Quickstart

Installing MLflow

You install MLflow by running:

Python R

```
# Install MLflow
pip install mlflow

# Install MLflow with the experimental MLflow Pipelines component
pip install mlflow[pipelines]  # for pip
conda install -c conda-forge mlflow-pipelines  # for conda

# Install MLflow with extra ML libraries and 3rd-party tools
pip install mlflow[extras]

# Install a lightweight version of MLflow
pip install mlflow-skinny
```

Note

MLflow works on MacOS. If you run into issues with the default system Python on MacOS, try installing Python 3 through the Homebrew package manager using `brew install python`. (In this case, installing MLflow is now `pip3 install mlflow`).

Note

To use certain MLflow modules and functionality (ML model persistence/inference, artifact storage options, etc), you may need to install extra libraries. For example, the `mlflow.tensorflow` module requires TensorFlow to be installed. See

https://github.com/mlflow/mlflow/blob/master/EXTRA_DEPENDENCIES.rst for more details.

Note

When using MLflow skinny, you may need to install additional dependencies if you wish to use certain MLflow modules and functionalities. For example, usage of SQL-based storage for MLflow Tracking (e.g. `mlflow.set_tracking_uri("sqlite:///my.db")`) requires `pip install mlflow-skinny sqlalchemy alembic sqlparse`. If using MLflow skinny for serving, a minimally functional installation would require `pip install mlflow-skinny flask`.

At this point we recommend you follow the tutorial for a walk-through on how you can leverage MLflow in your daily workflow.

Downloading the Quickstart

Download the quickstart code by cloning MLflow via `git clone https://github.com/mlflow/mlflow`, and cd into the `examples` subdirectory of the repository. We'll use this working directory for running the `quickstart`.

We avoid running directly from our clone of MLflow as doing so would cause the tutorial to use MLflow from source, rather than your PyPi installation of MLflow.

Using the Tracking API

The MLflow Tracking API lets you log metrics and artifacts (files) from your data science code and see a history of your runs. You can try it out by writing a simple Python script as follows (this example is also included in `quickstart/mlflow_tracking.py`):

Python R

```python
import os
from random import random, randint
from mlflow import log_metric, log_param, log_artifacts

if __name__ == "__main__":
    # Log a parameter (key-value pair)
    log_param("param1", randint(0, 100))

    # Log a metric; metrics can be updated throughout the run
    log_metric("foo", random())
    log_metric("foo", random() + 1)
    log_metric("foo", random() + 2)

    # Log an artifact (output file)
    if not os.path.exists("outputs"):
        os.makedirs("outputs")
    with open("outputs/test.txt", "w") as f:
        f.write("hello world!")
    log_artifacts("outputs")
```

Viewing the Tracking UI

By default, wherever you run your program, the tracking API writes data into files into a local `./mlruns` directory. You can then run MLflow's Tracking UI:

Python R

```
mlflow ui
```

and view it at http://localhost:5000.

Note

If you see message `[CRITICAL] WORKER TIMEOUT` in the MLflow UI or error logs, try using `http://localhost:5000` instead of `http://127.0.0.1:5000`.

Running MLflow Projects

MLflow allows you to package code and its dependencies as a *project* that can be run in a reproducible fashion on other data. Each project includes its code and a `MLproject` file that defines its dependencies (for example, Python environment) as well as what commands can be run into the project and what arguments they take.

You can easily run existing projects with the `mlflow run` command, which runs a project from either a local directory or a GitHub URI:

```
mlflow run sklearn_elasticnet_wine -P alpha=0.5

mlflow run https://github.com/mlflow/mlflow-example.git -P alpha=5.0
```

There's a sample project in `tutorial`, including a `MLproject` file that specifies its dependencies. if you haven't configured a tracking server, projects log their Tracking API data in the local `mlruns` directory so you can see these runs using `mlflow ui`.

> **Note**
>
> By default `mlflow run` installs all dependencies using conda. To run a project without using `conda`, you can provide the `--no-conda` option to `mlflow run`. In this case, you must ensure that the necessary dependencies are already installed in your Python environment.

For more information, see MLflow Projects.

Saving and Serving Models

MLflow includes a generic `MLmodel` format for saving *models* from a variety of tools in diverse *flavors*. For example, many models can be served as Python functions, so an `MLmodel` file can declare how each model should be interpreted as a Python function in order to let various tools serve it. MLflow also includes tools for running such models locally and exporting them to Docker containers or commercial serving platforms.

To illustrate this functionality, the `mlflow.sklearn` package can log scikit-learn models as MLflow artifacts and then load them again for serving. There is an example training application in `sklearn_logistic_regression/train.py` that you can run as follows:

```
python sklearn_logistic_regression/train.py
```

When you run the example, it outputs an MLflow run ID for that experiment. If you look at `mlflow ui`, you will also see that the run saved a `model` folder containing an `MLmodel` description file and a pickled scikit-learn model. You can pass the run ID and the path of the model within the artifacts directory (here "model") to various tools. For example, MLflow includes a simple REST server for python-based models:

```
mlflow models serve -m runs:/<RUN_ID>/model
```

> **Note**
>
> By default the server runs on port 5000. If that port is already in use, use the *–port* option to specify a different port. For example:
> `mlflow models serve -m runs:/<RUN_ID>/model --port 1234`

Once you have started the server, you can pass it some sample data and see the predictions.

The following example uses `curl` to send a JSON-serialized pandas DataFrame with the `split` orientation to the model server. For more information about the input data formats accepted by the pyfunc model server, see the MLflow deployment tools documentation.

```
curl -d '{"columns":["x"], "data":[[1], [-1]]}' -H 'Content-Type: application/json;
format=pandas-split' -X POST localhost:5000/invocations
```

which returns:

```
[1, 0]
```

For more information, see MLflow Models.

Logging to a Remote Tracking Server

In the examples above, MLflow logs data to the local filesystem of the machine it's running on. To manage results centrally or share them across a team, you can configure MLflow to log to a remote tracking server. To get access to a remote tracking server:

Launch a Tracking Server on a Remote Machine

Launch a tracking server on a remote machine.

You can then log to the remote tracking server by setting the `MLFLOW_TRACKING_URI` environment variable to your server's URI, or by adding the following to the start of your program:

Python R

```
import mlflow
mlflow.set_tracking_uri("http://YOUR-SERVER:4040")
mlflow.set_experiment("my-experiment")
```

Log to Databricks Community Edition

Alternatively, sign up for Databricks Community Edition, a free service that includes a hosted tracking server. Note that Community Edition is intended for quick experimentation rather than production use cases. After signing up, run `databricks configure` to create a credentials file for MLflow, specifying https://community.cloud.databricks.com as the host.

To log to the Community Edition server, set the `MLFLOW_TRACKING_URI` environment variable to "databricks", or add the following to the start of your program:

Python R

```python
import mlflow
mlflow.set_tracking_uri("databricks")
# Note: on Databricks, the experiment name passed to set_experiment must be a valid path
# in the workspace, like '/Users/<your-username>/my-experiment'. See
# https://docs.databricks.com/user-guide/workspace.html for more info.
mlflow.set_experiment("/my-experiment")
```