**Build and operate ML solutions with Azure Databricks**

Azure Databricks

Azure Databricks runs on top of a proprietary data processing engine called Databricks Runtime, an optimized version of Apache Spark. It allows up to 50x performance for Apache Spark workloads. Apache Spark is the core technology. Spark is an open-source analytics engine for large-scale data processing. In a nutshell: Azure Databricks offers a fast, easy, and collaborative Spark based analytics service. It is used to accelerate big data analytics, artificial intelligence, performant data lakes, interactive data science, machine learning, and collaboration.

The landing page shows the fundamental concepts to be used in Databricks:

1. The **cluster**: a set of computational resources on which we run the code.
2. The **workspace**: groups all the Databricks elements, clusters, notebooks, data.
3. The **notebook**: a document that contains runnable code, descriptive text, and visualizations.

# Clusters

We can create:

* An **all-purpose cluster**. Multiple users can share such clusters to do collaborative interactive analysis.
* A **job cluster** to run a specific job. The cluster will be terminated when the job completes (A job is a way of running a notebook or JAR either immediately or on a scheduled basis).

Before we can use a cluster, we have to choose one of the available **runtimes**. Databricks runtimes are the set of core components that run on Azure Databricks clusters. Azure Databricks offers several types of runtimes:

* **Databricks Runtime**: includes Apache Spark, components and updates that optimize the usability, performance, and security for big data analytics.
* **Databricks Runtime for Machine Learning**: a variant that adds multiple machine learning libraries such as TensorFlow, Keras, and PyTorch.
* **Databricks Light**: for jobs that don’t need the advanced performance, reliability, or autoscaling of the Databricks Runtime.

Once the cluster is running, we can select **Edit** to change its properties. In case we want to provision your cluster with additional libraries, we can select the **Libraries** and then choose **Install New**.

# Working with data

An Azure Databricks *database* is a collection of tables. An Azure Databricks *table* is a collection of structured data. We can cache, filter, and perform any operations supported by Apache Spark DataFrames on Azure Databricks tables. We can query tables with Spark APIs and Spark SQL. To add data, we can go to the landing page and select **Import & Explore Data**.

To get the data in a table, there are multiple options available:

* Upload a local file and import the data.
* Use data already existing under DBFS.
* Mount external data sources, like Azure Storage, Azure Data Lake and more.

Once the data is uploaded, it will be available as a table or as a mountpoint under the DBFS filesystem (/FileStore).

df = spark.read.csv('/dbfs/FileStore/tables/nyc\_taxi.csv', header="true", inferSchema="true")

## Using DBFS mounted data

Databricks File System (DBFS) is a distributed file system mounted into a Databricks workspace and available on Databricks clusters. DBFS is an abstraction on top of scalable object storage and offers the following benefits:

* Allows to you mount storage objects so that you can seamlessly access data without requiring credentials.
* Allows you to interact with object storage using directory and file semantics instead of storage URLs.
* Persists files to object storage, so you won’t lose data after you terminate a cluster.

The default storage location in DBFS is known as the DBFS root.

We can use the DBFS to access:

* Local files (previously imported). For example, the tables you imported above are available under /FileStore
* Remote files, objects kept in separate storages as if they were on the local file system

For example, to mount a remote Azure storage account as a DBFS folder, we can use the dbutils module:

data\_storage\_account\_name = '<data\_storage\_account\_name>'

data\_storage\_account\_key = '<data\_storage\_account\_key>'

data\_mount\_point = '/mnt/data'

data\_file\_path = '/bronze/wwi-factsale.csv'

dbutils.fs.mount(

source = f"wasbs://dev@{data\_storage\_account\_name}.blob.core.windows.net",

mount\_point = data\_mount\_point,

extra\_configs = {f"fs.azure.account.key.{data\_storage\_account\_name}.blob.core.windows.net": data\_storage\_account\_key})

display(dbutils.fs.ls("/mnt/data"))

#this path is available as dbfs:/mnt/data for spark APIs, e.g. spark.read

#this path is available as file:/dbfs/mnt/data for regular APIs, e.g. os.listdir

Notebooks support a shorthand - %fs magic command - for accessing the dbutils filesystem module. Most dbutils.fs commands are available using %fs magic commands:

# List the DBFS root

%fs ls

# Overwrite the file "/mnt/my-file" with the string "Hello world!"

%fs put -f "/mnt/my-file" "Hello world!"

## Notebooks

For runnable cells, the following programming languages are supported: Python, Scala, R, and SQL. You may choose the default language for the cells in a notebook. We can override the default language by specifying the language magic command %<language> at the beginning of a cell.

The supported magic commands are:

* %python
* %r
* %scala
* %sql

Notebooks also support a few auxiliary magic commands:

* %sh: Allows you to run shell code in your notebook
* %fs: Allows you to use dbutils filesystem commands
* %md: Allows you to include various types of documentation, including text, images, and mathematical formulas and equations.

## Dataframes

Spark uses 3 different APIs: RDDs, DataFrames, and DataSets. The architectural foundation is the resilient distributed dataset (RDD). The DataFrame API was released as an abstraction on top of the RDD, followed later by the Dataset API. We'll only use DataFrames in our notebook examples.

DataFrames are the distributed collections of data, organized into rows and columns. Each column in a DataFrame has a name and an associated type. Spark DataFrames can be created from various sources, such as CSV files, JSON, Parquet files, Hive tables, log tables, and external databases.

We can use Spark to load the table data by using the sql method:

df = spark.sql("SELECT \* FROM nyc\_taxi\_csv")

We can also read the data from the original files we've uploaded; or indeed from any other file available in the DBFS. The code is the same regardless of whether a file is local or in remote storage that was mounted, thanks to DBFS mountpoints.

df = spark.read.csv('dbfs:/FileStore/tables/nyc\_taxi.csv', header=True, inferSchema=True)

Spark supports many different data formats, such as CSV, JSON, XML, Parquet, Avro, ORC and more.

To get the number of rows available in a DataFrame, we can use the count() method.

df.count

To get the schema metadata for a given DataFrame, we can use the printSchema() method.

df.printSchema

Spark has a built-in function that allows to print the rows inside a DataFrame. By default it will only show the first 20 lines in your DataFrame and it will truncate long columns. Additional parameters are available to override these settings.

df.show(100, truncate=False) #show more lines, do not truncate

### Query dataframes

DataFrames allow the processing of huge amounts of data. Spark uses an optimization engine to generate logical queries. Data is distributed over your cluster and you get huge performance for massive amounts of data.

df = spark.read.format('json').load('sample/trips.json')

df.write.format('parquet').bucketBy(100, 'year', 'month').mode("overwrite").saveAsTable('table1'))

df.select('\*')

df.select('tripDistance', 'totalAmount')

To extract the first rows, use take:

df.take(15)

df.sort(df.tripDistance.desc())

To combine the rows in multiple DataFrames use union:

df1.union(df2)

To do a SQL-style set union (that does deduplication of elements), use this function followed by distinct().

To add or update columns use withColumn or withColumnRenamed:

df.withColumn('isHoliday', False)

df.withColumnRenamed('isDayOff', 'isHoliday')

To use aliases for the whole DataFrame or specific columns:

df.alias("myTrips")

df.select(df.passengerCount.alias("numberOfPassengers"))

To create a temporary view:

df.createOrReplaceTempView("tripsView")

To aggregate on the entire DataFrame without groups use agg:

df.agg({"age": "max"})

To do more complex queries, use filter, groupBy and join:

people \

.filter(people.age > 30) \

.join(department, people.deptId == department.id) \

.groupBy(department.name, "gender")

.agg({"salary": "avg", "age": "max"})

These join types are supported: inner, cross, outer, full, full\_outer, left, left\_outer, right, right\_outer, left\_semi, and left\_anti.

To use columns aggregations using windows:

w = Window.partitionBy("name").orderBy("age").rowsBetween(-1, 1)

df.select(rank().over(w), min('age').over(window))

To use a list of conditions for a column and return an expression use when:

df.select(df.name, F.when(df.age > 4, 1).when(df.age < 3, -1).otherwise(0)).show()

To check the presence of data use isNull or isNotNull:

df.filter(df.passengerCount.isNotNull())

df.filter(df.totalAmount.isNull())

To clean the data use dropna, fillna or dropDuplicates:

df1.fillna(1) #replace nulls with specified value

df2.dropna #drop rows containing null values

df3.dropDuplicates #drop duplicate rows

To get statistics about the DataDrame use summary or describe:

df.summary().show()

df.summary("passengerCount", "min", "25%", "75%", "max").show()

df.describe(['age']).show()

To find correlations between specific columns use corr. This operation currently only supports the Pearson Correlation Coefficient:

df.corr('tripDistance', 'totalAmount')

### Visualize data

Spark has a built-in show function, which allows to print the rows in a DataFrame. Azure Databricks adds its own display capabilities and adds various other types of visualizations out-of-the-box using the display and displayHTML functions. The following display options are available:

* We can choose the DataFrame columns to be used as axes (keys, values).
* We can choose to group our series of data.
* We can choose the aggregations to be used with our grouped data (avg, sum, count, min, max).

**More information:** for more information about the available visualizations, see [**Visualizations**](https://docs.microsoft.com/en-us/azure/databricks/notebooks/visualizations) in the Azure Databricks documentation.

Spark ML

Azure Databricks supports several libraries for machine learning. There's one key library, which has two approaches that are native to Apache Spark: **MLLib** and **Spark ML**. MLLib is a legacy approach for machine learning on Apache Spark. It builds off of Spark's [Resilient Distributed Dataset](https://spark.apache.org/docs/latest/rdd-programming-guide.html#resilient-distributed-datasets-rdds) (RDD) data structure. MLLib is still available and has not been deprecated, but there will be no new functionality added to the library. Instead, customers are advised to move to the org.apache.spark.ml library, commonly referred to as Spark ML. Spark ML is the primary library for machine learning development in Apache Spark. It supports DataFrames in its API, versus the classic RDD approach.

# Train and validate a model

The process of training and validating a machine learning model using Spark ML is fairly straightforward. The steps are as follows:

1. Splitting data: DataFrames support a randomSplit() method, which makes this process of splitting data simple.
2. Training a model: Training a model relies on three key abstractions: a **transformer**, an **estimator**, and a **pipeline**. Transformers are helpful for performing feature engineering and feature selection, as the result of a transformer is another DataFrame. Transformers will implement a .transform() method. An estimator takes a DataFrame as an input and returns a model, which is itself a transformer. Estimators implement a .fit() method. Pipelines combine together estimators and transformers and implement a .fit() method.
3. Validating a model: Spark ML includes built-in summary statistics for models based on the algorithm of choice. Using linear regression for example, the model contains a summary object. This will contain the summary measures based on the **training** data. From there, with a **validation** dataset, it is possible to calculate summary statistics on a never-before-seen set of data, running the model's transform() function against the validation dataset. From there, use evaluators such as the RegressionEvaluator to calculate measures such as RMSE, MAE, and R2.

# Other machine learning frameworks

Azure Databricks supports machine learning frameworks other than Spark ML and MLLib. For example, Azure Databricks offers support for popular libraries like TensorFlow and PyTorch. It is possible to install these libraries directly, but the best recommendation is to use the [Databricks Runtime for Machine Learning](https://docs.microsoft.com/en-us/azure/databricks/runtime/mlruntime). This runtime comes with various machine learning libraries pre-installed, including TensorFlow, PyTorch, Keras, and XGBoost. It also includes libraries essential for distributed training, allowing data scientists to take advantage of the distributed nature of Apache Spark. For libraries, which do not support distributed training, it is also possible to use a [single node cluster](https://docs.microsoft.com/en-us/azure/databricks/clusters/single-node). For example, [PyTorch](https://docs.microsoft.com/en-us/azure/databricks/applications/machine-learning/train-model/pytorch" \l "use-pytorch-on-a-single-node) and [TensorFlow](https://docs.microsoft.com/en-us/azure/databricks/applications/machine-learning/train-model/tensorflow#use-tensorflow-on-a-single-node) both support single node use.

MLFlow

**MLflow** is an open-source product designed to manage the Machine Learning development lifecycle. MLflow is an important part of machine learning with Azure Databricks, as it integrates key operational processes with the Azure Databricks interface. As a side note, MLflow will also operate on workloads outside of Azure Databricks.

There are four components to MLflow:

* MLflow Tracking: provides the ability to audit the results of prior model training executions. MLflow Tracking is built around **runs**, that is, executions of code for a data science task. These runs can be combined together into **experiments**, which are intended to collect and organize runs. Each run contains several key attributes, including:
  + **Parameters**: Key-value pairs, which represent inputs.
  + **Metrics**: Key-value pairs, which represent how the model is performing.
  + **Artifacts**: Output files.
* MLflow Projects: a way of packaging up code in a manner, which allows for consistent deployment and the ability to reproduce results. MLflow supports several environments for projects, including via Conda, Docker, and directly on a system. Each project includes at least one **entry point**, which is a file (either **.py** or **.sh**) that is intended to act as the starting point for project use. Projects also specify details about the **environment**.
* MLflow Models: a standardized format for packaging models for distribution. A **model** in MLflow is a directory containing an arbitrary set of files along with an **MLmodel** file in the root of the directory. This standardized model format allows MLflow to work with models generated from several popular libraries, including scikit-learn, Keras, MLlib, ONNX, and more. Review the [MLflow Models documentation](https://mlflow.org/docs/latest/models.html) for information on the full set of supported model flavors. Each model has a **signature**, which describes the expected inputs and outputs for the model.
* MLflow Model Registry: allows data scientists to register models in a registry. Each registered model may have multiple **versions**, which allow a data scientist to keep track of model changes over time. It is also possible to **stage** models. Each model version may be in one stage, such as **Staging**, **Production**, or **Archived**. Data scientists and administrators may **transition** a model version from one stage to the next.

MLflow Models and MLflow Projects combine with the MLflow Model Registry to allow operations team members to deploy models in the registry, serving them either through a REST API or as part of a batch inference solution using Azure Databricks.

# Run experiments

This is useful for comparing changes over time or comparing the relative performance of models with different hyperparameter values. Creating an experiment in Azure Databricks happens automatically when you start a run.

with mlflow.start\_run():

mlflow.log\_param("input1", input1)

mlflow.log\_param("input2", input2)

# Perform operations here like model training.

mlflow.log\_metric("rmse", rmse)

In this case, the experiment's name will be the name of the notebook. It is possible to export a variable named MLFLOW\_EXPERIMENT\_NAME to change the name of your experiment should you choose. Inside a notebook, the **Experiment** menu option displays a context bar, which includes information on runs of the current experiment. Selecting the External Link icon in the experiment run will provide additional details on a particular run. This link will provide the information that MLflow Tracker logged, including notes, parameters, metrics, tags, and artifacts.

# Model management

The two key steps for model management in MLflow are **registration** and **versioning** of models. Once a model is out in production, there is still more work to do. As models change over time, model management becomes a process of training new candidate models, comparing to the current version and prior candidate models, and determining whether a candidate is worthy of becoming the next production model.

## Register models

Registration is possible through the Azure Databricks UI and through code. In case of the latter, there are two ways we can register a model. The first method is to register directly from an experiment.

model\_details = mlflow.register\_model(model\_uri=model\_uri, name=model\_name)

The second method is to register during a run by naming registered\_model\_name.

with mlflow.start\_run() as run:

mlflow.log\_params("param1", 123)

mlflow.sklearn.log\_model(

sk\_model=model,

artifact\_path="model",

registered\_model\_name="sklearn Trained Model")

At this point, model registration will occur and you will have a new model. You can reference the model in code using the following method:

model = mlflow.sklearn.load\_model(

model\_uri=f"models:/{model\_name}/{model\_version}")

Model registration allows MLflow and Azure Databricks to keep track of models. This is important for two reasons. First, registering a model allows you to serve the model for real-time or batch scoring. Second, registering a model allows you to create new versions of that model over time. This gives you the opportunity to track model changes and even perform comparisons between different historical versions of models.

## Versioning

With machine learning, model training is not a one-time process. Instead, models will update over time. Keeping track of these changes is possible in MLflow using versioning. Versioning a model using the Azure Databricks UI is essentially the same as the model registration process.

In addition to creating versions of models, MLflow allows model versions to be in certain specified stages. These include:

* **Production**. This is a model version, which is intended for deployment.
* **Staging**. This is a model version, which is intended for testing prior to taking over in production.
* **Archived**. This is a model version, which is no longer intended for use, usually because it has been supplanted by a superior model version.

Model versions start out without a stage. There are two ways of doing this: through the Azure Databricks UI as well as through code. In order to transition a model version to a stage through code, use the following method:

client.transition\_model\_version\_stage(

name=model\_details.name,

version=model\_details.version,

stage='Staging',

)

After performing this transition, use the following method to retrieve a model at a particular stage:

import mlflow.pyfunc

model\_uri = "models:/{model\_name}/{model\_stage}".format(model\_name=model\_name, model\_stage=model\_stage)

model = mlflow.pyfunc.load\_model(model\_uri)

AzureML Integration

A recommended approach for running Azure Machine Learning (AML) Experiments on Azure Databricks cluster is to use MLflow Tracking and connect Azure Machine Learning as the backend for MLflow experiments. With MLflow Tracking, you track an experiment's run metrics and store model artifacts in your Azure Machine Learning workspace.

# Azure Machine Learning as the backend for MLflow experiments

## 1. Configure MLflow tracking URI to use AML

In order to configure MLflow Tracking and connect Azure Machine Learning as the backend for MLFlow experiments, you need to follow these steps as shown in the code snippet:

* Get your AML workspace object.
* From your AML workspace object, get the unique tracking URI address.
* Setup MLflow tracking URI to point to AML workspace.

import mlflow

from azureml.core import Workspace

# Get your AML workspace

ws = Workspace.from\_config()

# Get the unique tracking URI address to the AML workspace

tracking\_uri = ws.get\_mlflow\_tracking\_uri()

# Set up MLflow tracking URI to point to AML workspace

mlflow.set\_tracking\_uri(tracking\_uri)

## 2. Configure a MLflow experiment

Provide the name for the MLflow experiment as shown below. Note that the same experiment name will appear in Azure Machine Learning.

experiment\_name = 'MLflow-AML-Exercise'

mlflow.set\_experiment(experiment\_name)

## 3. Run your experiment and log metrics

In this section, we will look at how to log model metrics and artifacts to the MLflow logging API. These logged metrics and artifacts are then captured in an Azure Machine Learning workspace that provides a centralized, secure, and scalable location to store training metrics and artifacts. In your MLflow experiment, once you train and evaluate your model, you can use the MLflow logging API, mlflow.log\_metric(), to start logging your model metrics as shown below. Next, you can use MLflow’s log\_artifact() API to save model artifacts such as your Predicted vs True curve.

with mlflow.start\_run() as run:

...

...

# Make predictions on hold-out data

y\_predict = clf.predict(X\_test)

y\_actual = y\_test.values.flatten().tolist()

# Evaluate and log model metrics on hold-out data

rmse = math.sqrt(mean\_squared\_error(y\_actual, y\_predict))

mlflow.log\_metric('rmse', rmse)

mae = mean\_absolute\_error(y\_actual, y\_predict)

mlflow.log\_metric('mae', mae)

r2 = r2\_score(y\_actual, y\_predict)

mlflow.log\_metric('R2 score', r2)

plt.scatter(y\_actual, y\_predict)

plt.savefig("./outputs/results.png")

mlflow.log\_artifact("./outputs/results.png")

# Reviewing experiment metrics and artifacts in Azure ML Studio

Since Azure Machine Learning is set up as the backend for MLflow experiments, you can review all the training metrics and artifacts from within the Azure Machine Learning Studio. From within the studio, navigate to the Experiments tab, and open the experiment run that corresponds to the MLflow experiment. In the Metrics tab of the run, you will observe the model metrics that were logged via MLflow tracking APIs.

In summary, using MLflow integration with Azure Machine Learning, you can run experiments in Azure Databricks and leverage Azure Machine Learning workspace capabilities of centralized, secure, and scalable solution to store model training metrics and artifacts.

Azure ML pipelines on Azure Databricks compute

Each step can be run on a specific compute target, making it possible to combine different types of processing as required to achieve an overall goal.Azure Machine Learning supports a specialized pipeline step called **DatabricksStep** with which you can run a notebook, script, or compiled JAR on an Azure Databricks cluster.

# 1. Attaching Azure Databricks Compute to Azure Machine Learning workspace

# Load the workspace from the saved config file

ws = Workspace.from\_config()

# Specify a name for the compute (unique within the workspace)

compute\_name = 'db\_cluster'

# Define configuration for existing Azure Databricks cluster

db\_workspace\_name = 'db\_workspace'

db\_resource\_group = 'db\_resource\_group'

# Get the access token from the Databricks workspace

db\_access\_token = '1234-abc-5678-defg-90...'

db\_config = DatabricksCompute.attach\_configuration(resource\_group=db\_resource\_group,

workspace\_name=db\_workspace\_name,

access\_token=db\_access\_token)

# Create the compute

databricks\_compute = ComputeTarget.attach(ws, compute\_name, db\_config)

databricks\_compute.wait\_for\_completion(True)

# 2. Defining DatabricksStep in a pipeline

script\_directory = "./scripts"

script\_name = "process\_data.py"

dataset\_name = "nyc-taxi-dataset"

spark\_conf = {"spark.databricks.delta.preview.enabled": "true"}

databricksStep = DatabricksStep(name = "process\_data",

run\_name = "process\_data",

python\_script\_params=["--dataset\_name",

dataset\_name],

spark\_version = "7.3.x-scala2.12",

node\_type = "Standard\_DS3\_v2",

spark\_conf = spark\_conf,

num\_workers = 1,

python\_script\_name = script\_name,

source\_directory = script\_directory,

pypi\_libraries = [PyPiLibrary(package = 'scikit-

learn'),

PyPiLibrary(package = 'scipy'),

PyPiLibrary(package = 'azureml-

sdk'),

PyPiLibrary(package = 'azureml-

dataprep[pandas]')],

compute\_target = databricks\_compute,

allow\_reuse = False

)

The above step defines the configuration to create a new Databricks job cluster to run the Python script. The cluster is created on the fly to run the script and the cluster is subsequently deleted after the step execution is completed.

# 3. Submit the pipeline

# Construct the pipeline

pipeline = Pipeline(workspace = ws, steps = [databricksStep])

# Create an experiment and run the pipeline

experiment = Experiment(workspace = ws, name = "process-data-pipeline")

pipeline\_run = experiment.submit(pipeline)

# 4. Model deployment

In machine learning, **Model Deployment** can be considered as a process by which you integrate your trained machine learning models into a production environment such that your business or end-user applications can use the model predictions to make decisions or gain insights into your data. The most common way you deploy a model using Azure Machine Learning from Azure Databricks, is to deploy the model as a real-time inferencing service. In Azure Machine learning, you can create real-time inferencing solutions by deploying a model as a real-time service, hosted in a containerized platform such as Azure Kubernetes Services (AKS).

You can use the following compute targets to host your web service deployment:

|  |  |  |
| --- | --- | --- |
| Compute target | Usage | Description |
| Local web service | Testing/debug | Good for limited testing and troubleshooting. |
| Azure Kubernetes Service (AKS) | Real-time inference | Good for high-scale production deployments. Provides autoscaling, and fast response times. |
| Azure Container Instances (ACI) | Testing | Good for low scale, CPU-based workloads. |
| Azure Machine Learning Compute Clusters | Batch inference | Run batch scoring on serverless compute. Supports normal and low-priority VMs. |
| Azure IoT Edge | (Preview) IoT module | Deploy & serve ML models on IoT devices. |

1. Register a trained model.
2. Define an Inference Configuration.
3. Define a Deployment Configuration.
4. Deploy the Model.

This works in the same ways as for deployment of a non-DataBricks model. See other document (‘Deployment’ and ‘Real-time inference service’).

Hyperparameter tuning

# Automated MLflow

As you train multiple models with hyperparameter tuning, you want to avoid the need to make explicit API calls to log all necessary information about the different models to MLflow. To make tracking hyperparameter tuning easier, the [Databricks Runtime for Machine Learning](https://docs.databricks.com/runtime/mlruntime.html) also supports *automated* MLflow Tracking. When you use automated MLflow for model tuning, the hyperparameter values and evaluation metrics are automatically logged in MLflow and a hierarchy will be created for the different runs that represent the distinct models you train.

To use automated MLflow tracking, you have to do the following:

* Use a Python notebook to host your code.
* Attach the notebook to a cluster with Databricks Runtime or Databricks Runtime for Machine Learning.
* List the available hyperparameters for a specific algorithm.
* Set up the search space and sampling method.
* Run the code with automated MLflow, using CrossValidator or TrainValidationSplit.

MLflow will automatically create a main or parent run that contains the information for the method you chose: CrossValidator or TrainValidationSplit. MLflow will also create child runs that are nested under the main or parent run. Each child run will represent a trained model and you can see which hyperparameter values were used and the resulting evaluation metrics.

#Explore the hyperparameters of a specific machine learning algorithm

print(lr.explainParams())

After you select the hyperparameters, you can use ParamGridBuilder() to specify the **search space**. You can then specify how you want to choose values from that search space to train individual models with which is known as the **sampling method**. The most straight-forward sampling method is known as **grid sampling**. By default, the individual models will be trained in serial. It is possible to train models with different hyperparameter values in parallel. You can find more information on setting up the parameter grid in the documentation [here](https://spark.apache.org/docs/latest/ml-tuning.html).

#Set up a grid search to try out all possible combinations of parameters

from pyspark.ml.tuning import ParamGridBuilder

paramGrid = (ParamGridBuilder()

.addGrid(lr.maxIter, [1, 10, 100])

.addGrid(lr.fitIntercept, [True, False])

.addGrid(lr.standardization, [True, False])

.build()

)

To test how the model performs and to generate evaluation metrics, you can use a test dataset. If you want to train multiple models on the same training dataset and the same test dataset, you can use the TrainValidationSplit method to run your code, build the models, and log them automatically with MLflow.

In case you want to take extra measures to prevent overfitting, you can use the CrossValidator method to train the models with different training datasets for each model and different test datasets to calculate the evaluation metrics. To build the models for the linear regression model lr used in the examples above, you can create a RegressionEvaluator() to evaluate the grid search experiments. The settings for the hyperparameter tuning experiment can be set by using the CrossValidator() method as is done in the example below.

from pyspark.ml.evaluation import RegressionEvaluator

from pyspark.ml.tuning import CrossValidator

evaluator = RegressionEvaluator(

labelCol = "medv",

predictionCol = "prediction"

)

cv = CrossValidator(

estimator = pipeline, # Estimator (individual model or pipeline)

estimatorParamMaps = paramGrid, # Grid of parameters to try (grid search)

evaluator=evaluator, # Evaluator

numFolds = 3, # Set k to 3

seed = 42 # Seed to sure our results are the same if ran again

)

cvModel = cv.fit(trainDF)

bestModel = cvModel.bestModel

Alternatively, you can look at all models you trained through the UI of MLflow.

# Hyperopt

The main advantage to using Hyperopt is that it is flexible and it can optimize any Python model with hyperparameters. Hyperopt is already installed if you create a compute with the Databricks Runtime ML.

## 1. Define an objective function to minimize

The objective function represents what the main purpose is of training multiple models through hyperparameter tuning. Often, the objective is to minimize training or validation loss. When defining a function, you can make use of any evaluation metric that can be calculated with the algorithm you selected. For example, if we use a [support vector machine classifier from the scikit-learn library](https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html), you can vary the value for the regularization parameter c. The objective is to have the model with the highest accuracy. Since Hyperopt wants a function that it needs to minimize, you can define the objective function as the negative accuracy so that a lower score actually means a higher accuracy.

from sklearn.model\_selection import cross\_val\_score

from sklearn.svm import SVC

def objective(C):

clf = SVC(C)

accuracy = cross\_val\_score(clf, X, y).mean()

return {'loss': -accuracy, 'status': STATUS\_OK}

## 2. Define the hyperparameter search space

If you want to make use of Hyperopt's Bayesian approach to sampling, there is a set of expressions you can use to define the search space that is compatible with Hyperopt's approach to sampling.

Some examples of the expressions used to define the search space are:

* hp.choice(label, options): Returns one of the options you listed.
* hp.randint(label, upper): Returns a random integer in the range [0, upper].
* hp.uniform(label, low, high): Returns a value uniformly between low and high.
* hp.normal(label, mu, sigma): Returns a real value that's normally distributed with mean mu and standard deviation sigma.

For the complete list of expressions, see the [Hyperopt documentation](https://github.com/hyperopt/hyperopt/wiki/FMin" \l "21-parameter-expressions).

## 3. Select the search algorithm

There are two main choices in how Hyperopt will sample over the search space:

* hyperopt.tpe.suggest: Tree of Parzen Estimators (TPE), a Bayesian approach, which iteratively and adaptively selects new hyperparameter settings to explore based on past results.
* hyperopt.rand.suggest: Random search, a non-adaptive approach that samples over the search space.

## 4. Run the Hyperopt function fmin()

Finally, to execute a Hyperopt run, you can use the function fmin(). The fmin() function takes the following arguments:

* fn: The objective function.
* space: The search space.
* algo: The search algorithm you want Hyperopt to use.
* max\_evals: The maximum number of models to train.
* max\_queue\_len: The number of hyperparameter settings generated ahead of time. This can save time when using the TPE algorithm.
* trials: A SparkTrials or Trials object. SparkTrials is used for single-machine algorithms such as scikit-learn. Trials is used for distributed training algorithms such as MLlib methods or Horovod. When using SparkTrials or Horovod, automated MLflow tracking is enabled and hyperparameters and evaluation metrics are automatically logged in MLflow.

For more information on how to configure fmin() and SparkTrials, read about the Hyperopt concepts [here](https://docs.microsoft.com/en-us/azure/databricks/applications/machine-learning/automl-hyperparam-tuning/hyperopt-concepts).

Distributed Deep Learning

Deep learning is a subfield of machine learning and refers to models that consist of multiple layers, also known as deep neural networks. The training process starts with data being submitted to the input layer in **batches**. The data is analyzed by the input layer and passed on to the next layer until it reaches the output layer and produces a prediction. Predictions are compared to the actual known value and based on these results, weights, and bias values are revised to improve the model. Batches are processed by the network over multiple iterations or **epochs**. Each epoch, the model tries to further improve predictions by updating the weight and bias values.

# Deep learning with Azure Databricks

Within Azure Databricks, we can train deep learning models using the popular open-source frameworks for Python: TensorFlow, PyTorch, and Keras. When we use any of these single-node frameworks to train a deep learning model, we should use a *single-node* cluster in Azure Databricks.

Deep learning models benefit from having more data: the more data, the more likely it is we will get a better or more accurate model. Although it is advised to train deep learning models on single-node clusters, your model or your data may be too large to fit in the memory of a single machine. A single-node cluster being insufficient is the problem that data scientists may face when training deep learning models. Thankfully, Horovod can help in these scenarios.

# Horovod

Horovod is an open-source distributed training framework and is the alternative to training a model on a single-node cluster. Horovod allows data scientists to distribute the training process and make use of Spark's parallel processing. When Horovod is used on top of one of the deep learning frameworks (TensorFlow, PyTorch or Keras), it trains multiple models on different batches of the input dataset on separate workers. In other words, multiple models are trained in parallel on separate workers using different subsets of the data. At the end of an epoch, the weights are communicated between workers and the average weight of all workers is calculated. Next, a new epoch can start using the new average weight and during which again, multiple models are trained in parallel.

HorovodRunner is a general API, which triggers Horovod jobs. The benefit of using HorovodRunner instead of the Horovod framework directly, is that HorovodRunner has been designed to distribute deep learning jobs across *Spark* workers. As a result, HorovodRunner is more stable for long-running deep learning training jobs on Azure Databricks.

To distribute the training of a deep learning model using HorovodRunner, you should do the following:

* Prepare and test single-node code with TensorFlow, Keras, or PyTorch. Once it works, make sure to wrap the main training procedure into a single Python function. This function will be used later on to initiate the distributed execution of your code.
* Migrate the code to Horovod.
  1. Import the Horovod framework as hvd.
  2. Initialize the Horovod library with hvd.init().
  3. Pin one GPU per process. Pinning is necessary to disable random mapping of workers and avoid clashes. Pinning is skipped when using CPUs.
  4. Specify how you want to partition or sample the data so that each worker uses a unique subset of the data to train a model. As a best practice, make sure the subsets are all the same size. Depending on the input dataset, there are several techniques to do the sampling. For example, you could use Petastorm to work with datasets in Apache Parquet format. Learn more about the open-source library Petastorm [here](https://github.com/uber/petastorm).
  5. Scale the learning rate by the number of workers to make sure the weights are adjusted correctly after each epoch.
  6. Use the Horovod distributed optimizer to handle the communication between workers.
  7. Broadcast the initial parameters so al workers start with the same parameters.
  8. Save checkpoints only on worker 0 to prevent conflicts between workers.
* Use HorovodRunner to run the code and distribute your work. To run HovorodRunner, you have to create a HorovodRunner instance in which you specify how many nodes (defined by the argument np) you want to distribute your work to. You can specify to use one node if you want to test on a single-node cluster with np=-1. Finally, you can trigger the Horovod training job with HorovodRunner by invoking the Python function you created for your training code.