Automatically Tracking Metadata and Provenance of Machine Learning Experiments

Sebastian Schelter, Joos-Hendrik Böse, Johannes Kirschnick, Thoralf Klein, Stephan Seufert
Amazon
{sseb,jooshenb,kirschnj,thoralfk,seufert}@amazon.com

Abstract

We present a lightweight system to extract, store and manage metadata and provenance information of common artifacts in machine learning (ML) experiments: datasets, models, predictions, evaluations and training runs. Our system accelerates users in their ML workflow, and provides a basis for comparability and repeatability of ML experiments. We achieve this by tracking the lineage of produced artifacts and automatically extracting metadata such as hyperparameters of models, schemas of datasets or layouts of deep neural networks. Our system provides a general declarative representation of said ML artifacts, is integrated with popular frameworks such as MXNet, SparkML and scikit-learn, and meets the demands of various production use cases at Amazon.

1 Introduction

When developing and productionizing ML models, a major proportion of the time is spent on conducting model selection experiments which consist of training and tuning models and their corresponding features [22, 4, 14, 20, 27, 3]. Typically, data scientists conduct this experimentation in their own ad-hoc style without a standardized way of storing and managing the resulting experimentation data and artifacts. As a consequence, the results of these experiments are often not comparable, as there is no standard way to determine whether two models had been trained on the same input data, for example. Even more, it is tedious and time-consuming to repeat successful results later in time, and it is hard to get an overall picture of the progress made in ML tasks towards a specific goal, especially in larger teams. Simply storing the artifacts (datasets, models, feature sets, predictions) produced during experimentation in a central place is unfortunately insufficient to mitigate this situation. Achieving repeatability and comparability of ML experiments forces one to understand the metadata and, most importantly, the provenance of artifacts produced in ML workloads [14]. For example, in order to re-use a persisted model, it is not sufficient to restore its contents byte by byte; new input data must also be transformed into a feature representation that the model can understand, so information on these transforms must also be persisted. As another example, in order to reliably compare two experiments, we must ensure that they have been trained and evaluated using the same training and test data respectively, and that their performance was measured by the same metrics.

To address the aforementioned issues and assist data scientists in their daily tasks, we propose a lightweight system for handling the metadata of ML experiments. This system allows for managing the metadata (e.g., *Who created the model at what time? Which hyperparameters were used? What feature transformations have been applied?*) and lineage (e.g., *Which dataset was the model derived from? Which dataset was used for computing the evaluation data?*) of produced artifacts, and provides an entry point for querying the persisted metadata. Data scientists can leverage this service to enable a variety of previously hard-to-achieve functionality, such as regular automated comparisons of models in development to older models (similar to regression tests for software). Additionally, the proposed service helps data scientists to easily ad-hoc test their models in development and provides

Machine Learning Systems Workshop at NIPS 2017, Long Beach, CA, USA.

a starting point for quantifying the accuracy improvements that teams achieve over time towards a specific ML goal, e.g., by storing and analyzing the evaluation results of their models over time and showing them via a leaderboard.

In order to ease the adoption of our metadata tracking system, we explore techniques to automatically extract experimentation metadata from common abstractions used in ML pipelines, such as 'data frames' which hold denormalized relational data, and ML pipelines which comprise a way to define complex feature transformation chains composed of individual operators. For applications built on top of these abstractions, metadata tracking should not require more effort than exposing a few data structures to our tracking code. As already mentioned, we view the establishment of a central metadata repository as a foundation for further functionality to be built on top of. We especially think that such a repository of historical experimentation data would pave the way for advanced meta learning. The contributions of this paper are as follows:

- We introduce the design decisions for our system and present a database schema for modeling the corresponding metadata across a variety of ML frameworks (Section 2).
- We detail how to extract metadata describing ML models defined in the abstractions of popular ML frameworks such as pipelines and computational graphs of neural networks, and describe how to store this metadata using our database schema (Section 3).
- We discuss directions and requirements for future ML frameworks in order to streamline the metadata capturing process (Section 5).

2 System Design

We introduce the architecture of our system and discuss the underlying database schema which allows us to store general, declarative representations of the ML experiments.

Architecture. We briefly describe the architecture of our system as illustrated in Figure 1. The experimentation metadata is stored in a document database and our system backend runs in a serverless manner on AWS. Clients communicate with this backend via a *REST* API, for which we provide autogenerated clients for a variety of languages. We currently offer low-level REST-based clients for the JVM and Python, as well as high-level clients that are geared towards popular ML libraries such as *SparkML* [15], *scikit-learn* [18], *MXNet* [5] and *UIMA* [9]. These high-level clients provide convenience functionality for the libraries, especially in the form of automated metadata extraction from internal data structures. Examples include extracting the structure of pipelines and DataFrames in SparkML or the symbolic network definition in MXNet. The metadata and provenance information is consumed by applications such as dashboards, leaderboards, email notifiers and regression tests against historical evaluation results, and can also be queried interactively by individual users. Furthermore, we have means for importing publicly available experimentation data from repositories such as *OpenML* [25].

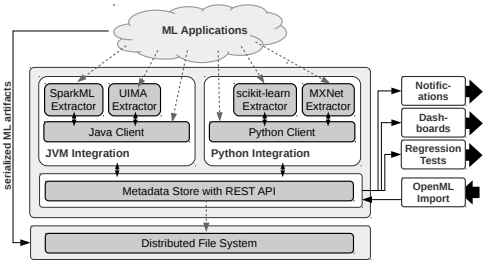

Figure 1: System architecture: ML applications store serialized artifacts on a distributed filesystem and use framework-specific extraction libraries to store the corresponding metadata in our centralized system. The resulting metadata is consumed by reporting and regression testing applications.

Data model. The major challenge in designing a data model for experimentation metadata is the *trade-off between generality and interpretability of the schema*. The most general solution would be to simply store all data as bytes with arbitrary tags in key-value form attached. Such metadata however would be very hard to automatically interpret and analyze later, as no semantics are enforced. A too narrow schema on the other hand might hinder adoption of our service, as it does not allow scientists to incorporate experimentation data from a variety of use cases.

We propose a middle ground with a schema that strictly enforces the storage of lineage information (e.g., which dataset was used to train a model) and ML-specific attributes (e.g., hyperparameters of a model), but still provides flexibility to its users by supporting arbitrary application-specific annotations. This choice is inspired by 'open' data models offered in modern data processing

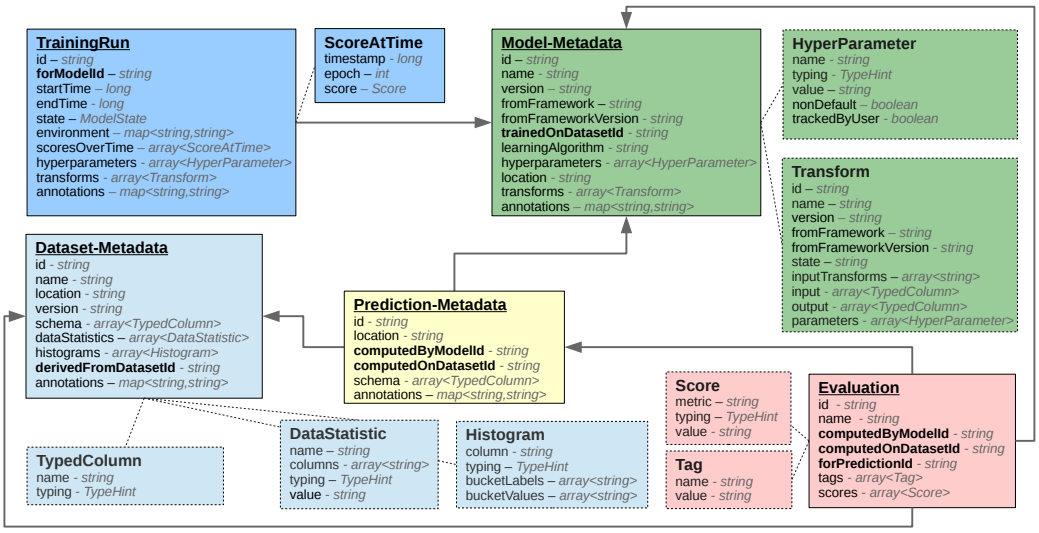

Figure 2: Simplified version of our schema used to store artifact metadata and lineage information. A detailed version available under Apache license can be found at `https://github.com/awslabs/ml-experiments-schema`. Bold attributes indicate lineage relationships and every entity can be extended via arbitrary key-value pairs stored in the annotation attribute.

systems [1]. The most important principle we embrace is *declarativity*: we store metadata of the artifacts but not code that produces them[1], and only store pointers to the actual input data or serialized model parameters. This enforces a strict decoupling, and enables querying and analysis of the metadata and lineage. Declarativity allows us to avoid storing compiled code, which is dangerous, as it moves this code out of control of the versioning and reviewing system, and makes it very hard to fix potential bugs. Another important principle for our service is *immutability*: metadata entries are only written once, ruling out a variety of potential consistency problems. Furthermore, immutability of written entries contributes to establishing a historical record of experiments.

We manage the metadata of the following experiment artifacts: *datasets*, *models*, *predictions*, and *training runs*. Furthermore, we explicitly manage the *evaluation* results of the experiments. Figure 2 illustrates the corresponding schema of the database of our service. The metadata for datasets is captured by the *Dataset-Metadata* entity, which stores name and version of the dataset, its column names and their respective types, as well as a URI pointer to the actual storage location of the dataset. Models are represented by the *Model-Metadata* entity which stores a link to the metadata of the input dataset to capture the lineage of the computation. We provide a name for the model, its hyperparameters and their types, as well as the name of the associated learning algorithm and its version. Furthermore, we represent the metadata of the data transformations describing the transformation of the input data by this model via a graph of *Transform* entities. Note that we do not explicitly represent the model's parameters in order to ensure generality of our system; we just store a pointer to a serialized version of the parameters. Additionally, a model can have an associated *Training-Run* entity (the only mutable entity in our schema), which tracks live execution and captures statistics like the training loss over time and the computational environment. The metadata for predictions produced by a model is represented by the *Prediction-Metadata* entity. Here we retain the lineage of the artifacts by storing links to the model which produced the predictions as well as to the dataset for which the predictions were computed. Additionally, we save the schema of the prediction data as well as a pointer to the actual, serialized prediction data. Evaluation data is represented by the *Evaluation* entity, for which we allow users to store customizable scores and tags. This entity has an optional link to the corresponding prediction entity, and mandatory pointers to the metadata entries for the model and evaluation dataset.

[1] We advise users to manage their ML code in git and use the git commit id to version the corresponding metadata.

3 Automating Metadata Extraction

In the following, we discuss our metadata extraction functionality for two different types of ML workloads: parameterized 'pipelines' in SparkML and scikit-learn which operate on tabular data and neural networks defined in MXNet via symbolic computational graphs that operate on tensor data.

Dataframes and pipelines in SparkML. ML workloads in SparkML operate on `DataFrames`, a relational abstraction for a partitioned (and often denormalized) table with a well-defined schema (modeled via Spark's `StructType` abstraction). The transformations applied to the data are modeled as stages of a pipeline (via the `PipelineStage` abstraction), inspired by scikit-learn's popular pipeline abstraction. Typically, such a pipeline consists of preprocessing and feature transformation operations, followed by a model training stage. After being 'fitted' to data, a pipeline produces a so-called `PipelineModel`, which represents a trained model with a fixed feature transformation procedure that can be applied to new data. The architecture of SparkML pipelines allows us to automatically track all the schema transformations (e.g, reading, adding and removing columns) which pipeline operators conduct, as well as the parameterization of the operators. The abstraction however does not allow us to inspect the internal implementation details (e.g., the mathematical operations applied by ML models), as pipeline operator implementations have been designed as black boxes. In order to track the data transformations applied by the pipeline in detail, we extract the schema of the input data frame and replay the changes the pipeline conducts by calling the `transformSchema` method of each pipeline stage and recording the resulting schema changes. Furthermore, we inspect certain pipeline stage parameters (modeled via Spark's `Params` class) like `HasInputCol` or `HasFeaturesCol` which denote which columns of the dataframe an operator consumes as input. This allows us to create a directed acyclic graph (DAG) representation of the data transformations applied by the pipeline, where the vertices of the DAG correspond to dataframe columns and the edges denote pipeline operators which consume said columns to produce additional columns. We model these DAGs via a graph of *Transformation* entities from our database schema and provide them to users for visual inspection of their data processing pipelines. Furthermore we inspect and store the parameterization of each pipeline operator, including the hyperparameters of models such as `maxIter` denoting the maximum number of iterations to run when learning a regression model.

Pipelines in scikit-learn. We process scikit-learn's pipelines (`sklearn.pipelines`) analogous to SparkML pipelines, with the difference that we cannot recreate the schema transformations, as these are not included in the `Pipeline` abstraction (which not necessarily operates on dataframe-like inputs). We are however able to extract a data transformation DAG (created by combining pipeline operators with `FeatureUnion`) and pipeline operator hyperparameters by inspecting the result of the `get_params()` method on pipeline stages.

Deep neural networks in MXNet. Tracking workloads employing deep neural networks (DNNs) lets us apply a more detailed extraction than in the previous cases. The reason for this is that DNN frameworks offer their users a very fine-grained abstraction to declaratively define their models by combining mathematical operators (e.g., linear layers, convolutions, activation functions) into the layout of the network to learn. Therefore, the models in DNN use cases are not black boxes (as in SparkML or scikit-learn), where the actual implementation details are hidden deeply in the code. We use the popular DNN engine MXNet [5] as an example and extract metadata about the models as follows: A DNN is modeled as a DAG of symbols representing the operators to apply (often this is called the computational graph of the network). This graph is typically given to a `Module` abstraction which represents a model combined with a learning algorithm and its hyperparameters, which can be trained on data. We get hold of the underlying computational graph by inspecting its output symbols from the module and call `symbol.tojson()` to retrieve a declarative representation of the DNN. We recreate the layout of the graph by inspecting the `op`, `input`, `name` and `attr` attributes of the graph operators, and convert the layout to a DAG of *Transform* entities in our database schema. We peek into the input data to extract its dimensionality and have MXNet infer the input and output dimensions of each operator via a call to `.infer_shape_partial` on the symbols of the graph. We store these shapes as parameterizations of the network operators. Furthermore, we track the hyperparameters of the applied learning algorithm by inspecting the corresponding optimizer settings in `module._optimizer`.

Example. We show how to enable metadata tracking for a classification model trained with MXNet. The code for this example is illustrated in Figure 3, where the highlighted parts refer to code

```
1  mod = mx.mod.Module(mlp)
2  run = experiment
3   .track_training(mod)
4
5  mod.fit(
6   train_data=trainset,
7   eval_data=validationset,
8   epoch_end_callback=
9    run.log_train_metric(),
10  eval_end_callback=
11   run.log_eval_metric())
12
13 experiment.track_model(mod,
14  name='MXNet ...', trained_on=
15  trainset_metadata,
16  training=run, serialize=True)
17
18 accuracy = mx.metric.Accuracy()
19 mod.score(testset, 'acc')
20
21 experiment.track_evaluation(
22  by_model=model_metadata,
23  on_dataset=testset_metadata,
24  scores=accuracy)
```

Figure 3: Tracking training loss (in realtime), model metadata and evaluation results in MXNet.

Figure 4: User interface with access to training loss over time, model hyperparameters, network layout and specifics of computing environment.

from our MXNet specific metadata extraction library. We assume that the dataset has already been registered with our service (whose API is available via the `experiment` variable) and the corresponding metadata entries for the trainset and testset have been loaded from our system as `trainset_metadata` and `testset_metadata`. In the first line, an MXNet module is created from the network definition `mlp`. Afterwards, a *Training-Run* for this module is registered with our system via the call to `track_training`. In the lines 5 to 11, the training for the module is conducted via its `fit` method. Note that we register two callback functions with the module in lines 9 and 11, which allow our previously created training run object to observe the training and validation loss of each epoch. These metrics are communicated to our system in a separate thread and allow the user executing the training to monitor the training in real time via an external dashboard application shown in Figure 4. Additionally, the training run automatically captures information about the computing environment, e.g. the type of processor used.

After having successfully trained the model, we register it with our system via the `track_model` call in lines 13 to 16. Users give a name to the model and again have to provide the metadata entry for the corresponding trainingset via the `trained_on` argument (in order to enable our system to record the lineage). Furthermore, they can specify that the resulting model should be serialized, in which case we store the model in a distributed filesystem and add a pointer to its storage location in the corresponding *Model-Metadata* entry in our database. Note that the implementation of `track_model` understands MXNet models and will automatically extract hyperparameters and network layout as previously discussed. Again, all extracted metadata is presented and visualized to users via the dashboard shown in Figure 4. Finally we evaluate the classification accuracy of our model on the testset via the `mod.score` call in lines 18 and 19. We store these evaluation results in our system with the `track_evaluation` call, which will create an *Evaluation* entry in our database. Note that we force users to specify both the metadata of the corresponding model (the `computed_by` argument) as well as the metadata of the test dataset used (the `on_dataset` argument), so that our system can track the provenance of the evaluation data by linking the corresponding *Model-Metadata*, *Dataset-Metadata* and *Evaluation* entries in our database. Retaining such lineage information is crucial when we have to decide later whether two evaluation results can be reliably compared.

4 Related Work

In order to foster collaboration between scientists, platforms such as *OpenML* [25] and the *W3C ML Schema* initiative [12] allow researchers to share descriptions and evaluation results of their ML experiments. However, these approaches do not model the artifacts produced during experiments (and

their lineage) in the same detail as our system; for example, there is no entity for modeling the DAG structure of feature transformation flows. Thereby, they partially lose potential for building automated systems that leverage the historical data. Managing and efficiently executing model selection workloads has been identified as an upcoming challenge [14, 13, 21] in the data management community. The *ModelDB* [26] project puts a specific focus on organizing models, and comes very close to our system design-wise with the difference that we support more general classes of models (such as neural networks) and add more detailed tracking of dataset provenance and applied schema transformations. Other classes of systems specialize on deep learning [17], aim at efficiently serving the resulting models for prediction [6, 7] or concentrate on tracking and indexing provenance information [16, 19, 8]. Modeling ML workloads via pipelines (which are typically inspired by the 'estimator/transformer' abstraction in *scikit-learn* [18]) and efficiently executing such pipelines at scale has become an active area of research. Established work describes production systems and platforms [4, 3], investigates software engineering aspects [22, 24] and pipeline abstractions for machine learning workloads [2], often on top of the massively parallel dataflow system *Apache Spark*, e.g., in *SparkML* [15] and *KeystoneML* [23]. An exciting area of research is the investigation of the potential of historical experimentation data for meta learning [11, 10].

5 Conclusion

We have presented our design choices for a lightweight system to extract and store metadata and provenance of ML experimentation data. Furthermore, we discussed in detail how to automatically extract such data from common abstractions in publicly available ML frameworks. We see a huge potential in enabling declarative management of said metadata: data scientists are provided with infrastructure that allows them to accelerate their experimentation via dashboards and leaderboards that list experiments, email notifications which summarize experimentation progress, and automated regression tests for the prediction quality of ML libraries during development, which compare the results on hold out data with historical prediction results upon every commit to the library. Systems like ours will allow companies to accelerate their experimentation and innovation cycle, and, at the same time, form a corner stone of replicable ML model training, which will become more important in the light of potentially upcoming legal requirements for the real-world usage of machine learning.

Limitations. A limitation of our approach is that the tracked ML application must be able to identify the logical operations which are applied to the data. In legacy applications, this might not necessarily be the case unfortunately, as the developers might not have introduced the necessary abstractions. In such cases, hand-tailored code is usually required to get the desired metadata. We think that the design of future ML frameworks should place a focus on a clear separation of logical and physical operations and build on existing abstractions for modeling ML pipelines. Additionally, external systems like ours should be able to query the metadata of the defined logical operations. As discussed in Section 3, the declarative definition of computational graphs for DNNs from mathematical operators is a prime example of how to achieve this separation. Furthermore, we currently rely on users understanding our complex schema and correctly integrating their code with our API. We aim to increase the automation of our extraction code and to decrease the amount of additional code required to enable the metadata tracking in a workload.

Ongoing and future work. During integrations of various use cases, we have come across a set of challenges, which we will address in the near future. Examples include the integration of ensembles that train a variety of models on different datasets and combine them in a complex final model, or how to model the provenance relations between the training sets and validation sets during cross-validation in our schema. Furthermore, we already encountered use cases where it would be beneficial to extend our data model with an enclosing abstraction for the produced artifacts (an 'experiment session'), e.g., in cases, where we train several models which we want to identify again later to compare them against each other. Users achieve this manually via dedicated entity annotations at the moment. Our long term research goals include the enablement of meta learning on top of our experiment repository, with the goal of recommending features or algorithms for new datasets. In order to achieve progress in this area, it might be necessary to introduce a 'task' abstraction for experiments, analogous to what OpenML provides – which can be quite difficult for real world use cases, where there is often no clear single target metric to optimize for [4].

Acknowledgements. We would like to thank our interns Jean-Baptiste Cordonnier and Minori Inoue for their contributions, and Matthias Seeger for fruitful discussion on the initial ideas for the system.

References

[1] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, et al. Asterixdb: A scalable, open source BDMS. *PVLDB*, 7(14):1905–1916, 2014.

[2] Pierre Andrews, Aditya Kalro, Hussein Mehanna, and Alexander Sidorov. Productionizing Machine Learning Pipelines at Scale. *Machine Learning Systems workshop at ICML*, 2016.

[3] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, et al. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *KDD*, pages 1387–1395, 2017.

[4] Joos-Hendrik Böse, Valentin Flunkert, Jan Gasthaus, Tim Januschowski, Dustin Lange, David Salinas, Sebastian Schelter, Matthias Seeger, and Yuyang Wang. Probabilistic Demand Forecasting at Scale. *PVLDB*, 10(12):1694–1705, 2017.

[5] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *Machine Learning Systems workshop at NIPS*, 2015.

[6] Daniel Crankshaw, Peter Bailis, Joseph E Gonzalez, Haoyuan Li, Zhao Zhang, Michael J Franklin, Ali Ghodsi, and Michael I Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. *CIDR*, 2015.

[7] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, pages 613–627, 2017.

[8] Deepsense.ai. Neptune - machine learning lab, https://neptune.ml/, 2017.

[9] David Ferrucci and Adam Lally. Uima: an architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering*, 10(3-4):327–348, 2004.

[10] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *NIPS*, pages 2962–2970, 2015.

[11] Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. Initializing Bayesian Hyperparameter Optimization via Meta-Learning. In *AAAI*, pages 1128–1135, 2015.

[12] Machine Learning Schema Community Group. W3c machine learning schema, 2017.

[13] Joseph M Hellerstein, Vikram Sreekanti, Joseph E Gonzalez, James Dalton, Akon Dey, Sreyashi Nag, Krishna Ramachandran, Sudhanshu Arora, Arka Bhattacharyya, Shirshanka Das, et al. Ground: A data context service. In *CIDR*, 2017.

[14] Arun Kumar, Robert McCann, Jeffrey Naughton, and Jignesh M Patel. Model Selection Management Systems: The Next Frontier of Advanced Analytics. *SIGMOD Record*, 2015.

[15] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *JMLR*, 17(34):1–7, 2016.

[16] Hui Miao, Ang Li, Larry S Davis, and Amol Deshpande. On model discovery for hosted data science projects. In *Workshop on Data Management for End-to-End Machine Learning at SIGMOD*, page 6, 2017.

[17] Hui Miao, Ang Li, Larry S Davis, and Amol Deshpande. Towards unified data and lifecycle management for deep learning. In *ICDE*, pages 571–582, 2017.

[18] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in Python. *JMLR*, 12:2825–2830, 2011.

[19] Joao Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. noworkflow: a tool for collecting, analyzing, and managing provenance from python scripts. *VLDB*, 10(12):1841–1844, 2017.

[20] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. Data management challenges in production machine learning. In *SIGMOD*, pages 1723–1726. ACM, 2017.

[21] Sebastian Schelter, Juan Soto, Volker Markl, Douglas Burdick, Berthold Reinwald, and Alexandre Evfimievski. Efficient sample generation for scalable meta learning. In *ICDE*, pages 1191–1202, 2015.

[22] D Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *NIPS*, pages 2503–2511, 2015.

[23] Evan R Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J Franklin, and Benjamin Recht. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. *ICDE*, 2017.

[24] Tom van der Weide, Dimitris Papadopoulos, Oleg Smirnov, Michal Zielinski, and Tim van Kasteren. Versioning for end-to-end machine learning pipelines. In *Workshop on Data Management for End-to-End Machine Learning at SIGMOD*, page 2, 2017.

[25] Joaquin Vanschoren, Jan N Van Rijn, Bernd Bischl, and Luis Torgo. OpenML: networked science in machine learning. *SIGKDD*, 15(2):49–60, 2014.

[26] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. ModelDB: A System for Machine Learning Model Management. In *Workshop on Human-In-the-Loop Data Analytics at SIGMOD*, pages 14:1–14:3, 2016.

[27] Martin Zinkevich. Rules of machine learning: Best practices for ml engineering, 2017.