Accelerating the Machine Learning Lifecycle with MLflow

Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Fen Xie, Corey Zumar
Databricks Inc.

Abstract

Machine learning development creates multiple new challenges that are not present in a traditional software development lifecycle. These include keeping track of the myriad inputs to an ML application (e.g., data versions, code and tuning parameters), reproducing results, and production deployment. In this paper, we summarize these challenges from our experience with Databricks customers, and describe MLflow, an open source platform we recently launched to streamline the machine learning lifecycle. MLflow covers three key challenges: experimentation, reproducibility, and model deployment, using generic APIs that work with any ML library, algorithm and programming language. The project has a rapidly growing open source community, with over 50 contributors since its launch in June 2018.

1 Introduction

Machine learning development requires solving new problems that are not part of the standard software development lifecycle. For example, while traditional software has a well-defined set of product features to be built, ML development tends to revolve around *experimentation*: the ML developer will constantly experiment with new datasets, models, software libraries, tuning parameters, etc. to optimize a business metric such as model accuracy. Because model performance depends heavily on the input data and training process, *reproducibility* is paramount throughout ML development. Finally, in order to have business impact, ML applications need to be *deployed* to production, which means both deploying a model in a way that can be used for inference (e.g., REST serving) and deploying scheduled jobs to regularly update the model. This is especially challenging when deployment requires collaboration with another team, such as application engineers who are not ML experts.

Based on our conversations with dozens of Databricks customers that use machine learning, these lifecycle problems are a major bottleneck in practice. Although today's ML libraries provide tools for part of the lifecycle, there are no standard systems and interfaces to manage the full process. For example, TensorFlow offers a training API and a Serving system [2], but TensorFlow Serving cannot easily be used for models from another ML library, or from an incompatible version of TensorFlow. In practice, an organization will need to run models from multiple ML libraries, TensorFlow versions, etc., and has to design its own infrastructure for this task.

Faced with these challenges, many organizations try to "lock down" the ML development process to obtain reproducibility and deployability. Some organizations develop internal guidelines for ML development, such as which libraries one can use that the production team will support. Others develop internal *ML platforms* (e.g., Facebook's FBLearner [6], Uber's Michelangelo [11] and Google's TFX [2]): APIs that ML developers must

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

use in order to build deployable models. Unfortunately, both approaches limit ML developers in the algorithms and libraries they can use, decreasing their ability to experiment, and both create substantial engineering work whenever the ML developers want to use new libraries or models.

In this paper, we summarize our experience with ML lifecycle challenges at Databricks customers and describe MLflow, an open source ML platform we are developing to address these challenges. MLflow's key principle is an *open interface* design, where data scientists and engineers can bring their own training code, metrics, and inference logic while benefitting from a structured development process. For example, a "model" saved in MLflow can simply be a Python function (and associated library dependencies) that MLflow then knows how to deploy in various environments (e.g., batch or real-time scoring). Other MLflow abstractions are likewise based on generic interfaces, such as REST APIs and Docker containers. Compared to existing ML platforms like FBLearner, Michelangelo and TFX, this open interface design gives users flexibility and control while retaining the benefits of lifecycle management. The current version of MLflow provides APIs for experiment tracking, reproducible runs and model packaging and deployment, usable in Python, Java and R. We describe these APIs and some sample MLflow use cases to show how the system can streamline the machine learning lifecycle.

2 Challenges in Machine Learning Development

ML faces many of the challenges in traditional software development, such as testing, code review, monitoring, etc. In other ways, however, ML applications are different from traditional software, and present new problems.

One of the main differences is that the goal in machine learning is to *optimize* a specific metric, such as prediction accuracy, instead of simply meeting a set of functional requirements. For example, for a retailer, every 1% improvement in prediction accuracy for a recommendation engine might lead to millions of dollars in revenue, so the ML team working on this engine will continuously want to improve the model. This means that ML developers wish to continuously experiment with the latest models, software libraries, etc., to improve target metrics. Beyond this difference in objective, ML applications are more complex to manage because their performance depends on training data, tuning, and concerns such as overfitting that do not occur in other applications. Finally, ML applications are often developed by teams or individuals with very different expertise, and hand-off between these individuals can be challenging. For example, a data scientist might be an expert at ML training, and use her skills to create a model, but she might need to pass the model to a software engineer for deployment within an application. Any errors in this process (e.g., mismatched software versions or data formats) might lead to incorrect results that are hard for a software engineer without ML knowledge to debug.

Based on these requirements in working with ML, we found four challenges to arise repeatedly at ML users:

1. Multitude of tools. Hundreds of software tools cover each phase of ML development, from data preparation to model training to deployment. However, unlike traditional software development, where teams select *one* tool for each phase, ML developers usually want to try *every* available tool (e.g., algorithm) to see whether it improves results. For example, a team might try multiple preprocessing libraries (e.g., Pandas and Apache Spark) to featurize data; multiple model types (e.g. trees and deep learning); and even multiple frameworks for the same model type (e.g., TensorFlow and PyTorch) to run various models published online by researchers.

2. Experiment tracking. Machine learning results are affected by dozens of configurable parameters, ranging from the input data to hyperparameters and preprocessing code. Whether an individual is working alone or on a team, it is difficult to track which parameters, code, and data went into each experiment to produce a model.

3. Reproducibility. Without detailed tracking, teams often have trouble getting the same code to work again. For example, a data scientist passing her training code to an engineer for use in production might see problems if the engineer modifies it, and even a user working alone needs to reliably reproduce old results to stay productive.

4. Production deployment. Moving an application to production can be challenging, both for inference and training. First, there are a plethora of possible inference environments, such as REST serving, batch scoring and mobile applications, but there is no standard way to move models from any library to these diverse environments. Second, the model training pipeline also needs to be reliably converted to a scheduled job, which requires care to reproduce the software environments, parameters, etc. used in development. Production deployment is especially challenging because it often requires passing the ML application to a different team with less ML expertise.

To address these problems, we believe that ML development processes should be explicitly designed to promote reproducibility, deployability, etc. The challenge is how to do so while leaving maximum flexibility for ML developers to build the best possible model. This led us to the open interface design philosophy for MLflow.

3 MLflow Overview

To structure the ML development process while leaving users maximum flexibility, we built MLflow around an *open interface* philosophy: the system should define general interfaces for each abstraction (e.g., a training step, a deployment tool or a model) that allow users to bring their own code or workflows. For example, many existing ML tools represent models using a serialization format, such as TensorFlow graphs [1], ONNX [14] or PMML [9], when passing them from training to serving. This restricts applications to using specific libraries. In contrast, in MLflow, a model can be represented simply as a Python function (and library dependency information), so any development tool that knows how to run a Python function can run such a model. For more specialized deployment tools, a model can also expose other interfaces called "flavors" (e.g., an ONNX graph) while still remaining viewable as just a Python function. As another example, MLflow exposes most of its features through REST APIs that can called from any programming language.

More specifically, MLflow provides three components, which can either be used together or separately:

- **MLflow Tracking**, which is an API for recording experiment runs, including code used, parameters, input data, metrics, and arbitrary output files. These runs can then be queried through an API or UI.
- **MLflow Projects**, a simple format for packaging code into reusable projects. Each project defines its environment (e.g., software libraries required), the code to run, and parameters that can be used to call the project programmatically in a multi-step workflow or in automated tools such as hyperparameter tuners.
- **MLflow Models**, a generic format for packaging models (both the code and data required) that can work with diverse deployment tools (e.g., batch and real-time inference).

3.1 MLflow Tracking

MLflow Tracking is an API for logging and querying *experiment runs*, which consist of parameters, code versions, metrics and arbitrary output files called *artifacts*. Users can start/end runs and log metrics, parameters and artifacts using simple API calls, as shown below using MLflow's Python API:

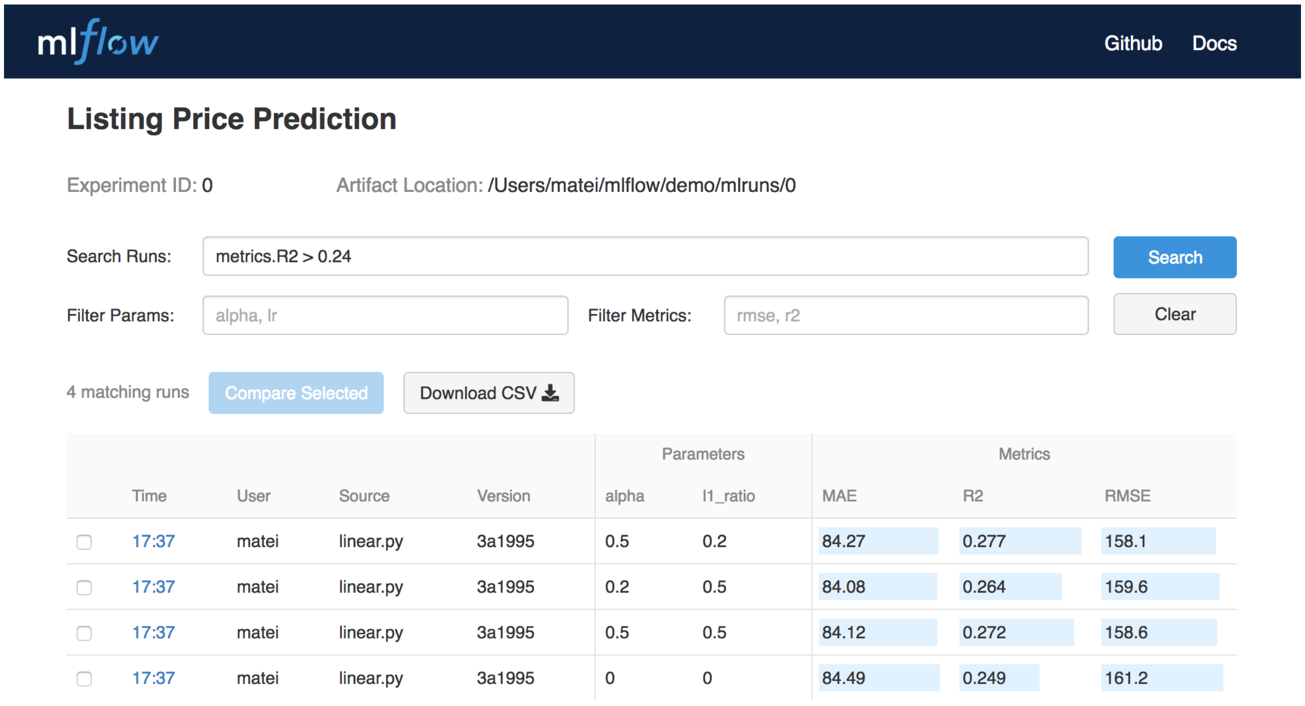
```
# Log parameters, which are arbitrary key-value pairs
mlflow.log_param("num_dimensions", 8)
mlflow.log_param("regularization", 0.1)

# Log metrics; each metric can also be updated throughout the run
mlflow.log_metric("accuracy", 0.8)
mlflow.log_metric("r2", 0.4)

# Log artifacts (arbitrary output files)
mlflow.log_artifact("precision_recall.png")
```

MLflow Tracking API calls can be inserted anywhere users run code (e.g., standalone applications or Jupyter notebooks running in the cloud). The tracking API logs results to a local directory by default, but it can also be configured to log over the network to a server, allowing teams to share a centralized MLflow tracking server and compare results from multiple developers.

Once users have recorded runs, MLflow allows users to query them through an API or web-based UI (Figure 1). This UI includes the ability to organize runs into groups called Experiments, search and sort them, and compare groups of runs, enabling users to build a custom leaderboard for each of their ML problems and even compare results across teams. The UI is inspired by experiment visualization tools such as Sacred [12], ModelDB [15] and TensorBoard [8], and supports similar visualizations and queries.



Listing Price Prediction

Experiment ID: 0 Artifact Location: /Users/matei/mlflow/demo/mlruns/0

Search Runs: metrics.R2 > 0.24

Filter Params: alpha, lr Filter Metrics: rmse, r2

4 matching runs

	Time	User	Source	Version	Parameters		Metrics		
					alpha	l1_ratio	MAE	R2	RMSE
☐	17:37	matei	linear.py	3a1995	0.5	0.2	84.27	0.277	158.1
☐	17:37	matei	linear.py	3a1995	0.2	0.5	84.08	0.264	159.6
☐	17:37	matei	linear.py	3a1995	0.5	0.5	84.12	0.272	158.6
☐	17:37	matei	linear.py	3a1995	0	0	84.49	0.249	161.2

Figure 1: MLflow Tracking UI showing several runs in an experiment. Clicking each run lists its metrics, artifacts and output details and lets the user post comments about the run.

3.2 MLflow Projects

MLflow Projects provide a simple format for packaging reproducible data science code. Each project is simply a directory with code or a Git repository, and uses a descriptor file to specify its dependencies and how to run the code. A MLflow Project is defined by a simple YAML file called MLproject, as shown below:

```
name: My Project
conda_env: conda.yaml
entry_points:
  main:
    parameters:
      data_file: path
      alpha: {type: float, default: 0.1}
    command: "python train.py --reg-param {alpha} --data {data_file}"
```

Projects can specify their dependencies through a Conda environment or (in an upcoming release) a Docker container specification. A project may also have multiple entry points for invoking runs, with named parameters that downstream users can provide without understanding the internals of the project.

Users can run projects using the `mlflow run` command line tool, either from local files or a Git repository:

```
mlflow run git@github.com:databricks/mlflow-example.git -P alpha=0.5
```

Alternatively, projects can be called programmatically using MLflow's API. This can be used to implement multi-step workflows or to pass projects a "black box" into automated tools such as hyperparameter search [3].

In either case, MLflow will automatically set up the project's runtime environment and execute it. If the code inside the project uses the MLflow Tracking API, MLflow will also remember the project version executed (that is, the Git commit) and show an `mlflow run` command to re-execute it in its UI. Finally, MLflow projects can also be submitted to cloud platforms such as Databricks for remote execution.

3.3 MLflow Models

MLflow Models are a convention for packaging machine learning models in multiple formats called "flavors", allowing diverse tools to understand the model at different levels of abstractions. MLflow also offers a variety of built-in tools to deploy models in its standard favors. For example, the same model can be deployed as a Docker container for REST serving, as an Apache Spark user-defined function (UDF) for batch inference, or into cloud-managed serving platforms like Amazon SageMaker and Azure ML.

Each MLflow Model is simply stored as a directory containing arbitrary files and an MLmodel YAML file that lists the flavors it can be used in and additional metadata about how it was created:

```
time_created: 2018-02-21T13:21:34.12
run_id: c4b65fc2c57f4b6d80c6e58a9dcb9f01
flavors:
  sklearn:
    sklearn_version: 0.19.1
    pickled_model: model.pkl
  python_function:
    loader_module: mlflow.sklearn
    pickled_model: model.pkl
```

In this example, the model can be used with tools that support either the `sklearn` or `python_function` model flavors. For example, the MLflow SciKit-Learn library knows how to load a `sklearn` model as a SciKit-Learn Python object, but other deployment tools, such as running the model in a Docker HTTP server, only understand lower-level flavors like `python_function`. In addition, models logged using MLflow Tracking APIs will automatically include a reference to that run's unique ID, letting users discover how they were built.

4 Example Use Cases

In this section, we describe three sample MLflow use cases to highlight how users can leverage each component.

Experiment tracking. A European energy company is using MLflow to track and update hundreds of energy grid models. This team's goal is to build a time series model for every major energy producer (e.g., power plant) and consumer (e.g., factory), monitor these using standard metrics, and combine the predictions to drive business processes such as pricing. Because a single team is responsible for hundreds of models, possibly using different ML libraries, it was important to have a standard development and tracking process. The team has standardized on using Jupyter notebooks for development, MLflow Tracking for metrics, and Databricks jobs for inference.

Reproducible projects. An online marketplace is using MLflow Projects to package deep learning jobs using Keras and run them in the cloud. Each data scientist develops models locally on his or her laptop using a small dataset, checks them into a Git repository with an MLproject file, and submits remote runs of the project to GPU instances in the cloud for large-scale training or hyperparameter search. Using MLflow Projects makes it easy to create the same software environment in the cloud and share project code between different data scientists.

Model packaging. The data science team at an e-commerce site is using MLflow Models to package recommendation models for use by application engineers. The technical challenge here was that the recommendation application includes both a standard, "off-the-shelf" recommendation model and custom business logic for pre- and post-processing. For example, the application might include custom code to make sure that the recommended items are diverse. This business logic needs to change in sync with the model, and the data science team wants to control both the business logic and the model, without having to submit a patch to the web application each time this logic has to change. Moreover, the team wants to A/B test distinct models with distinct versions of the processing logic. The solution was to package both the recommendation model and the custom logic using the `python_function` flavor in an MLflow Model, which can then be deployed and tested as a single unit.

5 Related Work

Many software systems aim to simplify ML development. The closest to our work are the end-to-end "ML platforms" at large web companies. For example, Facebook's FBLearner lets users write reusable workflow steps that run over data in Apache Hive [6]; Uber's Michelangelo gives users a toolkit of algorithms to choose from that it can automatically train and deploy [11]; and Google's TFX provides data preparation and serving tools around TensorFlow [2]. Anecdotally, these platforms greatly accelerate ML development, showing the benefits of standardizing the ML lifecycle. However, they generally restrict users to a specific set of algorithms or libraries, so teams are on their own when they step outside these boundaries. Our goal in MLflow is to let users easily bring their own tools and software in as many steps in the process as possible through our "open interface" design. This includes custom training steps, inference code, and logged parameters and artifacts.

Other systems also tackle specific problems within the ML lifecycle. For example, Sacred [12], ModelDB [15] and TensorBoard [8] let users track experiments; PMML [9] and ONNX [14] are cross-library model serialization formats; Clipper [3] can deploy arbitrary models as Docker containers; and CDE [10], CodaLab [13], Binder [4] and Repo2Docker [7] enable reproducible software runs. MLflow combines these concepts with new ones, such as multi-flavor model packaging, into a unified system design and API.

6 Conclusion

For machine learning to have widespread commercial impact, organizations require the same kinds of reliable engineering processes around ML that exist in other engineering disciplines such as software development. In this paper, we have described some of the key challenges that differentiate ML development from traditional software development, such as experimentation, reproducibility, and reliable production deployment. We have also described MLflow, a software platform that can structure the machine learning lifecycle while giving users broad flexibility to use their own ML algorithms, software libraries and development processes. MLflow is available as open source software at `https://www.mlflow.org`.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, volume 16, pages 265–283, 2016.

[2] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc, C. Y. Koo, L. Lew, C. Mewald, A. N. Modi, N. Polyzotis, S. Ramesh, S. Roy, S. E. Whang, M. Wicke, J. Wilkiewicz, X. Zhang, and M. Zinkevich. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 1387–1395, New York, NY, USA, 2017. ACM.

[3] J. Bergstra, B. Komer, C. Eliasmith, D. Yamins, and D. D. Cox. Hyperopt: a python library for model selection and hyperparameter optimization. *Computational Science and Discovery*, 8(1):014008, 2015.

[4] Binder. `https://mybinder.org`, 2018.

[5] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 613–627, Berkeley, CA, USA, 2017. USENIX Association.

[6] J. Dunn. Introducing FBLearner Flow: Facebook's AI backbone. `https://code.fb.com/core-data/introducing-fblearner-flow-facebook-s-ai-backbone/`.

[7] J. Forde, T. Head, C. Holdgraf, Y. Panda, G. Nalvarte, M. Pacer, F. Perez, B. Ragan-Kelley, and E. Sundell. Reproducible research environments with repo2docker. ICML, 07/2018 2018.

[8] Google. Tensorboard: Visualizing learning. `https://www.tensorflow.org/guide/summaries_and_tensorboard`.

[9] A. Guazzelli, W.-C. Lin, and T. Jena. *PMML in Action: Unleashing the Power of Open Standards for Data Mining and Predictive Analytics*. CreateSpace, Paramount, CA, 2nd edition, 2012.

[10] P. J. Guo. CDE: A tool for creating portable experimental software packages. *Computing in Science and Engineering*, 14(4):32–35, 2012.

[11] J. Hermann and M. D. Balso. Meet Michelangelo: Uber's machine learning platform. `https://eng.uber.com/michelangelo/`.

[12] Klaus Greff, Aaron Klein, Martin Chovanec, Frank Hutter, and Jürgen Schmidhuber. The Sacred Infrastructure for Computational Research. In Katy Huff, David Lippa, Dillon Niederhut, and M. Pacer, editors, *Proceedings of the 16th Python in Science Conference*, pages 49 – 56, 2017.

[13] P. Liang et al. CodaLab. `https://worksheets.codalab.org`, 2018.

[14] ONNX Group. ONNX. `https://onnx.ai`.

[15] M. Vartak, H. Subramanyam, W.-E. Lee, S. Viswanathan, S. Husnoo, S. Madden, and M. Zaharia. Modeldb: A system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, HILDA '16, pages 14:1–14:3, New York, NY, USA, 2016. ACM.