kedro-mlflow

Release 2.0.3

Yolan Honoré-Rougé

May 10, 2026

CONTENTS

1 Resources 3

Python Module Index 89

Index 91

`kedro-mlflow` is a Kedro plugin to integrate MLflow effortlessly inside Kedro projects.

Its main features are **automatic parameters tracking**, **datasets tracking as artifacts**, Kedro **pipelines packaging** and serving and **automatic synchronisation between training and inference** pipelines. It aims at providing a complete yet modular framework for high reproducibility of machine learning experiments and ease of deployment.

Experiment tracking

Track the **parameters**, **metrics**, **artifacts** and **models** of your kedro pipelines for reproducibility.

source/03_experiment_tracking/01_experiment_tracking/01_configuration.html Pipeline as model

Package any kedro pipeline to a **custom mlflow model** for deployment and serving. The custom model for an inference pipeline can be **registered** in mlflow **automatically** at the end of each training in a *scikit-learn* like way.

source/04_pipeline_as_model/01_pipeline_as_custom_model/01_mlflow_models.html

CHAPTER

ONE

RESOURCES

Quickstart

Get started in **1 mn** with experiment tracking!

Try out

source/02_getting_started/01_installation/01_installation.html Advanced tutorial

The `kedro-mlflow-tutorial` github repo contains a step-by-step tutorial to learn how to use kedro-mlflow as a MLOps framework!

Try on github

https://github.com/Galileo-Galilei/kedro-mlflow-tutorial Demonstration in video

A youtube video by the kedro team to introduce the plugin, with live coding.

Watch on youtube

https://www.youtube.com/watch?v=Az_6UKqbznw Tackling the ML Reproducibility Curse

A community video by Oleg Litvinov showcasing how to use the Kedro-MLflow plugin on an end to end project.

Watch on YouTube

https://youtu.be/mIfJR3CdBUE

1.1 Introduction

1.1.1 Introduction to `kedro` and `mlflow`

What is `Kedro`?

`Kedro` is a python package which facilitates the prototyping of data pipelines. It aims at enforcing software engineering best practices (separation between I/O and compute, abstraction, templating...). It is specifically useful for machine learning projects since it provides within the same interface interactive objects for the exploration phase, and *Command Line Interface* (CLI) and configuration files for the production phase. This makes the transition from exploration to production as smooth as possible.

For more details, see Kedro's official documentation.

What is `Mlflow`?

`Mlflow` is a library which manages the lifecycle of machine learning models. Mlflow provides 4 modules:

- `Mlflow Tracking`: This modules focuses on experiment versioning. Its goal is to store all the objects needed to reproduce any code execution. This includes code through version control, but also parameters and artifacts (i.e objects fitted on data like encoders, binarizers...). These elements vary wildly during machine learning

experimentation phase. `Mlflow` also enable to track metrics to evaluate runs, and provides a *User Interface* (UI) to browse the different runs and compare them.

- `Mlflow Projects`: This module provides a configuration files and CLI to enable reproducible execution of pipelines in production phase.
- `Mlflow Models`: This module defines a standard way for packaging machine learning models, and provides built-in ways to serve registered models. Such standardization enable to serve these models across a wide range of tools.
- `Mlflow Model Registry`: This modules aims at monitoring deployed models. The registry manages the transition between different versions of the same model (when the dataset is retrained on new data, or when parameters are updated) while it is in production.

For more details, see Mlflow's official documentation.

A brief comparison between `Kedro` and `Mlflow`

While `Kedro` and `Mlflow` do not compete in the same field, they provide some overlapping functionalities. `Mlflow` is specifically dedicated to machine learning and its lifecycle management, while `Kedro` focusing on data pipeline development. Below chart compare the different functionalities:

We discuss hereafter how the two libraries compete on the different functionalities and eventually complete each others.

Configuration and prototyping: Kedro 1 - 0 Mlflow

`Mlflow` and `Kedro` are essentially overlapping on the way they offer a dedicated configuration files for running the pipeline from CLI. However:

- `Mlflow` provides a single configuration file (the `MLProject`) where all elements are declared (data, parameters and pipelines). Its goal is mainly to enable CLI execution of the project, but it is not very flexible. This file is **production oriented** and is not really intended to use for and development.
- `Kedro` offers a bunch of files (`catalog.yml`, `parameters.yml`, `pipeline.py`) and their associated abstraction (`AbstractDataset`, `DataCatalog`, `Pipeline` and `node` objects). `Kedro` is much more opinionated: each object has a dedicated place (and only one!) in the template. This makes the framework both **exploration and production oriented**. The downside is that it could make the learning curve a bit sharper since a newcomer has to learn all `Kedro` specifications. It also provides a `kedro-viz` plugin to visualize the DAG interactively, which is particularly handy in medium-to-big projects.

Note

`Kedro` is a clear winner here, since it provides more functionnalities than `Mlflow`. It handles very well *by design* the exploration phase of data science projects when Mlflow is less flexible.

Versioning: Kedro 1 - 1 Mlflow

Kedro ahas made a bunch of attempts in the world of experiment tracking, with the `Journal` in early days (`kedro<=0.18`), then with an experiment tracking functionality which kept track of the parameters but which will be removed in `kedro>=0.20` due to the lack of traction (https://github.com/kedro-org/kedro-viz/issues/2202).

On the other hand, `Mlflow`:

- distinguishes between artifacts (i.e. any data file), metrics (integers that may evolve over time) and parameters. The logging is very straightforward since there is a one-liner function for logging the desired type. This separation makes further manipulation easier.

- offers a way to configure the logging in a database through the `mlflow_tracking_uri` parameter. This database-like logging comes with easy querying of different runs through a client (for instance "find the most recent run with a metric at least above a given threshold" is immediate with `Mlflow` but hacky in `Kedro`).
- comes with a *User Interface* (UI) which enable to browse / filter / sort the runs, display graphs of the metrics, render plots… This make the run management much easier than in `Kedro`.
- has a command to reproduce exactly the run from a given `git sha`, which is not possible in `Kedro`.

Note

`Mlflow` is a clear winner here, because *UI* and *run querying* are must-have for machine learning projects. It is more mature than `Kedro` for versioning and more focused on machine learning.

Model packaging and service: Kedro 1 - 2 Mlflow

`Kedro` offers a way to package the code to make the pipelines callable, but does not manage specifically machine learning models.

`Mlflow` offers a way to store machine learning models with a given "flavor", which is the minimal amount of information necessary to use the model for prediction:

- a configuration file
- all the artifacts, i.e. the necessary data for the model to run (including encoder, binarizer…)
- a loader
- a conda configuration through an `environment.yml` file

When a stored model meets these requirements, `Mlflow` provides built-in tools to serve the model (as an API or for batch prediction) on many machine learning tools (Microsoft Azure ML, Amazon Sagemaker, Apache SparkUDF) and locally.

Note

`Mlflow` is currently the only tool which adresses model serving. Some plugins address model deployment and serving in the Kedro ecosystem, but they are not as well maintained as the core framework.

Conclusion: Use Kedro and add Mlflow for machine learning projects

`Kedro`'s will to enforce software engineering best practice makes it really useful for machine learning teams. It is extremely well documented and the support is excellent, which makes it very user friendly even for people with no computer science background. However, it lacks some machine learning-specific functionalities (better versioning, model service), and it is where `Mlflow` fills the gap.

1.1.2 Motivation behind the plugin

When should I use kedro-mlflow?

Basically, you should use `kedro-mlflow` in **any `Kedro` project which involves machine learning** / deep learning. As stated in the introduction, `Kedro`'s current versioning (as of version `0.19.10`) is not sufficient for machine learning projects: it lacks a UI and a `run` management system. Besides, the `KedroPipelineModel` ability to serve a kedro pipeline as an API or a batch in one line of code is a great addition for collaboration and transition to production.

If you do not use `Kedro` or if you do pure data processing which does not involve *machine learning*, this plugin is not what you are seeking for ;-)

Why should I use kedro-mlflow?

Benchmark of existing solutions

This paragraph gives a (quick) overview of existing solutions for mlflow integration inside Kedro projects.

`Mlflow` is very simple to add to any existing code. It is a 2-step process:

- add `log_{XXX}` (either param, artifact, metric or model) functions where they are needed inside the code
- add a `MLProject` at the root of the project to enable CLI execution. This file must contain all the possible execution steps (like the `pipeline.py` / `hooks.py` in a kedro project).

Including mlflow inside a `kedro project` is consequently very easy: the logging functions can be added in the code, and the `MLProject` is very simple and is composed almost only of the `kedro run` command. You can find examples of such implementations:

- the medium paper by QuantumBlack employees.
- the associated github repo
- other examples can be found on Github, but AFAIK all of them follow the very same principles.

Enforcing Kedro principles

Above implementations have the advantage of being very straightforward and *mlflow compliant*, but they break several `Kedro` principles:

- the `MLFLOW_TRACKING_URI` which registers the database where runs are logged is declared inside the code instead of a configuration file, which **hinders portability across environments** and makes transition to production more difficult
- the logging of different elements can be put in many places in the `Kedro` template (in the code of any function involved in a `node`, in a `Hook`, in the `ProjectContext`, in a `transformer`...). This is not compliant with the `Kedro` template where any object has a dedicated location. We want to avoid the logging to occur anywhere because:
  - it is **very error-prone** (one can forget to log one parameter)
  - it is **hard to modify** (if you want to remove / add / modify an mlflow action you have to find it in the code)
  - it **prevents reuse** (re-usable function must not contain mlflow specific code unrelated to their functional specificities, only their execution must be tracked).

`kedro-mlflow` enforces these best practices while implementing a clear interface for each mlflow action in Kedro template. Below chart maps the mlflow action to perform with the Python API provided by `kedro-mlflow` and the location in Kedro template where the action should be performed.

`kedro-mlflow` does not currently provide interface to set tags outside a Kedro `Pipeline`. Some of above decisions are subject to debate and design decisions (for instance, metrics are often updated in a loop during each epoch / training iteration and it does not always make sense to register the metric between computation steps, e.g. as a an I/O operation after a node run).

Note

You do **not** need any `MLProject` file to use mlflow inside your Kedro project. As seen in the introduction, this file overlaps with Kedro configuration files.

1.2 Getting started

1.2.1 Installation guide

Pre-requisites

Create a virtual environment

I strongly recommend to create a virtual environment in order to avoid version conflicts between packages. I use conda in this tutorial.

I also recommend to read Kedro installation guide to set up your Kedro project.

```
conda create -n <your-environment-name> python=<3.[6-8].X>
```

For the rest of the section, we assume the environment is activated:

```
conda activate <your-environment-name>
```

Check your kedro version

If you have an existing environment with kedro already installed, make sure its version is above 0.16.0. kedro-mlflow cannot be used with kedro<0.16.0, and if you install it in an existing environment, it will reinstall a more up-to-date version of kedro and likely mess your project up until you reinstall the proper version of kedro (the one you originally created the project with).

```
pip show kedro
```

should return:

```
Name: kedro
Version: <your-kedro-version>  # <-- make sure it is above 0.16.0, <0.17.0
Summary: Kedro helps you build production-ready data and analytics pipelines
Home-page: https://github.com/quantumblacklabs/kedro
Author: QuantumBlack Labs
Author-email: None
License: Apache Software License (Apache 2.0)
Location: <...>\anaconda3\envs\<your-environment-name>\lib\site-packages
Requires: pip-tools, cachetools, fsspec, toposort, anyconfig, PyYAML, click, pluggy,␣
↪jmespath, python-json-logger, jupyter-client, setuptools, cookiecutter
```

Install the plugin

There are versions of the plugin compatible up to kedro>=0.16.0 and mlflow>=0.8.0. kedro-mlflow stops adding features to a minor version 2 to 6 months after a new kedro release.

Install with pip / uv

You can install kedro-mlflow plugin from PyPi with pip:

```
pip install --upgrade kedro-mlflow
```

If you prefer uv and have it installed, you can use:

```
uv pip install --upgrade kedro-mlflow
```

Install with conda / mamba / micromamba

You can install `kedro-mlflow` plugin with `conda` from the `conda-forge` channel:

```
conda install kedro-mlflow -c conda-forge
```

Install from github

You may want to install the master branch from source which has unreleased features:

```
pip install git+https://github.com/Galileo-Galilei/kedro-mlflow.git
```

Check the installation

Enter `kedro info` in a terminal with the activated virtual env to check the installation. If it has succeeded, you should see the following ascii art:

```
 _            _
| | _____  __| |_ __ ___
| |/ / _ \/ _` | '__/ _ \
|   <  __/ (_| | | | (_) |
|_|\_\___|\__,_|_|  \___/
v0.<minor>.<patch>

kedro allows teams to create analytics
projects. It is developed as part of
the Kedro initiative at QuantumBlack.

Installed plugins:
kedro_mlflow: 0.14.0 (hooks:global,project)
```

The version `0.14.0` of the plugin is installed and has both global and project commands.

That's it! You are now ready to go!

Available commands

With the `kedro mlflow -h` command outside of a kedro project, you now see the following output:

```
Usage: kedro mlflow [OPTIONS] COMMAND [ARGS]...

  Use mlflow-specific commands inside kedro project.

Options:
  -h, --help  Show this message and exit.
```

1.2.2 Initialize your Kedro project

This section assume that you have installed `kedro-mlflow` in your virtual environment.

Create a kedro project

This plugin must be used in an existing kedro project. If you do not have a kedro project yet, you can create it with `kedro new` command. See the kedro docs for a tutorial.

If you do not have a real-world project, you can use a kedro example and follow the "Quickstart in 1 mn" example to make a demo of this plugin out of the box.

Activate `kedro-mlflow` in your kedro project

In order to use the `kedro-mlflow` plugin, you need to setup its configuration and declare its hooks.

Setting up the `kedro-mlflow` configuration file

`kedro-mlflow` is configured through an `mlflow.yml` file. The recommended way to initialize the `mlflow.yml` is by using the `kedro-mlflow` CLI, but you can create it manually.

Note

Since `kedro-mlflow>=0.11.2`, the configuration file is optional. However, the plugin will use default `mlflow` configuration. Specifically, the runs will be stored in a `mlruns` folder at the root fo the kedro project since no `mlflow_tracking_uri` is configured.

Set the working directory at the root of your kedro project:

```
cd path/to/your/project
```

Run the init command :

```
kedro mlflow init
```

you should see the following message:

```
'conf/local/mlflow.yml' successfully updated.
```

Note: you can create the configuration file in another kedro environment with the `--env` *argument:*

```
kedro mlflow init --env=<other-environment>
```

Declaring `kedro-mlflow` hooks

`kedro_mlflow` hooks implementations must be registered with Kedro. There are 2 ways of registering hooks.

Important

You must register the hook provided by `kedro-mlflow` (the `MlflowHook`) to make the plugin work.

`kedro>=0.16.4` - auto-discovery

If you use `kedro>=0.16.4`, `kedro-mlflow` hooks are auto-registered automatically by default without any action from your side. You can disable this behaviour in your `settings.py` file.

`kedro>=0.16.0, <=0.16.3` - register in `settings.py`

If you have turned off plugin automatic registration, you can register its hooks manually by adding them to `settings.py`:

```python
# <your_project>/src/<your_project>/settings.py
from kedro_mlflow.framework.hooks import MlflowHook

HOOKS = (MlflowHook(),)
```

1.2.3 Goal of the tutorial

This "Getting started" section demonstrates how to use some basic functionalities of `kedro-mlflow` in an end to end example. It is supposed to be simple and self-contained and is partially redundant with other sections, but far from complete.

The **section only focuses on experiment tracking** part and **does *not* show the "machine learning framework" abilities** of the plugin. The goal is to give to a new user a quick glance to some capabilities so that he can decide whether the plugin suits its needs or not. It is totally worth checking the other sections to have a much more complete overview of what this plugin provides.

1.2.4 Example project

Install the plugin in a virtual environment

Create a conda environment and install `kedro-mlflow` (this will automatically install `kedro>=0.16.0`).

```
conda create -n km_example python=3.10 --yes
conda activate km_example
pip install kedro-mlflow
```

Install the toy project

For this end to end example, we will use the kedro starter with the iris dataset.

We use this project because:

- it covers most of the common use cases
- it is compatible with older version of `Kedro` so newcomers are used to it
- it is maintained by `Kedro` maintainers and therefore enforces some best practices.

`kedro>=0.19.0`

> **Warning**
>
> For `kedro>=0.19.0`, `pandas-iris` starter has been removed. It is recommended to install `spaceflights-pandas` starter instead.

`kedro>=0.16.3,<0.19`

The default starter is now called "pandas-iris". In a new console, enter:

```
kedro new --starter=pandas-iris
```

Answer `Kedro Mlflow Example`, `km-example` and `km_example` to the three setup questions of a new kedro project:

```
Project Name:
=============
Please enter a human readable name for your new project.
Spaces and punctuation are allowed.
 [New Kedro Project]: Kedro Mlflow Example

Repository Name:
================
```

(continues on next page)

(continued from previous page)

```
Please enter a directory name for your new project repository.
Alphanumeric characters, hyphens and underscores are allowed.
Lowercase is recommended.
 [kedro-mlflow-example]: km-example

Python Package Name:
====================
Please enter a valid Python package name for your project package.
Alphanumeric characters and underscores are allowed.
Lowercase is recommended. Package name must start with a letter or underscore.
 [kedro_mlflow_example]: km_example
```

kedro>=0.16.0, <=0.16.2

With older versions of Kedro, the starter option is not available, but this `kedro new` provides an "Include example" question. Answer `y` to this question to get the same starter as above. In a new console, enter:

```
kedro new
```

Answer `Kedro Mlflow Example`, `km-example`, `km_example` and `y` to the four setup questions of a new kedro project:

```
Project Name:
=============
Please enter a human readable name for your new project.
Spaces and punctuation are allowed.
 [New Kedro Project]: Kedro Mlflow Example

Repository Name:
================
Please enter a directory name for your new project repository.
Alphanumeric characters, hyphens and underscores are allowed.
Lowercase is recommended.
 [kedro-mlflow-example]: km-example

Python Package Name:
====================
Please enter a valid Python package name for your project package.
Alphanumeric characters and underscores are allowed.
Lowercase is recommended. Package name must start with a letter or underscore.
 [kedro_mlflow_example]: km_example

Generate Example Pipeline:
==========================
Do you want to generate an example pipeline in your project?
Good for first-time users. (default=N)
 [y/N]: y
```

Install dependencies

Move to the project directory:

```
cd km-example
```

Install the project dependencies :

Warning

Do not use `kedro install` commands which does not install the packages in your activated environment. It has been removed in `kedro>=0.19`.

```
pip install -r src/requirements.txt
```

1.2.5 First steps with the plugin

Initialize kedro-mlflow

Note

This step is optional if you use `kedro>=0.11.2`. If you do not create a `mlflow.yml` configuration file, `kedro-mlflow` will use the defaults. However this is heavily recommended because in professional setup you often need some specific enterprise configuration.

(Optional) Create a configuration file

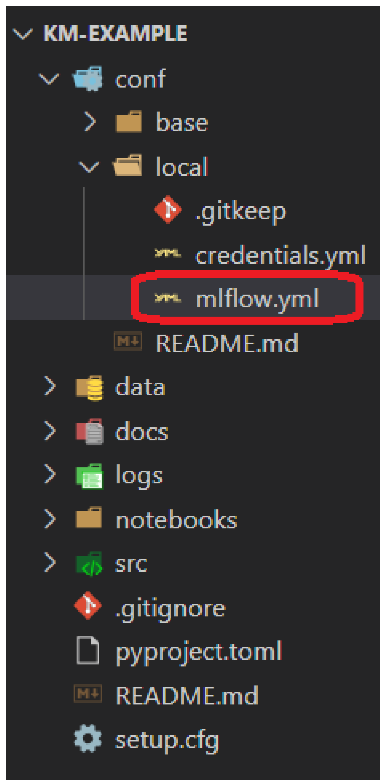
You can initialize your project with the plugin-specific configuration file with this command:

```
kedro mlflow init --env=local
```

You will see the following message:

```
'conf/local/mlflow.yml' successfully updated.
```

The `conf/local` folder is updated and you can see the `mlflow.yml` file:



(Optional) Specify the tracking uri

If you have configured your own mlflow server, you can specify the tracking uri in the `mlflow.yml` (replace the highlighted line below):

mlflow.yml

conf > base > mlflow.yml

```yaml
# GLOBAL CONFIGURATION -------------------

# `mlflow_tracking_uri` is the path where the runs will be recorded.
# For more informations, see https://www.mlflow.org/docs/latest/tracking.html#where-runs-are-recorded
# kedro-mlflow accepts relative path from the project root.
# For instance, default `mlruns` will create a mlruns folder
# at the root of the project
mlflow_tracking_uri: mlruns


# EXPERIMENT-RELATED PARAMETERS ----------

# `name` is the name of the experiment (~subfolder
# where the runs are recorded). Change the name to
# switch between different experiments
experiment:
  name: km_example
  create: True  # if the specified `name` does not exists, should it be created?


# RUN-RELATED PARAMETERS -----------------

run:
  id: null # if `id` is None, a new run will be created
  name: null # if `name` is None, pipeline name will be used for the run name
  nested: True  # # if `nested` is False, you won't be able to launch sub-runs inside your nodes

# UI-RELATED PARAMETERS ----------------

ui:
  port: null  # the port to use for the ui. Find a free port if null.
  host: null  # the host to use for the ui. Default to "localhost" if null.
```

Run the pipeline

Open a new command and launch

```
kedro run
```

If the pipeline executes properly, you should see the following log:

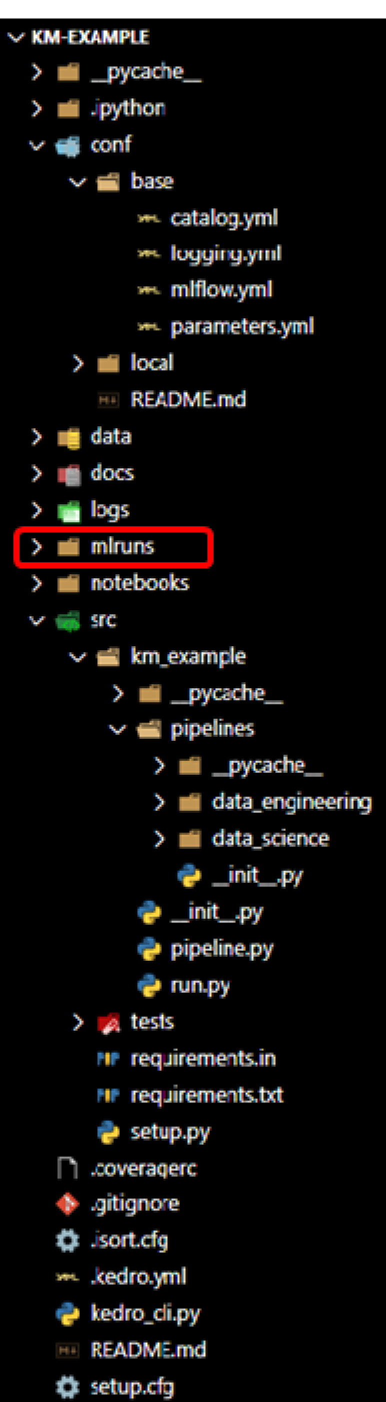
```
2020-07-13 21:29:25,401 - kedro.io.data_catalog - INFO - Loading data from `example_iris_data` (CSVDataset)...
2020-07-13 21:29:25,562 - kedro.io.data_catalog - INFO - Loading data from `params:example_test_data_ratio` (MemoryDataset)...
2020-07-13 21:29:25,969 - kedro.pipeline.node - INFO - Running node: split_data([example_iris_data,params:example_test_data_ratio]) -> [example_test_x,example_test_y,example_train_x,example_train_y]
2020-07-13 21:29:26,053 - kedro.io.data_catalog - INFO - Saving data to `example_train_x` (MemoryDataset)...
2020-07-13 21:29:26,368 - kedro.io.data_catalog - INFO - Saving data to `example_train_y` (MemoryDataset)...
2020-07-13 21:29:26,484 - kedro.io.data_catalog - INFO - Saving data to `example_test_x` (MemoryDataset)...
2020-07-13 21:29:26,486 - kedro.io.data_catalog - INFO - Saving data to `example_test_y` 
```

(continues on next page)

(continued from previous page)

```
↪(MemoryDataset)...
2020-07-13 21:29:26,610 - kedro.runner.sequential_runner - INFO - Completed 1 out of 4␣
↪tasks
2020-07-13 21:29:26,850 - kedro.io.data_catalog - INFO - Loading data from `example_
↪train_x` (MemoryDataset)...
2020-07-13 21:29:26,851 - kedro.io.data_catalog - INFO - Loading data from `example_
↪train_y` (MemoryDataset)...
2020-07-13 21:29:26,965 - kedro.io.data_catalog - INFO - Loading data from `parameters`␣
↪(MemoryDataset)...
2020-07-13 21:29:26,972 - kedro.pipeline.node - INFO - Running node: train_
↪model([example_train_x,example_train_y,parameters]) -> [example_model]
2020-07-13 21:29:27,756 - kedro.io.data_catalog - INFO - Saving data to `example_model`␣
↪(MemoryDataset)...
2020-07-13 21:29:27,763 - kedro.runner.sequential_runner - INFO - Completed 2 out of 4␣
↪tasks
2020-07-13 21:29:28,141 - kedro.io.data_catalog - INFO - Loading data from `example_
↪model` (MemoryDataset)...
2020-07-13 21:29:28,161 - kedro.io.data_catalog - INFO - Loading data from `example_test_
↪x` (MemoryDataset)...
2020-07-13 21:29:28,670 - kedro.pipeline.node - INFO - Running node: predict([example_
↪model,example_test_x]) -> [example_predictions]
2020-07-13 21:29:29,002 - kedro.io.data_catalog - INFO - Saving data to `example_
↪predictions` (MemoryDataset)...
2020-07-13 21:29:29,248 - kedro.runner.sequential_runner - INFO - Completed 3 out of 4␣
↪tasks
2020-07-13 21:29:29,433 - kedro.io.data_catalog - INFO - Loading data from `example_
↪predictions` (MemoryDataset)...
2020-07-13 21:29:29,730 - kedro.io.data_catalog - INFO - Loading data from `example_test_
↪y` (MemoryDataset)...
2020-07-13 21:29:29,911 - kedro.pipeline.node - INFO - Running node: report_
↪accuracy([example_predictions,example_test_y]) -> None
2020-07-13 21:29:30,056 - km_example.pipelines.data_science.nodes - INFO - Model␣
↪accuracy on test set: 100.00%
2020-07-13 21:29:30,214 - kedro.runner.sequential_runner - INFO - Completed 4 out of 4␣
↪tasks
2020-07-13 21:29:30,372 - kedro.runner.sequential_runner - INFO - Pipeline execution␣
↪completed successfully.
```

Since we have kept the default value of the `mlflow.yml`, the tracking uri (the place where runs are recorded) is a local `mlruns` folder which has just been created with the execution:



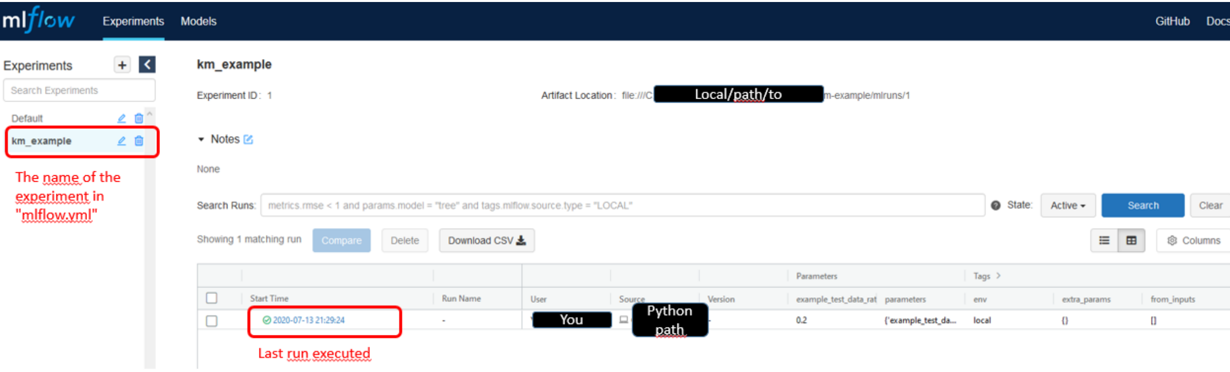
Open the UI

Launch the ui:

```
kedro mlflow ui
```

And open the following adress in your favorite browser

`http://localhost:5000/`



Click now on the last run executed, you will land on this page:

km_example > Run 9128c4c15e2c438db27749561f543c97

Date: 2020-07-13 21:29:24

Source: \km_example\Scripts\kedro

Duration: 5.6s

Status: FINISHED

Notes

None

Parameters

Name	Value
example_test_data_ratio	0.2
parameters	{'example_test_data_ratio': 0.2, 'example_num_train_iter': 10000, 'example_learning_rate': 0.01}

Metrics

Name	Value

Tags

Name	Value	Actions
env	local	
extra_params	{}	
from_inputs	[]	
from_nodes	[]	
git_sha	None	
kedro_command	kedro run	
kedro_version	0.16.3	
load_versions	{}	
node_names	()	
pipeline_name	None	
project_path	\km-example	
run_id	2020-07-13T19.29.20.514Z	
tags	()	
to_nodes	[]	

Add Tag

Name | Value | Add

Artifacts

No Artifacts Recorded

Use the log artifact APIs to store file outputs from MLflow runs.

Parameters tracking

Note that the parameters have been recorded *automagically*. Here, two parameters format are used:

1. The parameter `example_test_data_ratio`, which is called in the `pipeline.py` file with the `params:` prefix
2. the dictionary of all parameters in `parameters.yml` which is a reserved key word in `Kedro`. Note that **this is bad practice** because you cannot know which parameters are really used inside the function called. Another problem is that it can generate too long parameters names and lead to mlflow errors.

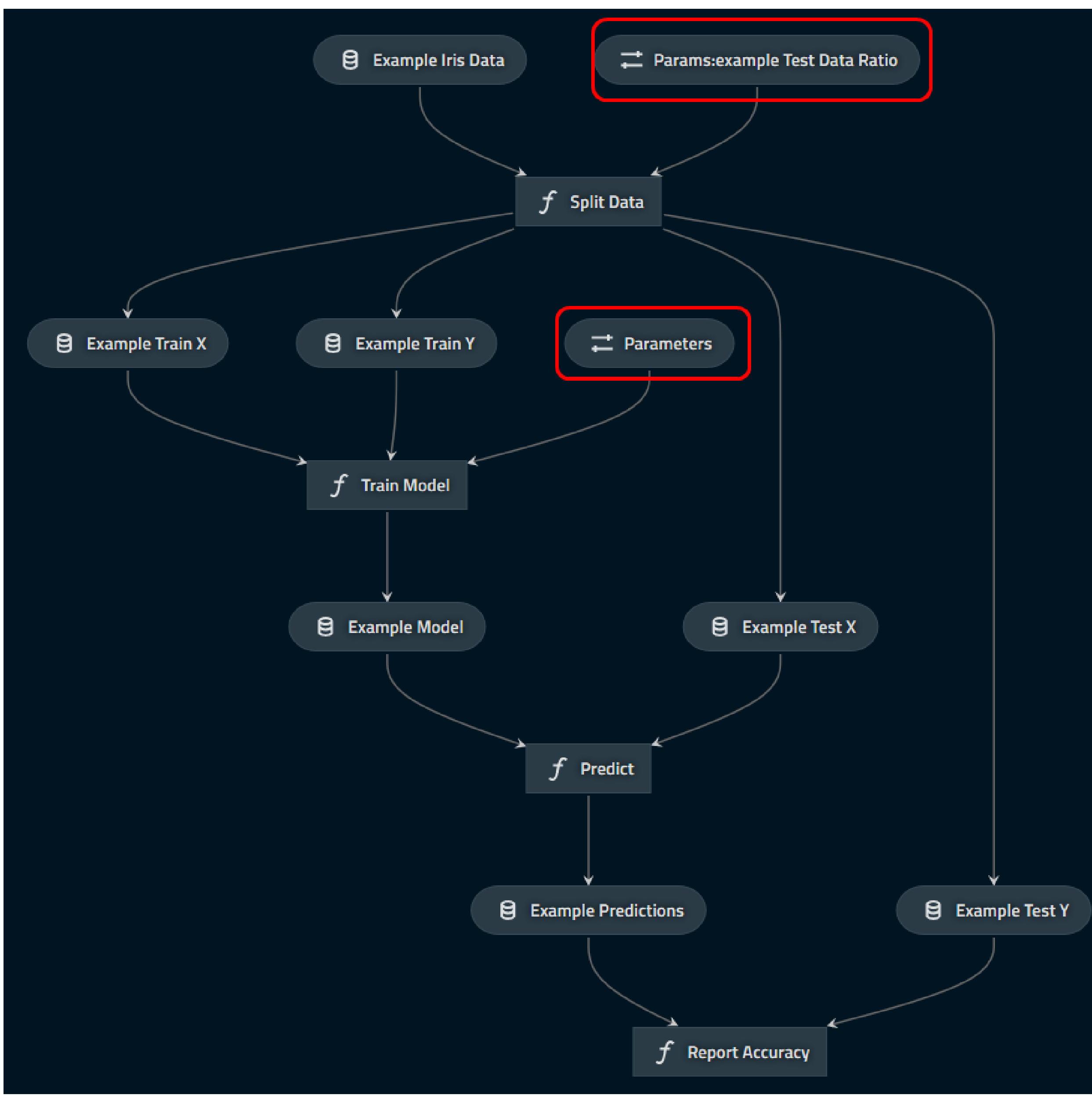
You can see that these are effectively the registered parameters in the pipeline with the `kedro-viz` plugin:

```
pip install kedro-viz
kedro viz
```

Open your browser at the following adress:

http://localhost:4141/

You should see the following graph:



which indicates clearly which parameters are logged (in the red boxes with the "parameter" icon).

Artifacts tracking

With this run, artifacts are empty. This is expected: mlflow does not know what it should log and it will not log all your data by default. However, you want to save your model (at least) or your run is likely useless!

First, open the `catalog.yml` file which should like this:

```yaml
# This is a data set used by the "Hello World" example pipeline provided with the project
# template. Please feel free to remove it once you remove the example pipeline.

example_iris_data:
  type: pandas.CSVDataset
  filepath: data/01_raw/iris.csv
```

And persist the model as a pickle with the `MlflowArtifactDataset` class:

```yaml
# This is a data set used by the "Hello World" example pipeline provided with the project
# template. Please feel free to remove it once you remove the example pipeline.

example_iris_data:
  type: pandas.CSVDataset
  filepath: data/01_raw/iris.csv

example_model:
  type: kedro_mlflow.io.artifacts.MlflowArtifactDataset
  dataset:
    type: pickle.PickleDataset
    filepath: data/06_models/trained_model.pkl
```

Rerun the pipeline (with `kedro run`), and reopen the UI. Select the last run and see that the file was uploaded:

This works for any type of file (including images with `MatplotlibWriter`) and the UI even offers a preview for `png` and `csv`, which is really convenient to compare runs.

Note: Mlflow offers specific logging for machine learning models that may be better suited for your use case, see `MlflowModelTrackingDataset`

Going further

Above vanilla example is just the beginning of your experience with `kedro-mlflow`. Check out the next sections to see how `kedro-mlflow`:

- offers advanced capabilities for machine learning versioning
- offers a way to create custom mlflow model from your kedro pipelines to deploy effortlessly in production

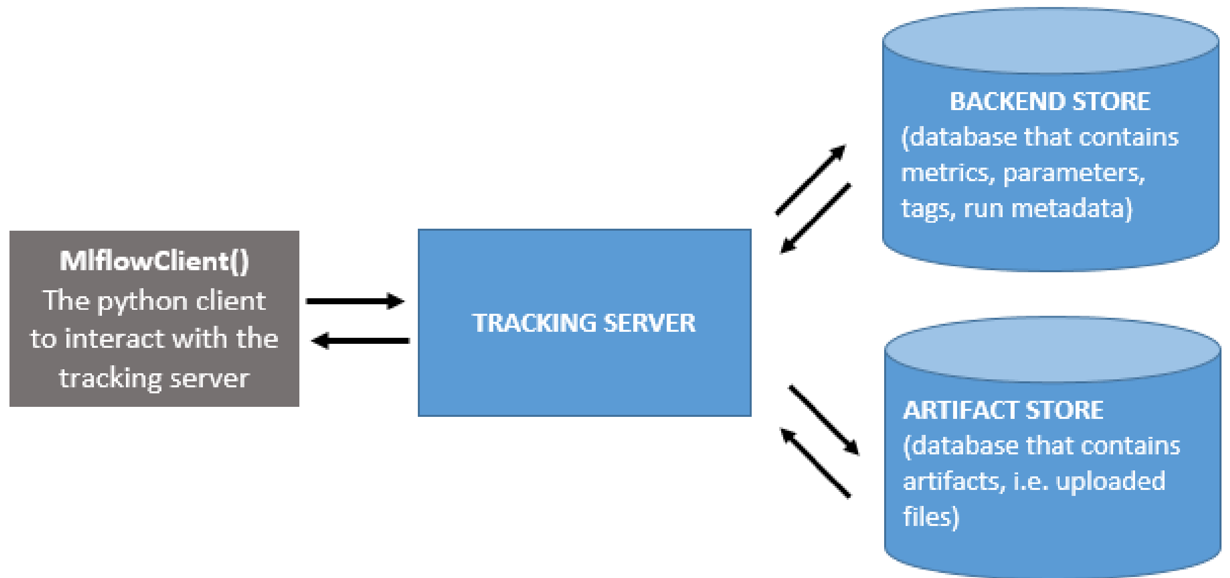
1.3 Experiment tracking

1.3.1 Configure mlflow inside your project

We assume in this section that you have installed `kedro-mlflow` in your virtual environment and you have configured your project with a `mlflow.yml` configuration file and hooks declaration.

Context: mlflow tracking under the hood

Mlflow is composed of four modules which are described in the introduction section. The main module is "tracking". The goal of this module is to keep track of every varying parameters across different code execution (parameters, metrics and artifacts). The following schema describes how this modules operates under the hood:



Basically, this schema shows that mlflow separates WHERE the artifacts are logged from HOW they are logged inside your code. You need to setup your mlflow tracking server separately from your code, and then each logging will send a request to the tracking server to store the elements you want to track in the appropriate location. The advantage of such a setup are numerous:

- once the mlflow tracking server is setup, there is single parameter to set before logging which is the tracking server uri. This makes configuration very easy in your project.
- since the different storage locations are well identified, it is easy to define custom solutions for each of them. They can be database or even local folders.

The rationale behind the separation of the backend store and the artifacts store is that artifacts can be very big and are duplicated across runs, so they need a special management with extensible storage. This is typically cloud storage like AWS S3 or Azure Blob storage.

The `mlflow.yml` file

The `mlflow.yml` file contains all configuration you can pass either to kedro or mlflow through the plugin. Note that you can duplicate `mlflow.yml` file in as many environments (i.e. `conf/` folders) as you need. To create a `mlflow.yml` file in a kedro configuration environment, use `kedro mlflow init --env=<your-env>`. You ‘ll get the following result:

```
# SERVER CONFIGURATION -------------------

# `mlflow_tracking_uri` is the path where the runs will be recorded.
# For more informations, see https://www.mlflow.org/docs/latest/tracking.html#where-runs-
↪are-recorded
# kedro-mlflow accepts relative path from the project root.
# For instance, default `mlruns` will create a mlruns folder
# at the root of the project

# All credentials needed for mlflow must be stored in credentials .yml as a dict
# they will be exported as environment variable
# If you want to set some credentials,  e.g. AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY
# > in `credentials.yml`:
```

(continues on next page)

(continued from previous page)

```
# your_mlflow_credentials:
#   AWS_ACCESS_KEY_ID: 132456
#   AWS_SECRET_ACCESS_KEY: 132456
# > in this file `mlflow.yml`:
# credentials: mlflow_credentials

server:
  mlflow_tracking_uri: null # if null, will use mlflow.get_tracking_uri() as a default
  mlflow_registry_uri: null # if null, mlflow_tracking_uri will be used as mlflow default
  credentials: null  # must be a valid key in credentials.yml which refers to a dict of␣
↪sensitive mlflow environment variables (password, tokens...). See top of the file.
  request_header_provider: # this is only useful to deal with expiring token, see https:/
↪/github.com/Galileo-Galilei/kedro-mlflow/issues/357
    type: null # The path to a class : my_project.pipelines.module.MyClass. Should␣
↪inherit from https://github.com/mlflow/mlflow/blob/master/mlflow/tracking/request_
↪header/abstract_request_header_provider.py#L4
    pass_context: False # should the class be instantiated with "kedro_context" argument?
    init_kwargs: {} # any kwargs to pass to the class when it is instantiated

tracking:
  # You can specify a list of pipeline names for which tracking will be disabled
  # Running "kedro run --pipeline=<pipeline_name>" will not log parameters
  # in a new mlflow run

  disable_tracking:
    disable_autologging: True  # If True, we force autologging to be disabled. This is␣
↪useful on databricks with autologging by default which conflicts with the plugin. If␣
↪False, we keep the default behaviour which is disable by default anayway.
    pipelines: []

  experiment:
    name: {{ python_package }}
    create_experiment_kwargs: # will be used only if the experiment does not exist yet␣
↪and is created.
      artifact_location: null # enable to specify an artifact location for the␣
↪experiment different than the global one for the mlflow server
      tags: null  # a dict of tags for the experiment
    restore_if_deleted: True  # if the experiment`name` was previously deleted experiment,
↪ should we restore it?

  run:
    id: null # if `id` is None, a new run will be created
    name: null # if `name` is None, pipeline name will be used for the run name. You can␣
↪use "${km.random_name:}" to generate a random name (mlflow's default)
    nested: True  # if `nested` is False, you won't be able to launch sub-runs inside␣
↪your nodes

  params:
    dict_params:
      flatten: False  # if True, parameter which are dictionary will be splitted in␣
↪multiple parameters when logged in mlflow, one for each key.
      recursive: True  # Should the dictionary flattening be applied recursively (i.e.␣
```

(continues on next page)

(continued from previous page)

```
↪for nested dictionaries)? Not use if `flatten_dict_params` is False.
      sep: "." # In case of recursive flattening, what separator should be used between␣
↪the keys? E.g. {hyperaparam1: {p1:1, p2:2}} will be logged as hyperaparam1.p1 and␣
↪hyperaparam1.p2 in mlflow.
    long_params_strategy: fail # One of ["fail", "tag", "truncate" ] If a parameter is␣
↪above mlflow limit (currently 250), what should kedro-mlflow do? -> fail, set as a tag␣
↪instead of a parameter, or truncate it to its 250 first letters?


# UI-RELATED PARAMETERS -----------------

ui:
  port: "5000" # the port to use for the ui. Use mlflow default with 5000.
  host: "127.0.0.1"  # the host to use for the ui. Use mlflow efault of "127.0.0.1".
```

Note

If no `mlflow.yml` file is found in the environment, `kedro-mlflow` will still work and use all `mlflow.yml` default values as configuration.

Important

If the kedro run is started in a process where a mlflow run is already active, `kedro-mlflow` will ignore all the configuration in `mlflow.yml` and use the active run. The mlflow run will NOT be closed at the end of the kedro run. This enables using `kedro-mlflow` with an orchestrator (e.g airflow, AzureML...) which starts the mlflow run itself.

Configure the tracking server

Configure the tracking and registry uri

`kedro-mlflow` needs the tracking uri of your mlflow tracking server to operate properly. The `mlflow.yml` file must have the `mlflow_tracking_uri` key with a valid mlflow_tracking_uri associated value. The `mlflow.yml` default have this keys set to `null`. This means that it will look for a `MLFLOW_TRACKING_URI` environment variable, and if it is not set, it will create a `mlruns` folder locally at the root of your kedro project. This enables you to use the plugin without any setup of a mlflow tracking server.

Tip

Unlike mlflow, `kedro-mlflow` allows the `mlflow_tracking_uri` to be a relative path. It will convert it to an absolute uri automatically and prefix it with `file:///`.

```
server:
  mlflow_tracking_uri: mlruns # or http://path/your/server
```

You can also specify the registry uri:

```yaml
server:
  mlflow_registry_uri: sqlite:///path/to/registry.db
```

Important

Unlike the `mlflow_tracking_uri`, the `mlflow_registry_uri` must be an *absolute* path prefixed with the database dialect of your database, likely `sqlite:///` for a local database.

Configure the credentials

Default credentials with environment variables

You can also specify some environment variables needed by mlflow (e.g `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`) in the credentials and specify them in the `mlflow.yml`. Any key specified will be automatically exported as environment variables.

Your `credentials.yml` will look as follows:

```yaml
my_mlflow_credentials:
  AWS_ACCESS_KEY_ID: <your-key>
  AWS_SECRET_ACCESS_KEY: <your-secret-key>
```

and your can supply the credentials key of the `mlflow.yml`:

```yaml
server:
  credentials: my_mlflow_credentials
```

Note

For security reasons, the credentials will not be accessible within `KedroMlflowConfig` objects. They will be exported as environment variables *on the fly* when running the pipeline.

Authentication with expiring tokens

Mlflow can be deployed with OAuth2.0. authentication method : in this case, secured MLflow instances require HTTP requests to have the `Authorization: Bearer <token>` header. Mlflow exposes a `RequestHeaderProvider` abstract class to mange this use case. If you need kedro-mlflow to use a *custom* header provider, you can configure your `mlflow.yml` as follow:

```yaml
# mlflow.yml
server:
    request_header_provider:
        type: path.to.your.class.CustomRequestHeaderProvider
        pass_context: True # if you want to pass context. it must be named ``kedro_
↪context`` in the ``__init__`` method of your custom ``request_header_provider``)
        init_kwargs:
            my_kwarg: 1
```

This will automatically register in the mlflow entrypoint the `CustomRequestHeaderProvider(kedro_context=<kedro-context>, my_kwarg=1)` request header when running a kedro pipeline.

Deactivate tracking under conditions

kedro-mlflow logs every run parameters in mlflow. You may want to avoid tracking some runs (for instance while debugging to avoid polluting your mlflow database, or because some pipelines are not ml related and it does not makes sense to log their parameters).

You can specify the name of the pipelines you want to turn off:

```
tracking:
  disable_tracking:
    pipelines:
      - <pipeline-name>
```

Notice that it will stop autologging parameters but also any `Mlflow<Artifact/Metrics/ModelTracking>Dataset` you may have in these deactivated pipelines.

Configure mlflow experiment

Mlflow enable the user to create "experiments" to organize his work. The different experiments will be visible on the left panel of the mlflow user interface. You can create an experiment through the `mlflow.yml` file with the `experiment` key:

```
tracking:
  experiment:
    name: {{ python_package }}
    create_experiment_kwargs: # will be used only if the experiment does not exist yet␣
↪and is created.
      artifact_location: null # enable to specify an artifact location for the␣
↪experiment different than the global one for the mlflow server
      tags: null  # a dict of tags for the experiment
    restore_if_deleted: True  # if the experiment`name` was previously deleted experiment,
↪ should we restore it?
```

Configure the run

When you launch a new `kedro` run, `kedro-mlflow` instantiates an underlying `mlflow` run through the hooks. By default, we assume the user want to launch each kedro run in separated mlflow run to keep a one to one relationship between kedro runs and mlflow runs. However, one may need to *continue* an existing mlflow run (for instance, because you resume the kedro run from a later starting point of your pipeline).

The `mlflow.yml` accepts the following keys:

```
tracking:
  run:
    id: null # if `id` is None, a new run will be created
    name: null # if `name` is None, pipeline name will be used for the run name. You can␣
↪use "${km.random_name:}" to generate a random name (mlflow's default)
    nested: True  # if `nested` is False, you won't be able to launch sub-runs inside␣
↪your nodes
```

Tip

If you want to generate a random name for each run (like mlflow's default), you can use the built-in `km.random_name` resolver:

```yaml
tracking:
  run:
    name: ${km.random_name:} # don't forget the trailing ":" at the end !
```

- If you want to continue to log in an existing mlflow run, write its id in the `id` key.
- If you want to enable the creation of sub runs inside your nodes (for instance, for model comparison or hyperparameter tuning), set the `nested` key to `True`

Extra tracking configuration

You may sometimes encounter an mlflow failure "parameters too long". Mlflow has indeed an upper limit on the length of the parameters you can store in it. This is a very common pattern when you log a full dictionary in mlflow (e.g. the reserved keyword `parameters` in kedro, or a dictionnary conaining all the hyperparameters you want to tune for a given model). You can configure the `kedro-mlflow` hooks to overcome this limitation by "flattening" automatically dictionaries in a kedro run.

The `mlflow.yml` accepts the following keys:

```yaml
tracking:
  params:
    dict_params:
      flatten: False  # if True, parameter which are dictionary will be splitted in␣
↪multiple parameters when logged in mlflow, one for each key.
      recursive: True  # Should the dictionary flattening be applied recursively (i.e␣
↪for nested dictionaries)? Not use if `flatten_dict_params` is False.
      sep: "." # In case of recursive flattening, what separator should be used between␣
↪the keys? E.g. {hyperaparam1: {p1:1, p2:2}} will be logged as hyperaparam1.p1 and␣
↪hyperaparam1.p2 in mlflow.
    long_params_strategy: fail # One of ["fail", "tag", "truncate" ] If a parameter is␣
↪above mlflow limit (currently 250), what should kedro-mlflow do? -> fail, set as a tag␣
↪instead of a parameter, or truncate it to its 250 first letters?
```

If you set `flatten` to `True`, each key of the dictionary will be logged as a mlflow parameters, instead of a single parameter for the whole dictionary. Note that it is recommended to facilitate run comparison.

The `long_parameters_strategy` key enable to define different way to handle parameters over the mlflow limit (currently 250 characters):

- `fail`: no special management of characters above the limit. They will be send to mlflow and as a result, in some backend they will be stored normally (e.g. for FileStore backend) and for some others logging will fail.
- `truncate`: All parameters above the limit will be automatically truncated to a 250-character length to make sure logging will pass for all mlflow backend.
- `tag`: Any parameter above the limit will be registered as a tag instead of a parameter as it seems to be the recommended mlflow way to deal with long parameters.

Configure the user interface

You can configure mlflow user interface default params inside the `mlflow.yml`:

```yaml
ui:
  port: null  # the port to use for the ui. Find a free port if null.
  host: null  # the host to use for the ui. Default to "localhost" if null.
```

The port and host parameters set in this configuration will be used by default if you use `kedro mlflow ui` command (instead of `mlflow ui`) to open the user interface. Note that the `kedro mlflow ui` command will also use the `mlflow_tracking_uri` key set inside `mlflow.yml`.

Overwrite configuration at runtime

In `kedro>=0.19.0`, it is possible to overwrite configuration at runtime with the `runtime_params` resolver.

Tip

`mlflow.yml` is handled exactly as Kedro native configuration files, so anything which is possible within `catalog.yml` or `parameters.yml` should work within `mlflow.yml`

For instance, assume you want to specify the run name through the CLI, you can configure your `mlflow.yml` as follows:

```
# mlflow.yml
tracking:
  run:
    name: ${runtime_params:mlflow_run_name, null} # mlflow_run_name can be any name you want
```

And then pass `mlflow_run_name` runtime param through the CLI:

```
kedro run --params mlflow_run_name="my-awesome-name"
```

Note

The `, null` part of above configuration is the default value which will be used if `mlflow_run_name` is not specified at runtime. In this case, `kedro-mlflow` will use the pipeline name as a run name.

You can nest resolvers, so the following config will default to a random name if the `mlflow_run_name` is not specified at runtime:

```
# mlflow.yml
tracking:
  run:
    name: ${runtime_params:mlflow_run_name, ${km.random_name:}}
```

1.3.2 Track parameters

Automatic parameters tracking

Parameters tracking is automatic when the `MlflowHook` is added to the hook list of the `ProjectContext`. The `mlflow.yml` configuration file has a parameter called `flatten_dict_params` which enables to log as distinct parameters the (key, value) pairs of a ```dict`` parameter.

You **do not need any additional configuration** to benefit from parameters versioning.

Hint

How does `MlflowHook` operates under the hood?

The medium post which introduces hooks explains in detail the steps `Kedro` executes when the user calls the `kedro run` command.

The `MlflowHook` registers the parameters before each node (entry point number 3 on above picture) by calling `mlflow.log_parameter(param_name, param_value)` on each parameters of the node.

Frequently asked questions

How are parameters detected by the plugin?

The hook **detects parameters through their prefix `params:` or the value `parameters`**. These are the reserved keywords used by Kedro to define parameters in the `pipeline.py` file(s).

Will parameters be recorded if the pipeline fails during execution?

The parameters are registered node by node (and not in a single batch at the beginning of the execution). If the pipeline fails in the middle of its execution, the **parameters of the nodes who have been run will be recorded**, but **not the parameters of non executed nodes**.

1.3.3 Track Datasets as artifacts

What is artifact tracking?

Mlflow defines artifacts as "any data a user may want to track during code execution". This includes, but is not limited to:

- data needed for the model (e.g encoders, vectorizer, the machine learning model itself…)
- graphs (e.g. ROC or PR curve, importance variables, margins, confusion matrix…)

Artifacts are a very flexible and convenient way to "bind" any data type to your code execution. Mlflow has a two-step process for such binding:

1. Persist the data locally in the desired file format
2. Upload the data to the artifact store

How to track data in a kedro project?

kedro-mlflow introduces a new AbstractDataset called MlflowArtifactDataset. It is a wrapper for any AbstractDataset which decorates the underlying dataset save method and logs the file automatically in mlflow as an artifact each time the save method is called.

Since it is an AbstractDataset, it can be used with the YAML API. Assume that you have the following entry in the catalog.yml:

```yaml
my_dataset_to_track:
    type: pandas.CSVDataset
    filepath: /path/to/a/destination/file.csv
```

You can change it to:

```yaml
my_dataset_to_track:
    type: kedro_mlflow.io.artifacts.MlflowArtifactDataset
    dataset:
        type: pandas.CSVDataset  # or any valid kedro DataSet
        filepath: /path/to/a/LOCAL/destination/file.csv # must be a local file, wherever␣
↪you want to log the data in the end
```

and this dataset will be automatically versioned in each pipeline execution.

Frequently asked questions

Can I pass extra parameters to the MlflowArtifactDataset for finer control?

The MlflowArtifactDataset takes a dataset argument which is a python dictionary passed to the __init__ method of the dataset declared in type. It means that you can pass any argument accepted by the underlying dataset in this dictionary. If you want to pass load_args and save_args in the previous example, add them in the dataset argument:

```yaml
my_dataset_to_version:
    type: kedro_mlflow.io.artifacts.MlflowArtifactDataset
    dataset:
        type: pandas.CSVDataset  # or any valid kedro DataSet
        filepath: /path/to/a/local/destination/file.csv
        load_args:
            sep: ;
        save_args:
            sep: ;
        # ... any other valid arguments for dataset
```

Can I use the MlflowArtifactDataset in interactive mode?

Like all Kedro AbstractDataset, MlflowArtifactDataset is callable in the python API:

```python
from kedro_mlflow.io.artifacts import MlflowArtifactDataset
from kedro_datasets.pandas import CSVDataset

csv_dataset = MlflowArtifactDataSet(
    dataset={
        "type": CSVDataset,  # either a string "pandas.CSVDataset" or the class
        "filepath": r"/path/to/a/local/destination/file.csv",
    }
```

(continues on next page)

(continued from previous page)

```
)
csv_dataset.save(data=pd.DataFrame({"a": [1, 2], "b": [3, 4]}))
```

How do I upload an artifact to a non local destination (e.g. an S3 or blog storage)?

The location where artifact will be stored does not depends of the logging function but rather on the artifact store specified when configuring the mlflow server. Read mlflow documentation to see:

- how to configure a mlflow tracking server
- how to configure an artifact store with cloud storage.

Setting the `mlflow_tracking_uri` key of `mlflow.yml` to the url of this properly configured server is the only additional configuration you need to send your datasets to this remote server.

> **Important**
>
> You still need to specify a **local** path for the underlying dataset (even to store it on a remote storage), mlflow will take care of the upload to the server by itself.

You can refer to this issue for further details.

Can I log an artifact in a specific run?

The `MlflowArtifactDataset` has an extra attribute `run_id` which specifies the run you will log the artifact in. **Be cautious, because this argument will take precedence over the current run** when you call `kedro run`, causing the artifact to be logged in another run that all the other data of the run.

```
my_dataset_to_version:
    type: kedro_mlflow.io.artifacts.MlflowArtifactDataset
    dataset:
        type: pandas.CSVDataset  # or any valid kedro DataSet
        filepath: /path/to/a/local/destination/file.csv  # must be a local filepath, no␣
↪matter what is your actual mlflow storage (S3 or other)
    run_id: 13245678910111213  # a valid mlflow run to log in. If None, default to␣
↪active run
```

Can I reload an artifact from an existing run to use it in another run ?

You may want to reuse th artifact of a previous run to reuse it in another one, e.g. to continue training from a pretrained model, or to select the best model among several runs created during an hyperparamter tuning. The `MlflowArtifactDataset` has an extra attribute `run_id` you can use to specify from which run you will load the artifact from. **Be cautious**, because:

- this argument will take precedence over the current run** when you call `kedro run`, causing the artifact to be loaded from another run that all the other data of the run
- the artifact will be downloaded and erase the existing file at your local filepath

```
my_dataset_to_reload:
    type: kedro_mlflow.io.artifacts.MlflowArtifactDataset
    dataset:
        type: pandas.CSVDataset  # or any valid kedro DataSet
        filepath: /path/to/a/local/destination/file.csv # must be a local filepath, no␣
```

(continues on next page)

(continued from previous page)

```
↪matter what is your actual mlflow storage (S3 or other)
    run_id: 13245678910111213  # a valid mlflow run with the existing artifact. It must␣
↪be named "file.csv"
```

Can I create a remote folder/subfolders architecture to organize the artifacts?

The `MlflowArtifactDataset` has an extra argument `artifact_path` which specifies a remote subfolder where the artifact will be logged. It must be a relative path.

With below example, the artifact will be logged in mlflow within a `reporting` folder.

```
my_dataset_to_version:
    type: kedro_mlflow.io.artifacts.MlflowArtifactDataset
    dataset:
        type: pandas.CSVDataset  # or any valid kedro DataSet
        filepath: /path/to/a/local/destination/file.csv
    artifact_path: reporting  # relative path where the remote artifact must be stored.␣
↪if None, saved in root folder.
```

Why does my `PartitionedDataset` log partitions from all previous runs?

Kedro provides a few datasets saving results in partitions (i.e., multiple files), most notably the `PartitionedDataset` and `MatplotlibDataset`.

Both would by default **not** delete partitions generated by previous runs if those from the current run don't overwrite exactly the same filenames.

However, for experiment tracking purposes, you would typically want to wipe all previously saved partitions, and log only those produced by current run.

This can be achieved by specifying `overwrite: true` in dataset specification, like so:

```
profiling.partitioned_scatterplots:
    type: kedro_mlflow.io.artifacts.MlflowArtifactDataset
    artifact_path: profiling
    dataset:
        type: matplotlib.MatplotlibDataset
        filepath: data/08_reporting/partitioned_scatterplots/
        overwrite: true  # Defaults to false, but we change it to ensure reproducibility
```

There is an ongoing discussion that maybe `true` should be the default, you can follow it here.

1.3.4 Track models

What is model tracking?

MLflow allows to serialize and deserialize models to a common format, track those models in MLflow Tracking and manage them using MLflow Model Registry. Many popular Machine / Deep Learning frameworks have built-in support through what MLflow calls flavors. Even if there is no flavor for your framework of choice, it is easy to create your own flavor and integrate it with MLflow.

How to track models using MLflow in Kedro project?

`kedro-mlflow` introduces two new `DataSet` types that can be used in `DataCatalog` called `MlflowModelTrackingDataset` and `MlflowModelLocalFileSystemDataset`. The two have very similar API, except that:

- the `MlflowModelTrackingDataset` is used to load from and save to from the mlflow artifact store. It can load from any given `model_uri`, including registered model, local models, models tied to a run…
- the `MlflowModelLocalFileSystemDataset` is used to load from and save to a given local path. It uses the standard `filepath` argument in the constructor of Kedro DataSets. Note that it **does not log in mlflow**.

Note: If you use `MlflowModelTrackingDataset`, it will be saved as a "LoggedModel" object and will be linked to your current run. However, you will need to specify the run id to predict with (since it is not persisted locally, it will not pick the latest model by default). You may prefer to combine `MlflowModelLocalFileSystemDataset` and `MlflowArtifactDataset` to make persist it both locally and remotely, see further.

Suppose you would like to register a `scikit-learn` model of your `DataCatalog` in mlflow, you can use the following yaml API:

```yaml
my_sklearn_model:
    type: kedro_mlflow.io.models.MlflowModelTrackingDataset
    flavor: mlflow.sklearn
```

More informations on available parameters are available in the dedicated section.

You are now able to use `my_sklearn_model` in your nodes. Since this model is registered in mlflow, you can also leverage the mlflow model serving abilities or predicting on batch abilities, as well as the mlflow models registry to manage the lifecycle of this model.

Frequently asked questions

How is it working under the hood?

For `MlflowModelTrackingDataset`

During save, a model object from node output is logged to mlflow using `log_model` function of the specified `flavor`. It is logged as a `LoggedModel` mlflow object, and eventually (this will actually always happen if you are using the `kedro run` command, because the `MlflowHook` creates a new run before each pipeline run) linked to your currently active mlflow run. If the `model_uri` is specified, the saving operation will fail. Consequently it will **never be possible to save in a specific mlflow run_id** if you launch a pipeline with the `kedro run` command because the `MlflowHook` creates a new run before each pipeline run.

During load, the model is retrieved from the `model_uri` if specified, else the previously saved is reloaded.

For `MlflowModelLocalFileSystemDataset`

During save, a model object from node output is saved locally under specified `filepath` using `save_model` function of the specified `flavor`.

When model is loaded, the latest version stored locally is read using `load_model` function of the specified `flavor`. You can also load a model from a specific kedro run by specifying the `version` argument to the constructor.

How can I track a custom MLflow model flavor?

To track a custom MLflow model flavor you need to set the `flavor` parameter to import the module of your custom flavor and to specify a pyfunc workflow which can be set either to `python_model` or `loader_module`. The former is the more high level and user friendly and is recommend by mlflow while the latter offers more control. We haven't tested the integration in `kedro-mlflow` of this second workflow extensively, and it should be used with caution.

```yaml
my_custom_model:
    type: kedro_mlflow.io.models.MlflowModelTrackingDataset
    flavor: my_package.custom_mlflow_flavor
    pyfunc_workflow: python_model # or loader_module
```

How can I save model locally and log it in MLflow in one step?

How can I save model locally and log it in MLflow in one step?

If you want to save your model both locally and remotely within the same run, you can leverage `MlflowArtifactDataset`:

```yaml
sklearn_model:
    type: kedro_mlflow.io.artifacts.MlflowArtifactDataset
    dataset:
        type: kedro_mlflow.io.models.MlflowModelLocalFileSystemDataset
        flavor: mlflow.sklearn
        filepath: data/06_models/sklearn_model
```

This might be useful if you want to always read the lastest model saved locally and log it to MLflow each time the new model is being trained for tracking purpose.

1.3.5 Track metrics

What is metric tracking?

MLflow defines a metric as “a (key, value) pair, where the value is numeric”. Each metric can be updated throughout the course of the run (for example, to track how your model’s loss function is converging), and MLflow records and lets you visualize the metric’s full history”.

How to version metrics in a kedro project?

`kedro-mlflow` introduces 3 `AbstractDataset` to manage metrics:

- `MlflowMetricDataset` which can log a float as a metric
- `MlflowMetricHistoryDataset` which can log the evolution over time of a given metric, e.g. a list or a dict of float.
- `MlflowMetricsHistoryDataset`. It is a wrapper around a dictionary with metrics which is returned by node and log metrics in MLflow.

Saving a single float as a metric with `MlflowMetricDataset`

The `MlflowMetricDataset` is an `AbstractDataset` which enable to save or load a `float` as a mlflow metric. You must specify the `key` (i.e. the name to display in mlflow) when creating the dataset. Somes examples follow:

- The most basic usage is to create the dataset and save a a value:

```python
from kedro_mlflow.io.metrics import MlflowMetricDataset

metric_ds = MlflowMetricDataset(key="my_metric")
with mlflow.start_run():
    metric_ds.save(
        0.3
    )  # create a "my_metric=0.3" value in the "metric" field in mlflow UI
```

Warning

Unlike mlflow default behaviour, if there is no active run, no run is created.

- You can also specify a `run_id` instead of logging in the active run:

```
from kedro_mlflow.io.metrics import MlflowMetricDataset

metric_ds = MlflowMetricDataset(key="my_metric", run_id="123456789")
with mlflow.start_run():
    metric_ds.save(
        0.3
    )  # create a "my_metric=0.3" value in the "metric" field of the run 123456789
```

It is also possible to pass `load_args` and `save_args` to control which step should be logged (in case you have logged several step for the same metric.) `save_args` accepts a `mode` key which can be set to `overwrite` (mlflow default) or `append`. In append mode, if no step is specified, saving the metric will "bump" the last existing step to create a linear history. **This is very useful if you have a monitoring pipeline which calculates a metric frequently to check the performance of a deployed model.**

```
from kedro_mlflow.io.metrics import MlflowMetricDataset

metric_ds = MlflowMetricDataset(
    key="my_metric", load_args={"step": 1}, save_args={"mode": "append"}
)

with mlflow.start_run():
    metric_ds.save(0)  # step 0 stored for "my_metric"
    metric_ds.save(0.1)  # step 1 stored for "my_metric"
    metric_ds.save(0.2)  # step 2 stored for "my_metric"

    my_metric = metric_ds.load()  # value=0.1 (step number 1)
```

Since it is an `AbstractDataset`, it can be used with the YAML API in your `catalog.yml`, e.g. :

```
my_model_metric:
    type: kedro_mlflow.io.metrics.MlflowMetricDataset
    run_id: 123456 # OPTIONAL, you should likely let it empty to log in the current run
    key: my_awesome_name # OPTIONAL: if not provided, the dataset name will be sued␣
↪(here "my_model_metric")
    load_args:
        step: ... # OPTIONAL: likely not provided, unless you have a very good reason to␣
↪do so
    save_args:
        step: ... # OPTIONAL: likely not provided, unless you have a very good reason to␣
↪do so
        mode: append #  OPTIONAL: likely better than the default "overwrite". Will be␣
↪ignored if "step" is provided.
```

Saving the evolution of a metric during training with `MlflowMetricHistoryDataset`

The `MlflowMetricDataset` is an `AbstractDataset` which enable to save or load the evolutionf of a metric with various formats. You must specify the `key` (i.e. the name to display in mlflow) when creating the dataset. Somes examples follow:

It enables logging either:

- a list of int as a metric with incremental step, e.g `[0.1,0.2,0.3]` with `mode=list` for either `save_args` or `load_args`

```
from kedro_mlflow.io.metrics import MlflowMetricHistoryDataset

metric_history_ds = MlflowMetricDataset(key="my_metric", save_args={"mode": "list"})

with mlflow.start_run():
    metric_history_ds.save([0.1, 0.2, 0.3])  # will be logged with incremental steps
```

- a dict of {step: value} as a metric:

```
from kedro_mlflow.io.metrics import MlflowMetricHistoryDataset

metric_history_ds = MlflowMetricDataset(key="my_metric", save_args={"mode": "dict"})

with mlflow.start_run():
    metric_history_ds.save(
        {0: 0.1, 1: 0.2, 2: 0.3}
    )  # will be logged with incremental steps
```

- a list of dict [{log_metric_arg: value}] as a metric, e.g:

```
from kedro_mlflow.io.metrics import MlflowMetricHistoryDataset

metric_history_ds = MlflowMetricDataset(key="my_metric", save_args={"mode": "history"})

with mlflow.start_run():
    metric_history_ds.save(
        [
            {"step": 0, "value": 0.1, "timestamp": 1345545},
            {"step": 1, "value": 0.2, "timestamp": 1345546},
            {"step": 2, "value": 0.3, "timestamp": 1345547},
        ]
    )
```

You can combine the different mode for save and load, e.g:

```
from kedro_mlflow.io.metrics import MlflowMetricHistoryDataset

metric_history_ds = MlflowMetricDataset(
    key="my_metric", save_args={"mode": "dict"}, save_args={"mode": "list"}
)

with mlflow.start_run():
    metric_history_ds.save(
        {0: 0.1, 1: 0.2, 2: 0.3}
```

(continues on next page)

(continued from previous page)

```
    )  # will be logged with incremental steps
metric_history_ds.load()  # return [0.1,0.2,0.3]
```

As usual, since it is an `AbstractDataset`, it can be used with the YAML API in your `catalog.yml`, and in this case, the `key` argument is optional:

```yaml
my_model_metric:
    type: kedro_mlflow.io.metrics.MlflowMetricHistoryDataset
    run_id: 123456 # OPTIONAL, you should likely let it empty to log in the current run
    key: my_awesome_name # OPTIONAL: if not provided, the dataset name will be used␣
↪(here "my_model_metric")
    load_args:
        mode: ... # OPTIONAL: "list" by default, one of {"list", "dict", "history"}
    save_args:
        mode: ... # OPTIONAL: "list" by default, one of {"list", "dict", "history"}
```

Saving several metrics with their entire history with `MlflowMetricsHistoryDataset`

Since it is an `AbstractDataset`, it can be used with the YAML API. You can define it in your `catalog.yml` as:

```yaml
my_model_metrics:
    type: kedro_mlflow.io.metrics.MlflowMetricsHistoryDataset
```

You can provide a prefix key, which is useful in situations like when you have multiple nodes producing metrics with the same names which you want to distinguish. If you are using the `v`, it will handle that automatically for you by giving as prefix metrics data set name. In the example above the prefix would be `my_model_metrics`.

Let's look at an example with custom prefix:

```yaml
my_model_metrics:
    type: kedro_mlflow.io.metrics.MlflowMetricsHistoryDataset
    prefix: foo
```

How to return metrics from a node?

Let assume that you have node which doesn't have any inputs and returns dictionary with metrics to log:

```python
def metrics_node() -> dict[str, Union[float, list[float]]]:
    return {
        "metric1": {"value": 1.1, "step": 1},
        "metric2": [{"value": 1.1, "step": 1}, {"value": 1.2, "step": 2}],
    }
```

As you can see above, `kedro_mlflow.io.metrics.MlflowMetricsHistoryDataset` can take metrics as:

- `[str, key]`
- `list[[str, key]]`

To store metrics we need to define metrics dataset in Kedro Catalog:

```yaml
my_model_metrics:
    type: kedro_mlflow.io.metrics.MlflowMetricsHistoryDataset
```

Within a kedro run, the MlflowHook will automatically prefix the metrics datasets with their name in the catalog. In our example, the metrics will be stored in Mlflow with the following keys: my_model_metrics.metric1, my_model_metrics.metric2.

It is also prossible to provide a prefix manually:

```
my_model_metrics:
    type: kedro_mlflow.io.metrics.MlflowMetricsHistoryDataset
    prefix: foo
```

which would result in metrics logged as foo.metric1 and foo.metric2.

As any entry in the catalog, the metrics data set must be defined in a Kedro pipeline:

```
def create_pipeline() -> Pipeline:
    return Pipeline(
        node(
            func=metrics_node,
            inputs=None,
            outputs="my_model_metrics",
            name="log_metrics",
        )
    )
```

1.3.6 Open the mlflow UI

The mlflow user interface

Mlflow offers a user interface (UI) that enable to browse the run history.

The kedro-mlflow helper

When you use a local storage for kedro mlflow, you can call a mlflow cli command to launch the UI if you do not have a mlflow tracking server configured.

To ensure this UI is linked to the tracking uri specified configuration, kedro-mlflow offers the following command:

```
kedro mlflow ui
```

which is a wrapper for kedro ui command with the tracking uri (as well as the port and host) specified the mlflow.yml file.

Opens http://localhost:5000 in your browser to see the UI after calling previous command. If your mlflow_tracking_uri is a http[s] URL, the command will automatically open it.

1.3.7 How to use kedro-mlflow in a notebook

Important

You need to install ipython to access notebook functionalities.

Reminder on mlflow's limitations with interactive use

Data science project lifecycle are very iterative. Mlflow intends to track parameters changes to improve reproducibility. However, one must be conscious that being able to **execute functions outside of a end to end pipeline** puts a strong burden on the user shoulders **because he is in charge to make the code execution coherent** by running the notebooks

cells in the right order. Any back and forth during execution to change some parameters in a previous notebook cells and then retrain a model creates an operational risk that the recorded parameter stored in mlflow is different than the real parameter used for training the model.

To make a long story short: **forget about efficient reproducibility** when using mlflow interactively.

It may **still be useful to track some experiments results** especially if they are long to run and vary wildly with parameters, e.g. if you are performing hyperparameter tuning.

These limitations are inherent to the data science process, not to mlflow itself or the plugin.

Setup mlflow configuration in your notebook

Open your notebook / ipython session with the Kedro CLI:

```
kedro jupyter notebook
```

Or if you are on JupyterLab,

```
%load_ext kedro.ipython
```

Kedro creates a bunch of global variables, including a `session`, a `context` and a `catalog` which are automatically accessible.

When the context was created, `kedro-mlflow` automatically:

- loaded and setup (create the tracking uri, export credentials…) the mlflow configuration of your `mlflow.yml`
- import `mlflow` which is now accessible in your notebook

If you change your `mlflow.yml`, reload the kedro extension for the changes to take effect.

Difference with running through the CLI

- The DataSets `load` and `save` methods works as usual. You can call `catalog.save("my_artifact_dataset", data)` inside a cell, and your data will be logged in mlflow properly (assuming "my_artifact_dataset" is a `kedro_mlflow.io.MlflowArtifactDataset`).
- The `hooks` which automatically save all parameters/metrics/artifacts in mlflow will work if you run the session interactively, e.g.:

```
session.run(
    pipeline_name="my_ml_pipeline",
    tags="training",
    from_inputs="data_2",
    to_outputs="data_7",
)
```

but it is not very likely in a notebook.

- if you need to interact manually with the mlflow server, you can use `context.mlflow.server._mlflow_client`.

Guidelines and best practices suggestions

During experimentation phase, you will likely not run entire pipelines (or sub pipelines filtered out between some inputs and outputs). Hence, you cannot benefit from Kedro's `hooks` (and hence from `kedro-mlflow` tracking). From this moment on, perfect reproducbility is impossible to achieve: there is no chance that you manage to maintain a perfectly linear workflow, as you will go back and forth modifying parameters and code to create your model.

I suggest to :

- **focus on versioning parameters and metrics**. The goal is to finetune your hyperparameters and to be able to remember later the best setup. It is not very important to this stage to version all parameters (e.g. preprocessing ones) nor models (after all you will need an entire pipeline to predict and it is very unlikely that you will need to reuse these experiment models one day.) It may be interesting to use `mlflow.autolog()` feature to have a easy basic setup.
- **transition quickly to kedro pipelines**. For instance, when you preprocessing is roughly defined, try to put it in kedro pipelines. You can then use notebooks to experiment / perfom hyperparameter tuning while keeping preprocessing "fixed" to enhance reproducibility. You can run this pipeline interactively with :

```
result = session.run(
    pipeline_name="my_preprocessing_pipeline",
    tags="training",
    from_inputs="data_2",
    to_outputs="data_7",
)
```

`result` is a python `dict` with the outputs of your pipeline (e.g. a "preprocessed_data" `pandas.DataFrame`), and you can use it interactively in your notebook.

1.4 Pipeline as model

1.4.1 Introduction to mlflow models

What are Mlflow Models ?

Mlflow Models are a standardised agnostic format to store machine learning models. They intend to be standalone to be as portable as possible to be deployed virtually anywhere and mlflow provides built-in CLI commands to deploy a mlflow model to most common cloud platforms or to create an API.

A Mlflow Model is composed of:

- a `MLModel` file which is a configuration file to indicate to mlflow how to load the model. This file may also contain the `Signature` of the model (i.e. the `Schema` of the input and output of your model, including the columns names and order) as well as example data.
- a `conda.yml` file which contains the specifications of the virtual conda environment inside which the model should run. It contains the packages versions necessary for your model to be executed.
- a `model.pkl` (or a `python_function.pkl` for custom model) file containing the trained model.
- an `artifacts` folder containing all other data necessary to execute the models

> **Important**
>
> Mlflow enable to create **custom models "flavors" to convert any object to a Mlflow Model** provided we have these informations. Inside a Kedro project, the `Pipeline` and `DataCatalog` objects contain all these informations. As a consequence, it is easy to create a custom model to convert entire Kedro `Pipelines` to mlflow models, and it the purpose of `pipeline_ml_factory` and `KedroPipelineModel` that we will present in the following sections.

Pre-requisite for converting a pipeline to a mlflow model

You can log any Kedro `Pipeline` matching the following requirements:

- one of its input must be a `pandas.DataFrame`, a `spark.DataFrame` or a `numpy.array`. This is the **input which contains the data to predict on**. This can be any Kedro `AbstractDataset` which loads data in one of the previous three formats. It can also be a `MemoryDataset` and not be persisted in the `catalog.yml`.

- all its other inputs must be persisted on disk (e.g. if the machine learning model must already be trained and saved so we can export it) or declared as "parameters" in the model `Signature`.

Warning

If the pipeline has parameters :

- For `mlflow<2.7.0` the parameters need to be persisted before exporting the model, which implies that you will not be able to modify them at runtime. This is a limitation of `mlflow<2.6.0`
- For `mlflow>=2.7.0` , they can be declared in the signature and modified at runtime. See https://github.com/Galileo-Galilei/kedro-mlflow/issues/445 for more information.

1.4.2 Scikit-learn like Kedro pipelines - Automatically log the inference pipeline after training

For consistency, you may want to **log an inference pipeline** (including some data preprocessing and prediction post processing) **automatically after you ran a training pipeline**, with all the artifacts generated during training (the new model, encoders, vectorizers…).

Hint

You can think of `pipeline_ml_factory` as "**scikit-learn like pipeline in kedro**". Running `kedro run -p training` performs the scikit-learn's `pipeline.fit()` operation, storing all components (e.g. a model) we need to reuse further as mlflow artifacts and the inference pipeline as code. Hence, you can later use this mlflow model which will perform the scikit-learn's `pipeline.predict(new_data)` operation by running the entire kedro inference pipeline.

Getting started with pipeline_ml_factory

Note

Below code assume that for inference, you want to skip some nodes that are training specific, e.g. you don't want to train the model, you just want to predict with it ; you don't want to fit and transform with you encoder, but only transform. Make sure these 2 steps ("train" and "predict", or "fit and "transform") are separated in 2 diffent nodes in your pipeline, so you can skip the train / transform step at inference time.

You can configure your project as follows:

1. Install `kedro-mlflow MlflowHook` (this is done automatically if you have installed `kedro-mlflow` in a `kedro>=0.16.5` project)
2. Turn your training pipeline in a `PipelineML` object with `pipeline_ml_factory` function in your `pipeline_registry.py`:

```python
# pipeline_registry.py for kedro>=0.17.2 (hooks.py for ``kedro>=0.16.5, <0.17.2)

from kedro_mlflow_tutorial.pipelines.ml_app.pipeline import create_ml_pipeline


def register_pipelines(self) -> [str, Pipeline]:
    ml_pipeline = create_ml_pipeline()
```

(continues on next page)

(continued from previous page)

```
    training_pipeline_ml = pipeline_ml_factory(
        training=ml_pipeline.only_nodes_with_tags(
            "training"
        ),  # nodes : encode_labels + preprocess + train_model + predict +␣
↪postprocess + evaluate
        inference=ml_pipeline.only_nodes_with_tags(
            "inference"
        ),  # nodes : preprocess + predict + postprocess
        input_name="instances",
        log_model_kwargs=dict(
            artifact_path="kedro_mlflow_tutorial",
            conda_env={
                "python": 3.10,
                "dependencies": [f"kedro_mlflow_tutorial=={PROJECT_VERSION}"],
            },
            signature="auto",
        ),
    )

    return {"training": training_pipeline_ml}
```

3. Persist all your artifacts locally in the `catalog.yml`

```yaml
label_encoder:
type: pickle.PickleDataset  # <- This must be any Kedro Dataset other than
↪"MemoryDataset"
filepath: data/06_models/label_encoder.pkl  # <- This must be a local path, no␣
↪matter what is your mlflow storage (S3 or other)
```

   and as well for your model if necessary.

4. Launch your training pipeline:

```
kedro run --pipeline=training
```

   The inference pipeline will *automagically* be logged as a custom mlflow model (a `KedroPipelineModel`) **at the end of the training pipeline!**.

5. Go to the UI, retrieve the run id of your "inference pipeline" model and use it as you want, e.g. in the `catalog.yml`:

```yaml
# catalog.yml

pipeline_inference_model:
type: kedro_mlflow.io.models.MlflowModelTrackingDataset
flavor: mlflow.pyfunc
pyfunc_workflow: python_model
artifact_path: kedro_mlflow_tutorial  # the name of your mlflow folder = the model_
↪name in pipeline_ml_factory
run_id: <your-run-id>
```

   Now you can run the entire inference pipeline inside a node as part of another pipeline.

Advanced configuration for pipeline_ml_factory

Register the model as a new version in the mlflow registry

The `log_model_kwargs` argument is passed to the underlying mlflow.pyfunc.log_model. Specifically, it accepts a `registered_model_name` argument :

```
pipeline_ml_factory(
    training=ml_pipeline.only_nodes_with_tags("training"),
    inference=ml_pipeline.only_nodes_with_tags("inference"),
    input_name="instances",
    log_model_kwargs=dict(
        artifact_path="kedro_mlflow_tutorial",
        registered_model_name="my_inference_pipeline",  # a new version of "my_infernce_pipeline" model will be registered each time you run the "training" pipeline
        conda_env={
            "python": 3.10,
            "dependencies": [f"kedro_mlflow_tutorial=={PROJECT_VERSION}"],
        },
        signature="auto",
    ),
)
```

Complete step by step demo project with code

A step by step tutorial with code is available in the kedro-mlflow-tutorial repository on github.

You have also other resources to understand the rationale:

- an explanation of the `PipelineML` class in the python objects section
- detailed explanations on this issue and this discussion.

1.4.3 Deployment patterns for kedro pipelines as model

A step by step tutorial with code is available in the kedro-mlflow-tutorial repository on github which explains how to serve the pipeline as an API or a batch.

Deploying a KedroPipelineModel

Reuse from a python script

Note

See tutorial: https://github.com/Galileo-Galilei/kedro-mlflow-tutorial?tab=readme-ov-file#scenario-1-reuse-from-a-python-script

If you want to load and predict with your model from python, the `load_model` function of mlflow is what you need:

```
PROJECT_PATH = r"<your/project/path>"
RUN_ID = "<your-run-id>"

from kedro.framework.startup import bootstrap_project
from kedro.framework.session import KedroSession
from mlflow.pyfunc import load_model
```

(continues on next page)

(continued from previous page)

```
bootstrap_project(PROJECT_PATH)
session = Kedrosession.create(
    session_id=1,
    project_path=PROJECT_PATH,
    package_name="kedro_mlflow_tutorial",
)
local_context = session.load_context()  # setup mlflow config

instances = local_context.io.load("instances")
model = load_model(f"runs:/{RUN_ID}/kedro_mlflow_tutorial")

predictions = model.predict(
    instances
)  # runs ``session.run(pipeline=inference)`` with the artifacts created ruing training.␣
↪You should see the kedro logs.
```

The predictions object is a pandas.DataFrame and can be handled as usual.

Reuse in a kedro pipeline

Note

See tutorial: https://github.com/Galileo-Galilei/kedro-mlflow-tutorial?tab=readme-ov-file#scenario-2-reuse-in-a-kedro-pipeline

Say that you want to reuse this trained model in a kedro Pipeline, like the user_app. The easiest way to do it is to add the model in the catalog.yml file

```
pipeline_inference_model:
  type: kedro_mlflow.io.models.MlflowModelLoggerDataSet
  flavor: mlflow.pyfunc
  pyfunc_workflow: python_model
  artifact_path: kedro_mlflow_tutorial  # the name of your mlflow folder = the model_
↪name in pipeline_ml_factory
  run_id: <your-run-id>  # put it in globals.yml to help people find out what to modify
```

Then you can reuse it in a node to predict with this model which is the entire inference pipeline at the time you launched the training.

```
# nodes.py
def predict_from_model(model, data):
    return model.predict(data)


# pipeline.py
def create_pipeline():
    return pipeline(
        [
            node(
                func=predict_from_model,
```

(continues on next page)

(continued from previous page)

```
                inputs={"model": pipeline_inference_model, "data": "validation_data"},
            )
        ]
    )
```

Serve the model with mlflow

Note

See tutorial: https://github.com/Galileo-Galilei/kedro-mlflow-tutorial?tab=readme-ov-file#scenario-3-serve-the-model-with-mlflow

Mlflow provide helpers to serve the model as an API with one line of code:

```
mlflow models serve -m "runs:/<your-model-run-id>/kedro_mlflow_tutorial"
```

This will serve your model as an API (beware: there are known issues on windows). You can test it with: `curl -d "{\"columns\":[\"text\"],\"index\":[0,1],\"data\":[[\"This movie is cool\"],[\"awful film\"]]}" -H "Content-Type: application/json" localhost:5000/invocations`

Frequently asked questions

How can I pass parameters at runtime to a `KedroPipelineModel`?

Since `kedro-mlflow>0.14.0`, you can pass parameters when predicting with a `KedroPipelineModel` object.

We assume you've trained a model with `pipeline_factory_function`. First, load the model, e.g. through the catalog or as described in the previous section:

```
# catalog.yml
pipeline_inference_model:
    type: kedro_mlflow.io.models.MlflowModelTrackingDataset
    flavor: mlflow.pyfunc
    pyfunc_workflow: python_model
    artifact_path: kedro_mlflow_tutorial  # the name of your mlflow folder = the model_
↪name in pipeline_ml_factory
    run_id: <your-run-id>
```

Then, pass params as a dict under the `params` argument of the `predict` method:

```
catalog.load("pipeline_inference_model")  # You can also load it in a node "as usual"
predictions = model.predict(input_data, params={"my_param": "<my_param_value>"})
```

Warning

This will only work if `my_param` is a parameter (i.e. prefixed with `params:`) of the inference pipeline.

Tip

Available params are visible in the model signature in the UI

How can I change the runner at runtime when predicting with a `KedroPipelineModel`?

Assuming the syntax of previous section, a special key in "params" is reserved for the kedro runner:

```
catalog.load("pipeline_inference_model")
predictions = model.predict(
    input_data, params={"my_param": "<my_param_value>", "runner": "ThreadRunner"}
)
```

> **Tip**
>
> You can pass any kedro runner, or even a custom runner by using the path to the module: `params={"runner": "my_package.my_module.MyRunner"}`

1.4.4 Custom registering of a `KedroPipelineModel`

> **Warning**
>
> The goal of this section is to give tool to machine learning engineer or platform engineer to reuse the objects and customize the workflow. This is specially useful in case you need high customisation or fine grained control of the kedro objects or the mlflow model attributes. This is **very unlikely you need this section** if you are using a kedro project "in the standard way" as a data scientist, in which case you should refer to the section scikit-learn like pipeline in kedro.

Log a pipeline to mlflow programatically with `KedroPipelineModel` custom mlflow model

> **Hint**
>
> When using the `KedroPipelineModel` programatically, we focus only on the `inference` pipeline. We assume That you already ran the `training` pipeline previously, and that you now want to log the `inference` pipeline in mlflow manually by retrieveing all the needed objects to create the custom model.

`kedro-mlflow` has a `KedroPipelineModel` class (which inherits from `mlflow.pyfunc.PythonModel`) which can turn any kedro `Pipeline` object to a Mlflow Model.

To convert a `Pipeline` to a mlflow model, you need to create a `KedroPipelineModel` and then log it to mlflow. An example is given in below snippet:

```
from pathlib import Path
from kedro.framework.session import KedroSession
from kedro.framework.startup import bootstrap_project

bootstrap_project(r"<path/to/project>")
session = KedroSession.create(project_path=r"<path/to/project>")

# "pipeline" is the Pipeline object you want to convert to a mlflow model

context = session.load_context()  # this setups mlflow configuration
catalog = context.catalog
pipeline = context.pipelines["<my-pipeline>"]
```

(continues on next page)

(continued from previous page)

```
input_name = "instances"


# artifacts are all the inputs of the inference pipelines that are persisted in the␣
↪catalog

# (optional) get the schema of the input dataset
input_data = catalog.load(input_name)
model_signature = infer_signature(
    model_input=input_data
)  # if you want to pass parameters in "predict", you should specify them in the␣
↪signature

# you can optionally pass other arguments, like the "copy_mode" to be used for each␣
↪dataset
kedro_pipeline_model = KedroPipelineModel(
    pipeline=pipeline, catalog=catalog, input_name=input_name
)

artifacts = kedro_pipeline_model.extract_pipeline_artifacts()

mlflow.pyfunc.log_model(
    artifact_path="model",
    python_model=kedro_pipeline_model,
    artifacts=artifacts,
    conda_env={"python": "3.10.0", dependencies: ["kedro==0.18.11"]},
    model_signature=model_signature,
)
```

Important

Note that you need to provide the `log_model` function a bunch of non trivial-to-retrieve informations (the conda environment, the "artifacts" i.e. the persisted data you need to reuse like tokenizers / ml models / encoders, the model signature i.e. the columns names and types and the predict parameters…). The `KedroPipelineModel` object has methods like `extract_pipeline_artifacts` to help you, but it needs some work on your side.

Note

Saving Kedro pipelines as Mlflow Model objects is convenient and enable pipeline serving. However, it does not does not solve the decorrelation between training and inference: each time one triggers a training pipeline, (s)he must think to save it immediately afterwards. `kedro-mlflow` offers a convenient API through hooks to simplify this workflow, as described in the section scikit-learn like pipeline in kedro .

Log a pipeline to mlflow with the CLI

Note

This command is mainly a helper to relog a model manually without retraining (e.g. because you slighlty modify

the preprocessing or post processing and don't want to train again.)

Warning

We **assume that you already ran the training pipeline previously**, which created persisted artifacts. Now you want to trigger logging the `inference` pipeline in mlflow trhough the CLI. This is dangerous because the commmand does not check that your pipeline is working correctly or that the perssited model has not been modified.

You can log a Kedro `Pipeline` to mlflow as a custom model through the CLI with `modelify` command:

```
kedro mlflow modelify --pipeline=<your-inference-pipeline> --input-name <name-in-catalog-
→of-input-data>
```

This command will create a new run with an artifact named `model` and persist it the code fo your pipeline and all its inputs as artifacts (hence they should have been created *before* running this command, e.g. the model should already be persisted on the disk). Open the user interface with `kedro mlflow ui` to check the result. You can also:

- specify the run id in which you want to log the pipeline with the `--run-id` argument, and its name with the `--run-name` argument.
- pass almost all arguments accepted by `mlflow.pyfunc.log_model`, see the list of all accepted arguments in the API documentation

1.4.5 Why we need a mlops framework to manage machine learning development lifecycle

Machine learning deployment is hard because it comes with a lot of constraints and no adequate tooling

Identifying the challenges to address when deploying machine learning

It is a very common pattern to hear that "machine learning deployment is hard", and this is supposed to explain why so many firms do not achieve to insert ML models in their IT systems (and consequently, not make money despite consequent investments in ML).

On the other hand, you can find thousands of tutorial across the web to explain how to deploy a ML API in 5 min, either locally or on the cloud. There is also a large amount of training sessions which can teach you "how to become a machine learning engineer in 3 months".

Who is right then? Both!

Actually, there is a confusion on what "deployment" means, especially in big enterprises that are not "tech native" or for newbies in ML world. Serving a model over an API pattern is a start, but you often need to ensure (at least) the following properties for your system:

- **scalability and cost control**: in many cases, you need to be able to deal with a lot of (possibly concurrent) requests (likely much more than during training phase). It may be hard to ensure that the app will be able to deal with such an important amount of data. Another issue is that ML oftens needs specific infrastrusture (e.g., GPU's) which are very expensive. Since the request against the model often vary wildly over time, it may be important to adapt the infrastructure in real time to avoid a lot of infrastructure costs
- **speed**: A lot of recent *state of the art* (SOTA) deep learning / ensemble models are computationally heavy and this may hurt inference speed, which is critical for some systems.

- **availability and resilience**: Machine learning systems are more complex than traditional softwares because they have more moving parts (i.e. data and parameters). This increases the risk of errors, and since ML systems are used for critical systems making both the infrastructure and the code robust is key.
- **portability / ease of integration with external components**: ML models are not intended to be directly used by the end users, but rather be consumed by another part of your system (e.g a call to an API). To speed up deployment, your model must be easy to be consumed, i.e. *as self contained as possible*. As a consequence, you must **deploy a ML pipeline which hanbles business objects instead of only a ML model**. If the other part which consumes your API needs to make a lot of data preprocessing *before* using your model, it makes it:
  - very risky to use, because preprocessing and model are decoupled: any change in your model must be reflected in this other data pipeline and there is a huge mismatch risk when redeploying
  - slow and costful to deploy because each deployment of your model needs some new development on the client side
  - poorly reusable beacuse each new app who wants to use the model needs some specific development on its own
- **reproducibility**: the ML development is very iterative by nature. You often try a simple baseline model and iterate (sometimes directly by discussing with the final user) to finally get the model that suits the most your needs (wich is the balance between speed / accuracy / interpretability / maintenance costs / inference costs / data availability / labelling costs…). Looping through these iterations, it is very easy to forget what was the best model. You should not rely only on your memory to compare your experiments. Moreover, many countries have regulatory constraints to avoid discriminations (e.g. GDPR in the EU), and your must be able to justify how your model was built and to that extent reproducibility is key. It is also essential when you redeploy a model that turns out to be worse than the old one and you have to rollback fast.
- **monitoring and ease of redeployment**: It is well known that model quality decay over time (sometimes quickly!), and if you use ML for a critical activity, you will have to retrain your model periodically to maintain its performance. It is critical to be able to redeploy easily your model. This implies that retraining (or even redeployment of a new model) must be as **automated as possible** to ensure deployment speed and model quality.

An additional problem that happens in real world is that it is sometimes poorly understood by end users that the ML model is not standalone. It implies **external developments** from the client side (at least to call the model, sometimes to adapt a data pipeline or to change the user interface to add the ML functionality) and they have hard times to understand the entire costs of a ML project as well as their responsibilities in the project. This is not really a problem of ML but rather on how to give a minimal culture on ML to business end users.

A comparison between traditional software development and machine learning projects

ML and traditional software have different moving parts

A traditional software project contains several moving parts:

- code
- environment (packages versions…)
- infrastructure (build on Windows, deploy on linux)

Since it is a more mature industry, efficient tools exists to manage these items (Git, pip/conda, infra as code…). On the other hand, a ML project has additional moving parts:

- parameters
- data

As machine learning is a much less mature field, efficient tooling to adress these items are very recent and not completely standardized yet (e.g. `Mlflow` to track parameters, `DVC` to version data, `great-expectations` to ensure data quality checks along your pipelines, `tensorboard` to monitor your model metrics…)

> **Mlflow is one of the most mature tool to manage these new moving parts.**

ML and traditional software have different development lifecycles

In traditional software, the development workflow is roughly the following:

- you create a git branch
- you develop your new feature
- you add tests and ensure there are no regression
- you merge the branch on the main branch to add your feature to the codebase

This is a **linear** development process based on **determinist** behaviour of the code.

However in ML development, the workflow is much more iterative and may looklike this:

- you create a notebook
- you make some exploration on the data with some descriptive analysis and a baseline model
- you switch to a git branch and python scripts once the model is quite stable
- you retrain the model, eventually do some parameter tuning
- you merge your code on the main branch to share your work

If you need to modify the model later (do a different preprocessing, change the model type…), you do not **add** code to the codebase, you **modify** the existing code. This makes unit testing much harder because the desired features change over time.

The other difficulty when testing machine learning applications is it hard to test for regression, since the model depends on underlying data and the chosen metrics. If a new model performs slightly better or worse than the previous one ont the same dataset, it may be due to randomness and not to code quality. If the metric varies on a different dataset, it is even harder to know if it is due to code quality or to innate randomness.

This is a **cyclic** development process based on a **stochastic** behaviour of the code.

> **Kedro is a very new tool and cannot be called "mature" at this stage but tries to solve this development lifecyle with a very fluent API to create and modify machine learning pipelines.**

Deployment issues addressed by `kedro-mlflow` and their solutions

Out of scope

We will focus on machine learning *development* lifecycle. As a consequence, these items are out of scope:

- Orchestration
- Issues related to infrastructure (related, but not limited to, the 3 first items of above list: scalability and cost control, inference speed, disponibility and resilience)
- Issues related to model monitoring: Data distribution changes over time, model decay, data access…

Issue 1: The training process is poorly reproducible

The main reason which explains why training is hard to reproduce is the iterative process. Data scientists launch several times the same run with slightly different parameters / data / preprocessing. If they rely on their memory to compare these runs, they will likely struggle to remember what was the best one.

> **`kedro-mlflow` offers automatic parameters versioning** when a pipeline is ran to easily link a model to its training parameters.

Note that there is also a lot of "innate" randomness in ML pipelines and if a seed is not set explictly as a parameter , the run will likely not be reproducible (separation train/test/validation, moving underlying data sources, random initialisation for optimizers, random split for bootstrap. . .).

Issue 2: The data scientist and stakeholders focus on training

While building the ML model, the inference pipeline is often completely ignored by the data scientist. The best example are Kaggle competitions where a very common workflow is the following:

- merge the training and the test data at the beginning of their script
- do the preprocessing on the entire dataset
- resplit just before training the model
- train the model on training data
- predict on test data
- anayze their metrics, finetune their hyper parameters
- submit their predictions as data (i.e. as a file) to Kaggle

The very important issue which arises with such a workflow is that **you completely ignore the non reproducibility which arises from the preprocessing (encoding, randomness. . .)**. Most Kaggle solutions are never tested on an end to end basis, i.e. by running the inference pipeline from the test data input file to the predictions file. This facilitates very bad coding practices and teaches beginner data scientists bad software engineering practice.

> `kedro-mlflow` enables to log the inference pipeline as a Mlflow Model (through a `KedroPipelineModel` class) to ensure that you deploy the inference pipeline as a whole.

Issue 3: Inference and training are entirely decoupled

As stated previous paragraph, the inference pipeline is not a primary concern when experimenting and developing your model. This raises strong reproducibility issues. Assume that you have logged the model and all its parameters when training (which is a good point!), you will still need to retrieve the code used during training to create the inference pipeline. This is in my experience quite difficult:

- in the best case, you have trained the model from a git sha which is logged in mlflow. Any potential user can (but it takes time) recreate the exact inference pipeline from your source code, and retrieve all necessary artifacts from mlflow. This is tedious, error prone, and gives a lot of responsibility and work to your end user, but at least it makes your model usable.
- most likely, you did not train your model from a version control commit. While experimenting /debug, it is very common to modify the code and retrain without committing. The exact code associated to a given model will likely be impossible to find out later.

> `kedro-mlflow` offers a `PipelineML` (and its helper `pipeline_ml_factory`) class which binds the `training` and `inference` pipeline (similarly to `scikit-learn` `Pipeline` object), and a hook which autolog such pipelines when they are run. This enables data scientists to ensure that each training model is logged with its associated inference pipeline, and is ready to use for any end user. This decreases a lot the necessary cognitive complexity to ensure coherence between training and inference.

Issue 4: Data scientists do not handle business objects

It is often said that data scientists deliver machine learning *models*. This assumes that all the preprocessing will be recoded the end user of your model. This is a major cause of poor adoption of your model in an enterprise setup because it makes your model:

- hard to use (developments are need on the client side)

- hard to update (it needs code update from the end user)
- very error prone (never trust the client!)

If you struggle representing it, imagine that you have developed a NLP model. Would you really ask your end user to give you a one-hot encoded matrix or BERT-tokenized texts with your custom embbeddings and vocabulary?

Your model must handle business objects (e.g. a mail, a movie review, a customer with its characteristic, a raw image. . .) to be usable.

> Kedro `Pipeline`'s are able to handle processing from the business object to the prediction. Your real model must be a `Pipeline`, and the `KedroPipelineModel` of `kedro-mlflow` helps to store them and log them in mlflow. Additionally, `kedro-mlflow` suggests how your project should be organized in "apps" to make this transition easy.

Overcoming these problems: support an organisational solution with an efficient tool

`kedro-mlflow` assume that we declare a clear contrat of what the output of the data science project is: it is an an inference pipeline. This defines a clear "definition of done" of the data science project: is it ready to deploy?

The downside of such an approach is that it increases data scientist's responsibilities, because s(he) is responsible for his code.

`kedro-mlflow` offers a very convenient way (through the `pipeline_ml_factory` function) to make sure that each experiment will result in creating a compliant "output".

This is very transparent for the data scientist who have no extra constraints (apart from developing in Kedro) to respect this contract. Hence, the data scientist still benefits from the interactivity he needs to work. This is why we want to leverage Kedro which is very flexible and offers a convenient way to transition from notebooks to pipeline, and leverage Mlflow for standardising the definiton of an "output" of a datscience project.

> Enforcing these solutions with a tool: `kedro-mlflow` at the rescue

1.4.6 The components of a machine learning application

Definition: apps of a machine learning projects

A machine learning project is composed of 3 main blocks that I will call "apps" in the rest of the paragraph. These 3 apps are:

- The *etl_app*, which is the application in charge of bringing the data to the machine learning pipeline
- The *ml_app*, which is the application in charge of managing the machine learning model (including training and inference)
- The *user_app* which is the application in charge of consuming the predictions of the machine learning model and doing the actual business logic with it

Difference between an app and a Kedro pipeline

Note that the previously defined "apps" are not pipelines in the Kedro sense. On the contrary, each app likely contain several (Kedro?) pipelines.

The main differences between these apps are:

- Each app development / deployment is likely under the responsibility of different people / teams.
- Each app has a different development lifecyle. It implies that development can be parallelized, and releasing one app to fix a bug does not imply to release the other ones. If your training pipeline is time /resources consuming, you do not want a bugfix in the *user_app* to trigger a retraining of your model, do you?

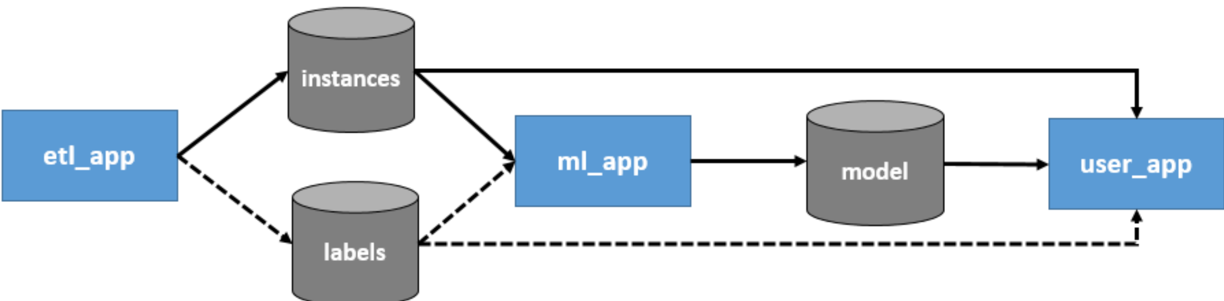
- Each app has its own orchestration timeline. For instance, the data produced by the etl can be stored independently of whether the *user_app* and the *ml_app* consume them "on the fly" or not.
- Each app do not communicate with the other apart from a clear interface: the data schema accepted as inputs/output of each app.

Apps development lifecycle in a machine learning project

The data scientist creates at least part of the 3 apps

Note that there are **as many *etl_app* and *user_app*** as needed for the different use of your model. Since **training the model is a specific use, the data scientist will need one to create its own *etl_app* and *user_app***. These apps will very likely be replaced later by the true business app dedicated to the model use.

We saw that the data scientist has to create some code that will be replaced by other people code when deploying the model. As a consequence, the interactions between these apps must be very clearly defined at the beginning of the project. We claim that it is possible to cover most use case with the following schema:



The *ml_app* takes `instances` (i.e. examples of the business object to handle) as input. This implies that the *ml_app* will include some machine learning-specific preprocessing and not only the model training. It also (optionally) takes labels as inputs if the underlying problem is supervised. Even in this situation, the labels will not be known at inference time so the *etl_app* does not necessarily produce them.

This is a key principle: anyone who wants to consume the model later will need to bring instances of the same business object.

The *etl_app*

The *etl_app* is the one in charge of bringing the data to the *ml_app*. As a consequence, each different *user_app* will likely have to develop its associated *etl_app* to consume the *ml_app*.

From the data scientist point of view, this app will create the training dataset. This app can do very different things:

- send request over an API
- extract from a database (with SQL, SAS…)
- scrape data from a website
- download data from an URL
- read data from disk
- …

For the labels, in addition of above possibility, this app can be a **labelling tool** with human labellers who provide the needed "true reference" as labels.

It is also common to mix several of above approaches to gather different data sources, and to have different Kedro pipelines in this app.

Note that during a training, this app very likely retrieves batch data from a given time period. This will necessarily be different when using the model, because the user often want to use live stream data.

The *ml_app*

This app is the core of the data scientist work. It is at least composed of two kedro pipelines:

- a *training* pipeline, which produces all the artifacts (e.g. any object fitted on data, including obviously the machine learning model itself)
- an *inference* pipeline which takes an instance as input and returns the prediction of the model

It is quite common to have other pipelines depending on the data scientist needs (an *evaluation* pipelines which produces metrics for a given model, an *explanation* pipeline to produce explanation for a specific instance like shap values or importance pixel, . . .).

It is quite common to see data scientists duplicate the code when creating the inference pipeline, because it is written after the training pipeline. **Thanks to kedro tags, it is possible to mark a node to use it in two different pipelines**. Reuse is a key component to improve quality and deployment speed. **Each time a node is created (i.e. a function is called), the data scientist should wonder if it will be used in *training* pipeline only or in both (*training* and *inference*), and tag it accordingly.**

The *user_app*

The *user_app* must not be aware of how the inference pipeline operates under the hood. The *user_app* must either:

- takes a *run_id* from mlflow to retrieve the model from mlflow and predict with it. This is mainly useful for batch predictions.
- call the served model from an API endpoint and only get predictions as inputs. This assumes that the model has been served, which is very easy with mlflow.

After that, the *user_app* can use the predictions and apply any needed business logic to them.

1.4.7 `kedro-mlflow` mlops solution

Reminder

We assume that we want to solve the following challenges among those described in "Why we need a mlops framework" section:

- serve pipelines (which handles business objects) instead of models
- synchronize training and inference by packaging inference pipeline at training time

Enforcing these principles with a dedicated tool

Synchronizing training and inference pipeline

To solve the problem of desynchronization between training and inference, `kedro-mlflow` offers a `PipelineML` class (which subclasses Kedro `Pipeline` class). A `PipelineML` is simply a Kedro standard `Pipeline` (the "training") which has a reference to another `Pipeline` (the "inference"). The two pipelines must share a common input DataSet name, which represents the data you will perform operations on (either train on for the training pipeline, or predict on for the inference pipeline).

This class implements several methods to compare the `DataCatalog`s associated to each of the two binded pipelines and performs subsetting oparations. This makes it quite difficult to handle directly. Fortunately, `kedro-mlflow` provides a convenient API to create `PipelineML` objects: the `pipeline_ml_factory` function.

The use of `pipeline_ml_factory` is very straightforward, especially if you have used the project architecture described previously. The best place to create such an object is your `hooks.py` file which will look like this:

```python
# hooks.py
from kedro_mlflow_tutorial.pipelines.ml_app.pipeline import create_ml_pipeline


class ProjectHooks:
    @hook_impl
    def register_pipelines(self) -> [str, Pipeline]:
        ml_pipeline = create_ml_pipeline()

        # convert your two pipelines to a PipelinML object
        training_pipeline_ml = pipeline_ml_factory(
            training=ml_pipeline.only_nodes_with_tags("training"),
            inference=ml_pipeline.only_nodes_with_tags("inference"),
            input_name="instances",
        )

        return {"__default__": training_pipeline_ml}
```

So, what? We have created a link between our two pipelines, but the gain is not obvious at first glance. The 2 following sections demonstrates that such a construction enables to package and serve automatically the inference pipeline when executing the training one.

Packaging and serving a Kedro Pipeline

Mlflow offers the possibility to create custom model class. Mlflow offers a variety of tool to package/containerize, deploy and serve such models.

`kedro-mlflow` has a `KedroPipelineModel` class (which inherits from `mlflow.pyfunc.PythonModel`) which can turn any kedro `PipelineML` object to a Mlflow Model.

To convert a `PipelineML`, you need to declare it as a `KedroPipelineModel` and then log it to mlflow:

```python
from pathlib import Path
from kedro.framework.context import load_context
from kedro_mlflow.mlflow import KedroPipelineModel
from mlflow.models import ModelSignature

# pipeline_training is your PipelineML object, created as previsously
catalog = load_context(".").io

# artifacts are all the inputs of the inference pipelines that are persisted in the␣
↪catalog
artifacts = pipeline_training.extract_pipeline_artifacts(catalog)

# (optional) get the schema of the input dataset
input_data = catalog.load(pipeline_training.input_name)
model_signature = infer_signature(model_input=input_data)

kedro_model = KedroPipelineModel(pipeline=pipeline_training, catalog=catalog)

mlflow.pyfunc.log_model(
    artifact_path="model",
    python_model=kedro_model,
    artifacts=artifacts,
```

(continues on next page)

(continued from previous page)

```
    conda_env={"python": "3.10.0", dependencies: ["kedro==0.18.11"]},
    signature=model_signature,
)
```

Note that you need to provide the `log_model` function a bunch of non trivial-to-retrieve informations (the conda environment, the "artifacts" i.e. the persisted data you need to reuse like tokenizers / ml models / encoders, the model signature i.e. the columns names and types. . .). The `PipelineML` object has methods like `extract_pipeline_artifacts` to help you, but it needs some work on your side.

> Saving Kedro pipelines as Mlflow Model objects is convenient and enable pipeline serving serving. However, it does not does not solve the decorrelation between training and inference: each time one triggers a training pipeline, (s)he must think to save it immediately afterwards. Good news: triggering operations at some "execution moment" of a Kedro `Pipeline` (like after it finished runnning) is exactly what hooks are designed for!

kedro-mlflow's magic: inference autologging

When running the training pipeline, we have all the desired informations we want to pass to the `KedroPipelineModel` class and `mlflow.pyfunc.log_model` function:

- the artifacts exist in the DataCatalog if they are persisted
- the "instances" dataset is loaded at the beginning of training, thus we can infer its schema (columns names and types)
- the inference and training pipeline codes are retrieved at the same moments, so consistency checks can be performed

Hence, `kedro-mlflow` provides a `MlflowHook.after_pipeline_run` hook which perfoms the following operations:

- check if the pipeline that have ust been run is a `PipelineML` object
- in case it is, create the `KedroPipelineModel` like above and log it to mlflow

  We have achieved perfect synchronicity since the exact inference pipeline (with code, and artifacts) will be logged in mlflow each time the training pipeline is executed. The model is than accessible in the mlflow UI "artifacts" section and can be downloaded, or served as an API with the `mlflow serve` command, or it can be used in the `catalog.yml` with the `MlflowModelTrackingDataset` for further reuse.

Reuse the model in kedro

Say that you an to reuse this inference model as the input of another kedro pipeline (one of the "user_app" application). `kedro-mlflow` provides a `MlflowModelTrackingDataset` class which can be used int the `catalog.yml` file:

```
# catalog.yml

pipeline_inference_model:
  type: kedro_mlflow.io.models.MlflowModelTrackingDataset
  flavor: mlflow.pyfunc
  pyfunc_workflow: python_model
  artifact_path: kedro_mlflow_tutorial  # the name of your mlflow folder = the model_
↪name in pipeline_ml_factory
  run_id: <your-run-id>
```

1.5 API

1.5.1 Datasets

MlflowArtifactDataset

`MlflowArtifactDataset` is a wrapper for any `AbstractDataset` which logs the dataset automatically in mlflow as an artifact when its `save` method is called. It can be used both with the YAML API:

```yaml
my_dataset_to_version:
    type: kedro_mlflow.io.artifacts.MlflowArtifactDataset
    dataset:
        type: pandas.CSVDataset  # or any valid kedro DataSet
        filepath: /path/to/a/local/destination/file.csv
```

or with additional parameters:

```yaml
my_dataset_to_version:
    type: kedro_mlflow.io.artifacts.MlflowArtifactDataset
    dataset:
        type: pandas.CSVDataset  # or any valid kedro DataSet
        filepath: /path/to/a/local/destination/file.csv
        load_args:
            sep: ;
        save_args:
            sep: ;
        # ... any other valid arguments for dataset
    run_id: 13245678910111213  # a valid mlflow run to log in. If None, default to␣
↪active run
    artifact_path: reporting  # relative path where the artifact must be stored. if None,
↪ saved in root folder.
```

or with the python API:

```python
from kedro_mlflow.io.artifacts import MlflowArtifactDataset
from kedro_datasets.pandas import CSVDataset

csv_dataset = MlflowArtifactDataset(
    dataset={"type": CSVDataset, "filepath": r"/path/to/a/local/destination/file.csv"}
)
csv_dataset.save(data=pd.DataFrame({"a": [1, 2], "b": [3, 4]}))
```

Metrics Datasets

MlflowMetricDataset

The `MlflowMetricDataset` is documented here.

MlflowMetricHistoryDataset

The `MlflowMetricHistoryDataset` is documented here.

Models `Datasets`

`MlflowModelTrackingDataset`

The `MlflowModelTrackingDataset` accepts the following arguments:

- flavor (str): Built-in or custom MLflow model flavor module. Must be Python-importable (e.g. `mlflow.sklearn`, `mlflow.pyfunc` ...)
- load_args (dict[str, Any], optional): Arguments to `load_model` function from specified `flavor`, see mlflow documentation (e.g. mlflow.sklearn.load_model) for each flavor. Defaults to None.
- save_args (dict[str, Any], optional): Arguments to `log_model` function from specified `flavor`, see mlflow documentation. Default to None, it is recommended to specify 'name'.
- metadata: Any arbitrary metadata. This is ignored by Kedro, but may be consumed by users or external plugins.

You can either only specify the flavor:

```python
from kedro_mlflow.io.models import MlflowModelTrackingDataset
from sklearn.linear_model import LinearRegression

mlflow_model_tracking = MlflowModelTrackingDataset(
    flavor="mlflow.sklearn", save_args={"name": "my_model"}
)
mlflow_model_tracking.save(LinearRegression())
```

Let assume that this first model has been saved once, and you want to retrieve it (for prediction for instance), you can specify the `model_uri`:

```python
mlflow_model_tracking = MlflowModelTrackingDataset(
    flavor="mlflow.sklearn", load_args={"model_uri": "models:/my_model"}
)
my_linear_regression = mlflow_model_tracking.load()
my_linear_regression.predict(
    data
)  # will obviously fail if you have not fitted your model object first :)
```

You can also specify some logging parameters:

```python
mlflow_model_tracking = MlflowModelTrackingDataset(
    flavor="mlflow.sklearn",
    save_args={
        "conda_env": {"python": "3.11.0", "dependencies": ["kedro==1.0.0"]},
        "input_example": data.iloc[0:5, :],
    },
)
mlflow_model_tracking.save(LinearRegression().fit(data))
```

As always with kedro, you can use it directly in the `catalog.yml` file:

```yaml
my_model:
    type: kedro_mlflow.io.models.MlflowModelTrackingDataset
    flavor: "mlflow.sklearn"
    save_args:
        conda_env:
            python: "3.11.0"
```

(continues on next page)

(continued from previous page)

```
            dependencies:
                - "kedro==1.0.0"
```

`MlflowModelLocalFileSystemDataset`

The `MlflowModelLocalFileSystemDataset` accepts the following arguments:

- flavor (str): Built-in or custom MLflow model flavor module. Must be Python-importable.
- filepath (str): Path to store the dataset locally.
- pyfunc_workflow (str, optional): Either `python_model` or `loader_module`. See mlflow workflows.
- load_args (dict[str, Any], optional): Arguments to `load_model` function from specified `flavor`. Defaults to None.
- save_args (dict[str, Any], optional): Arguments to `save_model` function from specified `flavor`. Defaults to None.
- version (Version, optional): Kedro version to use. Defaults to None.

The use is very similar to `MlflowModelTrackingDataset`, but you have to specify a local `filepath` instead of a `run_id`:

```python
from kedro_mlflow.io.models import MlflowModelTrackingDataset
from sklearn.linear_model import LinearRegression

mlflow_model_tracking = MlflowModelLocalFileSystemDataset(
    flavor="mlflow.sklearn", filepath="path/to/where/you/want/model"
)
mlflow_model_tracking.save(LinearRegression().fit(data))
```

The same arguments are available, plus an additional `version` common to usual `AbstractVersionedDataset`

```python
mlflow_model_tracking = MlflowModelLocalFileSystemDataset(
    flavor="mlflow.sklearn",
    filepath="path/to/where/you/want/model",
    version="<valid-kedro-version>",
)
my_model = mlflow_model_tracking.load()
```

and with the YAML API in the `catalog.yml`:

```yaml
my_model:
    type: kedro_mlflow.io.models.MlflowModelLocalFileSystemDataset
    flavor: mlflow.sklearn
    filepath: path/to/where/you/want/model
    version: <valid-kedro-version>
```

`MlflowModelRegistryDataset`

The `MlflowModelRegistryDataset` accepts the following arguments:

- `model_name` (str): The name of the registered model is the mlflow registry

- `stage_or_version` (str): A valid stage (either "staging" or "production") or version number for the registred model.Default to None,(internally converted to "latest" if no alias si provided) which fetch the last version and the higher "stage" available.
- `alias` (str): A valid alias, which is used instead of stage to filter model since mlflow 2.9.0. Will raise an error if both `stage_or_version` and `alias` are provided.
- `flavor` (str): Built-in or custom MLflow model flavor module. Must be Python-importable.
- `pyfunc_workflow` (str, optional): Either `python_model` or `loader_module`. See mlflow workflows.
- `load_args` (dict[str, Any], optional): Arguments to `load_model` function from specified `flavor`. Defaults to None.

We assume you have registered a mlflow model first, either with the `MlflowClient` or within the mlflow ui, e.g. :

```python
from sklearn.tree import DecisionTreeClassifier

import mlflow
import mlflow.sklearn

with mlflow.start_run():
    model = DecisionTreeClassifier()

    # Log the sklearn model and register as version 1
    mlflow.sklearn.log_model(
        sk_model=model, artifact_path="model", registered_model_name="my_awesome_model"
    )
```

You can fetch the model by its name:

```python
from kedro_mlflow.io.models import MlflowModelRegistryDataset

mlflow_model_tracking = MlflowModelRegistryDataset(model_name="my_awesome_model")
my_model = mlflow_model_tracking.load()
```

and with the YAML API in the `catalog.yml` (only for loading an existing model):

```yaml
my_model:
    type: kedro_mlflow.io.models.MlflowModelRegistryDataset
    model_name: my_awesome_model
```

1.5.2 Hooks

This package provides 1 new hook.

`MlflowHook`

This hook :

1. manages mlflow settings at the beginning and the end of the run (run start / end).
2. autolog nodes parameters each time the pipeline is run (with `kedro run` or programatically).
3. log useful informations for reproducibility as `mlflow tags` (including kedro `Journal` information for old kedro versions and the commands used to launch the run).
4. register the pipeline as a valid `mlflow model` if it is a `PipelineML` instance

1.5.3 Pipelines

PipelineML and pipeline_ml_factory

PipelineML is a new class which extends Pipeline and enable to bind two pipelines (one of training, one of inference) together. This class comes with a KedroPipelineModel class for logging it in mlflow. A pipeline logged as a mlflow model can be served using mlflow models serve and mlflow models predict command.

The PipelineML class is not intended to be used directly. A pipeline_ml_factory factory is provided for user friendly interface.

Example within kedro template:

```
# in src/PYTHON_PACKAGE/pipeline.py

from PYTHON_PACKAGE.pipelines import data_science as ds


def create_pipelines(**kwargs) -> dict[str, Pipeline]:
    data_science_pipeline = ds.create_pipeline()
    training_pipeline = pipeline_ml_factory(
        training=data_science_pipeline.only_nodes_with_tags(
            "training"
        ),  # or whatever your logic is for filtering
        inference=data_science_pipeline.only_nodes_with_tags("inference"),
    )

    return {
        "ds": data_science_pipeline,
        "training": training_pipeline,
        "__default__": data_engineering_pipeline + data_science_pipeline,
    }
```

Now each time you will run kedro run --pipeline=training (provided you registered MlflowHook in you run.py), the full inference pipeline will be registered as a mlflow model (with all the outputs produced by training as artifacts : the machine learning model, but also the *scaler*, *vectorizer*, *imputer*, or whatever object fitted on data you create in training and that is used in inference).

Note that:

- the inference pipeline input_name can be a MemoryDataset and it belongs to inference pipeline inputs
- Apart form input_name, all other inference pipeline inputs must be persisted locally on disk (i.e. it must not be MemoryDataset and must have a local filepath)
- the inference pipeline inputs must belong to training outputs (vectorizer, binarizer, machine learning model...)
- the inference pipeline must have one and only one output

Caution

PipelineML objects do not implement all filtering methods of a regular Pipeline, and you cannot add or substract 2 PipelineML together. The rationale is that a filtered PipelineML is not a PipelineML in general, because the filtering is not consistent between training and inference. You can see the ones which are supported in the code.

You can also directly log a PipelineML object in mlflow programatically:

```python
from pathlib import Path
from kedro.framework.context import load_context
from kedro_mlflow.mlflow import KedroPipelineModel
from mlflow.models import ModelSignature

# pipeline_training is your PipelineML object, created as previsously
catalog = load_context(".").io

# artifacts are all the inputs of the inference pipelines that are persisted in the␣
↪catalog
artifacts = pipeline_training.extract_pipeline_artifacts(catalog)

# get the schema of the input dataset
input_data = catalog.load(pipeline_training.input_name)
model_signature = infer_signature(model_input=input_data)

mlflow.pyfunc.log_model(
    artifact_path="model",
    python_model=KedroPipelineModel(pipeline=pipeline_training, catalog=catalog),
    artifacts=artifacts,
    conda_env={"python": "3.10.0", dependencies: ["kedro==0.18.11"]},
    signature=model_signature,
)
```

It is also possible to pass arguments to `KedroPipelineModel` to specify the runner or the copy_mode of `MemoryDataset` for the inference `Pipeline`. This may be faster especially for compiled model (e.g keras, tensorflow…), and more suitable for an API serving pattern. Since `kedro-mlflow==0.12.0`, `copy_mode="assign"` has become the default.

```python
KedroPipelineModel(pipeline=pipeline_training, catalog=catalog, copy_mode="assign")
```

Available `copy_mode` are `assign`, `copy` and `deepcopy`. It is possible to pass a dictionary to specify different copy mode for each dataset.

1.5.4 Cli commands

`init`

`kedro mlflow init`: this command is needed to initalize your project. You cannot run any other commands before you run this one once. It performs 2 actions: - creates a `mlflow.yml` configuration file in your `conf/local` folder - replace the `src/PYTHON_PACKAGE/run.py` file by an updated version of the template. If your template has been modified since project creation, a warning will be raised. You can either run `kedro mlflow init --force` to ignore this warning (but this will erase your `run.py`) or set hooks manually.

`init` has two arguments:

- `--env` which enable to specifiy another environment where the mlflow.yml should be created (e.g, `base`)
- `--force` which overrides the `mlflow.yml` if it already exists and replaces it with the default one. Use it with caution!

ui

kedro mlflow ui: this command opens the mlflow UI (basically launches the mlflow ui command)

ui accepts the port and host arguments of mlflow ui command. The default values used will be the ones defined in the mlflow.yml configuration file under the ui.

If you provide the arguments at runtime, they wil take priority over the mlflow.yml, e.g. if you have:

```yaml
# mlflow.yml
ui:
    localhost: "0.0.0.0"
    port: "5001"
```

then

```
kedro mlflow ui --port=5002
```

will open the ui on port 5002.

modelify

kedro mlflow modelify: this command converts a kedro pipeline to a mlflow model and logs it in mlflow. It enables distributing the kedro pipeline as a standalone model and leverages all mlflow serving capabilities (as an API).

modelify accepts the following arguments :

- --pipeline, -p: The name of the kedro pipeline name registered in pipeline_registry.py that you want to convert to a mlflow model.
- --input-name, -i: The name of the kedro dataset (in catalog.yml) which is the input of your pipeline. It contains the data to predict on.
- --infer-signature : A boolean which indicates if the signature of the input data should be inferred for mlflow or not.
- --infer-input-example : A boolean which indicates if the input_example of the input data should be inferred for mlflow or not
- --run-id, -r : The id of the mlflow run where the model will be logged. If unspecified, the command creates a new run.
- --run-name: The name of the mlflow run where the model will be logged. Defaults to "modelify".
- --copy-mode : The copy mode to use when replacing each dataset by a MemoryDataset. Either a string (applied all datasets) or a dict mapping each dataset to a copy_mode.
- --artifact-path" : The artifact path of mlflow.pyfunc.log_model, see https://www.mlflow.org/docs/latest/python_api/mlflow.pyfunc.html#mlflow.pyfunc.log_model
- --code-path : The code path of mlflow.pyfunc.log_model, see https://www.mlflow.org/docs/latest/python_api/mlflow.pyfunc.htm
- --conda-env : “The conda environment of mlflow.pyfunc.log_model, see https://www.mlflow.org/docs/latest/python_api/mlflow.pyfunc.html#mlflow.pyfunc.log_model
- --registered-model-name : The registered_model_name of mlflow.pyfunc.log_model, see https://www.mlflow.org/docs/latest/python_api/mlflow.pyfunc.html#mlflow.pyfunc.log_model
- --await-registration-for: The await_registration_for of mlflow.pyfunc.log_model, see https://www.mlflow.org/docs/latest/python_api/mlflow.pyfunc.html#mlflow.pyfunc.log_model*
- --pip-requirements : The pip_requirements of mlflow.pyfunc.log_model, see https://www.mlflow.org/docs/latest/python_api/mlflow.pyfunc.html#mlflow.pyfunc.log_model

- `--extra-pip-requirements` : The extra_pip_requirements of mlflow.pyfunc.log_model, see https://www.mlflow.org/docs/latest/python_api/mlflow.pyfunc.html#mlflow.pyfunc.log_model

1.5.5 Configuration

The python objecti is `KedroMlflowConfig` and it can be filled through `mlflow.yml`.

More details are coming soon.

1.5.6 kedro_mlflow package

Datasets

Artifact Dataset

class kedro_mlflow.io.artifacts.mlflow_artifact_dataset.**MlflowArtifactDataset**(*dataset: str | dict*, *run_id: str = None*, *artifact_path: str = None*, *credentials: dict[str, Any] = None*, *metadata: dict[str, Any] | None = None*)

Bases: `AbstractVersionedDataset`

This class is a wrapper for any kedro AbstractDataset. It decorates their `save` method to log the dataset in mlflow when `save` is called.

__init__(*filepath: PurePosixPath*, *version: Version | None*, *exists_function: Callable[[str], bool] | None = None*, *glob_function: Callable[[str], list[str]] | None = None*)

Creates a new instance of `AbstractVersionedDataset`.

Parameters

- **filepath** – Filepath in POSIX format to a file.
- **version** – If specified, should be an instance of `kedro.io.core.Version`. If its `load` attribute is None, the latest version will be loaded. If its `save` attribute is None, save version will be autogenerated.
- **exists_function** – Function that is used for determining whether a path exists in a filesystem.
- **glob_function** – Function that is used for finding all paths in a filesystem, which match a given pattern.

load() → Any

MlflowArtifactDataset is a factory for DataSet and consequently does not implements abtracts methods

save(*data: Any*) → None

MlflowArtifactDataset is a factory for DataSet and consequently does not implements abtracts methods

Metrics Dataset

class kedro_mlflow.io.metrics.mlflow_metric_dataset.**MlflowMetricDataset**(*key: str = None, run_id: str = None, load_args: dict[str, Any] = None, save_args: dict[str, Any] = None, metadata: dict[str, Any] | None = None*)

Bases: `MlflowAbstractMetricDataset`

DEFAULT_SAVE_MODE = 'overwrite'

SUPPORTED_SAVE_MODES = {'append', 'overwrite'}

__init__(*key: str = None, run_id: str = None, load_args: dict[str, Any] = None, save_args: dict[str, Any] = None, metadata: dict[str, Any] | None = None*)

Initialise MlflowMetricDataset. :param run_id: The ID of the mlflow run where the metric should be logged :type run_id: str

load() → None

Loads data by delegation to the provided load method.

Returns
Data returned by the provided load method.

Raises
`DatasetError` – When underlying load method raises error.

save(*data: float*) → None

Saves data by delegation to the provided save method.

Parameters
`data` – the value to be saved by provided save method.

Raises

- `DatasetError` – when underlying save method raises error.
- `FileNotFoundError` – when save method got file instead of dir, on Windows.
- `NotADirectoryError` – when save method got file instead of dir, on Unix.

class kedro_mlflow.io.metrics.mlflow_metric_history_dataset.**MlflowMetricHistoryDataset**(*key: str = None*, *run_id: str = None*, *load_args: dict[str, Any] = None*, *save_args: dict[str, Any] = None*, *metadata: dict[str, Any] | None = None*)

Bases: `MlflowAbstractMetricDataset`

__init__(*key: str = None*, *run_id: str = None*, *load_args: dict[str, Any] = None*, *save_args: dict[str, Any] = None*, *metadata: dict[str, Any] | None = None*)

Initialise MlflowMetricDataset. :param run_id: The ID of the mlflow run where the metric should be logged :type run_id: str

load() → None

Loads data by delegation to the provided load method.

Returns
Data returned by the provided load method.

Raises
DatasetError – When underlying load method raises error.

save(*data: list[int] | dict[int, float] | list[dict[str, float | str]]*) → None

Saves data by delegation to the provided save method.

Parameters
data – the value to be saved by provided save method.

Raises

- **DatasetError** – when underlying save method raises error.
- **FileNotFoundError** – when save method got file instead of dir, on Windows.
- **NotADirectoryError** – when save method got file instead of dir, on Unix.

class kedro_mlflow.io.metrics.mlflow_metrics_history_dataset.**MlflowMetricsHistoryDataset**(*run_id: str = None*, *prefix: str | None = None*, *metadata: dict[str, Any] | None = None*)

Bases: AbstractDataset

This class represent MLflow metrics dataset.

__init__(*run_id: str = None*, *prefix: str | None = None*, *metadata: dict[str, Any] | None = None*)

Initialise MlflowMetricsHistoryDataset.

Parameters

- **prefix** (*Optional[str]*) – Prefix for metrics logged in MLflow.
- **run_id** (*str*) – ID of MLflow run.

load() → dict[str, dict[str, float] | list[dict[str, float]]]

Load MlflowMetricDataSet.

Returns

dictionary with MLflow metrics dataset.

Return type

dict[str, Union[int, float]]

property run_id

Get run id.

If active run is not found, tries to find last experiment.

Raise *DatasetError* exception if run id can't be found.

Returns

String contains run_id.

Return type

str

save(*data: dict[str, dict[str, float] | list[dict[str, float]]]*) → None

Save given MLflow metrics dataset and log it in MLflow as metrics.

Parameters

data (*Metricsdict*) – MLflow metrics dataset.

Models Dataset

class kedro_mlflow.io.models.mlflow_abstract_model_dataset.**MlflowAbstractModelDataSet**(*filepath: str*, *flavor: str*, *pyfunc_workflow: str | None = None*, *load_args: dict[str, Any] = None*, *save_args: dict[str, Any] = None*, *version: Version = None*, *metadata: dict[str, Any] | None = None*)

Bases: `AbstractVersionedDataset`

Abstract mother class for model datasets.

__init__(*filepath: str*, *flavor: str*, *pyfunc_workflow: str | None = None*, *load_args: dict[str, Any] = None*, *save_args: dict[str, Any] = None*, *version: Version = None*, *metadata: dict[str, Any] | None = None*) → None

Initialize the Kedro MlflowAbstractModelDataSet.

Parameters are passed from the Data Catalog.

During save, the model is first logged to MLflow. During load, the model is pulled from MLflow run with *run_id*.

Parameters

- **filepath** (*str*) – Path to store the dataset locally.
- **flavor** (*str*) – Built-in or custom MLflow model flavor module. Must be Python-importable.

- **pyfunc_workflow** (*str, optional*) – Either *python_model* or *loader_module*. See https://www.mlflow.org/docs/latest/python_api/mlflow.pyfunc.html#workflows.
- **load_args** (*dict[str, Any], optional*) – Arguments to *load_model* function from specified *flavor*. Defaults to {}.
- **save_args** (*dict[str, Any], optional*) – Arguments to *log_model* function from specified *flavor*. Defaults to {}.
- **version** (*Version, optional*) – Specific version to load.
- **metadata** – Any arbitrary metadata. This is ignored by Kedro, but may be consumed by users or external plugins.

Raises

DatasetError – When passed *flavor* does not exist.

class kedro_mlflow.io.models.mlflow_model_tracking_dataset.**MlflowModelTrackingDataset**(*flavor: str, pyfunc_workflow: str | None = None, load_args: dict[str, Any] | None = None, save_args: dict[str, Any] | None = None, metadata: dict[str, Any] | None = None*)

Bases: *MlflowAbstractModelDataSet*

Wrapper for saving, logging and loading for all MLflow model flavor.

__init__(*flavor: str, pyfunc_workflow: str | None = None, load_args: dict[str, Any] | None = None, save_args: dict[str, Any] | None = None, metadata: dict[str, Any] | None = None*) → None

Initialize the Kedro MlflowModelDataSet.

Parameters are passed from the Data Catalog.

During save, the model is first logged to MLflow. During load, the model is pulled from MLflow through its model_id.

Parameters

- **flavor** (*str*) – Built-in or custom MLflow model flavor module. Must be Python-importable. ex: "mlflow.sklearn", "mlflow.pyfunc..."
- **pyfunc_workflow** (*str, optional*) – Either *python_model* or *loader_module*. See https://www.mlflow.org/docs/latest/python_api/mlflow.pyfunc.html#workflows.
- **load_args** (*dict[str, Any], optional*) – Arguments to *load_model* function from specified *flavor*, see mlflow documentation. Defaults to None.
- **save_args** (*dict[str, Any], optional*) – Arguments to *log_model* function from specified *flavor*, see mlflow documentation. Default to None, it is recommended to specify 'name'.
- **metadata** – Any arbitrary metadata. This is ignored by Kedro, but may be consumed by users or external plugins.

Raises

DatasetError – When passed *flavor* does not exist.

`load()` → LoggedModel

Loads an MLflow model from local path or from MLflow run.

Returns

Deserialized model.

Return type

LoggedModel

property model_uri: str | None

`save`(*model: Any*) → None

Save a model to local path and then logs it to MLflow.

Parameters

model (*Any*) – A model object supported by the given MLflow flavor.

class kedro_mlflow.io.models.mlflow_model_local_filesystem_dataset.**MlflowModelLocalFileSystemDataset**(*file str fla- vor str py- fun str | No = No loa dic An = No sav dic An = No log dic An = No ver sio Ver sio = No me dat dic An | No = No*

Bases: *MlflowAbstractModelDataSet*

Wrapper for saving, logging and loading for all MLflow model flavor.

__init__(*filepath: str*, *flavor: str*, *pyfunc_workflow: str | None = None*, *load_args: dict[str, Any] = None*, *save_args: dict[str, Any] = None*, *log_args: dict[str, Any] = None*, *version: Version = None*, *metadata: dict[str, Any] | None = None*) → None

Initialize the Kedro MlflowModelDataSet.

Parameters are passed from the Data Catalog.

During save, the model is saved locally at *filepath* During load, the model is loaded from the local *filepath*.

Parameters

- **flavor** (*str*) – Built-in or custom MLflow model flavor module. Must be Python-importable.
- **filepath** (*str*) – Path to store the dataset locally.
- **pyfunc_workflow** (*str, optional*) – Either *python_model* or *loader_module*. See https://www.mlflow.org/docs/latest/python_api/mlflow.pyfunc.html#workflows.
- **load_args** (*dict[str, Any], optional*) – Arguments to *load_model* function from specified *flavor*. Defaults to None.
- **save_args** (*dict[str, Any], optional*) – Arguments to *save_model* function from specified *flavor*. Defaults to None.
- **version** (*Version, optional*) – Kedro version to use. Defaults to None.
- **metadata** – Any arbitrary metadata. This is ignored by Kedro, but may be consumed by users or external plugins.

Raises
`DatasetError` – When passed *flavor* does not exist.

`load()` → Any

Loads an MLflow model from local path or from MLflow run.

Returns
Deserialized model.

Return type
Any

`save`(*model: Any*) → None

Save a model to local path and then logs it to MLflow.

Parameters
model (*Any*) – A model object supported by the given MLflow flavor.

class kedro_mlflow.io.models.mlflow_model_registry_dataset.**MlflowModelRegistryDataset**(*model_name: str*, *stage_or_version: str | int | None = None*, *alias: str | None = None*, *flavor: str | None = 'mlflow.pyfunc'*, *pyfunc_workflow: str | None = 'python_model'*, *load_args: dict[str, Any] | None = None*, *metadata: dict[str, Any] | None = None*)

Bases: `MlflowAbstractModelDataSet`

Wrapper for saving, logging and loading for all MLflow model flavor.

__init__(*model_name: str*, *stage_or_version: str | int | None = None*, *alias: str | None = None*, *flavor: str | None = 'mlflow.pyfunc'*, *pyfunc_workflow: str | None = 'python_model'*, *load_args: dict[str, Any] | None = None*, *metadata: dict[str, Any] | None = None*) → None

Initialize the Kedro MlflowModelRegistryDataset.

Parameters are passed from the Data Catalog.

During "load", the model is pulled from MLflow model registry by its name. "save" is not supported.

Parameters

- **model_name** (*str*) – The name of the registered model is the mlflow registry

- **stage_or_version** (*str*) – A valid stage (either "staging" or "production") or version number for the registred model. Default to "latest" which fetch the last version and the higher "stage" available.
- **flavor** (*str*) – Built-in or custom MLflow model flavor module. Must be Python-importable.
- **pyfunc_workflow** (*str, optional*) – Either *python_model* or *loader_module*. See https://www.mlflow.org/docs/latest/python_api/mlflow.pyfunc.html#workflows.
- **load_args** (*dict[str, Any], optional*) – Arguments to *load_model* function from specified *flavor*. Defaults to None.
- **metadata** – Any arbitrary metadata. This is ignored by Kedro, but may be consumed by users or external plugins.

Raises
`DatasetError` – When passed *flavor* does not exist.

`load()` → Any

Loads an MLflow model from local path or from MLflow run.

Returns
Deserialized model.

Return type
Any

`save`(*model: Any*) → None

Saves data by delegation to the provided save method.

Parameters
data – the value to be saved by provided save method.

Raises

- `DatasetError` – when underlying save method raises error.
- `FileNotFoundError` – when save method got file instead of dir, on Windows.
- `NotADirectoryError` – when save method got file instead of dir, on Unix.

CLI

Pipelines

exception kedro_mlflow.pipeline.pipeline_ml.**KedroMlflowPipelineMLError**

Bases: `Exception`

Error raised when the KedroPipelineModel construction fails

class kedro_mlflow.pipeline.pipeline_ml.**PipelineML**(*nodes: Iterable[Node | Pipeline], *args, tags: str | Iterable[str] | None = None, inference: Pipeline, input_name: str, kpm_kwargs: dict[str, str] | None = None, log_model_kwargs: dict[str, str] | None = None*)

Bases: `Pipeline`

IMPORTANT NOTE : THIS CLASS IS NOT INTENDED TO BE USED DIRECTLY IN A KEDRO PROJECT. YOU SHOULD USE `pipeline_ml_factory` FUNCTION FOR MODULAR PIPELINE WHICH IS MORE FLEXIBLE AND USER FRIENDLY. SEE INSERT_DOC_URL

A `PipelineML` is a kedro `Pipeline` which we assume is a "training" (in the machine learning way) pipeline. Basically, "training" is a higher order function (it generates another function). It implies that: - the outputs of this pipeline are considered as "fitted models", i.e. inputs of another inference pipeline (it is very likely that there are several outputs because we need to store any object that depends on the train data (e.g encoders, binarizers, vectorizer, machine learning models…) - These outputs will feed another "inference" pipeline (to be used for prediction purpose) whose inputs

are the outputs of the "training" pipeline, except for one of them (the new data to predict).

This class enables to "link" a training pipeline and an inference pipeline in order to package them in mlflow easily. The goal is to call the `MlflowHook` hook after a PipelineMl is called in order to trigger mlflow packaging.

KPM_KWARGS_DEFAULT = {}

LOG_MODEL_KWARGS_DEFAULT = {'name': 'model', 'signature': 'auto'}

__init__(*nodes: Iterable[Node | Pipeline]*, **args*, *tags: str | Iterable[str] | None = None*, *inference: Pipeline*, *input_name: str*, *kpm_kwargs: dict[str, str] | None = None*, *log_model_kwargs: dict[str, str] | None = None*)

Store all necessary information for calling mlflow.log_model in the pipeline.

Parameters

- **nodes** (*Iterable[Union[Node, Pipeline]]*) – The `node`s of the training pipeline.
- **tags** (*Union[str, Iterable[str]], optional*) – Optional set of tags to be applied to all the pipeline nodes. Defaults to None.
- **inference** (*Pipeline*) – A *Pipeline* object which will be stored in mlflow and use the output(s) of the training pipeline (namely, the model) to predict the outcome.
- **input_name** (*str, optional*) – The name of the dataset in the catalog.yml which the model's user must provide for prediction (i.e. the data). Defaults to None.
- **kpm_kwargs** – extra arguments to be passed to *KedroPipelineModel* when the PipelineML object is automatically saved at the end of a run. This includes:
  - *copy_mode*: the copy_mode to be used for underlying dataset

  when loaded in memory - *runner*: the kedro runner to run the model with
- **log_model_kwargs** –

  extra arguments to be passed to *mlflow.pyfunc.log_model*, e.g.:

  - "signature" accepts an extra "auto" which automatically infer the signature

  based on "input_name" dataset

filter(*tags: Iterable[str] | None = None*, *from_nodes: Iterable[str] | None = None*, *to_nodes: Iterable[str] | None = None*, *node_names: Iterable[str] | None = None*, *from_inputs: Iterable[str] | None = None*, *to_outputs: Iterable[str] | None = None*, *node_namespaces: Iterable[str] | None = None*) → Pipeline

Creates a new `Pipeline` object with the nodes that meet all of the specified filtering conditions.

The new pipeline object is the intersection of pipelines that meet each filtering condition. This is distinct from chaining multiple filters together.

Parameters

- **tags** – A list of node tags which should be used to lookup the nodes of the new `Pipeline`.

- **from_nodes** – A list of node names which should be used as a starting point of the new `Pipeline`.
- **to_nodes** – A list of node names which should be used as an end point of the new `Pipeline`.
- **node_names** – A list of node names which should be selected for the new `Pipeline`.
- **from_inputs** – A list of inputs which should be used as a starting point of the new `Pipeline`
- **to_outputs** – A list of outputs which should be the final outputs of the new `Pipeline`.
- **node_namespaces** – A list of node namespaces which should be used to select nodes in the new `Pipeline`.

Returns

A new Pipeline object with nodes that meet all of the specified filtering conditions.

Raises
ValueError – The filtered `Pipeline` has no nodes.

Example: ``` python pipeline = Pipeline(

[

node(func, "A", "B", name="node1"), node(func, "B", "C", name="node2"), node(func, "C", "D", name="node3"),

]

) pipeline.filter(node_names=["node1", "node3"], from_inputs=["A"]) # Gives a new pipeline object containing node1 and node3. ```

from_inputs(**inputs: str*) → *PipelineML*

Create a new `Pipeline` object with the nodes which depend directly or transitively on the provided inputs. If provided a name, but no format, for a transcoded input, it includes all the nodes that use inputs with that name, otherwise it matches to the fully-qualified name only (i.e. name@format).

Parameters
***inputs** – A list of inputs which should be used as a starting point of the new `Pipeline`

Raises
ValueError – Raised when any of the given inputs do not exist in the `Pipeline` object.

Returns

A new Pipeline object, containing a subset of the nodes of the current one such that only nodes depending directly or transitively on the provided inputs are being copied.

from_nodes(**node_names: str*) → *PipelineML*

Create a new `Pipeline` object with the nodes which depend directly or transitively on the provided nodes.

Parameters
***node_names** – A list of node_names which should be used as a starting point of the new `Pipeline`.

Raises
ValueError – Raised when any of the given names do not exist in the `Pipeline` object.

Returns

A new `Pipeline` object, containing a subset of the nodes of
the current one such that only nodes depending directly or transitively on the provided nodes are being copied.

property inference: str

property input_name: str

only_nodes(**node_names: str*) → Pipeline

Create a new `Pipeline` which will contain only the specified nodes by name.

Parameters
***node_names** – One or more node names. The returned `Pipeline` will only contain these nodes.

Raises
ValueError – When some invalid node name is given.

Returns
A new `Pipeline`, containing only `nodes`.

only_nodes_with_inputs(**inputs: str*) → Pipeline

Create a new `Pipeline` object with the nodes which depend directly on the provided inputs. If provided a name, but no format, for a transcoded input, it includes all the nodes that use inputs with that name, otherwise it matches to the fully-qualified name only (i.e. name@format).

Parameters
***inputs** – A list of inputs which should be used as a starting point of the new `Pipeline`.

Raises
ValueError – Raised when any of the given inputs do not exist in the `Pipeline` object.

Returns

A new `Pipeline` object, containing a subset of the
nodes of the current one such that only nodes depending directly on the provided inputs are being copied.

only_nodes_with_namespaces(*node_namespaces: str*) → Pipeline

Creates a new `Pipeline` containing only nodes with the specified namespaces.

Parameters
node_namespaces – A list of node namespaces.

Raises
ValueError – When pipeline contains no nodes with the specified namespaces.

Returns
A new `Pipeline` containing nodes with the specified namespaces.

only_nodes_with_outputs(**outputs: str*) → Pipeline

Create a new `Pipeline` object with the nodes which are directly required to produce the provided outputs. If provided a name, but no format, for a transcoded dataset, it includes all the nodes that output to that name, otherwise it matches to the fully-qualified name only (i.e. name@format).

Parameters
***outputs** – A list of outputs which should be the final outputs of the new `Pipeline`.

Raises
ValueError – Raised when any of the given outputs do not exist in the `Pipeline` object.

Returns
A new `Pipeline` object, containing a subset of the nodes of the current one such that only nodes which are directly required to produce the provided outputs are being copied.

`only_nodes_with_tags`(**tags: str*) → *PipelineML*

Creates a new `Pipeline` object with the nodes which contain *any* of the provided tags. The resulting `Pipeline` is empty if no tags are provided.

Parameters
`*tags` – A list of node tags which should be used to lookup the nodes of the new `Pipeline`.

Returns

A new `Pipeline` object, containing a subset of the
nodes of the current one such that only nodes containing *any* of the tags provided are being copied.

Return type
Pipeline

`tag`(*tags: str | Iterable[str]*) → *PipelineML*

Tags all the nodes in the pipeline.

Parameters
`tags` – The tags to be added to the nodes.

Returns
New `Pipeline` object with nodes tagged.

`to_nodes`(**node_names: str*) → *PipelineML*

Create a new `Pipeline` object with the nodes required directly or transitively by the provided nodes.

Parameters
`*node_names` – A list of node_names which should be used as an end point of the new `Pipeline`.

Raises
`ValueError` – Raised when any of the given names do not exist in the `Pipeline` object.

Returns

A new `Pipeline` object, containing a subset of the nodes of the
current one such that only nodes required directly or transitively by the provided nodes are being copied.

`to_outputs`(**outputs: str*) → *PipelineML*

Create a new `Pipeline` object with the nodes which are directly or transitively required to produce the provided outputs. If provided a name, but no format, for a transcoded dataset, it includes all the nodes that output to that name, otherwise it matches to the fully-qualified name only (i.e. name@format).

Parameters
`*outputs` – A list of outputs which should be the final outputs of the new `Pipeline`.

Raises
`ValueError` – Raised when any of the given outputs do not exist in the `Pipeline` object.

Returns
A new `Pipeline` object, containing a subset of the nodes of the current one such that only nodes which are directly or transitively required to produce the provided outputs are being copied.

property training: Pipeline

kedro_mlflow.pipeline.pipeline_ml_factory.**pipeline_ml_factory**(*training: Pipeline*, *inference: Pipeline*, *input_name: str = None*, *kpm_kwargs=None*, *log_model_kwargs=None*) → *PipelineML*

This function is a helper to create *PipelineML* object directly from two Kedro *Pipelines* (one of training and one of inference) .

Parameters

- **training** (*Pipeline*) – The *Pipeline* object that creates all mlflow artifacts for prediction (the model, but also encoders, binarizers, tokenizers…). These artifacts must be persisted in the catalog.yml.
- **inference** (*Pipeline*) – A *Pipeline* object which will be stored in mlflow and use the output(s) of the training pipeline (namely, the model) to predict the outcome.
- **input_name** (*str, optional*) – The name of the dataset in the catalog.yml which the model's user must provide for prediction (i.e. the data). Defaults to None.
- **kpm_kwargs** – extra arguments to be passed to *KedroPipelineModel* when the PipelineML object is automatically saved at the end of a run. This includes:
  - *copy_mode*: the copy_mode to be used for underlying dataset

  when loaded in memory - *runner*: the kedro runner to run the model with
- **logging_kwargs** – extra arguments to be passed to *mlflow.pyfunc.log_model* when the PipelineML object is automatically saved at the end of a run. See mlflow documentation to see all available options: https://www.mlflow.org/docs/latest/python_api/mlflow.pyfunc.html#mlflow.pyfunc.log_model

Returns

A *PipelineML* which is automatically
discovered by the *MlflowHook* and contains all the information for logging the inference pipeline as a Mlflow Model.

Return type

PipelineML

Custom Mlflow Models

Configuration

class kedro_mlflow.config.kedro_mlflow_config.**CreateExperimentOptions**(* *(Keyword-only parameters separator (PEP 3102))*, *artifact_location: str | None = None*, *tags: dict | None = None*)

Bases: BaseModel

class Config

Bases: object

extra = 'forbid'

artifact_location: str | None

model_config: ClassVar[ConfigDict] = {'extra': 'forbid'}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

tags: dict | None

class kedro_mlflow.config.kedro_mlflow_config.**DisableTrackingOptions**(**, pipelines: list[str] = [], disable_autologging: bool = True*)

Bases: BaseModel

class Config

Bases: object

extra = 'forbid'

disable_autologging: bool

model_config: ClassVar[ConfigDict] = {'extra': 'forbid'}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

pipelines: list[str]

class kedro_mlflow.config.kedro_mlflow_config.**ExperimentOptions**(**, name: str = 'Default', create_experiment_kwargs: CreateExperimentOptions = CreateExperimentOptions(artifact_location=None, tags=None), restore_if_deleted: Annotated[bool, Strict(strict=True)] = True*)

Bases: BaseModel

class Config

Bases: object

extra = 'forbid'

create_experiment_kwargs: *CreateExperimentOptions*

model_config: ClassVar[ConfigDict] = {'extra': 'forbid'}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_post_init(*context: Any, / (Positional-only parameter separator (PEP 570))*) → None

This function is meant to behave like a BaseModel method to initialize private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **context** – The context.

name: str

restore_if_deleted: Annotated[bool, Strict(strict=True)]

class kedro_mlflow.config.kedro_mlflow_config.**KedroMlflowConfig**(*, *server:* MlflowServerOptions *= MlflowServerOptions(mlflow_tracking_uri=None, mlflow_registry_uri=None, credentials=None, request_header_provider=RequestHeaderProviderOp pass_context=False, init_kwargs={})), tracking:* MlflowTrackingOptions *= MlflowTrackingOptions(disable_tracking=DisableTrackingOptions(pi disable_autologging=True), experiment=ExperimentOptions(name='Default', create_experiment_kwargs=CreateExperimentOptions tags=None), restore_if_deleted=True), run=RunOptions(id=None, name=None, nested=True), params=MlflowParamsOptions(dict_params=dictP recursive=True, sep='.'), long_params_strategy='fail')), ui:* UiOptions *= UiOptions(port='5000', host='127.0.0.1')*)

Bases: BaseModel

class Config

Bases: object

extra = 'forbid'

validate_assignment = True

model_config: ClassVar[ConfigDict] = {'extra': 'forbid', 'validate_assignment': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

server: *MlflowServerOptions*

setup(*context*)

Setup all the mlflow configuration

tracking: *MlflowTrackingOptions*

ui: *UiOptions*

class kedro_mlflow.config.kedro_mlflow_config.**MlflowParamsOptions**(*, *dict_params:* dictParamsOptions *= dictParamsOptions(flatten=False, recursive=True, sep='.'), long_params_strategy: Literal['fail', 'truncate', 'tag'] = 'fail'*)

Bases: `BaseModel`

class Config

Bases: `object`

extra = 'forbid'

dict_params: *dictParamsOptions*

long_params_strategy: Literal['fail', 'truncate', 'tag']

model_config: ClassVar[ConfigDict] = {'extra': 'forbid'}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

class kedro_mlflow.config.kedro_mlflow_config.**MlflowServerOptions**(*, *mlflow_tracking_uri: str | None = None*, *mlflow_registry_uri: str | None = None*, *credentials: str | None = None*, *request_header_provider:* RequestHeaderProviderOptions *= RequestHeaderProviderOptions(type=None, pass_context=False, init_kwargs={})*)

Bases: `BaseModel`

class Config

Bases: `object`

extra = 'forbid'

credentials: str | None

mlflow_registry_uri: str | None

mlflow_tracking_uri: str | None

model_config: ClassVar[ConfigDict] = {'extra': 'forbid'}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

model_post_init(*context: Any*, */*) → None

This function is meant to behave like a BaseModel method to initialize private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** – The BaseModel instance.
- **context** – The context.

request_header_provider: *RequestHeaderProviderOptions*

class kedro_mlflow.config.kedro_mlflow_config.**MlflowTrackingOptions**(*, *disable_tracking: DisableTrackingOptions = DisableTrackingOptions(pipelines=[], disable_autologging=True), experiment: ExperimentOptions = ExperimentOptions(name='Default', create_experiment_kwargs=CreateExperimentOp tags=None), restore_if_deleted=True), run: RunOptions = RunOptions(id=None, name=None, nested=True), params: MlflowParamsOptions = MlflowParamsOptions(dict_params=dictParamsOptions(flatten recursive=True, sep='.'), long_params_strategy='fail')*)

Bases: BaseModel

class Config

Bases: object

extra = 'forbid'

disable_tracking: *DisableTrackingOptions*

experiment: *ExperimentOptions*

model_config: ClassVar[ConfigDict] = {'extra': 'forbid'}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

params: *MlflowParamsOptions*

run: *RunOptions*

class kedro_mlflow.config.kedro_mlflow_config.**RequestHeaderProviderOptions**(*, *type: str | None = None, pass_context: bool = False, init_kwargs: dict[str, str] = {}*)

Bases: BaseModel

class Config

Bases: object

arbitrary_types_allowed = 'allowed'

extra = 'forbid'

init_kwargs: dict[str, str]

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': 'allowed', 'extra': 'forbid'}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

pass_context: bool

type: str | None

class kedro_mlflow.config.kedro_mlflow_config.**RunOptions**(**, id: str | None = None, name: str | None = None, nested: Annotated[bool, Strict(strict=True)] = True*)

Bases: BaseModel

class Config

Bases: object

extra = 'forbid'

id: str | None

model_config: ClassVar[ConfigDict] = {'extra': 'forbid'}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

name: str | None

nested: Annotated[bool, Strict(strict=True)]

class kedro_mlflow.config.kedro_mlflow_config.**UiOptions**(**, port: str = '5000', host: str = '127.0.0.1'*)

Bases: BaseModel

class Config

Bases: object

extra = 'forbid'

host: str

model_config: ClassVar[ConfigDict] = {'extra': 'forbid'}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

port: str

class kedro_mlflow.config.kedro_mlflow_config.**dictParamsOptions**(**, flatten: Annotated[bool, Strict(strict=True)] = False, recursive: Annotated[bool, Strict(strict=True)] = True, sep: str = '.'*)

Bases: BaseModel

class Config

Bases: object

extra = 'forbid'

flatten: Annotated[bool, Strict(strict=True)]

model_config: ClassVar[ConfigDict] = {'extra': 'forbid'}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

recursive: Annotated[bool, Strict(strict=True)]

sep: str

Hooks

Node Hook

1.6 Migration guides

1.6.1 Migration guide between kedro-mlflow versions

This page explains how to migrate an existing kedro project to a more up to date `kedro-mlflow` versions with breaking changes.

Migration from 0.13.x to 0.14.x

Upgrade mlflow to `mlflow>=2.7.0`.

Migration from 0.12.x to 0.13.x

Upgrade mlflow to `mlflow>=1.30`.

Migration from 0.11.x to 0.12.x

1. Upgrade your kedro project to `kedro>=0.19,<0.20`
2. Rename the following `DataSets` with the `Dataset` suffix (**without final capitalized S**) in your `catalog.yml` and change names to make them more explicit: | Name in `kedro_mlflow<=0.11`|Name in `kedro_mlflow>=0.12` | |——————————–|———————————–| |`MlflowArtifactDataSet` |`MlflowArtifactDataset` | |`MlflowAbstractModelDataSet` |`MlflowAbstractModelDataset` | |`MlflowModelRegistryDataSet` |`MlflowModelRegistryDataset` | |`MlflowMetricDataSet` |`MlflowMetricDataset` | |`MlflowMetricHistoryDataSet` |`MlflowMetricHistoryDataset` | |`MlflowModelLoggerDataSet` |`MlflowModelTrackingDataset` | |`MlflowModelSaverDataSet` |`MlflowModelLocalFileSystemDataset`| |`MlflowMetricsDataSet` |`MlflowMetricsHistoryDataset` |
3. Update your `MlflowArtifactDataset` catalog entry to rename the `data_set` key to `dataset`

```yaml
my_dataset:
    type: MlflowArtifactDataset
    dataset:
        type: ...
```

4. If you use `KedroPipelineModel` or `pipeline_ml_factory`, the default `copy_mode` is now `assign`because this is the most efficient setup (and usually the desired one) when serving a Kedro Pipeline as a Mlflow model. To get back to the previous `deepcopy` mode, change the entry to:

```
pipeline_ml_factory(
    training=training_pipeline,
    inference=inference_pipeline,
    kpm_kwargs=dict(copy_mode="deepcopy"),
)
```

Migration from 0.10.x to 0.11.x

1. If you are registering your `kedro_mlflow` hooks manually (instead of using automatic registering from plugin, which is the default), change your `settings.py`

from this:

```
# <your_project>/src/<your_project>/settings.py
from kedro_mlflow.framework.hooks import MlflowHook

HOOKS = (MlflowPipelineHook(), MlflowNodeHook())
```

to this:

```
# <your_project>/src/<your_project>/settings.py
from kedro_mlflow.framework.hooks import MlflowHook

HOOKS = (MlflowHook(),)
```

2. The `get_mlflow_config` public method has been removed and the mlflow configuration is now automatically stored in the `mlflow` attribute of `KedroContext`. if you need to access the mlflow configuration, you can use:

```
from kedro.framework.session import KedroSession
from kedro.framework.startup import bootstrap_project

bootstrap_project(project_path)
with KedroSession.create(
    project_path=project_path,
) as session:
    context = session.load_context()
    print(context.mlflow)  # this is where mlflow configuration is stored
```

3. Remove the `server.stores_environment_variables` key from `mlflow.yml`. This is a dead key which was unused. It will now throw an error if it is still written in `mlflow.yml`.

Migration from 0.9.x to 0.10.x

You must upgrade your kedro version to `kedro>=0.18.1` to use `kedro_mlflow>=0.10`.

Migration from 0.8.x to 0.9.x

There are no breaking change in this patch release except if you retrieve the mlflow configuration manually (e.g. in a script or a jupyter notebok). The `setup()` method needs to be called with `context`:

```
from kedro.framework.context import load_context
from kedro_mlflow.config import get_mlflow_config

context = load_context(".")
```

(continues on next page)

(continued from previous page)

```
# the new best practice is just to remove these lines
mlflow_config = get_mlflow_config(context)  # pass context instead of session
mlflow_config.setup(context)  # pass context instead of session
```

This is not necessary: the mlflow config is automatically set up when the context is loaded, so unless you need to access the config manually you can get rid of these 2 lines

Migration from 0.7.x to 0.8.x

- Update the `mlflow.yml` configuration file with `kedro mlflow init --force` command
- `pipeline_ml_factory(pipeline_ml=<your-pipeline-ml>,...)` (resp. `KedroPipelineModel(pipeline_ml=<your-pipeline-ml>, ...)`) first argument is renamed `pipeline`. Change the call to `pipeline_ml_factory(pipeline=<your-pipeline-ml>)` (resp. `KedroPipelineModel(pipeline=<your-pipeline-ml>, ...)`).
- Change the call from `pipeline_ml_factory(..., model_signature=<model-signature>, conda_env=<conda-env>, model_name=<model_name>)` to ``pipeline_ml_factory(..., log_model_kwargs=dict(signature=, conda_env=, artifact_path=<model_name>}). Notice that the arguments are renamed to match mlflow's and they are passed as a dict in log_model_kwargs`.

Migration from 0.6.x to 0.7.x

If you are working with `kedro==0.17.0`, update your template to `kedro>=0.17.1`.

Migration from 0.5.x to 0.6.x

`kedro==0.16.x` is no longer supported. You need to update your project template to `kedro==0.17.0` template.

Migration from 0.4.x to 0.5.x

The only breaking change with the previous release is the format of `KedroPipelineMLModel` class. Hence, if you saved a pipeline as a Mlflow Model with `pipeline_ml_factory` in `kedro-mlflow==0.4.x`, loading it (either with `MlflowModelTrackingDataset` or `mlflow.pyfunc.load_model`) with `kedro-mlflow==0.5.0` installed will raise an error. You will need either to retrain the model or to load it with `kedro-mlflow==0.4.x`.

Migration from 0.4.0 to 0.4.1

There are no breaking change in this patch release except if you retrieve the mlflow configuration manually (e.g. in a script or a jupyter notebok). You must add an extra call to the `setup()` method:

```
from kedro.framework.context import load_context
from kedro_mlflow.config import get_mlflow_config

context = load_context(".")
mlflow_config = get_mlflow_config(context)
mlflow_config.setup()  # <-- add this line which did not exists in 0.4.0
```

Migration from 0.3.x to 0.4.x

Catalog entries

Replace the following entries:

Hooks

Hooks are now auto-registered if you use `kedro>=0.16.4`. You can remove the following entry from your `run.py`:

```
hooks = (MlflowPipelineHook(), MlflowNodeHook())
```

KedroPipelineModel

Be aware that if you have saved a pipeline as a mlflow model with `pipeline_ml_factory`, retraining this pipeline with `kedro-mlflow==0.4.0` will lead to a new behaviour. Let assume the name of your output in the `DataCatalog` was `predictions`, the output of a registered model will be modified from:

```
{
    "predictions":
        {
            "<your model-predictions>"
        }
}
```

to:

```
{
    "<your model-predictions>"
}
```

Thus, parsing the predictions of this model must be updated accordingly.

1.6.2 Migration guide from kedro-viz experiment tracking

If you use Kedro's native experiment tracking functionality, it will be deprecated from `kedro-viz==0.11.0`.

The core team suggest migrating to kedro-mlflow and provides a blog post to explain the process.

News — 5 min read

Deprecating Experiment Tracking in Kedro Viz

Kedro-Viz will phase out its Experiment Tracking feature in the upcoming release of Kedro-Viz 11.0, with complete removal in version 12.0 due to low user adoption and the availability of robust alternatives like MLflow. This blog post includes detailed guidance on migrating to kedro-mlflow, a plugin that seamlessly integrates Kedro with MLflow.

28 Jan 2025 (last updated 28 Jan 2025)

https://kedro.org/blog/deprecate-experiment-tracking-kedro-viz

PYTHON MODULE INDEX

k

kedro_mlflow.config.kedro_mlflow_config, 79
kedro_mlflow.io.artifacts.mlflow_artifact_dataset, 64
kedro_mlflow.io.metrics.mlflow_metric_dataset, 65
kedro_mlflow.io.metrics.mlflow_metric_history_dataset, 65
kedro_mlflow.io.metrics.mlflow_metrics_history_dataset, 66
kedro_mlflow.io.models.mlflow_abstract_model_dataset, 68
kedro_mlflow.io.models.mlflow_model_local_filesystem_dataset, 70
kedro_mlflow.io.models.mlflow_model_registry_dataset, 72
kedro_mlflow.io.models.mlflow_model_tracking_dataset, 69
kedro_mlflow.pipeline.pipeline_ml, 74
kedro_mlflow.pipeline.pipeline_ml_factory, 79

INDEX

Symbols

`__init__()` (*kedro_mlflow.io.artifacts.mlflow_artifact_dataset.MlflowArtifactDataset method*), 64
`__init__()` (*kedro_mlflow.io.metrics.mlflow_metric_dataset.MlflowMetricDataset method*), 65
`__init__()` (*kedro_mlflow.io.metrics.mlflow_metric_history_dataset.MlflowMetricHistoryDataset method*), 66
`__init__()` (*kedro_mlflow.io.metrics.mlflow_metrics_history_dataset.MlflowMetricsHistoryDataset method*), 67
`__init__()` (*kedro_mlflow.io.models.mlflow_abstract_model_dataset.MlflowAbstractModelDataSet method*), 68
`__init__()` (*kedro_mlflow.io.models.mlflow_model_local_filesystem_dataset.MlflowModelLocalFileSystemDataset method*), 71
`__init__()` (*kedro_mlflow.io.models.mlflow_model_registry_dataset.MlflowModelRegistryDataset method*), 73
`__init__()` (*kedro_mlflow.io.models.mlflow_model_tracking_dataset.MlflowModelTrackingDataset method*), 69
`__init__()` (*kedro_mlflow.pipeline.pipeline_ml.PipelineML method*), 75

A

`arbitrary_types_allowed` (*kedro_mlflow.config.kedro_mlflow_config.RequestHeaderProviderOptions.Config attribute*), 83
`artifact_location` (*kedro_mlflow.config.kedro_mlflow_config.CreateExperimentOptions attribute*), 79

C

`create_experiment_kwargs` (*kedro_mlflow.config.kedro_mlflow_config.ExperimentOptions attribute*), 80
`CreateExperimentOptions` (*class in kedro_mlflow.config.kedro_mlflow_config*), 79
`CreateExperimentOptions.Config` (*class in kedro_mlflow.config.kedro_mlflow_config*), 79
`credentials` (*kedro_mlflow.config.kedro_mlflow_config.MlflowServerOptions attribute*), 82

D

`DEFAULT_SAVE_MODE` (*kedro_mlflow.io.metrics.mlflow_metric_dataset.MlflowMetricDataset attribute*), 65
`dict_params` (*kedro_mlflow.config.kedro_mlflow_config.MlflowParamsOp attribute*), 82
`dictParamsOptions` (*class in kedro_mlflow.config.kedro_mlflow_config*), 84
`dictParamsOptions.Config` (*class in kedro_mlflow.config.kedro_mlflow_config*), 84
`disable_autologging` (*kedro_mlflow.config.kedro_mlflow_config.DisableTrackingOptions attribute*), 80
`disable_tracking` (*kedro_mlflow.config.kedro_mlflow_config.MlflowTrackingOptions attribute*), 83
`DisableTrackingOptions` (*class in kedro_mlflow.config.kedro_mlflow_config*), 80
`DisableTrackingOptions.Config` (*class in kedro_mlflow.config.kedro_mlflow_config*), 80

E

`experiment` (*kedro_mlflow.config.kedro_mlflow_config.MlflowTrackingOp attribute*), 83
`ExperimentOptions` (*class in kedro_mlflow.config.kedro_mlflow_config*), 80
`ExperimentOptions.Config` (*class in kedro_mlflow.config.kedro_mlflow_config*), 80
`extra` (*kedro_mlflow.config.kedro_mlflow_config.CreateExperimentOptions attribute*), 79
`extra` (*kedro_mlflow.config.kedro_mlflow_config.dictParamsOptions.Config attribute*), 84
`extra` (*kedro_mlflow.config.kedro_mlflow_config.DisableTrackingOptions.C attribute*), 80
`extra` (*kedro_mlflow.config.kedro_mlflow_config.ExperimentOptions.Confi attribute*), 80
`extra` (*kedro_mlflow.config.kedro_mlflow_config.KedroMlflowConfig.Confi attribute*), 81
`extra` (*kedro_mlflow.config.kedro_mlflow_config.MlflowParamsOptions.Co*

attribute), 82

extra (*kedro_mlflow.config.kedro_mlflow_config.MlflowServerOptions.Config attribute*), 82

extra (*kedro_mlflow.config.kedro_mlflow_config.MlflowTrackingOptions.Config attribute*), 83

extra (*kedro_mlflow.config.kedro_mlflow_config.RequestHeaderProviderOptions.Config attribute*), 83

extra (*kedro_mlflow.config.kedro_mlflow_config.RunOptions.Config attribute*), 84

extra (*kedro_mlflow.config.kedro_mlflow_config.UiOptions.Config attribute*), 84

F

filter() (*kedro_mlflow.pipeline.pipeline_ml.PipelineML method*), 75

flatten (*kedro_mlflow.config.kedro_mlflow_config.dictParamsOptions attribute*), 85

from_inputs() (*kedro_mlflow.pipeline.pipeline_ml.PipelineML method*), 76

from_nodes() (*kedro_mlflow.pipeline.pipeline_ml.PipelineML method*), 76

H

host (*kedro_mlflow.config.kedro_mlflow_config.UiOptions attribute*), 84

I

id (*kedro_mlflow.config.kedro_mlflow_config.RunOptions attribute*), 84

inference (*kedro_mlflow.pipeline.pipeline_ml.PipelineML property*), 77

init_kwargs (*kedro_mlflow.config.kedro_mlflow_config.RequestHeaderProviderOptions attribute*), 83

input_name (*kedro_mlflow.pipeline.pipeline_ml.PipelineML property*), 77

K

kedro_mlflow.config.kedro_mlflow_config
  module, 79

kedro_mlflow.io.artifacts.mlflow_artifact_dataset
  module, 64

kedro_mlflow.io.metrics.mlflow_metric_dataset
  module, 65

kedro_mlflow.io.metrics.mlflow_metric_history_dataset
  module, 65

kedro_mlflow.io.metrics.mlflow_metrics_history_dataset
  module, 66

kedro_mlflow.io.models.mlflow_abstract_model_dataset
  module, 68

kedro_mlflow.io.models.mlflow_model_local_filesystem_dataset
  module, 70

kedro_mlflow.io.models.mlflow_model_registry_dataset
  module, 72

kedro_mlflow.io.models.mlflow_model_tracking_dataset
  module, 69

kedro_mlflow.pipeline.pipeline_ml
  module, 74

kedro_mlflow.pipeline.pipeline_ml_factory
  module, 79

KedroMlflowConfig (*class in kedro_mlflow.config.kedro_mlflow_config*), 81

KedroMlflowConfig.Config (*class in kedro_mlflow.config.kedro_mlflow_config*), 81

KedroMlflowPipelineMLError, 74

KPM_KWARGS_DEFAULT (*kedro_mlflow.pipeline.pipeline_ml.PipelineML attribute*), 75

L

load() (*kedro_mlflow.io.artifacts.mlflow_artifact_dataset.MlflowArtifactDa method*), 64

load() (*kedro_mlflow.io.metrics.mlflow_metric_dataset.MlflowMetricData method*), 65

load() (*kedro_mlflow.io.metrics.mlflow_metric_history_dataset.MlflowMet method*), 66

load() (*kedro_mlflow.io.metrics.mlflow_metrics_history_dataset.MlflowMe method*), 67

load() (*kedro_mlflow.io.models.mlflow_model_local_filesystem_dataset.M method*), 72

load() (*kedro_mlflow.io.models.mlflow_model_registry_dataset.MlflowMo method*), 74

load() (*kedro_mlflow.io.models.mlflow_model_tracking_dataset.MlflowMo method*), 70

LOG_MODEL_KWARGS_DEFAULT (*kedro_mlflow.pipeline.pipeline_ml.PipelineML attribute*), 75

long_params_strategy (*kedro_mlflow.config.kedro_mlflow_config.MlflowParamsOptions attribute*), 82

M

mlflow_registry_uri (*kedro_mlflow.config.kedro_mlflow_config.MlflowServerOptions attribute*), 82

mlflow_tracking_uri (*kedro_mlflow.config.kedro_mlflow_config.MlflowServerOptions attribute*), 82

MlflowAbstractModelDataSet (*class in kedro_mlflow.io.models.mlflow_abstract_model_dataset*), 68

MlflowArtifactDataset (*class in kedro_mlflow.io.artifacts.mlflow_artifact_dataset*), 64

MlflowMetricDataset (*class in kedro_mlflow.io.metrics.mlflow_metric_dataset*),

65
MlflowMetricHistoryDataset (*class in kedro_mlflow.io.metrics.mlflow_metric_history_dataset*), 65
MlflowMetricsHistoryDataset (*class in kedro_mlflow.io.metrics.mlflow_metrics_history_dataset*), 66
MlflowModelLocalFileSystemDataset (*class in kedro_mlflow.io.models.mlflow_model_local_filesystem_dataset*), 70
MlflowModelRegistryDataset (*class in kedro_mlflow.io.models.mlflow_model_registry_dataset*), 72
MlflowModelTrackingDataset (*class in kedro_mlflow.io.models.mlflow_model_tracking_dataset*), 69
MlflowParamsOptions (*class in kedro_mlflow.config.kedro_mlflow_config*), 81
MlflowParamsOptions.Config (*class in kedro_mlflow.config.kedro_mlflow_config*), 82
MlflowServerOptions (*class in kedro_mlflow.config.kedro_mlflow_config*), 82
MlflowServerOptions.Config (*class in kedro_mlflow.config.kedro_mlflow_config*), 82
MlflowTrackingOptions (*class in kedro_mlflow.config.kedro_mlflow_config*), 82
MlflowTrackingOptions.Config (*class in kedro_mlflow.config.kedro_mlflow_config*), 83
model_config (*kedro_mlflow.config.kedro_mlflow_config.CreateExperimentOptions attribute*), 80
model_config (*kedro_mlflow.config.kedro_mlflow_config.dictParamsOptions attribute*), 85
model_config (*kedro_mlflow.config.kedro_mlflow_config.DisableTrackingOptions attribute*), 80
model_config (*kedro_mlflow.config.kedro_mlflow_config.ExperimentOptions attribute*), 80
model_config (*kedro_mlflow.config.kedro_mlflow_config.KedroMlflowConfig attribute*), 81
model_config (*kedro_mlflow.config.kedro_mlflow_config.MlflowParamsOptions attribute*), 82
model_config (*kedro_mlflow.config.kedro_mlflow_config.MlflowServerOptions attribute*), 82
model_config (*kedro_mlflow.config.kedro_mlflow_config.MlflowTrackingOptions attribute*), 83
model_config (*kedro_mlflow.config.kedro_mlflow_config.RequestHeaderProviderOptions attribute*), 84
model_config (*kedro_mlflow.config.kedro_mlflow_config.RunOptions attribute*), 84
model_config (*kedro_mlflow.config.kedro_mlflow_config.UiOptions attribute*), 84
model_post_init() (*kedro_mlflow.config.kedro_mlflow_config.ExperimentOptions method*), 80
model_post_init() (*kedro_mlflow.config.kedro_mlflow_config.MlflowServerOptions method*), 82
model_uri (*kedro_mlflow.io.models.mlflow_model_tracking_dataset.Mlflow property*), 70
module
kedro_mlflow.config.kedro_mlflow_config, 79
kedro_mlflow.io.artifacts.mlflow_artifact_dataset, 64
kedro_mlflow.io.metrics.mlflow_metric_dataset, 65
kedro_mlflow.io.metrics.mlflow_metric_history_dataset, 65
kedro_mlflow.io.metrics.mlflow_metrics_history_dataset 66
kedro_mlflow.io.models.mlflow_abstract_model_dataset, 68
kedro_mlflow.io.models.mlflow_model_local_filesystem_d 70
kedro_mlflow.io.models.mlflow_model_registry_dataset, 72
kedro_mlflow.io.models.mlflow_model_tracking_dataset, 69
kedro_mlflow.pipeline.pipeline_ml, 74
kedro_mlflow.pipeline.pipeline_ml_factory, 79

N

name (*kedro_mlflow.config.kedro_mlflow_config.ExperimentOptions attribute*), 80
name (*kedro_mlflow.config.kedro_mlflow_config.RunOptions attribute*), 84
nested (*kedro_mlflow.config.kedro_mlflow_config.RunOptions attribute*), 84

O

only_nodes() (*kedro_mlflow.pipeline.pipeline_ml.PipelineML method*), 77
only_nodes_with_inputs() (*kedro_mlflow.pipeline.pipeline_ml.PipelineML method*), 77
only_nodes_with_namespaces() (*kedro_mlflow.pipeline.pipeline_ml.PipelineML method*), 77
only_nodes_with_outputs() (*kedro_mlflow.pipeline.pipeline_ml.PipelineML method*), 77
only_nodes_with_tags() (*kedro_mlflow.pipeline.pipeline_ml.PipelineML*

method), 78

P

params (*kedro_mlflow.config.kedro_mlflow_config.MlflowTrackingOptions attribute*), 83
pass_context (*kedro_mlflow.config.kedro_mlflow_config.RequestHeaderProviderOptions attribute*), 84
pipeline_ml_factory() (*in module kedro_mlflow.pipeline.pipeline_ml_factory*), 79
PipelineML (*class in kedro_mlflow.pipeline.pipeline_ml*), 74
pipelines (*kedro_mlflow.config.kedro_mlflow_config.DisableTrackingOptions attribute*), 80
port (*kedro_mlflow.config.kedro_mlflow_config.UiOptions attribute*), 84

R

recursive (*kedro_mlflow.config.kedro_mlflow_config.dictParamsOptions attribute*), 85
request_header_provider (*kedro_mlflow.config.kedro_mlflow_config.MlflowServerOptions attribute*), 82
RequestHeaderProviderOptions (*class in kedro_mlflow.config.kedro_mlflow_config*), 83
RequestHeaderProviderOptions.Config (*class in kedro_mlflow.config.kedro_mlflow_config*), 83
restore_if_deleted (*kedro_mlflow.config.kedro_mlflow_config.ExperimentOptions attribute*), 80
run (*kedro_mlflow.config.kedro_mlflow_config.MlflowTrackingOptions attribute*), 83
run_id (*kedro_mlflow.io.metrics.mlflow_metrics_history_dataset.MlflowMetricsHistoryDataset property*), 67
RunOptions (*class in kedro_mlflow.config.kedro_mlflow_config*), 84
RunOptions.Config (*class in kedro_mlflow.config.kedro_mlflow_config*), 84

S

save() (*kedro_mlflow.io.artifacts.mlflow_artifact_dataset.MlflowArtifactDataset method*), 64
save() (*kedro_mlflow.io.metrics.mlflow_metric_dataset.MlflowMetricDataset method*), 65
save() (*kedro_mlflow.io.metrics.mlflow_metric_history_dataset.MlflowMetricHistoryDataset method*), 66
save() (*kedro_mlflow.io.metrics.mlflow_metrics_history_dataset.MlflowMetricsHistoryDataset method*), 67
save() (*kedro_mlflow.io.models.mlflow_model_local_filesystem_dataset.MlflowModelLocalFileSystemDataset method*), 72
save() (*kedro_mlflow.io.models.mlflow_model_registry_dataset.MlflowModelRegistryDataset method*), 74
save() (*kedro_mlflow.io.models.mlflow_model_tracking_dataset.MlflowMo method*), 70
sep (*kedro_mlflow.config.kedro_mlflow_config.dictParamsOptions attribute*), 85
server (*kedro_mlflow.config.kedro_mlflow_config.KedroMlflowConfig attribute*), 81
setup() (*kedro_mlflow.config.kedro_mlflow_config.KedroMlflowConfig method*), 81
SUPPORTED_SAVE_MODES (*kedro_mlflow.io.metrics.mlflow_metric_dataset.MlflowMetricDatase attribute*), 65

T

tag() (*kedro_mlflow.pipeline.pipeline_ml.PipelineML method*), 78
tags (*kedro_mlflow.config.kedro_mlflow_config.CreateExperimentOptions attribute*), 80
to_nodes() (*kedro_mlflow.pipeline.pipeline_ml.PipelineML method*), 78
to_outputs() (*kedro_mlflow.pipeline.pipeline_ml.PipelineML method*), 78
tracking (*kedro_mlflow.config.kedro_mlflow_config.KedroMlflowConfig attribute*), 81
training (*kedro_mlflow.pipeline.pipeline_ml.PipelineML property*), 78
type (*kedro_mlflow.config.kedro_mlflow_config.RequestHeaderProviderOp attribute*), 84

U

ui (*kedro_mlflow.config.kedro_mlflow_config.KedroMlflowConfig attribute*), 81
UiOptions (*class in kedro_mlflow.config.kedro_mlflow_config*), 84
UiOptions.Config (*class in kedro_mlflow.config.kedro_mlflow_config*), 84

V

validate_assignment (*kedro_mlflow.config.kedro_mlflow_config.KedroMlflowConfig.Conf attribute*), 81