Chapter 11. Managing, Deploying, and Scaling Machine Learning Pipelines with Apache Spark

In the previous chapter, we covered how to build machine learning pipelines with MLlib. This chapter will focus on how to manage and deploy the models you train. By the end of this chapter, you will be able to use MLflow to track, reproduce, and deploy your MLlib models, discuss the difficulties of and trade-offs among various model deployment scenarios, and architect scalable machine learning solutions. But before we discuss deploying models, let's first discuss some best practices for model management to get your models ready for deployment.

Model Management

Before you deploy your machine learning model, you should ensure that you can reproduce and track the model's performance. For us, end-to-end reproducibility of machine learning solutions means that we need to be able to reproduce the code that generated a model, the environment used in training, the data it was trained on, and the model itself. Every data scientist loves to remind you to set your seeds so you can reproduce your experiments (e.g., for the train/test split, when using models with inherent randomness such as random forests). However, there are many more aspects that contribute to reproducibility than just setting seeds, and some of them are much more subtle. Here are a few examples:

Library versioning

When a data scientist hands you their code, they may or may not mention the dependent libraries. While you are able to figure out which libraries are required by going through the error messages, you won't be certain which library versions they used, so you'll likely install the latest ones. But if their code was built on a previous version of a library, which may be taking advantage of some default behavior that differs from the version you installed, using the latest version can cause the code to break or the results to differ (for example, consider how XGBoost changed how it handles missing values in v0.90).

Data evolution

Suppose you build a model on June 1, 2020, and keep track of all your hyperparameters, libraries, etc. You then try to reproduce the same model on July 1, 2020—but the pipeline breaks or the results differ because the underlying data has changed, which could happen if someone added an extra column or an order of magnitude more data after the initial build.

Order of execution

If a data scientist hands you their code, you should be able to run it top-to-bottom without error. However, data scientists are notorious for running things out of order, or running the same stateful cell multiple times, making their results very difficult to reproduce. (They might also check in a copy of the code with different hyperparameters than those used to train the final model!)

Parallel operations

To maximize throughput, GPUs will run many operations in parallel. However, the order of execution is not always guaranteed, which can lead to nondeterministic outputs. This is a known problem with functions like `tf.reduce_sum()` and when aggregating floating-point numbers (which have limited precision): the order in which you add them may generate slightly different results, which can be exacerbated across many iterations.

An inability to reproduce your experiments can often be a blocker in getting business units to adopt your model or put it into production. While you could build your own in-house tools for tracking your models, data, dependency versions, etc., they may become obsolete, brittle, and take significant development effort to maintain. Equally important is having industry-wide standards for managing models so that they can be easily shared with partners. There are both open source and proprietary tools that can help us with reproducing our machine learning experiments by abstracting away many of these common difficulties. This section will focus on MLflow, as it has the tightest integration with MLlib of the currently available open source model management tools.

MLflow

MLflow is an open source platform that helps developers reproduce and share experiments, manage models, and much more. It provides interfaces in Python, R, and Java/Scala, as well as a REST API. As shown in Figure 11-1, MLflow has four main components:

Tracking

Provides APIs to record parameters, metrics, code versions, models, and artifacts such as plots, and text.

Projects

A standardized format to package your data science projects and their dependencies to run on other platforms. It helps you manage the model training process.

Models

A standardized format to package models to deploy to diverse execution environments. It provides a consistent API for loading and applying models, regardless of the algorithm or library used to build the model.

Registry

A repository to keep track of model lineage, model versions, stage transitions, and annotations.

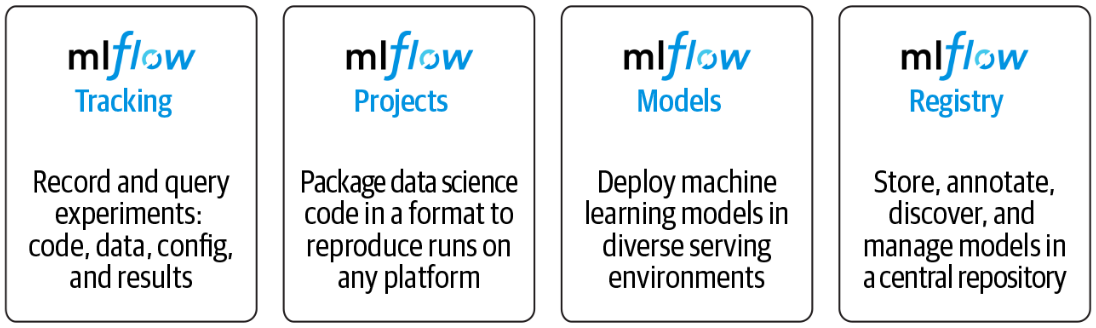

mlflow
Tracking
Record and query experiments: code, data, config, and results
mlflow
Projects
Package data science code in a format to reproduce runs on any platform
mlflow
Models
Deploy machine learning models in diverse serving environments
mlflow
Registry
Store, annotate, discover, and manage models in a central repository

Figure 11-1. MLflow components

Let's track the MLlib model experiments we ran in Chapter 10 for reproducibility. We will then see how the other components of MLflow come into play when we discuss model deployment. To get started with MLflow, simply run `pip install mlflow` on your local host.

Tracking

MLflow Tracking is a logging API that is agnostic to the libraries and environments that actually do the training. It is organized around the concept of *runs*, which are executions of data science code. Runs are aggregated into *experiments*, such that many runs can be part of a given experiment.

The MLflow tracking server can host many experiments. You can log to the tracking server using a notebook, local app, or cloud job, as shown in Figure 11-2.

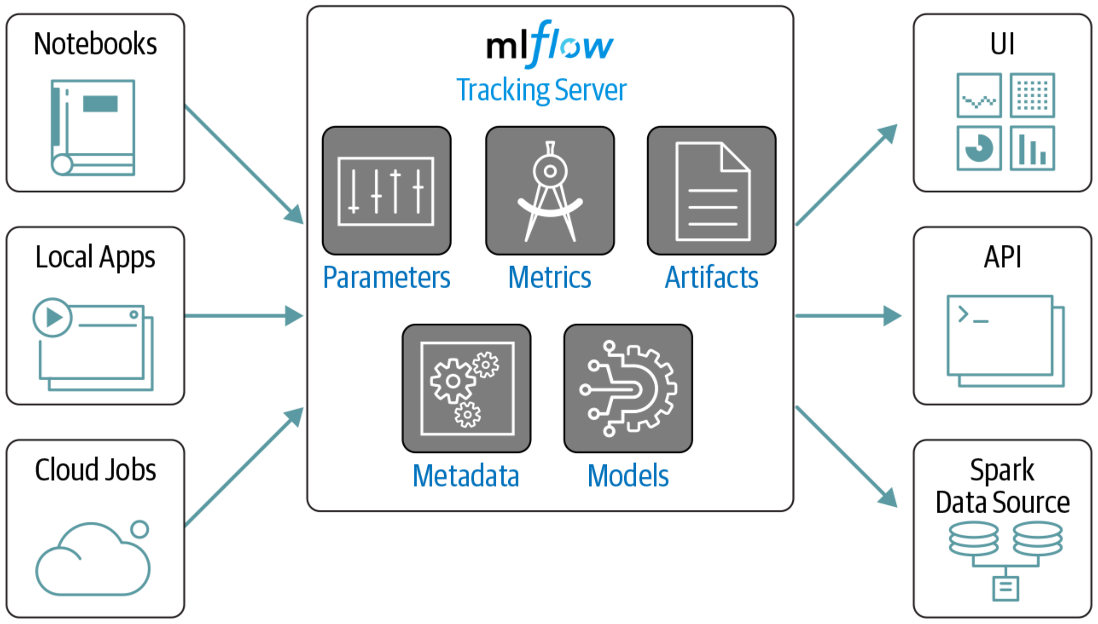


Figure 11-2. MLflow tracking server

Let’s examine a few things that can be logged to the tracking server:

Parameters

Key/value inputs to your code—e.g., hyperparameters like `num_trees` or `max_depth` in your random forest

Metrics

Numeric values (can update over time)—e.g., RMSE or accuracy values

Artifacts

Files, data, and models—e.g., `matplotlib` images, or Parquet files

Metadata

Information about the run, such as the source code that executed the run or the version of the code (e.g., the Git commit hash string for the code version)

Models

The model(s) you trained

By default, the tracking server records everything to the filesystem, but you can specify a database for faster querying, such as for the parameters and metrics. Let’s add MLflow tracking to our random forest code from Chapter 10:

```
# In Python
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, VectorAssembler
from pyspark.ml.regression import RandomForestRegressor
from pyspark.ml.evaluation import RegressionEvaluator

filePath = """/databricks-datasets/learning-spark-v2/sf-airbnb/
sf-airbnb-clean.parquet"""
airbnbDF = spark.read.parquet(filePath)
(trainDF, testDF) = airbnbDF.randomSplit([.8, .2], seed=42)

categoricalCols = [field for (field, dataType) in trainDF.dtypes
                   if dataType == "string"]
indexOutputCols = [x + "Index" for x in categoricalCols]
stringIndexer = StringIndexer(inputCols=categoricalCols,
                              outputCols=indexOutputCols,
                              handleInvalid="skip")

numericCols = [field for (field, dataType) in trainDF.dtypes
               if ((dataType == "double") & (field != "price"))]
assemblerInputs = indexOutputCols + numericCols
vecAssembler = VectorAssembler(inputCols=assemblerInputs,
                               outputCol="features")

rf = RandomForestRegressor(labelCol="price", maxBins=40, maxDepth=5,
                           numTrees=100, seed=42)

pipeline = Pipeline(stages=[stringIndexer, vecAssembler, rf])
```

To start logging with MLflow, you will need to start a run using `mlflow.start_run()`. Instead of explicitly calling `mlflow.end_run()`, the examples in this chapter will use a `with` clause to automatically end the run at the end of the `with` block:

```
# In Python
import mlflow
import mlflow.spark
import pandas as pd

with mlflow.start_run(run_name="random-forest") as run:
  # Log params: num_trees and max_depth
  mlflow.log_param("num_trees", rf.getNumTrees())
  mlflow.log_param("max_depth", rf.getMaxDepth())

  # Log model
```

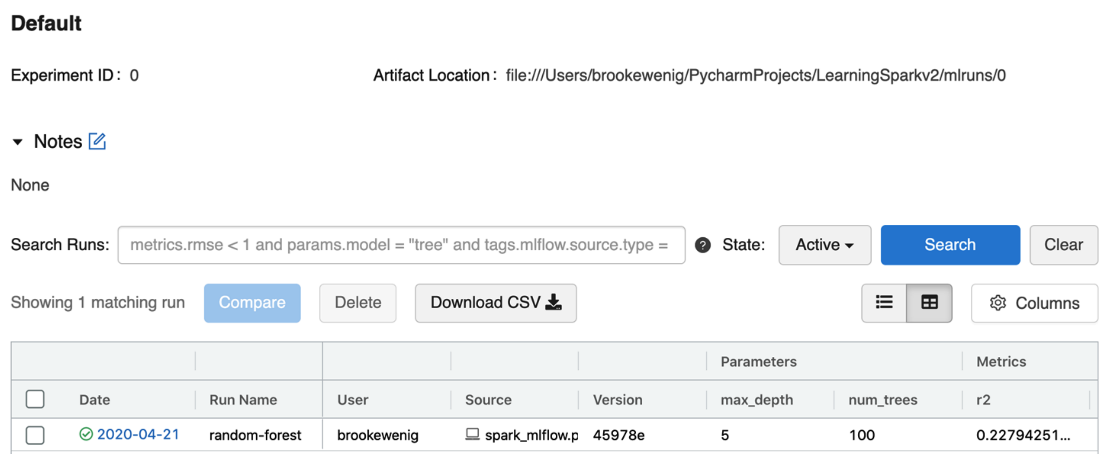
```
    pipelineModel = pipeline.fit(trainDF)
    mlflow.spark.log_model(pipelineModel, "model")

    # Log metrics: RMSE and R2
    predDF = pipelineModel.transform(testDF)
    regressionEvaluator = RegressionEvaluator(predictionCol="prediction",
                                              labelCol="price")
    rmse = regressionEvaluator.setMetricName("rmse").evaluate(predDF)
    r2 = regressionEvaluator.setMetricName("r2").evaluate(predDF)
    mlflow.log_metrics({"rmse": rmse, "r2": r2})

    # Log artifact: feature importance scores
    rfModel = pipelineModel.stages[-1]
    pandasDF = (pd.DataFrame(list(zip(vecAssembler.getInputCols(),
                                    rfModel.featureImportances)),
                               columns=["feature", "importance"])
                .sort_values(by="importance", ascending=False))

    # First write to local filesystem, then tell MLflow where to find that file
    pandasDF.to_csv("feature-importance.csv", index=False)
    mlflow.log_artifact("feature-importance.csv")
```

Let's examine the MLflow UI, which you can access by running `mlflow ui` in your terminal and navigating to *http://localhost:5000/*. Figure 11-3 shows a screenshot of the UI.



Default

Experiment ID: 0   Artifact Location: file:///Users/brookewenig/PycharmProjects/LearningSparkv2/mlruns/0

Notes

None

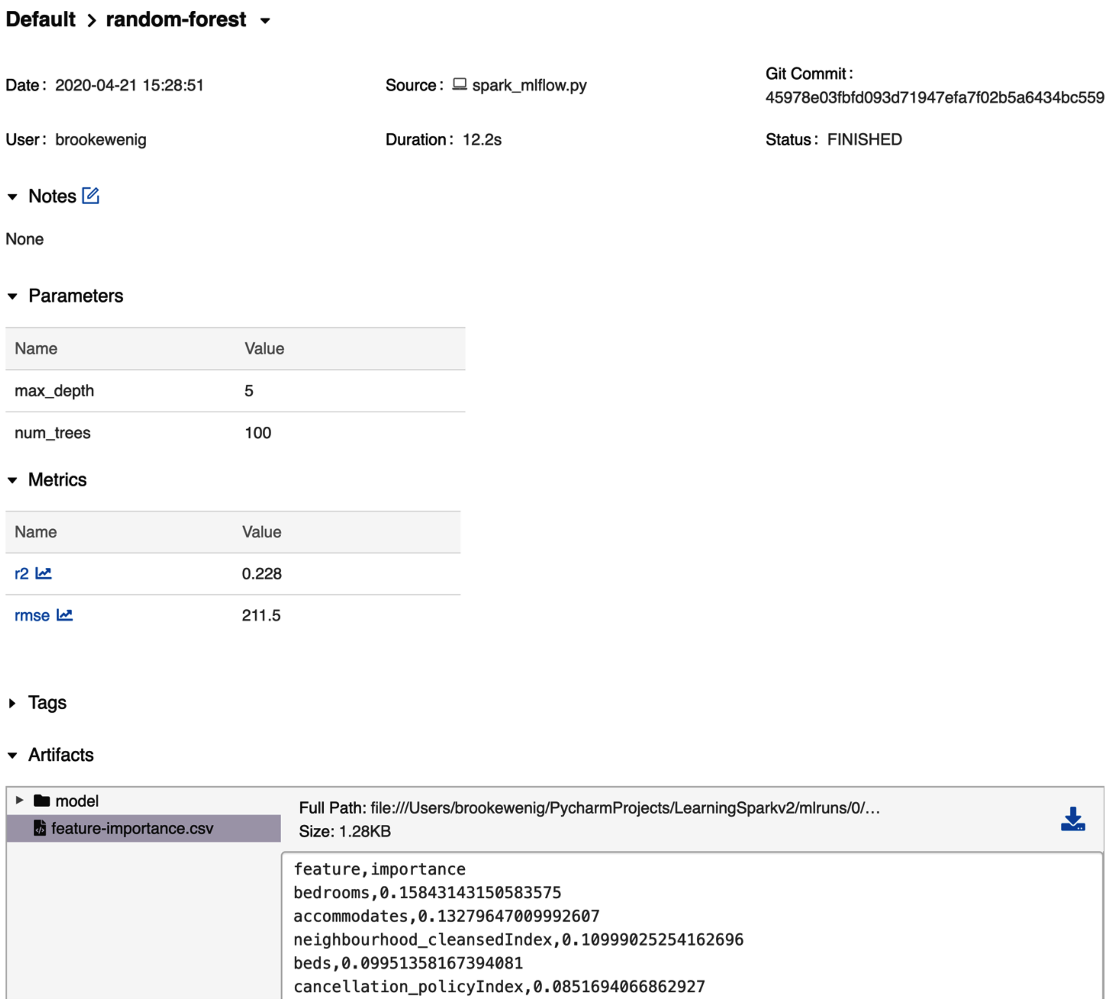
Search Runs: metrics.rmse < 1 and params.model = "tree" and tags.mlflow.source.type =   State: Active   Search   Clear

Showing 1 matching run   Compare   Delete   Download CSV   Columns

						Parameters		Metrics
	Date	Run Name	User	Source	Version	max_depth	num_trees	r2
	2020-04-21	random-forest	brookewenig	spark_mlflow.p	45978e	5	100	0.22794251...

Figure 11-3. The MLflow UI

The UI stores all the runs for a given experiment. You can search across all the runs, filter for those that meet particular criteria, compare runs side by side, etc. If you wish, you can also export the contents as a CSV file to analyze locally. Click on the run in the UI named `"random-forest"`. You should see a screen like Figure 11-4.



Date: 2020-04-21 15:28:51
Source: spark_mlflow.py
Git Commit: 45978e03fbfd093d71947efa7f02b5a6434bc559
User: brookewenig
Duration: 12.2s
Status: FINISHED

None

Name	Value
max_depth	5
num_trees	100

Name	Value
r2	0.228
rmse	211.5

Full Path: file:///Users/brookewenig/PycharmProjects/LearningSparkv2/mlruns/0/...
Size: 1.28KB

```
feature,importance
bedrooms,0.15843143150583575
accommodates,0.13279647009992607
neighbourhood_cleansedIndex,0.10999025254162696
beds,0.09951358167394081
cancellation_policyIndex,0.0851694066862927
```

Figure 11-4. Random forest run

You'll notice that it keeps track of the source code used for this MLflow run, as well as storing all the corresponding parameters, metrics, etc. You can add notes about this run in free text, as well as tags. You cannot modify the parameters or metrics after the run has finished.

You can also query the tracking server using the `MlflowClient` or REST API:

```
# In Python
from mlflow.tracking import MlflowClient

client = MlflowClient()
runs = client.search_runs(run.info.experiment_id,
                          order_by=["attributes.start_time desc"],
                          max_results=1)
```

```
run_id = runs[0].info.run_id
runs[0].data.metrics
```

This produces the following output:

```
{'r2': 0.22794251914574226, 'rmse': 211.5096898777315}
```

We have hosted this code as an MLflow project in the GitHub repo for this book, so you can experiment running it with different hyperparameter values for `max_depth` and `num_trees`. The YAML file inside the MLflow project specifies the library dependencies so this code can be run in other environments:

```
# In Python
mlflow.run(
  "https://github.com/databricks/LearningSparkV2/#mlflow-project-example",
  parameters={"max_depth": 5, "num_trees": 100})

# Or on the command line
mlflow run https://github.com/databricks/LearningSparkV2/#mlflow-project-exampl
-P max_depth=5 -P num_trees=100
```

Now that you have tracked and reproduced your experiments, let's discuss the various deployment options available for your MLlib models.

Model Deployment Options with MLlib

Deploying machine learning models means something different for every organization and use case. Business constraints will impose different requirements for latency, throughput, cost, etc., which dictate which mode of model deployment is suitable for the task at hand—be it batch, streaming, real-time, or mobile/embedded. Deploying models on mobile/embedded systems is outside the scope of this book, so we will focus primarily on the other options. Table 11-1 shows the throughput and latency trade-offs for these three deployment options for generating predictions. We care about both the number of concurrent requests and the size of those requests, and the resulting solutions will look quite different.

Table 11-1. Batch, streaming, and real-time comparison

	Throughput	**Latency**	**Example application**
Batch	High	High (hours to days)	Customer churn prediction
Streaming	Medium	Medium (seconds to minutes)	Dynamic pricing
Real-time	Low	Low (milliseconds)	Online ad bidding

Batch processing generates predictions on a regular schedule and writes the results out to persistent storage to be served elsewhere. It is typically the cheapest and easiest deployment option as you only need to pay for compute during your scheduled run. Batch processing is much more efficient per data point because you accumulate less overhead when amortized across all predictions made. This is particularly the case with Spark, because of the overhead of communicating back and forth between the driver and the executors—you wouldn't want to make predictions one data point at a time! However, its main drawback is latency, as it is typically scheduled with a period of hours or days to generate the next batch of predictions.

Streaming provides a nice trade-off between throughput and latency. You will continuously make predictions on micro-batches of data and get your predictions in seconds to minutes. If you are using Structured Streaming, almost all of your code will look identical to the batch use case, making it easy to go back and forth between these two options. With streaming, you will have to pay for the VMs or computing resources you use to continually stay up and running, and ensure that you have configured the stream properly to be fault tolerant and provide buffering if there are spikes in the incoming data.

Real-time deployment prioritizes latency over throughput and generates predictions in a few milliseconds. Your infrastructure will need to support load balancing and be able to scale to many concurrent requests if there is a large spike in demand (e.g., for online retailers around the holidays). Sometimes when people say "real-time deployment" they mean extracting precomputed predictions in real time, but here we're referring to *generating* model predictions in real time. Real-time deployment is the only option that Spark cannot meet the latency requirements for, so to use it you will need to export your model outside of Spark. For example, if you intend to use a REST endpoint for real-time model inference (say, computing predictions in under 50 ms), MLlib does not meet the latency requirements necessary for this application, as shown in Figure 11-5. You will need to get your feature preparation and model out of Spark, which can be time-consuming and difficult.

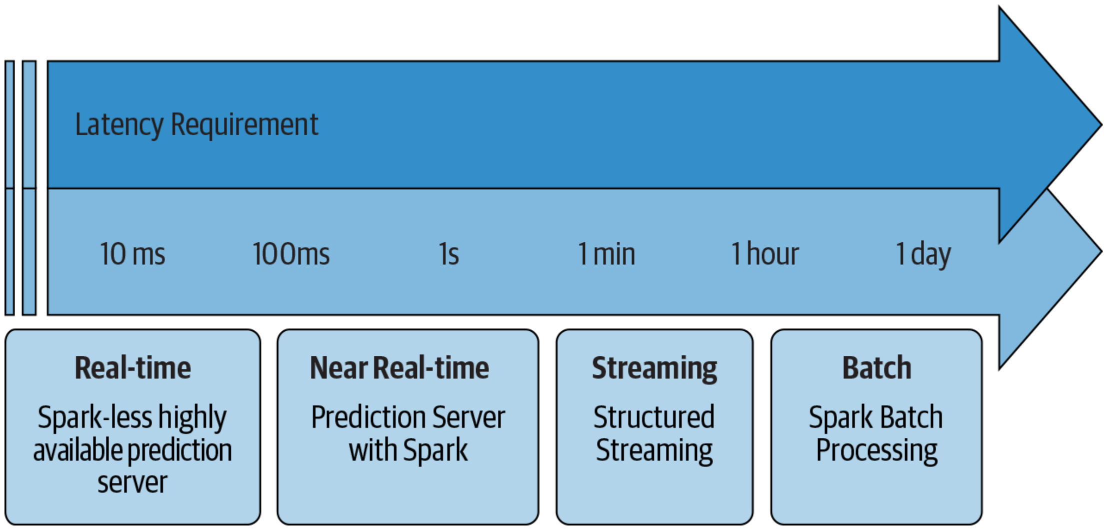


Figure 11-5. Deployment options for MLlib

Before you begin the modeling process, you need to define your model deployment requirements. MLlib and Spark are just a few tools in your toolbox, and you need to understand when and where they should be applied. The remainder of this section discusses the deployment options for MLlib in more depth, and then we'll consider the deployment options with Spark for non-MLlib models.

Batch

Batch deployments represent the majority of use cases for deploying machine learning models, and this is arguably the easiest option to implement. You will run a regular job to generate predictions, and save the results to a table, database, data lake, etc. for downstream consumption. In fact, you have already seen how to generate batch predictions in Chapter 10 with MLlib. MLlib's `model.transform()` will apply the model in parallel to all partitions of your DataFrame:

```
# In Python
# Load saved model with MLflow
import mlflow.spark
pipelineModel = mlflow.spark.load_model(f"runs:/{run_id}/model")

# Generate predictions
inputDF = spark.read.parquet("/databricks-datasets/learning-spark-v2/
  sf-airbnb/sf-airbnb-clean.parquet")

predDF = pipelineModel.transform(inputDF)
```

A few things to keep in mind with batch deployments are:

How frequently will you generate predictions?

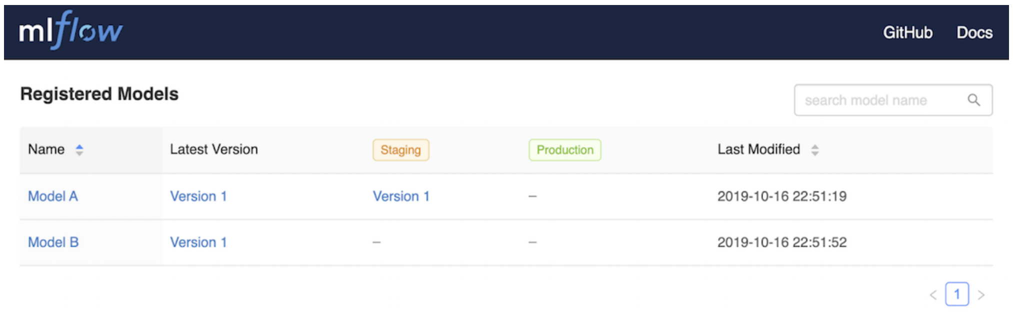
There is a trade-off between latency and throughput. You will get higher throughput batching many predictions together, but then the time it takes to receive any individual predictions will be much longer, delaying your ability to act on these predictions.

How often will you retrain the model?

Unlike libraries like sklearn or TensorFlow, MLlib does not support online updates or warm starts. If you'd like to retrain your model to incorporate the latest data, you'll have to retrain the entire model from scratch, rather than getting to leverage the existing parameters. In terms of the frequency of retraining, some people will set up a regular job to retrain the model (e.g., once a month), while others will actively monitor the model drift to identify when they need to retrain.

How will you version the model?

You can use the MLflow Model Registry to keep track of the models you are using and control how they are transitioned to/from staging, production, and archived. You can see a screenshot of the Model Registry in Figure 11-6. You can use the Model Registry with the other deployment options too.



Name	Latest Version	Staging	Production	Last Modified
Model A	Version 1	Version 1	–	2019-10-16 22:51:19
Model B	Version 1	–	–	2019-10-16 22:51:52

Figure 11-6. MLflow Model Registry

In addition to using the MLflow UI to manage your models, you can also manage them programmatically. For example, once you have registered your production model, it has a consistent URI that you can use to retrieve the latest version:

```
# Retrieve latest production model
model_production_uri = f"models:/{model_name}/production"
model_production = mlflow.spark.load_model(model_production_uri)
```

Streaming

Instead of waiting for an hourly or nightly job to process your data and generate predictions, Structured Streaming can continuously perform inference on incoming data. While this approach is more costly than a batch solution as you have to continually pay for compute time (and get lower throughput), you get the added benefit of generating predictions more frequently so you can act on them sooner. Streaming solutions in general are more complicated to maintain and monitor than batch solutions, but they offer lower latency.

With Spark it's very easy to convert your batch predictions to streaming predictions, and practically all of the code is the same. The only difference is that when you read in the data, you need to use `spark.readStream()` rather than `spark.read()` and change the source of the data. In the following example we are going to simulate reading in streaming data by streaming in a directory of Parquet files. You'll notice that we are specifying a `schema` even though we are working with Parquet files. This is because we need to define the schema a priori when working with streaming data. In this example, we will use the random forest model trained on our Airbnb data set from the previous chapter to perform these streaming predictions. We will load in the saved model using MLflow. We have partitioned the source file into one hundred small Parquet files so you can see the output changing at every trigger interval:

```
# In Python
# Load saved model with MLflow
pipelineModel = mlflow.spark.load_model(f"runs:/{run_id}/model")

# Set up simulated streaming data
repartitionedPath = "/databricks-datasets/learning-spark-v2/sf-airbnb/
  sf-airbnb-clean-100p.parquet"
schema = spark.read.parquet(repartitionedPath).schema

streamingData = (spark
                 .readStream
                 .schema(schema) # Can set the schema this way
                 .option("maxFilesPerTrigger", 1)
                 .parquet(repartitionedPath))

# Generate predictions
streamPred = pipelineModel.transform(streamingData)
```

After you generate these predictions, you can write them out to any target location for retrieval later (refer to Chapter 8 for Structured Streaming tips). As you can see, the code is virtually unchanged between the batch and streaming scenarios, making MLlib a great solution for both. However, depending on the latency demands of your task, MLlib may not be the best choice. With Spark there is significant overhead involved in generating the query plan and communicating the task and results between the driver and the worker. Thus, if you need really low-latency predictions, you'll need to export your model out of Spark.

NEAR REAL-TIME

If your use case requires predictions on the order of hundreds of milliseconds to seconds, you could build a prediction server that uses MLlib to generate the predictions. While this is not an ideal use case for Spark because you are processing very small amounts of data, you'll get lower latency than with streaming or batch solutions.

Model Export Patterns for Real-Time Inference

There are some domains where real-time inference is required, including fraud detection, ad recommendation, and the like. While making predictions with a small number of records may achieve the low latency required for real-time inference, you will need to contend with load balancing (handling many concurrent requests) as well as geolocation in latency-critical tasks. There are popular managed solutions, such as AWS SageMaker and Azure ML, that provide low-latency model serving solutions. In this section we'll show you how to export your MLlib models so they can be deployed to those services.

One way to export your model out of Spark is to reimplement the model natively in Python, C, etc. While it may seem simple to extract the coefficients of the model, exporting all the feature engineering and preprocessing steps along with them (`OneHotEncoder`, `VectorAssembler`, etc.) quickly gets troublesome and is very error-prone. There are a few open source libraries, such as MLeap and ONNX, that can help you automatically export a supported subset of the MLlib models to remove their dependency on Spark. However, as of the time of this writing the company that developed MLeap is no longer supporting it. Nor does MLeap yet support Scala 2.12/Spark 3.0.

ONNX (Open Neural Network Exchange), on the other hand, has become the de facto open standard for machine learning interoperability. Some of you might recall other ML interoperability formats, like PMML (Predictive Model Markup Language), but those never gained quite the same traction as ONNX has now. ONNX is very popular in the deep learning community as a tool that allows developers to easily switch between libraries and languages, and at the time of this writing it has experimental support for MLlib.

Instead of exporting MLlib models, there are other third-party libraries that integrate with Spark that are convenient to deploy in real-time scenarios, such as XGBoost and H2O.ai's Sparkling Water (whose name is derived from a combination of H2O and Spark).

XGBoost is one of the most successful algorithms in Kaggle competitions for structured data problems, and it's a very popular library among data scientists. Although XGBoost is not technically part of MLlib, the XGBoost4J-Spark library allows you to integrate distributed XGBoost into your MLlib pipelines. A benefit of XGBoost is the ease of deployment: after you train your MLlib pipeline, you can extract the XGBoost model and save it as a non-Spark model for serving in Python, as demonstrated here:

```
// In Scala
val xgboostModel =
  xgboostPipelineModel.stages.last.asInstanceOf[XGBoostRegressionModel]
xgboostModel.nativeBooster.saveModel(nativeModelPath)
```

```
# In Python
import xgboost as xgb
bst = xgb.Booster({'nthread': 4})
bst.load_model("xgboost_native_model")
```

NOTE

At the time of this writing, the distributed XGBoost API is only available in Java/Scala. A full example is included in the book's GitHub repo.

Now that you have learned about the different ways of exporting MLlib models for use in real-time serving environments, let's discuss how we can leverage Spark for non-MLlib models.

Leveraging Spark for Non-MLlib Models

As mentioned previously, MLlib isn't always the best solution for your machine learning needs. It may not meet super low-latency inference requirements or have built-in support for the algorithm you'd like to use. For these cases, you can still leverage Spark, but not MLlib. In this section, we will discuss how you can use Spark to perform distributed inference of single-node models using Pandas UDFs, perform hyperparameter tuning, and scale feature engineering.

Pandas UDFs

While MLlib is fantastic for distributed training of models, you are not limited to just using MLlib for making batch or streaming predictions with Spark—you can create custom functions to apply your pretrained models at scale, known as user-defined functions (UDFs, covered in Chapter 5). A common use case is to build a scikit-learn or TensorFlow model on a single machine, perhaps on a subset of your data, but perform distributed inference on the entire data set using Spark.

If you define your own UDF to apply a model to each record of your DataFrame in Python, opt for pandas UDFs for optimized serialization and deserialization, as discussed in Chapter 5. However, if your model is very large, then there is high overhead for the Pandas UDF to repeatedly load the same model for every batch in the same Python worker process. In Spark 3.0, Pandas UDFs can accept an iterator of `pandas.Series` or `pandas.DataFrame` so that you can load the model only once instead of loading it for every series in the iterator. For more details on what's new in Apache Spark 3.0 with Pandas UDFs, see Chapter 12.

NOTE

If the workers cached the model weights after loading it for the first time, subsequent calls of the same UDF with the same model loading will become significantly faster.

In the following example, we will use `mapInPandas()`, introduced in Spark 3.0, to apply a `scikit-learn` model to our Airbnb data set. `mapInPandas()` takes an iterator of `pandas.DataFrame` as input, and outputs another iterator of `pandas.DataFrame`. It's flexible and easy to use if your model requires all of your columns as input, but it requires serialization/deserialization of the whole DataFrame (as it is passed to its input). You can control the size of each `pandas.DataFrame` with the `spark.sql.execution.arrow.maxRecordsPerBatch` config. A full copy of the code to generate the model is available in this book's GitHub repo, but here we will just focus on loading the saved `scikit-learn` model from MLflow and applying it to our Spark DataFrame:

```
# In Python
import mlflow.sklearn
import pandas as pd

def predict(iterator):
  model_path = f"runs:/{run_id}/random-forest-model"
  model = mlflow.sklearn.load_model(model_path) # Load model
  for features in iterator:
    yield pd.DataFrame(model.predict(features))

df.mapInPandas(predict, "prediction double").show(3)

+-----------------+
|       prediction|
+-----------------+
| 90.4355866254844|
|255.3459534312323|
| 499.625544914651|
+-----------------+
```

In addition to applying models at scale using a Pandas UDF, you can also use them to parallelize the process of building many models. For example, you might want to build a model for each IoT device type to predict time to failure. You can use `pyspark.sql.GroupedData.applyInPandas()` (introduced in Spark 3.0) for this task. The function takes a `pandas.DataFrame` and returns another `pandas.DataFrame`. The book's GitHub repo contains a full example of the code to build a model per IoT device type and track the individual models with MLflow; just a snippet is included here for brevity:

```
# In Python
df.groupBy("device_id").applyInPandas(build_model, schema=trainReturnSchema)
```

The `groupBy()` will cause a full shuffle of your data set, and you need to ensure that your model and the data for each group can fit on a single machine. Some of you might be familiar with `pyspark.sql.GroupedData.apply()` (e.g., `df.groupBy("device_id").apply(build_model)`), but that API will be deprecated in future releases of Spark in favor of `pyspark.sql.GroupedData.applyInPandas()`.

Now that you have seen how to apply UDFs to perform distributed inference and parallelize model building, let's look at how to use Spark for distributed hyperparameter tuning.

Spark for Distributed Hyperparameter Tuning

Even if you do not intend to do distributed inference or do not need MLlib's distributed training capabilities, you can still leverage Spark for distributed hyperparameter tuning. This section will cover two open source libraries in particular: Joblib and Hyperopt.

Joblib

According to its documentation, Joblib is "a set of tools to provide lightweight pipelining in Python." It has a Spark backend to distribute tasks on a Spark cluster. Joblib can be used for hyperparameter tuning as it automatically broadcasts a copy of your data to all of your workers, which then create their own models with different hyperparameters on their copies of the data. This allows you to train and evaluate multiple models in parallel. You still have the fundamental limitation that a single model and all the data have to fit on a single machine, but you can trivially parallelize the hyperparameter search, as shown in Figure 11-7.

Figure 11-7. Distributed hyperparameter search

To use Joblib, install it via `pip install joblibspark`. Ensure you are using `scikit-learn` version 0.21 or later and `pyspark` version 2.4.4 or later. An example of how to do distributed cross-validation is shown here, and the same approach will work for distributed hyperparameter tuning as well:

```
# In Python
from sklearn.utils import parallel_backend
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
import pandas as pd
from joblibspark import register_spark

register_spark() # Register Spark backend

df = pd.read_csv("/dbfs/databricks-datasets/learning-spark-v2/sf-airbnb/
  sf-airbnb-numeric.csv")
X_train, X_test, y_train, y_test = train_test_split(df.drop(["price"], axis=1),
  df[["price"]].values.ravel(), random_state=42)

rf = RandomForestRegressor(random_state=42)
param_grid = {"max_depth": [2, 5, 10], "n_estimators": [20, 50, 100]}
gscv = GridSearchCV(rf, param_grid, cv=3)

with parallel_backend("spark", n_jobs=3):
  gscv.fit(X_train, y_train)

print(gscv.cv_results_)
```

See the `scikit-learn` GridSearchCV documentation for an explanation of the parameters returned from the cross-validator.

Hyperopt

Hyperopt is a Python library for “serial and parallel optimization over awkward search spaces, which may include real-valued, discrete, and conditional dimensions." You can install it via `pip install hyperopt`. There are two main ways to scale Hyperopt with Apache Spark:

- Using single-machine Hyperopt with a distributed training algorithm (e.g., MLlib)
- Using distributed Hyperopt with single-machine training algorithms with the `SparkTrials` class

For the former case, there is nothing special you need to configure to use MLlib with Hyperopt versus any other library. So, let's take a look at the latter case: distributed Hyperopt with single-node models. Unfortunately, you can't combine distributed hyperparameter evaluation with distributed training models at the time of this writing. The full code example for parallelizing the hyperparameter search for a Keras model can be found in the book's GitHub repo; just a snippet is included here to illustrate the key components of Hyperopt:

```
# In Python
import hyperopt

best_hyperparameters = hyperopt.fmin(
  fn = training_function,
  space = search_space,
  algo = hyperopt.tpe.suggest,
  max_evals = 64,
  trials = hyperopt.SparkTrials(parallelism=4))
```

`fmin()` generates new hyperparameter configurations to use for your `training_function` and passes them to `SparkTrials`. `SparkTrials` runs batches of these training tasks in parallel as a single-task Spark job on each Spark executor. When the Spark task is done, it returns the results and the corresponding loss to the driver. Hyperopt uses these new results to compute better hyperparameter configurations for future tasks. This allows for massive scale-out of hyperparameter tuning. MLflow also integrates with Hyperopt, so you can track the results of all the models you've trained as part of your hyperparameter tuning.

An important parameter for `SparkTrials` is `parallelism`. This determines the maximum number of trials to evaluate concurrently. If `parallelism=1`, then you are training each model sequentially, but you might get better models by making full use of adaptive algorithms. If you set `parallelism=max_evals` (the total number of models to train), then you are just doing a random search. Any number between `1` and `max_evals` allows you to have a trade-off between scalability and adaptiveness. By default, `parallelism` is set to the number of Spark executors. You can also specify a `timeout` to limit the maximum number of seconds that `fmin()` is allowed to take.

Even if MLlib isn't suitable for your problem, hopefully you can see the value of using Spark in any of your machine learning tasks.

KOALAS

Pandas is a very popular data analysis and manipulation library in Python, but it is limited to running on a single machine. Koalas is an open source library that implements the Pandas DataFrame API on top of Apache Spark, easing the transition from Pandas to Spark. You can install it with `pip install koalas`, and then simply replace any `pd` (Pandas) logic in your code with `ks` (Koalas). This way, you can scale up your analyses with Pandas without needing to entirely rewrite your codebase in PySpark. Here is an example of how to change your Pandas code to Koalas (you'll need to have PySpark already installed):

```
# In pandas
import pandas as pd
pdf = pd.read_csv(csv_path, header=0, sep=";", quotechar='"')
pdf["duration_new"] = pdf["duration"] + 100


# In koalas
import databricks.koalas as ks
kdf = ks.read_csv(file_path, header=0, sep=";", quotechar='"')
kdf["duration_new"] = kdf["duration"] + 100
```

While Koalas aims to implement all Pandas features eventually, not all of them are implemented yet. If there is functionality that you need that Koalas does not provide, you can always switch to using the Spark APIs by calling `kdf.to_spark()`. Alternatively, you can bring the data to the driver by calling `kdf.to_pandas()` and use the Pandas API (be careful the data set isn't too large or you will crash the driver!).

Summary

In this chapter, we covered a variety of best practices for managing and deploying machine learning pipelines. You saw how MLflow can help you track and reproduce experiments and package your code and its dependencies to deploy elsewhere. We also discussed the main deployment options—batch, streaming, and real-time—and their associated trade-offs. MLlib is a fantastic solution for large-scale model training and batch/streaming use cases, but it won’t beat a single-node model for real-time inference on small data sets. Your deployment requirements directly impact the types of models and frameworks that you can use, and it is critical to discuss these requirements before you begin your model building process.

In the next chapter, we will highlight a handful of key new features in Spark 3.0 and how you can incorporate them into your Spark workloads.