kedro-mlflow

Release 0.7.4

Yolan Honoré-Rougé

Aug 30, 2021

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Introduction

1.1.1 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.

1.1.2 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.

1.1.3 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.

1.1.3.1 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. In my opinion, this file is **production oriented** and is not really intended to use for exploration.
- `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.

  `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.

1.1.3.2 Versioning: Kedro 1 - 1 Mlflow

The `Kedro` `Journal` aims at reproducibility, but is not focused on machine learning. The `Journal` keeps track of two elements:

- the CLI arguments, including *on the fly* parameters. This makes the command used to run the pipeline fully reproducible.
- the `AbstractVersionedDataSet` for which versioning is activated. It consists in copying the data whom `versioned` argument is `True` when the `save` method of the `AbstractVersionedDataSet` is called. This approach suffers from two main drawbacks:
  - the configuration is assumed immutable (including parameters), which is not realistic ni machine learning projects where they are very volatile. To fix this, the `git sha` has been recently added to the `Journal`, but it has still some bugs in my experience (including the fact that the current `git sha` is logged even if the pipeline is ran with uncommitted change, which prevents reproducibility). This is still recent and will likely evolve in the future.
  - there is no support for browsing old runs, which prevents cleaning the database with old and unused datasets, compare runs between each other...

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`.

  `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.

1.1.3.3 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.

> **`Mlflow` is currently the only tool which adresses model serving. This is currently not the top priority for `Kedro`, but may come in the future (through Kedro Server maybe?)**

1.1.3.4 Conclusion: Use Kedro and add Mlflow for machine learning projects

In my opinion, `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.

1.2 Motivation

1.2.1 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 `0.16.6`) 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 do not involve *machine learning*, this plugin is not what you are seeking for ;)

1.2.2 Why should I use kedro-mlflow?

1.2.2.1 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.

1.2.2.2 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.

In the current version (`kedro_mlflow=0.7.4`), `kedro-mlflow` does not 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: *the version `0.7.4` does not need any `MLProject` file to use mlflow inside your Kedro project. As seen in the introduction, this file overlaps with Kedro configuration files.*

CHAPTER

TWO

INTRODUCTION

2.1 Installation guide

2.1.1 Pre-requisites

2.1.1.1 Create a virtual environment

I strongly recommend to use `conda` (a package manager) to create an environment in order to avoid version conflicts between packages.

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 envirpnment is activated:

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

2.1.1.2 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
```

2.1.2 Install the plugin

The current version of the plugin is compatible with `kedro>=0.16.0`. Since Kedro tries to enforce backward compatibility, it will very likely remain compatible with further versions.

2.1.2.1 Install from PyPI

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

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

2.1.2.2 Install from sources

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

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

2.1.3 Check the installation

Type `kedro info` in a terminal to check the installation. If it has succeeded, you should see the following ascii art:

```
 _            _
| | _____  __| |_ __ ___
| |/ / _ \/ _` | '__/ _ \
|   <  __/ (_| | | | (_) |
|_|\_\___|\__,_|_|  \___/
v0.16.<x>

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

Installed plugins:
kedro_mlflow: 0.7.4 (hooks:global,project)
```

The version `0.7.4` of the plugin is installed and has both global and project commands.

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

2.1.4 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.
```

(continues on next page)

(continued from previous page)

```
Commands:
  new  Create a new kedro project with updated template.
```

Note: For now, the `kedro mlflow new` *command is not implemented. You must use* `kedro new` *to create a project, and then call* `kedro mlflow init` *inside this new project.*

2.2 Initialize your Kedro project

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

2.2.1 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 "Getting started" example to make a demo of this plugin out of the box.

2.2.2 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. Those 2 actions are detailled in the following paragraphs.

2.2.2.1 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. **It is mandatory for the plugin to work.**

Set the working directory at the root of your kedro project (i.e. the folder with the `.kedro.yml` file)

```
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>
```

2.2.2.2 Declaring kedro-mlflow hooks

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

Note that you must register the two hooks provided by kedro-mlflow (MlflowPipelineHook and MlflowNodeHook) for the plugin to work.

Declaring hooks through auto-discovery (for kedro>=0.16.4) [Default behaviour]

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 .kedro.yml or your pyproject.toml file.

Declaring hooks through code, in ProjectContext (for kedro>=0.16.0, <=0.16.3)

By declaring mlflow_pipeline_hook and mlflow_node_hook in (src/package_name/run.py) ProjectContext:

```
from kedro_mlflow.framework.hooks import mlflow_pipeline_hook, mlflow_node_hook

class ProjectContext(KedroContext):
    """Users can override the remaining methods from the parent class here,
    or create new ones (e.g. as required by plugins)
    """

    project_name = "<project-name>"
    project_version = "0.16.X" # must be >=0.16.0
    hooks = (
        mlflow_pipeline_hook,
        mlflow_node_hook
    )
```

Declaring hooks through static configuration in .kedro.yml or pyproject.toml [Only for kedro >= 0.16.5 if you have disabled auto-registration]

In case you have disabled hooks for plugin, you can add them manually by declaring mlflow_pipeline_hook and mlflow_node_hook in .kedro.yml :

```
context_path: km_example.run.ProjectContext
project_name: "km_example"
project_version: "0.16.5"
package_name: "km_example"
hooks:
  - <your-project>.hooks.project_hooks
  - kedro_mlflow.framework.hooks.mlflow_pipeline_hook
  - kedro_mlflow.framework.hooks.mlflow_node_hook
```

Or by declaring mlflow_pipeline_hook and mlflow_node_hook in pyproject.toml :

```
# <your-project>/pyproject.toml
[tool.kedro]
```

(continues on next page)

(continued from previous page)

```
hooks=["kedro_mlflow.framework.hooks.mlflow_pipeline_hook",
       "kedro_mlflow.framework.hooks.mlflow_node_hook"]
```

2.3 Migration guide

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

2.3.1 Migration from 0.5.0 to 0.6.0

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

2.3.2 Migration from 0.4.1 to 0.5.0

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 `MlflowModelLoggerDataSet` 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`.

2.3.3 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.framework.context 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
```

2.3.4 Migration from 0.3.0 to 0.4.0

2.3.4.1 Catalog entries

Replace the following entries:

2.3.4.2 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()
)
```

2.3.4.3 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.

CHAPTER

THREE

INTRODUCTION

3.1 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 the versioning 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 capabiltiies 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.

3.2 Example project

3.2.1 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=3.6.8 --yes
conda activate km_example
pip install kedro-mlflow==0.7.4
```

3.2.2 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.

3.2.2.1 Installation with kedro>=0.16.3

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:
================
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
```

3.2.2.2 Installation with 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.
```

(continues on next page)

(continued from previous page)

```
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
```

3.2.3 Install dependencies

Move to the project directory:

```
cd km-example
```

Install the project dependencies (**Warning: Do not use `kedro install` commands** **does not install the packages in your activated environment**):

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

3.3 First steps with the plugin

3.3.1 Initialize kedro-mlflow

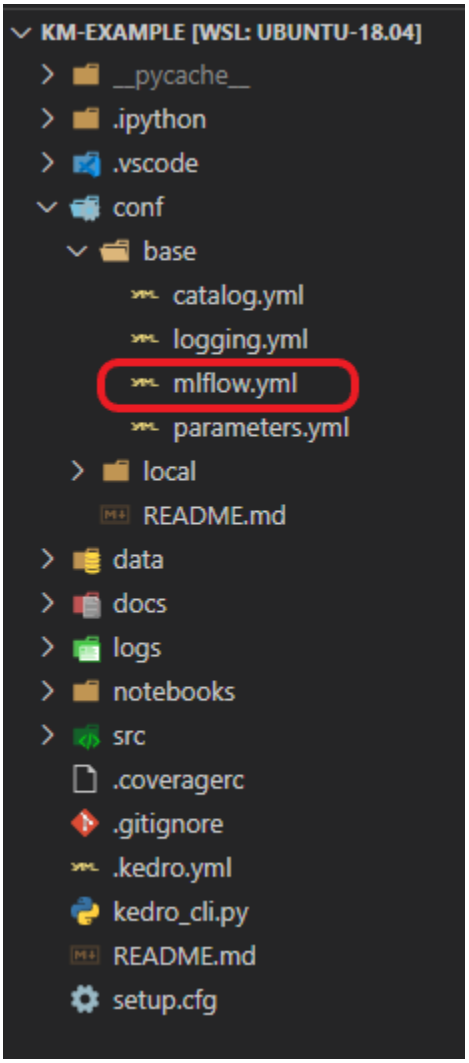
First, you need to initialize your project and add the plugin-specific configuration file with this command:

```
kedro mlflow init
```

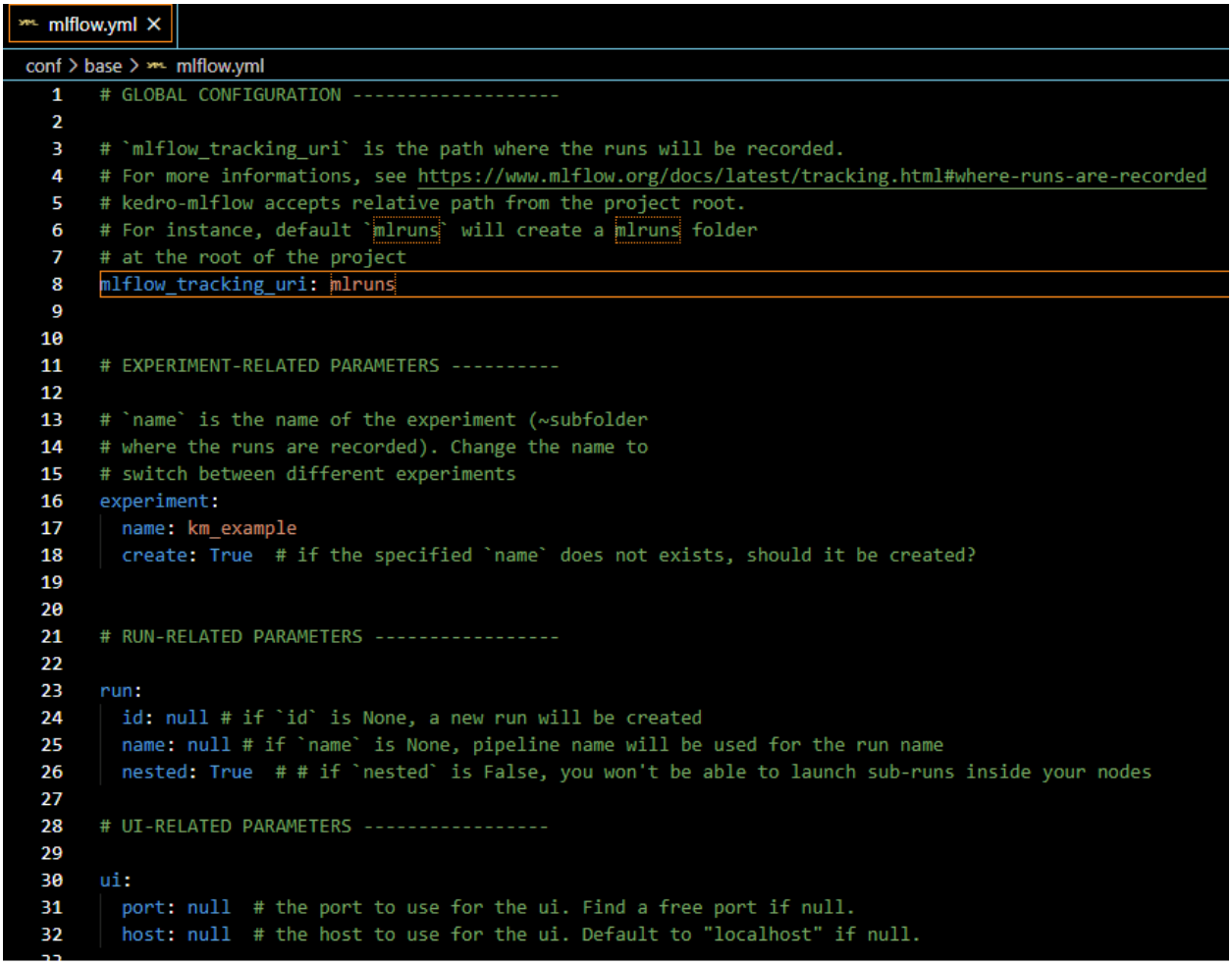
You will see the following message:

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

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



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



3.3.2 Run the pipeline

Open a new command and launch

```
kedro run
```

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

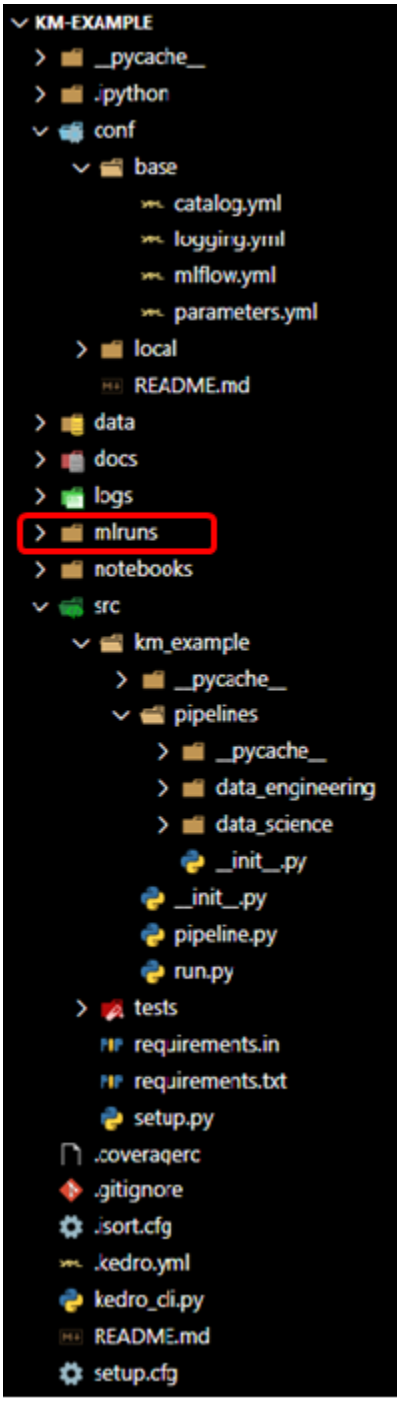
```
2020-07-13 21:29:24,939 - kedro.versioning.journal - WARNING - Unable to git describe␣
↪path/to/km-example
2020-07-13 21:29:25,401 - kedro.io.data_catalog - INFO - Loading data from `example_iris_
↪data` (CSVDataSet)...
2020-07-13 21:29:25,562 - kedro.io.data_catalog - INFO - Loading data from␣
↪`params:example_test_data_ratio` (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)...
```

(continues on next page)

(continued from previous page)

```
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`␣
↪(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:


KM-EXAMPLE
__pycache__
.python
conf
base
catalog.yml
logging.yml
mlflow.yml
parameters.yml
local
README.md
data
docs
logs
mlruns
notebooks
src
km_example
__pycache__
pipelines
__pycache__
data_engineering
data_science
__init__.py
__init__.py
pipeline.py
run.py
tests
requirements.in
requirements.txt
setup.py
.coveragerc
.gitignore
.isort.cfg
.kedro.yml
kedro_cli.py
README.md
setup.cfg

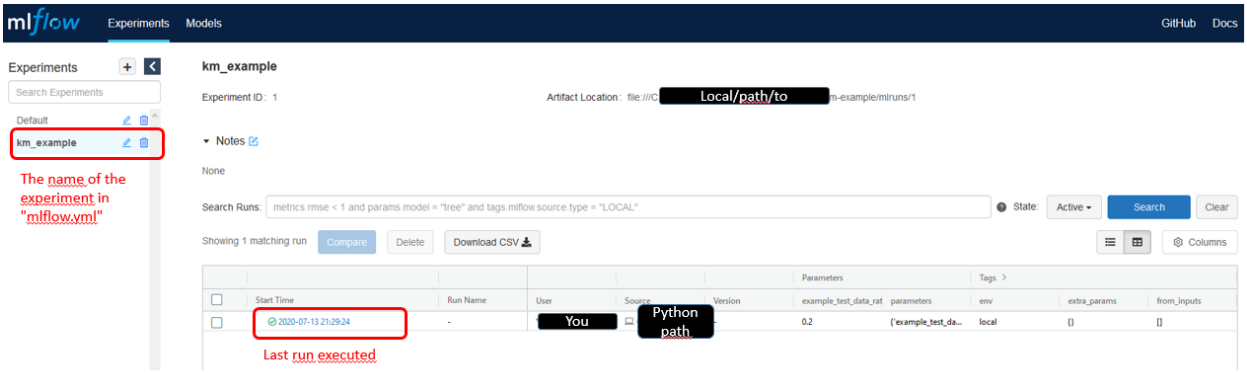
3.3.3 Open the UI

Launch the ui:

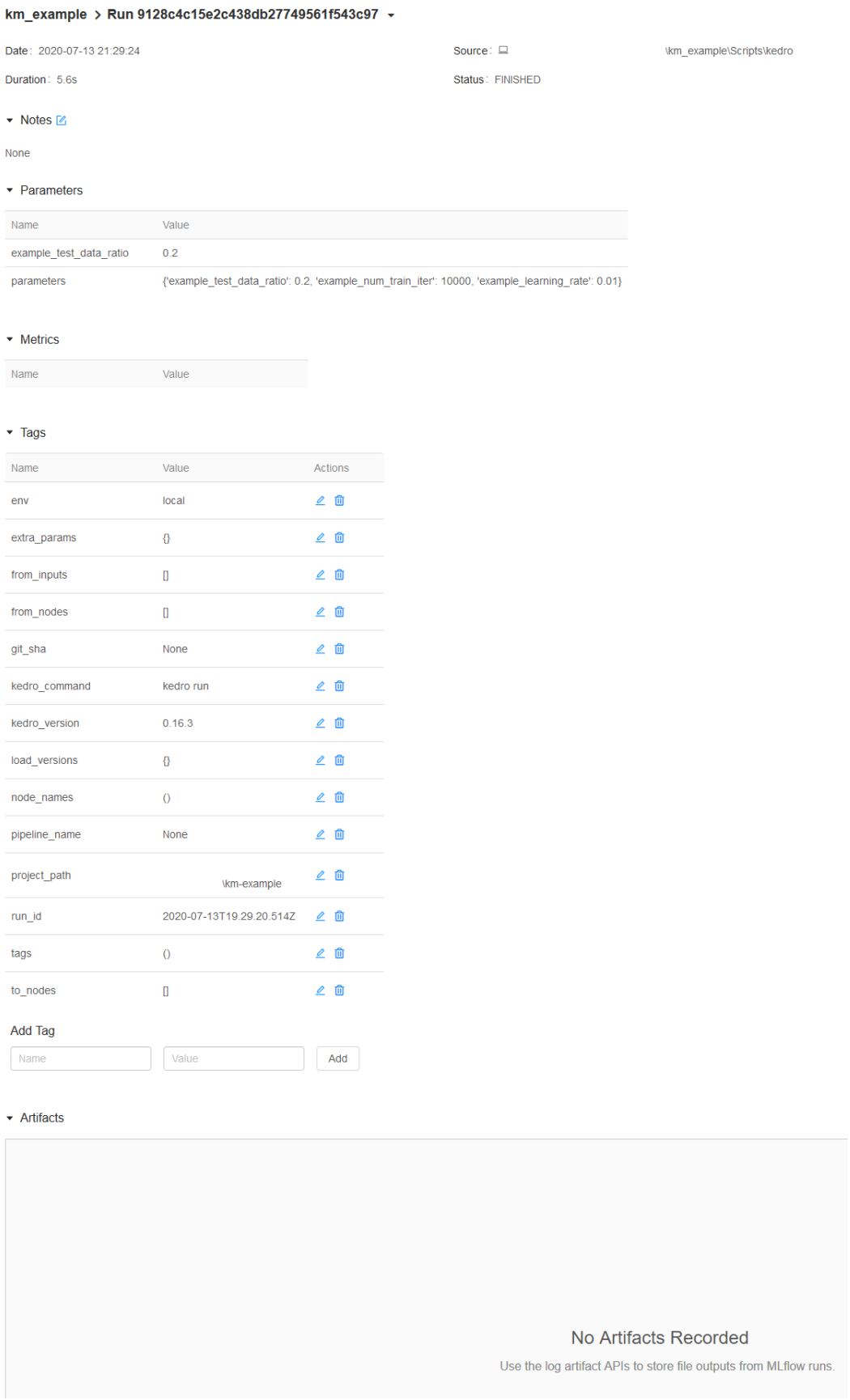
```
kedro mlflow ui
```

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.

3.3.3.1 Parameters versioning

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.

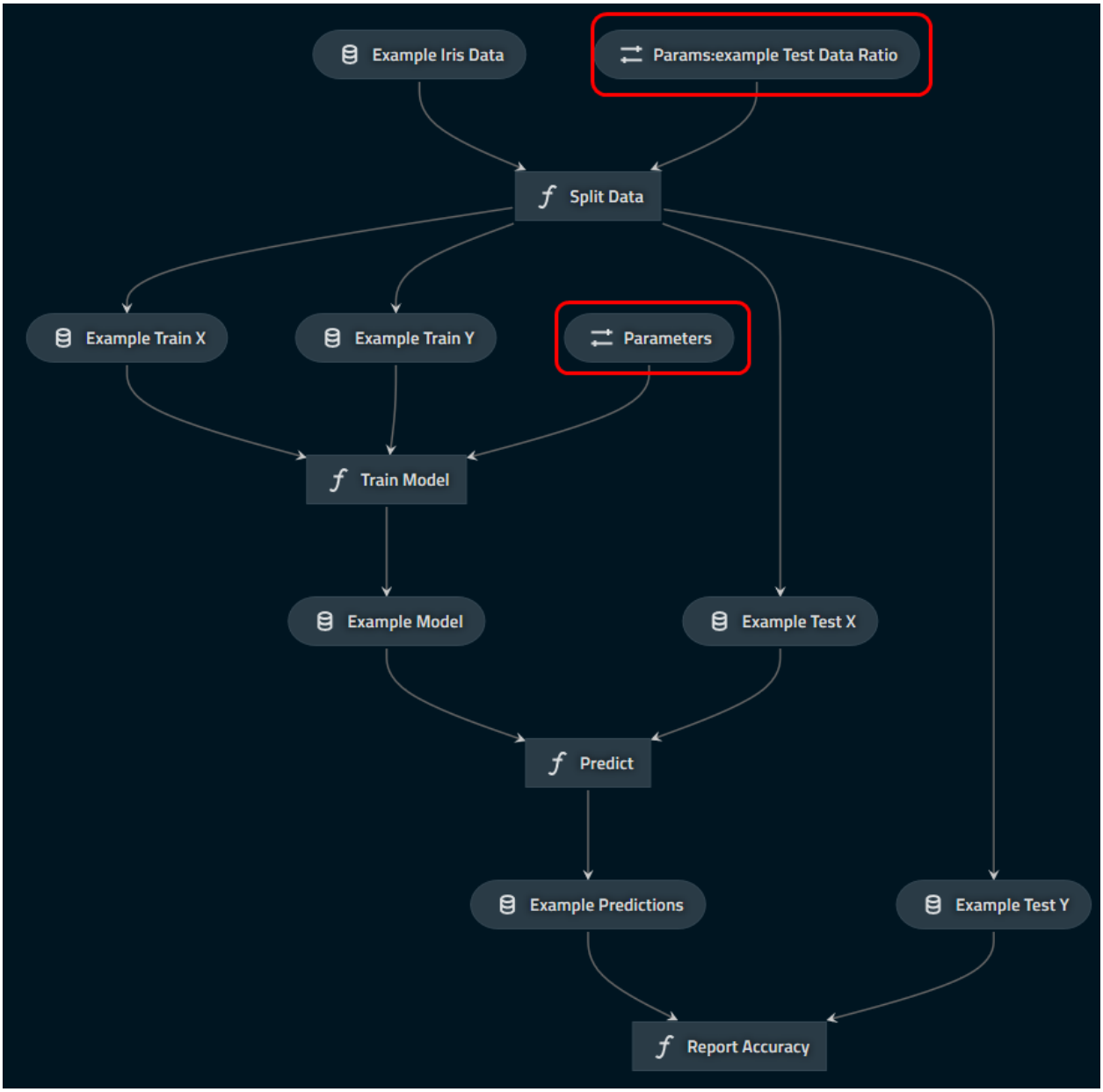
You can see that these are effectively the registered parameters in the pipeline with the `kedro-viz` plugin:

```
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).

3.3.3.2 Journal information

The informations provided by the `Kedro`'s `Journal` are also recorded as `tags` in the mlflow ui in order to make reproducible. In particular, the exact command used for running the pipeline and the kedro version used are stored.

3.3.3.3 Artifacts

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:

```
# 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:

```
# 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
  data_set:
    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 `MlflowModelLoggerDataSet`

3.3.4 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
- can help to create standardize pipelines for deployment in production

CHAPTER

FOUR

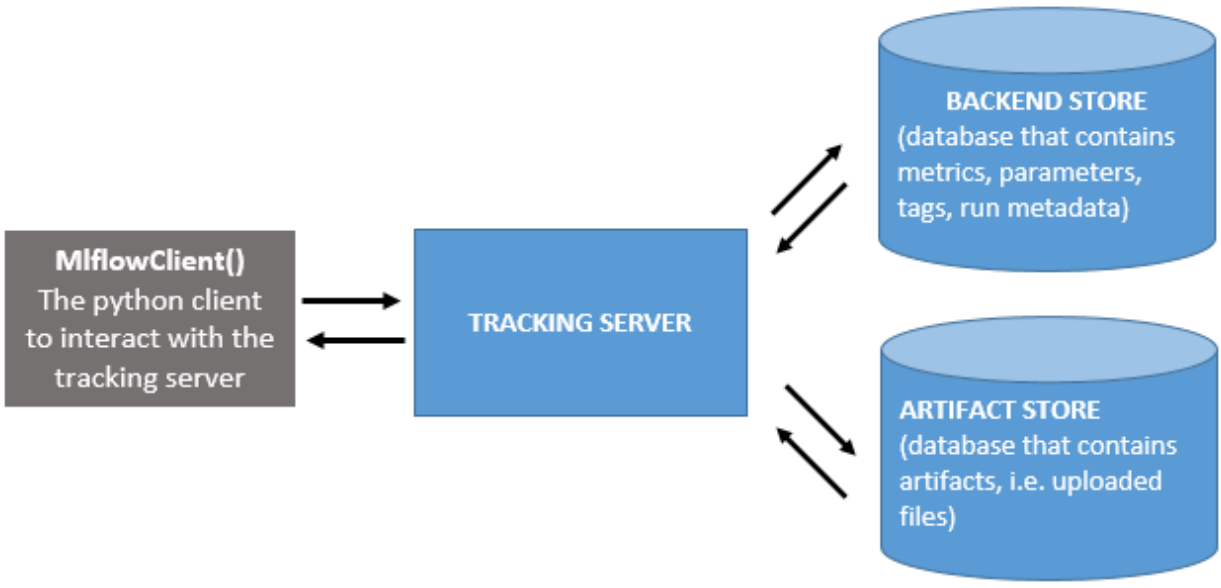
INTRODUCTION

4.1 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.

4.1.1 Context: mlflow tracking under the hood

Mlflow is composed of four modules which are described in the introduction section. The ain 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.

4.1.2 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.

4.1.2.1 Configure the tracking server

`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 `mlruns`. This will create a `mlruns` folder locally at the root of your kedro project and enable you to use the plugin without any setup of a mlflow tracking server.

Unlike mlflow, `kedro-mlflow` allows the `mlflow_tracking_uri` to be a relative path. It will convert it to an absolute uri automatically.

```
mlflow_tracking_uri: mlruns
```

This is the **only mandatory key in the `mlflow.yml` file**, but there are many others described hereafter that provide fine-grained control on your mlflow setup.

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:

```
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`:

```
credentials: my_mlflow_credentials
```

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

4.1.2.2 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:

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

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

4.1.2.3 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 witht the `experiment` key:

```
experiment:
  name: <your-experiment-name>  # by default, the name of your python package in your␣
↪kedro project
  create: True  # if the specified `name` does not exists, should it be created?
```

Note that by default, mlflow crashes if you try to start a run while you have not created the experiment first. `kedro-mlflow` has a `create` key (`True` by default) which forces the creation of the experiment if it does not exist. Set it to `False` to match mlflow default value.

4.1.2.4 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:

```
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
```

- 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`

4.1.2.5 Configure the hooks

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:

```
hooks:
  node:
    flatten_dict_params: 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 oin mlflow.
```

(continues on next page)

(continued from previous page)

```
    long_parameters_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_dict_params` 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.

4.1.2.6 Configure the user interface

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

```
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`.

4.2 Parameters versioning

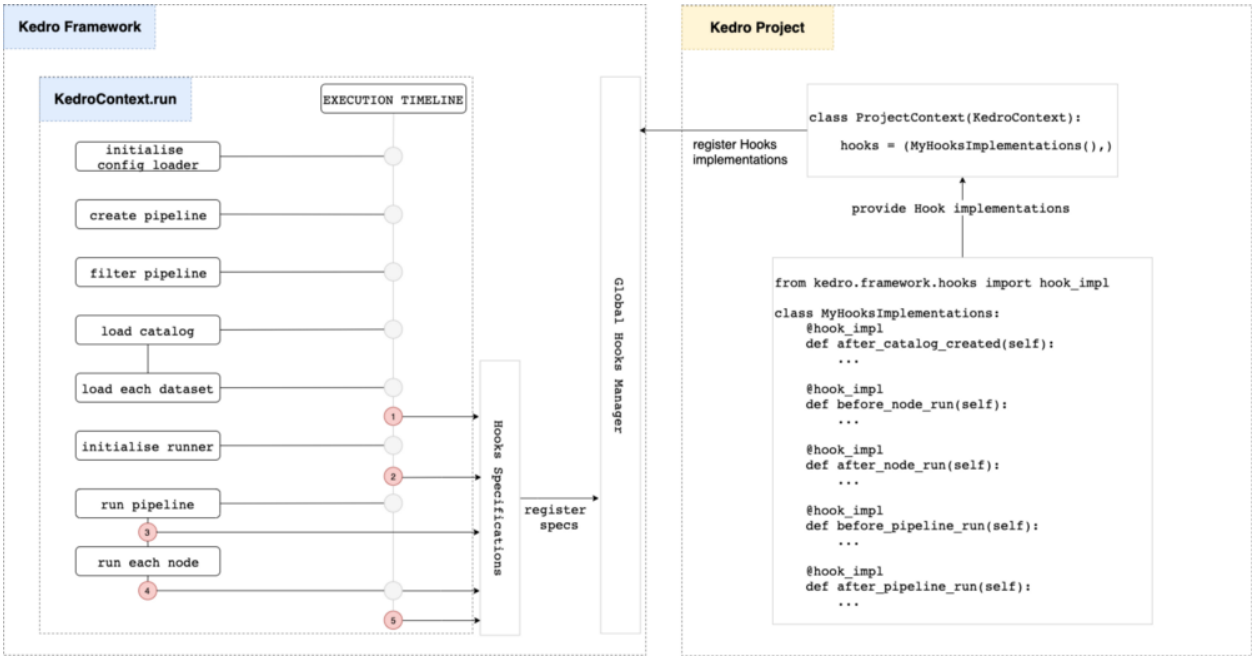
4.2.1 Automatic parameters versioning

Parameters versioning is automatic when the `MlflowNodeHook` is added to the hook list of the `ProjectContext`. In `kedro-mlflow==0.7.4`, 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.

4.2.2 How does `MlflowNodeHook` operates under the hood?

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



The `MlflowNodeHook` 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.

4.2.3 Frequently Asked Questions

4.2.3.1 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**.

4.2.3.2 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).

4.2.3.3 How can I register a parameter if I use a `TemplatedConfigLoader`?

If you use a `TemplatedConfigLoader` to enable dynamic parameters contruction at runtime or dependency between configuration files, and if we assume your `src/<project-name>/run.py` file looks like:

```
from kedro.config import TemplatedConfigLoader  # new import
from datetime import date

class ProjectContext(KedroContext):
    def _create_config_loader(self, conf_paths: Iterable[str]) -> TemplatedConfigLoader:
        return TemplatedConfigLoader(
            conf_paths,
            globals_pattern="*globals.yml",  # read the globals dictionary from project␣
↪config
```

(continues on next page)

(continued from previous page)

```
            globals_dict={  # extra keys to add to the globals dictionary, take␣
↪precedence over globals_pattern
            execution_date: date.today()
            },
        )
```

Then **you need to add this entry in your `conf/<env>/parameters`** to ensure that the parameter will be properly recorded:

```
execution_date: ${execution_date}
```

4.3 Versioning Kedro DataSets

4.3.1 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

4.3.2 How to version 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`:

```
my_dataset_to_version:
    type: pandas.CSVDataSet
    filepath: /path/to/a/destination/file.csv
```

You can change it to:

```
my_dataset_to_version:
    type: kedro_mlflow.io.artifacts.MlflowArtifactDataSet
    data_set:
        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.

4.3.3 Frequently asked questions

4.3.3.1 Can I pass extra parameters to the `MlflowArtifactDataSet` for finer control?

The `MlflowArtifactDataSet` takes a `data_set` 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 `data_set` argument:

```
my_dataset_to_version:
    type: kedro_mlflow.io.artifacts.MlflowArtifactDataSet
    data_set:
        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 data_set
```

4.3.3.2 Can I use the `MlflowArtifactDataSet` in interactive mode?

Like all Kedro `AbstractDataSet`, `MlflowArtifactDataSet` is callable in the python API:

```
from kedro_mlflow.io.artifacts import MlflowArtifactDataSet
from kedro.extras.datasets.pandas import CSVDataSet
csv_dataset = MlflowArtifactDataSet(data_set={"type": CSVDataSet, # either a string
↪"pandas.CSVDataSet" or the class
                                    "filepath": r"/path/to/a/local/destination/file.csv
↪"})
csv_dataset.save(data=pd.DataFrame({"a":[1,2], "b": [3,4]}))
```

4.3.3.3 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 server is the only additional configuration you need to send your datasets to this remote server. Note that you still need to specify a **local** path for the underlying dataset, mlflow will take care of the upload to the server by itself.

You can refer to this issue for further details.

In `kedro-mlflow==0.7.4` you must configure these elements by yourself. Further releases will introduce helpers for configuration.

4.3.3.4 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
    data_set:
        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
```

4.3.3.5 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
    data_set:
        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 with the existing artifact. It must␣
↪be named "file.csv"
```

4.3.3.6 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
    data_set:
        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.
```

4.4 Version model

4.4.1 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.

4.4.2 How to track models using MLflow in Kedro project?

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

- the `MlflowModelLoggerDataSet` is used to load from and save to from the mlflow artifact store. It uses optional `run_id` argument to load and save from a given `run_id` which must exists in the mlflow server you are logging to.
- the `MlflowModelSaverDataSet` is used to load from and save to a given 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 `MlflowModelLoggerDataSet`, it will be saved during training in 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 `MlflowModelSaverDataSet` 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:

```
my_sklearn_model:
    type: kedro_mlflow.io.models.MlflowModelLoggerDataSet
    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.

4.4.3 Frequently asked questions?

4.4.3.1 How is it working under the hood?

For `MlflowModelLoggerDataSet`

During save, a model object from node output is logged to mlflow using `log_model` function of the specified `flavor`. It is logged in the `run_id` run if specified and if there is no active run, else in the currently active mlflow run. If the `run_id` is specified and there is an active run, 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 `MlflowPipelineHook` creates a new run before each pipeline run.

During load, the model is retrieved from the `run_id` if specified, else it is retrieved from the mlflow active run. If there is no mlflow active run, the loading fails. This will never happen if you are using the `kedro run` command, because the `MlflowPipelineHook` creates a new run before each pipeline run.

For `MlflowModelSaverDataSet`

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.

4.4.3.2 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 offer more control. We haven't tested the integration in `kedro-mlflow` of this second workflow extensively, and it should be used with caution.

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

4.4.3.3 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`:

```
sklearn_model:
    type: kedro_mlflow.io.artifacts.MlflowArtifactDataSet
    data_set:
        type: kedro_mlflow.io.models.MlflowModelSaverDataSet
        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.

4.5 Version metrics

4.5.1 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".

4.5.2 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.
- `MlflowMetricsDataSet` (**DEPRECATED**). It is a wrapper around a dictionary with metrics which is returned by node and log metrics in MLflow.

4.5.2.1 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:

```
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
```

Note: Beware: 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.
```

4.5.2.2 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},
```

(continues on next page)

(continued from previous page)

```
            {"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}) # 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:

```
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"}
```

4.5.2.3 Saving several metrics with their entire history with `MlflowMetricsDataSet`

Warning: This class is deprecated and will be removed soon. Use `MlflowMetricDataSet` or `MlflowMetricHistoryDataSet` instead.

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

```
my_model_metrics:
    type: kedro_mlflow.io.metrics.MlflowMetricsDataSet
```

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 `MlflowPipelineHook`, 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:

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

4.5.3 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:

```
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.MlflowMetricsDataSet` can take metrics as:

- `Dict[str, key]`
- `List[Dict[str, key]]`

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

```
my_model_metrics:
    type: kedro_mlflow.io.metrics.MlflowMetricsDataSet
```

Within a kedro run, the `MlflowPipelineHook` 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.MlflowMetricsDataSet
    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",
    ))
```

4.6 Opening the UI

4.6.1 The mlflow user interface

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

4.6.2 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.

CHAPTER

FIVE

INTRODUCTION

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

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

5.1.1.1 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.
- **disponibility 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.

5.1.1.2 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 ML 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 monitor data which go through 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.**

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

5.1.2.1 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…

5.1.2.2 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...).

5.1.2.3 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.

5.1.2.4 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 helpers `pipeline_ml_factory`) class which binds the `training` and `inference` pipeline, 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.

5.1.2.5 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.

5.1.2.6 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

5.2 The components of a machine learning application

5.2.1 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

5.2.2 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:

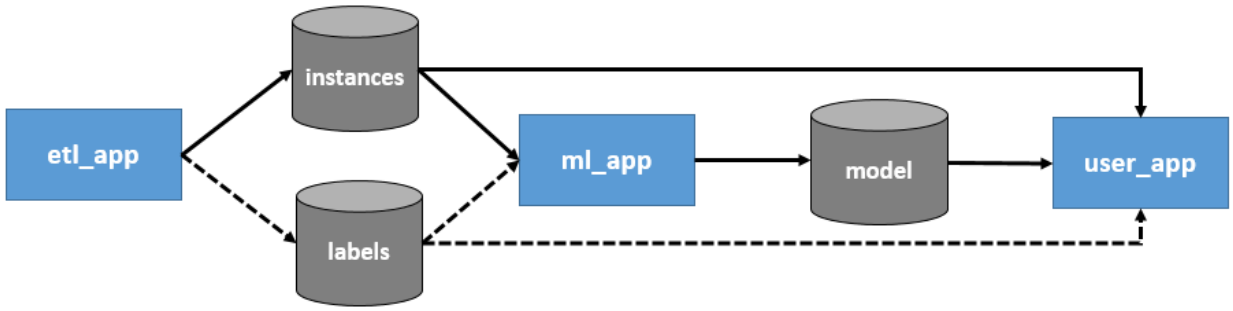
- Each app development / deployment is likely under the responsibility of different people / teams.
- Each app has a different development lifecyle. It implies that development can be parallelized, and releasing one app to fix a bug does not imply to release the other ones. If your training pipeline 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.

5.2.3 Apps development lifecycle in a machine learning project

5.2.3.1 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.

5.2.3.2 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.

5.2.3.3 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.**

5.2.3.4 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.

5.3 kedro-mlflow mlops solution

5.3.1 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

5.3.2 Enforcing these principles with a dedicated tool

5.3.2.1 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 DataCatalogs 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:

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

class ProjectHooks:
    @hook_impl
    def register_pipelines(self) -> Dict[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.

5.3.2.2 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:

```
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_ml=pipeline_training,
    catalog=catalog
)

mlflow.pyfunc.log_model(
    artifact_path="model",
    python_model=kedro_model,
    artifacts=artifacts,
    conda_env={"python": "3.7.0", dependencies: ["kedro==0.16.5"]},
    model_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!

5.3.2.3 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 `MlflowPipelineHook.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 `MlflowModelLogger` for further reuse.

5.3.2.4 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 `MlflowModelLoggerDataSet` class which can be used int the `catalog.yml` file:

```
# catalog.yml

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>
```

5.4 Project example

5.4.1 5 mn summary

If you don't want to read the entire explanations, here is a summary:

1. Install `kedro-mlflow` `MlflowPipelineHook` (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 `hooks.py`:

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


...
```

(continues on next page)

(continued from previous page)

```
class ProjectHooks:
    @hook_impl
    def register_pipelines(self) -> Dict[str, Pipeline]:

        ...

        ml_pipeline = create_ml_pipeline()
        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",
            model_name="kedro_mlflow_tutorial",
            conda_env={
                "python": 3.7,
                "dependencies": [f"kedro_mlflow_tutorial=={PROJECT_VERSION}"],
            },
            model_signature="auto",
        )

        ...

        return {
            "training": training_pipeline_ml,
            ...
        }
```

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

```
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)
```

4. Launch your training pipeline:

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

The inference pipeline will *automagically* be logged as a mlflow model at the end!

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`:

```
# catalog.yml

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>
```

5.4.2 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.

CHAPTER

SIX

INTRODUCTION

6.1 New DataSet

6.1.1 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:

```
my_dataset_to_version:
    type: kedro_mlflow.io.artifacts.MlflowArtifactDataSet
    data_set:
        type: pandas.CSVDataSet  # or any valid kedro DataSet
        filepath: /path/to/a/local/destination/file.csv
```

or with additional parameters:

```
my_dataset_to_version:
    type: kedro_mlflow.io.artifacts.MlflowArtifactDataSet
    data_set:
        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 data_set
    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:

```
from kedro_mlflow.io.artifacts import MlflowArtifactDataSet
from kedro.extras.datasets.pandas import CSVDataSet
csv_dataset = MlflowArtifactDataSet(data_set={"type": CSVDataSet,
                                    "filepath": r"/path/to/a/local/destination/file.csv
↪"})
csv_dataset.save(data=pd.DataFrame({"a":[1,2], "b": [3,4]}))
```

6.1.2 Metrics DataSets

6.1.2.1 MlflowMetricDataSet

The MlflowMetricDataSet is documented here.

6.1.2.2 MlflowMetricHistoryDataSet

The MlflowMetricHistoryDataSet is documented here.

6.1.3 Models DataSets

6.1.3.1 MlflowModelLoggerDataSet

The MlflowModelLoggerDataSet accepts the following arguments:

- flavor (str): Built-in or custom MLflow model flavor module. Must be Python-importable.
- run_id (Optional[str], optional): MLflow run ID to use to load the model from or save the model to. It plays the same role as "filepath" for standard mlflow datasets. Defaults to None.
- artifact_path (str, optional): the run relative path tothe model.
- 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 `log_model` function from specified `flavor`. Defaults to None.

You can either only specify the flavor:

```
from kedro_mlflow.io.models import MlflowModelLoggerDataSet
from sklearn.linear_model import LinearRegression

mlflow_model_logger=MlflowModelLoggerDataSet(flavor="mlflow.sklearn")
mlflow_model_logger.save(LinearRegression())
```

Let assume that this first model has been saved once, and you xant to retrieve it (for prediction for instance):

```
mlflow_model_logger=MlflowModelLoggerDataSet(flavor="mlflow.sklearn", run_id=<the-model-
↪run-id>)
my_linear_regression=mlflow_model_logger.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:

```
mlflow_model_logger=MlflowModelLoggerDataSet(
    flavor="mlflow.sklearn",
     run_id=<the-model-run-id>,
     save_args={
         "conda_env": {"python": "3.7.0", , "dependencies": ["kedro==0.16.5"]},
          "input_example": data.iloc[0:5,:]
```

(continues on next page)

(continued from previous page)

```
        }
    )
mlflow_model_logger.save(LinearRegression().fit(data))
```

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

```
my_model:
    type: kedro_mlflow.io.models.MlflowModelLoggerDataSet
    flavor: "mlflow.sklearn"
    run_id: <the-model-run-id>,
    save_args:
        conda_env:
            python: "3.7.0"
            dependencies:
                - "kedro==0.16.5"
```

6.1.3.2 `MlflowModelSaverDataSet`

The `MlflowModelLoggerDataSet` 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 MlflowModelLoggerDataSet, but that you specify a filepath instead of a `run_id`:

```
from kedro_mlflow.io.models import MlflowModelLoggerDataSet
from sklearn.linear_model import LinearRegression

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

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

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

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

```
my_model:
    type: kedro_mlflow.io.models.MlflowModelSaverDataSet
```

(continues on next page)

(continued from previous page)

```
    flavor: mlflow.sklearn
    filepath: path/to/where/you/want/model
    version: <valid-kedro-version>
```

6.2 Hooks

This package provides 2 new hooks.

6.2.1 MlflowPipelineHook

This hook :

1. manages mlflow settings at the beginning and the end of the run (run start / end).
2. log useful informations for reproducibility as `mlflow tags` (including kedro `Journal` information and the commands used to launch the run).
3. register the pipeline as a valid `mlflow model` if *it is a `PipelineML` instance*

6.2.2 MlflowNodeHook

This hook:

1. must be used with the `MlflowPipelineHook`
2. autolog nodes parameters each time the pipeline is run (with `kedro run` or programatically).

6.3 Pipelines

6.3.1 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"))
```

(continues on next page)

(continued from previous page)

```
    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 `MlflowPipelineHook` 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`

Note: If you want to log a `PipelineML` object in `mlflow` programatically, you can use the following code snippet:

```
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_ml=pipeline_training,
            catalog=catalog
        ),
    artifacts=artifacts,
    conda_env={"python": "3.7.0", , "dependencies": ["kedro==0.16.5"]},
    model_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.

```
KedroPipelineModel(
            pipeline_ml=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 fo each dataset.

6.4 Cli commands

6.4.1 `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!

6.4.2 `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:

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

then

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

will open the ui on port 5002.

6.5 Configuration

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

More details are coming soon.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search