Databricks Machine Learning Professional Notes

ML in Production

▼ Section 1 : **Experimentation**

▼ **Data Management**

Production machine learning solution start with reproducible data management

Data Management and reproducibility

Managing the machine learning lifecycle means

- Reproducilibty of data
- Reproducibility of models
- Reproducibility of code
- Automated Integration of production systems

It can be achieve by number of ways

- Saving snapshot of the data
- Table versioning and time travel using delta.
- Using a feature table
- Saving a hash of your data to detect changes

let's load the data and generate a unique id for the each listing

```
from pyspark.sql.functions import monotonically_increasing_id

airbnb_df = (spark.read
                        .format("delta")
                        .load("/mnt/training/airbnb/sf-listings-2019-03-06-clean-del
                        .withColumn("index",monotonically_increasing_id)
                        )
display(airbnb_df)
```

Versioning of Delta Tables

Create a Delta Table

```
delta_path = working.dir.replace("/dfbs","dbfs":") + "/delta-example"
dbutils.fs.rm(delta_path,recurse=True)

airbnb_df.write.format("delta").save(delta_path")
```

let's read our delta table and modify it dropping the cancellation policy and instant_bookable columns

```
delta_df = (spark.read
                            .format("delta")
                            .load(delta_path)
                            .drop("cancellation_policy","instant_bookable")
                    )
display(delta_df)
```

Now we can overwrite our Delta Table using the mode parameter.

```
delta_df.write.format("delta").mode("overwrite").save(delta_path)
```

if we want to keep **cancellation_policy** column .lucky we can use data versioning to return to an older version of the table

Start by using the **DESCRIBE HISTORY** Sql Command

```
display(spark.sql(f"DESCRIBE HISTORY DELTA"{delta_path}'"))
```

as we can see in the operationParameters columns in version 1 we overwrite the table . we now need to travel back in time to load in version 0 to get all original columns then we can delete the instant_bookable column

```
delta_df = spark.read.format("delta").option("versionAsof",0).load(delta_path)
display(delta_df)
```

you can also query through timestamp

NOTE: the ability to query an older snapshot of a table (time travel) is lost after running a VACCUM command

```
timestamp = spark.sql(f"DESCRIBE HISTORY delta'{delta_table}'.first().timestamp
        display(spark.read
                        .format("delta")
                        .option("timestampAsof",timestamp)
                        .load(delta_path)
```

Now we can drop instant_bookable and overwrite the table

```
delta_df.drop("instant_bookable").write.format("delta").mode("overwrite").save(del
```

version 2 is our latest and most accurate table version

```
display(spark.sql(f"DESCRIBE HISTORY DELTA.'{delta_table}'"))
```

▼ **Feature Store**

Feature Store is a centralized repository of features . It enables feature sharing and discovery across your organization and also ensures that the same feature computation code is used for model training and inference

Create a new database and unique table name (in case you re-run the notbook multiple times - currently no support for features tables or features)

```
import uuid
import re

spark.sql(f"CREATE DATABASE IF NOT EXISTS {clean_username}")
table_name = f"{clean_username}.airbnb_" + str(uuid.uuid4())[:6]

print(table_name)
```

Let's start creating a Feature Store Client so we can populate our feature store

```
from databricks import feature_store

fs = feature_store.FeatureStoreClient()

help(fs.create_feature_table)
```

Next we can create the Feature Table using create_feature_table method

This method can take few parameters as inputs

- name - a feature table name of the form <database name>.<table name>
- keys - The primary keys if multiple columns are required .specify a list of column names
- features_df - Data to insert into this feature table . The schema of features_df will be used as feature table schema
- schema - Feature table schema .Note that either schema or feature_df must be provided
- description - Description of the feature table
- partition_columns - Column(s) used to partition the feature table

```
fs.create_feature_table(
        name=table_name,
        keys=['index'],
        features = airbnb_df,
        partition_columns = ['neighbourhood_cleansed'],
        description ="Original Airbnb data"
)
```

Now let's explore the UI and see how it tracks the table

In Machine learning workspace → Click Feature Store icon on bottom left of the navigation bar

Update the feature Store . we can begin to refine our table by filtering out some rows which don't match our specifications . we start by looking at some of the bed_type values

```
display(airbnb_df.groupby("bed_type").count())
```

Since we only want real beds we can drop the other records from the Dataframe

```
airbnb_df_real_beds = airbnb_df.filter("bed_type = "Real Bed')
display(airbnb_df_real_beds)
```

Merge Features

Now that we have filtered some of our data we can merge the existing features table in the Feature Store with the new table .Merging updates the feature table schema and adds new feature values based on the primary key

```
fs.write_table(
name=table_name,
df= airbnb_df_real_beds,
mode="merge"
)
```

we need to find average review score for each listing

```
from pyspark.sql.functions import lit,expr

reviewColumns = ["review_scores_accuracy", "review_scores_cleanliness", "review_s
                 "review_scores_communication", "review_scores_location", "review

airbnb_df_short_reviews = (airbnb_df_real_beds.withColumn("average_review_score",
                                                          .drop(*reviewColumns)
                                                          )
display(airbnb_df_short_reviews)
```

Overwrite Feature - here we have use overwrite instead of merge since we have deleted some features columns and want them to be removed from feature table entirely

```
fs.write_table(
    name =table_name,
    df = airbnb_df_short_reviews
    mode = "overwrite"  )
```

By navigating back to the UI we can see again that the modified data has changed and new column has been added to feature list . However note that the columns that we need deleted are also still present , in the next command we can see how the data store has changed , however the columns present in the original table will remain in the feature table

Now we can read in the feature data from feature store into a new dataframe .optionally we can use Delta Time Travel to read from a specific timestamp of the feature table

Note : the values of the deleted columns has been replaced by null

```
feature_df = fs.read_table(
    name=table_name
    )
display(feature_df)
```

if you have a use case to join feature for real time prediction you can publish your features to an online store and finally we can perform Access Control using built in features in the feature store UI

▼ **MD5 hash**

last data management technique

md5 hash which allows you to confirm that data has not been modified or corrupted though this does not give you a full diff if your data does not match

```
import hashlib
import pandas as pd

pd_df = airbnb-df.toPandas()

m1 = hashlib.md5(pd.util.hash_pandas_object(pd_df, index = True).values).hexdigest()
m1
```

now we have hexdigest of the original file , any changes that are made to the underlying DataFrame will result in a different hexdigest

```
try:
    assert m1 == m2 , "The dataset do not have the same hash"
    raise Exception("Exception Failure")
exception AssertionError:
    print('Failed as expected')
```

▼ **Review**

Why do we care about data management ?

Answer : Data management is an oftentimes overlooked aspect of end-to-end reproducibilty

how do we version data with delta tables ?

Answer: Delta Tables are automatically versioned everytime a new data is written .Accessing a previous version of the table is a simple as using display(spark.sql.(f"DESCRIBE HISTORY delta.'{delta_path}")) to find the version to revert to and loading it in . you can also revert to previous version using timestamps

what challenges does the feature shop help solve?

Answer a key issue many ML pipelines struggle with is feature reproducibility and data sharing . The feature store lets different users across the same organization utilize the same feature computation code

what does hashing a dataset help me do ?

Answer: It can help confirm whether a dataset is or is not the same as another . This is helpful is data reproducibility . it cannot however tell you full diff between two datasets

where can i learn more about delta tables ?

check out https://databricks.com/session_na21/intro-to-delta-lake

where can i learn about feature store ?

https://docs.databricks.com/applications/machine-learning/feature-store.html

where can i learn about reproducibility and its importance ?

https://databricks.com/blog/2021/04/26/reproduce-anything-machine-learning-meets-data-lakehouse.html

▼ **Experiment Tracking**

ML life cycle involves training multiple algorithms using different hyperparameters and libraries all with different performance and results and trained models .

this lesson help in tracking those experiment to organize machine learning life cycle

this lesson include

- Intro Tracking ML experiments in MLflow.
- log an experiment and explore the results in UI
- Record parameters ,metrics and model
- Query past runs programmatically

Over the course of machine learning life cycle

- data scientist may test different models and hyperparameter

Tracking various result poses an organizational challenge including

- Storing experiments
- Results
- Models
- Supplementary artifacts
- Code
- Data snapshots

▼ Tracking Experiments with MLFlow

MLflow tracking is

- a logging API specific for machine learning
- agonistic to libraries and environments that do the training
- organized around the concept of runs which are executions of data science code
- runs are aggregrated into experiments where many runs can be a part of a given experiment
- an MLflow server can host many experiments

Each run can record the following information

- Parameters : key - value pairs of input parameters such as the number of trees in a random forest model
- Metrics : Evaluation metrics such as RMSE or Area Under ROC Curve
- Artifacts : Arbitary output files in any format . This can include images,pickled models and data files
- Source : The code that originally ran the experiment

Experiments can be tracked using libraries in Python ,R and Java as well as using CLI and REST calls

Databricks hosts MLflow for you which reduces deployment configuration and adds security benefits , it is accessible on all Databricks workspaces in Azure , Google , AWS

See quick to setup on local or server : https://mlflow.org/docs/latest/quickstart.html#

Import a dataset of airbnb listings and featurize the data . we'll use to train a model

```
import pandas as pd

df = pd.read_csv("/dfbs/mnt/training/airbnb/sf-listings/airbnb-cleaned-mlflow.csv")
```

Perform Train/Test split

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(df.drop(["price"], axis=1), df[[
```

Navigate to the MLflow UI by clicking on the Runs button on the top of the screen

Every python notebook in Databricks Workspace has its own experiment . When you use MLflow in a notebook its records run the notebook experiment . A notebook experiment shares the share name and ID as its corresponding notebook.

Log a basic experiment by doing the following

1. Start a experiment using mlflow.start_run() and passing it a name for the run
2. Train your model
3. Log the model using mlflow.sklearn.log_model()
4. Log the model error using mlflow.log_metric()
5. Print out the run id using run.info_run_id

```
import mlflow.sklearn
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error

with mlflow.start_run(run_name="Basic RF Experiment") as run:
    rf = RandomForestRegressor(random_state=4)
    rf.fit(X_train,y_train)
    predictions = rf.predict(X_test)

    mlflow.sklearn.log_model(rf,"random-forest-model")

    mse = mean_squared_error(y_test,predictions)

    mlflow.log_metric("mse",mse)

    runID = run.info.run_id

    experimentID = run.info.experiment_id
```

Examine the results in the UI .Look for the following

1. the Experiment ID
2. The artifact location . This is where the artifacts are stored in DBFS which is backed by cloud storage
3. The time the run was executed .Click the timestamp to see more information on the run
4. The code that executed the run

After clicking on the time of the run take a look of the following

1. The Run ID will match what we printed above
    a. The model that we saved , included a picked version of the model as well as the Conda environment and the ML model file, which will be discussed in next lesson.

▼ Parameters , Metrics and Artifacts

last example you logged the run name , an evaluation metric and your model itself as an artifact

Now lets log parameter , multiple metrics and other artifacts including feature importances

First create to function to perform this

To log artifacts we have to save them somewhere before MLflow can log them . The code acomplishes that by using a temporary file that it then deletes

```
def log_rf(experimentID,run_name,params,X_train,X-test,y_train,y_test):
import os
import matplotlib.pyplot as plt
import mlflow.sklearn
import seaborn as sns
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error ,mean_absolute_error ,r2_score
import tempfile

with mlflow.start_run(experiment_id=experimentID,run_name=run_name) as run:
rf = RandomForestRegressor(**params)
rf.fit(X_train,y_train)
predictions = rf.predict(X_test)

mlflow.log_params(params)

mlflow.log({
"mse":mean_squared_error(y_test,predictions),
"mae":mean_absolute_error(y_test,predictions),
"r2" : r2_score(y_test,predictions)
})

#Log feature importance
importance = pd.DataFrame(list(zip(df.columns.rf.feature_importance_)),
                          columns = ["Feature","Importance"]
                          ).sort_values("Importance",ascending=False)
importance.to_csv(userhome+"/importance.csv",index=False)
mlflow.log_artifact(userhome+"/importance.csv","feature-importance.csv")

#Log plot
fig,ax = plt.subplots()
importance.plot.bar(ax=ax)
plt.title("Feature Importances")
fig.savefig(userhome+/"importance.png")
mlflow.log_artifact(userhome+"importance.png","importance.png")
display(fig)

return run.info.run_id
```

Run with new parameters

```
params = {
"n_estimators": 100,
"max_depth":5,
"random_state":42
}
log_rf(experiment_id,"Second_Run",params,X_train,X_test,y_train,y_test)
```

Check the UI to see how this appears . Take a look at the artifact to see where the plot was saved . Now , run a third run

```
params_100_trees = (
"n_estimators": 1000,
"max_depth": 10,
"random_state" : 42
)
log_rf(experimentID,"Third Run",params_1000_trees,X_train,X_test,y_train,y_test
```

▼ Querying Past Runs

you can query runs programatically in order to use this data back in python . The pathway to doing this is an MLFlowclient object

you can also set tags for runs using client.set_tag(run.info,run_id,"tag_key","tag_value")

```
from mlflow.tracking import MlflowClient
client = MlflowClient()
```

Now list all the runs for your experiment using .list_run_infos() , which takes your experiment *experiment_id* as a parameter

```
display(client.list_run_infos(experimentID))
```

We can list the artifacts for any run by using the MLflowClient().list_artifacts(run_id) method:

```
client.list_artifacts(runID)
```

Pull out a few fields and create a spark DataFrame with it

```
runs = spark.read.format("mlflow-experiment").load(experimentID)
display(runs)
```

Pull the last run and take a look at the associated artifacts

```
from pyspark.sql.functions import col
run_rf = runs.orderby(col("start_time").desc()).first()
client.list_artifacts(run_rf.run_id)
```

Return the evaluation metrics for the last run

```
client.get_run(run_rf.run_id).data.metrics
```

Reload the model and take a look at the feature importance

```
import mlflow.sklearn
model = mlflow.sklearn.load_model(run_rf.artifact_uri+"/random-forest-model")
model.feature_importances_
```

Question

What can MLflow Tracking log ?

MLflow can log the following

- Parameters : inputs to a model
- Metrics : the performance of the model

- Artifacts : any object including data , model and images
- Source : the original code, including the commit hash if linked to git

How do you log experiments ?

Experiments are logged by first creating a run and using the logging methods

on that run object (eg: run_log_params('MSE",.2))

Where do you logged artifacts get saved ?

Logged artifacts are saved in the directory of your choosing . On databricks this would be DBFS (Databricks File System)

How can i query past runs ?

This can be done using a Mlflowclient object . This allows you do everything you can within the UI programmatically so you never have to step outside of the programming environment

▼ Advanced MLflow Tracking

In this lesson more advanced MLflow tracking options tracking to extend a logger to wide variety of use case

you learn about

- Managed model inputs and outputs with MLflow signatures and input examples
- Explore nested runs for hyperparameter tuning and iterative training
- Integrate MLflow with HyperOpt
- Log SHAP values and visualizations

▼ Signatures and Input Examples

Previously logging a model in MLflow we only logged the model and name for the model artifacts with .log_model(model,model_name)

However it is best practice to also a log model signature and input examples . This allows for better schema checks and therefore integration with automated deployment tools

Signature

- A model signature is just the schema of the input(s) and the output(s) of the model
- We usually get this with the infer_schema function

Input Examples

- This is simply a few examples inputs to the model
- This will be converted to JSON and stored in our MLflow run
- It integrates well with MLflow model serving

In general logging a model with these looks like
.log_model(model,model_name.signature=signature ,input_example = input_example)

```
import pandas as pd
from sklearn.model_selection import train_test_split
pdf = pd.read_parquet("/dbfs/mnt/training/airbnb")
X = pdf.drop("price",axis=1)
```

```
Y = pdf['Price']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rand
```

let's train our model and log it with Mlflow . This time we will add a signature and input_examples when we log our model.

```
import mlflow
from mlflow.models.signature import infer_signature
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error

with mlflow.start_run(run_name="Signature Example") as run:
  rf = RandomForestRegressor(random_state=42)
  rf_model = rf.fit(X_train, y_train)
  mse = mean_squared_error(rf_model.predict(X_test), y_test)
  mlflow.log_metric("mse", mse)

  # Log the model with signature and input example
  signature = infer_signature(X_train, pd.DataFrame(y_train))
  input_example = X_train.head(3)
  mlflow.sklearn.log_model(rf_model, "rf_model", signature=signature, input_e
```

▼ Nested Runs

A useful organization tool provided by Mlflow is nested runs . Nested runs allow for parent runs and child runs in a tree structure . In the Mlflow UI you can click on a parent run to expand it and see the child runs

In hyperparameter tuning - you can nest all associated model runs under a parent run to better organize and compare hyperparameters

In parallel training many models such as IOT devices you can better aggregrate the models .

Iterative thinking - such as neural network you can checkpoint results after n epochs to save the model and related metrics

```
with mlflow.start_run(run_name="Nested Example") as run:
    with mlflow.start_run(run_name="Child 1",nested=True):
        mlflow.log_param("run_name","child_1")
    with mlflow.start_run(run_name="Child_2",nested=True):
        mlflow.log_param("run_name","child_2")
```

▼ Hyperparameter Tuning

One of the most common use case for nested runs is hyperparameter tuning . For example when running HyperOpt with SparkTrials on Databricks . it will automatically track the candidate models , parameter etc as child runs in the MLflow UI

HyperOpt allows for efficient hyperparameters tuning and now integrates with Apache Spark via

- **Trials -** sequential training of single node or distributed ML models (eg:MLlib)
- **SparkTrials** - Parallel training of single node models . The amount of parallelism is controlled via the parallelism parameter

Let's try using Hyperopt runs with SparkTrials to find the best sklearn random forest model

Set up the Hyperopt runs .We need to define an objective function to minimize and a search space for the parameters for our Hyperopt run

Hyperopt will work to minimize the objective function so here we simply return the loss as mse , since that is what we are trying to minimize

Note : if you 're trying to maximize a metric such as accuracy or r2 you would need to return -accuracy or -r2 so Hyperopt can minimize it

```
from hyperopt import fmin,type,hp,SparkTrials

def objective(params):
    model = RandomForestRegressor(n_estimators = int(params['n_estimators'
                    max_depth = int(params['max_depth'])
                    min_samples_leaf = int(params['min_samples_leaf'])
                    max_samples_leaf = int([params['min_sample_leaf'])
    model.fit(X_train,y_train)
    pred = model.predict(X_train)
    score = mean_squared_error(pred,y_train)
    #hyperopt minimize score here we minimize mse
    return score
```

Execute MLFlow HyperOpt runs

Note : The code using autologging You can also turn this on for other libraries such as mlflow.tensorflow_autolog()

from hyperopt import SparkTrials

```
search_space = {"n_estimators": hp.quniform("n_estimators", 100, 500, 5),
                "max_depth": hp.quniform("max_depth", 5, 20, 1),
                "min_samples_leaf": hp.quniform("min_samples_leaf", 1, 5,
                "min_samples_split": hp.quniform("min_samples_split", 2, 6


spark_trails = SparkTrials(parallelism=2)

with mlflow.start_run(run_name="HyperOpt"):
    argmin = fmin(
                    fn=objective,
                    space = search_space,
                    algo=tpe.suggest,
          max_eval = 16,
          trials = spark_trails)
```

if we select all the nested runs in this run and select compare we can also create useful visualizations to better understand the hyperparameter tuning process

Select compare as shown above and then Parallel Co-ordinate Plot in the next window to generate the following image

Note : you will have to add which parameters and metrics you want to generate visualization

▼ Advanced Artifact Tracking

In addition to the logging of artifacts you have already seen . there are some advanced options

we will now look at

- mlflow.shap - Automatically calculates and logs Shapley feature importance plots
- mlflow.log.figure - Logs matplotlib and plotly plots

```
import matplotlib.pyplot as plt

with mlflow.start_run(run_name="Feature Importance Scores"):
 mlflow.shap_log_explanation(rf.predict,X_train[:5])

 feature_importances = pd.Series(rf_model.feature_importances_,index=X.columr
 fig ,ax  = plt.subplots()
 feature_importances.plot.bar(ax=ax)
 ax.set_title("Features importances using MDI")
 ax.set_title("Mean decrease in impurity")
 mlflow.log_figure(fig,"feature_importance_rf.png")
```

▼ Section 2 : **Model Management**

▼ Model Management

An ml models is a standard format for packaging models that can be used on a variety of downstream tools . This lesson provides a generalizable way of handling machine learning models created in deployed to variety of environments

- Introduce model management best practices
- Store and use different flavor of models for different deployment environments
- Apply models combined with arbitrary pre and post processing code using Python models

▼ Managing Machine learning Models

Once a models has been trained and bundled with the environments it was trained in

The next steps is to package the model so that it can be used by a variety of serving tools

Current deployment options include

- Container based REST servers
- Continuous deployment using Spark streaming
- Batch
- Managed Platform such as Azure ML and Azure Sagemaker

  Packaging the final model in a platform agnostic way offers the most flexibility in deployment options and allows for model reuse across a no of platforms

MLflow models is a convention for packaging machine learning models that offers self- contained code, environments , and models

- The main abstraction is this package is the concept of flavors
- a falvours is a different ways the model can be used
- For instance , a Tensorflow model can be loaded as a tensorflow DAG or as a Python function
- Using an MLflow model convention allows for both of these flavours.
- The difference between projects and models is that models are for inference and serving while projects are for reproducibility

- The python_function flavour of models gives a generic ways of bundling ways
- we can thereby deploy a python function without worrying about the underlying format of the model

MLFLOW therefore maps any training framework to any deployment massively reducing complexity

artibitary pre and post processing step can be included such as data loading,cleansing and featurization . This means that the full pipeline , not just the model , can be preserved

Model Flavors

Flavors offer a way of saving models in a way that's agnostic to the training development ,making it significant easier to be used in various deployment options . Some of the most popular built in Flavors include the following

- mlflow.pyfunc
- mlflow.keras
- mlflow.pytorch
- mlflow.sklearn
- mlflow.spark
- mlflow.tensorflow

Model also offer reproducilibty since the run ID and the timestamp of the run are preserved as well.

To demonstrate the power of model flavours ,let create two models using different framework

Import the data

```
import pandas as pd
from sklearn.model_selection import train_test_split

df = pd.read_csv("/dbfs/mnt/training/airbnb/sf-listings/airbnb-cleaned-mlflow.csv")
X_train, X_test, y_train, y_test = train_test_split(df.drop(["price"], axis=1), df[[
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

rf = RandomForestRegressor(n_estimators=100, max_depth=5)
rf.fit(X_train, y_train)

rf_mse = mean_squared_error(y_test, rf.predict(X_test))

rf_mse
```

Train a neural network. Also, enable auto-logging. The autologger has produced a run corresponding to our single training pass. It contains the layer count, optimizer name, learning rate and epsilon value as parameters; loss and finally, the model checkpoint as an artifact.

```
import tensorflow as tf
import mlflow.tensorflow

tf.random.set_seed(42) # For reproducibility

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense


nn = Sequential([
  Dense(40, input_dim=21, activation='relu'),
  Dense(20, activation='relu'),
  Dense(1, activation='linear')
```

```
])

# Enable Auto-Logging
mlflow.tensorflow.autolog()

nn.compile(optimizer="adam", loss="mse")
nn.fit(X_train, y_train, validation_split=.2, epochs=40, verbose=2)

# nn.evaluate(X_test, y_test)
nn_mse = mean_squared_error(y_test, nn.predict(X_test))

nn_mse
```

Now log the two models .Make sure to add a model signature with the infer_signature function .This will allow you to easily view which parameters were factored into the model , and which model and which column was used as the output.

You can include input_example in mlflow.sklearn.log_model so that you can click on show examples for MLflow models Serving .

```
import mlflow.sklearn
from mlflow.models.signature import infer_signature
with mlflow.start_run(run_name="RF Model") as run:
  example = X_train.head(3)
  signature = infer_signature(X_train,y_train)
  mlflow.sklearn_log_model(rf,"model",signature=signature,input_example = example )
  mlflow.log_metric('mse",rf_mse)

  sklearnRunID = run.info.run_id
  sklearnURI = run.info.artifact_url
  experimentID = run.info.experiment_id
```

```
import mlflow.keras
with mlflow.start_run(run_name="NN Model") as run:
    example = X_train.head()
    signature = infer_signature(X_train,y_train)
    mlflow.keras.log_model(nn,"model",signature=signature,input_example=example)
    mlflow.log_metric("mse",nn_mse)
    kerasRunID = run.info.run_id
    kerasURI = run.info.artifact_uri
```

Now we can use both of these models in the same way even though they were trained by different packages

```
import mlflow.pyfunc

rf_pyfunc_model = mlflow.pyfunc.load_model(model_uri=(sklearnURI+"/model"))
type(rf_pyfunc_model)
```

```
import mlflow.pyfunc
nn_pyfunc_model = mlflow.pyfunc.load_model(model_uri=(KerasURI+"/model"))
type(nn_pyfunc_model)
```

both will implement a predict method . The sklearn model is still of type sklearn because the package natively implements the method

```
rf_pyfunc_model.predict(X_test)
nn_pyfunc_model.predict(X_test)
```

▼ Pre and Post Processing Code using Pyfunc

A pyfunc is a generic python model that can be define any model regardless of the libraries used to train it . As such it's defined as a directory structure with all the dependencies . It is then "just an object" with predict method .Since it make very few assumptions . It can be deployed using Mlflow , Sagemaker , a Spark UDF or in any other environment

To demonstrate how pyfunc works, create a basic class that adds n to the input values..

Define a model class

```
class ADDN(mlflow.pyfunc.PythonModel):
    def __init__(self,n):
     self.n = n
    def predict(self,context,model_input):
     return model_input.apply(lambda column: column + self.n)
```

Construct and save the model

```
import shutil

model_path = f"{working_dir}"/add_n_model2".replace("dfbs","dbfs")

try:
    shutil.rmtree(model_path)
except:
  None

add5_model = AddN(n=5)

mlflow.pyfunc.save_model(path=model_path,python_model = add5_model
```

Load the model in python_function format

```
loaded_model = mlflow.pyfunc.load_model(model_path)
```

Evaluate the model

```
import pandas as pd
model_input = pd.DataFrame([range(10)]

assert model_output.equals(pd.DataFrame([range(5,15)]
```

```
model_output
```

Review

Question: How do Mlflow projects differ from models ?

The focus of Mllfow projects is reproduciblity of runs and packaging of code .Mlflow models focuses on various deployments environment

Question: What is an ML Flavor ?

Answer : Flavors are a convention that deployment tools can use to understand the model , which makes it possible to write a tool that work with models from any ML library without having to integrate each tool with each library . Instead of having to map each training environment to a deployment environment ,ML model manages this mapping for you.

Question: How do i add pre and post processing logic to my models ?

Answer: A model class that extends mlflow.pyfunc.PythonModel allows you to have load,pre-processing and post-processing logic

▼ Model Registry

MLFlow Model Registry is a collaborative hub where teams can share ML models,work together from experimentation to online testing and production integrate with approval and goverance workflows and monitor ML deployments and their performance .

- Register a model using MLflow
- Deploy that model into production
- Update a model in production to a new version including a staging phase for testing
- Archive and delete models

Model Registry

The MLflow Model Registry component is a centralized model store , set of APIs and UI ,to collaboratively manage the full lifecycle of an MLflow Model. It provides model lineage (which Mlflow Experiment and Run produced the model) , model versioning ,stage transitions (eg from staging to production) annotations (eg with comments,tags) and deployment management (eg: which production) job have requested a specific model version)

Model Registry has the following features.

Central Repository : Register MLflow with the MLflow Model Registry . A registered model has a unique name,version .stage and other metadata.

Model Versioning : Automatically keep track of versions for registered models when updated

Model Stage: Assigned preset or custom stages to each model version like “Staging” and “Production” to represent the lifecycle of the model

Model Stage Transitions : Record new registrations events or changes as activities that automatically log users , changes and additional metadata as comments

CI/CD Workflow Integrations : Record stage transitions , request ,review and approve changes as part of CI/CD pipelines for better control and governance

Registering a Model

The workflow will work with either the UI or in pure Python . The notebook will use pure Python

Train a model and log it MLflow

```
import mlflow
import mlflow.sklearn
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.model_selection import train_test_split
from mlflow.models.signature import infer_signature

df = pd.read_csv("/dbfs/mnt/training/airbnb/sf-listings/airbnb-cleaned-mlflow.csv")
X_train, X_test, y_train, y_test = train_test_split(df.drop(["price"], axis=1), df["pri

rf = RandomForestRegressor(n_estimators=100, max_depth=5)
rf.fit(X_train, y_train)

input_examples = X_train.head()
signature = infer_signature(X_train,pd.DataFrame(y_train))

with mlflow.start_run(run_name="RF Model") as run:
    mlflow.sklearn.log_model(rf,"model",input_example=input_example,signature=signature
    mlflow.log-metric("mse",mean-squared_error(y_test,rf.predict(X_test)))
    runID = run.info.run_id
```

Create a unique model name so you don't clash with other workspace user

```
import uuid
model_name = f"airbnb_rf_model(uuid.uuid().hex[:10]}"
model_name
```

Register a model

```
model_uri = "runs:/{runs_id}model".format(run_id=runID)
model_details = mlflow.register_model(model_uri=model_uri,name=model_name)
```

Open the models tab on the left of the screen to explore the registered models

- It logged who trained the model and what code was used
- It logged the history of actions taken on this model
- It logged this model as a first version

Check the status . It will intitally be in PENDING_REGISTRATION status

```
from mlflow.tracking.client import MlflowClient

client = MlflowClient()

model_version_details = client.get_model_version(name=model_name,version=1)
```

```
model_version_details.status
```

now add a model description

```
client.update_registered_model(
name = model_details.name,
description="This model forecasts Airbnb housing list prices based on various listing i
)
```

Add a version specific description

```
client.update_model_version(
name=model_details.name,
version=model_details.version,
```

▼ Deploying a Model

The MLflow Model Registry defines a several model stages : None ,Staging,Production and Archived .Each stage has a unique meaning For example : Staging is meant for model testing While Production is for models that have completed the testing or review process and have been deployed to applications

user with appropriate permission can transition models between stages In private preview any user transition a model to any stage In the near future administrators in your organizations will be able to control these permissions on a per-user and per-model basis

if you permission to transition a model to a particular stage you can make the transition directly the MlflowClient.update_model_version() function . if you do not have permission you can request a stage transition using the REST API

preview/mlflow/transition-request

Now that you learned about stage transition ,transition to the model to the Production stage

```
import time
time.sleep(10)
```

```
client.transition_model_version_stage(
name=model_details.name,
version = model_details.version,
stage="Production"
)
```

Fetch the model's current status

```
model_version_details = client.get_model_version(
name=model_details.name,
version=model_details.version,
)
#for stage we have to fetch the variable model_version_details.current_stage
```

Fetch the latest model using a pyfunc loading the model in this way allows us to use the model regardless of the package that was used to train it

you can load a specific version of the model too

```
import mlflow.pyfunc
model_version_uri = "models:/{model_name}/1".format(model_name)
print("loading registered model version from URI " model_uri)
model_version_1 = mlflow.pyfunc.load_model(model_version_uri)
```

Apply the model

```
model_version_1.predict(X_test)
```

▼ Deploying a New Model Version

The Mlflow Model registry enables you to create multiple versions corresponding to a single registered model . By performing stage transitions you can seamlessly integrate new model versions into your staging or production environments

Create a new model version and register the model when it's logged

```
import mlflow
import mlflow.sklearn
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.model_selection import train_test_split

df = pd.read_csv("/dbfs/mnt/training/airbnb/sf-listings/airbnb-cleaned-mlflow.csv")
X_train, X_test, y_train, y_test = train_test_split(df.drop(["price"], axis=1), df["

rf = RandomForestRegressor(n_estimators=300, max_depth=10)
rf.fit(X_train, y_train)

input_example = X_train.head(3)
signature = infer_signature(X_train, pd.DataFrame(y_train))

with mlflow.start_run(run_name="RF_model") as run:
    mlflow.sklearn.log_model(
    sk_model = rf,
    artifact_path = "sklearn-model",
    registered_model_name = model_name,
    input_examples = input_examples,
    signature = signature,
    )
mlflow.log_metric("mse",mean_squared_error(y_test,rf.predict(X_test))

runID = run_info.run_id
```

Use the search functionality to grab the latest model version.

model_version_infos = client.search_model_versions(f"name = '{model_name}'")

new_model_version = max([model_version_info.version for model_version_info in model_version_infos])

Add a description to this new version.

client.update_model_version(
name=model_name,
version=new_model_version,
description="This model version is a random forest containing 300 decision trees and a max depth of 10 that
in scikit-learn."
)

Put this new model version into `Staging`

import time
time.sleep(10) # In case the registration is still pending

client.transition_model_version_stage(
name=model_name,
version=new_model_version,
stage="Staging",
)

Sicne this model is now in staging, you can execute an automated CI/CD pipeline against it to test it before going into production. Once that is completed, you can push that model into production.

client.transition_model_version_stage(
name=model_name,
version=new_model_version,
stage="Production",
)

▼ Archiving and Deleting

You can archieve and delete old versions of the model

```
client.transition_model_version_stage(
name=model_name,
version=1,
stage="Archived"
)
```

Deleted version 1 : You cannot delete a model that is not first achieved

```
client.delete_model_version(
name=model_name,
version=1
)
```

Archive version 2 of the model too

```
client.transition_model_version_stage(name=model_name,version=2,stage="Archived")
```

Now delete the entire registered model

client.delete_registered_model(model_name)

▼ Review

Que : How does MLlfow tracking differ from the model registry?

Ans : Tracking is meant for experimentation and development . The model registry is designed to take a model from tracking and put it through staging and into production

This is often the point that a data engineer of machine learning engineer takes responsibility for the deployment process

Why do i need model registry ?

Just as Mlflow tracking provides end - to - end reproducibility for the machine learning training process a model registry provides reproduciblity and goveraance for the deployment process. Since production system are mission critical , components can be isolated with ACL so only specific individuals can alter production models. Version control and CI/CD workflow integration is also a critical dimension of deploying models into production

What can i do programmatically versus using the UI ?

Most operations can be done using the UI or in pure python . A model must be tracked using python but from that point on everything can be done either way . For instance , a model logged using the MLflow tracking API can then be registered using the UI and can then be pushed into production

▼ Webhooks and Testing

MLFlow Webhook and Testing

Webhook trigger the execution of code (oftentimes tests) upon some events .This lesson explorer how to employ webhooks to trigger automated tests against models in the model registry

In this lesson you will learn

- Explore the role of webhooks in ML pipelines
- Create a Job to test models in the model registry
- Examine a job that imports a model and runs tests
- Automate the job using MLflow webhooks

Automated Testing

The backbone of the continuous integration , continuous deployment (CI/CD) process is the automated building ,testing and deployment of the code .A webhook or trigger causes the execution of code based on some events . This is commonly when new code is pushed to a code repository . In case of machine learning jobs. this could be the arrival of a new model in the model registry.

This lesson uses Mlflow webhooks to trigger the execution of a job upon the arrival of a new model with given name in the model registry . The function of the job is to

- import the new model version
- Test the schema of its inputa and outputs
- Pass example code through the model

This covers many of the dersired tests for ML models. however throughput testing could be performed using this paradigm .Also the model could also be promoted to the production stage in a automated fashion

Creating a Job

create a databricks job using another notebook in this directory

ensure you are admin on this workspace and that you re not using community edition (which has job disabled)

Create a user access token using the following steps

1. Click the user profile icon User Profile in the upper right corner of your databricks workspace
2. Click User Settings
3. Go to the Access Token User
4. Click the Generate New Token button
5. Optionally enter a description and expiration period
    a. click on generated button and copy the generaated token and paste

```
from mlflow.models.signature import infer_signature
from sklearn.metrics import mean_squared_error
import mlflow.sklearn
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split

with mlflow.start_run(run_name="Webhook RF Experiment") as run:
  # Data prep
  df = pd.read_csv("/dbfs/mnt/training/airbnb/sf-listings/airbnb-cleaned-mlflow.csv")
  X_train, X_test, y_train, y_test = train_test_split(df.drop(["price"], axis=1), df["p
  signature = infer_signature(X_train, pd.DataFrame(y_train))
  example = X_train.head(3)

  # Train and log model
  rf = RandomForestRegressor(random_state=42)
  rf.fit(X_train, y_train)
  mlflow.sklearn.log_model(rf, "random-forest-model", signature=signature, input_exampl
  mse = mean_squared_error(y_test, rf.predict(X_test))
  mlflow.log_metric("mse", mse)
  run_id = run.info.run_id
  experiment_id = run.info.experiment_id
```

Register the model

```
name = f"webhook_demo_{run_id}"
model_uri = "runs:/{run_id}/random-forest-model".format(run_id=run_id)

model_details = mlflow.register_model(model_uri=model_uri, name=name)
```

Create a Webhook

There a few different events that can trigger a webhook in this notebook we will experimenting with triggering a job when our model transitions between stages

```
    #we need an endpoint at which to execute our request which we can get from the note
    java_tags = dbutils.notebook.entry_points.getDbutils().notebook().getContext().tags
    headers = {"Authorization":"Bearer {token}"}
```

```
    #next we need an endpoint at which to execute our request which we can get from the
    java_tags = dbutils.notebook.entry_point.getDbutils().notebook().getContext().tags(
# This ojbect comes from the Java CM - Convert the Java Map opject to a Python dictiona
tags = sc._jvm.scala.collection.JavaConversions.mapAsJavaMap(java_tags)
# Lastly, extract the databricks instance (domain name) from the dictionary
instance = tags['browserHostName']
```

```
import json
import urllib

url = f"https://{instance}"
endpoint = f"{url}/api/2.0/mlflow/registry-webhooks/create"

new_json = {"model_name": name,
  "events": ["MODEL_VERSION_TRANSITIONED_STAGE"],
  "description": "Job webhook trigger",
  "status": "Active",
  "job_spec": {
    "job_id": job_ID,
    "workspace_url": url,
    "access_token": token}}

json_body = json.dumps(new_json)
json_bytes = json_body.encode('utf-8')

req = urllib.request.Request(endpoint, data=json_bytes, headers=headers)
response = urllib.request.urlopen(req)

print(json.load(response))
```

Now that we have registered the webhook, we can **test it by transitioning our model from stage `None` to `Staging` in the Experiment UI.** We should see in the Jobs tab that our Job has run.

To get a list of active Webhooks, use a GET request with the LIST endpoint. Note that this command will return an error if no Webhooks have been created for the Model.

```
new_url = f"{url}/api/2.0/mlflow/registry-webhooks/list/?model_name={name.replace(' ',
req = urllib.request.Request(new_url, headers=headers)
response = json.load(urllib.request.urlopen(req))

for webhook in response.get("webhooks"):
  print(webhook)
  print()
```

Finally, delete the webhook by copying the webhook ID to the curl or python request. You can confirm that the Webhook was deleted by using the list request.

```
delete_hook = "<insert your webhook id here>"

json_bytes = json.dumps({"id": delete_hook}).encode('utf-8')

req = urllib.request.Request(f"{url}/api/2.0/mlflow/registry-webhooks/delete", data=jso
response = urllib.request.urlopen(req)
```

```
print(json.load(response))
```

▼ Webhooks Job Demo

Load the model name. The `event_message` is automatically populated by the webhook.

```
import json

event_message = dbutils.widgets.get("event_message")
event_message_dict = json.loads(event_message)
model_name = event_message_dict.get("model_name")

print(event_message_dict)
print(model_name)
```

Use the model name to get the latest model version.

```
from mlflow.tracking import MlflowClient
client = MlflowClient()

version = client.get_registered_model(model_name).latest_versions[0].version
version
```

Use the model name and version to load a `pyfunc` model of our model in production.

```
import mlflow

pyfunc_model = mlflow.pyfunc.load_model(model_uri=f"models:/{model_name}/{version}")
```

Get the input and output schema of our logged model.

```
input_schema = pyfunc_model.metadata.get_input_schema().as_spark_schema()
output_schema = pyfunc_model.metadata.get_output_schema().as_spark_schema()
```

Here we define our expected input and output schema.

```
from pyspark.sql.types import StructType, StructField, LongType, DoubleType

expected_input_schema = (StructType([
  StructField("host_total_listings_count", DoubleType(), True),
  StructField("neighbourhood_cleansed", LongType(), True),
  StructField("zipcode", LongType(), True),
  StructField("latitude", DoubleType(), True),
  StructField("longitude", DoubleType(), True),
  StructField("property_type", LongType(), True),
  StructField("accommodates", DoubleType(), True),
  StructField("bathrooms", DoubleType(), True),
  StructField("bedrooms", DoubleType(), True),
  StructField("beds", DoubleType(), True),
```

```
  StructField("bed_type", LongType(), True),
  StructField("minimum_nights", DoubleType(), True),
  StructField("number_of_reviews", DoubleType(), True),
  StructField("review_scores_rating", DoubleType(), True),
  StructField("review_scores_accuracy", DoubleType(), True),
  StructField("review_scores_cleanliness", DoubleType(), True),
  StructField("review_scores_checkin", DoubleType(), True),
  StructField("review_scores_communication", DoubleType(), True),
  StructField("review_scores_location", DoubleType(), True),
  StructField("review_scores_value", DoubleType(), True)
]))

expected_output_schema = StructType([StructField("price", DoubleType(), True)])
```

```
assert expected_input_schema.fields.sort(key=lambda x: x.name) == input_schema.fields.s
assert expected_output_schema.fields.sort(key=lambda x: x.name) == output_schema.fields
```

Load the dataset and generate some predictions to ensure our model is working correctly.

```
import pandas as pd

df = pd.read_csv("/dbfs/mnt/training/airbnb/sf-listings/airbnb-cleaned-mlflow.csv")
predictions = pyfunc_model.predict(df)

predictions
```

Make sure our prediction types are correct.

```
import numpy as np

assert type(predictions) == np.ndarray
assert type(predictions[0]) == np.float64
```

print("All tests passed!")

▼ Section 3: **Deplopyment Pardigm**

Batch Deployment

Batch inference is the most common way of deploying machine learning models. This lesson introduces various strategies for deploying models using batch including Spark, write optimizations, and on the JVM.

- Explore batch deployment options
- Apply an sklearn model to a Spark DataFrame and save the results
- Employ write optimization including partitioning .bucketing and z-order
- Compare other batch deployment options
-