaper it is to fix them. The consequences of having an inaccurate model in production can be catastrophic.

Observability for Cloud MLOps

It is safe to say that most machine learning is taking place in a cloud environment. As such, there are special services from cloud providers that enable observability. For example, on AWS, they have Amazon CloudWatch, on GCP they have the Google Cloud operations suite, and on Azure they have Azure Monitor.

In Figure 6-1, Amazon CloudWatch is a better example of how these monitoring services work. At a high level, every component of the cloud system sends metrics and logs to CloudWatch. These tasks include the servers, the application logs, the training metadata for machine learning jobs, and the results of production machine learning endpoints.

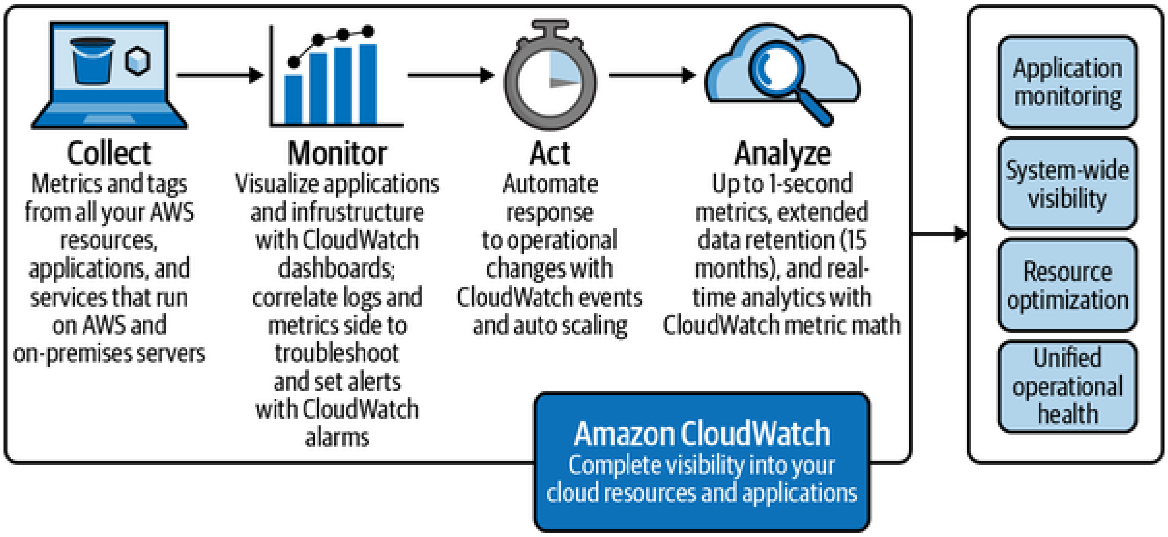

Collect
Metrics and tags from all your AWS resources, applications, and services that run on AWS and on-premises servers
Monitor
Visualize applications and infrustructure with CloudWatch dashboards; correlate logs and metrics side to troubleshoot and set alerts with CloudWatch alarms
Act
Automate response to operational changes with CloudWatch events and auto scaling
Analyze
Up to 1-second metrics, extended data retention (15 months), and real-time analytics with CloudWatch metric math
Amazon CloudWatch
Complete visibility into your cloud resources and applications
Application monitoring
System-wide visibility
Resource optimization
Unified operational health

Figure 6-1. AWS CloudWatch

Next, this information becomes part of many different tools, from dashboards to auto-scaling. For example, in AWS SageMaker, this could mean that the production ML service will scale automatically if the individual endpoints exceed more than 75% of their total CPU or memory. Finally, all of this observability allows humans and machines alike to analyze what is going on with a production ML system and act on it.

The experienced cloud software engineer already knows that cloud observability tools are nonoptional components of a cloud software deployment. However, what is unique about MLOps in the cloud is that new components also need granular monitoring. For example, in Figure 6-2, a whole new series of actions take place in an ML model deployment. Notice, though, that again, CloudWatch collects these new metrics.

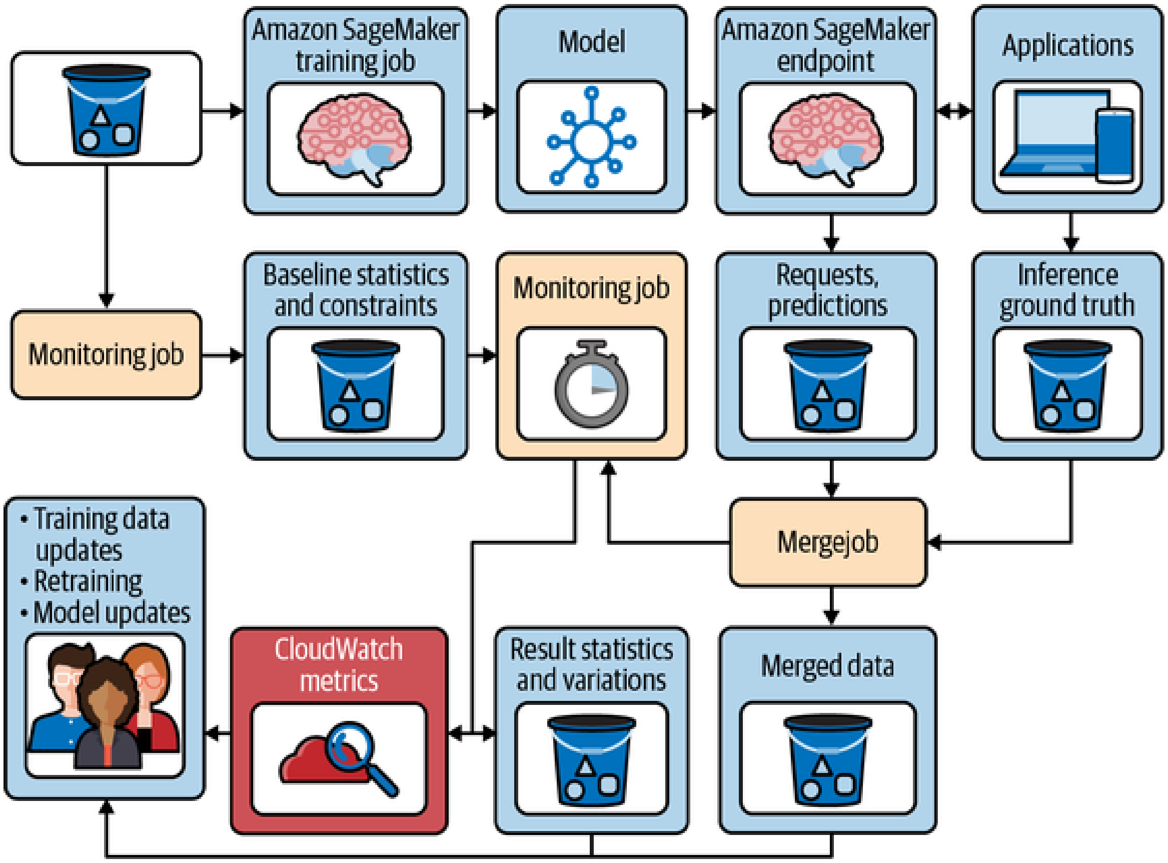


Figure 6-2. AWS SageMaker model monitoring

As the rest of the chapter unfolds, keep in mind that at a high level on a master system like CloudWatch, it collects data, routes alerts, and engages with the dynamic components of cloud computing like elastic scaling. Next, let's get into the details of a finer-grained member of observability: logging.

Introduction to Logging

Most logging facilities have common aspects in how they work. Systems define the log levels they can operate with, and then the user can select at what level those statements should appear. For example, the Nginx web server comes with a default configuration for access logs saved at */var/log/nginx/access.log* and error logs to */var/log/nginx/error.log*. As an Nginx user, the first thing you need to do if you are troubleshooting the web server is to go into those fails and see the output.

I installed Nginx in an Ubuntu server with default configurations and sent some HTTP requests. Right away, the access logs started getting some information:

```
172.17.0.1 [22/Jan/2021:14:53:35 +0000] "GET / HTTP/1.1" \
    "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:84.0) Firefox/84.0" "-"
172.17.0.1 [22/Jan/2021:14:53:38 +0000] "GET / HTTP/1.1" \
    "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:84.0) Firefox/84.0" "-"
```

The long log lines pack lots of useful (configurable) information that includes the server’s IP address, the time, and type of request alongside the user-agent information. The user-agent in my case is my browser running on a Macintosh computer. These aren’t errors, though. These lines show access to the server. To force Nginx to get into an error condition, I changed the permissions on a file to be unreadable by the server. Next, I sent new HTTP requests:

```
2021/01/22 14:59:12 [error] open() "/usr/share/nginx/html/index.html" failed \
  (13: Permission denied), client: 172.17.0.1, server: localhost, \
   request: "GET / HTTP/1.1", host: "192.168.0.200"
```

The log entry explicitly shows that the level of the information is of *“error.”* This allows a consumer, like myself, to identify the severity of the information produced by the web server. Back when I was a Systems Administrator and was getting started in necessary tasks like configuration and deployment of production environments, it wasn’t clear why these levels were useful. The main point for me was that I could identify an error from the logs’ informational content.

Although the idea of making it easier for consumers to identify if an error is valid, it isn’t all there is to log levels. If you’ve tried to debug a program before, you’ve probably used `print()` statements to help you get useful information about a running program. There are many other ways to debug programs, but using `print()` is still valuable. One drawback is that you have to clean up and remove all those `print()` statements once the problem is solved. The next time you need to debug the same program, you will need to add all those statements back. This is not a good strategy, and it is one of the many situations where logging can help.

Now that some basics on logging are clear, I’ll discuss how to configure logging in an application, which tends to be complicated since there are so many different options and decisions to make.

Logging in Python

I’ll be working in Python, but most of the concepts in this section should apply cleanly to other languages and frameworks. Log levels, redirection of output, and other facilities are commonly available in other applications. To start applying logging, I will create a short Python script to process a CSV file. No functions or modules; the example script is as close to a Jupyter Notebook cell that you can find.

Create a new file called *describe.py* to see how logging can help in a basic script:

```python
import sys
import pandas as pd

argument = sys.argv[-1]

df = pd.read_csv(argument)
print(df.describe())
```

The script will take the input from the last argument on the command line and tell the Pandas library to read it and describe it. The idea is to produce a description of a CSV file, but that isn’t what happens when you run it with no arguments:

```
$ python derscribe.py
         from os import path
count                      5
unique                     5
top     print(df.describe())
freq                       1
```

What happens here is that the last argument in that example is the script itself, so Pandas is describing the contents of the script. This is not very useful, and to someone that hasn't created the script, the results are shocking, to say the least. Let's take this brittle script a step forward and pass an argument to a path that doesn't exist:

```
$ python describe.py /bogus/path.csv
Traceback (most recent call last):
  File "describe.py", line 7, in <module>
    df = pd.read_csv(argument)
  File "/../site-packages/pandas/io/parsers.py", line 605, in read_csv
    return _read(filepath_or_buffer, kwds)
  ...
  File "/../site-packages/pandas/io/common.py", line 639, in get_handle
    handle = open(
FileNotFoundError: [Errno 2] No such file or directory: '/bogus/path.csv'
```

There is no error checking to tell us if the input is valid and what the script expects. This problem is worse if you are waiting on a pipeline run or some remote data processing job to complete, and you are getting these types of errors. Some developers try to defend against these by catching all exceptions and obscuring the actual error, making it impossible to tell what is going on. This slightly modified version of the script highlights the problem better:

```python
import sys
import pandas as pd

argument = sys.argv[-1]

try:
    df = pd.read_csv(argument)
    print(df.describe())
except Exception:
    print("Had a problem trying to read the CSV file")
```

Running it produces an error that would make me very upset to see in production code:

```
$ python describe.py /bogus/path.csv
Had a problem trying to read the CSV file
```

The example is trivial, and because the script is just a few lines long and you know its contents, it isn't that difficult to point at the problem. But if this is running in an automated pipeline remotely, you don't have any context, and it becomes challenging to understand the problem. Let's use the Python logging module to provide more information about what is going on in this data processing script.

The first thing to do is to configure logging. We don't need anything too complicated here, and adding a few lines is more than enough. Modify the *describe.py* file to include these lines, then rerun the script:

```
import logging

logging.basicConfig()
logger = logging.getLogger("describe")
logger.setLevel(logging.DEBUG)

argument = sys.argv[-1]
logger.debug("processing input file: %s", argument)
```

Rerunning it should look similar to this:

```
$ python describe.py /bogus/path.csv
DEBUG:describe:processing input file: /bogus/path.csv
Had a problem trying to read the CSV file
```

Still, not very useful yet, but already informational. This might feel like a lot of boilerplate code for something that a simple `print()` statement could've also accomplished. The logging module is resilient to failures when constructing the message. For example, open a Python interpreter and try using fewer arguments to `print`:

```
>>> print("%s should break because: %s" % "statement")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: not enough arguments for format string
```

Now let’s try the same operation with the logging module:

```
>>> import logging
>>> logging.warning("%s should break because: %s", "statement")
--- Logging error ---
Traceback (most recent call last):
  ...
  File "/.../python3.8/logging/__init__.py", line 369, in getMessage
    msg = msg % self.args
TypeError: not enough arguments for format string
Call stack:
  File "<stdin>", line 1, in <module>
Message: '%s should break because: %s'
Arguments: ('statement',)
```

The last example would not break a production application. Logging should never break any application at runtime. In this case, the logging module is trying to do the variable replacement in the string and failing, but instead of breaking it is advertising the problem and then continuing. A print statement can’t do this. In fact, I see using `print()` in Python much like `echo` in shell scripts. There is no control, it is easy to break a production application, and it is hard to control the verbosity.

Verbosity is crucial when logging, and aside from the *error level* in the Nginx example, it has several facilities to empower log consumers to fine-tune the information needed. Before going into log-level granularity and verbosity control, give the script an upgrade in the log formatting to look nicer. The Python logging module is compelling and allows lots of configuration. Update the *describe.py* script where the logging configuration happens:

```
log_format = "[%(name)s][%(levelname)-6s] %(message)s"
logging.basicConfig(format=log_format)
logger = logging.getLogger("describe")
logger.setLevel(logging.DEBUG)
```

The `log_format` is a template with some keywords used when constructing the log line. Note how I didn’t have a timestamp before, and although I still don’t have it with this update, the configuration does allow me to include it. For now, the logger’s name (`describe` in this case), the log level, and the message are all there with some padding and using square brackets to separate the information for better readability. Rerun the script once again to check how the output changes:

```
$ python describe.py
[describe][DEBUG ] processing input file: describe.py
Had a problem trying to read the CSV file
```

Another superpower of logging is to provide the traceback information along with the error. Sometimes it is useful to capture (and show) the traceback without getting into an error condition. To do this, update the *describe.py* script in the `except` block:

```
try:
    df = pd.read_csv(argument)
    print(df.describe())
except Exception:
    logger.exception("Had a problem trying to read the CSV file")
```

It looks very similar to the `print()` statement from before. Rerun the script and check the results:

```
$ python logging_describe.py
[describe][DEBUG ] processing input file: logging_describe.py
[describe][ERROR ] Had a problem trying to read the CSV file
Traceback (most recent call last):
  File "logging_describe.py", line 15, in <module>
    df = pd.read_csv(argument)
  File "/.../site-packages/pandas/io/parsers.py", line 605, in read_csv
    return _read(filepath_or_buffer, kwds)
  ...
  File "pandas/_libs/parsers.pyx", line 1951, in pandas._libs.parsers.raise
ParserError: Error tokenizing data. C error: Expected 1 field in line 12, saw 2
```

The traceback is the string representation of the error, not an error itself. To verify this, add another log line right after the `except` block:

```
try:
    df = pd.read_csv(argument)
    print(df.describe())
```

```
except Exception:
    logger.exception("Had a problem trying to read the CSV file")

logger.info("the program continues, without issue")
```

Rerun to verify the outcome:

```
[describe][DEBUG ] processing input file: logging_describe.py
[describe][ERROR ] Had a problem trying to read the CSV file
Traceback (most recent call last):
[...]
ParserError: Error tokenizing data. C error: Expected 1 field in line 12, saw 2

[describe][INFO  ] the program continues, without issue
```

Modifying Log Levels

These types of informational facilities provided by logging are not straightforward with `print()` statements. If you are writing a shell script, it would be borderline impossible. In the example, we now have three log levels: debug, error, and info. Another thing logging allows is to selectively set the level to what we are interested in. Before changing it, the levels have a *weight* associated with them, and it is essential to grasp them so that changes reflect the priority. From most to least verbose, the order is as follows:

1. `debug`
2. `info`
3. `warning`
4. `error`
5. `critical`

Although this is the Python logging module, you should expect a similar weighted priority from other systems. A log level of `debug` will include every other level, as well as `debug`. A `critical` log level will include only `critical`-level messages. Update the script once again to set the log level to `error`:

```
log_format = "[%(name)s][%(levelname)-6s] %(message)s"
logging.basicConfig(format=log_format)
logger = logging.getLogger("describe")
logger.setLevel(logging.ERROR)
```

```
$ python logging_describe.py
[describe][ERROR ] Had a problem trying to read the CSV file
Traceback (most recent call last):
  File "logging_describe.py", line 15, in <module>
    df = pd.read_csv(argument)
  File "/.../site-packages/pandas/io/parsers.py", line 605, in read_csv
    return _read(filepath_or_buffer, kwds)
  ...
  File "pandas/_libs/parsers.pyx", line 1951, in pandas._libs.parsers.raise
ParserError: Error tokenizing data. C error: Expected 1 field in line 12, saw 2
```

The change causes only the error log message to be displayed. This is useful when trying to reduce the amount of logging to what may be of interest. Debugging encompasses most (if not all) messages, while errors and critical situations are much more sporadic and usually not seen as often. I recommend setting debug log levels for new production code so that it is easier to catch potential problems. You should expect problems when producing and deploying new code. Highly verbose output is not that useful in the long run when an application has proven stable, and there aren't many surprising issues. Fine-tuning the levels to `info` or `error` only is something you can do progressively.

Logging Different Applications

So far, we've seen log levels of a script trying to load a CSV file. And I haven't gone into the details of why the logger's name (*"describe"* in the past examples) is significant. In Python, you import modules and packages, and many of these packages come with their own loggers. The logging facility allows you to set particular options for these loggers independently from your application. There is also a hierarchy for loggers, where the *"root"* logger is the parent of all loggers and can modify settings for all applications and loggers.

Changing the logging level is one of the many things you can configure. In one production application, I created two loggers for the same application: one would emit messages to the terminal, while the other would write to a logfile. This allowed having user-friendly messages go to the terminal, omitting large tracebacks and errors polluting the output. At the same time, the internal, developer-oriented logging would go to a file. This is another example of something that would be very difficult and complicated to do with a `print()` statement or an `echo` directive in a shell script. The more flexibility you have, the better you can craft applications and services.

Create a new file and save it as *http-app.py* with the following contents:

```python
import requests
import logging

logging.basicConfig()

# new logger for this script
logger = logging.getLogger('http-app')

logger.info("About to send a request to example.com")
requests.get('http://example.com')
```

The script configures the logging facility with a basic configuration that emits messages to the terminal by default. It then tries to make a request using the *requests* library. Run it and check the output. It might feel surprising that nothing shows up in the terminal after executing the script. Before explaining exactly why this is happening, update the script:

```python
import requests
import logging

logging.basicConfig()
root_logger = logging.getLogger()

# Sets logging level for every single app, the "parent" logger
root_logger.setLevel(logging.DEBUG)
```

```
# new logger for this script
logger = logging.getLogger('http-app')

logger.info("About to send a request to example.com")
requests.get('http://example.com')
```

Rerun the script and take note of the output:

```
$ python http-app.py
INFO:http-app:About to send a request to example.com
DEBUG:urllib3.connectionpool:Starting new HTTP connection (1): example.com:80
DEBUG:urllib3.connectionpool:http://example.com:80 "GET / HTTP/1.1" 200 648
```

The Python logging module can set configuration settings globally for every single logger in all packages and modules. This *parent* logger is called the *root* logger. The reason there is output is that the changes set the root logger level to debug. But there is more output than the single line of the *http-app* logger. This happens because the *urllib3* package has its logger as well. Since the root logger changed the global log level to debug, the *urllib3* package is now emitting those messages.

It is possible to configure several different loggers and fine-tune the granularity and verbosity of the levels (as well as any other logging configuration). To demonstrate this, change the *urllib3* package's logging level by adding these lines at the end of the *http-app.py* script:

```
# fine tune the urllib logger:
urllib_logger = logging.getLogger('urllib3')
urllib_logger.setLevel(logging.ERROR)

logger.info("About to send another request to example.com")
requests.get('http://example.com')
```

The updated version retrieves the logger for *urllib3* and changes its log level to error. The script's logger emits a new message before calling `requests.get()`, which in turn uses the *urllib3* package. Rerun the script once more to check the output:

```
INFO:http-app:About to send a request to example.com
DEBUG:urllib3.connectionpool:Starting new HTTP connection (1): example.com:80
DEBUG:urllib3.connectionpool:http://example.com:80 "GET / HTTP/1.1" 200 648
INFO:http-app:About to send another request to example.com
```

Since the log level of *urllib3* got changed before the last request, no more debug messages appear. The info-level message does show up because that logger is still configured at a debug level. These combinations of logging configurations are powerful because it allows you to select what is interesting from other information that can cause *noise* in the output.

Imagine using a library that interacts with a cloud's storage solution. The application you are developing performs thousands of interactions by downloading, listing, and uploading content to the storage server. Suppose the primary concern of the application is to manage datasets and offload them to the cloud provider. Do you think it would be interesting to see an informational message that says a request is about to be made to the cloud provider? In most situations, I would say that is far from useful. Instead, it would be critical to be alerted when the application fails to perform a particular operation with the storage. Further, it may be possible that you are dealing with timeouts when requesting the storage, in which case, changing the log level to indicate when the request happens is crucial to get the time delta.

It is all about flexibility and adapting to your application's lifecycle needs. What is useful today can be information overload tomorrow. Logging goes hand-in-hand with monitoring (covered in the next section), and it is not uncommon to hear about observability in the same conversation. These are foundational to DevOps, and they should be part of the ML lifecycle.

Monitoring and Observability

When I was a professional athlete, my dad, who was also my coach, added a particular chore that I despised: write every day, in a journal, about the workout that I'd just completed. The journal entry needed the following items:

- The planned workout. For example, if it was 10 repetitions of 300 meters at a 42 second pace.
- The results of the workout. For the 300 meters, it would be the actual time performed at each repetition.
- How I felt during and after the session.
- Any other relevant information, like feeling sick or pain from an injury, for example.

I started training professionally at 11 years old. As a teenager, this journaling task felt worse than the worst workouts. I didn't understand the big deal about journaling and why it was vital for me to cover the workout details. I had the guts to tell my dad that this seemed his job, not mine. After all, I was already doing the workouts. That argument didn't go very well for me. The problem is that I didn't understand. It felt like a useless task, without any benefits. I couldn't see or feel the benefits.

It felt more like a disciplinary task at the end of the day instead of a crucial training aspect. A few years after journaling every day (I worked out 14 times a week on average) and having a great season behind me, I sat down with my dad to plan the next season. Instead of a couple of hours of hearing my dad talk to me about what was coming for the next season, he illuminated me by demonstrating how powerful journaling was. *"Alright, Alfredo, let me pull up the last two journals, to check what we did and how you felt, to adapt, increase, and have a great season once again."*

The journals had every piece of information that we needed to plan. He used a phrase that stuck with me for years, and I hope it demonstrates why monitoring and metrics are critical in any situation: *"If we can measure, then we can compare. And if we can compare, only then can we improve."*

Machine learning operations are no different. When a new iteration of a model ships to production, you have to know if it performs better or worse. Not only do you have to know, but the information has to be accessible and straightforward to produce. Processes can create friction and make everything go slower than it should. Automation brings down silly processes and can effortlessly create information availability.

A few years ago, I worked at a startup where the head of sales would send me a CSV file with new accounts to “run some numbers” on it and send him a PDF back with the results. This is horrible; it doesn’t scale. And it doesn’t survive the bus test.

The bus test is where I can get hit by a bus today, and everything should still work, and everyone that works with me can pick up the slack. All effort in automation and producing metrics is instrumental for shipping robust models to production.

Basics of Model Monitoring

Monitoring in ML operations means everything and anything that has to do with getting a model into production—from capturing information about the systems and services to the performance of the model itself. There is no single silver bullet to implement monitoring that will make everything right. Monitoring and capturing metrics is somewhat similar to knowing the data before figuring out what features and algorithms to use to train a model. The more you know about the data, the better decisions you can make about training the model.

Similarly, the more you know about the steps involved in getting a model to production, the better the choices you make when capturing metrics and setting monitoring alerts. The answer to *“what metric should you capture when training a model?”* is one that I dislike, but is so accurate in this situation: it depends.

Although there are differences between the metrics and types of alerts you can set, there are some useful foundational patterns that you can default to. These patterns will help you clarify what data to collect, how often that collection should occur, and how to best visualize it. Finally, depending on the step in producing a model, the key metrics will be different. For example, when collecting data and cleaning it up, it might be substantial to detect the number of empty values per column and perhaps the time to completion when processing the data.

The past week I was processing system vulnerability data. And I had to make some changes to the application in charge of reading JSON files and save the information in a database. After some changes, the database went from several gigabytes in size to just 100 megabytes. The code change wasn't meant to reduce the size at all, so I immediately knew that I'd made a mistake that needed correction. Encountering these situations are excellent opportunities to determine what metrics (and when) to capture them.

There are a few types of metrics that you can find in most metric-capturing systems:

Counter

> As the name implies, this type of metric can be useful when counting any type of item. It is useful when iterating over items. For example, this could be useful for counting empty cell values per column.

Timer

> A timer is excellent when trying to determine how long some action will take. This is crucial for performance monitoring, as its core responsibility functionality is to measure time spent during an action. Time spent is commonly seen in monitoring graphs for hosted HTTP APIs. If you have a hosted model, a timer would help capture how long the model took to produce a prediction over HTTP.

Value

> When counters and timers do not fit the metric to capture, values come in handy. I tend to think of values like parts of an algebra equation: even if I don't know what *X* is, I want to capture its value and persist it. Reusing the example of processing JSON files at work and saving information to a database, a fair use for this metric would be the size (in gigabytes) of the resulting database.

There are two common operations specific to ML that cloud providers need to monitor and capture useful metrics. The first is the target dataset. This can be the same dataset you used to train a model, although special care needs to be put into ensuring that the number (and order) of features doesn't change. The second one is the baseline. The baseline determines what differences may (or may not) be acceptable when a model gets trained. Think about baseline as the acceptable thresholds to determine how fit a model is for production usage.

Now that the basics are clear and that a target dataset with a baseline is understood, let's use them to capture useful metrics when training models.

Monitoring Drift with AWS SageMaker

As I mentioned already, cloud providers will usually need a target dataset and a baseline. This is no different with AWS. In this section, we will produce metrics and capture data violations from an already deployed model. SageMaker is an incredible tool for inspecting datasets, training models, and provisioning models into production environments. Since SageMaker tightly integrates with other AWS offerings like S3 storage, you can leverage saving target information that can quickly be processed elsewhere that has access.

One thing in particular that I like about SageMaker is its Jupyter Notebook offering. The interface is not as polished as Google's Colab, but it packs features like the AWS SDK preinstalled and substantial kernel types to choose from—from (the now deprecated) Python 2.7 to Conda environments running Python 3.6, as shown in Figure 6-3.

> **NOTE**
>
> This section doesn't cover the specifics of deploying a model. If you want to dive into model deployment in AWS, see Chapter 7.

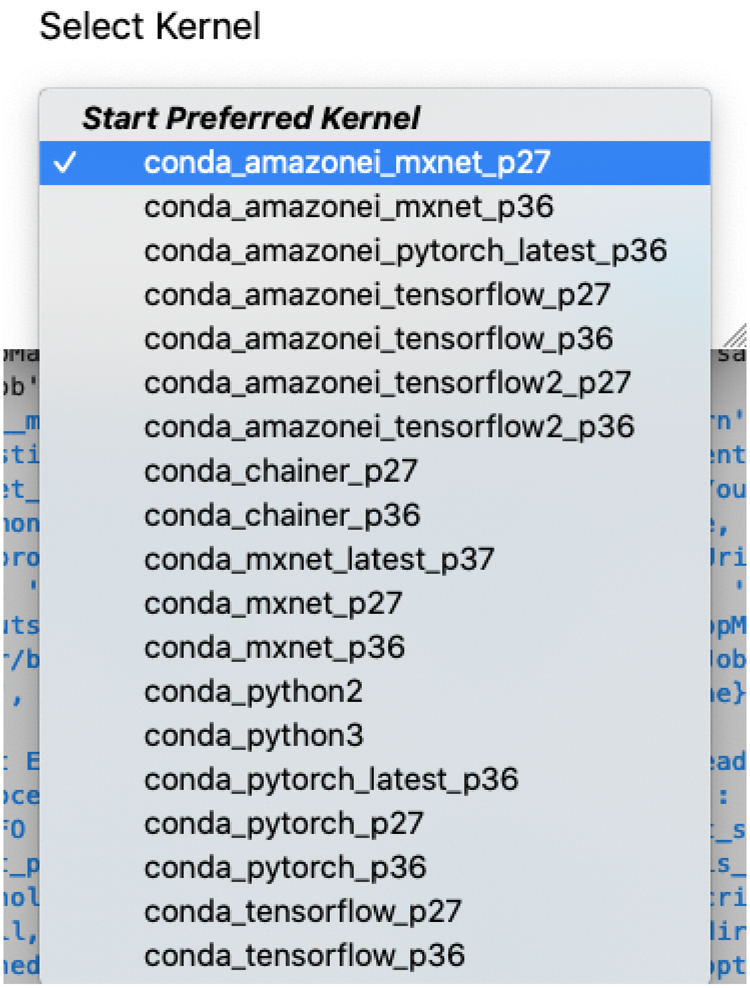


Figure 6-3. SageMaker kernels

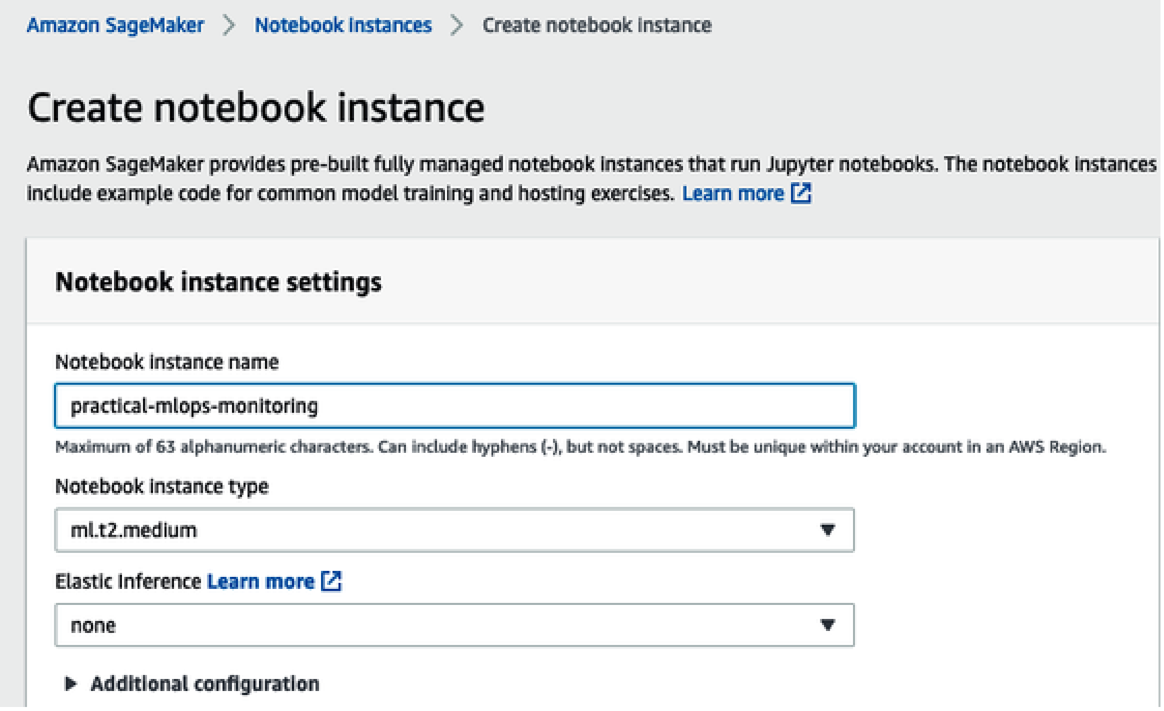
Use a SageMaker notebook for this section. Log in to the AWS console, and find the SageMaker service. Once that is loaded, on the left column, find the Notebook Instances link and click it. Create a new instance with a meaningful name. There is no need to change any of the defaults, including the machine type. I've named my notebook *practical-mlops-monitoring* (see Figure 6-4).

When deploying the model, it is important to enable capturing of data. So make sure you use the `DataCaptureConfig` class to do so. This is a quick example that will save it to an S3 bucket:

```
from sagemaker.model_monitor import DataCaptureConfig

s3_capture_path = "s3://monitoring/xgb-churn-data"
```

```
data_capture_config = DataCaptureConfig(
    enable_capture=True,
    sampling_percentage=100,
    destination_s3_uri=s3_capture_path
)
```



Amazon SageMaker > Notebook instances > Create notebook instance

Create notebook instance

Amazon SageMaker provides pre-built fully managed notebook instances that run Jupyter notebooks. The notebook instances include example code for common model training and hosting exercises. Learn more

Notebook instance settings

Notebook instance name

practical-mlops-monitoring

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

Notebook instance type

ml.t2.medium

Elastic Inference Learn more

none

▶ Additional configuration

Figure 6-4. SageMaker notebook instance

Use the `data_capture_config` when calling `model.deploy()`. In this example, I've previously created a `model` object using the `Model()` class, and assigned the data capture configuration to it, so that when the model gets exercised, the data gets saved to the S3 bucket:

```
from sagemaker.deserializers import CSVDeserializer

predictor = model.deploy(
    initial_instance_count=1,
    instance_type="ml.m4.large",
    endpoint_name="xgb-churn-monitor",
    data_capture_config=data_capture_config,
    deserializer=CSVDeserializer(),
)
```

After the model is deployed and available, send some requests to start making predictions. By sending requests to the model, you are causing the capturing configuration to save critical data needed to create the baseline. You can send prediction requests to the model in any way. In this example, I’m using the SDK to send some requests using sample CSV data from a file. Each row represents data that the model can use to start predicting. Since the input is data, that is the reason I’m using a CSV deserializer, so that the endpoint understands how to consume that input:

```python
from sagemaker.predictor import Predictor
from sagemaker.serializers import CSVDeserializer, CSVSerializer
import time

predictor = Predictor(
    endpoint_name=endpoint_name,
    deserializer=CSVDeserializer(),
    serializer=CSVSerializer(),
)


# About one hundred requests should be enough from test_data.csv
with open("test_data.csv") as f:
    for row in f:
        payload = row.rstrip("\n")
        response = predictor.predict(data=payload)
        time.sleep(0.5)
```

After running, double-check that there is output captured in the S3 bucket. You can list the contents of that bucket to ensure there is actual data. In this case, I’m going to use the AWS command line tool, but you can use the web interface or the SDK (the method doesn’t matter in this case):

```
$ aws s3 ls \
  s3://monitoring/xgb-churn-data/datacapture/AllTraffic/2021/02/03/13/
2021-02-03 08:13:33  61355 12-26-957-d5938b7b-fbd8-4e3c-9dbd-741f71b.jsonl
2021-02-03 08:14:33   1566 13-27-365-a59180ea-591d-4562-925b-6472d55.jsonl
2021-02-03 08:33:33  31548 32-24-577-20217dd9-8bfa-4ba2-a7f1-d9717ef.jsonl
2021-02-03 08:34:33  31373 33-25-476-0b843e95-5fe0-4b79-8369-b099d0e.jsonl
[...]
```

The bucket lists about 30 items, confirming that the prediction requests were successful and that data was captured and saved to the S3 bucket. Each file has a JSON entry with some information. The specifics in each entry are hard to grasp. This is what one of the entries looks like:

```
{
  "captureData": {
    "endpointInput": {
      "observedContentType": "text/csv",
      "mode": "INPUT",
      "data": "92,0,176.3,85,93.4,125,207.2,107,9.6,1,2,0,1,00,0,0,1,1,0,1,0",
      "encoding": "CSV"
    },
  [...]
}
```

Once again, the entries reference the CSV content type throughout. This is crucial so other consumers of this data can consume the information correctly. So far, we configured the model to capture data and save it to an S3 bucket. This all happened after generating some predictions with test data. However, there is no baseline yet. The data captured in the previous step is required to create the baseline. The next step requires a target training dataset. As I've mentioned before, the training dataset can be the same one used to train the model. A subset of the dataset might be acceptable if the resulting model doesn't change drastically. This *target dataset* has to have the same features (and in the same order) as the dataset used to train the production model.

NOTE

It is common to find online documentation that refers to the *baseline dataset* interchangeably with *target dataset*, since both can initially be the same. This can make it confusing when trying to grasp some of these concepts. It is useful to think about the baseline dataset as the data used to create the gold standard (the baseline) and any newer data as the *target*.

SageMaker makes it easy to save and retrieve data by relying on S3. I've defined locations throughout the SDK examples already, and for the baselining, I will do the same. Start by creating a monitor object; this object is able to produce a baseline and save it to S3:

```
from sagemaker.model_monitor import DefaultModelMonitor

role = get_execution_role()

monitor = DefaultModelMonitor(
    role=role,
    instance_count=1,
    instance_type="ml.m5.xlarge",
    volume_size_in_gb=20,
    max_runtime_in_seconds=3600,
)
```

Now that the monitor is available, use the `suggest_baseline()` method to produce a default baseline for the model:

```
from sagemaker.model_monitor.dataset_format import DatasetFormat
from sagemaker import get_execution_role

s3_path = "s3://monitoring/xgb-churn-data"

monitor.suggest_baseline(
    baseline_dataset=s3_path + "/training-dataset.csv",
    dataset_format=DatasetFormat.csv(header=True),
    output_s3_uri=s3_path + "/baseline/",
    wait=True,
)
```

When the run completes, a lot of output will get produced. The start of the output should be similar to this:

```
Job Name:  baseline-suggestion-job-2021-02-03-13-26-09-164
Inputs:  [{'InputName': 'baseline_dataset_input', 'AppManaged': False, ...}]
Outputs:  [{'OutputName': 'monitoring_output', 'AppManaged': False, ...}]
```

There should be two files saved in the configured S3 bucket: *constraints.json* and *statistics.json*. You can visualize the constraints with the Pandas library:

```
import pandas as pd

baseline_job = monitor.latest_baselining_job
constraints = pd.json_normalize(
    baseline_job.baseline_statistics().body_dict["features"]
)
schema_df.head(10)
```

This is a short subset of the constraints table that Pandas generates:

```
name          inferred_type   completeness  num_constraints.is_non_negative
Churn           Integral  1.0           True
Account Length  Integral  1.0           True
Day Mins  Fractional  1.0           True
[...]
```

Now that we have a baseline made with a very similar dataset to the one used to train the production model, and we have captured relevant constraints, it is time to analyze it and monitor data drift. There have been several steps involved so far, but most of these steps will not change much from these examples to other more complex ones, which means there are many opportunities here to automate and abstract a lot. The initial collection of data happens once when setting the baseline, and then it shouldn’t change unless changes to the baseline are needed. These will probably not happen often, so the heavy lifting of setting the baseline shouldn’t feel like a burden.

To analyze the collected data, we need a monitoring schedule. The example schedule will run every hour, using the baseline created in the previous steps to compare against traffic:

```
from sagemaker.model_monitor import CronExpressionGenerator

schedule_name = "xgb-churn-monitor-schedule"
s3_report_path = "s3://monitoring/xgb-churn-data/report"

monitor.create_monitoring_schedule(
    monitor_schedule_name=schedule_name,
    endpoint_input=predictor.endpoint_name,
    output_s3_uri=s3_report_path,
    statistics=monitor.baseline_statistics(),
    constraints=monitor.suggested_constraints(),
    schedule_cron_expression=CronExpressionGenerator.hourly(),
    enable_cloudwatch_metrics=True,
)
```

Once you create the schedule, it will require traffic to generate reports. If the model is already in a production environment, then we can assume (and reuse) existing traffic. If you are testing out the baseline on a test model like I've done in these examples, you will need to generate traffic by invoking a request to the deployed model. A straightforward way to generate traffic is to reuse the training dataset to invoke the endpoint.

The model I deployed ran for a few hours. I used this script to generate some predictions from the previously deployed model:

```
import boto3
import time

runtime_client = boto3.client("runtime.sagemaker")


with open("training-dataset.csv") as f:
    for row in f:
        payload = row.rstrip("\n")
        response = runtime_client.invoke_endpoint(
            EndpointName=predictor.endpoint_name,
            ContentType="text/csv",
            Body=payload
        )
        time.sleep(0.5)
```

Since I configured the monitoring schedule to capture every hour, SageMaker will not immediately generate reports onto the S3 bucket. After two hours, it is possible to list the S3 bucket and check if reports appear in there.

NOTE

Although the model monitor will run hourly, AWS has a 20-minute buffer that may cause a delay of up to 20 minutes after the hour mark. If you’ve seen other scheduling systems, this buffer might be surprising. This happens because, behind the scenes, AWS is load balancing the resources for scheduling.

The reports consist of three JSON files:

- *contraint_violations.json*
- *contraint.json*
- *statistics.json*

The interesting information related to monitoring and capturing drift is in the *constraint_violations.json* file. In my case, most of the violations look like this entry:

```
feature_name:   State_MI
constraint_check_type:  data_type_check
description:
Data type match requirement is not met. Expected data type: Integral,
Expected match: 100.0%.  Observed: Only 99.71751412429379% of data is Integral.
```

The suggested baseline required 100% of data integrity, and here we are seeing that the model is close enough at 99.7%. Because the constraint is to meet 100%, the violation gets generated and reported. Most of those numbers are similar in my case except for one row:

```
feature_name:   Churn
constraint_check_type:  data_type_check
description:
Data type match requirement is not met. Expected data type: Integral,
Expected match: 100.0%. Observed: Only 0.0% of data is Integral.
```

A 0% is a critical situation here, and this is where all the hard work of setting up the systems to catch and report these changes in predictions is crucial. I want to emphasize that although it took several steps and boilerplate code with the AWS Python SDK, it isn't too complicated to automate and start generating these reports automatically for target datasets. I created the baseline with an automated suggestion, and this will mostly require fine-tuning to define acceptable values to prevent generating violations that are not useful.

Monitoring Drift with Azure ML

MLOps plays a significant role in cloud providers. It isn't surprising that monitoring and alerting (core pillars of DevOps) get offered with well-thought-out services. Azure can analyze data drift and set alerts to capture potential issues before models get into production. It is always useful to see how different cloud providers solve a problem like data drift—perspective is an invaluable asset and will make you a better engineer. The amount of thought, documentation, and examples in the Azure platform makes for a smoother onboarding process. It doesn't take much effort to find learning resources to get up to speed with Azure's offerings.

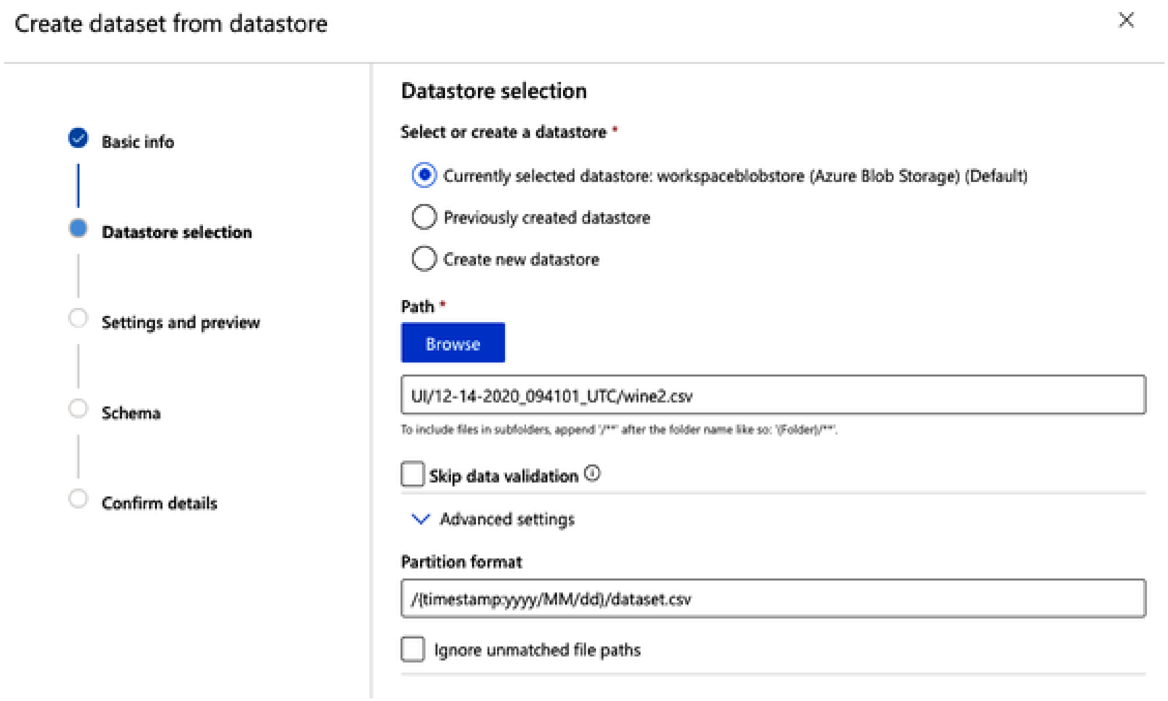
> **NOTE**
>
> At the time of this writing, data drift detection on Azure ML is on preview, and some minor issues still need to get worked out that prevent giving solid code examples to try out.

Data drift detection in Azure works similarly to using AWS SageMaker. The objective is to be alerted when drift happens between training and serving datasets. As with most ML operations, it is critical to have a deep understanding of the data (and, therefore, the dataset). An overly simplistic example is a dataset that captures swimsuit sales over a year: if the number of sales per week drops to zero, does that mean the dataset has drifted and should not be used in production? Or that it is probably the middle of winter and no one is buying anything? These open questions are easy to answer when the details of the data are well understood.

There are several causes for data drift, many of which can be wholly unacceptable. Some examples of these causes involve changes in value types (e.g., Fahrenheit to Celsius), empty or null values, or in the example of the swimsuit sale, a *natural drift*, where seasonal changes can affect predictions.

The patterns for setting up and analyzing drift on Azure require a baseline dataset, a target dataset, and a monitor. These three requirements work together to produce metrics and create alerts whenever drift is detected. The monitoring and analysis workflow allows you to detect and alert data drift when there is new data in a dataset while allowing profiling new data over time. You will probably not use historical profiling as much as current drift detection because it is more common to use the most current comparison point than checking against several months ago. It is still useful to have as a comparator where the previous year's performance is more meaningful than comparing against last month. This is particularly true with datasets that are affected by seasonal events. There is not much use in comparing Christmas tree sales against August metrics.

To set up a data drift workflow in Azure, you must start by creating a target dataset. The target dataset requires a *time-series* set on it either using a timestamp column or a *virtual column*. The virtual column is a nice feature because it infers the timestamp from the path where the dataset is stored. This configuration attribute is called the *partition format*. And if you are configuring a dataset to use the virtual column, you will see a partition format referenced in both Azure ML Studio and the Python SDK (see Figure 6-5).



Create dataset from datastore

Datastore selection

Select or create a datastore *

● Currently selected datastore: workspaceblobstore (Azure Blob Storage) (Default)

○ Previously created datastore

○ Create new datastore

Path *

UI/12-14-2020_094101_UTC/wine2.csv

To include files in subfolders, append '/**' after the folder name like so: '{Folder}/**'.

□ Skip data validation

Advanced settings

Partition format

/{timestamp:yyyy/MM/dd}/dataset.csv

□ Ignore unmatched file paths

Figure 6-5. Azure partition format

In this case, I'm using a partition format helper so that Azure can use the path to infer the timestamp. This is nice because it instructs Azure that the convention is setting the standard. A path like */2021/10/14/dataset.csv* means that the dataset will end up with a virtual column of October 14th, 2021. Configuration by convention is a golden nugget of automation. Whenever you see an opportunity to abstract (or entirely remove) configuration that can be inferred by a convention (like a path in this case), you should take advantage of it. Less configuration means less overhead, which enables speedy workflows.

Once you have a time-series dataset, you can proceed by creating a dataset monitor. To get everything working, you will need a target dataset (in this case, the time-series dataset), the baseline dataset, and the monitor settings.

The baseline dataset has to have the same features (or as similar as possible) as those of the target dataset. One exciting feature is selecting a time range to slice the dataset with relevant data for the monitoring task. The monitor's configuration is what brings all the datasets together. It allows you to create a schedule to run with a set of exciting features and set the threshold for the data drift percentage tolerable.

The drift results will be available at the Dataset Monitors tab in the Assets section in Azure ML Studio. All the configured monitors are available with highlighted metrics about drift magnitude and a sorted list of the top features with drift. The simplicity of exposing data drift is one thing I like about Azure ML Studio because it can quickly surface the essential pieces of information useful to take corrective decisions.

If there is a need to go deep into the metrics' details, you can use Application Insights to query logs and metrics associated with the monitors. Enabling and setting up Application Insights is covered in "Application Insights".

Conclusion

Logging, monitoring, and metrics are so critical that any model getting shipped into production *must implement them* at the risk of hard-to-recover catastrophic failure. High confidence in robust processes for reproducible results requires all of these components. As I've mentioned throughout this chapter, you must make decisions with accurate data that can tell you if accuracy is as high as ever or if the number of errors has increased significantly.

Many examples can be tricky to grasp, but they can all be automated and abstracted away. Throughout this book, you will consistently read about DevOps' core pillars and how they are relevant to operationalizing machine learning. Automation is what binds pillars like logging and monitoring together. Setting up logging and monitoring is usually not exciting work, especially if the idea is to get state-of-the-art prediction models doing exceptional work. But exceptional results cannot happen consistently with a broken foundation.

When I first started training to be a professional athlete, I always doubted my coach, who didn't allow me to do the High Jump every day. *"Before becoming a High Jumper, you must become an athlete."* Strong foundations enable strong results, and in DevOps and MLOps, this is no different.

In the next chapters, you get a chance to dive deeper into each of the three major cloud providers, their concepts, and their machine learning offerings.

Exercises

- Use a different dataset and create a drift report with violations using AWS SageMaker.
- Add Python logging to a script that will log errors to `STDERR`, info statements to `STDOUT`, and all levels to a file.
- Create a time-series dataset on Azure ML Studio.
- Configure a dataset monitor in Azure ML Studio that will send an email when a drift is detected beyond the acceptable threshold.

Critical Thinking Discussion Questions

- Why might it be desirable to log to multiple sources at the same time?
- Why is it critical to monitor data drift?
- Name three advantages of using logging facilities versus `print()` or `echo` statements.
- List the five most common log levels, from least to most verbose.
- What are three common metric types found in metric-capturing systems?