# Tutorial: Basics of MLFlow

## Preface

This tutorial aims at introducing the most important features of MLFlow with the aid of a sample project. The MLFlow version used was 1.10. Examples are stored in this GitHub repository[[1]](#footnote-1). The goal of the project was to forecast the sales of Walmart with the data provided in the M5 Kaggle competition.[[2]](#footnote-2) Users looking for specific features or who are interested to know about all possible configurations of a command are advised to look at the extensive documentation[[3]](#footnote-3). Furthermore, note that additional examples using typical machine learning packages can be found in the MLFlow GitHub repository[[4]](#footnote-4).

## Basic Idea

MLFlow is an “open source platform for managing the end-to-end machine learning lifecycle”. The aim of MLFlow is to make performance tracking, sharing, storing as well as tracking the development of models simpler by providing a structure in which these tasks can be organized. For each of the tasks MLFlow has one component, namely: Tracking, Projects, Models and Registry. In the following each component is introduced within the above-mentioned example.

## Tracking

MLFlow tracking allows to store modelling results in runs. A run is one pass threw the model in which parameters, metrics and so-called artifacts are stored. Parameters store model parameter values, metrics the values of accuracy metrics and artifacts are left for any datatype that is of further interest. Artifacts are mostly graphs or generated data saved as csv. In general artifacts can be any information from a specific run a user wants to save. A full list of all data file types supported by artifacts are can be found here[[5]](#footnote-5).

A run is part of an experiment. This way experiments help to organize runs. If for example multiple runs are for the same model with varying starting parameters, it makes sense to store them in one experiment. By then logging in starting values as parameters and accuracy measures as metrics one can find out which starting parameters made the model preform. The visualization of these results is in the MLFlow GUI which, run locally, is a locally hosted server showing all experiments, there runs and any logged information.

Basic commands of tracking inside the python script of the model:

1. **Creating a run**

with mlflow.start\_run():

* The actual computation of the model including fitting the model as well as forecasting testing data is done within the run loop
* Logging parameters, metrics and artifacts is also contained within loops
* the “with” makes the coding simpler as otherwise users would have to define the start and ending of the run

1. **Logging parameters**

mlflow.log\_param(key="Parameter\_Name", value=Parameter\_Value)

1. **Logging metrics**

mlflow.log.metric(key=“Metric\_Name”, value=Metric\_Value, step=Step\_Value)

* “steps” are optional, they give an extra option by allowing a metric to have multiple values depending on their step. A typical use of steps is to for the steps to be the forecasting horizon. Thus, one can store an accuracy metric for different horizons. Another advantage of steps is that in the GUI the metrics can directly be plotted dependent on the steps.

1. **Logging Artifacts**

mlflow.log\_artifact("./Artifact\_File\_Name.Typ", “folder\_name”)

* before saving a file as an artifact it needs to be created and saved in the current working directory. Thus, one first creates a files, stores it in the working directory and then MLFlow transfers the file to the artifact folder of the current run.
* Unde folder name users can specify a folder to be created in the artifact folder and in which the artifact should be logged. Multiple artifacts can be logged in one folder.

1. **Setting an experiment**

mlflow.set\_experiment("Experiment\_Name")

* The experiment can either be set in the run or before.
* In Jupiter Notebooks the experiment can also be defined in a prior cell. Once it is defined all runs of that kernel are saved in the experiment.

1. **Saving the runs in a certain directory**

mlflow.set\_tracking\_uri("file:///Local\_Directory")

* In this example I use a local directory, thus the appreciation “file:///” is set at the beginning. Note that there are other options as runs can be stored on Databases using various sql flavors, on http servers as well as on Databricks
* The directory should lead to the folder where an mlrun folder should be created or already exists. It can also lead directly to an existing mlrun folder
* Windows users should note that the local directory should not include the “C:/” at the beginning but rather start at the typical “User” folder. Further they should note that all slashes are directed towards the right “/”

1. **Starting the GUI**

get\_ipython().system\_raw("mlflow ui --backend-store-uri file:/// Local\_Directory --port 5000 &")

* The above command starts the MLFlow server inside a jupiter notebook. This leads to an issue. The notebook cannot be used further as the local GUI server runs as long as the command runs. Thus a simpler approach would be to run the command directly in the anaconda prompt powershell as follows:

mlflow ui --backend-store-uri file:///Local\_Directory

* The directory is specified identically to the specifications above in saving the runs
* The GUI only shows experiments from the specified directory
* More information on how to set up the GUI in jupiter notebooks can be found in the documentation and the following example[[6]](#footnote-6)

## The GUI

The core of MLFlow is its GUI. It provides an intuitive interface for users to grasp information about there projects. After creating and storing runs as well as starting the GUI as explained above the locally run GUI Server can be accessed threw the browser by the following link: http://localhost:5000/#/

The GUI is self-explanatory. Thus, I advise anyone who wants to work with it, to simply log in a few runs within different experiments and explore it. A Detailed Tutorial including screenshots can be found here.[[7]](#footnote-7)

## The folder structures

Finally, in order to better understand MLFlow it is sensible to take examine the underlying folder structure and its interaction with the GUI. To save information MLFlow creates a folder called mlrun. It contains all information used by the GUI. On the first level it is composed of folders specifying the experiments as well as a trash folder. The experiment are simply enumerated folders, starting at “0” for the default experiment. The trash folder is where experiments are moved to if they are deleted within the GUI. Inside each experiment one can find a folder per logged run. These are named after the runs URI. Next to these folders there is also a yaml file called meta. It contains the meta information about the experiment. This includes the location, number, name and life cycle stage of the experiment. Experiments can be deleted in the GUI by pressing the trash icon next to there name. Then the entire experiment folder is moved into the trash folder.

Inside each run folder one can find subfolders for the parameters, metrics, artifacts and tags as well as a meta yaml file. The parameter and metric folders contain text files for each logged parameter and metric. The artifact folder contains all data files in the format they were logged such as png or csv. The tag folder saves all tags that are added to a run.[[8]](#footnote-8) In analogy to the meta file of the experiment the meta file of each run contains the name, experiment id, artifact URI and the lifecycle stage. In addition her to it contains additional information such as the user who ran it and the time it was started. In an essence it contains all the information about the run visible in the GUI. The lifecycle stage is either active or deleted and can be set to the latter by deleting it in the GUI. To recover it simply change the status to active in the yaml file.

If experiments are trashed they can be pulled back into the mlrun folder and then show up in GUI after refreshing. New and reactivated runs are also visible after refresh. Thus, the GUI can be used in parallel to working on a model. Note that the status of experiments remains active even if they are moved to the trash. For runs however if they are deleted their stage changes from active to deleted.

## 

Is a format to package code and make results reproducible. It is basically a convention in which to organize and describe code. In addition, it includes an API allowing to run the projects. This makes reproducing results simple and allows to include projects into workflows.

A project is a directory of files stored either locally or in a repository such as GitHub. It is made up by an MLProject text file, a Conda environment, the python script containing the model and any other files required to run the model. These can be additional scripts with required functions or a csv file containing training data.

The MLProject file is the head of the project. It defines of the name of the project, the environment file and the model script. Furthermore, it also specifies the parameters that the project requires to run the model. The specification includes the parameter names as well as their type which is either a string or float. Note that lists cannot be passed on, thus if multiple parameters are passed on, they are required to be defined individually. Finally, it includes the command for the command line which is to run the python script depending on all included parameters.

The Conda environment defines the environment required to run the python script. Each time a new project is downloaded, or a project environment is altered and the project in question is run again, anaconda creates a new environment to run the project script inside.

The python script containing the model needs to be adjusted in comparison to running the script directly. For one, the parameters defined in the MLProject run need to be passed on. Here for parameters are accessed with the sys package function sys.argv[Number]. It gives the script access to a list containing the command-line arguments. The list is basically the command past in the MLProject file. Note that the first argument is always the script name. Thus, the first parameter is accessed with sys.argv[1], the second with sys.argv[2] and so on. Note that it is sensible to convert these new variables to strings, floats or integers depending on the model in question. The second adjustment to make is to include the run and all parameter definitions inside a name equal to main condition. This simply set the starting point of the program and is a basic tool in python.

Basic command of project with which a project can be run inside a python environment:

1. **Running a project**

mlflow.projects.run (uri='URI\_to\_Repository',

parameters={'Parameter\_name': Parameter\_value})

* The URI can also lead to a local folder
* Parameters are passed on in a dictionary.
* After running the code warnings should show that the data is fetched, the Conda environment is created, the command running the script is executed and finally that the run was successfully executed

The runs created by a project are saved in the mlrun folder of the current or specified directory. To add the runs to an experiment either define the experiment in the python script in the repository or define it prior to running the project. In essence the rules about saving runs and setting experiments that apply to local runs also apply to runs of projects.

A final practical note. As MLFlow pulls the script and environment the possible errors returned by python are not very helpful in bug fixing, thus it is sensible to first write the script locally, then to create a specific environment for it and to run the script inside the environment with the command line. Once it works simply export the environment as a yaml file and upload it with the model python script.

## Model

The Model component allows users to store trained models. Its purpose is to make models available so that new observational data can be passed threw in order to obtain forecasts. The advantage lies in the simplicity as forecasts can be obtained with one command and without fitting the model each time a new prediction is required. In terms of model deployment, it is possible to access models with online Inference via Rest API or offline with batch inference.

A MLModel is stored in a directory and contains an MLModel file, the python model in pickle format, the conda environment in yaml format and an artifacts folder with all additional files required to run the model. The MLModel file defines the model flavor[[9]](#footnote-9). It contains the names and URIs of all required artifacts, the name of the conda environment and the cloudpickle version used to store the model.

Storing a model as a MLModel depends on its flavor. Most models which are part of a popular ML library can be logged as standardized flavors in MLFlow. For these models there is a simple log function that can be used to store them. It is basically the log model function introduced below, but for the specific flavor of the model in question. This is very convenient and a big advantage of MLModel as one simple command stores a complex model. Custom models or ones from non-standardized packages can also be stored. In our example we use a custom model to highlight how MLModel works.[[10]](#footnote-10)

A Custom model must be defined in a certain form to be accessible. It must be a class inheriting its properties from the mlflow.pyfunc.PythonModel. This creates the access point of MLFlow to the model and includes two standard functions that the model should have. The first is compulsory and is called predict. As its name indicates it should compute and return model predictions. The second optional function is load\_context. It is used to import all packages required by the model as well as any artifacts, thus data files, the model requires for the prediction computing. A small additional note on the artifacts is important. Artifacts only supports certain schemes to import data files and ‘https’, which would be required to download files from a GitHub repository, is not included. To import files from a GitHub repository, use pandas and set the url to the raw file[[11]](#footnote-11). Thus, if you only import files from GitHub, artifacts are not required. Finally, as is standard in object-oriented programming all parameters of a model are defined in the initialization.

After defining the model, the user must define the artifacts required. They are defined in a simple dictionary with the name of the artifact and the absolute path. Further they are also referenced in the MLModel file. Supported schemes are ['', 'file', 's3', 'gs', 'wasbs', 'ftp', 'sftp', 'dbfs', 'hdfs', 'viewfs', 'runs', 'models'][[12]](#footnote-12).

Finally, one needs to define the conda environment in json format as a variable which will be passed to the model creating command. Note that the package cloudpickle is required for model serialization.

The last step is to log the model by passing the artifact path, model and environment. MLModel then creates all model files as defined above. The following function is used:

1. mlflow.pyfunc.log\_model(artifact\_path="model",python\_model=Model\_with\_Parameters, conda\_env=conda\_env, artifacts=artifacts)

* the artifact path is relative, by setting it to model the model is saved in the runs artifact folder under a folder called model
* the python model needs to be the model to store which is of the class mlflow.pyfunc.PythonModel
* the conda environment and artifacts are equal to variables where they are defined in the script

To then call the model and make it return predictions for an observation set simply load the model with:

1. loaded\_model = mlflow.pyfunc.load\_model(model\_path)

* model path is the folder in which the entire model is saved, in our case the model folder in the artifact folder of the run
* It can be referenced in multiple ways, a simple way is to use the id of the run that created the model[[13]](#footnote-13)

Then call the prediction function with data passed on loaded\_model.predict(data).Note that the observations are called ‘data’ in the MLModel framework and should also be specified in this manner in the model class definition. In our example as we do not input new observational data as our goal is forecasting time series data. Thus, we use data to define the forecasting horizon we require from the model.

## Registry

MLFlow Registry is the next step after logging a model. It serves as a tool to organize models. Users can add a model to the registry and then modify, update, transition or delete it via the GUI or API. Basic functions are naming a model, giving it a version starting at 1 and incrementally increasing as it is updated and setting the model stage which can be staging, production or achieving.

Note however that the Registry requires a database-backed backend store. Thus, it is not part of this tutorial as we work simply with our local resources and GitHub for storage. To find out more check out the documentation including screenshots of the GUI to guide threw the various functionalities.[[14]](#footnote-14)

1. Link to the GitHub repository: https://github.com/MatthiasHerp/ETS\_Ex\_BA\_MLFlow [↑](#footnote-ref-1)
2. Link to the Kaggle M5 competition: https://www.kaggle.com/c/m5-forecasting-accuracy [↑](#footnote-ref-2)
3. Link to documentation: https://mlflow.org/docs/latest/index.html [↑](#footnote-ref-3)
4. Link to MLFlow Github repository: https://github.com/mlflow/mlflow/tree/master/examples [↑](#footnote-ref-4)
5. Link to data types of artifacts: <https://towardsdatascience.com/5-tips-for-mlflow-experiment-tracking-c70ae117b03f> [↑](#footnote-ref-5)
6. Jupiter notebooks example for GUI setup: https://github.com/dmatrix/google-colab/blob/master/mlflow\_issue\_3317.ipynb [↑](#footnote-ref-6)
7. GUI Tutorial Link: https://www.mlflow.org/docs/latest/tracking.html#tracking-ui [↑](#footnote-ref-7)
8. More information about tags in MLFlow: https://mlflow.org/docs/latest/tracking.html#id15 [↑](#footnote-ref-8)
9. Built in MLFlow flavors: https://mlflow.org/docs/latest/models.html#built-in-model-flavors [↑](#footnote-ref-9)
10. Further examples of custom models: https://www.mlflow.org/docs/latest/models.html#python-function-python-function [↑](#footnote-ref-10)
11. See in the example code under the load\_context function [↑](#footnote-ref-11)
12. Supported Schemes under Details at: https://mlflow.org/docs/latest/R-api.html#mlflow-load-model [↑](#footnote-ref-12)
13. MLModel schemes: https://www.mlflow.org/docs/latest/concepts.html#artifact-locations [↑](#footnote-ref-13)
14. MLFlow Registry: https://www.mlflow.org/docs/latest/model-registry.html [↑](#footnote-ref-14)