ml-experiment: A Python framework for reproducible data science

Antonio Molner Domenech

E-mail: antoniomolner@correo.ugr.es

Alberto Guillén

Computer Architecture and Technology Department, University of Granada, Spain.

E-mail: aguillen@ugr.es

Abstract. Nowadays, data science projects are usually developed in an unstructured way, which makes it difficult to reproduce. It is also hard to move from an experimental environment to production. Operational workflows such as containerization, continuous deployment, and cloud orchestration allow data science researchers to move a pipeline from a local environment to the cloud. Being aware of the difficulties of setting those workflows up, this paper presents a framework to ease experiment tracking and operationalizing machine learning by combining existent and well-supported technologies. These technologies include Docker, Mlflow, Ray, among others. The framework provides an opinionated workflow to design and execute experiments either on a local environment or the cloud. ml-experiment includes: an automatic tracking system for the most famous machine learning libraries: Tensorflow, Keras, Fastai, Xgboost and Lightgdm, first-class support for distributed training and hyperparameter optimization, and a Command Line Interface (CLI) for packaging and running projects inside containers.

1. Introduction

Reproducibility is a challenge in modern research and produces a lot of discussion [1, 2, 3]. Among all types of reproducible research, the work presented in this paper focuses on computational research [4]: building a set of tools and a specific workflow based on the principles of version control, automation, tracking, and environment isolation [5, 6]. Version control enables keeping track of files changes through time and facilitate the collaboration. Automation, from shell scripts to fully defined pipelines, allows your audience to reproduce each step of your work easily. These steps include creating files, processing data, fitting models, etc. Tracking includes a set of techniques to record the research objects such as figures, data artifacts, results, etc, systematically. Finally, environment isolation makes it easier to install all dependencies of the analytical tools with its specific versions separately from the host machine. As long as we create an isolated environment that represents a "common scenario" for researchers, computational reproducibility will not be jeopardized.

On the other hand, computational reproducibility is not an issue specific to research, all these principles we are looking forward to applying in research can be used in the industry as well

[7]. Applied data science is a field heavily focused on research, so the same ideas should apply for both fields. Our motivation to develop this framework is to both facilitate the experimental work and to fill the gap between research and deployment in the industry.

2. Structure

The framework main core is divided into four modules (depicted in Figure 1) that interact with the user through a Command-Line Interface (CLI) and a Python library. The objective of the library is to minimize the code changes required to instrument scripts to be executed by the Job Runner and to provide the abstractions to interact with the Tracking and Hyperparameter Optimization engines. On the other hand, the CLI is in charge of executing scripts either in a local environment or a remote environment.

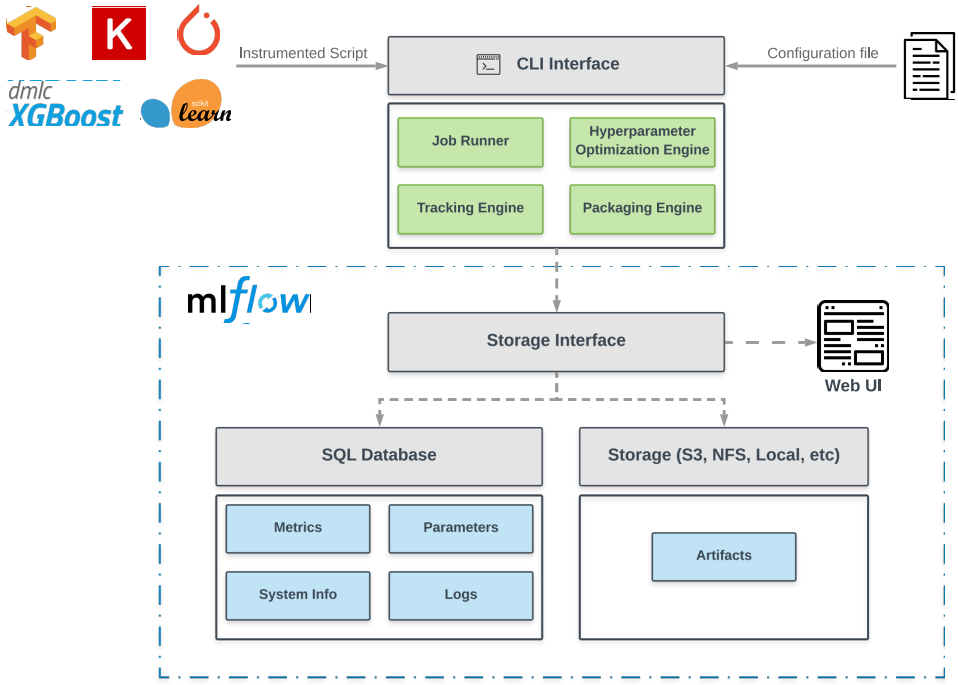

Figure 1. ml-experiment framework overview

The main idea around this tool is the concept of Job. A Job is an entity defined in a configuration file written in YAML or JSON which includes: the name of the job, the script, or list of scripts to execute, parameters, and other optional fields for containerized execution, Hyperparameter optimization, and Ray [8] cluster setup. This configuration file serves as input for the CLI which reads the file and executes the job according to the content. The only requirement for a Python script to be used in a job is to be instrumented in the following form:

```python
from ml_experiment import job

@job # Add this decorator
def main(param1, param2, ...):
    # your code goes here

if __name__ == '__main__':
    main()
```

Listing 1: Example of an instrumented python script

A job can represent any computation, from a data processing task to a set of experiments executed in parallel, so to differentiate between tasks that need to use the tracking engine or the hyperparameter optimization engine, two more entities have been derived from Job: Experiment and Group. An experiment is a job that uses the tracking engine to log the parameters, metrics, and artifacts generated during its execution. While a Group is a set of parallel experiments executed with different parameters using the hyperparameter optimization engine.

Packaging Engine is the last module, which is responsible for generating a shareable version of the project using Docker [9] technology to create an image of all dependencies. This module focus on simplifying the reproduction of scientific work across multiples and heterogeneous environments by providing an abstraction layer to create *Docker* images easily.

Finally, all these different modules rely on a common knowledge center. This knowledge center is composed of an SQL database for metrics, parameters, and system logs; and artifact storage - local environment, S3 bucket, NFS, etc - where heavy files are stored.

3. Experiment tracking

Experiments and groups of experiments use the tracking engine to keep record of the information produced during the execution. By instrumenting a python script we allow the framework to automatically record the following information:

- *Parameters*. The arguments used to execute the experiment.
- *System information*: CPU information, GPU drivers, hostname, etc.
- *Version control metadata*. Git commit hash.
- *Metrics and artifacts*. If the main function returns a dictionary of metrics and a dictionary of artifact paths optionally.
- *Logs*. The console output is saved in a file and uploaded to the server.

```
name: "SVM with RBF kernel"
kind: 'experiment'

params:
  C: 0.5
  kernel: 'rbf'
  gamma: 'auto'

run:
  - modelling/train_svm.py
```

Listing 2: Experiment configuration file example

4. Hyperparameter optimization and distributed learning

ml-experiment has first-class support for hyperparameter optimization and distributed learning. The framework allows executing multiples experiments simultaneously in a local or remote environment. It also enables the configuration of the resources distribution among experiments. Thus, we can specify the number of CPUs and GPUs (if available) to use and it will distribute the tasks between the different resources. As an example, 4 reflects the definition of a group of twelve experiments, where its parameters C and gamma are sampled from some distributions. The objective of the job is to maximize the metric called val_accuracy and the experiments will be distributed across four processes, it means three experiments will be executed per CPU core.

```python
from ml_experiment import job

@job
def main(C, kernel, gamma='scale'):
    np.random.seed(1234)
    X_train, X_val, y_train, y_val = load_data()
    model = SVC(C=C, gamma=gamma, kernel=kernel)
    model.fit(X_train, y_train)
    accuracy = model.score(X_val, y_val)
    return {'val_accuracy': accuracy}


if __name__ == '__main__':
    main()
```

Listing 3: Experiment python script example

Another important aspect of any hyperparameter optimization engine is the efficiency of the sample space evolution. To achieve competitive results in a limited amount of time, we need a set of tools and algorithms to construct parameter spaces that maximize the objective function in the minimum amount of time. Thanks to *Optuna* [**?**] we have designed an optimization module that supports state-of-the-art algorithms such as *TPE, CMA-ES*, and traditional ones like *GridSearch* and *RandomSearch* [10].

```yaml
name: "SVM"
kind: 'group'
num_trials: 12
sampler: TPE

param_space:
  C: loguniform(0.01, 1000)
  gamma: choice(['scale', 'auto'])

metric:
  name: val_accuracy
  direction: maximize

ray_config:
  num_cpus: 4

run:
  - modelling/train_svm.py
```

Listing 4: Group configuration file example

5. Web User Interface

Arguably the most relevant module of ml-experiment would be the Web UI (see Figure 2). All the experiments and group of experiments with its metrics, logs, parameters and artifacts are logged in the storage service described in 1. But is the web user interface what allows researchers or machine learning practitioners to examine and compare between different experiments. For this module, we opted for using Mlflow [11] as it is a well-supported and rapidly growing tool used by companies like Microsoft or Databricks.

6. Conclusions

This paper presents a simple interface to generate a container based application that allows to scale experiments and to easily deploy the models obtained in a secure and controlled

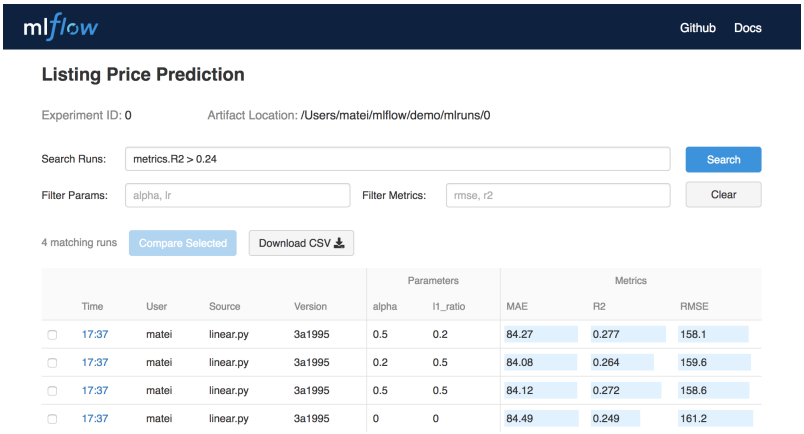

Figure 2. Mlflow Web UI screenshot

manner. Thus, reproducibility is achieved by carefully logging all the stages of the machine learning pipeline. Another important aspect is the homogeneity in the interface that will allow practitioners to share and reuse their pipelines in different scenarios without rewriting code.

Acknowledgements

This research has been possible thanks to the support of projects: FPA2017-85197-P (Spanish Ministry of Economy and Competitiveness –MINECO– and the European Regional Development Fund. –ERDF).

References

[1] Matthew Hutson. Artificial intelligence faces reproducibility crisis. *Science*, 359(6377):725–726, 2018.

[2] Juliana Freire, Norbert Fuhr, and Andreas Rauber. Reproducibility of Data-Oriented Experiments in e-Science (Dagstuhl Seminar 16041). *Dagstuhl Reports*, 6(1):108–159, 2016.

[3] Juliana Freire, Philippe Bonnet, and Dennis Shasha. Computational reproducibility: State-of-the-art, challenges, and database research opportunities. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 593–596, New York, NY, USA, 2012. Association for Computing Machinery.

[4] Victoria Stodden, David H. Bailey, Jonathan M. Borwein, Randall J. LeVeque, William J. Rider, and William Stein. Setting the default to reproducible reproducibility in computational and experimental mathematics. 2013.

[5] Babatunde Kazeem Olorisade, Pearl Brereton, and Peter Andras. Reproducibility in machine learning-based studies: An example of text mining. 2017.

[6] Greg Wilson, Jennifer Bryan, Karen Cranston, Justin Kitzes, Lex Nederbragt, and Tracy K. Teal. Good enough practices in scientific computing. *PLOS Computational Biology*, 13(6):1–20, 06 2017.

[7] Grigori Fursin, Herve Guillou, and Nicolas Essayan. Codereef: an open platform for portable mlops, reusable automation actions and reproducible benchmarking, 2020.

[8] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging ai applications, 2017.

[9] Carl Boettiger. An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, Jan 2015.

[10] James Bergstra, R. Bardenet, Balázs Kégl, and Y. Bengio. Algorithms for hyper-parameter optimization. 12 2011.

[11] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Fen Xie, and Corey Zumar. Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull.*, 41:39–45, 2018.