First Impression on MLflow

MLflow is one of the latest open source projects added to the **Apache Spark** ecosystem by databricks. It first debut in the Spark + AI Summit 2018. The source code is hosted in the mlflow-github and is still in the alpha release stage. The current version is 0.4.1 released on 08/03/2018.

Blogs and meetups from databricks describe *MLflow* and its roadmap, including Introducing MLflow: an Open Source Machine Learning Platform and MLflow: Infrastructure for a Complete Machine Learning Life Cycle. Users and developers can find useful information to try out *MLflow* and further contribute to the project.

This blog, however, will dig further and describe some internals of the *MLflow* based on the firsthand experience and the study of the source code. It will also provide suggestions on places *MLflow* may be improved.

What is MLflow

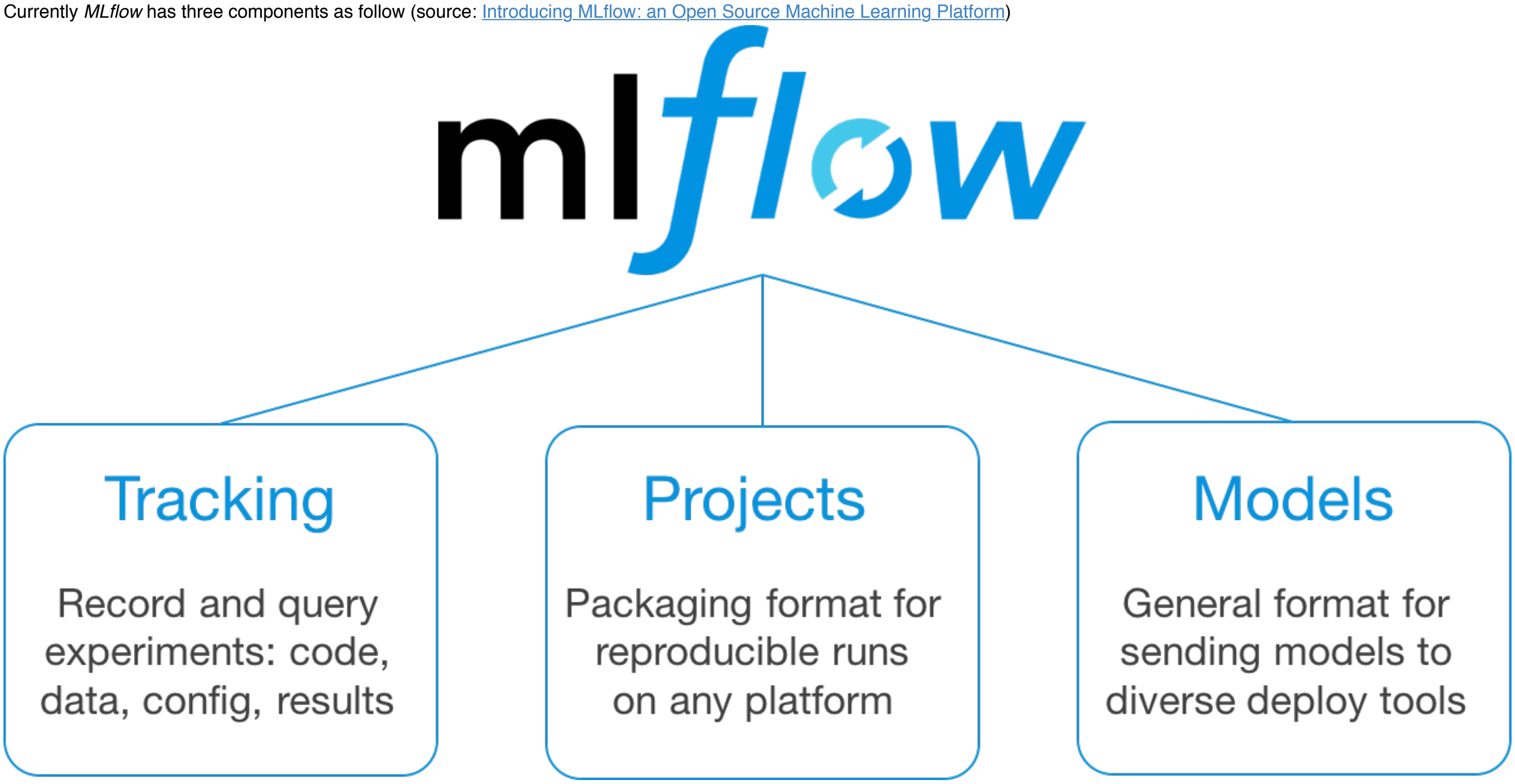
MLflow is targeted as an open source platform for the complete machine learning lifecycle. A complete machine learning lifecycle at least includes raw data ingestion, data analysis and preparing, model training, model evaluation, model deployment and finally model maintenance. *MLflow* is built as a Python package and provides open REST APIs and commands to

- log important parameters, metrics and other data that are mattered to the machine learning model
- track the environment a model is run on
- run any machine learning codes on that environment
- deploy and export models to various platforms with multiple packaging formats

MLflow is implemented as several modules, where each module supports a specific function.

MLflow components

Currently *MLflow* has three components as follow (source: Introducing MLflow: an Open Source Machine Learning Platform)



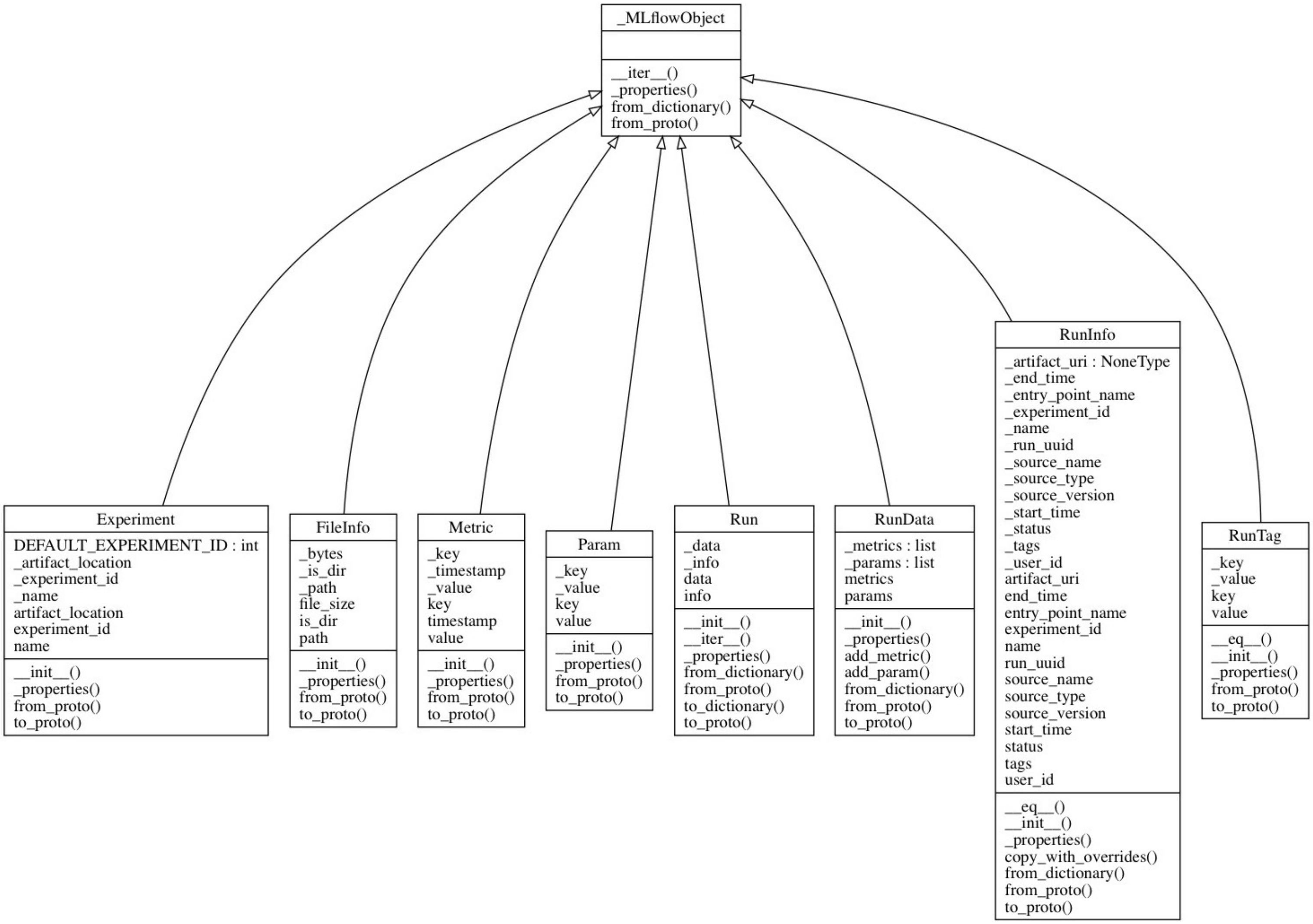
Further description of each component can be found in the blog mentioned above and the link to the *MLflow* Documentation. Rest of the section will give a high level overview of the internals and implementation of each component.

Tracking

`Tracking` component implements REST APIs and UI for parameters, metrics, artifacts and source logging and viewing. The backend is implemented with Flask and run on gunicorn HTTP server while the UI is implemented with React.

The Python module for tracking is `mlflow.tracking`.

Every time users train a model on the machine learning platform *MLflow* creates a `Run` and save the `RunInfo` meta info onto disk. Python APIs are provided to log parameters and metrics for a `Run`. The output of the run such as the model are saved in the `artifacts` for a `Run`. Each individual `Run` is grouped into an `Experiment`. Following class diagram shows classes defined in *MLflow* to support tracking function.

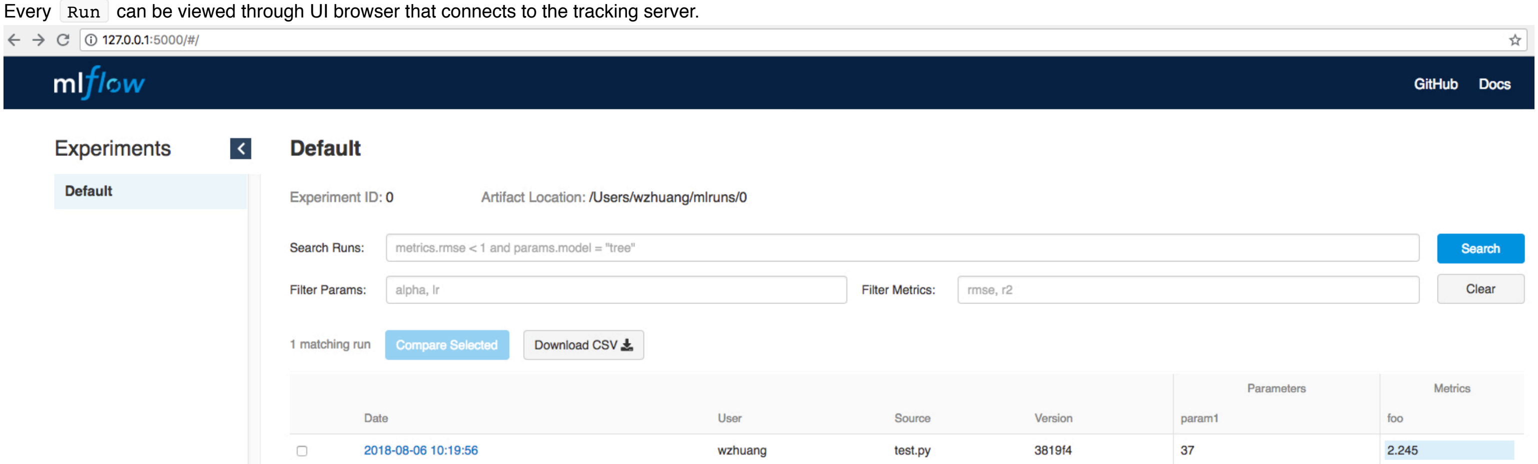


The model training source code needs to call *MLflow* APIs to log the data to be tracked. For example, calling `log_metric` to log the metrics and `log_param` to log the parameters.

MLflow tracking server currently uses file system to persist all `Experiment` data. The directory structure looks like below:

`mlruns └── 0 ├── 7003d550294e4755a65569dd846a7ca6 | ├── artifacts | | └── test.txt | ├── meta.yaml | ├── metrics | | └── foo |`

Every `Run` can be viewed through UI browser that connects to the tracking server.



Users can search and filter models with `metrics` and `params` , and compare and retrieve model details.

Projects

`Projects` component defines the specification on how to run the model training code. It includes the platform configuration, the dependencies, the source code, the data and others that allows the model training to be executed through *MLflow*. Following is an example provided by the *MLflow*: “` name: tutorial

conda_env: conda.yaml

entrypoints: main: parameters: alpha: float l1ratio: {type: float, default: 0.1} command: "python train.py {alpha} {l1_ratio}" ”`

The `mlflow run` command looks for `MLproject` file for the spec and download the dependencies if needed, then runs the model training with the source code and data specified in the `MLproject` . `mlflow run mlflow/example/tutorial -P alpha=0.4`

The `MLproject` specifies the command to run the source code, therefore, the source code can be in any languages, including Python. Projects can be run on many machine learning platforms, including tensorflow, pyspark, scikit-learn and others. If the dependent Python packages are available to download by Anaconda, they can be added to `conda.yaml` file and *MLflow* will set up the packages automatically.

Models

`Models` component defines the general model format in the `MLmodel` file as follow: `artifact_path: model flavors: python_function: data: model.pkl loader_module: mlflow.sklearn sklearn: pickled_model: model.pkl sklearn_versi` It specifies different `flavors` for different tools to deploy and load the model. This allows the model to be saved in its original binary persistence output from the platform training the model. For example, in scikit-learn, the model is serialized with Python `pickle` package. The model can then be deployed to the environment which understands this format. With the `sklearn` flavor, if the environment has the scikit-learn installed, it can directly load the model and serve. Otherwise, with the `python_function` flavor, *MLflow* provides the `mlflow.sklearn` Python module as the helper to load the model.

So far *Mlflow* supports models load, save and deployment with scikit-learn, tensorflow, sagemaker, h2o, azure and spark platforms.

With *MLflow*’s modular design, the current `Tracking` , `Projects` and `Models` components touch most parts of the machine learning lifecycle. Users can also choose to use one component but not the others if they like. With its REST APIs, these components can also be easily integrated into other machine learning workflows.

Experiencing MLflow

Installing *MLflow* is quick and easy if Anaconda has been installed and a virtual env has been created. `pip install mlflow` will install the latest *MLflow* release.

To train the model with `tensorflow` , run `pip install tensorflow` to install the latest version of `tensorflow` .

A simple example to train a tensorflow model with following code tf-example.py

```
import tensorflow as tf
from tensorflow import keras
import numpy as np

import mlflow
from mlflow import tracking

# load dataset
dataset = np.loadtxt("/Users/wzhuang/housing.csv", delimiter=",")

# save the data as artifact
mlflow.log_artifact("/Users/wzhuang/housing.csv")

# split the features and label
X = dataset[:, 0:15]
Y = dataset[:, 15]

# define the model
first_layer_dense = 64
second_layer_dense = 64
model = keras.Sequential([
    keras.layers.Dense(first_layer_dense, activation=tf.nn.relu,
                       input_shape=(X.shape[1],)),
    keras.layers.Dense(second_layer_dense, activation=tf.nn.relu),
    keras.layers.Dense(1)
  ])

# log some parameters
mlflow.log_param("First_layer_dense", first_layer_dense)
mlflow.log_param("Second_layer_dense", second_layer_dense)

optimizer = tf.train.RMSPropOptimizer(0.001)

model.compile(loss='mse',
              optimizer=optimizer,
              metrics=['mae'])

# train
model.fit(X, Y, epochs=500, validation_split=0.2, verbose=0)

# log the model artifact
model_json = model.to_json()
with open("model.json", "w") as json_file:
    json_file.write(model_json)
mlflow.log_artifact("model.json")
```

The first call to the `tracking` API will start the tracking server and log all the data sent through the current and subsequent APIs. These logged data can then be viewed in the *MLflow* UI. From the example above, it is quite easy to just call the logging APIs in any place users want to track.

Packaging this project is also very simple by just creating a MLproject file as such:
`name: tf-example conda_env: conda.yaml entry_points: main: command: "python tf-example.py"` with conda.yaml
`name: tf-example channels: - defaults dependencies: - python=3.6 - numpy=1.14.3 - pip: - mlflow - tensorflow`

Then `mlflow run tf-example` will run the project on any environment. It first creates a `conda` environment with the required Python packages installed and then run the tf-example.py inside that virtual env. As expected, the run result is also logged to the *MLflow* tracking server.

MLflow also comes with a server implementation where the `sklearn` and other types of models can be deployed and served. The *MLflow* github README.md illustrates the usage. However, to deploy and serve the model built by the above example requires new code that understands Keras models. This is beyond this blog's scope.

To summarize, the experience with *MLflow* is smooth. There were several bugs here and there but overall was satisfied with what the project claims to be. Of course since *MLflow* is still in its alpha phase, bugs and lacking of some features are expected.

Things *MLflow* can be enhanced

MLflow so far provides an open source solution to track the data science processing, package and deploy machine learning model. As it claims, it targets the management of the machine learning lifecycle. The current alpha version releases the `Tracking`, `Projects` and `Models` components that tackle individual stages of the machine learning workflow. The tool is compact in Python language while providing APIs and UI to be integrated with any machine learning platform easily.

However, there are still many places that *MLflow* may be improved. There are also new features required for the tool to fully manage and monitor all aspects of the lifecycle of machine learning.

At the Databricks' meetup on 07/19/2018, several items have been mentioned in the longer-term road map of *MLflow* according to the presentation. There are four categories, including improving current components, new MLflow Data component, hyperparameter tuning and language and library integrations. Some items are really important so they need extra explain.

Implementing a database backend for `Tracking` component is included in the first category. As mentioned above, the *MLflow* tracking server logs every run info in local file system. This looks like a quick and easy implementation. A better solution will be using a database as the tracking store. When the number of machine learning runs grows, database has its obvious advantage on data queries and retrieval.

Model metadata support is also included in the first category. This is extremely important. Current `Tracking` component does not describe the model and all runs are viewed as a flatten list ordered by date. The tool allows the search based on the parameters and metrics, but it is far away from enough. Users certainly would like to quickly retrieve the models by model name, algorithm, platform etc. This requires metadata input when a model training is tracked. Tracking server logs the file name of the source code. This does not provide any value to identify a model. Instead, it should allow to input a description of the model. Furthermore, the access control is also essential and can be part of the metadata. And model management should also have versioning support.

In the second category, *MLflow* will introduce a new `Data` component. It will build on top of Spark's Data Source API and allows projects to load data from many formats. This can be viewed as an effort to tighten the *MLflow* relationship with *Spark*. What should be done further is of course maintaining the metadata for the data.

In the fourth category, the integration with R and Java is also important. Although Python today becomes the most adopted language in machine learning, there are many data scientists still using R and other languages. *MLflow* needs to provide R and Java APIs so those machine learning workflows can be managed as well.

There are other important features not included in the current road map. From this blog's viewpoint, following list of items are also desired and may help complete *MLflow* as a full machine learning data and model management tool.

- Register APIs
  MLflow provides the APIs to log run info. These APIs have to be called inside the model training source code and they are called at runtime. This approach becomes inconvenient either users just want to track the previously runs without these APIs, or runs without access to the source code. To solve such problem, a set of REST APIs that can be called post run to register the run info will be very helpful. The run info, such as parameters, metrics and artifacts, can be part of the JSON input.
- UI view enhancement
  In the `Experiments` UI view, `Parameters` and `Metrics` columns display all parameters and metrics for all runs. The row will become unfriendly long and difficult to view when more types of parameters and metrics are tracked. Instead, for each run, the view should just display a hyperlink to the detailed run info where the parameters and metrics will show only for this run. Furthermore, this approach can help solve the problem on logging
- Artifact location
  MLflow can take artifacts from either local or github. It would be a great improvement to support the load and save data, source code and model from other sources, such as S3 Object Storage, HDFS, Nexus etc.
- Import and export
  Once the tracking store is implemented with database as backend, the next thing will be to support import and export all experiments stored in different databases.
- Run projects remotely
  `Projects` component specifies the command to run the project and the command is displayed in the tracking UI. But since the project can only run on the specific machine learning platform, which can be different from the tracking server, users still have to connect to the platform remotely and issue the command line. The `MLproject` specification should include the platform information, such as hostname and credentials. With these info, the tracking UI should add an action to kick off the run through the UI.
- Tuning
  Adding the parameter tuning functionality through the tracking UI is an important feature. Users will be allowed to change the parameters and kick off the run if the project is tracked by the `Projects` component.
- Common model format
  `Models` component defines `flavors` for a model. However, every model still stores in its original format only understood by that training tool. There is still gap between the model development and production. Portable Format for Analytics is a specification that can help bridge the gap. `MLmode` can be improved to understand PFA and/or convert models into PFA for easy deploying models to PFA-enabled platforms.
- Pipeline integration
  A complete machine learning lifecycle also includes data preparation and other pipelines. *MLflow* so far only tracks the training step. The `MLproject` may be enhanced to include the specification of other pipelines. Some pipelines may be shared by projects as well.