Chapter 16. Model Monitoring and Logging

By now, you should be familiar with the MLOps modeling lifecycle, as shown in Figure 16-1, which starts with building your models but doesn't end with deployment.

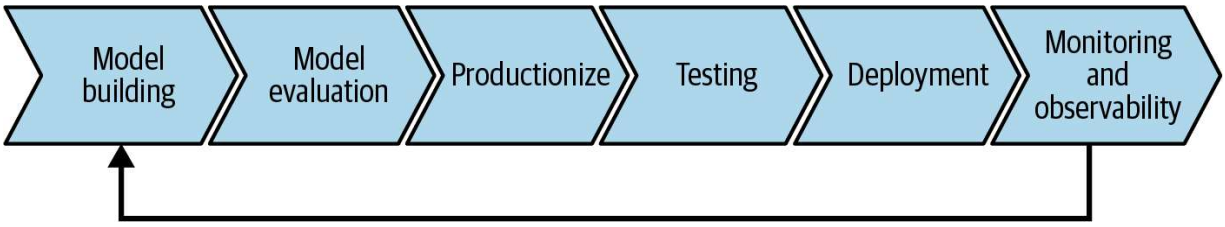

Figure 16-1. The MLOps lifecycle

The last task, monitoring your model in production, is an ongoing task for as long as your model is in production. The data you gather by monitoring will guide how you build the next version of your model and make you aware of changes in your data and changes in your model performance. So, as you can see in Figure 16-1, this is a cyclical, iterative process that requires the last step, monitoring, in order to be complete.

You should note here that this diagram is only looking at monitoring that is directly related to your model performance, and you will also need to include monitoring of the systems and infrastructure that are included in your entire product or service, such as databases and web servers. That kind of monitoring is only concerned with the basic operation of your product or

service, and not the model itself, but it’s critical to your users’ experience. Basically, if the system is down, it really doesn’t matter how good your model is.

The Importance of Monitoring

> *An ounce of prevention is worth a pound of cure.*
>
> —Benjamin Franklin

In 1733, Benjamin Franklin visited Boston and was impressed with the fire prevention measures the city had established, so when he returned to his home in Philadelphia he tried to get his city to adopt similar measures. Franklin was talking about preventing actual fires, but in our case, you might apply this same idea to preventing fire drills, the kinds where your system is performing poorly and it’s suddenly an emergency to fix it. These are the kinds of fire drills that can happen if you don’t monitor your model performance.

If your training data is too old, even when you first deploy a new model, you can have immediate data skews. If you don’t monitor right from the start, you may be unaware of the problem, and your model will not be accurate, even when it’s new.

Of course, as we previously discussed, models will also become *stale*, or inaccurate, because the world constantly changes and the training data you originally collected might no longer reflect the current state. Again, without monitoring, you are unlikely to be aware of the problem.

You can also have negative feedback loops. This turns out to be a complex issue that arises when you automatically train models on data collected in production. If this data is biased or corrupted in any way, the models trained on that data will perform poorly. Monitoring is important even for automated processes, because they too can have problems.

ML monitoring or *functional monitoring* deals with keeping an eye on model predictive performance and on changes in serving data. This type of monitoring looks at the metrics the model optimized during training and the distributions and characteristics of each feature in the serving data.

System monitoring or *nonfunctional monitoring* refers to monitoring the performance of the entire production system, the system status, and the reliability of the serving system. This includes things like the queries per second, failures, latencies, and resource utilization.

ML monitoring is different from traditional system monitoring. Unlike a more traditional software system, there are two additional components to consider in an ML system: the data and the model. Unlike in traditional software systems, the accuracy of an ML system depends on how well the

model reflects the world it is meant to model, which in turn depends on the data used for training and the data it receives while serving requests. It's not simply a matter of monitoring for system failures such as segmentation faults, out-of-memory conditions, or network connectivity issues. The model and the data require additional, very specialized monitoring as well.

Code and configuration also take on additional complexity and sensitivity in an ML system due to two aspects of the ML system: entanglement and configuration. With *entanglement* (and we're not referring to quantum entanglement), changing one thing changes everything. Here you need to be careful with feature engineering and feature selection, and you need to understand your model's sensitivity. Configuration can also be an issue because model hyperparameters, versions, and features are often controlled in a system config, and the slightest error here can cause radically different model behavior that won't be picked up with traditional software tests—again requiring additional, very specialized monitoring.

Observability in Machine Learning

Observability measures how well you can infer the internal states of a system by only knowing the inputs and outputs. For ML, this means monitoring and analyzing the prediction requests and the generated predictions from your models.

Observability isn’t a new concept. It actually comes from control system theory, where it has been well established for decades. In control system theory, observability and controllability are closely linked. You can only control a system to the extent that you can observe it. Looking at an ML-based product or service, this maps to the idea that controlling the accuracy of the results overall, usually across different versions of the model, requires observability. The need for observability also adds to the importance of model interpretability.

In ML systems, observability becomes a complex problem, since you need to consider monitoring and aggregating multiple interacting systems and services, such as cloud deployments, containerized infrastructure, distributed systems, and microservices. Often, this means relying on vendor monitoring systems to collect and sometimes aggregate data, because the observability of each instance can be limited. For example, monitoring CPU utilization across an autoscaling containerized application is much different than simply monitoring CPU usage on a single server.

Observability is about making measurements, and just like when you’re analyzing your model performance during training, measuring top-level metrics is not enough; it will provide an incomplete picture. You need to slice your data to understand how your model performs for various data subsets. For example, in an autonomous vehicle, you need to understand performance in both rainy and sunny conditions, and measure them

separately. More generally speaking, data slices provide a useful way to analyze different groups of people or different types of conditions.

This means domain knowledge is important in observing and monitoring your systems in production, just like it is when you're training your models. In general, it's your domain knowledge that will guide how you slice your data.

The TFX framework and TensorFlow Model Analysis are very powerful tools, and they include functionality for doing observability analysis on multiple slices of data for your deployed models. This is true for both supervised and unsupervised monitoring of your models. In a supervised setting, the true labels are available to measure the accuracy of your predictions. In an unsupervised setting, you will monitor for things like the means, medians, ranges, and standard deviations of each feature. In both supervised and unsupervised settings, you need to slice your data to understand how your system behaves for different subsets. Going back to the autonomous vehicle example, slicing by weather conditions is important to avoid things like making poor driving decisions in the rain.

The main goal of observability in the context of monitoring is to prevent or act upon system failures. For this, the observations need to provide alerts when a failure happens, and ideally they should provide recommended actions to bring the system back to normal behavior. More specifically, *alertability* refers to designing metrics and thresholds that make it very

clear when a failure happens. This may include defining rules to link more than one measurement to identify a failure.

Knowing that your system is failing is a good start, but an actionable recommendation based on the nature of the failure is much more helpful to correct this behavior. Ideally, actionable alerts should clearly identify the root cause of the system's failure. At a bare minimum, your system should gather sufficient information to enable root cause analysis. Both alertability and actionability are goals, and the effectiveness of your system is a reflection of how well it achieves those goals.

What Should You Monitor?

Starting with the basics, you can monitor the inputs and outputs of your system. Statistical testing and comparisons are the basic tools you can use to analyze your inputs and outputs. Typical descriptive statistics include median, mean, standard deviation, and range values.

The inputs in a deployed system are the prediction requests, each of which is a feature vector. You can use statistical measures of each feature, including their distributions, and look for changes that may be associated with failures. Again, this should not just be top-level measurements, but measurements on slices that are relevant to your domain.

The prediction requests, whether you're doing real-time or batch predictions, form a large part of the observable data you have for a

deployment. For each feature, you should monitor for errors such as values falling outside an allowed range or set of categories, where these error conditions are often defined based on domain knowledge. You should also monitor how each feature distribution changes over time and compare those to the training data. Monitoring for errors and changes is better done with sliced data so that you can better understand and identify potential system failures.

The outputs are the model's predictions, which you can also monitor and measure. This should include an understanding of the deployment of different model versions to help you understand how different versions perform. You should also consider performing correlation analysis to understand how changes in your inputs affect your model outputs, and again this should be done on slices of your data. For example, correlation analysis can help you detect how seemingly harmless changes in your inputs cause prediction failures.

In some scenarios, such as predicting click-through where labels are available, you can also do comparisons between known labels and model predictions. It's also important to consider that if you have altered the distributions of the training data to correct for things like class imbalance or fairness issues, you need to take that into account when comparing previous datasets to the distributions of the input data gathered through monitoring prediction requests.

Monitoring your model is not enough, however, since you need to keep a production system healthy. That requires system monitoring of your production infrastructure. Monitoring in the realm of software engineering is a far more well-established area. The operational concerns around an ML system in production may include monitoring system performance measures such as latency; I/O, memory, and disk utilization; or system reliability in terms of uptime. Monitoring can even happen while taking auditability into account.

In software engineering, talking about monitoring is, strictly speaking, talking about events. Events can be almost anything, including receiving an HTTP request, entering or leaving a function (which may or may not contain ML code), a user logging in, reading from network resources, writing to the disk, and so on. All of these events also have some context. To understand how your systems are performing in both technical and business terms, and for debugging, it would be ideal to have all of the event information available. But collecting all the context information is often not practical, as the amount of data to process and store could be very large.

Custom Alerting in TFX

In production, you may need to set up custom alerting for your training pipeline, for things like sending failure notifications or emails when the pipeline experiences a system failure. For pipelines that are running TFX on Vertex AI, TFX provides a way to define custom components that can be

triggered by a status change of the pipeline through the use of an exit handler. The component is triggered when the pipeline exits. When the pipeline status changes, including success, pending, or failure, the custom component will be triggered. This process is shown in Figure 16-2.

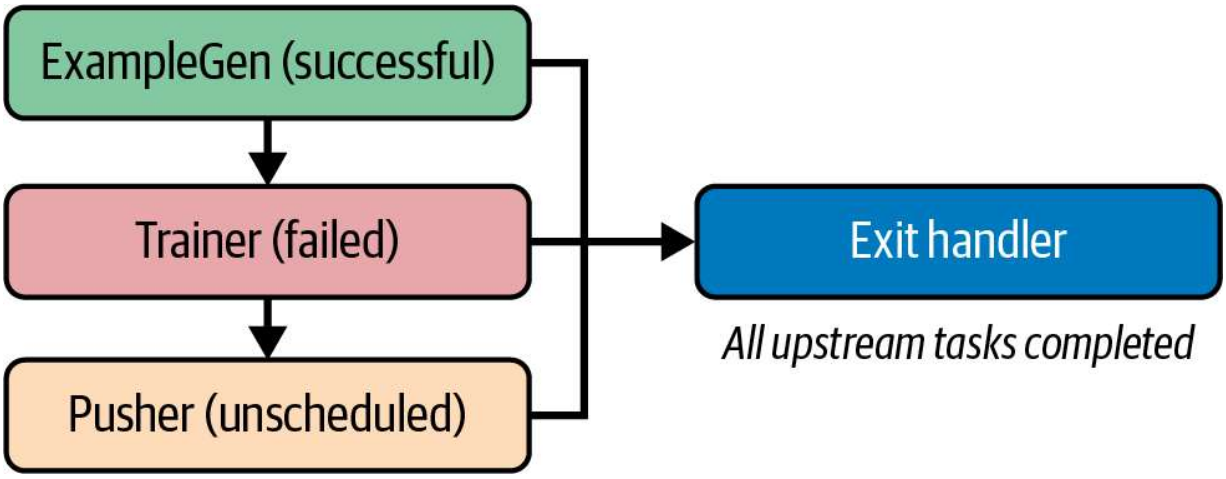

Figure 16-2. The TFX pipeline with an exit handler based on a different triggering rule

Defining an exit handler is similar to defining a custom component, using a special decorator named `exit_handler`. Following is pseudocode for defining an `exit_handler` in a pipeline:

```
from tfx.orchestration.kubeflow import decorators
import tfx.v1 as tfx

@decorators.exit_handler
def test_exit_handler(final_status: tfx.dsl.compo
// put custom logic for alerting
print('exit handler executing')
```

```
  # use the FinalStatusStr to define examples
  exitHandler = ExitHandler(
      final_status=tfx.orchestration.experimental.F
  Pipeline = tfx.Pipeline(...)

  # Register the exit handler with Kubeflow
  kubeflow_v2_dag_runner.setExitHandler(exitHandler
  kubeflow_v2_dag_runner.run(pipeline = Pipeline)
```

Logging

To avoid making the same mistake twice, it's important to learn from history. This is where logging comes into play. A log is almost always the source of the data you will use to monitor your models and systems. A *log* is an immutable, timestamped record of discrete events that happened over time for your ML system, along with additional information.

Log messages are very easy to generate, since they are just a string, a blob of JSON, or typed key-value pairs. Event logs provide valuable insight along with context, offering detail that averages and percentiles don't surface. However, it's not always easy to give the right level of context without obscuring the really valuable information in too much extraneous detail.

While metrics show the trends of a service or an application, logs focus on specific events. This includes both log messages printed from your application, and warnings, errors, or debug messages generated automatically. The information in logs is often the only information available when investigating incidents and to help with root cause analysis.

Some red flags to watch out for in your logs may include basic things like a feature becoming unavailable. Catching this is especially important when you're including historical data in your prediction requests, which needs to be retrieved from a datastore. In other cases, notable shifts in the distribution of key input values are important—an example would be a categorical value that was relatively rare in the training data becoming more common. Patterns specific to your model—such as in an NLP scenario, a sudden rise in the number of words not seen in the training data—can be another sign of a potential change that can lead to problems.

But logging isn't perfect. For example, excessive logging can negatively impact system performance. As a result of these performance concerns, aggregation operations on logs can be expensive, and for this reason, alerts based on logs should be treated with caution. Raw logs should be normalized, filtered, and processed by a tool such as Fluentd, Scribe, Logstash, or Heka before being persisted in a datastore such as Elasticsearch or BigQuery. Setting up and maintaining this tooling requires effort and discipline, which can be avoided by using managed services.

You could start with the out-of-the-box logs and metrics. These will usually give you some basic overall monitoring capabilities, which you can then add to. For example, in Google's Compute Engine platform, if you need additional application logs, you can install agents to collect those logs. Google Cloud Monitoring collects metrics from all the cloud services by default, which you can use to build dashboards. When you need additional application- or business-level metrics, you can use those custom metrics to monitor over time. Using aggregate sinks and workspaces allows you to centralize your logs from many different sources or services to create a unified view of your application.

Cloud providers also offer managed services for logging of cloud-based distributed services. These include Google Cloud Monitoring, Amazon CloudWatch, and Azure Monitor, as well as several managed offerings from third parties.

Log data is, of course, also the basis for your next training dataset. At the very least, collecting prediction requests should provide the feature vectors that are representative of the current state of the world your application lives in, so this data is very valuable. If possible, you should also capture any available data that shows what the correct label should be for a prediction request. For example, if you are trying to predict click-through, you should capture what the user actually clicked on. What's most important here is that you capture this valuable data so that you can keep your model in sync with a changing world.

Distributed Tracing

Tracing focuses on monitoring and understanding system performance, especially for microservices-based applications. Tracing is a part of system monitoring, since it does not analyze changes in data or model results.

With a distributed system, suppose you're trying to troubleshoot a prediction latency problem. Imagine that your system is made of many independent services, and the prediction is generated through many downstream services. You have no idea which of those services are causing the slowdown. You have no clear understanding of whether it's a bug, an integration issue, a bottleneck due to a poor choice of architecture, or poor networking performance.

In monolithic systems, it's relatively easy to collect diagnostic data from the different parts of a system. All modules might even run within one process and share common resources for logging.

Solving this problem becomes even more difficult if your services are running as separate processes in a distributed system. You cannot depend on the traditional approaches that helped diagnose monolithic systems. You need to have finer-grained visibility into what's going on inside each service and how the services interact with one another over the lifetime of a user request. It becomes harder to follow a call starting from the frontend web server to all its backends until a prediction is returned to the user.

Now imagine that your architecture scales dynamically to reflect load. Systems come and go as needed, so looking back in the tracing data requires you to also track how many and which systems were running at any given time.

To properly inspect and debug issues with latency for requests in distributed systems, you need to understand the sequencing and parallelism of the services, and the latency contribution of each, to the final latency of the system.

To address this problem, Google developed the distributed tracing system Dapper to instrument and analyze its production services. Google's technical report on Dapper has inspired many open source projects, such as Zipkin and Jaeger, and Dapper-style tracing, as shown in Figure 16-3, has emerged as an industry-wide standard.

In service-based architectures, Dapper-style tracing works by propagating tracing data between services. Each service annotates the trace with additional data and passes the tracing header to other services until the final request completes. Services are responsible for uploading their traces to a tracing backend. The tracing backend then puts related latency data together like the pieces of a puzzle. Tracing backends also provide UIs to analyze and visualize traces.

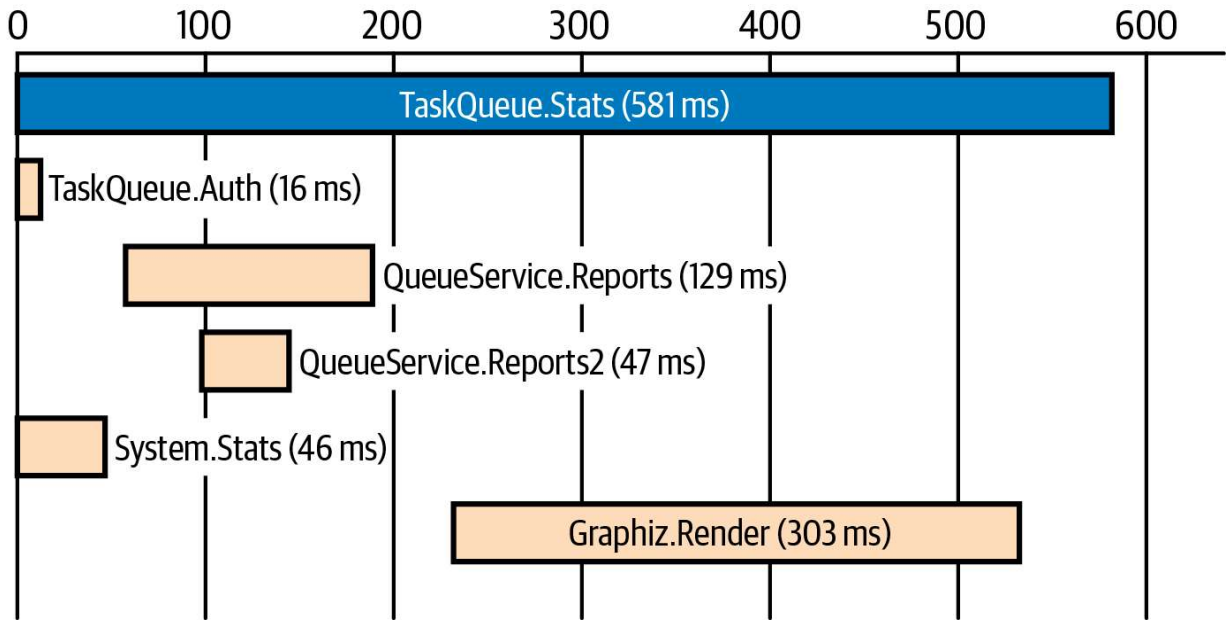

Figure 16-3. Dapper-style tracing (ms)

Each trace is a call tree, beginning with the entry point of a request and ending with the server's response, including all remote procedure calls (RPCs) along the way. Each trace consists of small units called *spans*. In Figure 16-3, the whole trace for TaskQueue.Stats takes 581 ms to complete. TaskQueue.Stats makes calls to five other services, creating five spans, each of which contributes to the time required for TaskQueue.Stats to run. Often, those calls are RPCs.

Monitoring for Model Decay

One of the key problems in many domains is *model decay*. Detecting model decay is an important part of ML monitoring or functional monitoring, since it's directly concerned with the data and model results that your system is

designed to consume and produce. Understanding your model decay is a key part of designing processes to prevent it before it impacts your results in unacceptable ways.

Production ML models often operate in dynamic environments. Over time, dynamic environments change. That's what makes them dynamic. Think of a recommender system, for example, that is trying to recommend which music to listen to. Music changes constantly, with new music becoming popular and tastes changing.

If the model is static and continues to recommend music that has gone out of style, the quality of the recommendations will decline. The model is moving away from the current ground truth, the current reality. It doesn't understand the current styles, because it hasn't been trained for them.

Data Drift and Concept Drift

There are two primary causes of model decay: data drift and concept drift. *Data drift* occurs when statistical properties of the inputs (the features) change. As the input changes, the prediction requests (the input) move farther and farther away from the data that the model was trained with, and the model accuracy suffers.

Changes like these often occur in demographic features such as age, which may change over time. The graph in Figure 16-4 shows how there is an increase in mean and variance for the age feature. This is data drift.

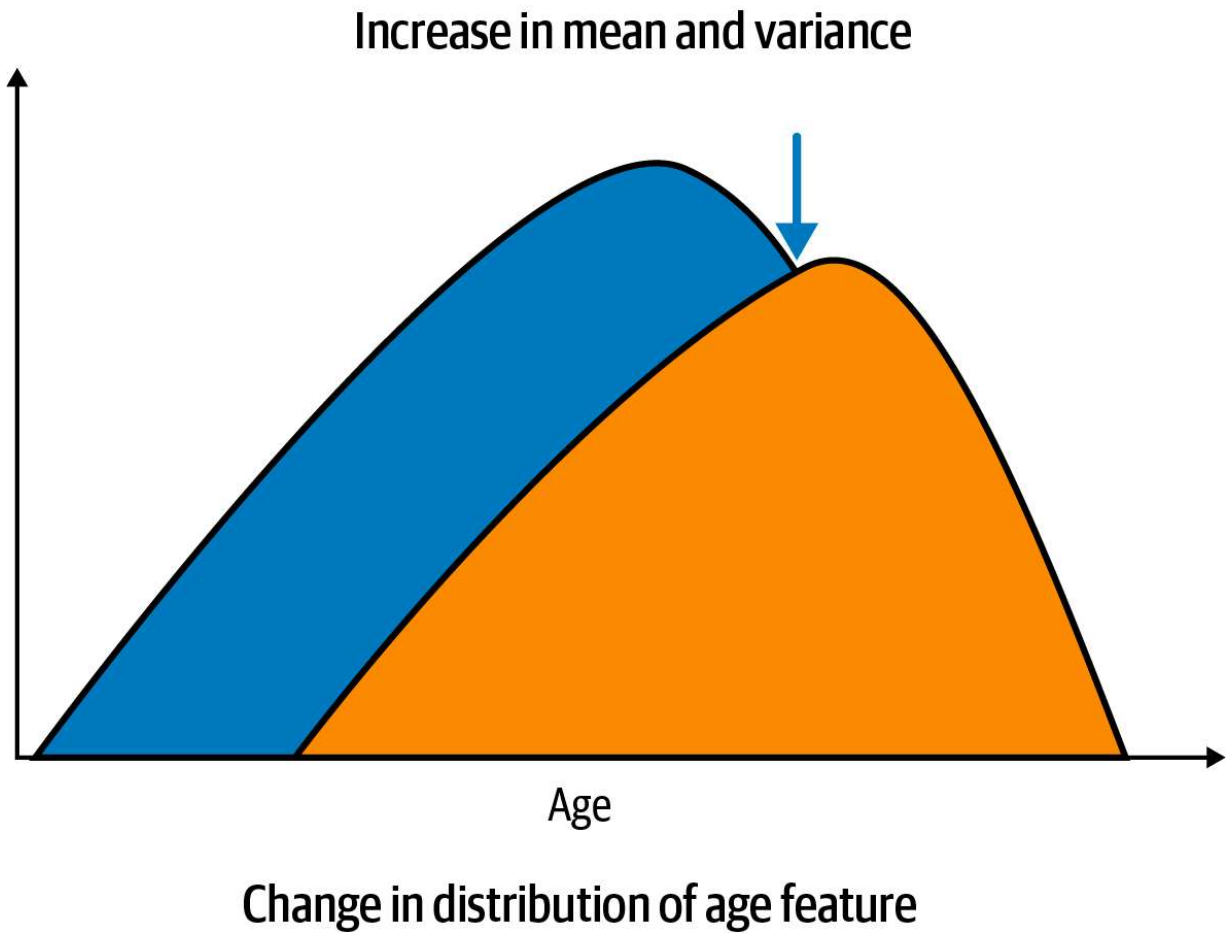

Figure 16-4. An example of data drift

Concept drift, as shown in Figure 16-5, occurs when the relationship between the features and the labels changes. When a model is trained, it learns a relationship between the inputs and ground truth, or labels.

If the relationship between the inputs and the labels changes over time, it means that the very meaning of what you are trying to predict changes. The world has changed, but your model doesn't know it. For example, take a look at the graph in Figure 16-5. You can see that the distribution of the features for the two classes, the dark and light dots, changes over time

intervals T1, T2, and T3. If your model is still predicting for T1 when the world has moved to T3, many of its predictions will be incorrect.

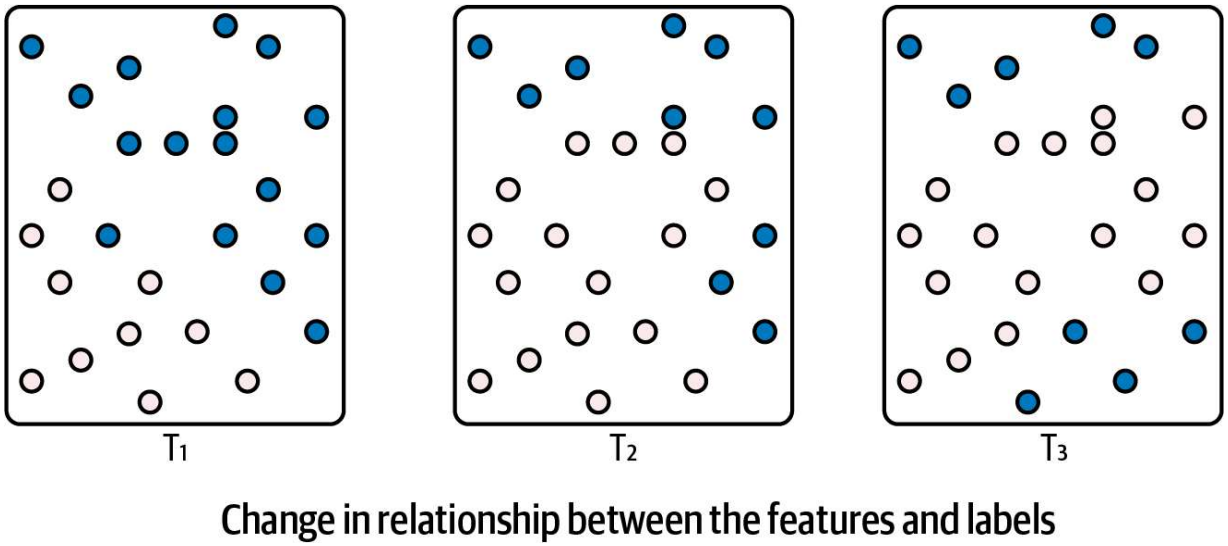

Figure 16-5. An example of concept drift

If you don’t plan ahead for drift, it can slowly creep into your system over time. How quickly your system drifts depends on the nature of the domain you’re working in. Some domains, such as markets, can change within hours or even minutes. Others change more slowly.

There is also the idea of an emerging concept. An *emerging concept* refers to new patterns in the data distribution that weren’t previously present in your dataset. This can happen in several ways. Labels may also have become obsolete and new labels may need to be added as the world changes.

Based on the type of distribution change, the dataset shift can be classified into two types: covariate shift and prior probability shift. In *covariate shift*, the distribution of your input data changes, but the conditional probability of output over input remains the same—the distribution of your labels doesn't change. *Prior probability shift* is basically the opposite of covariate shift. The distribution of your labels changes, but your input data stays the same. Concept drift can be thought of as a type of prior probability shift.

If drift, either data drift or concept drift or both, is not detected, your model accuracy will suffer and you won't be aware of it. This can lead to emergency retraining of your model, which is something to avoid. So, monitoring and planning ahead are important. Knowing that you've planned ahead and have systems in place just might make it easier for you to sleep at night.

Model Decay Detection

Detecting decay, whether it's the result of data drift or concept drift or both, starts with collecting current data. You should collect all the data in the incoming prediction requests to your model, along with the predictions that your model makes.

If it's possible in your application, also collect the correct label or ground truth that your model should have predicted. This is also extremely valuable for retraining your model. But at a minimum, you should capture the

prediction request data, which you can use to detect data drift using unsupervised statistical methods.

The process is really fairly straightforward. Once you're set up to continuously monitor and log your data, you employ tools that use well-known statistical methods to compare your current request data with your previous training data.

You can also use dashboards to monitor for trends and seasonality over time. Essentially, you'll be working with time series data, since your requests are ordered data that is associated with a time component. This is especially true with online serving of requests, but it is generally also true for batch processing. And you don't have to reinvent the wheel here; there are good tools and libraries available to help you do this kind of analysis. These include TensorFlow Data Validation (TFDV) and the scikit-multiflow library.

Cloud providers including Google offer managed services such as Google's Vertex Prediction that help you perform continuous evaluation of your prediction requests. Continuous evaluation helps catch problems early by regularly sampling prediction input and output from trained ML models that you have deployed to Vertex Prediction. If necessary, the Vertex Data Labeling Service can then assign actual people to assign ground truth labels for your data. Alternatively, you can provide your own labels. Azure, AWS, and other cloud providers offer similar services.

Supervised Monitoring Techniques

If your dataset is labeled, and if you're able to generate new labels for a sample of your incoming requests, then supervised monitoring techniques are a robust method for monitoring.

Statistical process control

One supervised technique is statistical process control (SPC). Statistical process control has been used in manufacturing for quality control since the 1920s. It uses statistical methods to monitor and control a process, which in the case of your deployed model is the incoming stream of raw data for prediction requests. This is useful to detect drift.

SPC assumes that the stream of data will be stationary (which it may or may not be, depending on your application) and that the errors follow a binomial distribution. It analyzes the rate of errors, and since it's a supervised method, it requires us to have labels for our incoming stream of data. Essentially, this method triggers a drift alert if the parameters of the distribution go beyond a certain threshold.

Sequential analysis

Another supervised technique is sequential analysis. In sequential analysis, we use a method called *linear four rates*. The basic idea is that if data is stationary, the contingency table should remain constant.

The contingency table in this case corresponds to the truth table for a classifier that you're probably familiar with: true positive, false positive, false negative, and true negative. You use those to calculate the four rates: net predictive value, precision, recall, and specificity. If the model is predicting correctly, these four values should continue to remain fairly constant.

Error distribution monitoring

The last supervised technique we'll review here is error distribution monitoring. We'll only discuss one method of choice here, known as adaptive windowing, although you should be aware that there are other methods.

In adaptive windowing, you divide the incoming data into windows, the size of which adapts to the data. Then, you calculate the mean error rate at every window of data. Next, you calculate the absolute difference of the mean error rate at every successive window and compare it with a threshold based on Hoeffding's bound. Hoeffding's bound is used for testing the difference between the means of two populations.

Unsupervised Monitoring Techniques

The main problem with supervised techniques is that you need to have labels, and generating labels can be expensive and slow. In unsupervised

techniques, you don’t need labels. Note that you can also use unsupervised techniques in addition to supervised techniques, even when you do have labeled data.

Clustering

Let’s start with clustering, or novelty detection. In this method, you cluster the incoming data to one of the known classes. If you see that the features of the new data are far away from the features of known classes, you know you’re seeing an emerging concept.

Based on the type of clustering you choose, there are multiple algorithms available. These include OLINDDA, MINAS, ECSMiner, and GC3, but the details of these algorithms are beyond the scope of this discussion.

While the visualization and ease of working with clustering work well with low-dimensional data, the curse of dimensionality kicks in once the number of dimensions grows significantly. Eventually, these methods start to become inefficient, but you can use dimensionality reduction techniques such as principal component analysis (PCA) to help make them manageable. However, this is the only method that helps you in detecting emerging concepts. One downside of this method is that it detects only cluster-based drift and not population-based changes.

Feature distribution monitoring

In feature distribution monitoring, you monitor each feature of the dataset separately. You split the incoming dataset into uniformly sized windows and then compare the individual features against each window of data.

There are multiple algorithms available to do the comparison, including Linear Four Rates (LFR) and Hellinger Distance Drift Detection Method (HDDDM). Pearson correlation is used in the Change of Concept technique, while Hellinger distance is used in HDDDM to quantify the similarity between two probability distributions.

Similar to the case of clustering or novelty detection, if the curse of dimensionality kicks in, you can make use of dimensionality reduction techniques like PCA to reduce the number of features. The downside of HDDDM is that it is not able to detect population drift, since it only looks at individual features.

Model-dependent monitoring

This method monitors the space near the decision boundaries, or margins, in the latent feature space of your model. One of the algorithms used is Margin Density Drift Detection, or MD3.

Space near the margins, where the model has low confidence, matters more than in other places, and this method looks for incoming data that falls into

the margins. A change in the number of samples in the margin (the margin density) indicates drift. This method is very good at reducing the rate of false alarms.

Mitigating Model Decay

OK, so now you've detected drift, which has led to model decay. What can you do about it?

Let's start with the basics. When you detect model decay, you need to let others know about it. That means informing your operational and business stakeholders about the situation, along with some idea about how severe you think the drift has become. Then, you'll work on bringing the model back to acceptable performance.

First, try to determine which data in your previous training dataset is still valid, by using unsupervised methods such as clustering or statistical methods that look at divergence. Many options exist, including Kullback–Leibler (K–L) divergence, Jensen–Shannon (J–S) divergence, and the Kolmogorov–Smirnov (K–S) test. This step is optional, but especially when you don't have a lot of new data, it can be important to try to keep as much of your old data as possible.

Another option is to simply discard that part of your training dataset that was collected before a certain date, under the assumption that the age of the

data reflects the divergence, and then add your new data. Or, if you have enough newly labeled data, you can just create an entirely new dataset. The choice between these options will probably be dictated by the realities of your application and your ability to collect new labeled data.

Retraining Your Model

Now that you have a new training dataset, you have basically two choices for how to retrain your model. You can either continue training your model, fine-tuning it from the last checkpoint using your new data, or start over by reinitializing your model and completely retraining it. Either approach is valid, and the choice between these two options will largely be dictated by the amount of new data that you have and how far the world has drifted since the last time you trained your model. Ideally, if you have enough new data, you should try both approaches and compare the results.

When to Retrain

It’s usually a good idea to establish policies around when you’re going to retrain your model. There’s really no right or wrong answer here, so this will depend on what works in your particular situation. You could simply choose to retrain your model whenever it seems to be necessary. That includes situations where you’ve detected drift, but also situations where

you may need to make structural changes to your dataset, such as adding or removing class labels or features, for example.

You could also just retrain your model according to a schedule, whether it needs it or not. In practice, this is what many people do because it’s simple to understand and in many domains it works fairly well. It can, however, incur higher training and data gathering costs than necessary, or alternatively it can allow for greater model decay than might be ideal, depending on whether your schedule has your model training too often or not often enough. It also assumes that change in the world happens at a fairly steady rate, which is often not the case.

And finally, you might be limited by the availability of new training data. This is especially true in circumstances where labeling is slow and expensive. As a result, you may be forced to try to retain as much of your old training data for as long as possible, and avoid fully retraining your model.

Automated Retraining

Automating the process of detecting the conditions that require model retraining would be ideal. Automating would include being able to detect model performance degradation (or data drift), continuously collecting enough training data, and triggering retraining.

Of course, you can only retrain when sufficient data is available. Ideally, you also have continuous training, integration, and deployment set up as well, to make the process fully automated. For some domains, where change is fast and frequent retraining is required, these automated processes become requirements instead of luxuries.

When your model decays beyond an acceptable threshold, when the meaning of the variable you are trying to predict deviates significantly, or when you need to make changes such as adding or removing features or class labels, you might have to redesign your data preprocessing steps and model architecture. We like to think of this as an opportunity to make improvements.

You may have to rethink your feature engineering and feature selection to make your model work with the current data and retrain your model from scratch, rather than applying fine-tuning. You might have to investigate other potential model architectures (which we find is a lot of fun!). The point here is that no model lives forever, and periodically you need to go “back to the drawing board” and start over, applying what you’ve learned since the last time you updated your model.

Conclusion

The world changes. Delivering good results consistently over the life of your application requires monitoring your model and data and taking action when necessary to improve the results it generates. Although we have not discussed it in this chapter, this also applies in the world of generative AI (GenAI) and language modeling, where grounding requires keeping your model up-to-date with the latest news and other developments in the world. This chapter focused on that monitoring process (which includes logging) and discussed some of the actions you can take when your model performance declines.