CHAPTER 6

Building Inference Pipelines

If you've made it this far, you've already created MLOps infrastructure, build a feature store, designed and built an end to end training pipeline complete with MLFlow experiment tracking for reproducibility and model storage in the MLFlow model registry, and tried monitoring and logging. It might seem like you're almost done; however, we're still missing a critical piece of the MLOps puzzle: Once you've trained your model, what do you do with it?

This is such a critical piece of the MLOps lifecycle that it's surprising so many data scientists leave the design and construction of the inference pipeline to the last minute or bury it away as a backlog item. The reality is, the inference pipeline is one of the most important parts of any stochastic system because it's where you will actually use your model to make a prediction. The success or failure of your model depends on how well stakeholders are able to use your model and action upon it to make business decisions; when they need it and without an understanding of this stage of the lifecycle, your project is doomed to failure. Not only that, but it's the inference pipeline where you will store the model output to incorporate feedback loops and add monitoring and data drift detection, so you can understand the output of your model and be able to analyze its results.

A lot can go wrong as well, and if you aren't aware of how to measure data drift and production-training skew, then your model may fail when it hits production data.

© Dayne Sorvisto 2023
D. Sorvisto, *MLOps Lifecycle Toolkit*, https://doi.org/10.1007/978-1-4842-9642-4_6

In this chapter, we will look at how we can reduce the negative consequences of production-training skew and monitor the output of our model to detect changes in problem definition or changes in the underlying distribution of features. We will also take a detailed look at performance considerations for real-time and batch inference pipelines and design an inference API capable of supporting multi-model deployments and pulling models from a central model repository similar to an architecture described in Figure 6-1.

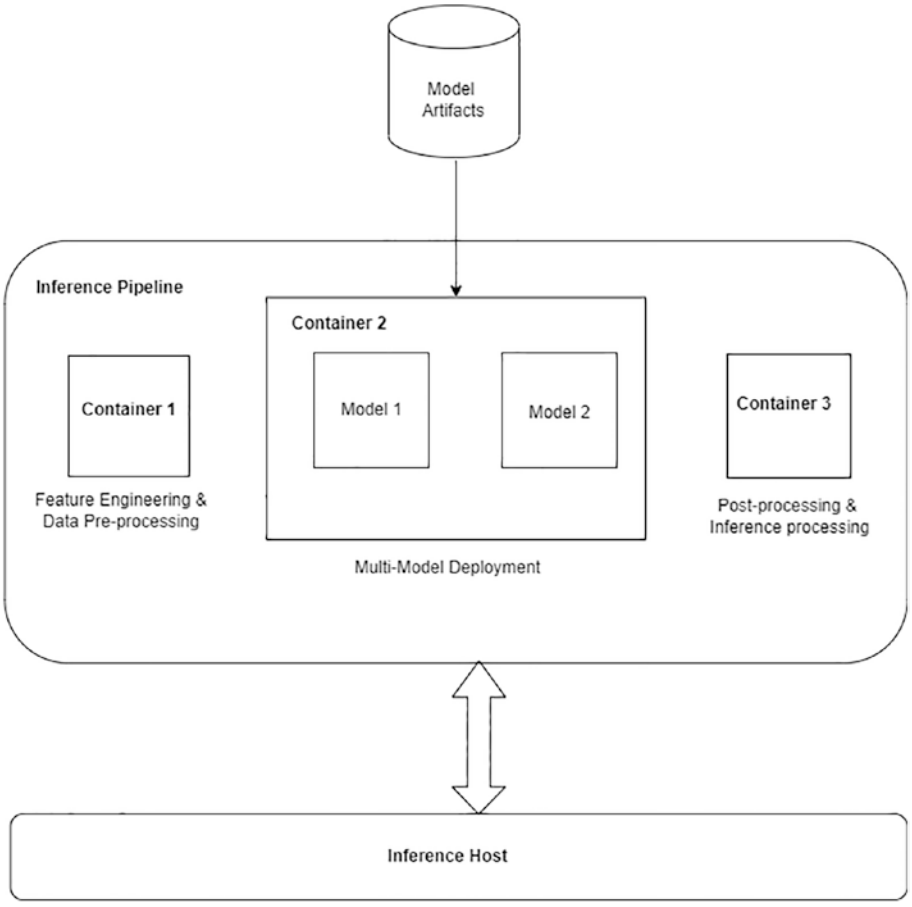

Figure 6-1. *Inference pipeline supporting multi-model deployment*

Reducing Production-Training Skew

Your model was trained on data that was carefully collected and curated for your specific problem. At this stage, you likely have a good idea of the distribution of features that go into your model, and you've translated certain assumptions about your model into the training pipeline, dealing with issues like imbalanced data and missing values.

But what happens when your model hits production and needs to make a prediction on unseen data? How can we guarantee that the new data follows the same distribution as the training data? How can we guarantee the integrity of the model output so that stakeholders can trust the output enough to action on the insights the model provides? This is where the concept of production-training skew comes into our vocabulary and starts to impact the technical decisions we make around model deployment.

Production-training skew can be formally defined as a difference in model performance during production and training phases. Performance here can mean the accuracy of the model itself (e.g., the unseen data has a different probability distribution than expected or can be caused by failing to handle certain edge cases in our training pipeline that crop up when we go to production).

It's worth noting that sometimes issues happen in production that are not anticipated even if we have a really good understanding of the assumptions of our data and models. For example, we might expect certain features to be available at inference time because they were available at training time, but some features might need to be computed on the fly and the data just may not be available.

In general, it is best practice to ensure your inference pipeline has safeguards in place to check our assumptions prior to using the model, and if features are not available, or if certain statistical assumptions are not met, we can have a kill switch in the inference to prevent the model from making an erroneous prediction.

This highlights an important difference between stochastic systems and traditional software systems because the consequences of actioning on a bad model output may be disastrous. As I've mentioned, stakeholders may lose trust in your model, or the model may be used as part of a decision process that impacts real people in a negative way. Therefore, it's not enough to fail gracefully or ensure our model always has an output; as an MLOps practitioner, you need to put model safety first and foremost and ensure that if critical assumptions are not met, then what went wrong gets logged and the inference pipeline fails.

Let's take a look at how we can set up monitoring and alerting to ensure the safety and integrity of our model.

Monitoring Infrastructure Used in Inference Pipelines

Although we have a firm grasp of infrastructure, we need to take a brief moment to talk about the type of infrastructure you will need to set up for monitoring your inference pipelines. There are various cloud-based monitoring services in all of the major cloud platforms such as Amazon CloudWatch, Azure Monitor, or Google Cloud Monitoring. These tools provide monitoring and alerting capabilities that can be integrated with data pipelines. Your organization may have their own monitoring infrastructure set up already in which case you should consider leveraging this instead of creating new services.

There are also specific monitoring tools for data pipelines and ELT frameworks; for example, AWS Glue and Airflow both have built-in monitoring, and you can use this to build your own custom data drift detection solution by creating a separate pipeline and setting up hooks that can talk to other infrastructure.

The difference between the data specific monitoring tools and the more general cloud monitoring tools is the more general cloud monitoring tools also can monitor resource utilization and you can use this to get a sense of where performance bottlenecks exist in your code. You may have to read the documentation for these cloud services and learn the SDK (software development kit), so you integrate these tools with your own code base. Whether you choose a stand-alone cloud monitoring service or leverage an existing one or one built-in with your ELT framework will depend on your project and the specific problem you're trying to solve.

Okay, so once you have made the technical decision on what type of monitoring service you want to use for your data drift and model drift detection, then we can talk about how you can implement monitoring in your inference pipeline and some of the challenges that you might encounter.

Monitoring Data and Model Drift

Monitoring is an essential part of nearly every IT operational system. It also happens to be one of the ways we can make data-driven decisions about our production models. Monitoring is a way of collecting data (strictly speaking, this is logging) and the capability of observing data over a period of time, for example, to check if certain conditions are met that are actionable. The action is usually called an alert.

It's important to realize that when working with monitoring systems, this data is collected in the form of logs, but the logs need not be centralized and are typically streamed via standard output and standard error and then consolidated using some logging service. Services include cloud services like Data Dog or Azure Monitoring or open source solutions like Ganglia, ElasticSearch.

In the context of machine learning and stochastic systems, monitoring means monitoring the model specifically for data drift and model drift and ensures the reliability and integrity of our model. We define these related terms.

Data drift: Data drift is related to the statistical properties and the probability distribution that underlies the features that go into training the model. When the underlying distributions of features shift in terms of mean, variance, skewness, or other statistical properties we can track, then it may invalidate assumptions we have made in the training pipeline and render model output invalid. A way to continuously monitor these statistical properties needs to be implemented.

How do you measure the difference between the distribution of features 6 months ago and at the present time? There are several ways to approach this, and one way is to measure the "distance" between two probability distributions such as with KL divergence or Mahalanobis distance. The important detail here is that we need to first measure a baseline and we compute this distance against the baseline, usually by defining a threshold value. If the divergence between our observed and baseline exceeds this threshold, then we can choose to send out an alert (e.g., an email to relevant stakeholders). It's important we actually send out an alert and build out the code to do this, for example, if your team uses Slack, you may consider building a slack bot to alert your data drift has occurred since important decisions need to be made on whether to retrain the model and understand the root cause of the shift.

Another approach to data drift is hypothesis testing. We can set the null hypothesis to the features that have not changed or come from a well-known distribution like the normal distribution if your data is normally distributed. One commonly employed hypothesis test is the Kolmogorov-Smirnov test where the null hypothesis is that the data comes from the normal distribution.

Once we have confirmed that data drift has occurred, we have to make a technical decision: Should we retrain our model? This is the first kind of feedback we can introduce into the training pipeline and is more sophisticated than the alternative which is periodically retraining on a set schedule (which may be a waste of resources if data drift has not occurred or is within the SLA threshold).

Model drift: Model drift is a slightly different concept than data drift and can indicate that the business problem has changed. It's important to define the business problem and the definitions of features as part of the feature engineering step so that you can validate if model drift has actually occurred once detected.

Detecting possible model drift is fairly straightforward but verifying it is not. In order to detect model drift, we only need to monitor the predicted values (or more generally, the output of the model) and compare them to the expected values over time. For example, if we have a multi-class classification problem, we might record the total number of predictions made for each class and the breakdown of our predictions by each class, counting the number of predictions made for each class. We could visualize this as a simple histogram where the bins are the classes in our model, and if we find this histogram changes too dramatically from the baseline (using since threshold we define for the specific problem), then we have data drift and suspected model drift (performance of our model may have degraded over time).

We may also keep track of accuracy and other performance metrics and keep track of the performance of our model over time and a baseline and confidence intervals if possible.

Once we have found that either the model output has changed or the model performance metrics are degrading, then we need to investigate if model drift is actually concept drift, meaning the business problem has shifted in some way. This may lead us not only to retrain the model but possibly to have to add new features, revise features, or even change the model and its assumptions entirely to match the new business problem.

In order to keep track of the model output, we need a reliable way to make predictions with our model (if the mechanism isn't reliable or at least as reliable as the model output, then we won't be able to tell when we model drift has occurred). Creating the API for inference is not only about user experience but also ensuring the accuracy and reliability of the model output. In the next section, we'll go over some of the considerations that go into designing a reliable inference API.

Designing Inference APIs

Okay so let's say we have the most reliable inference API, we trust the data and the output of our model, and our stakeholders and users trust the output. The next focus needs to be on performance. We've noted previously there are technical trade-offs between accuracy and model performance, and while we should always consider performance early, it's important not to sacrifice accuracy or fairness of the model for performance. On the other hand, if we don't consider scalability and optimize our inference pipeline for performance, then the output may be rendered completely invalid by the time the prediction is made (e.g., delivering the prediction the next day if there is a hard requirement on the latency of the system). Due to this performance-accuracy, trade-off in some sense performance is a two-sided problem in machine learning.

In the next section, we'll take a detailed look at what we mean by performance in the context of inference pipelines in terms of both scalability and latency but also accuracy and validity and some of the important performance metrics we should be tracking in our monitoring solution. We'll also discuss the important problem of alignment in data science and how it plays a role in deciding what performance metrics to track.

Comparing Models and Performance for Several Models

In Chapter 5, we looked at model training and talked about the model tuning step. On a real-world problem, you may have many different types of models that you need to compare. You may have to dynamically select the best model, and we need a way to compare models for a problem type to choose the best model we should use for model inference.

One approach is once our models are tuned, we evaluate their performance using k-fold cross-validation and by selecting the model that has the best performance, for example, accuracy of F-1 score. This "outer validation loop" may use cross-validation but is done after hyper-parameter tuning since we need to compare models once they are already tuned; it would make little sense to make a decision on what is the best model if we haven't even gone through the effort of fine-tuning the model.

Since we'll typically be working to solve one problem type like classification or regression or anomaly detection, there are common performance metrics we can use to decide objectively what the best model should be, and there needs to be code that can handle this part of the process. Let's take a detailed look at some of these metrics and performance considerations used for comparing models across problem types.

Performance Considerations

Model performance can refer to the accuracy and validity of our model or scalability, throughput and latency. In terms of accuracy and validity, there are many metrics, and it's important to choose the metrics that are aligned with the goals of the project and the business problem we want to solve.

Here are some examples; in this table, we try to break them down by type of problem to emphasize that we need to consider the *alignment of the model* with the goal. We call Table 6-1 the alignment table for data science.

Table 6-1. *Alignment table for data science*

Problem type	Metrics
Classification	Accuracy
Classification	Precision/recall
Classification	F1 score
Regression	RMSE/MAE
Recommendation	Precision at k
Recommendation	Recall at k
Clustering	Davies-Bouldin Index
Clustering	Silhouette distance
Anomaly detection	Area under curve (AUC)
All problem types listed	Cyclomatic complexity

Of course this is not an exhaustive list since we can't possibly list every problem type you may encounter. I hope it provides a good starting point for designing your inference pipeline. In the next section, we'll take a deep dive into the other side of performance: scalability and latency.

Scalability

How can your machine learning system handle increasing amounts of data? Typically, data collection, one of the first phases of the MLOps lifecycle, grows over time. Without further information, we don't know

at what rate this data collection process grows, but even if we assume logarithmic growth, over time, we need to scale with the increasing data volume.

You might have heard the word scalability before in the context of machine learning, the ability of your system to adapt to changes in data volume. Actually, scalability goes in both directions; in fact, cloud services are often described as being "elastic," when you don't use them they should scale down and during peak periods of activity, they scale up.

What does it mean to scale up and down? We usually speak of horizontal scalability and vertical scalability.

Vertical scalability: Vertical scalability means we add additional memory, CPU, and GPUs or in the case of cloud services increase these physical resources on the virtual machine or compute we are running. By vertically scaling, we're adding more horsepower to a single worker machine, not adding new machines. This gets expensive after a while since as your memory or compute needs grow, at some point it is no longer feasible to upgrade the machine, and this is why for data science, we consider horizontally scaling workflows rather than vertically so we can leverage several inexpensive worker machines (often commodity hardware) to reach our compute and memory needs.

Horizontal scalability: Horizontal scalability means we add additional worker machines and consider the total compute (number of cores) or total memory of the entire cluster together. Usually, this comes with hidden complexity such as how we can network the machines together and shard the data across workers. Algorithms like map reduce are used to process big data sets across workers.

We mentioned in the previous chapter that we could use this horizontal scaling pattern for distributed training, but what about inference? When it comes to inference, we usually consider two types of patterns: batch mode inference and real-time inference.

Both of these patterns require different architectures and infrastructure but which one you choose depends on your particular use case (remember, we should always try and align technical decisions with our use case). Here is the definition of both batch inference and real-time inference.

Batch inference: Batch inference means we break our feature set into batches and use our model to run predictions on each batch. This type of pattern can be scaled out horizontally and also has the advantage of not requiring an API, load balancer, caching, API throttling, and other kinds of considerations that come with designing an API. If you only need to populate a table for a dashboard, for example, you might consider using batch inference. However, this pattern might be ill-suited for use cases requiring real-time or near real-time inference or on demand predictions.

Real-time inference: If your requirement is to have sub-second latency in your inference pipeline and event driven prediction or allowing the end user to make on demand predictions, then you may want to move away from batch mode and consider building an API. Your API can still be scaled horizontally using a load balancer, but you will need to set up additional infrastructure and an online feature store. If your requirement is sub-second latency, you may also need to use GPUs to make the prediction (or distributed pipelines). This is a complicated topic, and so in the next lab we'll discuss some of the components that go into building an inference API, and then you'll use MLFlow to register a model in production, pull it from the model registry, and explore how you might expose the model using an gRPC or RESTful API.

What Is a RESTful API?

A RESTful API is an interface between containers (or even remote servers) used to facilitate communication over the Internet (the communication protocol is called the HTTP protocol). RESTful APIs are created in frameworks like Flask to exchange data.

When an API endpoint is called either programmatically via a POST request (we can also do this manually using tools like Postman) or in the web browser (e.g., through a GET request), data (usually in the form of JSON) is serialized (converted to bytes) and sent across the Internet in a process called *marshaling*. The bytes are then converted back into artifacts like a model using a load function that is called deserialization. All of this happens transparently when you use a framework like Flask, and you can define endpoints (e.g., localhost:80/predict) which can be called either by other APIs or by applications that want to use your API (you could do this using Python's request library; you just need to specify the endpoint, the data, and if it's a POST or GET request you need to make).

APIs are one of the many ways to build inference pipelines that the user can interact with and are particularly suited as mentioned before for on demand use cases (you can just call the endpoint when you need it) or when you need a sophisticated application that uses your model (these applications are often built as microservices).

Although building a full API is beyond the scope of this book, it is worth being aware of a few technologies that are used in building large scale applications often called *microservices*.

What Is a Microservice?

A microservice architecture is an architectural pattern for software development that organizes applications (e.g., APIs) into collections of independent (in software development parlance, this is often called loosely coupled) components called services. We've already seen examples of services when we used docker-compose to build our Jupyter lab service and MLFlow service, but you can also build your own services. In practice, these services are self-contained API endpoints written in a framework like Flask. Since the services are loosely coupled, they will need to talk to each other by sending data in the form of *messages*. These messages are usually

sent by calling an API. Since the services are loosely coupled docker containers, they can be scaled horizontally by adding more containers and distributing the load over several containers using a component called a *load balancer*. Figure 6-2 shows a typical REST API endpoint for prediction in Flask. The function predict exposes a *route* called /predict and expects features to be passed in as JSON strings in the body of a POST request (a standard way HTTP endpoints accept data). The model is loaded or *deserialized* and then used to make a prediction on the input data. The prediction is then returned as a json string, called a response.

```
import mlflow
from flask import Flask, jsonify, request
import pandas as pd

app = Flask(__name__)

@app.route("/predict", methods=["POST"])
def predict():
    # Get the JSON data from the request
    input_data = request.get_json()

    # Convert the JSON data to a pandas DataFrame
    input_df = pd.DataFrame.from_dict(input_data, orient="index").T

    # logged_model is instantiated in cell above
    loaded_model = mlflow.sklearn.load_model(logged_model)

    # Make a prediction using the loaded model
    prediction = loaded_model.predict(input_df)

    # Convert the prediction to a JSON response
    response = {"prediction": prediction.tolist()}

    return jsonify(response)
```

Figure 6-2. *Flask API prediction endpoint*

If you want to learn more about Flask, it's recommended you read the Flask documentation or several books available on microservice architecture in Flask. For most data scientists, building a microservice would be overkill, require teams of developers, and if attempted yourself would open up your project to security vulnerabilities and problems with scalability. Remember, you need to have additional components like load balancers and container orchestration frameworks like Kubernetes (docker-compose was the container orchestration tool we learned, but Kubernetes requires specific expertise to use effectively).

However, the pattern in Figure 6-2 is called a *scoring script*, and if you choose a cloud service that supports model inference, it will likely have support for creating your own inference scoring scripts which allow you to wrap the prediction logic in a function and expose a REST endpoint. Examples of cloud services that support these scoring or inference scripts include Databricks, AWS SageMaker, and Azure Machine Learning Service and MLFlow. In the lab, we'll look at how to build your own inference API and some of the details involved in registering a model, loading a model, and exposing an API endpoint in enough detail that you will have the hands-on skill to work with many different cloud services.

Lab: Building an Inference API

In the hands-on lab, you will take the code we wrote for training and adapt it for model inference. First, let's look at some of the components of an inference API. You're encouraged to do the supplementary reading before continuing to the hands-on Jupyter notebook for this chapter (I've already included them when you start the Jupyter lab for convenience, but you should try to import them yourself.).

Step 1. Run docker-compose up to start MLFlow and Jupyter lab services.

Step 2. Open your Chapter 6 lab notebook called Chapter_6_model_inference_lab.

Step 3. Run all cells to register the model and increment the model version number.

Step 4. Pull the registered model from MLFlow model registry. An example is given in Figure 6-3.

```
# Set up MLFlow logging
mlflow.set_tracking_uri("http://mlflow_server:5000")
mlflow.set_experiment("my_experiment")
client = MlflowClient()
```

```
# Get the best run
runs = client.search_runs(experiment_id, order_by=["metrics.accuracy DESC"], max_results=1)
best_run_id = runs[0].info.run_id

logged_model = f'runs:/{best_run_id}/logistic_model'

# Load model as a PyFuncModel.
loaded_model = mlflow.sklearn.load_model(logged_model)
```

Figure 6-3. *Pulling a model from MLFlow registry for use in an inference pipeline*

Step 5. Use the model to make a prediction.

Step 6. Open the deployment notebook (called Chapter_6_model_inference_api_lab) to see how MLFlow serve can be used to expose your model as an inference API.

Keeping Model Training and Inference Pipelines in Sync

In Chapter 1, we talked about how technical debt could build up in a data science project. In fact, data science projects have been described as the high interest credit card of technical debt. One subtle way projects can accumulate technical debt at the inference stage of the lifecycle is by not keeping the training and inference pipelines in sync.

The same features the model was trained on are required at time of prediction. So there we must generate those features somehow. It's convenient to think we could reuse the exact same code, but sometimes not all features will be available at prediction time and additional pipelines are necessary. A great example is a feature like customer tenure, very common in finance which technically changes every instant. This should be recomputed at inference time before being fed into the model especially if there's a large lag between when the features get refreshed and when the model is applied. Keeping training and inference pipelines in sync via shared libraries and the feature store pattern can shave off technical debt. While the problem of keeping pipelines in sync is a software engineering problem, some problems cannot be solved with software engineering since the root cause of the problem is a lack of data. One such example is the so-called "cold-start" problem.

The Cold-Start Problem

The cold-start problem is something we see in recommender systems but more generally when we're working with transactional data, for example, customer or product data in retail, finance, or insurance. The cold-start problem is a scenario where we don't have all of the history for a customer or we want to make predictions about something completely new. Since we may not have any information about a customer or product, our model won't be able to make a prediction without some adjustment. Collaborative filtering, an approach in machine learning to filter on "similar" customers or products where we do have information available, can be used to solve the cold-start problem and make predictions on completely new data points[1].

[1] In situations where there is no data, collaborative filtering may need to be supplemented with approaches such as content-based filtering.

Although we've covered quite a few things that could go wrong in our inference pipeline, we can't anticipate every possibility, and while continuous monitoring plays a crucial role in making our inference pipelines more robust, sometimes things go wrong, code gets handed off to other teams, and we need to dig deeper into the system for technical specifics. This is where documentation can be a lifesaver.

Documentation for Inference Pipelines

If you're a data scientist, you probably have copious amounts of documentation for features and statistical properties of those features, but one area where documentation may be lacking is around the assumptions that go into building an inference pipeline.

For example, do you have a naming convention for models in production? How about model versioning? Can you explain the process for updating a model or what to do if your inference pipeline breaks in production and your model isn't able to generate a prediction? All of these steps should be documented somewhere, usually in the form of a *run book*. It is also critical to have internal documentation such as a wiki that gets updated regularly. This documentation can be used for onboarding and hands-offs and to improve the quality of code and can save you when something inevitable breaks in production. Since documentation tends to only be used when things go wrong and stakeholders usually don't like reading large volumes of technical documentation, we also need a way of reporting performance metrics to stakeholders.

Reporting for Inference Pipelines

Reporting is another critical component of machine learning and in particular building the inference and training pipeline. With respect to the inference pipeline and model output, reporting is particularly important because the model output needs to be translated into business language using familiar terms that the stakeholders understand.

Since the ultimate purpose of the model was to solve a business problem, reporting could arguably be the most important piece of the puzzle as far as determining the value of your model.

Reporting can contribute to understanding the model, how the users are interacting with the model, and understanding areas of improvement and should be seen as a communication tool.

Reporting can take many forms from simple automated emails (remember we discussed one form of this for use in data drift and model drift monitoring) but also more sophisticated solutions like dashboards. Dashboards themselves should be viewed as operational systems that provide accurate data to an end user, bringing together multiple disparate data systems. Such systems may include the model output, feature store, user interaction with the model output (feedback loops), as well as other transactional or analytic database systems used by the business.

The type of dashboard you build depends on the business problem and how end users will ultimately interact with your models, but one dashboard that can add immediate value and prevent your model from ending up in the model graveyard is an explainability dashboard.

The ability to explain your model results with stakeholders is a crucial part of any data scientists' day to day role and information about model training, and what features are important when making a prediction (such as Shap value or lime) can serve as an invaluable communication tool. Some common use cases for reporting in MLOps include the following:

- Reporting on performance metrics
- Reporting on model explainability

- Reporting on model fairness for model bias reduction
- Reporting on feature importance
- Reporting on how the model output translates into key business KPIs

Reporting on how your model translates into key business KPIs is a critical exercise that should be taken into account from the beginning of the project before you even build the model, but keeping in mind that you need to translate this into a deliverable in the form of a dashboard at the end of the project can contribute to project planning help data scientists work backward from the dashboard through to the types of data and code needed to support the dashboard so a critical path for the project can be well-defined. Since data science projects have a tendency to suffer from lack of requirements or ambiguity, having a concrete deliverable in mind can reduce ambiguity and help to prioritize what is important in the project throughout the entire MLOps lifecycle.

Summary

We've come a long way in this chapter. We discussed how to build inference pipeline code examples along the way, and we actually built an inference pipeline with MLFlow and Sklearn in our hands-on lab. You should have a thorough understanding of the challenges that exist at this stage of the lifecycle from model monitoring, data drift, and model drift detection, aligning our problem to performance metrics and figuring out how to keep track of all of these performance metrics in a sane way. We discussed how to choose the best model when we have several different types of models. We gave some examples of performance metrics you

may encounter in the real world for various problem types like anomaly detection, regression, and classification. We also discussed the importance of reporting, documentation, and keeping our training and inference pipelines in sync. Some of the core topics you should now have expertise include the following:

- Reducing Production-Training Skew
- Monitoring Data and Model Drift
- Designing Inference APIs
- Performance Considerations

In the next chapter, we'll look at the final stage of the MLOps lifecycle and formally define the lifecycle, taking a step back from the technical and developing a more holistic approach to MLOps.