Bulletin of the Technical Committee on

Data Engineering

December 2018 Vol. 41 No. 4

IEEE Computer Society

Letters

Special Issue on Machine Learning Life-cycle Management

Conference and Journal Notices

Farewell

It was way back in 1992 that Rakesh Agrawal, then the TCDE Chair, appointed me as Editor-in-Chief of the Data Engineering Bulletin. At the time, I saw it as a great opportunity. But it did not occur to me that it would become such an enormous part of my career. Now, 26 years later, it is time, perhaps past time, for me to pass this position on to younger hands, in this case to the capable hands of Haixun Wang. It should not come as a surprise that I am stepping down. Rather, the surprise should be "why did I stay so long?" This message is a combination of answer to that question and historical sketch of my time as EIC. These are not unrelated.

When I first became EIC, the Bulletin had already established a reputation as an industry and engineering focused publication, each issue of which was on a special topic. Won Kim, my predecessor, had very capably established that publication model. Papers are solicited by each issue editor, with the editor selecting which authors to invite. The papers are a mix of work in progress, position statements, surveys, etc. But all focused on the special topic. I was determined not to screw this up. Indeed, I accepted the EIC appointment because I believed that the role that the Bulletin played is unique in our database community. I stayed so long because I still believe that.

Over the years, the Bulletin went through several major changes. As early as 1993, the Bulletin could be accessed online as well as via print subscription. This was a major transition. Mark Tuttle, then a colleague of mine in Digital (DEC) Cambridge Research Lab designed the latex style files that enabled this. Shortly thereafter, to economize on costs, the Bulletin became one of the earliest all electronic publications in our field.

In 1995, hosting the Bulletin web site was provided by Microsoft- continuing until three years ago. Around 2010, the IEEE Computer Society became the primary host for the Bulletin. Around 2000, at the suggestion (prodding) of Toby Lehman,individual articles in addition to complete issues were served from the Bulletin web sites. Over this time, the style files and my procedures for generating the Bulletin evolved as well. Mark Tuttle again, and S. Sudarshan, who had been a Bulletin editor, provided help in evolving procedures used to generate the Bulletin and its individual articles.

The Computer Society, and specifically staff members John Daniel, Carrie Clark Walsh, and Brookes Little, provided a TCDE email membership list used to distribute issue announcements, as well as helping in myriad other ways. The existence of dbworld (one of Raghu Ramakrishnan enduring contributions) enabled wider announcement distribution to our database community. The cooperation of Michael Ley with the prompt indexing of the Bulletin at dblp both ensured wider readership and provided an incentive for authors to contribute. Over the years, I was given great support by TCDE Chairs, starting with Rakesh Agrawal, then Betty Salzberg, Erich Neuhold, Paul Larson, Kyu-Young Whang, and Xiaofang Zhou.

The most important part of being Bulletin EIC was the chance to work with truly distinguished members of the database community. It was enormously gratifying to have stars of our field (including eight Codd Award winners- so far) serving as editors. I take pride in appointing several of them as editors prior to their wider recognition. It is the editors that deserve the credit for producing, over the years, a treasure trove of special issues on technologies that are central to our data engineering field. Superlative editors, and their success in recruiting outstanding authors, is the most important part of the Bulletin's success. Successfully convincing them to serve as editors is my greatest source of pride in the role I played as Bulletin EIC.

Now I am happy to welcome Haixun to this wonderful opportunity. Haixun's background includes outstanding successes in both research and industry. He recently served ably as a Bulletin associate editor for issues on "Text, Knowledge and Database" and "Graph Data Processing". His background and prior editorial experience will serve our data engineering community well and ensure the ongoing success of the Bulletin. I wish him and the Bulletin all the best.

And so "farewell". I will always treasure having served as Bulletin EIC for so many years. It was a rare privilege that few are given. Knowing that we were reaching you with articles that you found valuable is what

has made the effort so rewarding to me personally. Thank you all for being loyal readers of the Bulletin.

David Lomet
Microsoft Corporation

Letter from the Editor-in-Chief

Thank You, David!

I know I represent the readers, the associate editors, and also the broad database community when I say we are extremely grateful to David Lomet for his distinguished and dedicated service as the Editor-in-Chief of the Data Engineering Bulletin for the last 26 years.

Since its launch in 1977, the Bulletin has produced a total of 154 issues. Reading through the topics of the past issues that spanned more than four decades makes me feel nothing short of amazing. They show not just how far the database research has come, but to a certain extent, how much the entire field of computer science and the IT industry have evolved. While important topics never fail to arise in the Bulletin in a timely fashion, it is also interesting to observe in the 154 issues many recurring topics, including query optimization, spatial and temporal data management, data integration, etc. It proves that the database research has a solid foundation that supports many new applications, and at the same time, it demonstrates that the database research is constantly reinventing itself to meet the challenges of the time. What the Bulletin has faithfully documented over the last 42 years is nothing else but this amazing effort.

Among the 154 issues since the launch of the Bulletin, David had been the Editor-in-Chief for 103 of them. This itself is a phenomenal record worth an extra-special celebration. But more importantly, David shaped the discussions and the topics in the long history of the Bulletin. I had the honor to work with David in 2016 and 2017 when I served as the associate editor for two Bulletin issues. What was most appealing to me was the opportunity of working with the top experts on a topic that I am passionate about. The Bulletin is truly unique in this aspect.

I understand the responsibility and the expectation of the Editor-in-Chief, especially after David set such a great example in the last 26 years. I thank David and the associate editors for their trust, and I look forward to working with authors, readers, and the database community on the future issues of the Data Engineering Bulletin.

The Current Issue

Machine learning is changing the world. From time to time, we are amazed at what a few dozen lines of python code can achieve (e.g., using PyTorch, we can create a simple GAN in under 50 lines of code). However, for many real-life machine learning tasks, the challenges lie beyond the dozen lines of code that construct a neural network architecture. For example, hyperparameter tuning is still considered a "dark art," and having a platform that supports parallel tuning is important for training a model effectively and efficiently. Model training is just one component in the life cycle of creating a machine learning solution. Every component, ranging from data preprocessing to inferencing, requires just as much support on the system and infrastructure level.

Joseph Gonzalez put together an exciting issue on the life cycle of machine learning. The papers he selected focus on systems that help manage the process of machine learning or resources used in machine learning. They highlight the importance of building such supporting systems, especially for production machine learning platforms.

Haixun Wang
WeWork Corporation

Letter from the Special Issue Editor

Machine learning is rapidly maturing into an engineering discipline at the center of a growing range of applications. This widespread adoption of machine learning techniques presents new challenges around the management of the data, code, models, and their relationship throughout the machine learning life-cycle. In this special issue, we have solicited work from both academic and industrial leaders who are exploring how data engineering techniques can be used to address the challenges of the machine learning life-cycle.

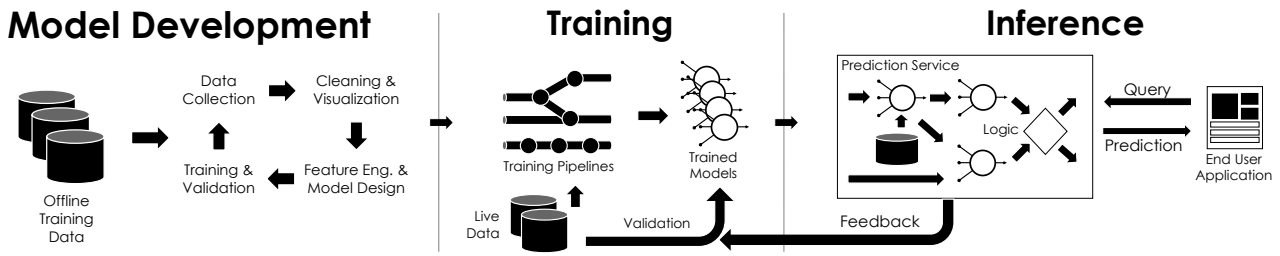


Figure 1: **Machine Learning Life-cycle.** A simplified depiction of the key stages of a machine learning application.

The machine learning life-cycle (Fig. 1) spans not only the model development but also production training and inference. Each stage demands different skills (e.g., neural network design, data management, and cluster management) and imposes different requirements on the underlying systems. Yet there is an overwhelming need for unifying design principles and technologies to address pervasive problems including feature management, data provenance, pipeline reproducibility, low-latency serving, and prediction monitoring just to name a few.

There has been a flurry of recent progress in systems to aid in managing the machine learning life-cycle. Large industrial projects like FB Learner Flow from Facebook, Michelangelo from Uber, and TFX from Google have received considerable recent attention. In this issue, we have solicited papers from several recent industrial and academic projects that have received slightly less attention.

The first paper provides an overview of several real-world use cases and then outlines the key conceptual, data management, and engineering challenges faced in production machine learning systems. The second and third papers explores the challenges of model management and provenance across the machine learning life-cycle. They motivate the need for systems to track models and their meta-data to improve reproducibility, collaboration, and governance. The second paper introduces, ModelDB, an open-source system for model management and describe some of the functionality and design decisions. The third paper describes a related system, ProvDB, that uses a graph data model to capture and query fine-grained versioned lineage of data, scripts, and artifacts throughout the data analysis process. The fourth paper describes, MLFlow, a new open-source system to address the challenges of experimentation, reproducibility, and deployment. This work leverages containerization to capture the model development environment and a simple tracking API to enable experiment tracking. The fifth paper focuses on inference and explores the challenges and opportunities of serving white-box prediction pipelines. Finally, we solicited a summary of the recent Common Modeling Infrastructure (CMI) workshop at KDD 2018, which provides a summary of the keynotes and contributed talks.

The work covered here is only a small sample of the emerging space of machine learning life-cycle management systems. We anticipate that this will be a growing area of interest for the data engineering community.

Joseph E. Gonzalez
University of California at Berkeley
Berkeley, CA

On Challenges in Machine Learning Model Management

Sebastian Schelter, Felix Biessmann, Tim Januschowski,
David Salinas, Stephan Seufert, Gyuri Szarvas
{sseb,biessman,tjnsch,dsalina,seufert,szarvasg }@amazon.com
Amazon Research

Abstract

The training, maintenance, deployment, monitoring, organization and documentation of machine learning (ML) models – in short model management – is a critical task in virtually all production ML use cases. Wrong model management decisions can lead to poor performance of a ML system and can result in high maintenance cost. As both research on infrastructure as well as on algorithms is quickly evolving, there is a lack of understanding of challenges and best practices for ML model management. Therefore, this field is receiving increased attention in recent years, both from the data management as well as from the ML community. In this paper, we discuss a selection of ML use cases, develop an overview over conceptual, engineering, and data-processing related challenges arising in the management of the corresponding ML models, and point out future research directions.

1 Introduction

Software systems that learn from data are being deployed in increasing numbers in industrial application scenarios. Managing these machine learning (ML) systems and the models which they apply imposes additional challenges beyond those of traditional software systems [18, 26, 10]. In contrast to textbook ML scenarios (e.g., building a classifier over a single table input dataset), real-world ML applications are often much more complex, e.g., they require feature generation from multiple input sources, may build ensembles from different models, and frequently target hard-to-optimize business metrics. Many of the resulting challenges caught the interest of the data management research community only recently, e.g., the efficient serving of ML models, the validation of ML models, or machine learning-specific problems in data integration. A major issue is that the behavior of ML systems depends on the data ingested, which can change due to different user behavior, errors in upstream data pipelines, or adversarial attacks to name just some examples [3]. As a consequence, algorithmic and system-specific challenges can often not be disentangled in complex ML applications. Many decisions for designing systems that manage ML models require a deep understanding of ML algorithms and the consequences for the corresponding system. For instance, it can be difficult to appreciate the effort of turning raw data from multiple sources into a numerical format that can be consumed by specific ML models – yet this is one of the most tedious and time consuming tasks in the context of ML systems [32].

In this paper, we introduce a set of intricate use cases in Section 2, and elaborate on the corresponding general challenges with respect to model management. We discuss conceptual challenges in Section 3.1, e.g.,

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

how to exactly define the model to manage [35], how to validate the predictions of a model both before and after deployment [3], and how to decide when to retrain a model. Next, we elaborate on data-management related challenges in Section 3.2. These evolve around the fact that ML models are part of larger ML pipelines, which contain the corresponding feature transformations and related metadata [31, 34]. Efficient model management requires us to be able to capture and query the semantics, metadata and lineage of such pipelines [37, 27, 36]. Unfortunately, this turns out to be a tough problem as there is no declarative abstraction for ML pipelines that applies to all ML scenarios. Finally, we list a set of engineering-related challenges, which originate from a lack of agreed-upon best practices as well as from the difficulties of building complex systems based on components in different programming languages (Section 3.3).

2 Selected Use Cases

Time Series Forecasting. Large-scale time series prediction or *forecasting* problems have received attention from the ML, statistics and econometrics community. The problem of forecasting the future values of a time series arises in numerous scientific fields and commercial applications. In retail, an accurate forecast of product demand can result in significant cost reductions through optimal inventory management and allocation. In modern applications, many time series need to be forecasted for simultaneously. Classical forecasting techniques, such as ARIMA models [7], or exponential smoothing and its state-space formulation [14] train a single model per time series. Since these approaches are typically fast to train [33, 15], it is possible to retrain the models every time a new forecast needs to be generated. In theory, this would mean that little to no management of the resulting models is required; however, the practice very often differs significantly. While classical forecasting models excel when time series are well-behaved (e.g., when they are regular, have sufficiently long history, are non-sparse), they struggle with many characteristics commonly encountered in real-world use cases such as cold-starts (new products), intermittent time series and short life cycles [30]. Such complications require models that can transfer information across time series. Even for well-behaved time series, we may often want to be able to learn certain effects (e.g., the promotional uplift in sales) across a number of products. Therefore, forecasting systems in the real-world become quite complex even if the classical models at its core are simple [6]. Maintaining, managing, and improving the required ML pipelines in such systems is a non-trivial challenge – not only in production environments which may require complex ensembles of many simple models, but especially in experimental settings when different individual models are evaluated on different time series. We found that a natural alternative to such complex ensembles of simple models is end-to-end learning via deep learning models for forecasting [11]. Neural forecasting models have the ability to learn complex patterns across time series. They make use of rich metadata without requiring significant manual feature engineering effort [13], and therefore generalize well to different forecasting use cases. Modern, general-purpose implementations of neural forecasting mdoels are available in cloud ML services such as *AWS SageMaker* [17]. However, when such neural forecasting models are deployed in long-running applications, careful management of the resulting models is still a major challenge, e.g., in order to maintain backwards compatibility of the produced models.

Missing Value Imputation. Very often, ML is leveraged to automatically fix data quality problems. A prominent issue in this context are missing values. A scenario in which missing values are problematic are product catalogs of online retailers. Product data needs to be complete and accurate, otherwise products would not be found by customers. Yet in many cases, product attributes may not be entered by sellers, or they might apply a different schema and semantic, which results in conflicting data that would need to be discarded. Manually curating this data does not scale, hence automatic solutions leveraging ML technology are a promising option for ensuring high data quality standards. While there are many approaches dealing with missing values, most of these methods are designed for matrices only. In many real-world use cases however, data is not available in numeric formats but rather in text form, as categorical or boolean values, or even as images. Hence most datasets

must undergo a feature extraction process that renders the data amenable to imputation. Such feature extraction code can be difficult to maintain, and not every data engineer that is facing missing values in a data pipeline will necessarily be able to implement or adapt such code. This is why implementing the feature extraction steps and the imputation algorithms in one single pipeline (that is learned end-to-end) greatly simplifies model management [4] for end users. An end-to-end imputation model that uses hyperparameter optimization to automatically perform model selection and parameter tuning can be applied and maintained by data engineers without in depth understanding of all algorithms used for feature extraction and imputation.

Content Moderation. Another broad set of ML-related tasks can be summarized under the umbrella of content moderation. There is a wide range of such use cases, some simple in their nature (e. g., the detection of swear words in content), while others are more complex, e. g., the detection of fake news, or the detection of copyright infringement in music or video. As an example, we focus onto user comments in online communities [24]. Before publishing user provided content, moderation might take place, often via a mixture of ML methods and human curators. Common tasks are to mark content which contains inappropriate language or does not adhere to a community standard. Such moderation consists of offline training of models using manually labelled data. However, during the production usage of such models, we need to constantly monitor the model accuracy. A typical approach looks as follows: content that can be classified with high accuracy by automated models does not require a human interaction. However, content for which the model output is inconclusive (for this, probabilistic classifiers are of utmost importance) are directly passed to human annotators. This data can then be used to re-train the model online in an active learning setting. Depending on their overall capacity, we can also randomly select sample content that was classified with high probability. Periodically, we should also select and relabel a completely random subset of the incoming content. In this way, we constantly update the model and improve its performance.

Automating Model Metadata Tracking. An orthogonal data management use case of great importance in all ML applications is tracking the metadata and lineage of ML models and pipelines. 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 [29, 6, 18, 26, 40, 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 models resulting from these experiments are often not comparable, since there is no standard way to determine whether two models had been trained on the same input data. It is often technically difficult to reproduce successful experiments later in time. To address the aforementioned issues and assist data scientists in their daily tasks, we built a lightweight system for handling the metadata of ML experiments [27]. 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. Such a system enables regular automated comparisons of models in development to older models, and thereby provides a starting point for quantifying the accuracy improvements that teams achieve over time towards a specific ML goal. The resulting database of experiments and outcomes can furthermore be leveraged later for accelerating meta learning tasks [12].

3 Model Management Challenges

The aforementioned use cases lead to a number of challenges that we discuss in the following subsections. We categorize the challenges broadly into three categories: (i) *conceptual challenges*, involving questions such

as what is part of a model, (ii) *data management challenges*, which relate to questions about the abstractions used in ML pipelines and (iii) *engineering challenges*, such as building systems using different languages and specialized hardware.

3.1 Conceptual Challenges

Machine Learning Model Definition. It is surprisingly difficult to define the actual model to manage. In the most narrow sense, we could consider the model parameters obtained after training (e.g., the weights of a logistic regression model) to be the model to manage. However, input data needs to be transformed into the features expected by the model. These corresponding feature transformations are an important part of the model that needs to be managed as well. Therefore, many libraries manage ML *pipelines* [25, 22] which combine feature transformations and the actual ML model in a single abstraction. There are already established systems for tracking and storing models which allow to associate feature transformations with model parameters [35, 37, 27]. Due to the i.i.d.-assumption inherent in many ML algorithms, models additionally contain implicit assumptions on the distribution of the data on which they are applied later. Violations of these assumptions (e.g., covariate shift in the data) can lead to decreases in prediction quality. Capturing and managing these implicit assumptions is tricky however, and systematic approaches are in the focus of recent research [3, 16, 28]. A further distinction from a systems perspective is whether the model is considered to be a 'black-box' with defined inputs and outputs (e.g., a docker image with a REST-API or a scikit-learn pipeline with transformers and estimators) or whether the model is presented in declarative form comprised of operations with known semantics (e.g., the computational graph of a neural network). Even along these outlined dimensions, the heterogeneity of ML models and the complexity of many real-world ML applications render the definition of ML models difficult. For example: How should one handle combinations of models (e.g., in ensembles when models from different languages or libraries are combined, or when an imputation model is part of the preprocessing logic of another model) or meta-models, e.g., meta-learning algorithms or neural architecture searches.

Model Validation. The ability to back-test the accuracy of models over time is crucial in many real-world usages of ML. Models evolve continuously as data changes, methods improve or software dependencies change. Every time such a change occurs, model performance must be re-validated. These validations introduce a number of challenges and pitfalls: When we compare the performance of ML models, we must ensure that the models have been trained and evaluated using the same training, test and validation set. Additionally, it is crucial that the same code for evaluating metrics is used throughout time and across different models to be able to guarantee comparability. The more experiments we conduct however, the more we risk to overfit on the test or validation data. A further practical problem in backtesting comes from long-running experiments and complex interactions between pipelines of models. A change in backtesting results is often hard to attribute to specific code and configuration changes. Furthermore, in long-existing, highly-tuned ML systems, it is unlikely that the overall accuracy improves significantly. Usually, a new model introduces an improvement but this improvement comes at some cost (such as longer prediction times), which makes wrapping-up release candidates challenging. Specific domains introduce additional challenges. For instance, the train/validation/test split that is required in virtually all ML evaluations is often conducted by randomly partitioning an initial dataset into given proportions, such as 80%, 10% and 10% of the data. In real-world scenarios, in particular in those where the i.i.d.-assumption often made in ML does not hold, we must partition the data more carefully to avoid leakage. For instance in the case of forecasting, it would not be appropriate to split the data per time-series. We need to make sure that no future information such as seasonality is revealed on a training set. Further practical considerations are to consider dynamic backtesting scenarios where we conduct rolling evaluations over time to ensure that our backtest results do not depend on a specific point in time where we performed the split. Each evaluation should track such information in order to always be able to assert that no data leakage happened in a previous evaluation.

In other use cases that involve classification models, such as the imputation of missing values, naive train/test

splits can lead to problems as well. Real world datasets often have very skewed distributions of categories. Few categories are observed very often, but many categories are observed only very rarely. Conducting a naive cross-validation split often results in training sets or test sets that contain not enough data for single categories to either train a sensible model or to get a reasonable estimate of the generalization error.

Apart from validation of models in offline backtests, assessing the performance of models in production is crucial. Making implicit assumptions about data explicit and enabling automatic data validation [28] is even more important in online scoring scenarios. When launching a new model, we can validate its effectiveness in various degrees. Launching the model in "shadow mode" (i.e., routing part of the data to the model, but not consuming its output) often helps to catch operational problems. Whenever possible, one should conduct rolling comparisons between ground truth and predictions. A/B or bandit tests are often employed to explose a model in an end-user experience. Note that for forecasting, A/B tests can be quite challenging due to the long and indirect feedback cycles. Independent of the model launch mechanism, sanity-checking the model output is best practice. Since having high-quality sanity checks in place is as difficult as solving the original ML problem typically, we can employ simple, cheap-to-run or robust prediction models for comparisons, work with thresholding or randomized human inspection. If the core models allows to quantify its reliability via probabilities, this helps in making more informed choices on the model validity.

Decisions on Model Retraining. Deciding when to retrain a model is challenging. Training usually is done offline and models are loaded at prediction/inference time in a serving system. If new events occur that our models have not been trained on, we need to trigger retraining or even re-modelling. In the retail demand forecasting domain, examples of such events are the introduction of a new public holiday, a new promotional activity or a weather irregularity. Detecting change in the data is critical, as issues and changes in datasets drastically affect the quality of the model and can invalidate evaluations. When such data changes impact the final metrics (as noticed through backtests), backtracking the root-cause to the dataset is only possible when dataset metadata and lineage have been recorded beforehand. Similar challenges occur in the imputation use case. Standard cross-validation techniques help to estimate the generalization error and tune precision thresholds for missing value imputation models. But in the presence of noisy training data, it is important to always have some fraction of the predictions audited by human experts and mark imputations as such. When humans are in the loop of such a system, interesting model management challenges can emerge. If auditors are able to update the training data with their audited data, they may often tend to solidarize with the ML pipeline in that they try to increase the model performance as much as they can, which can lead to severe overfitting. As a consequence, everyone involved in the model lifecycle should have a minimal understanding of the caveats of ML systems, such as overfitting. Furthermore, such cases show that it is important to carefully control evaluation and training samples, and guaranteeing this in continuously changing data pipelines can be quite challenging.

Adversarial Settings. In our exemplary scenario for content moderation in communities, an adversarial setting can occur where malicious users try to understand the boundary conditions of the classifier. For example, users could try to reverse engineer the thresholds under which content is automatically classified and then try to exploit weaknesses of the classifier. In such settings, incoming data will shift over time, which requires careful monitoring of the input and the distribution of the output probabilities. We would assume that more and more content contributions are close to the prediction threshold so that the distribution shifts over time. Once such a distribution shift is deemed to be severe enough, one needs to retrain the model and for this, must obtain new labeled data. Another approach is to rely on active learning. Given a model that outputs a probabilistic classification, it can be constantly updated by acquiring (manually labeled) data from both the decision boundary as well as randomly picked instances. Modeling the adversarial actions themselves by another model is possible in theory and certainly desirable as it eliminates the need to keep labelling data, but in practice, this is often extremely difficult, therefore careful monitoring of the deployed models is crucial.

3.2 Data Management Challenges

Lack of a Declarative Abstraction for the whole ML Pipeline. A major success factor of relational databases is that that their data processing operations are expressed in a declarative language (SQL) with algebraic properties (relational algebra). This allows the system to inspect computations (e.g., for recording data provenance) and to rewrite them (e.g., for query optimization). Unfortunately, there is a lack of a declarative abstraction for end-to-end ML pipelines. These comprise of heterogenous parts (e.g., data integration, feature transformation, model training) which typically apply different operations based on different abstractions. Loading and joining the initial data applies operations from relational algebra. Feature transformations such as one-hot-encoding categorial variables, feature hashing, and normalization often apply map-reduce like transformations, composed in a 'pipeline'-like abstraction popularized by scikit-learn [25, 22, 31]. The actual model training phase uses mostly operations from linear algebra which in many cases require specialized systems and/or hardware to be run efficiently [2, 9]. While general dataflow systems such as Apache Spark support all of the listed phases, it turns out that the dataflow abstraction is very inefficient for many advanced analytics problems [21, 5] and most models are implemented as black-boxes in such systems. This led to the development of specialized systems, such as engines for training deep neural networks, that are tailored towards certain learning algorithms (mini-batch stochastic gradient descent) and very efficiently leverage specialized hardware such as GPUs. Unfortunately, these systems do not support general relational operations, e.g., they lack join primitives. As a consequence, different systems need to be "glued" together in many real-world ML deployments, and often require external orchestration frameworks that coordinate the workloads. This situation has tremendous negative effects for model management tasks: It complicates the extraction of metadata corresponding to a particular pipeline (e.g., because the hyperparameters of feature transformations are contained the relational part of pipeline, while hyperparameters for the model training part are found in the computational graph of a neural network), it makes replicability and automation of model selection difficult as many different systems have to be orchestrated, and as a consequence makes it hard to automate model validation.

Querying Model Metadata. In order to automate and accelerate model lifecycle management, it is of crucial importance to understand metadata and lineage of models (e.g., their hyperparameters, evaluation scores as well as the datasets on which they were trained and validated). This metadata is required for comparing models in order to decide which one to put into production, for getting an overview of team progress, and for identifying pain points as a basis for deciding where to invest tuning efforts. Additionally, a centralized metadata store forms a basis for accelerating learning processes (e.g., through warm-starting of hyperparameter search [12]) and is also helpful for automating certain simple error checks (e.g., "did someone accidentally evaluate on their training set?"). Unfortunately, many existing ML frameworks have not been designed to automatically expose their metadata. Therefore, metadata capturing systems [35, 37, 27] in general require that data scientists instrument their code for capturing model metadata, e.g., to annotate hyperparameters of feature transformations or data access operations. Unfortunately, we found that data scientists often forget to add these annotations or are reluctant to spend the extra effort this imposes. In order to ease the adoption of metadata tracking systems, we explore techniques to automatically extract model metadata from ML applications. In our experience, this works well, as long as certain common abstractions are used in ML pipelines, e.g., 'data frames' in pandas or Spark which hold denormalized relational data, and pipelines (in scikit-learn and SparkML) which comprise a way to declare complex feature transformation chains composed of individual operators. For applications built on top of these abstractions, it is relatively simple to automate metadata tracking via reflection and inspection of these components at runtime [27]. In applications not built from declarative components (e.g., systems that rely on shell-scripts to orchestrate model training) on the other hand, it is very hard to automate the metadata extraction process, and a custom component that exposes the metadata has to be built manually. An extension of this problem is that for debugging purposes, often intermediate model outputs need to be inspected [39, 36], in addition to model metadata and lineage.

3.3 Engineering Challenges

Multi-Language Code Bases. A hard-to-overcome practical problem in model management originates from the fact that end-to-end ML applications often comprise of components written in different programming languages. Many popular libraries for ML and scientific computations are written in python, often with native extensions. Examples include scikit-learn [25], numpy and scipy as well as the language frontends of major deep learning systems such as MXNet [9], Tensorflow [2] or PyTorch [23]. For data pre-processing, JVM-based systems such as Apache Spark [38] are often preferred, due to static typing in the language and better support for parallel execution. Such heterogeneous code bases are often hard to keep consistent as automatic refactoring and error checking tools can only inspect either the python or the JVM part of the code, but will not be able to tackle problems across the language barrier. This problem has been identified as "Multiple-Language Smell" [29] in the past already. Furthermore, such code bases are hard to deploy later on, as they require setups with many different components that need to be orchestrated. E.g., a Spark cluster must be spun up for pre-processing, afterwards the data must be moved to a single machine for model training, and the cluster must be torn down afterwards. An orthogonal problem is the efficient and reliable exchange of data between the components of the system written in different languages. Often, this happens via serialization to a common format on disk, which must be kept in sync. In the future, language-independent columnar memory formats such as Apache Arrow for data or ONNX for neural networks [1] are promising developments to address this issue.

Heterogeneous Skill Level of Users. Teams that provide ML technology often have to support users that come from heterogeneous backgrounds. Some user teams have a strong engineering culture, are maintaining large-scale data pipelines and have many job roles dedicated to highly specific tasks. On the other hand, there are product teams with fewer or no members in technical roles. And there are researchers with little experience in handling production systems. ML products should be usable for all customers along this spectrum of technical skills and ensure a smooth transition from one use case to another. The broad range of use cases also requires one to build ML models that work well for large and small datasets at the same time - although these often require different approaches. Some larger datasets can only be tackled with specialized hardware, such as GPUs. But the methods that excel at these tasks with large datasets are often suboptimal when dealing with small data. E.g, if initial training datasets are manually curated by human experts, there is no point in using complex neural network models. In our experience, robust but simple linear models with carefully designed features often work better in these cases. Another dimension that distinguishes the various use cases is how the ML models are deployed afterwards. Some teams might run training and serving using scheduled jobs, but smaller teams often do not want to maintain this type of infrastructure. As a consequence, models have to be designed such that they can be deployed in large-scale scheduled workflows but also quickly tested in experimental prototypes.

Backwards Compatibility of Trained Models. Systems such as Sagemaker [17] or OpenML [35] require backwards compatibility of ML models. A model that was trained last month or last year should still be working today, in particular when trained models might be used in production deployments. Various degrees of backwards compatibility can be satisfied. We may want to impose that the *exact* same result can be obtained, that a *similar* result can be achieved, or in the weakest case that the model can still run. Ensuring those conditions reveals several challenges on model management. Clearly, when a model is deployed and serves predictions, making sure that the exact same result can be retrieved is a strict requirement. To this end, storing all the components defining a model is fundamental in being able to guarantee backwards compatibility and thereby long-term usage. One needs to store at least the data transformations that were required, the implementation of the algorithm, its configuration and finally all its dependencies which can be done by storing model container versions. In the case of training, ensuring that the exact result can be obtained over time may be too strict. This would prevent opportunities to improve models and may be hard to achieve in systems where execution is not fully deterministic (e.g., distributed systems or neural network models when they rely on GPU computation). In

such cases, one may prefer to ensure that training results are similar (or better) over time.

4 Conclusion

We introduced a set of ML use cases and discussed the resulting general challenges with respect to model management. Our focus was on conceptual challenges, challenges related to data-management and engineering. Our experience confirms the need for research on data management challenges for model management and ML systems in general [29, 18]. The problems in this space have their roots in the inherent complexity of ML applications, the lack of a declarative abstraction for end-to-end ML, and the heterogeneity of the resulting code bases. The recent success of deep neural networks in the ML space is often attributed to the gains in prediction accuracy provided by this class of models. We think however that another success factor of neural networks is often overlooked. Neural networks provide an algebraic, declarative abstraction for ML: they are built from computational graphs comprised of tensor operators with clearly defined semantics. Similar to query plans built from relational operators in SQL, this allows for easy combination of complex models and for automatic optimization of the execution on different hardware. With the rise of symbolic neural network APIs it became much simpler for practitioners to experiment with new models, which was one of the main factors contributing to the recent advances we have seen in this field. We strongly feel that the ML space can greatly benefit from adopting proven principles from the data management community such as declarativity and data independence.

Additionally, we observe the rediscovery of many best practices from software engineering and their adaption to managing ML models [40]. Examples are scores for quantifying the production-readiness of an ML system [8] or unit-tests for the implicit data assumptions inherent in ML models [28]. A promising research direction here is to tackle the challenges stemming from the lack of a common declarative abstraction for end-to-end ML, and to design new languages that support relational algebraic and linear primitives at the same time (and can reason about them them holistically [19, 20]). Finally, we would like to point out that questions of model management encounter a growing interest in the academic community. Examples of venues for related research are the 'Systems for Machine Learning' workshop[1] at the NIPS, ICML and SOSP conferences, the 'Data Management for End-to-End Machine Learning' workshop[2] at ACM SIGMOD, and the newly formed SysML conference[3].

References

[1] ONNX open neural network exchange format, 2017.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. *OSDI*, 16:265–283, 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. *KDD*, pages 1387–1395, 2017.

[4] Felix Biessmann, David Salinas, Sebastian Schelter, Philipp Schmidt, and Dustin Lange. "deep" learning for missing value imputationin tables with non-numerical data. *CIKM*, 2018.

[5] Christoph Boden, Tilmann Rabl, and Volker Markl. Distributed machine learning-but at what cost? *NIPS Workshop on Machine Learning Systems*, 2017.

[1] `http://learningsys.org`
[2] `http://deem-workshop.org`
[3] `http://sysml.cc`

[6] 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.

[7] George E P Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[8] Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, and D Sculley. What's your ml test score? a rubric for ml production systems. *NIPS Workshop on Reliable Machine Learning in the Wild*, 2016.

[9] 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. *arXiv preprint arXiv:1512.01274*, 2015.

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

[11] Christos Faloutsos, Jan Gasthaus, Tim Januschowski, and Yuyang Wang. Forecasting big time series: old and new. *PVLDB*, 11(12):2102–2105, 2018.

[12] Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. Initializing bayesian hyperparameter optimization via meta-learning. *AAAI*, pages 1128–1135, 2015.

[13] Valentin Flunkert, David Salinas, and Jan Gasthaus. Deepar: Probabilistic forecasting with autoregressive recurrent networks. *arXiv preprint arXiv:1704.04110*, 2017.

[14] Rob Hyndman, Anne B Koehler, J Keith Ord, and Ralph D Snyder. *Forecasting with exponential smoothing: the state space approach*. Springer Science & Business Media, 2008.

[15] Rob J Hyndman, Yeasmin Khandakar, et al. *Automatic time series for forecasting: the forecast package for R*. Number 6/07. Monash University, Department of Econometrics and Business Statistics, 2007.

[16] Nick Hynes, D Sculley, and Michael Terry. The data linter: Lightweight, automated sanity checking for ml data sets. *NIPS Workshop on Machine Learning Systems*, 2017.

[17] Tim Januschowski, David Arpin, David Salinas, Valentin Flunkert, Jan Gasthaus, Lorenzo Stella, and Paul Vazquez. Now available in amazon sagemaker: Deepar algorithm for more accurate time series forecasting. *https://aws.amazon.com/blogs/machine-learning/now-available-in-amazon-sagemaker-deepar-algorithm-for-more-accurate-time-series-forecasting/*, 2018.

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

[19] Andreas Kunft, Alexander Alexandrov, Asterios Katsifodimos, and Volker Markl. Bridging the gap: towards optimization across linear and relational algebra. *SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, 2016.

[20] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, and Volker Markl. Blockjoin: efficient matrix partitioning through joins. *PVLDB*, 10(13):2061–2072, 2017.

[21] Frank McSherry, Michael Isard, and Derek Gordon Murray. Scalability! but at what cost? *HotOS*, 15:14–14, 2015.

[22] 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.

[23] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[24] John Pavlopoulos, Prodromos Malakasiotis, and Ion Androutsopoulos. Deeper attention to abusive user content moderation. *EMNLP*, pages 1125–1135, 2017.

[25] 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.

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

[27] Sebastian Schelter, Joos-Hendrik Boese, Johannes Kirschnick, Thoralf Klein, and Stephan Seufert. Automatically Tracking Metadata and Provenance of Machine Learning Experiments. *NIPS Workshop on Machine Learning Systems*, 2017.

[28] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. Automating large-scale data quality verification. *PVLDB*, 11(12):1781–1794, 2018.

[29] 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. *NIPS*, pages 2503–2511, 2015.

[30] Matthias Seeger, David Salinas, and Valentin Flunkert. Bayesian intermittent demand forecasting for large inventories. *NIPS*, pages 4646–4654, 2016.

[31] Evan R Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J Franklin, and Benjamin Recht. Keystoneml: Optimizing pipelines for large-scale advanced analytics. *ICDE*, pages 535–546, 2017.

[32] Michael Stonebraker and Ihab F Ilyas. Data integration: The current status and the way forward. *Data Engineering Bulletin*, 41(2):3–9, 2018.

[33] Sean J Taylor and Benjamin Letham. Forecasting at scale. *The American Statistician*, 2017.

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

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

[36] Manasi Vartak, Joana M F da Trindade, Samuel Madden, and Matei Zaharia. Mistique: A system to store and query model intermediates for model diagnosis. *SIGMOD*, pages 1285–1300, 2018.

[37] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. Model db: a system for machine learning model management. *SIGMOD Workshop on Human-In-the-Loop Data Analytics*, page 14, 2016.

[38] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *NSDI*, 2012.

[39] Zhao Zhang, Evan R. Sparks, and Michael J. Franklin. Diagnosing machine learning pipelines with fine-grained lineage. *HPDC*, pages 143–153, 2017.

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

MODELDB: Opportunities and Challenges in Managing Machine Learning Models

Manasi Vartak
MIT CSAIL
mvartak@csail.mit.edu

Samuel Madden
MIT CSAIL
madden@csail.mit.edu

Abstract

Machine learning applications have become ubiquitous in a variety of domains. Powering each of these ML applications are one or more machine learning models that are used to make key decisions or compute key quantities. The life-cycle of an ML model starts with data processing, going on to feature engineering, model experimentation, deployment, and maintenance. We call the process of tracking a model across all phases of its life-cycle as **model management***. In this paper, we discuss the current need for model management and describe* MODELDB*, the first open-source model management system developed at MIT. We also discuss the changing landscape and growing challenges and opportunities in managing models.*

1 Introduction

Machine learning (ML) has become ubiquitous in a variety of applications including voice assistants, self-driving cars, and recommendation systems. Each ML-based application employs one or more machine learning models that are used to make key decisions or compute key quantities such as recognizing spoken words, detecting a pedestrian on the road, and identifying the best products for customers. Models are to ML-based applications what databases are to stateful web-applications; they are crucial for the correct functioning of these applications. Consequently, just as database management systems (or DBMSs) are used to manage state in applications, we find the need for systems to manage models in ML applications, i.e., *model management systems*.

To understand the requirements of a model management system, we begin with a brief overview of the life-cycle of a machine learning model. We divide the ML life-cycle into five phases, namely: (1) *Data Preparation* - Obtaining the training and test data to develop a model; (2) *Feature Engineering* - Identifying or creating the appropriate descriptors from the input data (i.e., features) to be used by the model; (3) *Model Training and Experimentation* - Experimenting with different models on the training and test data and choosing the best; (4) *Deployment* - Deploying the chosen model in a live system; and (5) *Maintenance* - Monitoring the live model performance, updating the model as needed, and eventually retiring the model. While we describe these phases as occurring in a linear sequence, the empirical nature of building an ML model causes multiple phases to be revisited frequently. We define a model management system as one that follows a model throughout the five phases of its life-cycle and captures relevant metadata at each step to enable model tracking, reproducibility, collaboration, and governance.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

For instance, since the key requirement during the Experimentation phase is to enable the data scientists to choose the best model, metadata captured in this phase includes items such as performance metrics for the model, hyperparameter values used during training, etc. In contrast, for a model in the Deployment phase, metadata might include the version of the model, where it is deployed, and how to query it. Although one can imagine a model management system that directly stores models and supports prediction serving operations, the heterogeneity in models, as well as their hardware and software requirements make such a solution sub-optimal. Therefore, we take the view that model management systems are best suited to store *metadata* about models throughout their life-cycle. Thus, **we define a model management system as a system that tracks metadata about models through the five phases of their life-cycle**.

Given the rapid proliferation of machine learning applications, systems have been proposed in academia as well as industry to address different aspects of the model management problem. In this paper, we focus on MODELDB, the first open-source system we developed at MIT for model management. Other academic systems that seek to address similar problems include the ModelHub system [18] (to explore deep learning architectures and efficiently store network weights), ProvDB [19] (to manage metadata collected via collaborative data science), Ground [14] (to provide a common framework for tracking data origin and use via generic abstractions), and the work on model selection management systems by Kumar et. al. [15].

An example of a commercial model management system is the SAS Model Manager which tracks models built and deployed within the SAS platform [24]. Today, most proprietary ML platforms such as the Michelangelo platform at Uber [11] and FBLearner at Facebook [6] include a model management or model repository component. Similarly, vast majority of data science teams end up re-building a model management system to suit their needs. In [25], Sculley et. al. elegantly present the challenges with building and productionizing ML at Google and highlight the need to manage "pipeline jungles" and model configurations.

Model management systems are also closely related to workflow and experiment management systems such as Kepler [16], Taverna workbench [29], Galaxy [27], VisTrails [1, 3, 2] as well as recently introduced workflow engines tailored for data processing such as Apache Airflow [11] and Luigi [26]. While workflow systems can address some of the model management needs during the first three phases of the model life-cycle, these systems require extensions for ML-specific abstractions.

The rest of the paper is organized as follows. In Section 2, we describe the motivation behind model management systems; in Section 3, we describe the challenges faced in building model management systems; and in Section 4, we describe the MODELDB system developed at MIT. We conclude the paper in Section 5 with a discussion of how we see model management evolving in the future.

2 Why Model Management?

To understand the need for model management, we studied modeling workflows across companies and research labs ranging from large internet companies to small start-ups building ML-based applications. Across the different phases of the model life-cycle, the need for model management was apparent in three key areas: managing modeling experiments, enabling reproducibility, supporting sharing of models, and model governance.

The first and primary area where the need for model management is evident is during the Experimentation phase of the model life-cycle. The empirical nature of ML model development means that data scientists and ML developers experiment with hundreds of models before identifying one that meets some acceptance criteria. This is particularly true when data scientists perform hyperparameter optimization on models. Data about previously built models is necessary to inform the next set of models and experiments, and to identify the best model produced so far. Consequently, tracking these experiments is of paramount importance. The absence of an experiment tracking system leads to models and experiments being lost and valuable time and resources being spent in reproducing model results. For example, one ML developer at a large tech company related how she had spent over a week merely re-running a modeling experiment another employee had previously conducted

since the experimental setup and results were not recorded.

The second area that requires model management is enabling model reproducibility. A key use case for reproducibility deals with the situation where the new version of a deployed model performs poorly compared to the previous one and must be reverted to the older version. If the previous model version is not available, re-creating it requires precise information about what data was used, how it was processed, libraries and versions used, and details of how the model was trained (including random seeds). A similar need becomes evident when a model has to be updated with new data or when a discrepancy is detected in an offline and live model.

The third area that requires model management is supporting model sharing. As data science teams increase in size, multiple data scientists want to collaborate on the same model or build on top of each others' work. In such cases, the lack of a centralized repository of models along with the right access control patterns hampers sharing of information about models and experiments. Finally, for software developers who wish to use existing ML models, the absence of a central repository makes it challenging to discover and integrate models into products and business processes.

The last area where we find a growing need for model management relates to model governance. For many regulated industries, (e.g., banking and healthcare), government regulations require that any model used to make automated decisions be documented and available for audit. Furthermore, government legislation now being implemented (e.g., the GDPR regulations in the European Union [21]) requires that companies be able to explain any decisions made without human intervention. These trends highlight the need for a centralized system that manages and explains models.

3 Challenges in Model Management

As defined above, model management covers the entire life-cycle of models starting with Data Processing, Feature Engineering, Experimentation, Deployment, and Maintenance. The hallmark of machine learning model development is the heterogenity in environments and frameworks used at every phase of the ML life-cycle. As a result, the primary challenge for model management is to consistently capture metadata across many environments at each phase of the modeling life-cycle. We illustrate this challenge with a few examples.

During the Data Processing or Feature Engineering phases, our goal is to track the transformations applied to data so that they may be accurately reproduced later. Since these tasks may be done in very different languages and data processing environments (e.g., Spark, Teradata, HBase), we must ensure that data transformations can be accurately recorded across a wide range of data processing environments. Moreover, even within a single environment, we must ensure high coverage for all the ways in which a specific data transformation may be applied — a significant challenge unless this functionality has been built into the system from the ground-up.

Similarly, during the Experimentation phase, there is a large diversity in machine learning frameworks and libraries in different programming languages that a data scientist may use (e.g., scikit-learn, Tensorflow, PyTorch in Python; ML libraries in R; H2O framework in Java). Each library has a unique way of defining models (e.g., graphs in Tensorflow vs. plain objects in scikit-learn) and their associated attributes. Consequently, one representation is often inadequate to capture models built across all frameworks. This diversity in machine learning frameworks during Experimentation also translates to diversity in deployment methods during the Deployment phase. For example, while TF-serving is a popular means to serve Tensorflow models, a Flask-based deployment method is most popular for scikit-learn models.

Lastly, once a model is deployed, i.e. during the Maintenance phase, while some properties are common to all models (e.g., latency of predictions, number of requests), many properties are application dependent (e.g., accuracy of model for new users on an e-commerce site) and might require input from disparate systems outside of the ML environment (e.g., prediction storage or logging systems). Thus we see that addressing the heterogeneity at every phase of the modeling life-cycle is the key challenge for model management.

While the above challenges arise from the diversity of environments for ML, two overarching requirements

```
# version 61
#   Drop fireplacecnt and fireplaceflag, following Jayaraman:
#     https://www.kaggle.com/valadi/xgb-w-o-outliers-lgb-with-outliers-combo-tune5

# version 60
#   Try BASELINE_PRED=0.0115, since that's the actual baseline from
#     https://www.kaggle.com/aharless/oleg-s-original-better-baseline

# version 59
#   Looks like 0.0056 is the optimum BASELINE_WEIGHT

# versions 57, 58
#   Playing with BASELINE_WEIGHT parameter:
#     3 values will determine quadratic approximation of optimum

...

# version 49
#   My latest quadratic approximation is concave, so I'm just taking
#     a shot in the dark with lgb_weight=.3

# version 45
#   Increase lgb_weight to 0.25 based on new quadratic approximation.
#   Based on scores for versions 41, 43, and 44, the optimum is 0.261
#     if I've done the calculations right.
#   I'm being conservative and only going 2/3 of the way there.
#   (FWIW my best guess is that even this will get a worse score,
#    but you gotta pay some attention to the math.)

# version 44
#   Increase lgb_weight to 0.23, per Nikunj's suggestion, even though
#     my quadratic approximation said I was already at the optimum
```

Figure 1: Model Versioning comments by Kaggle competitor

emerge from the data scientists' perspective as well. First, data scientists want a model management solution that minimizes developer intervention and requires minimal changes to the current developer workflow. For example, many data scientists are resistant to choosing a new ML environment or significantly changing their modeling workflow to account for model management. And second, data scientists want a vendor and framework-neutral model management system so that they are not tied into one particular provider or framework and can choose the best solutions for the particular modeling task.

To summarize, in order to build a model management system, we must address the problem of supporting a variety of ML environments, imposing common abstractions across diverse environments, and capturing sufficient metadata while requiring minimal changes from data scientists. In the next section, we describe MODELDB, an open-source model management system developed at MIT that takes the first steps in tackling the challenges identified above and focuses particularly on the Experimentation phase of the model life-cycle.

4 MODELDB

Building an ML model for real-world applications is an iterative process. Data scientists and ML developers experiment with tens to hundreds of models before identifying one that meets some acceptance criteria on model performance. For example, top competitors in the Kaggle competition to predict Zillow Home prices [5] made more than 250 submissions (and therefore built at least as many models), while those in the Toxic Comment classification competition [4] made over 400 submissions. As an example of actual experimentation performed during model building, Figure 1 reproduces code comments by an expert Kaggle competitor (ranked in top 500) written in order to track models built for the Zillow Price Prediction Challenge. As we can see from this listing,

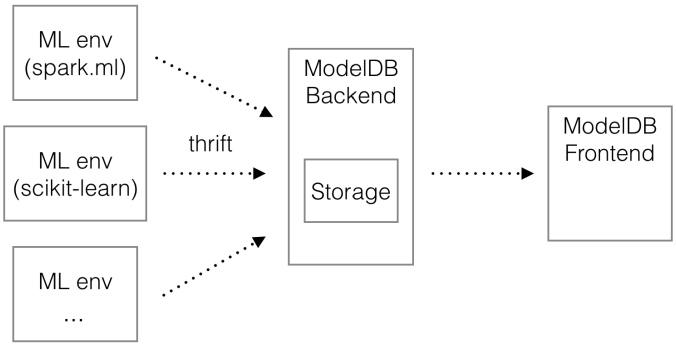


Figure 2: ModelDB Architecture

a data scientist typically tests a large number of model versions before identifying the best one. Moreover, although data scientists (and data science teams) build many tens to hundreds of models when developing an ML application, they currently have no way to keep track of all the models they have built. Consequently, insights are lost, models cannot be reproduced or shared, and model governance becomes challenging.

To address these problems, we developed a system at MIT called MODELDB [28]. MODELDB is the first open-source machine learning model management system and currently focuses on tracking models during the Experimentation phase. MODELDB automatically tracks models as they are built, records provenance information for each step in the pipeline used to generate the model, stores this data in a standard format, and makes it available for querying via an API and a visual interface.

4.1 Architecture

Figure 2 shows the high-level architecture of our system. MODELDB consists of three key components: client libraries for different machine learning environments, a backend that stores model data, and a web-based visualization interface. Client libraries are responsible for automatically extracting models and pipelines from code and passing them to the MODELDB backend. The MODELDB backend exposes a thrift[1] interface to allow clients in different languages to communicate with the MODELDB backend. MODELDB client libraries are currently available for scikit-learn and spark.ml, along with a Light Python API that can be used in any Python-based machine learning environment. This means that ML developers can continue to build models and perform experimentation in these environments while the native libraries passively capture model metadata. The backend can use a variety of storage systems to store the model metadata. The third component of MODELDB, the visual interface, provides an easy-to-navigate layer on top of the backend storage system that permits visual exploration and analyses of model metadata.

4.2 Client Libraries

Many existing workflow management programs (e.g. VisTrails [3]) require that the user create a workflow in advance, usually by means of a GUI. However, the data scientists we interviewed overwhelmingly concurred that GUIs restricted their flexibility in defining modeling pipelines and iterating over them. Moreover, we found that data scientists were resistant to significant changes in their modeling workflows. Therefore, our primary design constraint while creating the MODELDB client libraries was to minimize any changes the data scientist would need to make both to code and the existing modeling process. To meet this constraint, we chose to make

[1] https://thrift.apache.org/

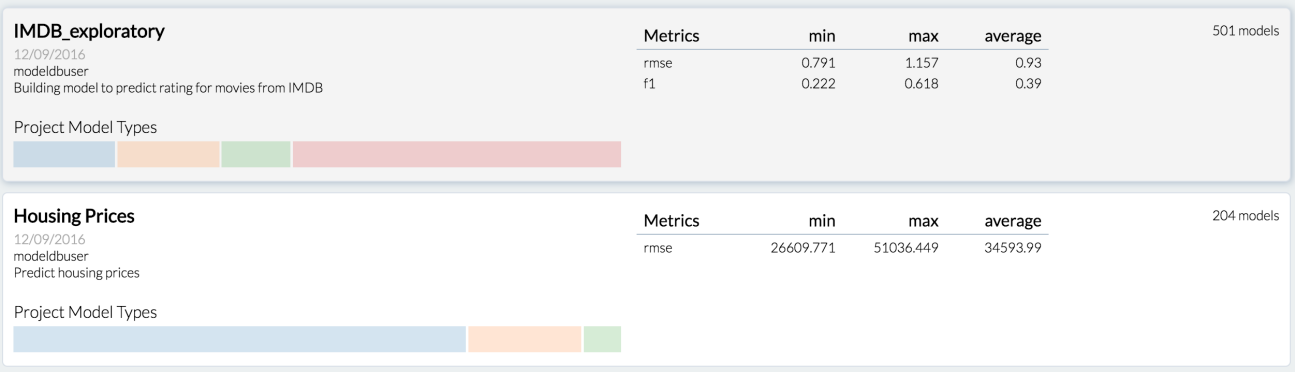


Figure 3: Projects Summary View

model logging accessible directly through code (as opposed to a GUI) and to build logging libraries for different ML environment. The spark.ml and scikit-learn libraries are architected such that data scientists can use these environments for analysis exactly as they normally would and the library transparently and automatically logs the relevant data to the backend.

4.3 Frontend

MODELDB captures a large amount of metadata about models. To support easy access to this data, MODELDB provides a visual interface. We provide three key views for exploring the data stored in MODELDB. A user starts with the projects summary page (Fig. 3) that provides a high level overview of all the projects in the system. The user can then click on a particular project to see the models for that project. We present models via two key views. The first view presents model summaries through two visualizations (Fig. 4): a summary visualization showing the evolution of model metrics over time as well as a custom visualization builder for model meta-analyses. The second is a tabular view of models in a particular project (Fig. 5) along with interactions such as filtering, sorting, grouping, and search. From any of the above interfaces, the ML developer can drill-down into a single model to visualize the model pipeline that was automatically inferred by the client library.

4.4 MODELDB Adoption and Future Work

We officially released MODELDB as an open-source model management system in Feb. 2017 and since then there has been a large amount of interest and adoption of MODELDB. Over the last year, our GitHub repository [12] has garnered $>$ 500 stars, has been cloned over a thousand times, and has been forked $>$100 times. MODELDB has been tested at multiple small and large companies and has been deployed in financial institutions. MODELDB has also served as inspiration for other model management systems such as [20]. The rapid adoption of MODELDB indicates a strong need for model management, and our current work focuses on expanding MODELDB to support other phases of the ML life-cycle.

5 Evolution of Model Management Systems

As machine learning models proliferate into every business process and product, we posit that the task of managing the life-cycle of models will become as key as managing the life-cycle of code. Just as version control systems such as SVN and Git made source code development and collaboration robust and rapid, we envision that model management systems will serve a similar purpose in the future. We imagine model management systems to become the system of record for all models and model-related information. Current model management

Figure 4: Model Summary Visualization and Custom Visualization Builder

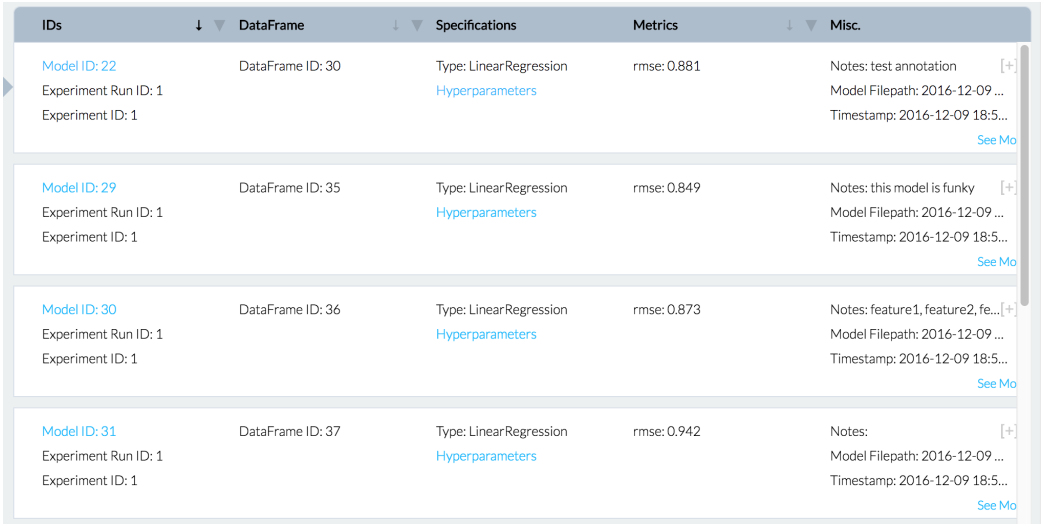


Figure 5: Tabular Models View

systems have focused on tracking models in a limited fashion; consequently, to support diverse ML applications, we expect model management systems to evolve in the following directions.

Model Data Pipelines. For many machine learning applications, the ultimate performance of the machine learning model depends on the features or data attributes used by the model. As a result, when a model is to be reproduced, it requires accurate records of the data and transformations used to produce features. We expect model management systems to evolve to accurately record data versions (train and test) as well data transformations that are used to generate features. This metadata may come from a separate data processing or workflow system as opposed to being generated by the model management system, however, the model management system would track all the metadata required to create the model end-to-end.

Model Interoperability. One of the key challenges in model management described before is the diversity in ML models and frameworks. While some frameworks support rapid development, others might be more suitable for deployment in production settings. Given the rapid proliferation of ML frameworks, we expect model management systems to support, if not provide, a level of interoperability between different ML environments and frameworks (e.g., PMML [9] and ONNX [14] provide a good start). This will enable data scientists to pick and choose the best framework for each phase of the model life-cycle without being tied to a single framework.

Model Testing. As more decisions and business logic get delegated to machine learning models, the importance of testing models (similar to code testing) will increase and will become a key part of managing the model life-cycle. For instance, defining unit tests for models and their input data will become commonplace. Similarly, we expect integration tests with models to become more prevalent as predictions from one model get used as input for other models. Finally, we note that as adversarial attacks on models increase, testing of models and edge cases will become key, requiring the development of new techniques to prevent adversarial attacks (e.g., [10, 9]).

Model Monitoring. While model testing takes place before a model is deployed, model monitoring takes place once the model is deployed in a live system. Model monitoring today is largely limited to monitoring system-level metrics such as the number of prediction requests, latency, and compute usage. However, we are already seeing the need for data-level monitoring of models (e.g., as noted in [25]) to ensure that the offline and live data fed to a model is similar. We expect model management systems to encompass model monitoring modules to ensure continued model health, triggering alerts and actions as appropriate.

Model Interpretability and Fairness. As models are used for automated decision making in regulated industries and increasingly used by non-technical users, explaining the results of models will become a key aspect of the management of deployed models (as evidenced by the rich research on interpretability [17, 6, 23]). We view a model management system as the system of record for all models and, therefore, the logical gateway for model interpretability and understanding. In the future, we therefore expect model management systems to expose interpretability functionality for every recorded model.

6 Conclusion

ML models are becoming ubiquitous is a variety of domains. This proliferation of ML brings to the forefront the need for systems that are responsible for managing models across their entire life-cycle starting with data preparation to model retirement. In this paper, we discussed the motivation for model management systems as well as challenges associated with consistently tracking models throughout their life-cycle. We described MODELDB, the first open-source model management system developed at MIT. Finally, we described the evolving opportunities in model management.

References

[1] Louis Bavoil, Steven P Callahan, Patricia J Crossno, Juliana Freire, Carlos E Scheidegger, Cláudio T Silva, and Huy T Vo. Vistrails: Enabling interactive multiple-view visualizations. In *Visualization, 2005. VIS 05. IEEE*, pages 135–142. IEEE, 2005.

[2] Steven P Callahan, Juliana Freire, Emanuele Santos, Carlos E Scheidegger, Claudio T Silva, and Huy T Vo. Managing the evolution of dataflows with vistrails. In *Data Engineering Workshops, 2006. Proceedings. 22nd International Conference on*, pages 71–71. IEEE, 2006.

[3] Steven P Callahan, Juliana Freire, Emanuele Santos, Carlos E Scheidegger, Cláudio T Silva, and Huy T Vo. Vistrails: visualization meets data management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 745–747. ACM, 2006.

[4] Kaggle competition. Toxic comment classification challenge. `https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge`.

[5] Kaggle Competition. Zillow prize: Zillow's home value prediction (zestimate). `https://www.kaggle.com/c/zillow-prize-1`.

[6] Been Doshi-Velez, Finale; Kim. Towards a rigorous science of interpretable machine learning. In *eprint arXiv:1702.08608*, 2017.

[7] Facebook Engineering. Introducing fblearner flow: Facebook's ai backbone. `https://code.facebook.com/posts/1072626246134461/introducing-fblearner-flow-facebook-s-ai-backbone/`.

[8] Uber Engineering. Meet michelangelo: Uber's machine learning platform. `https://eng.uber.com/michelangelo/`.

[9] Reuben Feinman, Ryan R Curtin, Saurabh Shintre, and Andrew B Gardner. Detecting adversarial samples from artifacts. *arXiv preprint*, 2017.

[10] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *ICLR*, 2015.

[11] Apache Software Foundation. Airflow.

[12] MIT DB Group. Modeldb. `https://github.com/mitdbg/modeldb`, 2017.

[13] Alex Guazzelli, Wen-Ching Lin, and Tridivesh Jena. *PMML in Action: Unleashing the Power of Open Standards for Data Mining and Predictive Analytics*. CreateSpace, Paramount, CA, 2010.

[14] 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.

[15] Arun Kumar, Robert McCann, Jeffrey Naughton, and Jignesh M. Patel. Model selection management systems: The next frontier of advanced analytics. *SIGMOD Rec.*, 44(4):17–22, May 2016.

[16] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[17] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems 30*, pages 4768–4777. Curran Associates, Inc., 2017.

[18] H. Miao, A. Li, L. S. Davis, and A. Deshpande. Modelhub: Deep learning lifecycle management. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1393–1394, April 2017.

[19] Hui Miao, Amit Chavan, and Amol Deshpande. Provdb: Lifecycle management of collaborative analysis workflows. 2017.

[20] MLFlow. Mlflow. `https://github.com/mlflow/mlflow`.

[21] Official Journal of the European Union. General data protection regulation. `https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=celex%3A32016R0679`.

[22] ONNX. Onnx: Open neural network exchange. https://github.com/onnx/onnx.

[23] Maithra Raghu, Justin Gilmer, Jason Yosinski, and Jascha Sohl-Dickstein. Svcca: Singular vector canonical correlation analysis for deep learning dynamics and interpretability. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 6078–6087. Curran Associates, Inc., 2017.

[24] SAS. Sas model manager. https://www.sas.com/en_us/software/model-manager.html.

[25] D Sculley, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. Machine learning: The high-interest credit card of technical debt.

[26] Spotify. Luigi. https://github.com/spotify/luigi.

[27] The Galaxy Team and Community. The galaxy project. https://galaxyproject.org/.

[28] 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 *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, HILDA '16, pages 14:1–14:3, New York, NY, USA, 2016. ACM.

[29] Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic, Paul Fisher, et al. The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic acids research*, 41(W1):W557–W561, 2013.

ProvDB: Provenance-enabled Lifecycle Management of Collaborative Data Analysis Workflows

Hui Miao, Amol Deshpande
University of Maryland, College Park, MD, USA
{hui, amol}@cs.umd.edu

Abstract

Collaborative data science activities are becoming pervasive in a variety of communities, and are often conducted in teams, with people of different expertise performing back-and-forth modeling and analysis on time-evolving datasets. Current data science systems mainly focus on specific steps in the process such as training machine learning models, scaling to large data volumes, or serving the data or the models, while the issues of end-to-end data science lifecycle management are largely ignored. Such issues include, for example, tracking provenance and derivation history of models, identifying data processing pipelines and keeping track of their evolution, analyzing unexpected behaviors and monitoring the project health, and providing the ability to reason about specific analysis results. In this article, we present an overview of a unified provenance and metadata management system, called PROVDB*, that we have been building to support lifecycle management of complex collaborative data science workflows.* PROVDB *captures a large amount of fine-grained information about the analysis processes and* versioned *data artifacts in a semi-passive manner using a flexible and extensible ingestion mechanism; provides novel querying and analysis capabilities for simplifying bookkeeping and debugging tasks for data analysts; and enables a rich new set of capabilities like identifying flaws in the data science process.*

1 Introduction

Over the last two decades, we have seen a tremendous increase in collaborative data science activities, where teams of data engineers, data scientists, and domain experts work together to analyze and extract value from increasingly large volumes of data. This has led to much work in industry and academia on developing a wide range of tools and techniques, including big data platforms, data preparation and wrangling tools, new and sophisticated machine learning techniques, and new environments like Jupyter Notebooks targeted at data scientists. However, the widespread use of data science has also exposed the shortcomings of the overall ecosystem, and has brought to the forefront a number of critical concerns about how data is collected and processed, and how insights or results are obtained from those. There are numerous examples of biased and unfair decisions made by widely used data science pipelines, that are often traced back to flaws in the training data, or mistakes or assumptions made somewhere in the process of constructing those pipelines. Data analysis processes are relatively easy to "hack" today, increasingly resulting in a general lack of trust in published results ("fake news") and many incidents of unreproducible and/or retracted results.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Addressing these concerns requires a holistic approach to conducting, auditing, and continuously monitoring collaborative data science projects. This however is very challenging, because of the fundamentally ad hoc nature of collaborative data science. It typically features highly unstructured and noisy datasets that need to be cleaned and wrangled; amalgamation of different platforms, tools, and techniques; significant back-and-forth among the members of a team; and trial-and-error to identify the right analysis tools, algorithms, machine learning models, and parameters. The data science pipelines are often spread across a collection of analysis scripts, possibly across different systems. Metadata or provenance information about how datasets were generated, including the programs or scripts used for generating them and/or values of any crucial parameters, is often lost. As most datasets and analysis scripts evolve over time, there is also a need to keep track of their *versions* over time; using version control systems (VCS) like *git* can help to some extent, but those don't provide sufficiently rich introspection capabilities.

Lacking systematic platforms to support collaborative data science, we see users manually track and act upon such information. For example, users often manually keep track of which derived datasets need to be updated when a source dataset changes; they use spreadsheets to list which parameter combinations have been tried out during the development of a machine learning model; and so on. This is however tedious and error-prone, requires significant discipline from the users, and typically lacks sufficient coverage.

In this article, we present an overview of our PROVDB system, for unified management of all kinds of metadata about collaborative data science workflows that gets generated during the analysis processes. PROVDB is being developed under the umbrella of the **DataHub** project [13], where our overarching goal is to understand and investigate the major pain points for data engineers and scientists today, and develop tools and techniques for managing a large number of datasets, their versions over time, and derived data products and for supporting the end-to-end data science lifecycle. PROVDB is designed to be a standalone system, with complementary functionality to dataset versioning tools developed in [14, 28, 33, 17, 15]. Specifically, PROVDB manages and enables querying information about: **(a)** version lineages of data, scripts, and results (collectively called *artifacts*), **(b)** data provenance among artifacts which may or may not be structured, and **(c)** workflow metadata on derivations and dependencies among artifact snapshots, and **(d)** other context metadata that may be collected (e.g., summary logs). By making it easy to mine and query this information in a unified fashion, PROVDB enables a rich set of functionality to simplify the lives of data scientists, to make it easier to identify and eliminate errors, and to decrease the time to obtain actionable insights.

To accommodate the wide variety of expected use cases, PROVDB adopts a "schema-later" approach, where a small base schema is fixed, but users can add arbitrary semistructured information (in JSON) for recording additional metadata. The specific data model we use generalizes the standard W3C PROV data model, extended to allow flexibly capturing a variety of different types of information including versioning and provenance information, parameters used during experiments or modeling, statistics gathered to make decision, analysis scripts, and notes or tags. We map this logical data model to a *property graph data model*, and use the Neo4j graph database to store the information.

PROVDB features a suite of extensible *provenance ingestion* mechanisms. The currently supported ingestors are targeted at users who conduct the data analysis using either `UNIX Shell` or `Jupyter Notebooks`. In the former case, user-run `shell` commands to manipulate files or datasets are intercepted, and the before- and after-state of the artifacts is analyzed to generate rich metadata. Several such ingestors are already supported and new ingestors can be easily registered for specific commands. For example, such an ingestion program is used to analyze log files generated by Caffe (a deep learning framework) and to generate metadata about accuracy and loss metrics (for learned models) in a fine-grained manner. PROVDB also features APIs to make it easy to add provenance or context information from other sources.

Finally, PROVDB features many different mechanisms for querying and analyzing the captured data. In addition to being able to use the querying features of `Neo4j` (including the Cypher and Gremlin query languages), PROVDB also features two novel graph query operators, *graph segmentation* and *graph summarization*, that are geared towards the unique characteristics of provenance information.

2 ProvDB Architecture

PROVDB is a stand-alone system, designed to be used in conjunction with a dataset version control system (DVCS) like `git` or `DataHub` [13, 18, 17, 14] (Figure 1). The DVCS will handle the actual version management tasks, and the distributed and decentralized management of individual *repositories*. A repository consists of a set of *versions*, each of which is comprised of a set *artifacts* (scripts, datasets, derived results, etc.). A version, identified by an ID, is immutable and any update to a version conceptually results in a new version with a different version ID (physical data structures are typically not immutable and the underlying DVCS may use various strategies for compact storage [14]). New versions can also be created through the application of transformation programs to one or more existing versions. We assume the DVCS supports standard version control operations including CHECKOUT a version, COMMIT a new version, create a BRANCH, MERGE versions, etc.

Most of the PROVDB modules are agnostic to the DVCS being used, as long as appropriate pipelines to transfer versioning metadata and context data are set up. The current implementation is based on **`git`**, in which case, the repository contents are available as files for the users to operate upon; the users can run whichever analysis tools they want on those after checking them out, including distributed toolkits like Hadoop or Spark.

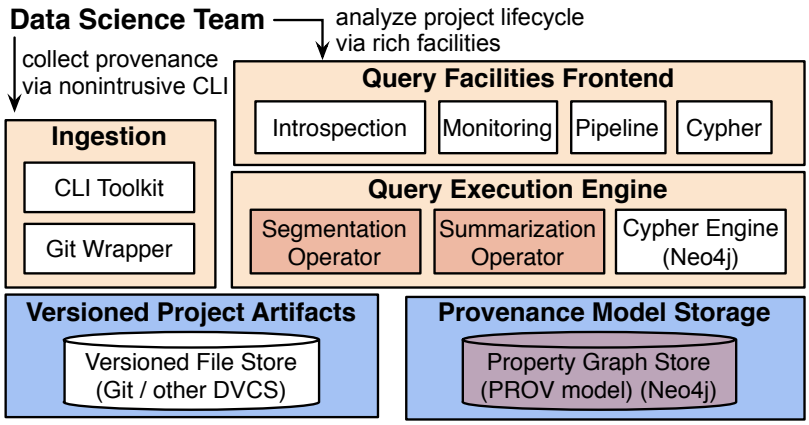


Figure 1: Architecture Overview of PROVDB

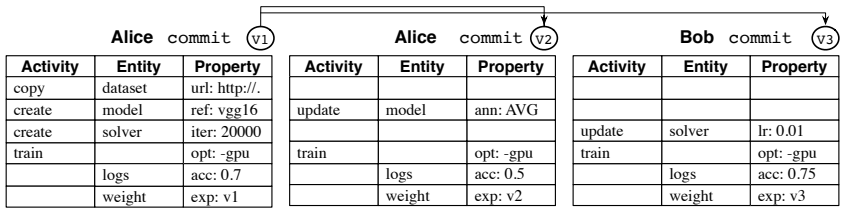
Broadly speaking, the data maintained across the system can be categorized into:

(a) raw data that the users can directly access and analyze including the datasets, analysis scripts, and any derived artifacts such as trained models, and

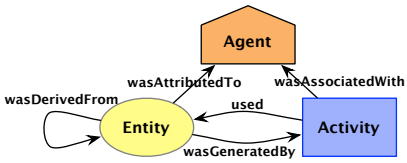
(b) metadata or provenance information transparently maintained by the system, and used for answering queries over the versioning or provenance information.

Fine-grained record-level provenance information may or may not be directly accessible to the users depending on the ingest mechanism used. Note that, the split design that we have chosen to pursue requires duplication of some information in the DVCS and PROVDB. We believe this is a small price to pay for the benefits of having a standalone provenance management system.

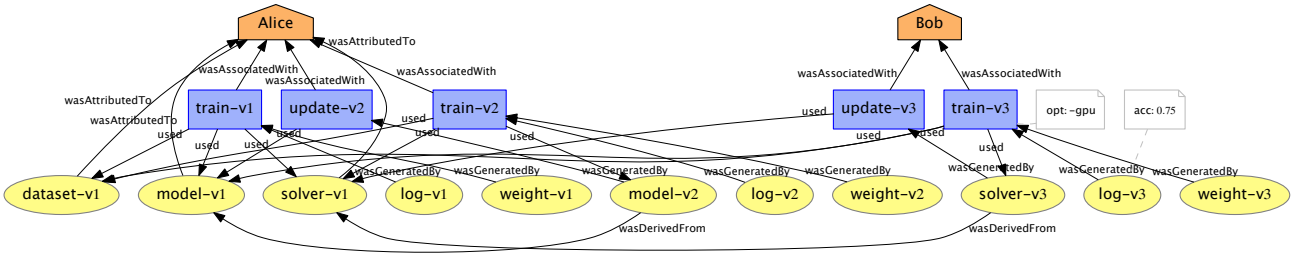
PROVDB **Ingestion module** is a thin layer on top of the DVCS (in our case, `git`) that is used to capture the provenance and metadata information. This layer needs to support a variety of functionality to make it easy to collect a large amount of metadata and provenance information, with minimal overhead to the user (Sec. 4). The PROVDB instance itself is a separate process, and currently uses the Neo4j graph database for **provenance model storage**; we chose Neo4j because of its support for the flexible property graph data model, and graph querying functionality out-of-the-box (Sec. 3). The data stored inside PROVDB can be queried using Cypher through the Neo4j frontend; PROVDB **query execution engine** also supports two novel types of *segmentation* and *summarization* queries that we breifly describe in Section 5. Richer queries can be supported on top of these low-level querying constructs; PROVDB currently supports a simple form of continuous monitoring query, and also has a visual frontend to support a variety of provenance queries (Section 5).



(a) An Example for a Data Science Project Lifecycle & Associated Provenance



(b) Illustration of the W3C PROV Data Model



(c) Provenance Graph for the Lifecycle Example (Some edges and properties are not shown due to space constraints)

Figure 2: Illustration of Provenance Data Model and Query Operators in Data Science Lifecycles

3 Provenance Data Model and Provenance Graphs

The ingested provenance over the lifecycle of a data science project naturally forms a directed acyclic graph[1], combining heterogeneous information including a version graph representing the artifact changes, a workflow graph reflecting the derivations of those artifact versions, and a conceptual model graph showing the involvement of problem solving methods in the project [21, 30]. To represent the provenance graph and keep our discussion general to other provenance systems, after originally using a custom model, we switched to using the W3C PROV data model [35], which is a standard interchange model for different provenance systems. We use the core set of PROV data model shown in Fig. 2(b). There are three types of vertices ($\mathbb{V}$) in the provenance graph in our context:

a) Entities ($\mathcal{E}$) are the project artifacts (e.g., files, datasets, scripts) which the users work on and talk about;

b) Activities ($\mathcal{A}$) are the system or user actions (e.g., `train`, `git commit`, `cron` jobs) which act upon or with entities over a period of time, $[t_i, t_j)$;

c) Agents ($\mathcal{U}$) are the parties who are responsible for some activity (e.g., a team member, a system component).

Among the vertices, there are five types of directed edges ($\mathbb{E}$):

i) An activity started at time t_i often uses some entities ('used', $U \subseteq \mathcal{A} \times \mathcal{E}$);

ii) then some entities were generated by the same activity at time t_j ($t_j \geq t_i$) ('wasGeneratedBy', $G \subseteq \mathcal{E} \times \mathcal{A}$);

iii) An activity is associated with some agent during its period of execution ('wasAssociatedWith', $S \subseteq \mathcal{A} \times \mathcal{U}$);

iv) Some entity's presence can be attributed to some agent ('wasAttributedTo', $A \subseteq \mathcal{E} \times \mathcal{U}$); and

v) An entity was derived from another entity ('wasDerivedFrom', $D \subseteq \mathcal{E} \times \mathcal{E}$), such as versions of the same artifact (e.g., different model versions in v_1 and v_2 in Fig. 2(a)).

[1] We use versioning to avoid cyclic self-derivations of the same entity and overwritten entity generations by some activity.

In the provenance graph, both vertices and edges have a label to encode their vertex type in $\{\mathcal{E}, \mathcal{A}, \mathcal{U}\}$ or edge type in $\{U, G, S, A, D\}$. All other ingested provenance records are modeled as properties, that are ingested by a set of configured project ingestors during activity executions and represented as key-value pairs.

PROV standard defines various serializations of the concept model (e.g., RDF, XML, JSON) [37]. In our system, we use a physical property graph data model to store it, as it is more natural for the users to think of the artifacts as nodes when writing queries using Cypher or Gremlin. It is also more compact than RDF graph for the large amount of provenance records, which are treated as literal nodes.

Definition 1 (Provenance Graph): Provenance in a data science project is represented as a directed acyclic graph, $\mathcal{G}(\mathbb{V}, \mathbb{E}, \lambda_v, \lambda_e, \sigma, \omega)$, where vertices have three types, $\mathbb{V}= \mathcal{E} \cup \mathcal{A} \cup \mathcal{U}$, and edges have five types, $\mathbb{E}= U \cup G \cup S \cup A \cup D$. Label functions, λ_v: $\mathbb{V} \mapsto \{\mathcal{E}, \mathcal{A}, \mathcal{U}\}$, and λ_e: $\mathbb{E} \mapsto \{U, G, S, A, D\}$ are total functions associating each vertex and each edge to its type. In a project, we refer to the set of property types as $\mathcal{P}$ and their values as $\mathcal{O}$, then vertex and edge properties, σ: $\mathbb{V} \times \mathcal{P} \mapsto \mathcal{O}$ and ω: $\mathbb{E} \times \mathcal{P} \mapsto \mathcal{O}$, are partial functions from vertex/edge and property type to some value.

Example 1: In Fig. 2(a), Alice and Bob work together on a classification task to predict face IDs given an image. Alice starts the project and creates a neural network by modifying a popular model. She downloads the dataset and edits the model definitions and solver hyperparameters, then invokes the training program with specific command options. After training the first model, she examines the accuracy in the log file, annotates the weight files, then commits a version using `git` via ProvDB command-line interface (CLI, Fig. 1). As the accuracy of the first model is not ideal, she changes the network by editing the model definition, trains it again and derives new log files and weight parameters. However the accuracy drops, and she turns to Bob for help. Bob examines what she did, trains a new model following best practices by editing the learning rate in solver configuration in version v_1.

Behind the scenes, ProvDB tracks users' activities, ingests provenance, and manages versioned artifacts (e.g., datasets, models, solvers). In the Fig. 2(a) tables, we show ingested information in detail: *a)* history of user activities (e.g., the first *train* command uses model v_1 and solver v_1 and generates logs v_1 and weights v_1), *b)* versions and changes of entities (e.g., weights v_1, v_2 and v_3) and derivations among those entities (e.g., model v_2 is derived from model v_1), and *c)* provenance records as associated properties to activities and entities (e.g., dataset is copied from some url, Alice changes a pool layer type to AVG in v_2, accuracy in logs v_3 is 0.75).

Example 2: In Fig. 2(c), we show the corresponding provenance graph of the project lifecycle illustrated in Example 1. Names of the vertices (e.g., 'model-v1', 'train-v3', 'Alice') are made by using their representative properties and suffixed using the version ids to distinguish different snapshots. Activity vertices are ordered from left to right w.r.t. the temporal order of their executions. We label the edges using their types and show a subset of the edges in Fig. 2(a) to illustrate usages of five relationship types. Note there are many snapshots of the same artifact in different versions, and between the versions, we maintain derivation edges 'wasDerivedFrom' (D) for efficient versioning storage. The property records are shown as white boxes but not treated as vertices in the property graph.

4 Capturing and Ingesting Provenance

The biggest challenge for a system like ProvDB is capturing the requisite provenance and context information. Transparent instrumentation approaches, where the information is captured with minimal involvement from the users are most likely to be used in practice, but are difficult to develop (since they must cover different user environments) and may generate large amounts of frivolous information (since they cannot be targetted). On the other hand, approaches that require explicit instrumentation or hints from the users can capture more useful information, but require significant discipline from the users and rarely work in the long run. We use a hybrid

approach in PROVDB, where we transparently instrument some of the most common data science environments, but allow users to explcitly instrument and send additional information to the PROVDB server. Here we briefly enumerate the ingestion mechanisms that PROVDB currently supports, which include a general-purpose UNIX shell-based ingestion framework, ingestion of DVCS versioning information, and a mechanism called **file views** which is intended to both simplify workflow and aid in fine-grained provenance capture.

Shell command-based Ingestion Framework: The provenance ingestion framework is centered around the UNIX commandline shell (e.g., `bash, zsh`, etc). We provide a special command called `provdb` that users can prefix to any other command, and that triggers provenance ingestion (shell extensions can be leveraged to do this automatically for every command). Each run of the command results in creation of a new *implicit* version, which allows us to capture the changes at a fine granularity. In other words, we `commit` a new version before and after running every such command. These implicit versions are kept separate from the explicit versions created by a user through use of `git commit`, and are not visible to the users. A collection of *ingestors* is invoked by matching the command that was run, against a set of regular expressions registered a priori along with the ingestors. PROVDB schedules ingestor to run before/during/after execution the user command, and expects the ingestor to return a JSON property graph consisting of a set of key-value pairs denoting properties of the snapshots or derivations. Note that, it is vital that we take a snapshot before running the command in order to properly handle modifications made using external tools (e.g., text editors) or commands not prefixed with `provdb`. That way we can capture a modification even if we don't know the specific operation(s) that made that modification. An ingestor can also provide *record-level provenance* information, if it is able to generate such information.

A default ingestor handles abitrary commands by parsing them following POSIX standard (IEEE 1003.1-2001) to annotate utility, options, option arguments and operands. For example, `mkdir -p dir` is parsed as utility *mkdir*, option *p* and operand *dir*. Concatenations of commands are decomposed and ingested separately, while a command with pipes is treated as a single command. If an external tool has been used to make any edits (e.g., a text editor), an implicit version is created next time `provdb` is run, and the derivation information is recorded as missing.

Caffe (Deep Learning Framework) Ingestor: PROVDB also supports several specialized ingestion plugins and configurations to cover important data science workflows. In particular, it has an ingestor capable of ingesting provenance information from runs of the `Caffe` deep learning framework. If the command is recognized as a `caffe` command, then the ingestor parses the configuration file (passed as an argument to the command) to extract and record the *learning hyperparameters*. The ingestor also extracts accuracy and loss scores on an iteration-by-iteration basis from the result logging file. All of these are attached as properties to the coarse-grained derivation edge that is recorded in the provenance graph.

File Views: PROVDB provides a functionality called *file views* to assist dataset transformations and to ingest provenance among data files. Analogous to views in relational databases, a file view defines a virtual file as a transformation over an existing file. A file view can be defined either: (a) as a script or a sequence of commands (e.g., `sort | uniq -c`, which is equivalent to an aggregate count view), or (b) as an SQL query where the input files are treated as tables. For instance, the following query counts the rows per label that a classifier predicts wrongly comparing with ground truth.

```
provdb fileview -c -n='results.csv' -q='
  select t._c2 as label, count(*) as err_cnt
  from {testfile.csv} as t, {predfile.csv} as r
  where t._c0 = r._c0 and t._c2 != r._c2 group by t._c2'
```

The SQL feature is implemented by loading the input files into an in-memory `sqlite` database and executing the query against it. Instead of creating a view, the same syntax can be used for creating a new file instead, saving a user from coding similar functionality.

File views serves as an example of a functionality that can help make the ad hoc process of data science

more structured. Aside from making it easier to track dependencies, SQL-based file views also enable capturing record-level provenance by drawing upon techniques developed over the years for provenance in databases.

User Annotations: Apart from plugin framework, PROVDB GUI allows users to organize, add, and annotate properties, along with other query facilities. The user can annotate project properties, such as usage descriptions for collaborations on artifacts, or notes to explain rationale for a particular derivation. A user can also annotate a property as parameter and add range/step to its domains, which turns a derivation into a template and enables batch run of an experiment. For example, a grid search of a template derivation on a start snapshot can be configured directly in the UI. Maintaining such user annotations (and file views discussed above) as the datasets evolve is a complicated issue in itself [25].

5 Storing, Querying and Analyzing Provenance Graphs

Provenance graphs represent a unique and new usage scenario for graph databases, that necessitates development of new techniques both for storing and for querying them. In particular, the following characteristics of the provenance graph need to be considered:

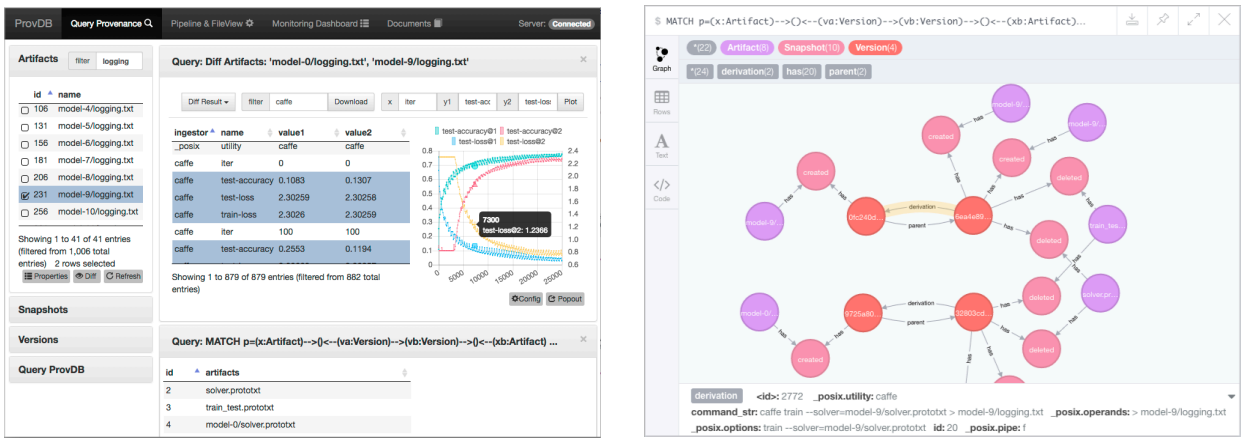
- **Versioned Artifacts**: Each entity is a point-in-time snapshot of some artifact in the project, e.g., the query 'accuracy of model-v1' discusses a particular ***snapshot*** of the model artifact, while 'what are the common updates for *solver* before `train`' refer to the ***artifact*** but not an individual snapshot. This impacts both storage and declarative query facilities. First, given the potentially large *properties* associated with the nodes in the graph (e.g., scripts, notebooks, log files), the standard model of storing each node independently of the others does not work because of the verbosity and duplication in the properties of the data [29]. Second, any high-level query languages must support versioning as a first-class construct [18].

- **Evolving Workflows**: Data science lifecycle is exploratory and collaborative in nature, so *there is no static workflow skeleton, and no clear boundaries for individual runs* in contrast with workflow systems [20]; e.g., the modeling methods may change (e.g., from SVM to neural networks), the data processing steps may vary (e.g., split, transform or merge data files), and the user-committed versions may be mixed with code changes, error fixes, thus may not serve as query boundaries. Thus, we cannot assume the existence of a workflow skeleton and we need to allow flexible boundary conditions.

- **Partial Knowledge in Collaboration**: Each team member may work on and be familiar with a subset of artifacts and activities, and may use different tools or approaches, e.g., in Example 1, Alice and Bob use different ways to improve accuracy. When querying retrospective provenance of the snapshots or understanding activity process over team behaviors, the user may only have partial knowledge at query time, thus may find it difficult to compose the right query. Hence, it is important to support queries with partial information reflecting users' understanding and induce correct result.

- **Verboseness for Usage**: In practice, the provenance graph would be very verbose for humans to use and in large volume for the system to manage. Depending on the granularity, storing the graphs could take dozens of GBs within several minutes [9]. With similar goals to recent research efforts [22, 34, 36, 2], our system aims to let the users understand the essence of provenance at their preference level by transforming the provenance graph. *R4: The query facility should be scalable to large graph and process queries efficiently.*

In addition, most of the provenance query types of interest involve paths [1], and require returning paths instead of answering yes/no queries like reachability [22]. Writing queries to utilize the lifecycle provenance is beyond the capabilities of the pattern matching query (BPM) and regular path query (RPQ) support in popular graph databases [7, 6, 42]. For example, answering 'how is today's result file generated from today's data file' requires a segment of the provenance graph that includes not only the mentioned files but also others that are not



(a) Diff Artifacts (Result logging files for two deep neural networks)

(b) Cypher Query to Find Related Changes via Derivations

Figure 3: Illustration of PROVDB interfaces

on the lineage paths and the users may not know at all (e.g., 'a configuration file'); answering 'how do the team members typically generate the result file from the data file?' requires summarizing several query results of the above query while keeping the result meaningful from provenance perspective.

This lack of proper query facilities in modern graph databases not only limits the value of lifecycle provenance systems for data science, but also of other provenance systems. The specialized query types of interest in the provenance domain [1, 22] had often led provenance systems to implement specialized storage systems [38] and query interfaces [16, 4] on their own [20]. Recent works in the provenance community propose various task-specific graph transformations, which are essentially different template queries from the graph querying perspective; these include grouping vertices together to handle publishing policies [34], aggregating similar vertices to understand commonalities and outliers [36], segmenting a provenance graph for feature extractions in cybersecurity [2].

To handle the wide variety of usage scenarios, PROVDB currently supports several different mechanisms to interact with the provenance information, that we briefly discuss next.

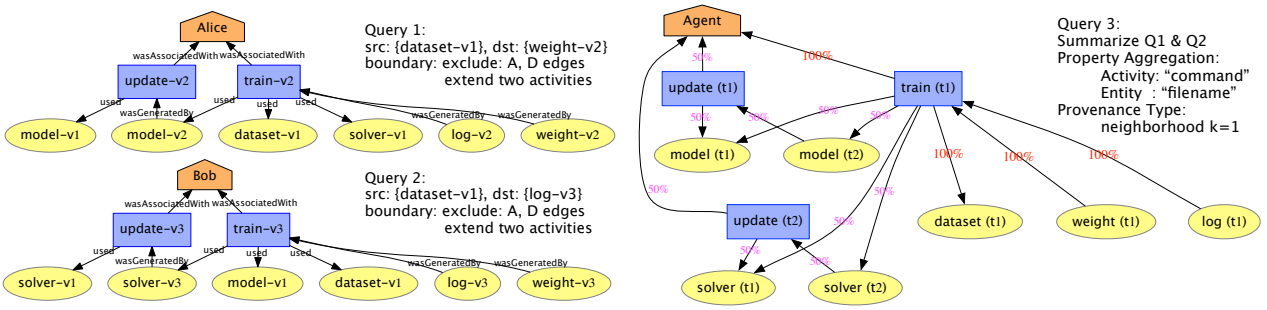
5.1 Interactive Queries using Cypher

In a collaborative workflow, provenance queries to identify what revision and which author last modified a line in an artifact are common (e.g., `git blame`). PROVDB allows such queries at various levels (version, artifact, snapshot, record) and also allows querying the properties associated with the different entities (e.g., details of what parameters have been used, temporal orders of commands, etc). In fact, all the information exposed in the property graph can be directly queried using the Neo4j Cypher query language, which supports graph traversal queries and aggregation queries.

The latter types of queries are primarily limited by the amount of context and properties that can be automatically ingested. Availability of this information allows users to ask more meaningful queries like: *what scikit-learn script files contain a specific sequence of commands; what is the learning accuracy curve of a caffe model artifact; enumerate all different parameter combinations that have been tried out for a given learning task*, and so on.

Many such queries naturally result in one or more time series of values (e.g., properties of an artifact over time as it evolves, results of "diff" queries discussed below); PROVDB supports a uniform visual interface for plotting such time series data, and comparing two different time series (see below for an example).

Illustrative Example: Figure 3 shows the PROVDB Web GUI using a `caffe` deep learning project. In this project, 41 deep neural networks are created for a face classification task. The user tries out models by editing



(a) Segmentation Query Examples

(b) Summarization Query Examples

Figure 4: Examples of Segmentation and Summarization Queries

and training models. In Fig 3(a), an introspection query asks how different are two trained models (*model-0* and *9*). Using the GUI, the user filters artifacts, and diffs their result logging files. In the right side query result pane, the ingested properties are diffed. The `caffe` ingestor properties are numerical time series; using the provided charting tool, the user plots the training loss and accuracy against the iteration number. From the results, we can see that *model-9* does not train well in the beginning, but ends up with similar accuracy. To understand why, a deep diff between the two can be issued in the GUI and complex Cypher queries can be used as well. In Fig. 3(b), the query finds previous derivations and shared snapshots, which are training config files; more introspection can be done by finding changed hyperparameters.

5.2 Graph Segmentation and Summarization Queries

As discussed above, the extant graph query languages are not rich enough to support common provenance analysis queries. One of our goals with ongoing work is to initiate a more systematic study of abstract graph operators that modern graph databases need to support to be a viable option for storing provenance graphs. By observing the characteristics of the provenance graph in analytics lifecycle and identifying the requirements for the query facilities, we propose two new graph operators as a starting point:

Segmentation: A very important provenance query type of interest is querying ancestors and descendants of entities [20, 22]. In our context, the users introspect the lifecycle and identify issues by analyzing dependencies among snapshots. Lack of a workflow skeleton and clear boundaries makes the provenance query more difficult. Moreover the user may not be able to specify all interested entities in a query due to partial knowledge. We propose a segmentation operator that takes sets of source and destination entities, and induces other important unknown entities satisfying a set of specified boundary criteria.

Example 3: In Fig. 4(a), we show two examples of segmentation query. In Query 1 (Q_1), Bob was interested in what Alice did in version v_2. He did not know the details of activities and the entities Alice touched, instead he set $\{dataset\}$, $\{weight\}$ as querying entities to see how the *weight* in Alice's version v_2 was connected to the *dataset*. He filtered out uninterested edge types (.e.g, A, D) and excluded actions in earlier commits (e.g., v_1) by setting the boundaries as two activities away from those querying entities. The system found connections among the querying entities, and included vertices within the boundaries. After interpreting the result, Bob knew Alice updated the model definitions in *model*. On the other hand, Alice would understand how Bob improved the accuracy and learn from him. In Query 2 (Q_2), instead of learned *weight*, accuracy property associated *log* entity is used as querying entity along with *dataset*. The result showed Bob only updated solver file.

Summarization: In workflow systems, querying the workflow skeleton (aka prospective provenance) is an important use case [12] and one of the *provenance challenges* [1]. In our context, even though a static workflow skeleton is not present, summarizing a skeleton of similar processing pipelines, showing commonalities and

identifying abnormal behaviors are very useful query capabilities. However, general graph summarization techniques [26] are not applicable to provenance graphs due to the subtle provenance meanings and constraints of the data model [35, 34, 36]. Hence, we propose a summarization operator with multi-resolution capabilities for provenance graphs. It operates over query results of segmentation and allows tuning the summary by ignoring vertex details and characterizing local structures, and ensures provenance meaning through path constraints.

Example 4: In Fig. 4(b), an outsider to the team (e.g., auditor, new team member) wanted to see the activity overview in the project. Segmentation queries (e.g., Q_1, Q_2 in Fig. 4(a)) only show individual trails of the analytics process at the snapshot level. The outsider issued a summarization query, Query 3 (Q_3), by specifying the aggregation over three types of vertices (viewing Alice and Bob as an abstract team member, ignoring details of files and activities), and defining the provenance meanings as a 1-hop neighborhood. The system merged Q_1 and Q_2 into a summary graph. In Fig. 4(b), the vertices suffixed name with provenance types to show alternative generation process, while edges are labeled with their frequency of appearance among segments. The query issuer would vary the summary conditions at different resolutions.

See [31] for a more in-depth discussion, including new query execution techniques for such queries.

6 Related Work

There has been much work on scientific workflow systems over the years, with some of the prominent systems being Kepler, Taverna, Galaxy, iPlant, VisTrails, Chimera, Pegasus, to name a few[2]. These systems often center around creating, automating, and monitoring a well-defined workflow or data analysis pipeline. But they cannot easily handle fast-changing pipelines, and typically are not suitable for ad hoc collaborative data science workflows where clear established pipelines may not exist except in the final, stable versions. Moreover, these systems typically do not support the entire range of tools or systems that the users may want to use, they impose a high overhead on the user time and can substantially increase the development time, and often require using specific computational environment. Further, many of these systems require centralized storage and computation, which may not be an option for large datasets.

Many users find **version control systems** (e.g., `git, svn`) and related hosted platforms (e.g., `GitHub, GitLab`) more appropriate for their day-to-day needs. However, these systems are typically too "low-level", and don't support capturing higher-level workflows or provenance information. The versioning API supported by these systems is based on a notion of files, and is not capable of allowing data researchers to reason about data contained within versions and the relationships between the versions in a holistic manner. PROVDB can be seen as providing rich introspection and querying capabilities those systems lack.

Recently, there is emerging interest in developing systems for managing different aspects in the modeling lifecycle, such as building modeling lifecycle platforms [11], accelerating iterative modeling process [45], managing developed models [44, 33], organizing lifecycle provenance and metadata [21, 30, 41], auto-selecting models [27], hosting and discovering reference models [43, 32], and assisting collaboration [24]. Issues of querying evolving and verbose provenance effectively are typically not considered in that work.

There has also been much work on **provenance**, with increasing interest in the recent years. The work in that space can be roughly categorized in two types: data provenance (aka fine-granularity) and workflow provenance (aka coarse-granularity). Data provenance is discussed in dataflow systems, such as RDBMS, Pig Latin, and Spark [19, 3, 23], while workflow provenance studies address complex interactions among high-level conceptual components in various computational tasks, such as scientific workflows, business processes, and cybersecurity [20, 20, 12, 9]. Unlike retrospective query facilities in scientific workflow provenance systems [20], their processes are predefined in *workflow skeletons*, and multiple executions generate different instance-level

[2]We omit the citations for brevity; a more detailed discussion with citations can be found in [30, 31]

provenance *run graphs* and have clear boundaries. Taking advantages of the skeleton, there are lines of research for advanced ancestry query processing, such as defining user views over such skeleton to aid queries on verbose run graphs [16], executing reachability query on the run graphs efficiently [8], storing run graphs generated by the skeletons compactly [5], and using visualization as examples to ease query construction [10].

Most relevant work is querying evolving script provenance [39, 40]. Because script executions form clear run graph boundary, query facilities to visualize and difference execution run graphs are proposed. In our context, as there are no clear boundaries of run graphs, it is crucial to design query facilities allowing the user to express the logical run graph segments and specify the boundary conditions first. Our method can also be applied on script provenance by segmenting within and summarizing across evolving run graphs.

7 Conclusion

In this paper, we presented our approach to simplify lifecycle management of ad hoc, collaborative analysis workflows that are becoming prevalent in most application domains today. Our system, PROVDB, is based on the premise that a large amount of provenance and metadata information can be captured passively, and analyzing that information in novel ways can immensely simplify the day-to-day processes undertaken by data analysts. Our initial experience with using this prototype for a deep learning workflow (for a computer vision task) shows that even with limited functionality, it can simplify the bookkeeping tasks and make it easy to compare the effects of different hyperparameters and neural network structures. However, many interesting and hard systems and conceptual challenges remain to be addressed in capturing and exploiting such information to its fullest extent. In particular, there are many potential ways to use the rich provenance and context metadata that PROVDB collects to answer explanation queries to understand the origins of a piece of data or introspection queries to identify practices like *p-value hacking*. Continuous monitoring of this data can also help identify issues during deployment of data science pipelines (e.g., *concept drifts*). In addition, the large volumes of versioned provenance information requires development of new graph data management systems, including new graph query operators, that we are actively investigating in ongoing work.

References

[1] Provenance Challenge. `http://twiki.ipaw.info`. Accessed: 2017-07.

[2] R. Abreu, D. Archer, E. Chapman, J. Cheney, H. Eldardiry, and A. Gascón. Provenance segmentation. In *8th Workshop on the Theory and Practice of Provenance (TaPP)*, 2016.

[3] Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. Putting lipstick on pig: Enabling database-style workflow provenance. *PVLDB*, 5(4):346–357, 2011.

[4] M. K. Anand, S. Bowers, and B. Ludäscher. Techniques for efficiently querying scientific workflow provenance graphs. In *EDBT*, 2010.

[5] M. K. Anand, S. Bowers, T. M. McPhillips, and B. Ludäscher. Efficient provenance storage over nested data collections. In *EDBT*, 2009.

[6] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc. Foundations of modern query languages for graph databases. *ACM Computing Surveys*, 50(5):68:1–68:40, 2017.

[7] P. B. Baeza. Querying graph databases. In *PODS*, 2013.

[8] Z. Bao, S. B. Davidson, S. Khanna, and S. Roy. An optimal labeling scheme for workflow provenance using skeleton labels. In *SIGMOD*, 2010.

[9] A. M. Bates, D. Tian, K. R. B. Butler, and T. Moyer. Trustworthy whole-system provenance for the linux kernel. In *24th USENIX Security Symposium*, 2015.

[10] L. Bavoil, S. P. Callahan, C. E. Scheidegger, H. T. Vo, P. Crossno, C. T. Silva, and J. Freire. Vistrails: Enabling interactive multiple-view visualizations. In *16th IEEE Visualization Conference (VIS)*, 2005.

[11] D. Baylor et al. TFX: A tensorflow-based production-scale machine learning platform. In *KDD*, 2017.

[12] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying business processes. In *VLDB*, 2006.

[13] A. P. Bhardwaj, S. Bhattacherjee, A. Chavan, A. Deshpande, A. Elmore, S. Madden, and A. Parameswaran. DataHub: Collaborative Data Science & Dataset Version Management at Scale. *CIDR*, 2015.

[14] S. Bhattacherjee, A. Chavan, S. Huang, A. Deshpande, and A. Parameswaran. Principles of Dataset Versioning: Exploring the Recreation/Storage Tradeoff. *PVLDB*, 2015.

[15] S. Bhattacherjee, A. Deshpande. Storing and querying versioned documents in the cloud. In *ICDE*, 2018.

[16] O. Biton, S. C. Boulakia, S. B. Davidson, and C. S. Hara. Querying and managing provenance through user views in scientific workflows. In *ICDE*, 2008.

[17] A. Chavan and A. Deshpande. DEX: query execution in a delta-based storage system. In *SIGMOD*, 2017.

[18] A. Chavan, S. Huang, A. Deshpande, A. Elmore, S. Madden, and A. Parameswaran. Towards a Unified Query Language for Provenance and Versioning. *Theory and Practice of Provenance (TaPP)*, 2015.

[19] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *FnT*, 2009.

[20] J. Freire, D. Koop, E. Santos, and C. T. Silva. Provenance for computational tasks: A survey. *Computing in Science & Engineering*, 2008.

[21] J. M. Hellerstein, V. Sreekanti, J. E. Gonzalez, J. Dalton, A. Dey, S. Nag, K. Ramachandran, S. Arora, A. Bhattacharyya, S. Das, M. Donsky, G. Fierro, C. She, C. Steinbach, V. Subramanian, and E. Sun. Ground: A data context service. In *CIDR*, 2017.

[22] D. A. Holland, U. J. Braun, D. Maclean, K.-K. Muniswamy-Reddy, and M. I. Seltzer. Choosing a data model and query language for provenance. In *Intl. Provenance and Annotation Workshop (IPAW)*, 2010.

[23] M. Interlandi, K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. D. Millstein, and T. Condie. Titian: Data provenance support in spark. *PVLDB*, 9(3):216–227, 2015.

[24] E. Kandogan, M. Roth, P. M. Schwarz, J. Hui, I. Terrizzano, C. Christodoulakis, and R. J. Miller. Labbook: Metadata-driven social collaborative data analysis. In *IEEE International Conference on Big Data*, 2015.

[25] R. H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Computing Surveys (CSUR)*, 22(4):375–409, 1990.

[26] A. Khan, S. S. Bhowmick, and F. Bonchi. Summarizing static and dynamic big graphs. *PVLDB*, 2017.

[27] A. Kumar, R. McCann, J. F. Naughton, and J. M. Patel. Model selection management systems: The next frontier of advanced analytics. *SIGMOD Record*, 44(4):17–22, 2015.

[28] M. Maddox, D. Goehring, A. J. Elmore, S. Madden, A. G. Parameswaran, and A. Deshpande. Decibel: The relational dataset branching system. *PVLDB*, 9(9):624–635, 2016.

[29] R. Mavlyutov, C. Curino, B. Asipov, and P. Cudré-Mauroux. Dependency-driven analytics: A compass for uncharted data oceans. In *CIDR*, 2017.

[30] H. Miao, A. Chavan, and A. Deshpande. ProvDB: lifecycle management of collaborative analysis workflows. In *2nd Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD*, 2017.

[31] H. Miao and A. Deshpande. Understanding data science lifecycle provenance via graph segmentation and summarization. CoRR abs/1810.04599, 2018.

[32] H. Miao, A. Li, L. S. Davis, and A. Deshpande. On model discovery for hosted data science projects. In *Workshop on Data Management for End-To-End Machine Learning, DEEM@SIGMOD*, 2017.

[33] H. Miao, A. Li, L. S. Davis, and A. Deshpande. Towards unified data and lifecycle management for deep learning. In *ICDE*, 2017.

[34] P. Missier, J. Bryans, C. Gamble, V. Curcin, and R. Dánger. ProvAbs: Model, policy, and tooling for abstracting PROV graphs. In *Intl. Provenance and Annotation Workshop (IPAW)*, 2014.

[35] P. Missier and L. Moreau. PROV-dm: The PROV data model. W3C recommendation, W3C, 2013. http://www.w3.org/TR/2013/REC-prov-dm-20130430/.

[36] L. Moreau. Aggregation by provenance types: A technique for summarising provenance graphs. In *Proceedings Graphs as Models, GaM@ETAPS 2015, London, UK, 11-12 April 2015.*, pages 129–144, 2015.

[37] L. Moreau and P. Groth. PROV-overview. W3C note, W3C, 2013. http://www.w3.org/TR/2013/NOTE-prov-overview-20130430/.

[38] K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference*, 2006.

[39] L. Murta, V. Braganholo, F. Chirigati, D. Koop, and J. Freire. noworkflow: Capturing and analyzing provenance of scripts. In *International Provenance and Annotation Workshop (IPAW)*, 2014.

[40] J. F. Pimentel, J. Freire, V. Braganholo, and L. Murta. Tracking and analyzing the evolution of provenance from scripts. In *International Provenance and Annotation Workshop (IPAW)*, 2016.

[41] S. Schelter, J. Boese, J. Kirschnick, T. Klein, and S. Seufert. Automatically tracking metadata and provenance of machine learning experiments. In *NIPS Workshop on ML Systems (LearningSys)*, 2017.

[42] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi. PGQL: a property graph query language. In *Proc. of the Intl. Workshop on Graph Data Management Experiences and Systems (GRADES)*, 2016.

[43] J. Vanschoren, J. N. van Rijn, B. Bischl, and L. Torgo. Openml: networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60, 2013.

[44] 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 *Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD*, 2016.

[45] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. *ACM Transactions on Database Systems (TODS)*, 41(1):2, 2016.

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.

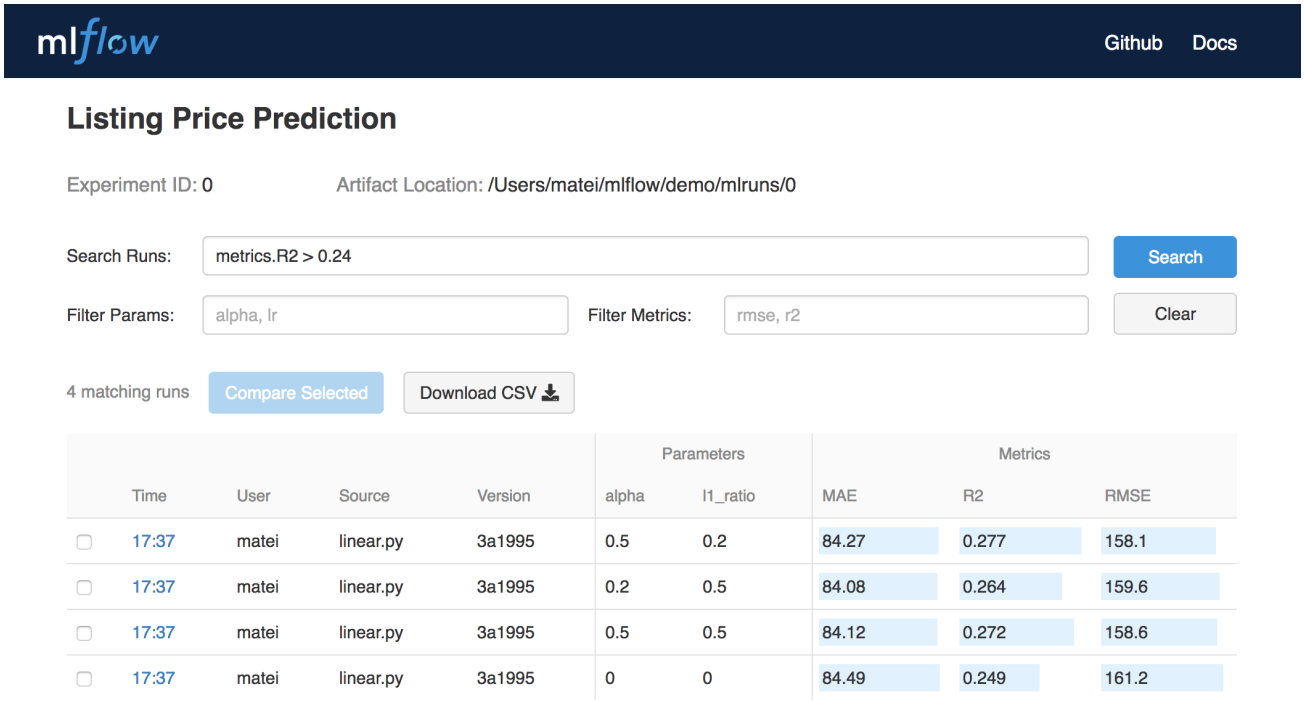


Figure 1: MLflow Tracking UI showing several runs in an experiment. Clicking each run lists its metrics, artifacts and output details and lets the user post comments about the run.

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.

From the Edge to the Cloud: Model Serving in ML.NET

Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Markus Weimer and Matteo Interlandi
{yunseong, bgchun}@snu.ac.kr, alberto.scolari@polimi.it, {mweimer, mainterl}@microsoft.com

Abstract

As Machine Learning (ML) is becoming ubiquitously used within applications, developers need effective solutions to build and deploy their ML models across a large set of scenarios, from IoT devices to the cloud. Unfortunately, the current state of the art in model serving suggests to deliver predictions by running models in containers. While this solution eases the operationalization of models, we observed that it is not flexible enough to address the variety of ML scenarios encountered in large companies such as Microsoft. In this paper, we will overview ML.NET—a recently open sourced ML pipeline framework—and describe how ML models written in ML.NET can be seamlessly integrated into applications. Finally, we will discuss how model serving can be cast to a database problem, and provide insights on our recent experience in building a database optimizer for ML.NET pipelines.

1 Introduction

Machine Learning (ML) is transitioning from an art and science into a technology readily available to every developer. In the near future, every application on every platform will rely on trained models for functionalities that evade traditional programming due to their complex statistical nature. This unfolding future—where most applications make use of at least one model—profoundly differs from the current practice in which data science and software engineering are performed in separate and different processes and sometimes even by different teams and organizations. Furthermore, in current practice, models are routinely deployed and managed in completely distinct ways from other software artifacts: while typical software libraries are seamlessly compiled and run on a myriad of heterogeneous devices, ML models are often implemented in high-level languages (e.g., Python) and relegated to be run as web services in remotely hosted containers [3, 10, 12, 15, 22]. Ad-hoc solutions or bespoken re-engineering strategies can be pursued to address specific applications (e.g., low latency scenarios as in obstacle detection for self-driving cars), but these efforts are not scalable in general. Therefore they are inappropriate for enterprise-scale ML needs as the one that can be observed in large companies. This pattern not only severely limits the kinds of applications one can build with ML capabilities, but also discourages developers from embracing ML as a core component of applications.

ML.NET [9] is the end-to-end solution provided by Microsoft to address the above problems. ML.NET is an open source ML framework allowing developers to author and deploy in their applications complex ML pipelines composed of data featurizers and state of the art ML models. Pipelines implemented and trained using ML.NET can be seamlessly surfaced for prediction without any modification, and adding a model into an application is as easy as importing the ML.NET runtime and binding the input/output data sources. ML.NET's

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

ability to capture full, end-to-end pipelines has been demonstrated by the fact that thousands of Microsoft's data scientists and developers have been using ML.NET over the past decade, infusing hundreds of products and services with ML models used by hundreds of millions of users worldwide.

In this paper, we will give an overview of current state of the art practices for surfacing ML pipelines predictions into applications, and we will highlight the limitations of using containers to operationalize models for application consumption. We will then introduce how ML.NET allows developers to design their data-driven applications end-to-end without having to rely on any external resource. Finally, we will present few challenges we have observed in running ML.NET models in production, and how these can be addressed by considering models as Direct Acyclic Graphs (DAGs) of operators instead of black-box executable code. Specifically, we will describe a new ML.NET runtime for model scoring called PRETZEL. PRETZEL treats model scoring as a database problem and, as such, it employs database techniques to optimize the performance of predictions.

2 Background: ML Pipelines

Many ML frameworks such as Spark MLlib [2], H2O [6], Scikit-learn [23], or Microsoft ML.NET [9] allow data scientists to declaratively author pipelines of transformations for better productivity and easy operationalization. Model pipelines are internally represented as DAGs of pre-defined operators [1] comprising *data transformations* and *featurizers* (e.g., string tokenization, hashing, etc.), and *ML models* (e.g., decision trees, linear models, SVMs, etc.). Figure 1 shows an example pipeline for text analysis whereby input sentences are classified according to the expressed sentiment.

ML.NET is an open-source C# library running on a managed runtime with garbage collection and Just-In-Time (JIT) compilation [2]. ML.NET's main abstraction is called DataView, which borrows ideas from the database community. Similarly to (intensional) database relations, the DataView abstraction provides compositional processing of schematized data, but specializes it for ML pipelines. In relational databases, the term *view* typically indicates the result of a query on one or more tables (base relations) or views, and is generally immutable [18]. Views have interesting properties that differentiate them from tables and make them appropriate abstractions for ML: (1) views are *composable*—new views are formed by applying transformations (queries) over other views; (2) views are *virtual*, i.e., they can be lazily computed on demand from other views or tables without having to materialize any partial results; and (3) since a view does not contain values but merely computes values from its source views, it is *immutable* and *deterministic*: the exact same computation applied over the same input data always produces the same result. Immutability and deterministic computation enables transparent data caching (for speeding up iterative computations such as ML algorithms) and safe parallel execution. DataView inherits the aforementioned database view properties, namely: composability, lazy evaluation, immutability, and deterministic execution.

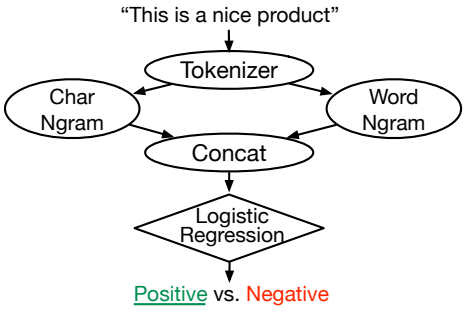


Figure 1: A Sentiment Analysis (SA) pipeline consisting of operators for featurization (ellipses), followed by a ML model (diamond). *Tokenizer* extracts tokens (e.g., words) from the input string. *Char* and *Word Ngrams* featurize input tokens by extracting n-grams. *Concat* generates a unique feature vector which is then scored by a *Logistic Regression* predictor. This is a simplification: the actual DAG contains about 12 operators.

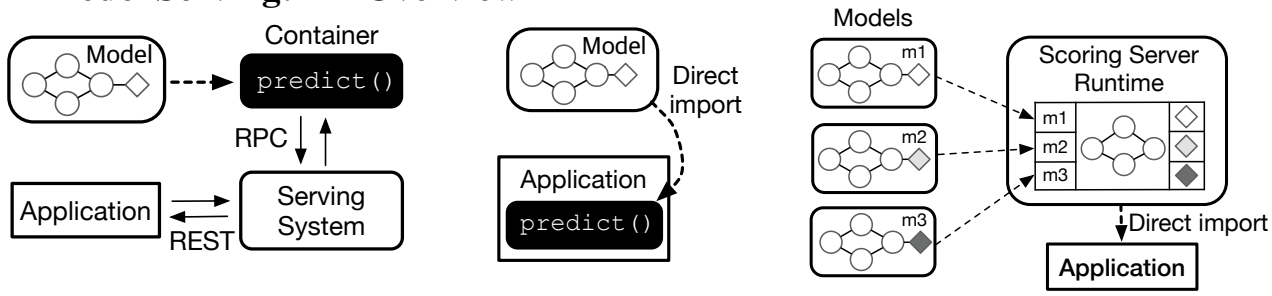
In ML.NET, pipelines are represented as DAGs of operators, each of them implementing the DataView

[1] Note that user-defined code can still be executed through a second order operator accepting arbitrary UDFs.

[2] Unmanaged C/C++ code can also be employed to speed up processing when possible.

interface and executing a featurization step or a ML model. Upon pipeline initialization, the operators composing the model DAG are analyzed and arranged to form a chain of function calls which, at execution time, are JIT-compiled to form a unique function executing the whole DAG on a single call. Operators are able to gracefully and efficiently handle high-dimensional and large datasets thanks to *cursoring*, which resembles the well-known iterator model of databases [17]: within the execution chain, inputs are pulled through each operator to produce intermediate vectors that are input to the following operators, until a prediction or a trained model is rendered as the final output of the pipeline. We refer readers to [13] for further details on ML.NET.

3 Model Serving: An Overview



(a) Models are deployed into containers, connected to a Serving System via RPC. To score the models, applications must call into the Web Server hosted on the Serving System using a REST API.

(b) ML.NET allows developers to deploy models directly into their applications and without any additional custom engineering effort.

(c) With the white-box approach, models' structural information can be used to optimize the execution. As in (b), applications can directly access the models by importing the Scoring Server Runtime.

Figure 2: Three different ways to deploy models into applications. (a) and (b) represent two variations of the black-box approach where the invocation of the function chain (e.g., `predict()`) on a pipeline returns the result of the prediction. (c) shows the white-box approach.

In this Section, we survey how models are commonly operationalized in industry. The most popular (and easiest) method to deploy ML models (in general, and pipelines in particular) is what we refer to as *black box*. Under this approach, internal pipelines' information and structures are not considered inasmuch as pipelines are opaque executable code accepting some input record(s) and producing a prediction. Within the black box approach, there are two possible ways for a developer to deploy models, and consequently for an application to request and consume predictions. The first option (à la Clipper [3], depicted in Figure 2(a) and further described in Section 3.1) is to ship models into containers (e.g., Docker [4]) wired with proper Remote Procedure Calls (RPCs) to a Web Server. With this approach, predictions have to go through the network and be rendered on the cloud: low latency or edge scenarios are therefore out of scope. The second option (Figure 2(b) and detailed in Section 3.2) is to integrate the model logic directly into the application (à la ML.NET: the model is a dynamic library the application can link). This approach is suitable for the cloud as well as for edge devices and it unlocks low latency scenarios. However, we still find this approach sub-optimal with respect to customized solutions because it ships the same training pipeline code for prediction. In fact, while using the same code is a great advantage because it removes the need for costly translation work, it implicitly assumes that training and prediction happen in the same regime. However, prediction serving is much more latency sensitive. The *white box* approach (Section 3.3) depicted in Figure 2(c) tackles the aforementioned problem by considering ML pipelines not anymore as black-box artifacts, but as DAGs of operators, and therefore it tries to rewrite them using optimizations specifically tailored to prediction-time scenarios. We next provide additional details on each of the three possibilities.

3.1 Deploying Models into Containers

Most serving systems in the state of the art [3, 8, 10, 12, 15, 22] aim to minimize the burden of deploying trained pipelines in production by serving them in containers, where the same code is used for both training and inference [3]. This design allows decoupling models from serving system development, and eases the implementation of mechanisms and policies for fault tolerance and scalability. Furthermore, hardware acceleration can be exploited when available. A typical container-based, model serving system follows the design depicted in Figure 2(a): containers are connected to a Serving System (e.g., Clipper) via RPC, and, to score models, applications should contact the Serving System by invoking a Web Server through a REST API. Developers are responsible for setting up the communication between their applications and the Serving System, but this is in general an easy task as most Serving Systems provide convenient libraries (e.g., Microsoft ML Server [8]). Implementing model containers for new ML frameworks and integrating them with the Serving System requires a reasonable amount of effort: for example, a graduate student spent a couple of weeks to implement the protocol for integrating an ML.NET container into Clipper.

Limitations. While serving models via containers greatly eases operationalization, we found though that it is not flexible enough to accommodate the requirements stemming from running ML models at Microsoft scale. For instance, containers allow resource isolation and thus achieve effective multitenancy, but each container comes with its own runtime (e.g., an ML.NET instance) and set of processes, thus introducing memory overheads that can possibly be higher than the actual model size. Additionally, the RPC layer and REST API introduce network communication costs, which are especially relevant for models that have millisecond-level prediction latency. Finally, only a restricted set of optimizations are available, specifically those that do not require any knowledge of the internals of the pipelines; examples are handling multiple requests in batches and caching prediction results if some inputs queries are frequently issued for the same pipeline. Instead, any optimization specific to the model is out of scope, as from the inscrutable nature of its container [4].

3.2 Importing Models Directly into Applications

At Microsoft, we have encountered the problem of model deployment across a wide spectrum of applications ranging from Bing Ads to Excel, PowerPoint and Windows 10, and running over diverse hardware configurations ranging from desktops, to custom hardware (e.g., XBox and IoT devices) and to high performance servers [1, 5, 7]. To allow such diverse use cases, an ML toolkit deeply embedded into applications should not only satisfy several intrinsic constraints (e.g., scale up or down based on the available main memory and number of cores) but also preserve the benefits commonly associated with model containerization, i.e., (1) it has to capture the full prediction pipeline that takes a test example from a given domain (e.g., an email with headers and body) and to produce a prediction that can often be structured and domain-specific (e.g., a collection of likely short responses); and (2) it has to allow to seamlessly carry the complete train-time pipeline into production for model inference. This later requirement is the keystone for building effective, reproducible pipelines [27].

ML.NET is able to implement all the above desiderata. Once a model is trained in ML.NET, the full training pipeline can be saved and directly surfaced for prediction serving without any external modification. Figure 2(b) depicts the ML.NET solution for black-box model deployment and serving: models are integrated into application logic natively and predictions can be served in any OS (Linux, Windows, Android, MacOS) or device supported by the .NET Core framework. This approach removes the overhead of managing containers and implementing RPC functionalities to communicate with the Serving System. In this way, application developers

[3] Note that TensorFlow Serving [12] is slightly more flexible since users are allowed to split model pipelines and serve them into different containers (called *servables*). However, this process is manual and occurs when building the container image, ignoring the final running environment.

[4] In the case of ML.NET pipelines, the C# runtime in the container can optimize *the code* of the model, but not the model itself as we propose in the following.

are facilitated for writing applications with ML models inside. Nevertheless, models can still be deployed in the cloud if suggested by the application domain (e.g., because of special hardware requirements).

Limitations. ML.NET assumes no knowledge and no control over the pipeline inasmuch as the same code is executed both for training and prediction. [5] This is in general good practice because it simplifies the process of training-inference skew debugging [27]. Nevertheless, we found that such approach is sub-optimal from a performance perspective. For instance, transfer learning, A/B testing and model personalization are getting popular. Such trends produce models DAGs with high chance of overlapping structure and similar parameters, but these similarities cannot be recognized nor exploited using a black-box approach. Furthermore, it is common practice for in-memory data-intensive systems to pipeline operators in order to minimize memory accesses for memory-intensive workloads, and to vectorize compute-intensive operators in order to minimize the number of instructions per data item [16, 28]. ML.NET's operator-at-a-time model [28] is sub-optimal because computation is organized around logical operators, ignoring how those operators behave together: in the example of the sentiment analysis pipeline of Figure 1, logistic regression is commutative and associative (e.g., dot product between vectors) and can be pipelined with Char and WordNgram, eliminating the need for the Concat operation and the related buffers for intermediate results. Note that this optimization is applicable only at prediction-time whereas at training-time logistic regression runs the selected optimization algorithm. We refer readers to [19] for further limitations arising when serving models for prediction using the black-box approach.

3.3 White Box Model Serving

As we have seen so far, black-box approaches disallow any optimization and sharing of resources among models. Such limitations are overcome by the *white box approach* embraced by systems such as TVM [14] and PRETZEL [19]. Figure 2(c) sketches white-box prediction serving. Models are registered to a Runtime that considers them not as mere executable code but as DAGs of operators. Applications can request predictions by directly including the Runtime in their logic (similarly to how SQLite databases [11] can be integrated into applications), or by submitting a REST request to a cloud-hosted Runtime. The white box approach enables the Runtime to apply optimizations over the models such as operator reordering to improve latency or operator and sub-graph sharing to improve memory consumption and computation reuse (through caching). Thorough scheduling of pipelines' components can be managed within the Runtime, which controls the whole workload so that optimal allocation decisions can be made for running machines to high utilization and avoid many of the aforementioned overheads. In general, we have identified the following optimization opportunities for white-box model serving.

End-to-end Optimizations: The operationalization of models for prediction should focus on computation units making optimal decisions on how data are processed and results are computed, to keep low latency and graceful degradation of performance with increasing load. Such computation units should: (1) avoid memory allocation on the data path; (2) avoid creating separate routines per operator when possible, which are sensitive to branch mis-prediction and poor data locality [21]; and (3) avoid reflection and JIT compilation at prediction time. Optimal computation units can be compiled Ahead-Of-Time (AOT) since pipeline and operator characteristics are known upfront, and often statistics from training are available. The only decision to make at runtime is where to allocate the computation units based on available resources and constraints.

Multi-model Optimizations: To take full advantage of the fact that pipelines often use similar operators and parameters, shareable components have to be uniquely stored in memory and reused as much as possible to achieve optimal memory usage. Similarly, execution units should be shared at runtime and resources should be properly pooled and managed, so that multiple prediction requests can be evaluated concurrently. Partial results, for example outputs of featurization steps, can be saved and re-used among multiple similar pipelines.

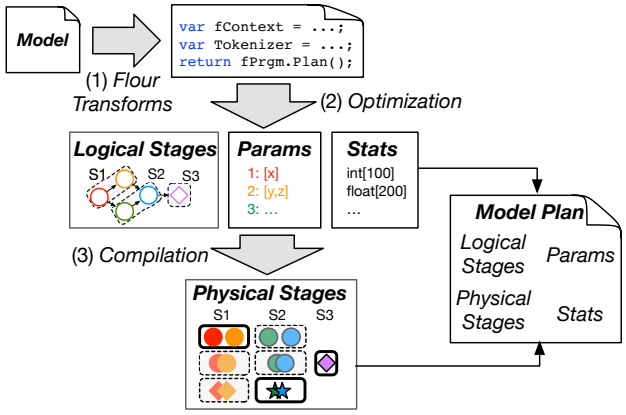
Out of these guidelines, the next Section describes a prototype runtime for ML.NET enabling white-box model serving.

[5] Indeed, already trained operators will bypass the execution of the learning algorithm and directly apply the previously learned parameters.

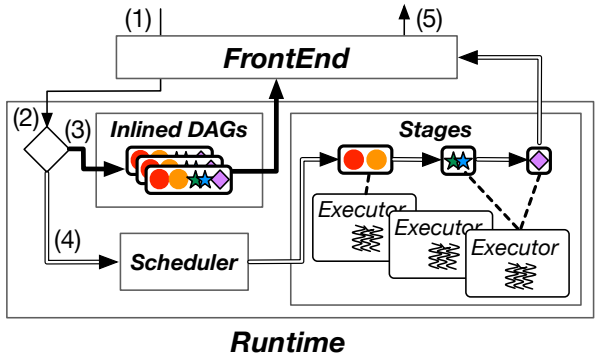
4 A Database Runtime for Model Serving

Following the guidelines of white-box model serving, we implemented PRETZEL [19, 24], a new runtime for ML.NET specifically tailored for high-performance prediction serving. PRETZEL views models as database queries and employs database techniques to optimize DAGs and to improve end-to-end performance. The problem of optimizing co-located pipelines is casted as a multi-query optimization and techniques such as view materialization are employed to speed up pipeline execution. Memory and CPU resources are shared in the form of vector and thread pools, such that overheads for instantiating memory and threads are paid only upfront at initialization time. PRETZEL is organized in 6 main components. A *data-flow-style language integrated API* called Flour with related *compiler* and *optimizer* called Oven are used in concert to convert ML.NET pipelines into *model plans*. An Object Store saves and shares parameters among plans. A Runtime manages compiled plans and their execution, while a Scheduler manages the dynamic decisions on how to schedule plans based on machine workload. Finally, a FrontEnd is used to submit prediction requests to the system.

In PRETZEL, deployment and serving of model pipelines follow a two-phase process as illustrated in Figure 3(a) and 3(b). During the *off-line phase*, pre-trained ML.NET pipelines are translated into Flour transformations. Oven optimizer re-arranges and fuses transformations into model plans composed of parameterized logical units called *stages*. Each logical stage is then AOT-compiled into physical computation units. Logical and physical stages together with model parameters and training statistics form a model plan. Model plans are registered for prediction serving in the Runtime where physical stages and parameters are shared among pipelines with similar model plans. In the *on-line phase*, when an inference request for a registered model plan is received, physical stages are parameterized dynamically with the proper values maintained in the Object Store. The Scheduler is in charge of binding physical stages to shared execution units.



(a) Model optimization and compilation in PRETZEL: (1) A model is translated into a Flour program. (2) Oven Optimizer generates a DAG of logical stages from the program. Additionally, parameters and statistics are extracted. (3) A DAG of physical stages is generated by the Oven Compiler using logical stages, parameters, and statistics. A model plan is the union of all the elements.



(b) (1) When a prediction request is issued, (2) the Runtime determines whether to serve the prediction using (3) the request/response engine or (4) the batch engine. In the latter case, the Scheduler takes care of properly allocating stages over the Executors running concurrently on CPU cores. (5) The FrontEnd returns the result to the Client once all stages are complete.

Figure 3: How PRETZEL system works in two phases: (a) Offline and (b) Online.

PRETZEL compiles a model pipeline into an optimized, executable plan following 3 steps:

1. **Model conversion:** Flour is an an intermediate representation, which makes PRETZEL's optimizations applicable to various ML frameworks (although the currently implementation is for ML.NET). Once a pipeline is ported into Flour, it can be optimized and compiled into a model plan. Flour provides a language-integrated API similar to KeystoneML [25] or LINQ [20], where sequences of transformations

are chained into DAGs and lazily compiled for execution.

2. **Optimization:** Oven's rule-based Optimizer rewrites a model pipeline represented in Flour into a graph of logical stages. Initially, the Optimizer applies rules for schema propagation, schema validation and graph validation. The rules check whether the input schema of each data transformation is valid (e.g., WordNgram in Figure 1 takes a text as input) and the structure of the graph is well-formed (e.g., only one final predictor model at the end). The next rules are used to build stages by traversing the entire graph to find *pipeline-breaking* transformations that require all inputs to be fully materialized (e.g., normalizer): all transformations up to that point are then grouped into a stage. By leveraging stage graphs similar to Spark [26], PRETZEL can run computations more efficiently than the operator-at-a-time strategy of ML.NET. The stage graph is then optimized by recursively applying rules such as (1) removing unnecessary branches (similar to common sub-expression elimination); (2) merging stages containing identical transformations; (3) inlining stages that contain only one transformation.

3. **Compilation:** After building the stage graph, a *Model Plan Compiler* (MPC) translates the graphs into physical stages, which are AOT-compiled, parameterized, and result in lock-free computation unit. Logical and physical stages have a 1-to-n mapping, and MPC selects the most efficient physical implementation given the logical stage's parameters (e.g., the maximum length of n-grams) and statistics (e.g., whether the vector is dense or sparse). During the translation process, MPC saves the additional parameters required for running stage code (e.g., a dictionary consisting of frequency of n-grams) into the ObjectStore in order to share them with other stages with the same parameters. Finally, model plans are registered into the Runtime. Model plans consist of mappings between logical representations, physical implementations and the associated parameters. Upon registration, physical stages composing a plan are loaded into a system catalog. When a prediction request is submitted to the system, the AOT-compiled physical stages are initialized with the parameters from the mapping in the model plan, which allows PRETZEL Runtime to share the same physical implementation among multiple pipelines.

5 Conclusion

Inspired by the growth of ML applications and ML-as-a-service platforms, this paper identifies three strategies for operationalizing trained models: container-based, direct import into applications, and the white-box approach. Using ML.NET as use case, we listed a set of limitations on how existing systems fall short in key requirements for ML prediction-serving, disregarding the optimization of model execution in favor of ease of deployment. Finally, we describe how the problem of serving predictions can be casted as a database problem, whereby end-to-end and multi-query optimization strategies are applied to ML pipelines.

We recognize that much work remains to be done for achieving a seamless and efficient integration of ML models with applications and development processes. While we believe that ML.NET and PRETZEL are a step in the right direction, equivalents in data science for common tools and techniques in software development (e.g., unit/integration test, build server, code review, versioning, backward compatibility, and lifecycle management) are not defined yet. We encourage the community to engage in the work towards closing those gaps.

References

[1] AI Platform for Windows Developers. `https://blogs.windows.com/buildingapps/2018/03/07/ai-platform-windows-developers`

[2] Apache Spark MLlib. `https://spark.apache.org/mllib`

[3] Clipper. `http://clipper.ai`

[4] Docker. https://docker.com

[5] The future of AI marketing: human ingenuity amplified. https://advertise.bingads.microsoft.com/en-us/insights/g/artificial-intelligence-for-marketing

[6] H2O.ai. https://www.h2o.ai

[7] Microsoft Looks To Patent AI For Detecting Video Game Cheaters. https://www.cbinsights.com/research/microsoft-xbox-machine-learning-cheat-detection-gaming-patent

[8] Microsoft Machine Learning Server. https://docs.microsoft.com/en-us/machine-learning-server

[9] ML.Net. https://dot.net/ml

[10] MXNet Model Server (MMS). https://github.com/awslabs/mxnet-model-server

[11] SQLite. https://www.sqlite.org

[12] TensorFlow Serving. https://www.tensorflow.org/serving

[13] Z. Ahmed, and et al. Machine Learning for Applications, not Containers. *Under Submission*, 2018.

[14] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. *OSDI*, 2018

[15] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A Low-Latency Online Prediction Serving System. *NSDI*, 2017.

[16] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Cetintemel, and S. Zdonik. An Architecture for Compiling UDF-centric Workflows. *VLDB*, 2015.

[17] G. Graefe. Volcano: An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.*, February 1994.

[18] A. Y. Halevy Answering Queries Using Views: A Survey. *VLDB Journal*, December 2001.

[19] Y. Lee, A. Scolari, B. -G. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi. PRETZEL: Opening the black box of Machine Learning Prediction Serving. *OSDI*, 2018.

[20] E. Meijer, B. Beckman, and G. Bierman LINQ: Reconciling Object, Relations and XML in the .NET Framework. *SIGMOD*, 2006.

[21] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *VLDB Endowment*, June 2011.

[22] C. Olston, F. Li, J. Harmsen, J. Soyke, K. Gorovoy, L. Lao, N. Fiedel, S. Ramesh, and V. Rajashekhar. TensorFlow-Serving: Flexible, High-Performance ML Serving. *Workshop on ML Systems at NIPS*, 2017.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and É. Duchesnay. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.*, 2011.

[24] A. Scolari, Y. Lee, M. Weimer, and M. Interlandi. Towards Accelerating Generic Machine Learning Prediction Pipelines. *IEEE ICCD*, 2017.

[25] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. *ICDE*, 2017

[26] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. *NSDI*, 2012

[27] M. Zinkevich. Rules of Machine Learning: Best Practices for ML Engineering. https://developers.google.com/machine-learning/rules-of-ml

[28] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman. MonetDB/X100 - A DBMS in the CPU Cache. *IEEE Data Eng. Bull.*, 2005.

Report from the Workshop on Common Model Infrastructure, ACM KDD 2018

Chaitanya Baru
UC San Diego
baru@sdsc.edu

Abstract

Here we provide a summary of the Workshop on Common Model Infrastructure which was organized at KDD 2018 to focus on model lifecycle management and the tools and infrastructure needed to support development, testing, discovery, sharing, reuse, and reproducibility of machine learning, data mining, and data analytics models.

This workshop was organized by Chaitan Baru, UC San Diego; Amol Deshpande, University of Maryland; Robert Grossman, University of Chicago; Bill Howe, University of Washington; Jun Huan, Baidu; Vandana Janeja, University of Maryland Baltimore County; and, Arun Kumar, UC San Diego. The workshop organizers would like to acknowledge the strong support and encouragement provided by KDD Workshop Chairs, Jun Huan and Nitesh Chawla.

1 Motivation

The *Workshop on Common Model Infrastructure* was organized at KDD 2018 to focus on *model life-cycle management* and the tools and infrastructure needed to support development, testing, discovery, sharing, reuse, and reproducibility of machine learning, data mining, and data analytics models.

The workshop provided a forum for discussion of emerging issues, challenges, and solutions in this area, focusing on the principles, services and infrastructure needed to help data scientists of every ilk—from scientific researchers to industry analysts, and other practitioners—share data analytics models, reproduce analysis results, support transfer learning, and reuse pre-constructed models.

The continuing, rapid accumulation of large amounts of data and increasing use of machine learning, data mining, and data analytics techniques across a broad range of applications creates the need for systematic approaches for managing the increasingly complex of the modeling processes, given the large numbers of data-driven models being generated. However, current modeling practices are rather ad hoc, often depending upon the experience and expertise of individual data scientists and types of pre-processing used, which may be specific to domains. Different application domains/disciplines may use similar models and modeling tools, yet, sharing is limited; modeling results often have poor reproducibility; information on when/how a model works, and when it may fail, is oftentimes not clearly recorded; model provenance and the original intent behind the knowledge discovery process is not well-recorded; and, furthermore, many predictive analytics algorithms are not transparent to end-users. The CMI workshop was designed as a forum to discuss these and related issues.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2 Workshop Format

CMI 2018 was a half-day workshop held on August 20th at KDD 2018, London, UK (`https://cmi2018.sdsc.edu/`). The workshop included three invited talks and two long lightning talks and four short lightning talks, selected from a set of submitted papers. Two of invited talks were from industry and one was from academia; four of the six lightning talks were from industry and two were from academia. The workshop ended with a group discussion, with all the speakers serving as an ad hoc panel. There were about 60 attendees at the workshop at any given time.

3 Invited Talks

The invited talk, *"What is the code version control equivalent for ML and data science workflows?"* by Clemens Mewald, Product Lead of TensorFlow Extended (TFX) from the Research & Machine Intelligence group at Google discussed how Google AI is addressing the challenges of managing the new paradigm of software development workflows being introduced in machine learning projects, which require code, data, and all derived artifacts (including models) to be indexed, updated, and shared. Traditionally, software developers primarily dealt with code that was compiled or interpreted, usually with deterministic behavior. With ML, products now rely on behaviors and patterns, often expressed as predictions, that are a function of code and evolving data plus models, leading to dynamic behaviors. The talk discussed the specific challenges faced by researchers and software engineers, and how systems like Google's TensorFlow Extended and TensorFlow Hub are addressing these issues.

In the invited talk, *"Why it is Important to Understand the Differences Between Deploying Analytic Models and Developing Analytic Models?"*, Professor Bob Grossman from the University of Chicago highlighted the two cultures in data science and analytics — one that is focused on *developing* analytic models, and the other that focuses on *deploying* analytic models into operational systems. The talk described the typical lifecycle of analytic models and provided an overview of some of the approaches that have been developed for managing and deploying analytic models and workflows, which included an overview of languages for analytic models, such as PMML, and for analytic workflows, such as PFA. It was posited that there is the emerging discipline of AnalyticOps that has borrowed some of the techniques of DevOps.

The third invited talk on, *"Beaker: A collaborative platform for rapid and reproducible research"* was delivered by Marc Milstone, leader of the Aristo Engineering team at the Allen Institute for Artificial Intelligence. The talk described Beaker, a robust experimentation platform for computational researchers that helps streamline the reproducible training, analysis and dissemination of machine learning results. Beaker was designed collaboratively with researchers at the Allen Institute for AI, focusing on ease-of-use for AI research workflows. The goal of the Beaker system is to provide a tool to help researchers deal with the details of organizing results and scaling experiments, so that they may spend more time on discovery and validation of new ideas that advance their field, rather than on the various ancillary details of running computational experiments. Beaker utilizes systems like Docker and Kubernetes to help reduce the "cognitive overhead" of infrastructure.

4 Lightning Talks

Papers for all the talks are available here `https://cmi2018.sdsc.edu/#accepted`.

One of the "long talks" was on *"Context: The Missing Piece in the Machine Learning Lifecycle"*, presented by Rolando Garcia, UC Berkeley, on behalf of co-authors, Vikram Sreekanti, Neeraja Yadwadkar, Daniel Crankshaw, Joseph E. Gonzalez, and Joseph M. Hellerstein. The ML lifecycle is characterized as consisting of three phases—Pipeline Development, Training, and Inferencing—and they identify that the crucial missing

piece within and across every phase is context, viz., "all the information surrounding the use of data in an organization." In current practice, the transitions between stages and teams are usually ad hoc and unstructured, which means that no single individual or system has a global, end-to-end view of the ML Lifecycle, leading to issues like irreproducibility, over-fitting, and missed opportunities for improved productivity, performance, and robustness.

The second "long talk" on *"Fighting Redundancy and Model Decay with Embeddings"* was presented by Dan Shiebler from the Twitter Cortex group, on behalf of co-authors Luca Belli, Jay Baxter, Hanchen Xiong, Abhishek Tayal. Twitter faces the issue that topics of interest are perpetually changing and evolving on the Internet at fairly rapid pace. Thus, models must keep up with the pace of change in the actual contents of the data streams. This talk detailed the commoditized tools and pipelines that Twitter has developed, and is developing, to regularly generate high quality, up-to-date embeddings and share them broadly across the company. Every day, hundreds of millions of new Tweets containing over 40 languages of ever-shifting vernacular flow through Twitter. Models that attempt to extract insight from this firehose of information must face the torrential covariate shift that is endemic to the Twitter platform. While regularly-retrained algorithms can maintain performance in the face of this shift, fixed model features that fail to represent new trends and tokens can quickly become stale, resulting in performance degradation. To mitigate this problem, Twitter employs learned features, or embedding models, that can efficiently represent the most relevant aspects of a data distribution. Sharing these embedding models across teams can also reduce redundancy and multiplicatively increase cross-team modeling productivity.

The remaining four presentation were "short talks". First was a talk on *"Recommender systems for machine learning pipelines"*, presented by Raymond Wright on behalf of the co-authors Insorge Silva and Ilknur Kaynar-Kabul, all from the SAS Institute Inc, about enhancing the existing SAS Model Studio interactive workbench to recommend one or more pipelines to use for a new, unseen dataset based on metafeatures derived from the dataset, since novice users often struggle with how to build or select a pipeline. The pipelines typically include feature generation, feature selection, model building, ensembling, and selection of a champion model, and represents a reproducible machine learning process. Once a general-purpose recommender model has been built using data from a variety of domains, the hope is that one can then generate specialized recommender rules for domains such as insurance, credit risk, etc. The work is inspired by recent work in automated machine learning, such as AUTO-SKLEARN, with the variation that here they aim to select from among a finite set of existing pipelines that have been found generally useful among users of the workbench.

Next, was a paper on *"Building a Reproducible Machine Learning Pipeline"* by Peter Sugimara and Florian Hartl from Tala Co, a startup company. The objective here is to automate and abstract the process from idea to deploying a machine learning model in production—which includes many steps—from machine learning practitioners, in order to improve modeling speed and quality, and realize key benefits like reproducibility of results. The presentation described the framework, which is comprised of four main components—data, feature, scoring, and evaluation layers—which are themselves comprised of well—defined transformations. This enables exact replication of models, and the reuse of transformations across different models. As a result, the platform can dramatically increase the speed of both offline and online experimentation while also ensuring model reproducibility.

The third presentation was on *"Knowledge Aggregation via Epsilon Model Spaces"* by Neel Guha who presented work done while he was a student at Stanford University. This work tackles scenarios where the machine learning task is divided over multiple agents, where each agent learns a different task and/or learns from a different dataset, which is a situation that can occur in many practical applications. The Epsilon Model Spaces (EMS), framework that was presented learns a global model by aggregating local learnings performed by each agent. This approach forgoes sharing of data between agents, makes no assumptions on the distribution of data across agents, and requires minimal communication between agents. Experiments were performed on the MNIST dataset, providing a validation of the methods used for both shallow and deep aggregate models. EMS is among the first to lay out a general methodology for "combining" distinct models. The EMS approach could help in future by allowing the development of "libraries" of models that could act as building blocks when

learning new models.

The fourth talk was by Microsoft on *"An Open Platform for Model Discoverability"*, presented by Yasin Hajizadeh, on behalf of co-authors Vani Mandava and Amit Arora. Recognizing that the need to rapidly experiment, share and enable collaboration at scale between a fast-growing community of machine learning experts and domain experts has created an acute need for a platform for management, discoverability, and sharing of machine learning models, Microsoft has developed a novel cloud-based AI Gallery that enables data scientists and domain experts to seamlessly share and collaborate on machine learning solutions. While there are siloed systems that address this issue, there is no single comprehensive platform that addresses all these challenges. The Azure AI Gallery, `https://gallery.azure.ai/`models, provides a gallery of models, projects and solution templates for data scientists, application developers and model managers that works with local and cloud environments. The gallery currently has several ONNX models from `https://onnx.ai` and is expected to grow as more models are contributed.

5 Summary

The first community workshop on *"Common Model Infrastructure"*—informally referred to as the *ModelCommons*—held at KDD 2018, London, UK was, indeed, a success. There was significant industry presence among the speakers at the workshop as well as in the audience, indicating a strong industry interest in the topic, as well as the relevance of this topic area to practical applications. The workshop concluded with a panel of all presenters engaging in some question and answers with the audience, but also in a group discussion. First, it was felt that this is an important topic that should be continued in the future, including with a workshop at the next KDD Conference in Anchorage, Alaska in 2019. Indeed, the next meeting on this topic is sponsored by Baidu and will occur at the NIPS Expo on December 2, in conjunction with the NIPS 2018 Conference in Montreal, Canada . There was also a discussion of the appropriate venue for publication of full-length papers from this workshop. One idea was to request for a special issue of the new ACM Transactions on Data Science.

There was an important discussion on whether "Common Model Infrastructure" was sufficiently descriptive of the range of issues that fall under the "ML lifecycle". There was agreement that a more descriptive term is needed that indicates coverage of the broad set of issues—some of which may be research while others may be infrastructural in nature—but the group did not discuss what the new description should be. Future workshop may well adopt a different name for this set of topics.

Call for Papers

35th IEEE International Conference on Data Engineering

8-12 April 2019, Macau SAR, China

The annual IEEE International Conference on Data Engineering (ICDE) addresses research issues in designing, building, managing and evaluating advanced data-intensive systems and applications. It is a leading forum for researchers, practitioners, developers, and users to explore cutting-edge ideas and to exchange techniques, tools, and experiences. The 35th ICDE will take place at Macau, China, a beautiful seaside city where the old eastern and western cultures as well as the modern civilization are well integrated.

Topics of Interest

We encourage submissions of high quality original research contributions in the following areas. We also welcome any original contributions that may cross the boundaries among areas or point in other novel directions of interest to the database research community:

- Benchmarking, Performance Modelling, and Tuning
- Data Integration, Metadata Management, and Interoperability
- Data Mining and Knowledge Discovery
- Data Models, Semantics, Query languages
- Data Provenance, cleaning, curation
- Data Science
- Data Stream Systems and Sensor Networks
- Data Visualization and Interactive Data Exploration
- Database Privacy, Security, and Trust
- Distributed, Parallel and P2P Data Management
- Graphs, RDF, Web Data and Social Networks
- Database technology for machine learning
- Modern Hardware and In-Memory Database Systems
- Query Processing, Indexing, and Optimization
- Search and Information extraction
- Strings, Texts, and Keyword Search
- Temporal, Spatial, Mobile and Multimedia
- Uncertain, Probabilistic Databases
- Workflows, Scientific Data Management

Important Dates

For the first time in ICDE, ICDE2019 will have two rounds' submissions. All deadlines in the following are 11:59PM US PDT.

First Round:

Abstract submission due: May 25, 2018
Submission due: June 1, 2018
Notification to authors (Accept/Revise/Reject): August 10, 2018
Revisions due: September 7, 2018
Notification to authors (Accept/Reject): September 28, 2018
Camera-ready copy due: October 19, 2018

Second Round:

Abstract submission due: September 28, 2018
Submission due: October 5, 2018
Notification to authors (Accept/Revise/Reject): December 14th, 2018
Revisions due: January 11th, 2019
Notification to authors (Accept/Reject): February 1, 2019
Camera-ready copy due: February 22, 2019

General Co-Chairs
Christian S. Jensen, Aalborg University
Lionel M. Ni, University of Macau
Tamer Özsu, University of Waterloo

PC Co-Chairs
Wenfei Fan, University of Edinburgh
Xuemin Lin, University of New South Wales
Divesh Srivastava, AT&T Labs Research

For more details: http://conferences.cis.umac.mo/icde2019/

Data Engineering

It's FREE to join!

TCDE

tab.computer.org/tcde/

The Technical Committee on Data Engineering (TCDE) of the IEEE Computer Society is concerned with the role of data in the design, development, management and utilization of information systems.

- Data Management Systems and Modern Hardware/Software Platforms
- Data Models, Data Integration, Semantics and Data Quality
- Spatial, Temporal, Graph, Scientific, Statistical and Multimedia Databases
- Data Mining, Data Warehousing, and OLAP
- Big Data, Streams and Clouds
- Information Management, Distribution, Mobility, and the WWW
- Data Security, Privacy and Trust
- Performance, Experiments, and Analysis of Data Systems

The TCDE sponsors the International Conference on Data Engineering (ICDE). It publishes a quarterly newsletter, the Data Engineering Bulletin. If you are a member of the IEEE Computer Society, you may join the TCDE and receive copies of the Data Engineering Bulletin without cost. There are approximately 1000 members of the TCDE.

Join TCDE via Online or Fax

ONLINE: Follow the instructions on this page:

www.computer.org/portal/web/tandc/joinatc

FAX: Complete your details and fax this form to **+61-7-3365 3248**

Name _____

IEEE Member # _____

Mailing Address _____

_____

Country _____

Email _____

Phone _____

TCDE Mailing List

TCDE will occasionally email announcements, and other opportunities available for members. This mailing list will be used only for this purpose.

Membership Questions?

Xiaoyong Du
Key Laboratory of Data Engineering and Knowledge Engineering
Renmin University of China
Beijing 100872, China
duyong@ruc.edu.cn

TCDE Chair

Xiaofang Zhou
School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane, QLD 4072, Australia
zxf@uq.edu.au

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398