10.22 : Introduction to MLflow

```
### 이번 DataCamp 강의를 수강하면서 배운점
- MLflow가 어떤식으로 구성되어있고,
어떤 식으로 사용하는 것인지 개론적인 내용을 알 수 있었고, 익숙해질 수 있는
### 강의 내용중 궁금했던점
- 없습니다.
### 함께 이야기 나누며 얻은 것
- 시대의 흐름에 맞게, MLflow도 업데이트 되고 있다.
staging을 표시하는 과정은 2.9 version에선 derpreciated되었다고 합니다
```

What is MLflow?

The machine learning lifecycle

00:07 - 00:28

The machine learning lifecycle follows the same principles as traditional software development but includes additional challenging steps such as model engineering, evaluation, and deployment.

Difficulties of machine learning

00:28 - 00:50

These extra steps are complex and often difficult for organizations. Tracking experiments and models is challenging, and it's hard to reproduce code across platforms and environments. Deployment lacks standardized methods.

What is MLflow?

00:50 - 01:08

MLflow is an open-source platform that simplifies these challenges by providing tools for experiment tracking, reproducibility, deployment, and a centralized model registry.

Components of MLflow

01:08 - 02:03

MLflow consists of four components:

- **Tracking**: Records and queries experiment data, storing models and artifacts for easy retrieval.
- **Models**: Standardizes model packaging for streamlined deployment with customization options for model inference.
- **Model Registry**: Central storage with version control for models, tagged by development environment.
- **Projects**: Standardizes ML code packaging for reproducibility across platforms, aiding automation.

MLflow adoption

02:03 - 02:17

MLflow has been widely adopted across industries due to its simplicity and success in managing the ML lifecycle.

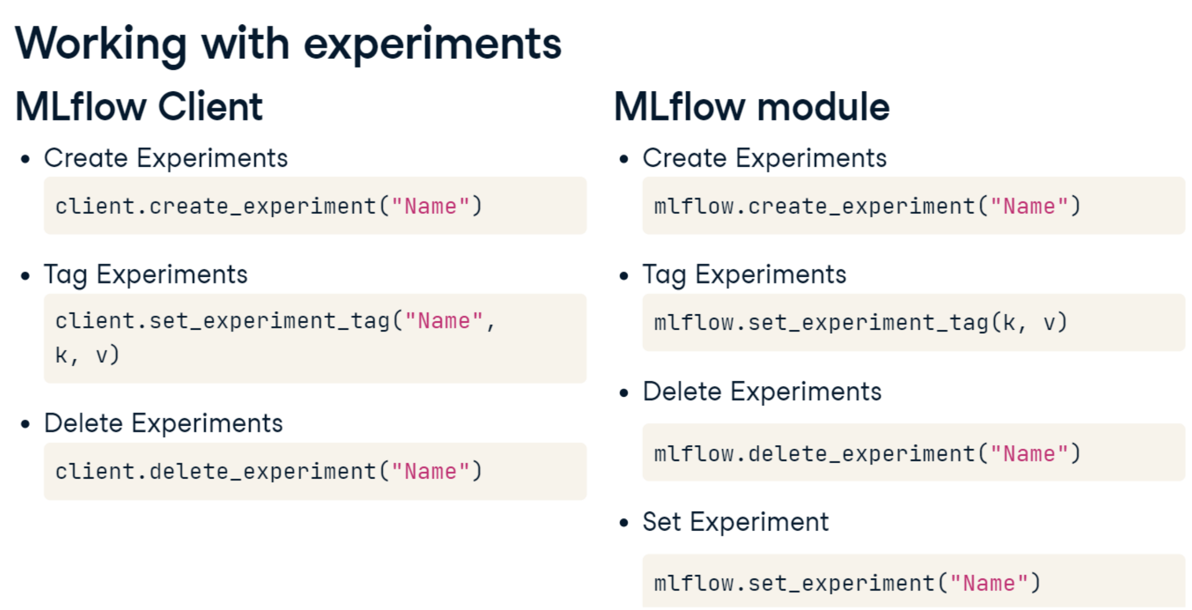
MLflow experiments

02:17 - 02:39

An MLflow experiment organizes and tracks model training runs. When tracking data, you must specify which experiment to use to find the data later.

Working with experiments

02:39 - 03:26



You can interact with experiments via the MLflow client (a lower-level API) or the MLflow module (higher-level API). Both allow creating, tagging, and deleting experiments. The MLflow module can set an experiment for current training runs.

Starting a new experiment

Starting a new experiment

```python
import mlflow
# Create new Experiment
mlflow.create_experiment("My Experiment")
# Tag new experiment
mlflow.set_experiment_tag("scikit-learn", "lr")
# Set the experiment
mlflow.set_experiment("My Experiment")
```

03:26 - 03:54

When starting a new ML application, use the `create_experiment` function to start a new experiment and add tags. Use `set_experiment` to assign the experiment so that MLflow knows where to store the data.

MLflow Tracking

Tracking data about models

00:10 - 00:32

Model performance relies on metrics generated during training, which can vary with changes in code, data, or model parameters. Keeping track of these metrics and other artifacts used during each model training iteration becomes increasingly difficult as the number of training runs grows.

What is MLflow Tracking?

00:32 - 00:54

MLflow Tracking solves this issue by providing an API to track metrics, parameters, and artifacts like code files. In MLflow, this process is referred to as “logging” data or artifacts to the MLflow Tracking system.

Training runs

00:54 - 01:12

MLflow Tracking uses the concept of “runs” to organize model training data. A new run represents a new model training session, and all data is logged to MLflow. Each run belongs to an experiment, and a run can be initiated using the `start_run` function.

Starting a training run

01:12 - 01:32

Starting a training run

```python
import mlflow

# Start a run
mlflow.start_run()
```

```
<ActiveRun: >
```

```python
# End a run
mlflow.end_run()
```

When a training run starts, it becomes active, and all metrics, parameters, and artifacts logged will be associated with this active run until the run is manually ended or the script completes.

Setting a training run variable

01:32 - 02:15

Setting a training run variable

```python
import mlflow
# Set experiment
mlflow.set_experiment("My Experiment")
# Start a run
run = mlflow.start_run()
# Print run info
run.info
```

```
<RunInfo: artifact_uri='./mlruns/0/9de5df4d19994546b03dce09aefb58af/artifacts',
  end_time=None, experiment_id='31', lifecycle_stage='active',
  run_id='9de5df4d19994546b03dce09aefb58af', run_name='big-owl-145',
  run_uuid='9de5df4d19994546b03dce09aefb58af', start_time=1676838126924,
  status='RUNNING', user_id='user'>
```

You can also store the return value of `start_run` in a variable to access metadata like `artifact_uri` or `run_id`. This helps identify the location of resources linked to the run. By setting the `experiment_id`, you ensure all logged data belongs to the correct experiment.

Logging to MLflow Tracking

02:15 - 02:55

- Metrics
  - `log_metric("accuracy", 0.90)`
  - `log_metrics({"accuracy": 0.90, "loss": 0.50})`
- Parameters
  - `log_param("n_jobs", 1)`
  - `log_params({"n_jobs": 1, "fit_intercept": False})`
- Artifacts
  - `log_artifact("file.py")`
  - `log_artifacts("./directory/")`

Logging involves saving metrics, parameters, and artifacts to MLflow Tracking. Use functions like `log_metric` for a single metric or `log_metrics` for multiple metrics (by passing a dictionary). For parameters, use `log_param` or `log_params`. Artifacts can be logged using `log_artifact` for a file or `log_artifacts` for a directory of files.

Logging a run

02:55 - 03:28

```
import mlflow
# Set Experiment
mlflow.set_experiment("LR Experiment")

# Start a run
mlflow.start_run()

# Model Training Code here
lr = LogisticRegression(n_jobs=1)

# Model evaluation Code here
lr.fit(X, y)
score = lr.score(X, y)
```

```
# Log a metric
mlflow.log_metric("score", score)

# Log a parameter
mlflow.log_param("n_jobs", 1)

# Log an artifact
mlflow.log_artifact("train_code.py")
```

To log a run, set the desired experiment, activate the run with `start_run`, train the model, and log the results to MLflow Tracking. You can log metrics like a model's score, parameters such as the number of cores, and even the Python file used for training.

Open MLflow UI

03:28 - 03:39

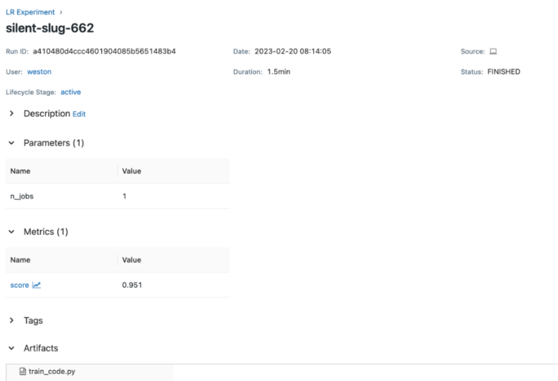
```
# Open MLflow Tracking UI
mlflow ui
```

Go to: http://localhost:5000

To view the MLflow Tracking UI, run a command in the terminal to start the UI, then access it in your browser at localhost:5000.

Tracking UI experiment view

03:39 - 03:57



In the UI, the experiment view shows all the runs logged under the specified experiment, including metrics for comparison across runs.

Tracking UI run view

03:57 - 04:10

Clicking on a specific run allows you to explore detailed information about its metrics, parameters, artifacts, and state.

Querying runs

Querying runs

00:00 - 00:11

MLflow Tracking allows us to log metrics, parameters, and artifacts to a centralized location during the model engineering and evaluation phases of the ML lifecycle.

Model data

00:11 - 00:23

After building and experimenting with many models, we need to choose the best one for our ML application by comparing metrics and other logged data.

Runs data

00:23 - 00:40

The MLflow Tracking UI provides a view of all runs within an experiment but lacks the ability to easily compare or analyze the data. Fortunately, MLflow allows us to query run data programmatically for deeper analysis.

Searching runs

00:40 - 01:05

Searching runs

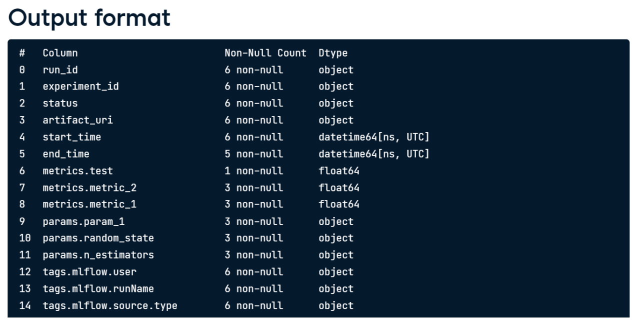
```
mlflow.search_runs()
```

pandas

The `search_runs` function in the MLflow module provides access to query runs programmatically. The output can be used with tools like the pandas library for further analysis. Pandas is the default output format for the `search_runs` function.

Output format

01:05 - 01:37

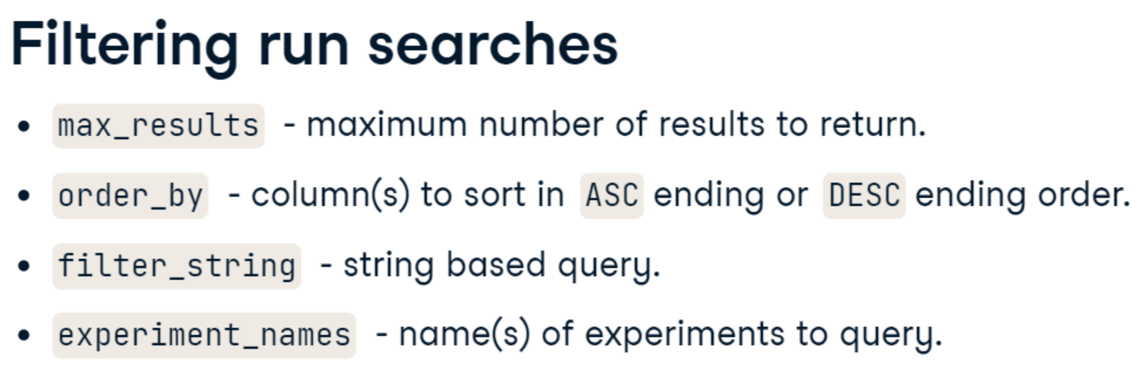

Output format

#	Column	Non-Null Count	Dtype
0	run_id	6 non-null	object
1	experiment_id	6 non-null	object
2	status	6 non-null	object
3	artifact_uri	6 non-null	object
4	start_time	6 non-null	datetime64[ns, UTC]
5	end_time	5 non-null	datetime64[ns, UTC]
6	metrics.test	1 non-null	float64
7	metrics.metric_2	3 non-null	float64
8	metrics.metric_1	3 non-null	float64
9	params.param_1	3 non-null	object
10	params.random_state	3 non-null	object
11	params.n_estimators	3 non-null	object
12	tags.mlflow.user	6 non-null	object
13	tags.mlflow.runName	6 non-null	object
14	tags.mlflow.source.type	6 non-null	object

The output of `search_runs` is a pandas DataFrame where each metric and parameter has its own column, prefixed by "metrics." for metrics and "params." for parameters. Other columns include `run_id`, status, start and end times, and tags.

Filtering run searches

01:37 - 02:10



`search_runs` is flexible and accepts arguments like `max_results` to limit the number of runs, `order_by` to sort columns, and `filter_string` to query specific runs based on conditions. The `experiment_names` argument allows users to query runs from specific experiments, even specifying multiple experiments.

Tracking UI

02:10 - 02:26

Let's apply `search_runs` to an experiment with four runs that contain metrics and parameters, using it to retrieve and format run data for analysis.

Search runs example

02:26 - 02:56

Search runs example

```
import mlflow
# Filter string
f1_score_filter = "metrics.f1_score > 0.60"
# Search runs
mlflow.search_runs(experiment_names=["Insurance Experiment"],
    filter_string=f1_score_filter,
    order_by=["metrics.precision_score DESC"])
```

If we want to query runs from the "Insurance Experiment" where `f1_score` is greater than 0.6 and order the results by `precision_score` in descending order, we can define our filter string and pass it to the `search_runs` function.

Example output

```
# Search runs from Insurance Experiment
mlflow.search_runs(experiment_names=["Insurance Experiment"],
    filter_string=f1_score_filter,
    order_by=["metrics.precision_score DESC"])
```

```
                             run_id experiment_id  ... tags.mlflow.source.type tags.mlflow.user
0  90407e29a5aa4a31954bed874c7d4337            27  ...                   LOCAL             user
1  c335c0b16a5d4cf398aaa7189362b577            27  ...                   LOCAL             user
```

Example output

02:56 - 03:02

The query results return two runs where the `f1_score` is greater than 0.6, sorted by `precision_score` in descending order.

Introduction to MLflow Models

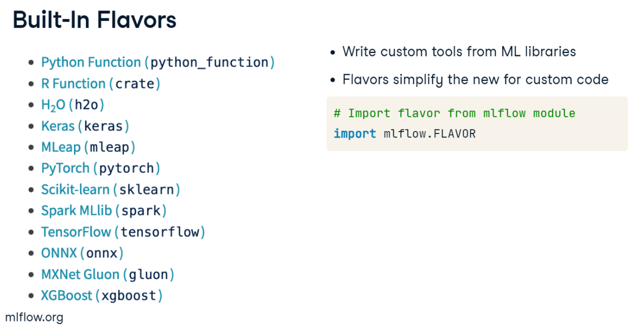
Introduction to MLflow Models 00:00 - 00:06

MLflow provides a component called MLflow Models to standardize the packaging of machine learning models.

MLflow Models 00:06 - 00:31

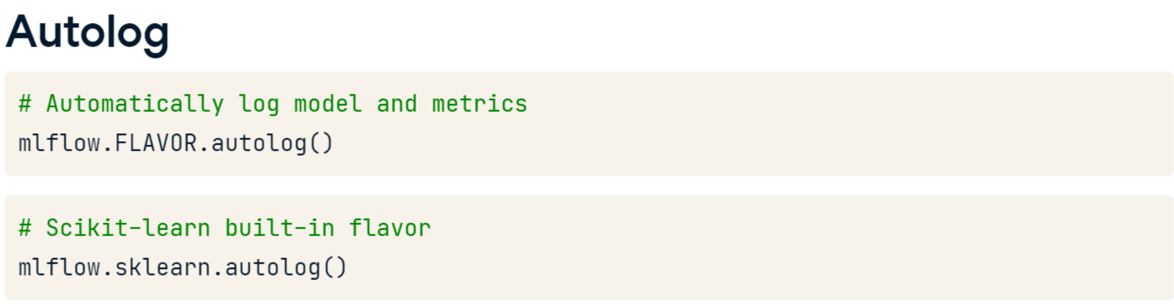
Standardizing models allows for easier integration between popular ML libraries and deployment tools. Packaging involves organizing all necessary application files and resources to enable easier distribution. This standardization is achieved through "Flavors," which are conventions that allow models to be saved in formats that different downstream tools can understand.

Built-In Flavors 00:31 - 00:56



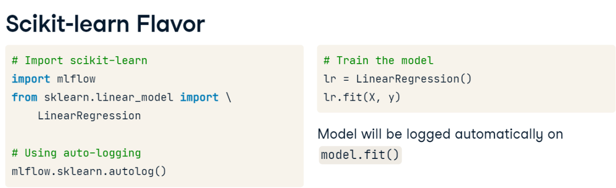
MLflow Models use "Flavors" to work with models from popular ML libraries. Flavors simplify the logging, packaging, and loading processes, reducing the need for custom code. To use a flavor, simply import it from MLflow.

Autolog 00:56 - 01:21



MLflow supports auto logging through the `autolog` method. Auto logging automatically handles the logging of metrics, parameters, and models without needing explicit logging statements. The scikit-learn Flavor can use this method, which logs these items when a model is fit.

Scikit-learn Flavor 01:21 - 01:47



By training a LinearRegression model, we can track the training run to MLflow Tracking. The scikit-learn Flavor, combined with the `autolog` method, logs a variety of metrics and parameters, along with the model, when the `.fit` method is called.

Autolog common metrics and parameters 01:47 - 02:28

- Regression
  - mean squared error
  - root mean squared error
  - mean absolute error
  - r2 score
- Classification
  - precision score
  - recall score
  - f1 score
  - accuracy score

```
MODEL.get_params()
```

Autolog automatically logs common metrics and parameters for both regression and classification models. For regression, metrics like mean squared error, root mean squared error, and r2 score are logged, while for classification, metrics such as precision, recall, and accuracy are captured. Parameters are gathered using the `get_params` method from the model.

Common parameters 02:28 - 02:35

We can retrieve parameters from our model with the `get_params` method, which logs these values to MLflow.

Autolog parameters 02:35 - 02:41

Metrics (5)

Name	Value
training_mean_absolute_error	6640.5
training_mean_squared_error	89637953.1
training_r2_score	0.952
training_root_mean_squared_error	9467.7
training_score	0.952

```
# Model
lr = LinearRegression()
lr.fit(X, y)
```

In the MLflow Tracking UI, autolog automatically logs default parameters for the model.

Autolog metrics 02:41 - 02:51

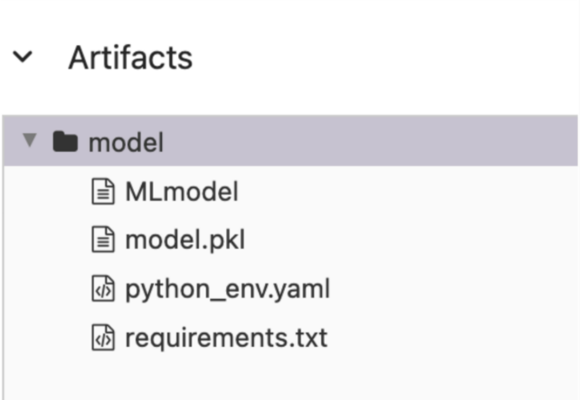
Even without explicitly specifying the metrics to log, autolog captures several common metrics automatically and logs them to MLflow Tracking.

Storage format 02:51 - 03:06

Directory structure for a model:

```
model/
    MLmodel
    model.pkl
    python_env.yaml
    requirements.txt
```

```
# Autolog
mlflow.sklearn.autolog()
```



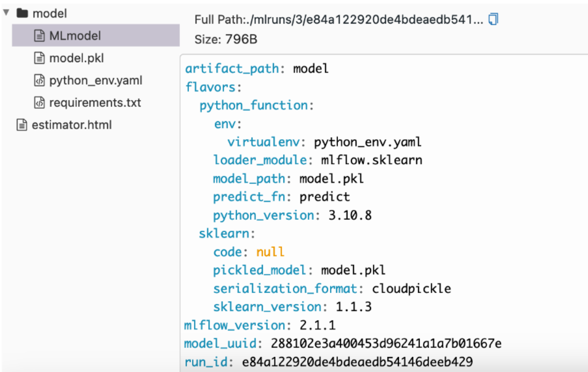
MLflow packages models using a standardized directory structure. In our example, the Scikit-learn Flavor is used to autolog the model to an MLflow tracking server.

Contents of MLmodel 03:06 - 03:33

```
artifact_path: model
flavors:
  python_function:
    env:
      virtualenv: python_env.yaml
    loader_module: mlflow.sklearn
    model_path: model.pkl
    predict_fn: predict
    python_version: 3.10.8
  sklearn:
    code: null
    pickled_model: model.pkl
    serialization_format: cloudpickle
    sklearn_version: 1.1.3
```

The `MLmodel` file is a YAML configuration file that provides key information about the model, including how to load it, the virtual environment, Python version, and scikit-learn version used for training.

MLmodel 03:33 - 03:55



The `MLmodel` file in this example contains two flavors: Scikit-learn and `python_function`. This means the model can be loaded using either of these flavors, with `python_function` being a generic flavor that supports customization while offering the same utilities.

Model API

Model API 00:00 - 00:10

MLflow Models standardize how ML models are packaged. Let's now explore the ways MLflow uses the Model API to save and load models.

MLflow REST API 00:10 - 00:28

MLflow uses a REST API that enables users to programmatically create, list, and retrieve information from every component of MLflow. An API, or application programming interface, allows two software components to communicate through a set of definitions.

The Model API 00:28 - 00:39

The Model API is used to interact with models, allowing users to save, log, and load MLflow models using different flavors.

Model API functions 00:39 - 01:07

Model API functions

```
# Save a model to the local filesystem
mlflow.sklearn.save_model(model, path)
```

```
# Log a model as an artifact to MLflow Tracking.
mlflow.sklearn.log_model(model, artifact_path)
```

```
# Load a model from local filesystem or from MLflow Tracking.
mlflow.sklearn.load_model(model_uri)
```

MLflow integrates with several popular libraries like scikit-learn. Using the `mlflow.sklearn` module, users can:

- Save models to the local filesystem with the `save_model` function.
- Log models to MLflow Tracking as artifacts with the `log_model` function.
- Load models from the local filesystem or MLflow Tracking with the `load_model` function.

Load model 01:07 - 01:34

When loading models, MLflow supports various location formats. The `load_model` function can load models from the local filesystem using relative or absolute paths or from MLflow Tracking by specifying a "runs" format that includes the run ID and model path. MLflow also supports loading from cloud storage like AWS S3.

Save model 01:34 - 01:53

In the example, we use a logistic regression model from scikit-learn and save it locally with the `save_model` function. Using the command line `ls`, we can verify that the model is saved locally using MLflow’s standardized storage format.

Load local model 01:53 - 02:05

Load local model

```
# Load model from local path
model = mlflow.sklearn.load_model("local_path")

# Show model
model
```

```
LogisticRegression()
```

To load the saved model from the local filesystem, we use the `load_model` function and provide the relative path. Once loaded, we print the model to confirm success.

Log model 02:05 - 02:24

Using the same model, we log it to MLflow Tracking with the `log_model` function and define the path as "tracking_path." Unlike `autolog`, `log_model` allows users to specify additional options like the artifact path or model name.

Tracking UI 02:24 - 02:38

In the MLflow Tracking UI, we can see that our model was logged as an artifact under "tracking_path" using the standardized storage format.

Last active run 02:38 - 02:58

To load a model from MLflow Tracking, we need the run ID. MLflow provides the `last_active_run` function, which returns metadata about the latest run using the `Run` class.

Last active run id 02:58 - 03:04

Using the `Run` class's `info` property, we can retrieve the run ID.

Setting the run id 03:04 - 03:17

Setting the run id

```
# Get last active run
run = mlflow.last_active_run()
# Set run_id variable
run_id = run.info.run_id
run_id
```

```
'8c2061731caf447e805a2ac65630e70c'
```

In the example, we set a variable `run_id` to `run.info.run_id`. This string value is then passed to the `load_model` function.

Load model from MLflow Tracking 03:17 - 03:35

Load model from MLflow Tracking

```
# Pass run_id as f-string literal
model = mlflow.sklearn.load_model(f"runs:/{run_id}/tracking_path")
# Show model
model
```

```
LogisticRegression()
```

The example code uses an f-string to pass the `run_id` and `tracking_path` to the `load_model` function. Finally, we print the model to confirm it was loaded successfully.

Custom models

Custom models 00:00 - 00:17

MLflow models and the Model API make building, logging, and loading models in many popular ML libraries straightforward. However, MLflow may not cover every possible use case.

Example use cases 00:17 - 00:51

Different machine learning applications have unique requirements. For instance, a natural language processing model may require tokenizers, a classification model might need a label encoder, or pre/post-processing tasks may be required around model inference. What if we need an ML library that MLflow doesn't support natively? MLflow offers a solution for these custom use cases.

Custom Python models 00:51 - 01:15

MLflow allows "Model Customization" using custom Python models through the python_function flavor. This flavor enables saving, logging, and loading models via the `mlflow.pyfunc` module, similar to other built-in flavors.

Custom model class 01:15 - 01:50

Custom model class

- Custom model class
  - `MyClass(mlflow.pyfunc.PythonModel)`
- PythonModel class
  - `load_context()` - loads artifacts when `mlflow.pyfunc.load_model()` is called
  - `predict()` - takes model input and performs user defined evaluation

To create a custom model, you start by creating a Python Class that inherits the `PythonModel` Class from MLflow. The `load_context` method loads artifacts when `load_model` is called, and the `predict` method handles model inference, allowing pre- and post-inference customization.

Python class 01:50 - 02:16

In Python, a Class is a blueprint for creating objects. These objects combine data and methods. In the example, an object is created and calls a method within the class to print "Hello!"

Example custom Class 02:16 - 02:35

Example custom Class

```python
import mlflow.pyfunc

# Define the model class
class CustomPredict(mlflow.pyfunc.PythonModel):
    # Load artifacts
    def load_context(self, context):
        self.model = mlflow.sklearn.load_model(context.artifacts["custom_model"])
    # Evaluate input using custom_function()
    def predict(self, context, model_input):
        prediction = self.model.predict(model_input)
        return custom_function(prediction)
```

The `CustomPredict` class defines the `load_context` method for loading artifacts and the `predict` method for inference, which can include user-defined functions like `custom_function`.

Saving and logging a custom model 02:35 - 02:51

To save the custom model, use `mlflow.pyfunc.save_model`. To log it to MLflow Tracking, use `mlflow.pyfunc.log_model`. Both functions use the custom model class.

Loading custom models 02:51 - 03:02

Loading custom models

```
# Load model from local filesystem
mlflow.pyfunc.load_model("local")
```

```
# Load model from MLflow Tracking
mlflow.pyfunc.load_model("runs:/run_id/tracking_path")
```

Loading a custom model follows the same process as built-in flavors. You call `load_model` from `mlflow.pyfunc` and specify the correct URI.

Model Evaluation 03:02 - 03:22

MLflow offers the `evaluate` function for model performance evaluation. This function supports both classification and regression models, helping to explain the model's behavior.

Evaluation Example 03:22 - 03:45

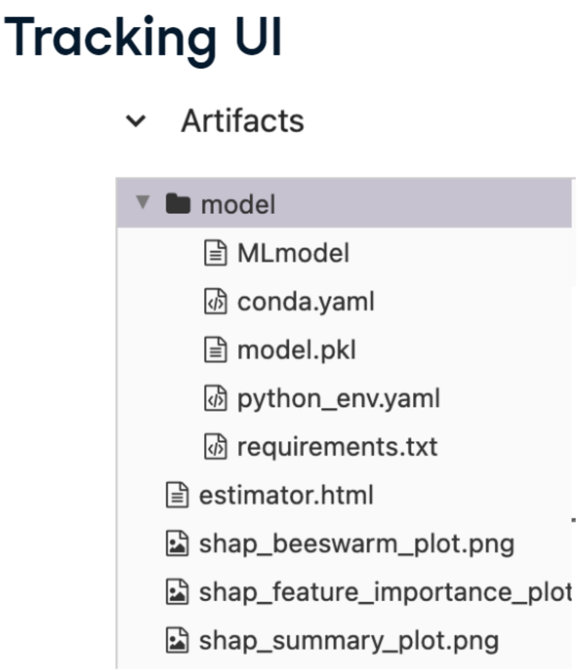
Evaluation Example

```
# Training Data
X_train, X_test, y_train, y_test = \
    train_test_split(X, y,
    train_size=0.7,random_state=0)

# Linear Regression model
lr = LinearRegression()
lr.fit(X_train, y_train)
```

```
# Dataset
eval_data = X_test
eval_data["test_label"] = y_test

# Evaluate model with Dataset
mlflow.evaluate(
    "runs:/run_id/model",
    eval_data,
    targets="test_label",
    model_type="regressor"
)
```

The `evaluate` function evaluates a linear regression model using a dataset called `eval_data` with the target `y_test`. The features and labels need to be in the same dataset.

Tracking UI 03:45 - 04:04



SHAP

[1] shap.readthedocs.io

In MLflow Tracking, SHAP plot files are automatically added as artifacts during evaluation. SHAP is a tool for explaining machine learning models, and these plots provide helpful visualizations during model evaluation.

Model serving

Model serving 00:00 - 00:11

Now that we have learned how to package and log models using MLflow Models, it's time to explore how to serve those models for deployment.

MLflow Models 00:11 - 00:28

MLflow Models help standardize model packaging, log models for tracking, and evaluate their performance, covering key steps like "Model Engineering" and "Model Evaluation" in the ML lifecycle.

Model Deployment 00:28 - 00:38

Another critical step is model deployment, and MLflow simplifies this process through standardized model packaging.

REST API 00:38 - 01:21

REST API

- `/ping` - for health checks
- `/health` - for health checks
- `/version` - for getting the version of MLflow
- `/invocations` - for model scoring
- Port 5000

MLflow serves models through a REST API with four endpoints: ping and health (for service status), version (for MLflow version), and invocations (for getting scores from the deployed model). The API runs on port 5000 by default and can be accessed via HTTP.

Invocations endpoint 01:21 - 01:45

Invocations endpoint

`/invocations`

```
No,Name,Subject
1,Bill Johnson,English
2,Gary Valentine,Mathematics
```

`Content-Type` : `application/json` or `application/csv`

```
{
    "1": {
        "No": "1",
        "Name": "Bill Johnson",
        "Subject": "English"
    },
    "2": {
        "No": "2",
        "Name": "Gary Valentine",
        "Subject": "Mathematics"
    }
}
```

The invocations endpoint accepts either CSV or JSON input formats. It also requires specifying the content-type header as either application/json or application/csv, depending on the format being used.

CSV and JSON format 01:45 - 02:07

For CSV input, a pandas DataFrame can be converted using the `to_csv` method. For JSON, input must be a dictionary with either dataframe_split or dataframe_records specified, indicating the type of data being passed.

DataFrame split 02:07 - 02:27

The dataframe_split orientation is recommended for JSON input to preserve column order. The example shows JSON input with dataframe_split defining columns and rows as lists.

Serving Models 02:27 - 02:38

MLflow's "Serve" command launches a local web server to run the REST API for serving models.

Serve URI 02:38 - 03:00

Serve uri

```
# Local Filesystem
mlflow models serve -m relative/path/to/local/model
```

```
# Run ID
mlflow models serve -m runs:/<mlflow_run_id>/artifacts/model
```

```
# AWS S3
mlflow models serve -m s3://my_bucket/path/to/model
```

The "Serve" command uses the `-m` option to specify the model's URI, which can be from the local filesystem, a run id with artifact path, or cloud storage like AWS S3.

Serve example 03:00 - 03:12

An example is using the "Serve" command to run a model that predicts if a person is a smoker, which starts a web server on port 5000.

Invocations Request 03:12 - 03:46

Invocations Request

```
# Send dataframe_split orientation payload to MLflow
curl http://127.0.0.1:5000/invocations -H 'Content-Type: application/json' -d '{
  "dataframe_split": {
      "columns": ["sex", "age", "weight"],
      "data": [["male", 23, 160], ["female", 33, 120]]
  }
}'
```

```
{"predictions": [1, 0]}
```

To get a model prediction, you can use curl to send a request to the invocations endpoint with the content-type header and JSON input. The response returns a list of predictions, such as 1 for smoker or 0 for non-smoker.

Introduction to MLflow Model Registry

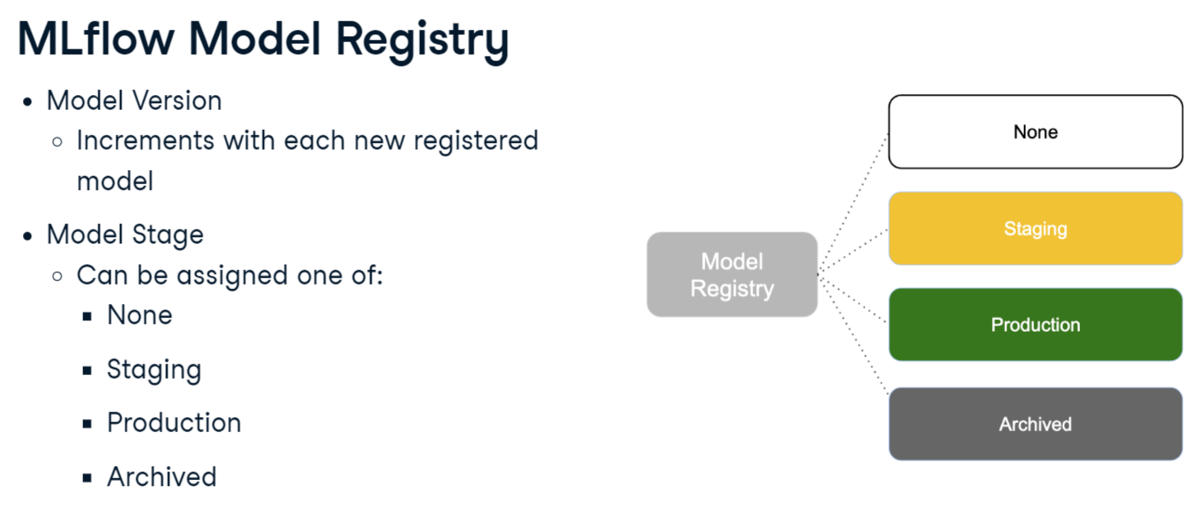
Introduction to MLflow Model Registry 00:00 - 00:07

So far, we've learned how to track MLflow Models using the Model API and MLflow's built-in Flavors.

Model lifecycle 00:07 - 00:25

While tracking and logging models are key for Model Engineering and Evaluation, we need to start managing the model lifecycle through various environments such as development, staging, and production as we move toward deployment.

MLflow Model Registry 00:25 - 00:39



The MLflow Model Registry provides collaboration through a UI and the MLflow Client module. It allows users to manage the model lifecycle, including versioning and staging.

MLflow Model Registry 00:39 - 01:08

There are four major concepts in the MLflow Model Registry: a Model, created and logged to MLflow Tracking, can be registered, gaining a version and a stage such as "Staging", "Production", or "Archived".

MLflow Model Registry 01:08 - 01:27

Each registered model gets a version, incrementing with every new registration. Models can be assigned to different stages like "Staging", "Production", or "Archived."

Working with the Model Registry 01:27 - 01:50

You can interact with the Model Registry programmatically using the MLflow client module or through the MLflow UI under the "Models" tab. The UI displays registered models, their versions, and assigned stages.

MLflow Client module 01:50 - 02:03

The MLflow client module allows for programmatic interaction with Experiments, Runs, Model Versions, and Registered Models. We will use this to work with the Model Registry.

Using MLflow client module 02:03 - 02:22

Using MLflow client module

```python
# Import from MLflow module
from mlflow import MlflowClient

# Create an instance
client = MlflowClient()

# Print the object
client
```

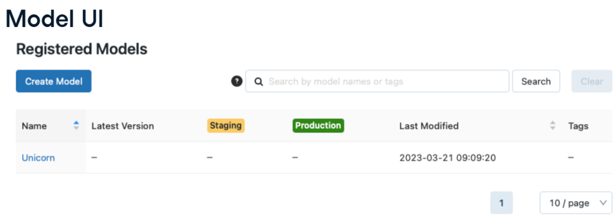
```
<mlflow.tracking.client.MlflowClient object at 0x101d55f30>
```

To use the client module, import the MlflowClient class from the MLflow module and create an instance of the class. This instance will allow interaction with the Model Registry.

Registering a model 02:22 - 02:39

We can create a new empty model in the Model Registry using the client's create_registered_model function by passing the desired model name. Registering existing models will be covered in the next video.

Model UI 02:39 - 02:47



The newly created model can now be viewed in the MLflow UI. We will use this model to register models into the Model Registry.

Searching registered models 02:47 - 03:01

As more models are registered, you can search for models in the MLflow Model Registry using the MLflow client module.

Searching registered models 03:01 - 03:28

Searching registered models

```
# Search for registered models
client.search_registered_models(filter_string=MY_FILTER_STRING)
```

- Identifiers
  - name - of the model
  - tags - tags associated with model
- Comparators
  - `=` - equal to
  - `!=` - not equal to
  - `LIKE` - case-sensitive pattern match
  - `ILIKE` - case-insensitive pattern match

The search_registered_models function allows you to filter models based on attributes and comparators. You can search for names, tags, and other criteria using comparators like equal to or pattern matching.

Example search 03:28 - 04:03

To search for models containing the word "Unicorn," we can use the LIKE comparator and include a `%` wildcard in the filter string. The search returns two registered models: Unicorn and Unicorn 2.0.

Registering Models

Registering Models 00:00 - 00:09

Now that we've created Model placeholders in the MLflow Model Registry, let's dive into registering our own MLflow Models.

Registering MLflow Models 00:09 - 00:43

Registering MLflow Models to the Registry provides benefits such as version control, which allows tracking changes and reverting if necessary. It also facilitates collaboration between teams like Data Scientists and Developers, and within teams, to improve or test models.

Model lifecycle management 00:43 - 00:51

Registering models to the MLflow Model Registry enhances lifecycle management for MLflow Models.

Ways to register models 00:51 - 01:27

MLflow offers two ways to register models:

1. Use the `register_model` function from the MLflow module for existing models, specifying the correct `model_uri`.
2. For new models, use the `log_model` function, passing the `registered_model_name` argument to register the model during logging.

Registering model example 01:27 - 01:47

The `register_model` function can register models from either the local filesystem or a Tracking server. The code registers models to the "Unicorn" Model in the Model Registry, one from the local filesystem and one from the Tracking Server.

Registering local model 01:47 - 02:04

When a model is registered, MLflow checks if it exists in the Registry and creates it if not. The version starts at 1 for new models.

Registering tracking model 02:04 - 02:21

For models already containing a version, MLflow increments the version by 1 with each new registration. In the example, registering a model from the Tracking Server created version 2 of the "Unicorn" Model.

Models UI 02:21 - 02:31

In the Model Registry UI, we can see the Unicorn Model with the latest version as 2.

Unicorn versions 02:31 - 02:40

Clicking on the Unicorn Model in the UI shows all registered models and their associated versions.

Logging model 02:40 - 03:11

Logging model

```python
# Import modules
import mlflow
import mlflow.sklearn
from sklearn.linear_model import LogisticRegression

# Model
lr = LogisticRegression()
lr.fit(X, y)

# Log model
mlflow.sklearn.log_model(lr, "model", registered_model_name="Unicorn")
```

```python
# Log model
mlflow.sklearn.log_model(lr, "model", registered_model_name="Unicorn")
```

```
Registered model 'Unicorn' already exists. Creating a new version of this model...
2023/03/24 17:31:10 INFO mlflow.tracking._model_registry.client:
Waiting up to 300 seconds for model version to finish creation.
Model name: Unicorn, version 3
Created version '3' of model 'Unicorn'.
<mlflow.models.model.ModelInfo object at 0x14734d330>
```

We can also register new models during training runs using the `log_model` function from model flavors. Passing the `registered_model_name` argument allows models to be registered to a specific model during logging to MLflow Tracking.

Logging model output 03:11 - 03:17

Registering a new model to Unicorn incremented the version to 3.

Model stages

Model stages 00:00 - 00:08

The Model Registry, combined with model versioning, provides a meaningful way to manage the lifecycle of models throughout the ML lifecycle.

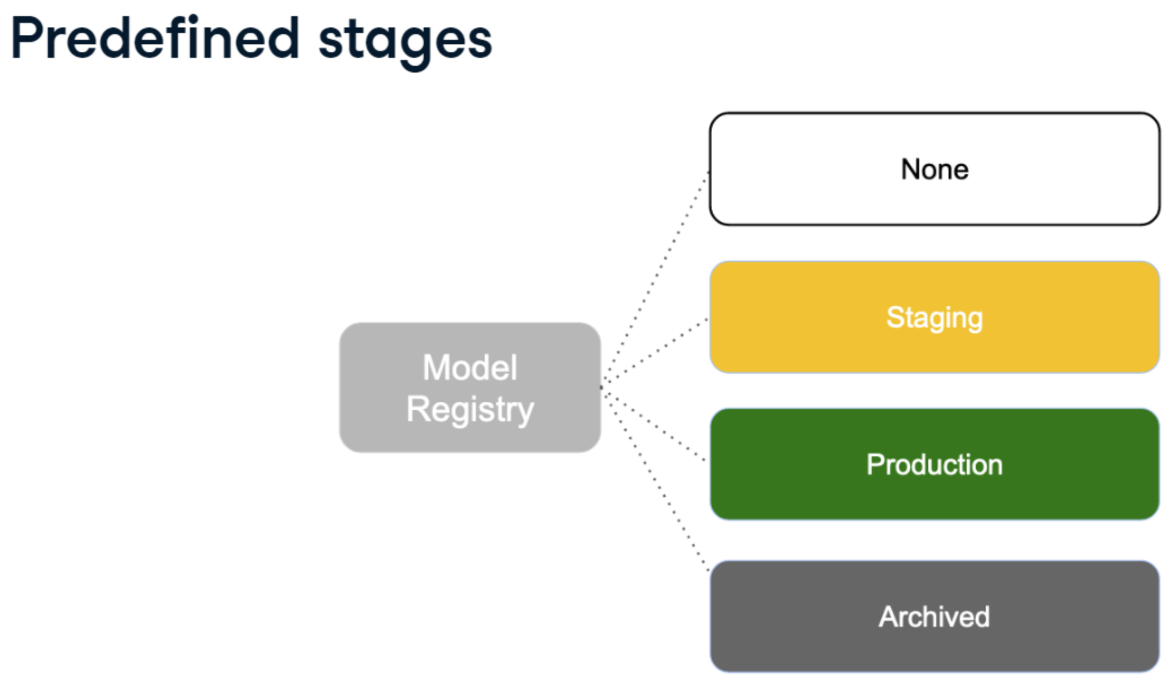
Software environments 00:08 - 00:20

In addition to versioning, the MLflow Model Registry helps manage model progression through different software environments.

MLflow model stages 00:20 - 00:46

MLflow provides predefined stages such as "None," "Staging," "Production," and "Archived." A model version can be assigned a stage, and models can transition from one stage to another. However, only one stage can be assigned at a time.

Predefined stages 00:46 - 01:00



Model stages represent phases in the model lifecycle, defined by individual organizations based on their software development process.

None 01:00 - 01:07

The default stage, "None," is assigned when a model hasn't received a stage yet.

Staging 01:07 - 01:12

"Staging" is assigned when a model is in the testing and evaluation phase.

Production 01:12 - 01:19

"Production" is for models that have passed tests and are ready for live environments.

Archived 01:19 - 01:24

"Archived" is assigned to models no longer in use.

Transitioning models 01:24 - 02:00

Model stages can be transitioned using either the Registry UI or by calling the `transition_model_version_stage` function from the MLflow client module. In the UI, you can transition a model via a dropdown selection for a specific version. Using the MLflow Client, you assign the model name, version number, and the new stage as arguments in the function.

Transition model version staging 02:00 - 02:11

Transition model version staging

```python
# Transition to Staging
client.transition_model_version_stage(name="Unicorn", version=3, stage="Staging")
```

```
<ModelVersion: creation_timestamp=1679693470034, current_stage='Staging',
description=None, last_updated_timestamp=1679699050734, name='Unicorn',
run_id='a1454f2865e449f8835f38f71e53e547', run_link=None,
source='./mlruns/1/a1454f2865e449f8835f38f71e53e547/artifacts/model',
status='READY', status_message=None, tags={}, user_id=None, version=3>
```

For example, transitioning the Unicorn model version 3 to Staging shows an output where the current stage is now "Staging."

Registry UI 02:11 - 02:17

This can be confirmed in the UI where version 3 is now in the Staging stage.

Transitioning to production 02:17 - 02:38

Once all tests are passed in Staging, you can transition the model to "Production" using the same function. The model is then ready for production environments.

Model deployment

ML lifecycle 00:11 - 00:19

The MLflow Model Registry plays an essential role in the "Model Deployment" phase by enabling streamlined model deployment.

Model versions and stages 00:19 - 00:37

The registry simplifies model deployment, providing a centralized repository for managing models across their lifecycle. Models can be deployed based on either their version or stage.

Ways to deploy models 00:37 - 00:55

Ways to deploy models

Load model

```
# MLflow flavor
mlflow.FLAVOR.load_model()
```

Serve model

```
# MLflow serve command-line
mlflow models serve
```

MLflow provides two options for deploying models: via the `load_model` function from the model flavor or using the `serve` command from the MLflow command line tool.

Models URI 00:55 - 01:09

Models URI

Convention

```
models:/
```

Model version

```
models:/model_name/version
```

Model stage

```
models:/model_name/stage
```

MLflow uses a model URI convention to specify model deployment. The full model URI consists of a model name and a model version or stage.

Load models 01:09 - 01:27

Load models

```
# Import flavor
import mlflow.FLAVOR

# Load version
mlflow.FLAVOR.load_model("models:/model_name/version")

# Load stage
mlflow.FLAVOR.load_model("models:/model_name/stage")
```

To load a model from the registry, import the model flavor and use the `load_model` function, passing in the model's URI, either by version or stage.

Load models example 01:27 - 01:53

Load models example

```
# Import flavor
import mlflow.sklearn

# Load Unicorn model in Staging
model = mlflow.sklearn.load_model("models:/Unicorn/Staging")
# Print model
model
```

```
LogisticRegression()
```

```
# Inference
model.predict(data)
```

In this example, we load the Unicorn model in the Staging stage using the scikit-learn flavor's `load_model` function. The URI contains the model name (Unicorn) and stage (Staging). The model is loaded successfully and can be used for inference.

Serving models 01:53 - 02:29

Serving models

```
# Serve Unicorn model in Production stage
mlflow models serve -m "models:/Unicorn/Production"
```

```
2023/03/26 15:07:00 INFO mlflow.models.flavor_backend_registry:
Selected backend for flavor 'python_function'
2023/03/26 15:07:00 INFO mlflow.pyfunc.backend: === Running command 'exec gunicorn
--timeout=60 -b 127.0.0.1:5000 -w 1 ${GUNICORN_CMD_ARGS} --
mlflow.pyfunc.scoring_server.wsgi:app'
[2023-03-26 15:07:00 -0400] [86409] [INFO] Starting gunicorn 20.1.0
[2023-03-26 15:07:00 -0400] [86409] [INFO] Listening at: http://127.0.0.1:5000
[2023-03-26 15:07:00 -0400] [86409] [INFO] Using worker: sync
[2023-03-26 15:07:00 -0400] [86410] [INFO] Booting worker with pid: 86410
```

Models can also be served using the `serve` command from the MLflow command line. Serving refers to setting up an endpoint that receives input data and returns predictions. In this example, the Production stage Unicorn model is deployed using the `serve` command, and input data is sent to the invocations endpoint.

Invocations endpoint 02:29 - 02:49

When serving a model via the `serve` command, MLflow provides an API service with the invocations endpoint, which expects either CSV or JSON input on port 5000.

CSV and JSON 02:49 - 03:12

CSV format

```
pandas_df.to_csv()
```

JSON format

```
{
  "dataframe_split": {
      "columns": ["R&D Spend", "Administration", "Marketing Spend", "State"],
      "data": [["165349.20", 136897.80, 471784.10, 1]]
  }
}
```

CSV inputs must be valid pandas DataFrames, while JSON inputs should be dictionaries containing either `dataframe_split` or `dataframe_records` fields to define the input format.

Model prediction 03:12 - 03:26

Model prediction

```
# Send payload to invocations endpoint
curl http://127.0.0.1:5000/invocations -H 'Content-Type: application/json' -d
{
  "dataframe_split": {
      "columns": ["R&D Spend", "Administration", "Marketing Spend", "State"],
      "data": [["165349.20", 136897.80, 471784.10, 1]]
  }
}
```

```
[[104055.1842384]]
```

Whether using the `load_model` function or the `serve` command, inference returns the model's predictions. Here, the Unicorn model provides predicted profit from test data.

Introduction to MLflow Projects

Introduction to MLflow Projects 00:00 - 00:08

So far, we have explored various components of MLflow designed to address challenges in the machine learning lifecycle.

MLflow Projects 00:08 - 00:48

In this chapter, we will dive into MLflow Projects, which simplifies the ML lifecycle by organizing and running ML code in a reproducible way. Projects package code into reusable units, facilitating collaboration and making it easy to run code across different environments, including local machines and the cloud. This leads to improved productivity and efficiency.

MLproject 00:48 - 01:04

MLproject

```
project/
    MLproject
    train_model.py
    python_env.yaml
    requirements.txt
```

At its core, an MLflow Project is a directory containing ML code, which can be stored locally or in repositories like GitHub. The project is described by an MLproject file.

MLproject file 01:04 - 01:46

The MLproject file is a YAML configuration file that outlines the project properties, including the project name, entry points (commands to run different tasks), and environment dependencies. Entry points can execute Python or shell scripts, and the environment section ensures the code runs consistently across different platforms.

MLproject example 01:46 - 02:12

In this example, the MLproject file defines a project called "salary_model" with an entry point named *main*, which runs the command `python train_model.py`. The Python environment is specified in the `python_env.yaml` file to ensure reproducibility.

train_model.py 02:12 - 02:46

train_model.py

```python
# Set Auto logging for Scikit-learn flavor
mlflow.sklearn.autolog()

# Train the model
lr = LinearRegression()
lr.fit(X_train, y_train)
```

The `train_model.py` script contains code to train a linear regression model using salary data. Metrics and parameters are automatically logged to MLflow Tracking using the autolog function from the scikit-learn flavor. Once the model is trained, autolog captures key metrics.

python_env.yaml 02:46 - 03:15

python_env.yaml

```
python: 3.10.8
build_dependencies:
- pip
- setuptools
- wheel
dependencies:
- -r requirements.txt
```

The `python_env.yaml` file specifies the Python environment version (3.10.8) and lists the necessary dependencies such as pip and setuptools. It ensures the project runs the same way across different machines.

requirements.txt 03:15 - 03:30

requirements.txt

```
mlflow
scikit-learn
```

The `requirements.txt` file lists the required libraries, including MLflow and scikit-learn. These libraries are installed automatically when running the project, enabling interaction with MLflow and training the model.

Running MLflow Projects

Running MLflow Projects 00:00 - 00:07

Now that we understand the structure for creating an MLflow Project, let's learn how to execute it.

API and command line 00:07 - 00:23

MLflow Projects can be executed programmatically using an API from MLflow or via the command line, offering flexibility for automation and the ability to chain multiple projects together into workflows.

Projects API 00:23 - 01:04

Projects API

```
mlflow.projects
```

`mlflow.projects.run()`

- `uri` - URI to MLproject file
- `entry_point` - Entry point to start run from MLproject
- `experiment_name` - Experiment to track training run
- `env_manager` - Python environment manager: `local` or `virtualenv`

```python
# Run MLflow Project
mlflow.projects.run(
    uri='./',
    entry_point='main',
    experiment_name='My Experiment',
    env_manager='virtualenv'
)
```

MLflow provides the *mlflow.projects* module, which contains a *run* method for executing local or Git-stored projects. Key arguments include: `uri` to specify the project location, `entry_point` to point to the entry point in the MLproject, `experiment_name` to log the run under a specific experiment, and `env_manager` to define the Python environment.

MLproject 01:04 - 01:23

MLproject

```
name: salary_model
python_env: python_env.yaml
entry_points:
  main:
    command: "python train_model.py"
```

Before running our project, we review the local MLproject file, which defines the entry point called *main*, executing the *train_model.py* script. It also includes a *python_env* file to define the environment.

train_model.py 01:23 - 01:41

train_model.py

```python
# Import libraries and modules
import mlflow
import mlflow.sklearn
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

# Training Data
df = pd.read_csv('Salary_predict.csv')
X = df[["experience", "age", "interview_score"]]
y = df[["Salary"]]
```

The *train_model.py* script trains a linear regression model predicting salary based on experience, age, and interview score. The required libraries are imported, and the data is loaded from a *Salary_predict.csv* file.

train_model.py 01:41 - 01:53

We split the dataset into training and test sets. The *autolog* method from the scikit-learn flavor is used to automatically log metrics and parameters to MLflow Tracking during training.

Projects run 01:53 - 02:22

Projects run

```python
# Import MLflow Module
import mlflow

# Run local Project
mlflow.projects.run(uri='./', entry_point='main',
                    experiment_name='Salary Model')
```

To execute the project using *mlflow.projects.run*, we set the `uri` argument to the current directory (`./`), the `entry_point` as *main*, and the `experiment_name` as *Salary Model* to log the run.

Run output 02:22 - 02:37

Run output

```python
# Run local Project
mlflow.projects.run(uri='./', entry_point='main',
                    experiment_name='Salary Model')
```

```
2023/04/02 14:33:23 INFO mlflow.projects: 'Salary Model' does not exist.
Creating a new experiment
2023/04/02 14:33:23 INFO mlflow.utils.virtualenv: Installing python 3.10.8 if it
does not exist
2023/04/02 14:33:23 INFO mlflow.utils.virtualenv: Creating a new environment
/.mlflow/envs/mlflow-44f5094bba686a8d4a5c772
created virtual environment CPython3.10.8.final.0-64 in 236ms
2023/04/02 14:33:23 INFO mlflow.utils.virtualenv: Installing dependencies
```

When executed, MLflow creates a new experiment (if it doesn't exist), sets up the Python environment, and installs dependencies as defined in the *python_env.yaml* file.

Run output 02:37 - 02:50

After dependencies are installed, the *train_model.py* script is executed, and once completed, a success message is displayed, indicating the model was trained successfully.

MLflow Tracking 02:50 - 02:56

The run and the model are successfully logged in MLflow Tracking, as confirmed by the Tracking UI.

Command line 02:56 - 03:07

Command line

```
mlflow run
```

- `--entry-point` - Entry point to start run from MLproject
- `--experiment-name` - Experiment to track training run
- `--env-manager` - Python environment manager: `local` or `virtualenv`
- `URI` - URI to MLproject file

MLflow Projects can also be executed via the command line using the *mlflow run* command. It supports the same options as the *mlflow.projects* module.

Run command 03:07 - 03:33

Run command

```
# Run main entry point from Salary Model experiment
mlflow run --entry-point main --experiment-name "Salary Model" ./
```

```
2023/04/02 15:23:34 INFO mlflow.utils.virtualenv: Installing python 3.10.8 if it
does not exist
2023/04/02 15:23:34 INFO mlflow.utils.virtualenv: Environment
/.mlflow/envs/mlflow-44f5094bba686a8d4a5c772 already exists
2023/04/02 15:23:34 INFO mlflow.projects.backend.local: === Running command 'source
/Users/weston/.mlflow/envs/mlflow-44f5094bba686a8d4a5c772/bin/activate && python
train_model.py' in run with ID 'da5b37b6f53245e5bca59ba8ed6d7dc1' ===
```

We use the `mlflow run` command with `--entry-point main` to specify the entry point and `--experiment-name Salary Model` for the experiment name. It runs the project, reuses the previous Python environment, and completes the training, showing a success message.

Specifying parameters

Specifying parameters 00:00 - 00:06

So far, we have learned how MLflow Projects can be used to create reproducible code for the ML Lifecycle.

Parameters 00:06 - 00:26

Parameters block

```
name: project_name
python_env: python_env.yaml
entry_points:
  main:
    parameters:
      parameter_1:
        type: data_type
        default: default_value
      parameter_2:
        type: data_type
        default: default_value
    command: "python train.py {parameter_1_name} {parameter_2_name}"
```

MLflow Projects allows flexibility and customization through the use of parameters. Parameters are variables that can be specified by the user when running an MLflow Project, simplifying the process of exploring different configurations, such as hyperparameters during model training.

Specifying parameters 00:26 - 00:38

Parameters are declared within the *MLproject* file and are given a specified name. Each parameter can have a data type and a default value.

Specifying parameters 00:38 - 00:52

Data types can be Python data types such as *float* or *string*, with *string* as the default if none is specified. A default value is used when a parameter value is not provided during the project run.

Parameters block 00:52 - 01:03

A block of parameters is placed within an *entry_point* in the *MLproject* file, and these parameters are passed to the command within the entry point as arguments.

train_model.py 01:03 - 01:15

train_model.py

```
# Import libraries and modules
import mlflow
import mlflow.sklearn
import pandas as pd
import sys
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

# Training Data
df = pd.read_csv('Salary_predict.csv')
X = df[["experience", "age", "interview_score"]]
y = df[["Salary"]]
```

Let's start by modifying the *train_model.py* code used to train our salary model. We import the *sys* module and other necessary libraries.

train_model.py 01:15 - 01:35

train_model.py

```
# Train test split
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.7,
                                                    random_state=0)

# Set Auto logging for Scikit-learn flavor
mlflow.sklearn.autolog()
# Parameters
n_jobs_param = int(sys.argv[1])
fit_intercept_param = bool(sys.argv[2])
# Train model
lr = LinearRegression(n_jobs=n_jobs_param, fit_intercept=fit_intercept_param)
lr.fit(X_train, y_train)
```

We add variables *n_jobs_param* and *fit_intercept_param*, which are set to the first and second arguments passed to the script using *sys.argv*. These variables are used as hyperparameters when training the model.

MLproject 01:35 - 01:52

MLproject

```
name: salary_model
python_env: python_env.yaml
entry_points:
  main:
    parameters:
      n_jobs_param:
        type: int
        default: 1
      fit_intercept_param:
        type: bool
        default: True
    command: "python train_model.py {n_jobs_param} {fit_intercept_param}"
```

In the *MLproject* file, we add a new parameters block, specifying *n_jobs_param* and *fit_intercept_param* with data types and default values. The command is updated to pass these parameters to the *train_model.py* script.

Running parameters 01:52 - 02:26

Running parameters

Python

```
mlflow.projects.run()
```

```
parameters={
        'parameter_1': data_value,
        'parameter_2': data_value,
    }
```

CLI

```
mlflow run
```

```
-P parameter_1=parameter_1_value
```

```
-P parameter_2=parameter_2_value
```

To run the parameterized ML code, you can use the *mlflow.projects* module with the *run* method or the *mlflow run* command. The *run* method takes a "parameters" argument, which is a dictionary containing the parameters. The *mlflow run* command uses the *-P* argument for each parameter in the form of *parameter-name=parameter-value*.

Projects run 02:26 - 02:36

Projects run

```python
# Import MLflow Module
import mlflow

# Run local Project
mlflow.projects.run(
    uri='./', entry_point='main',
    experiment_name='Salary Model',
    parameters={
        'n_jobs_param': 2,
        'fit_intercept_param': False
})
```

To run the project using the *mlflow.projects* module, we include the *parameters* argument in the *run* method, passing the dictionary of parameters we want to use in the training script.

Output 02:36 - 02:50

When the project is executed, the parameters are passed to the command as defined in the dictionary. A new run is started, and when it finishes, MLflow confirms the run succeeded.

Run command 02:50 - 03:14

Run command

```
# Run main entry point from Salary Model experiment
mlflow run --entry-point main --experiment-name "Salary Model" \
    -P n_jobs_param=3 -P fit_intercept_param=True ./
```

We can also run the project with the *mlflow run* command. Here, we use two *-P* arguments to specify the parameters for *n_jobs_param* (set to 3) and *fit_intercept_param* (set to True).

Output 03:14 - 03:21

In this run, the training code is executed with the arguments *3* and *True* for *n_jobs_param* and *fit_intercept_param*, respectively.

Workflows

Workflows 00:00 - 00:08

So far, we have learned how MLflow Projects package ML code in a way that allows it to be easily reproduced and run across different environments.

MLflow Projects 00:08 - 00:31

We've also explored how parameters in MLflow Projects improve the flexibility and usability of experiments. Another feature is running multi-step workflows, where each step automatically triggers the next once it finishes.

MLproject 00:31 - 00:47

MLproject

```
name: project_name
python_env: python_env.yaml
entry_points:
  step_1:
    command: "python train_model.py"
  step_2:
    command: "python evaluate_model.py {run_id}"
    parameters:
        run_id:
            type: str
            default: None
```

An *MLproject* file can contain multiple entry points, making it possible to call each as a step in a single Python program. For instance, step 2 might expect a *run_id* as input.

Workflows 00:47 - 01:02

Workflows

```python
import mlflow

# Step 1
step_1 = mlflow.projects.run(
    uri='./',
    entry_point='step_1'
)
# Step 2
step_2 = mlflow.projects.run(
    uri='./',
    entry_point='step_2'
)
```

To execute each entry point in a single program, use the *run* method from the MLflow Projects module. For each step, you assign *mlflow.projects.run* as a variable and specify which entry point to run.

Projects run 01:02 - 01:23

Projects run

```python
import mlflow

# Step 1
step_1 = mlflow.projects.run(
    uri='./',
    entry_point='step_1'
)

print(step_1)
```

```
<mlflow.projects.submitted_run.LocalSubmittedRun object at 0x125eac8b0>
```

Each call to the *run* method returns a *run* object, whose attributes can be passed to other workflow steps. This allows for complex workflows where outputs from one step can be inputs for the next.

Projects run 01:23 - 01:56

Projects run

`step_1.cancel()` - Terminate a current run

`step_1.get_status()` - Get the status of a run

`step_1.run_id` - `run_id` of the run

`step_1.wait()` - Wait for the run to finish

The *run* object returns attributes such as:

- *cancel*, to terminate a current run.
- *get_status*, to check the run's status.
- *run_id*, helpful for tracking artifacts like models.
- *wait*, which waits for the run to complete and returns *True* if successful, *False* otherwise.

Projects run 01:56 - 02:12

Projects run

```
import mlflow

# Step 1
step_1 = mlflow.projects.run(
    uri='./',
    entry_point='step_1'
)

# Set variable for step_1 run_id
step_1_run_id = step_1.run_id
```

```
# Step 2
step_2 = mlflow.projects.run(
    uri='./',
    entry_point='step_2',
    parameters={
        'run_id': step_1_run_id
    }
)
```

In the workflow's Python program, you can set a variable, like *step_1_run_id*, equal to *step_1.run_id*. This will pass the *run_id* as an input parameter for *step_2*.

ML Lifecycle 02:12 - 02:24

Due to its flexibility, MLflow Projects are ideal for managing various steps in the ML lifecycle. For example, a multi-step workflow could handle both Model Engineering and Model Evaluation.