CHAPTER 5

Building Training Pipelines

In this chapter, you will build your own toolkit for model training. We will start by discussing the training and how it relates to the other stages of the MLOps lifecycle including the previous stage feature engineering. We'll consider several different problems that make this part of the lifecycle challenging such as identifying runtime bottlenecks, managing features and schema drift, setting up infrastructure for reproducible experiment tracking, and how to store and version the model once it's trained. We'll also look at logging metrics, parameters, and other artifacts and discuss how we can keep the model, code, and data in sync. Now, let's start by talking at defining the general problem of building training pipelines.

Pipelines for Model Training

Building pipelines are a critical part of the MLOps lifecycle and arguably the most essential part of the development and deployment of machine learning systems since training the model is the process that allows you to determine what combination of weights, biases, and other parameters best fit your training data. If model training is done correctly, meaning we've correctly minimized a cost function that maps to our business problem,

D. Sorvisto, *MLOps Lifecycle Toolkit*, https://doi.org/10.1007/978-1-4842-9642-4_5

then our end result of this process will be a model capable of generalizing beyond our training set, to unseen data, making predictions that can be actioned upon by decision-makers.

In this chapter, we will take a step back looking at model training instead as a process. We'll learn how to represent this process in a natural way as a machine learning pipeline. We'll also consider what can go wrong in this critical step of the MLOps lifecycle including what happens when we can't train our model in a reasonable amount of time, what happens when our model doesn't generalize, and how we can bring transparency and reproducibility into the training process by setting up experiment tracking. We'll also consider a part of model training that is often overlooked: model explainability and bias elimination. Let's look at some high level steps you might encounter in a training pipeline.

ELT and Loading Training Data

Model training typically occurs after you've already collected your data and, preferably, you have a feature engineering pipeline in place to refresh the data. This is a complicated step. We looked at some of the data infrastructure you can use for building feature stores in the last chapter such as relational databases, massively parallel databases, and Feast and Databricks, but if you've ever had to build an ETL (extract, transform, and load) or ELT (extract, load, and transform) pipeline, you know that it involves setting up connection strings to databases and writing SQL queries to read data, transform it, and load it into a target database. You need to set up tables, handle schema drift, and decide what tools to use for scheduling your pipeline. This is a large topic within data engineering, and we can't possibly cover every detail of this process, but we can provide you with knowledge of a few tools for building feature engineering pipelines:

Tools for Building ELT Pipelines

Data science projects need a solid foundation of data engineering in order to support the feature engineering process. Challenges exist around the part of the MLOps lifecycle between when data is collected and when that data is cleansed, transformed, and stored for downstream model training tasks. The steps that go into this are commonly called ELT or ETL (extract, transform, load), and there are data specialists that focus on this area alone. ELT is the preferred choice for data science teams since we want to first extract and then load the data in a database. Once the data is loaded, the data science team is free to transform the data as they wish without having to specify the transformation beforehand. With the ETL pattern, you need to transform the data on the fly before it is loaded which can become difficult. In the ELT pattern, the data science team can select the features that they want with data already loaded in the database and run experiments on raw data or iterate toward the feature engineering required for building the models. We also want to separate our extract, transform, and load steps, and we need a tool that is capable of passing data between steps and comes with monitoring, scheduling, logging, and ability to create parameterized pipelines. More specifically for data science, we also want to support both Python and SQL in our pipeline. Let's take a look at a few of these tools for ELT in data science.

Airflow v2: Airflow (version 2) provides an abstraction called a DAG (directed acyclic graph) where you can build pipelines in Python, specify dependencies between steps (e.g., read data, transform data, and load data), have steps run in parallel (this is why we use a DAG to represent the pipeline as opposed to a more linear data structure), and provide a convenient web interface for monitoring and scheduling pipeline runs. You will want to use at least version 2 of Airflow since version 1 requires you pass data between steps using xargs. You can build full end-to-end training pipelines in Airflow locally, but when it comes time to deploy your models in production (we'll talk about this in depth in a coming chapter), you might want to

set up Airflow as a cloud service. There are a couple options available for production Airflow workflows in the cloud such as Astronomer or Google Cloud Composer (based on Google Kubernetes Engine).

The other much more difficult option is to deploy your own Airflow instance to Kubernetes. This option is not recommended for the data scientist that wants to manage their own end-to-end lifecycle because setting up your own Airflow instance in production on Kubernetes does require knowledge of infrastructure and there are many cloud services available that provide high availability and reliability, so if you are managing the entire lifecycle end to end, it's recommended you choose a cloud platform like Astrologer provides Airflow as a service, so you don't have to deal with the low level details required to configure Airflow.

Azure Data Factory and AWS Glue

If you've worked on ETL or ELT pipelines in the cloud before, you've probably heard of AWS Glue or Azure Data Factory depending on your choice of cloud provider. Both of these options can be used especially in combination with PySpark since Azure Data Factory has an "activity" (pipeline step) for running notebook Databricks, and AWS Glue can also run PySpark for extract, transform, and load steps. One thing to consider when choosing an ELT tool is which dialect of Python is supported since for data science, you will likely be writing your extract, load, and transform steps in a combination of Python and SQL. Although this isn't a hard requirement, if the rest of your workflow is written in Python such as the data wrangling or feature engineering steps, you would need to figure out how to operationalize this code as part of your pipeline, and if you choose a low code or visual ELT tool that doesn't support Python, you will have to have the additional step of translating your entire workflow which may not be possible especially if you have complicated statistical functions. This also leads to the second consideration for choosing an ELT tool for feature engineering pipelines: Does the tool support statistical functions required by your workflow? If the tool supports Python scripts, then the answer is probably

yes, but you should still consider what kind of packages can be installed. The same applies if your data science workflow is in another language other than Python, for example, Julia or R, and you need to consider how much community support there is for your language, and using a language that isn't widely used may restrict the options you have for building your pipeline.

Another option for ELT is choosing a tool that supports the entire machine learning lifecycle end to end such as Databricks. The advantage of having a single platform is reduced effort and fewer integrations compared to a component-based system, but you still need to consider many questions such as how you're going to organize your feature engineering pipeline, what does the folder structure look like? Where will the ELT scripts live? How can I add Git integration and set up jobs to run these scripts to refresh and update data required for the model?

The last piece of advice for this section is to have as much explicit logging and error handling as possible baked into your pipeline. Although as data scientists, we might be more focused on accuracy of our scripts, when you go to deploy your pipeline to production and it breaks, you will wish you had more information in the logs and spent more time handling errors in a graceful way. Adding some basic retry logic, try-except blocks, and basic logging can go a long way to making your feature engineering pipelines robust and reliable.

Using Production Data in Training Pipeline

It goes without saying that you need production data in your training pipeline. It makes very little sense to train a model if the data is not accurate and up to date. This may pose some challenges for teams that have strict security protocols. You may need to communicate your need for production data and the business need for requiring daily or real-time refreshes of this data. For most workflows, daily frequency should be adequate, but know that if you require low latency data refreshes, it may require additional infrastructure and code changes to support this. You may have to consider using an event driven architecture rather than a batch ELT pipeline.

Preprocessing the Data

Okay so you have your ELT pipeline, and you've decided how you're going to refresh the data and the frequency of updates and have chosen your feature store where your features will live. Your pipeline runs daily. You have code to read this data into a dataframe, maybe a Pandas dataframe or a PySpark dataframe if you're working on a structured data set, or maybe you use some other libraries like Spacy for processing text based data in an unstructured format.

The point is, whether the data is structured or unstructured, the shape, volume, quality of the data, and type of machine learning problem determine how it will be processed. There are many variables here so your preprocessing steps may be different.

What matters is how you are going to translate your assumptions about your model into code. Your data may have many missing values, and your model might require a value so you will have a preprocessing step to handle missing values. You may be solving a classification problem and found your data set is imbalanced, so you may have another step that resamples your data to handle this. Other steps might include scaling the data and getting the data in a shape the model expects. Take a look at Listing 5-1 for an example.

Listing 5-1. A code snippet showing preprocessing steps

```
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

X = df.drop('label', axis=1)
y = df['label'']
```

```
# train test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_
size=0.3)

scaler = StandardScaler()
# fit transform on X_train
X_train = scaler.fit_transform(X_train)

# transform X_test
scaler.transform(X_test
```

So how do we handle all of these preprocessing steps? It's really important to keep them all in sync, and this is why you need to use a pipeline. Although your ELT pipeline should be deployed using something like Airflow, for the complex sequence of transforms, most machine learning frameworks have a concept of a pipeline you can leverage for the transformations. For example, in sklearn, you can import pipeline from sklearn;pipeline as shown in Listing 5-2.

Listing 5-2. Importing sklearn's pipeline class

```
from sklearn.pipeine import Pipeline
```

Handling Missing Values

Missing values in data can have many root causes. It is important to assess the reason why data is missing before building your training pipeline. Why? The reason is simple: Missing values mean you do not have all of the information available for prediction, but it could also indicate a problem with the data generating process itself, human error, or inaccurate data.

Missing at random (MAR) is a term used in data analysis and statistics to indicate that the missing data can be predicted from other observed values in the data set. While not completely random, data that is missing at random or MAR can be handled using techniques like multiple imputation

and other model based approaches to predict the missing value, so it's important to understand if the data qualifies as MAR or not. An example in finance would be a stock market forecast. Let's suppose you are tasked to build an LSTM model to forecast the price of a stock. You notice data is missing. Upon further investigation, you realize the missing data is correlated with another variable that indicates the stock market was closed or it was a holiday. Knowing these two indicator variables can be used to predict if the value was missing, so we say the price of the stock is missing at random. We might consider multiple imputation as a technique in our preprocessing steps to replace this missing value, or maybe it makes more sense to drop these values entirely from our model if the loss of data won't impact the accuracy of our forecast too much.

In addition to MAR, there is also MCAR (missing completely at random) and MNAR (missing not at random). With MCAR we assume that the missing data is unrelated (both to covariate and response variables). Both MCAR and MAR are ignorable; however, MNAR is not ignorable meaning the pattern of missing values is related to other variables in the data set. An example of MNAR would be an insurance survey where respondents fail to report their health status when they have a health status that might impact the insurance premium.

Knowing When to Scale Your Training Data

Scaling is applied when we have different units and scales in our training data and we want to make unbiased comparisons. Since some machine learning models are sensitive to scale, knowing when to include scaling in your training pipeline is important. Some guidelines for knowing when to scale your training data are as follows:

1. Do variables have different units, for example, kilograms and miles?

2. Are you using regularization techniques such as Ridge or Lasso? You should scale your data so that the regularization penalty is applied fairly, or you may have a situation where variables with larger ranges are penalized more than variables with smaller ranges.

3. Are you using a clustering algorithm that is distance based? Euclidean distance is highly sensitive to outliers and scale, so scaling your training data is necessary to avoid some variables dominating the computation

A general rule of thumb is to apply scaling to the numerical variables in your data since from an MLOPs perspective, even if the model does not require it, you can improve the numerical stability and efficiency. Now that we've covered some of the preprocessing steps you might encounter in a training pipeline, let's talk about a problem you will face when features change: schema drift.

Understanding Schema Drift

Let us suppose you are a data scientist at a large financial institution. You are creating a model to predict customer churn but need to consider demographic and macroeconomic data. You recently were asked to add another variable to you model: the pricing and subscription type for each level of customer. You have five variables to add, one for each subscription type; however, you will have to adjust your entire training pipeline to accommodate them. This situation is called schema drift.

There are many ways to deal with schema drift, but as a general rule, you should build your training pipeline in a way that is flexible enough to accommodate future changes in variables since they will inevitably happen. This might be as simple as altering a table to add a new column or as complex as dynamically generating SQL including variable names

and data types, creating the table on the fly as part of the training pipeline. How you deal with schema drift is up to you, and some frameworks like Databricks provide options such as the "mergeSchema" option when writing to delta tables, so if you are using an end-to-end machine learning platform or feature store, you should consult the documentation to check if there is anything related to schema drift before building out a mechanism yourself.

Feature Selection: To Automate or Not to Automate?

Feature selection is important from an MLOps perspective because it can dramatically reduce the size of your training data. If you are working on a prediction problem, you may want to discard variables that are not correlated with your target variable

An interesting question is how much of this process needs to be automated? Should your training pipeline automatically add drop variables as needed? This is likely very unsafe and could lead to disastrous consequences, for example, if someone adds a field by accident that contains PII (personally identifiable information), demographic data that violates regulatory constraints on the model or introduces data leakage into your model. In general, your training pipeline should be able to handle adding and removing features (schema drift), and you should monitor features for data and model drift, but having a human as part of the feature selection process, understanding the business implication of the features that go into your model is a safer bet than taking a completely hands-off approach.

Building the Model

Once we have preprocessed the data, the next step is to build the machine learning model. In our case, we will be using scikit-learn's logistic regression model. We can define the model and fit it to the training data, as shown in Listing 5-3:

Listing 5-3. Fitting a model in Sklearn

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression()
model.fit(X_train, y_train)
```

Evaluating the Model

Once we have built the model, the next step is to evaluate the model. We will use scikit-learn's accuracy_score function to calculate the accuracy of our model on the test data, as shown in Listing 5-4.

Listing 5-4. Evaluating the model

```
from sklearn.metrics import accuracy_score

y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

print('Accuracy:', accuracy)
```

It's important here that model evaluation can be a complex process that happens both during hyper-parameter tuning and after tuning when comparing the performance of tuned models. The former is commonly called the "inner validation loop" and is used on a subset of training data before being retested on another subset. The purpose of this procedure is to find the best hyper-parameters for the model. Once our model is tuned,

we can compare tuned models, and this is called the "outer validation loop" where you may choose the best model. Optionally you may choose tore-train the best model on the combined train and test sets in hope of getting better generalization. In the lab, you will build a training pipeline and see how some of this process works in practice.

Automated Reporting

Recall when we defined the MLOps maturity model, we said the differentiator between the first and second phase was an automated training pipeline as well as automated reporting. While we will cover performance metrics and monitoring in the next chapter, it is imperative to have infrastructure set up for reporting during the training phase; otherwise, the model can be trained, and we need to make decisions on model performance. While many MLOps professionals consider reporting to be important, reporting on model performance, model drift, and feature drift or tying in the model output from the training phase with business KPIs is a difficult process. At minimum your team should have a dashboard so you can discuss the results of trained models with stakeholders. Examples include Power BI which can be deployed to a cloud service or rolling your own such as Dash in Python and hosting it on a web server in the cloud.

Batch Processing and Feature Stores

When training a model, you need to decide if you want to store all of the data in memory or process the data in a batch, updating the weights of the model for each batch. Although gradient descent is widely used, theoretically there are alternative methods for optimization, for example, Newton's method. However, one practical advantage of gradient descent based algorithms is it allows you to train the model in a distributed

fashion, breaking up the training set into batches. You should be aware if there are batch versions of your algorithm available. Gradient descent usually refers to batch gradient descent which trains on the entire data set in one go, but there are two modes for batch training you can code yourself when using gradient descent as an optimization algorithm, and they're available in most deep learning frameworks: mini-batch and stochastic gradient descent.

Mini-Batch Gradient Descent:

Mini-batch gradient descent is a tweak to the regular gradient descent algorithm that allows you to train your model on batches of data. The size of these batches of data can be tuned to fit in memory but is usually a power of 2 such as 64 or 512 to align with computer memory. Since the gradients are calculated over the entire mini-batch, the model weights get updated for each batch. This kind of divide and conquer strategy has many performance advantages, the most obvious one is the ability to run your computations on a smaller subset of data rather than than the entire data set in one shot. This translates into reduced memory footprint and faster computations. The trade-off you should be aware of is, unlike the regular batch gradient descent on the full training data, with mini-batch gradient descent, you are only approximating the true gradient. For most cases, this is acceptable, and for larger scale machine learning projects, training on the entire data set for several thousand epochs may not be feasible.

Stochastic Gradient Descent

Stochastic gradient descent is another variation of the classical gradient descent algorithm, this time using a randomly selected sample point to compute the gradients. The gradient of the loss function is used to update the model weights for each randomly selected sample point. The advantage, like mini-batch gradient descent, is less memory usage and

possibly faster convergence. However, since the points are randomly selected from the training data, we are still only approximating the true gradient, and this approximation can be particularly noisy. Therefore, stochastic gradient descent sometimes combines with mini-batch gradient descent, so the noise term gets averaged out over many samples, leading to a smoother approximation of the true gradient.

Implementing stochastic gradient descent in a deep learning framework like PyTorch is as simple as importing the SGD optimizer.[1]

Online Learning and Personalization

The definition of an online learning method is a scenario where you don't want to train on the entire data set but still have a need to update the weights of the model as new data flows in. This is intuitive if we understand Bayes' rule which provides one such mechanism for updating a probability distribution, but when it comes to classical machine learning, we need to use gradient descent.

Linear classifiers (SVM, logistic regression, linear regression, and so on) with SGD training may come with a function that can have online or mini-batch mode supporting delta data or both online and batch mode supporting both delta data and a full data set.

Linear estimators in Sklearn, for example, implement regularized linear models with stochastic gradient descent learning. In this case, the gradient of the loss is estimated for each data point that the weights are updated by computing the partial derivative (take a look at Chapter 2 for an example of working with partial derivatives and loss functions in code). An optimization that is used with stochastic gradient descent is decreasing the learning rate (impacting the model's ability to update its weights in response to new data) as well as scaling the training data with zero mean

[1] PyTorch SGD optimizer documentation https://pytorch.org/docs/stable/generated/torch.optim.SGD.html

and unit variance (we talked a bit about this earlier in the chapter). This method is called feature scaling and can improve the time it takes for the algorithm to converge (sometimes at the cost of changing the output as with an SVM).

While online learning through methodologies like the partial fit function can be used to reduce training time, you might ask first if it is necessary since you need to build mechanisms for incremental data load, support partial fit, fine-tune the last layer of the model, and freeze the rest or some other methodology for updating the weights on a small subset of data. This can complicate the training process, so unless there is a good reason for doing it, you might be better to consider hardware accelerated training or distributed training on a full data set. However, there are still great reasons to consider online learning other than performance, one being the ability to fine-tune a model and personalize the prediction and in such cases. In the next section, we'll take a look at another important aspect of model training: model explainability.

Shap Values and Explainability at Training Time

Machine learning algorithms are often viewed as "black boxes" that accept labels and input data and give some output. As we know from Chapter 2, most algorithms are not "black boxes"; they're built up from mathematical abstractions, and although these abstractions can be powerful, they're also low bias machines, ultimately trading interpretability for a higher variance (see bias-variance trade-off). Neural networks, especially deep neural networks consisting of several layers of neurons stacked, are an example with both high variance and low explainability.

Fortunately, solving the problem of model explainability has come a long way, and some of the most widely used tools are LIME and SHAP.

LIME: LIME is an acronym for local interpretable model-agnostic explanations. The goal of LIME is to show how each variable is used in the prediction. In order to achieve this, LIME perturbed each observation and

fits a local model to these perturbations. The model's coefficients are then used to assign a weighting to the feature importance of each variable. This weighting can then be interpreted as how much each variable contributed to the prediction, allowing the data scientist to explain the model. LIME is typically more performant than SHAP.

SHAP: Shapley Additive Explanations or SHAP relies on the concept of a Shapley value, a mathematical construct that uniquely satisfies some theoretical properties (from cooperative game theory) of an ideal model-agnostic explainability algorithm. The Shapley values can be interpreted as how much each feature contributed to the prediction. An interesting consequence of using Shapley values, which are available for each observation, is you can use it for model fairness as well, for example, to estimate the demographic parity of features in your model. SHAP aims to approximate the model globally and gives more accurate and consistent results, whereas LIME, which approximates the model locally, is much faster.

Feedback Loops: Augmenting Training Pipelines with User Data

One way to evaluate the maturity of an MLOps solution is by asking if it can incorporate output of the model back into the model training process, creating a simple kind of feedback loop. Feedback loops are ubiquitous throughout engineering.

Hyper-parameter Tuning

The final section we need to cover is hyper-parameter tuning and how it relates to the entire training pipeline. We know that models have hyper-parameters which are exactly that extra parameters like depth of a tree, number of leaves for tree based models, regularization parameters, and

many other parameters that can do a variety of things from preventing model-overfitting to changing the architecture or efficiency of the model.

If you look at a boosting model like gradient boosting machines, you may have many hyper-parameters, and knowing details about how the algorithm works, for example, does it grow leaf-wise or level-wise, is essential to using the model correctly and tuning it to your business problem.

How do we search through a search space? We face a problem of combinatorial explosion if we try to do a brute force approach. We might try random search which reduces the search space, but then we might randomly miss important parameters and not have the best model at the end. A common approach is to use Bayesian optimization for hyper-parameter search. With the Bayesian approach, the best combination of hyper-parameters is learned as the model is trained, and we can update our decisions on which parameters to search as the process progresses, leading to a much better chance of finding the best model.

How do we implement Bayesian hyper-parameter search? One library that you will likely run into is HyperOpt. One important point is that you can set up MLFlow's experiment tracking inside the Hyperopt objective function. This powerful combination of MLFlow and Hyperopt can be an invaluable piece of your workflow. If you run the lab, you can see this pattern implemented and integrate it into your own MLOps toolkit. In the next chapter, we'll build on top of this foundation and look at how we can leverage MLFlow for finding the "best model" to make predictions on unseen data and use this model and data as part of an inference pipeline, but first let's take a look at how hardware can help to accelerate the model training process.

So what can go wrong in the model training process? One problem is that of all the steps in the MLOps lifecycle, model training can take the most time to complete. In fact, it may never complete if we are only running on a CPU. Hardware acceleration, which as we discussed, refers

to the process of using GPUs (or TPUs) to speed up the training process for machine learning models, reducing the runtime by parallelizing matrix and tensor operations in an efficient way. Fortunately, there is a straightforward way to know when you might need to consider hardware accelerated training; you only need to ask yourself two questions:

- How long does it take to train my model?
- Is this training time reasonable given the business requirements?

If the answer to the second question is no, you will need to use hardware accelerated training. For example, if your model takes 3 days to train on your laptop, this is probably not acceptable, but in some cases, it may be less obvious, and you will need to consider other variables like if you can run the training pipeline automatically outside of business hours; maybe a few hours of training time is acceptable to you. You might also consider how fast the data is growing and if you will need to share resources with additional models in the future. In this case, although a few hours of training time might technically be feasible in the short term, long term you will need to consider solutions like hardware accelerated training to speed up the process, so you can accommodate the scale that you need in terms of volume of data or number of models.

Model architecture is also a critical variable to consider since, for example, deep learning models are often very expensive to train, requiring hours or days to fine-tune the models. Long short term models (LSTMs), large language models, and many generative models like generative adversarial networks are best trained on a GPU, whereas if your problem only requires decision trees or linear regression models, you may have more leeway in what hardware you use.

Hardware Accelerated Training Lab

Open the Hardware Accelerated Training Jupyter notebook in your MLOPs toolkit Jupyter notebook lab environment (named Chapter_5_gpu_accelerated_training_lab) or, optionally, in Google Colab if you do not have access to a GPU on your laptop.

In the example we've set up, you'll be using a slightly different deep learning framework than we've seen so far. You'll use this framework, TensorFlow, to train a simple neural network on the MNIST data set. As part of the training pipeline, you'll need to preprocess the data, define the model architecture, compile the model, and set up the models' optimizer and loss function.

The most important part of this lab is the line of code that sets the GPU explicitly, using GPU using the with tf.device("/GPU:0") context manager. This tells TensorFlow to use the first available GPU either on your laptop or in Google Colab to accelerate the training process.

Experimentation Tracking

Experiment Tracking software is a broad class of software used to collect, store, organize, analyze, and compare results of experiments across different metrics, models, and parameters.

Experiment tracking allows researchers and practitioners to better understand the cause and effect relationships that contribute to experimental outcomes, compare experiments to determine common factors that influence results, make complex decisions on how to improve models and metrics to improve experiment results, and also reproduce these results during model training.

Remember, model training is a process that involves data and code. We need a way to keep track of the different versions of code and models and what hyper-parameters, source code, and data went into this training process. If we don't log this information somewhere, we risk losing it,

and this means we're not able to reproduce the results of the experiment, keep track of which experiments were actually successful or even worse, and answer even the most basic questions around why an experiment went wrong.

One tool that is arguably the gold standard when it comes to experiment tracking in machine learning is MLFlow. MLFlow allows you to store models; increment model version numbers, log metrics, parameters, source code, and other artifacts; and use these artifacts at a later stage such as in a model serving pipeline.

MLFlow is itself designed for end-to-end machine learning and can be used in several stages of the MLOps lifecycle from training to deployment. It can even be used in a research context when there is a need to quickly iterate on results ad hoc and keep track of experiments across different frameworks, significantly speeding up your research.

MLFlow Architecture and Components

Experiment tracking: This component is used for logging metrics, parameters, and artifacts under a single experiment. The tracking component comes with a Tracking API which you can use in your training pipeline to log these metrics, parameters, and artifacts during the training process. In practice, experiment tracking can be set up in the hyper-parameter tuning step and used in combination with other frameworks like HyperOpt.

Projects: The MLFlow projects component is less of a traditional software component and more of a format for packaging data science code, data, and configuration. You might use projects to increase the reproducibility of your experiments by keeping data, code, and config in sync and deploying code to the cloud.

Model registry: The model registry component enables data scientists to store models with a version number. Each time the training pipeline runs, you can increment this version number and subsequently use the model API to pull a specific version number from the registry for use in a downstream model serving or deployment pipeline.

Model serving: The MLFlow model serving component allows you to expose your trained models as a RESTFul API for real-time inference or batch inference modes. You can also deploy models to a number of different environments including Docker and Kubernetes. We will cover model deployment in a subsequent chapter, but this is a vast topic that requires the deployment of not just the model itself but additional monitoring, authentication, and infrastructure to support the way in which the model is used by the end user.

Now that we've covered the basic components of MLFlow, how do we begin to use it and set up our own experiment tracking framework? Although we've worked with services in our MLOps toolkit like Feast and Jupyter labs, standing up these services as stand-alone Docker images and Python packages, MLFlow is a complex service with multiple components. For example, the model registry may need to support models that can get quite large and require either an external artifact store. We'll be using an s3 bucket for this. Technically, since we want to keep everything running locally, we'll be using another service called MinIO which emulates an s3 bucket for us where we will store our models.

Fortunately, since the docker-compose file is built for you in the last chapter, you only need to run. Go to Chapter 5 folder and run docker-compose up (Do you remember what this command does?). Listing 5-5 shows how to build all services from scratch.

Listing 5-5. Running docker-compose up with –build option

```
docker-compose up -d --build
```

You should notice this command spun up several services for you including MinIO (our cloud storage emulator for model storage), our MLFlow server (we use a relational database called MySQL for experiment tracking), and MLFlow web server where we'll be able to view our experiments and models once they're registered. You'll also notice our Jupyter lab notebook exists as a service and can talk to MLFlow through the docker-compose network backbone.

Okay, that's a lot of technical details, but how do we actually start using these services? If you look at the docker-compose file, you'll notice we exposed several ports. MLFlow web server is running on port 5000, our MinIO cloud storage service runs at port 9000, and our Jupyter lab server runs on port 8080 like before. If you open a browser and enter localhost:8080, you'll be able to access your Jupyter lab. This is where we'll run all of our code in this chapter. Table 5-1 summarizes these services and where you can access them.

Table 5-1. *Table of service endpoints used in this chapter*

Service	Endpoint	Description	Credentials
MLFlow web service	localhost:5000	View all experiments and registered models	None
Cloud storage service	localhost:9000	You need to access this once to create an s3 bucket called "mlflow"	MinIO MinIO123
Jupyter lab	localhost:8080	Where we'll be building our training pipeline	None

You should open a browser and navigate to each of these services.

Now that we have built and evaluated our machine learning model, the final step is to track our experiments using MLFlow.

Next, we need to import the mlflow package on PyPi and set the name of our experiment (we've already installed Mlflow for you as part of the Jupyter lab service but it is available as a stand-alone Python package).

When you set an experiment, all runs are grouped under this experiment name (each time you run your notebook, you are executing code and this is what is referred to as a run). You might want to establish a naming convention for experiments. For example, if you use a notebook, you could use some combination of notebook name, model types, and other parameters that define your experiment. An example code in Listing 5-6 shows similar code to what you'll find in the lab.

Listing 5-6. Creating an experiment in MLFlow using mlflow package

```
import mlflow

# Start an MLFlow experiment
mlflow.set_experiment('logistic-regression-mlflow')

# Log the parameters and metrics
with mlflow.start_run():
    mlflow.log_param('model', 'LogisticRegression')
    mlflow.log_param('test_size', 0.3)
    mlflow.log_metric('train_loss', train_loss)

    # Log the model as an artifact
    mlflow.sklearn.log_model(logistic_model, 'logistic_model')
```

What is this code doing? First, we start an MLFlow experiment by calling the set_experiment function and passing in the name of our experiment. MLFlow also comes in different flavors. For example, we can use the MLFlow lightgbm flavor to log a lightgbm model or sklearn flavor to log a sklearn model like logistic regression (we'll build on our logistic regression example from previous chapters).

Knowing which flavor of model API we're using is important when we *deserialize* the model (a fancy way of saying, loading the model back from the model registry) as we want the predict_proba and predict methods to be available. However, it can be challenging to handle different types of models in a general way.

You now have enough background knowledge to start the lab where you will build an end-to-end training pipeline and log model to MLFlow.

MLFlow Lab: Building a Training Pipeline with MLFlow

If you haven't done this already, now is time to run docker-compose up in the Chapter 5 folder and confirm all services are started by navigating to the service endpoints in Figure 5-1.

Step 1. Navigate to MinIO cloud storage service located at localhost:9000 and enter the credentials provided in Figure 5-1.

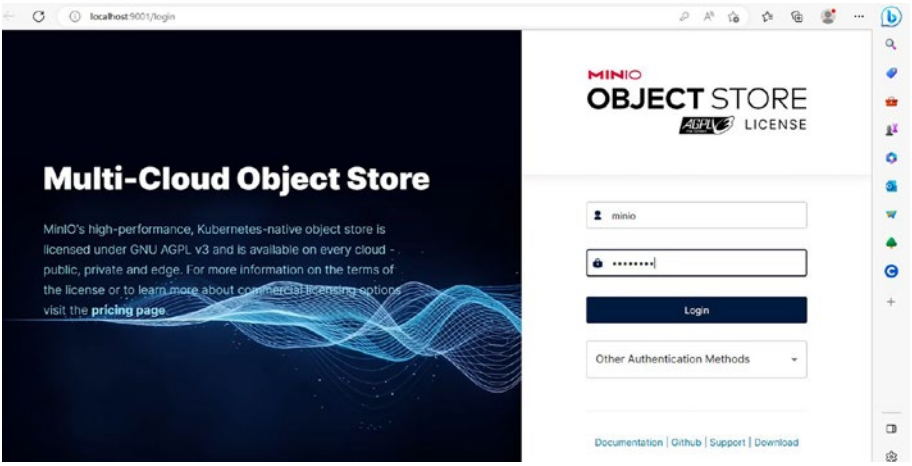

Figure 5-1. *MinIO Cloud Storage bucket*

Step 2. You need to create an s3 bucket where we'll store all of our models. Create a bucket called mlflow. If you're unfamiliar with cloud storage, you can think of this as an external drive, which we'll be referencing in our code. Figure 5-2 shows what the create bucket page looks like in MinIO.

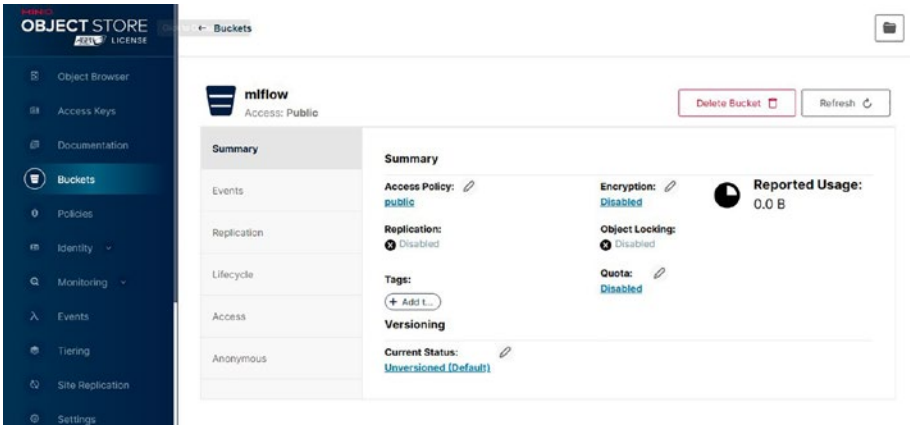

Figure 5-2. *Creating a bucket called mlflow in MinIO*

Step 3. Navigate to Jupyter lab service located at localhost:8080 in a browser, and import the notebook for Chapter_5_model_training_mlflow_lab. Read through all of the code first before running.

Step 4. Run all cells in the notebook, and navigate to the MLFlow web service located at localhost:5000. Confirm that you can see your experiment, runs, models, metrics, and parameters logged in the experiment tracking server. Figure 5-3 shows where MLFlow logs experiments.

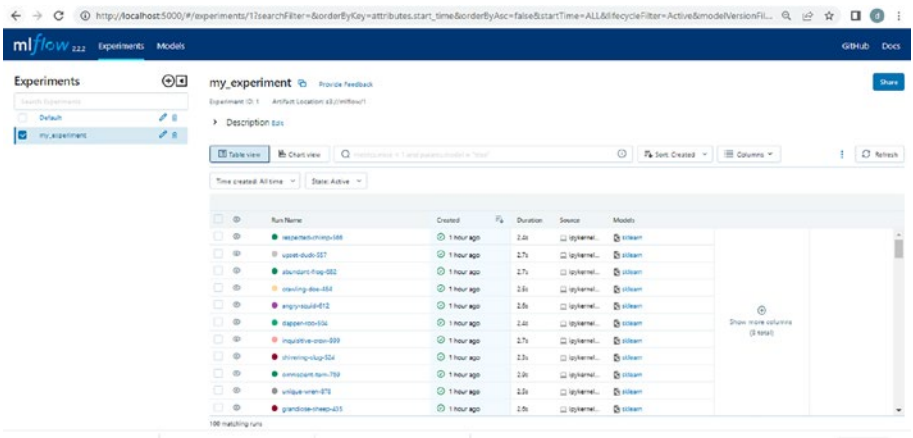

Figure 5-3. *MLFlow experiment component*

That is it! You've built an end-to-end training pipeline that trains a model and logs it to MLFlow, and you're able to search for the best run. Figure 5-4 shows the MLFlow model component.

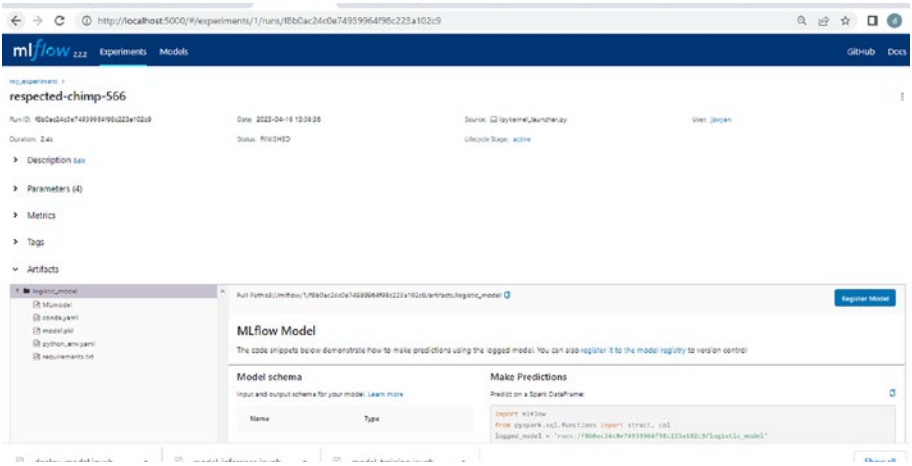

Figure 5-4. *MLFlow model registry component*

Notice the last cell uses HyperOpt's hyper-parameter tuning framework to fine-tune the model. The important detail is how we define our search space and then set MLFlow's experiment tracking inside the hyperopt objective function.

Summary

In this chapter, we learned about training pipelines, discussing how model training fits into the MLOps lifecycle, after we have made technical decisions around ELT and feature stores and we looked at some of the high level steps you might encounter as part of the transformation and data preprocessing steps. We looked at why we need to build a pipeline

and how we can make our pipelines more reliable and robust. We also discussed many of the technical aspects around setting up experiment tracking and hyper-parameter tuning. Here is a list of what you've learned up to this point.

- Tools for Building ELT Pipelines
- Preprocessing Data
- Hardware Accelerated Training
- Experimentation Tracking Using MLFlow
- Feature Stores and Batch Processing
- Shap Values and Explainability at Training Time
- Hyper-parameter Search
- Online Learning
- Setting Up an End-to-End Training Pipeline Using MLFlow

In the next chapter, we will build one some of the core ideas we learned to deploy models and build inference pipelines.