Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities

Doris Xin*
University of California, Berkeley
dorx@berkeley.edu

Hui Miao
Google
huimiao@google.com

Aditya Parameswaran
University of California, Berkeley
adityagp@berkeley.edu

Neoklis Polyzotis
Google
npolyzotis@google.com

ABSTRACT

Machine learning (ML) is now commonplace, powering data-driven applications in various organizations. Unlike the traditional perception of ML in research, ML production pipelines are complex, with many interlocking analytical components beyond training, whose sub-parts are often run multiple times on overlapping subsets of data. However, there is a lack of quantitative evidence regarding the lifespan, architecture, frequency, and complexity of these pipelines to understand how data management research can be used to make them more efficient, effective, robust, and reproducible. To that end, we analyze the provenance graphs of 3000 production ML pipelines at Google, comprising over 450,000 models trained, spanning a period of over four months, in an effort to understand the complexity and challenges underlying production ML. Our analysis reveals the characteristics, components, and topologies of typical industry-strength ML pipelines at various granularities. Along the way, we introduce a specialized data model for representing and reasoning about repeatedly run components in these ML pipelines, which we call model graphlets. We identify several rich opportunities for optimization, leveraging traditional data management ideas. We show how targeting even one of these opportunities, i.e., identifying and pruning wasted computation that does not translate to model deployment, can reduce wasted computation cost by 50% without compromising the model deployment cadence.

CCS CONCEPTS

• **Information systems** → **Data management systems**; • **Computing methodologies** → **Machine learning**.

KEYWORDS

machine learning pipelines, data management

ACM Reference Format:
Doris Xin, Hui Miao, Aditya Parameswaran, and Neoklis Polyzotis. 2021. Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3457566

*Work done while at Google.

SIGMOD '21, June 20–25, 2021, Virtual Event, China.

ACM ISBN 978-1-4503-8343-1/21/06.
https://doi.org/10.1145/3448016.3457566

1 INTRODUCTION

ML (Machine Learning) is now ubiquitous in data-driven organizations, consuming an increasing share of compute resources. A commonly held misconception is that ML is a one-step training procedure that accepts data as input and generates a model as output. Many research efforts, both within the DB community and in other communities, focus on this simplified view and aim to improve the effectiveness of ML training, e.g., by generating more powerful models through better training algorithms, or by reducing its resource footprint through various software and hardware optimizations. This view is reinforced by various leaderboard-style competitions popular among practitioners (e.g., Kaggle), and academics (e.g., ML benchmarks and KDD competitions).

At the same time, there is evidence from practitioners [16, 19, 48] that ML deployments in production are significantly more complicated. Specifically, ML in production involves pipelines with many interlocking steps, only one of which is training. This has spurred on the development of many end-to-end ML systems (e.g., TFX [16], MLFlow [52], Microsoft Azure ML [8], AWS Sagemaker [34]) and open-source ML libraries (e.g., MLlib [6], MetaFlow [2], and Scikit-Learn [7]), all of which provide native support for data pre-processing, data validation, model validation, and model deployment, in addition to model training, all within a single environment.

As an example, TFX [16] includes pipeline steps that perform different flavors of data analysis and transformation, or of data- and model-validation, both before and after the training step. The topology of the corresponding graph can be complicated. For instance, model chaining (where a model is used to generate data for another model) is becoming increasingly common, introducing model-to-model dependencies in the same pipeline. And there is the step of deploying a model after training, in a scalable serving infrastructure. Moreover, production ML pipelines often work continuously, with periodic retraining and deployment as fresh data becomes available. Overall, the steps across these pipelines interact in complex ways, and their compound effects might be hard to predict or debug, necessitating the management of provenance across them. Provenance management is one of the key value adds of end-to-end ML platforms like TFX [15], MLFlow [52] or MetaFlow [2]. While there is anecdotal evidence for end-to-end concerns beyond training, little is known about production ML deployments and the challenges they surface in terms of data management:

- **Coarse-grained pipeline characteristics.** What do production ML pipelines look like in a large data-driven organization? What types of models are used? What types of feature engineering procedures are used, and how complex are they from a data processing standpoint? What is the lifespan of typical pipelines?

- **Fine-grained pipeline characteristics.** How much overlap exist between executions of a given pipeline? For portions of a pipeline that are executed repeatedly to derive models, how does the data distribution change? How often are they executed, and how often are the resulting models deployed?
- **Opportunities.** Are there uniform ways to represent and reason about these pipelines? Any opportunities to make these pipelines more efficient, e.g., by leveraging sharing of computation, pruning redundant computation, making more efficient use of system resources, and leveraging incremental view maintenance?

Answering these questions can improve our understanding of production ML. In turn, this increased awareness can help the DB research community move beyond improved training efficiency to more effectively supporting end-to-end production ML pipelines. This involves addressing new incarnations of familiar challenges—from efficient data preparation, to optimized query plans, to dealing with streaming data, to sharing of computation, to materialization and reuse, to provenance for reproducibility and debugging.

In this paper, we take a first step in this direction by *analyzing a large corpus of 3000 production ML pipelines at Google, comprising over 450,000 trained models, over a period of four months.* To the best of our knowledge, no similar corpus has ever been analyzed in prior work. This unique corpus of thousands of TensorFlow Extended (TFX) pipelines, with hundreds of thousands of generated models, spans different modalities (tabular data, video and text embeddings, personalization), tasks (regression, classification), and environments (production, development). Our analysis reveals a number of interesting insights, including the fact that: *(a)* training accounts for only 20% of the total computation time, despite ∼ 60% of models being deep neural nets (DNNs); *(b)* the rest (40%) of the pipelines train traditional, non-DNN ML models, showing the value of simpler model architectures in production, but also the need to manage a diverse set of model types within the same organization; *(c)* the input data used for consecutive model updates have large overlaps but also significant differences in data distribution, underlining the need to cope with data and concept drift, *(d)* models in each pipeline are updated 7 times per day on average, with a substantial fraction (1.12%) of pipelines updating models over 100 times a day(!), giving rise to potential instability that requires special care, *(e)* only one in four model retrainings results in model deployment, with the three undeployed model updates representing wasted computation, discussed more later. While our results stem from analyzing production ML pipelines at Google, due to the commonalities between TFX and other end-to-end ML platforms, as well as the adoption of TFX in other large organizations [30], we expect our findings to generalize to other production use cases and thus be of general interest. Additionally, we hope our approach to analyzing this complex corpus can serve as inspiration for studying other corpora of ML systems history.

Specifically, we make the following contributions:

Coarse-grained analysis of pipeline lifespan, components, architecture, and complexity (Section 3). We provide the first-ever study of 3000 ML production pipelines captured over a four-month period to understand the underlying data management challenges. This analysis surfaces coarse-grained characteristics about these pipelines, such as their lifespan in the end-to-end training of several models, proportion of resources devoted to data analytics (beyond training), and the features, feature transformations, and model architectures in the pipelines.

Model graphlet abstraction and fine-grained analysis of frequency, failure, and overlap (Section 4). We proceed to analyze finer-grained properties of these pipelines through provenance analysis. Many previous studies analyzed provenance graphs in data workflows, but the complexity and unique characteristics of production ML necessitate a new approach. We introduce the notion of *model graphlets*, wherein the provenance graph is decomposed into sub-graphs to capture the end-to-end execution of the pipeline including several instances of model training. One can view model graphlets as a ML-oriented application of the more general concept of provenance segmentation [9]. We then characterize these graphlets in terms of their lifespan, complexity, overlap, failure points, and their connection to model deployment.

Eliminating 50% of the wasted computation that does not lead to model deployment (Section 5). As an immediate consequence of (and evidence for the value of) our analysis, we identify a significant optimization opportunity involving preemptively skipping pipeline executions. Overall, there are many wasteful graphlets that neither deploy models to serve applications nor help warm-start subsequent model training. We show that such graphlets have significant resource costs. Moreover, we show that the root causes for the wasteful graphlets are varied; hence, it is difficult to come up with simple heuristic strategies to identify them. Instead, we leverage the dataset at hand and develop an ML-based solution—we train a model that uses the current state of the pipeline to predict whether the graphlet run will result in a deployed model, without even running it. The model achieves high accuracy and allows us to save up to 50% of wasted computation, without compromising graphlet runs that deploy models. Beyond this direct benefit, this optimization is indicative of the opportunities to optimize ML deployments through a holistic analysis of the ML provenance graph, in addition to localized optimizations of individual steps (e.g., reducing the footprint of the trainer).

Related Work. We briefly cover previous studies related to tracking, analyzing, and optimizing ML pipelines in production.

End-to-end ML frameworks. Several end-to-end ML frameworks, both commercial and open-source, have been developed recently to address increasingly sophisticated ML use cases. While the programming interface and the runtime may vary across these frameworks, they all aim to support common operators in ML pipelines, including data ingestion and pre-processing, data validation, modeling training and validation, and model deployment. Systems such as Microsoft Azure ML [8], TensorFlow Extended (TFX) [15] and Kubeflow [1], and open-source libraries such as MLlib (native support in Databricks) [6], MetaFlow (in production at Netflix) [2], and Scikit-Learn (native support in all commercial systems) [7], have explicit declarative programming abstractions for composing ML pipelines from canned or custom operators, whereas AWS Sagemaker [34] captures the notion of an ML pipeline indirectly through integration with libraries with pipeline constructs. In terms of the runtime, all commercial cloud-based systems offer capabilities to provision for and schedule ML pipeline executions. In addition, TFX and Azure

ML also offer automated support for continuous model update and deployment of models.

Provenance management and analysis. Provenance for complex systems has been studied for relational databases [20], scientific workflows systems [22], dataflow systems [12, 27], data lakes [23, 25], and ML systems [35, 49, 52]. Previous work has even led to the standardization of provenance representations for workflows in the form of graphs [26, 38, 39]. Other research has proposed various ways to explore and analyze such provenance graphs, e.g., visualization [14], reachability query support [13], support for user-defined views [17], segmentation and summarization [9, 10, 36]. Our work introduces a framework to segment ML provenance graphs and demonstrates how this segmentation leads to further analysis and optimizations for ML pipelines.

Metadata tracking for the ML Lifecycle. In modern end-to-end ML systems, e.g., TFX [16], MLFlow [52], Kubeflow [1], and MetaFlow [2], data science development tools, e.g., ModelDB [49], noWorkflow [42], and Pachyderm [4], as well as model development practices in the industry, e.g., [45, 47], there is an effort towards tracking metadata to facilitate workflow orchestration and aid model reproducibility over the ML project lifecycle. These metadata tracking solutions vary in terms of ingestion method (user input vs. transparent), data model (relational vs. graph), and scope of the metadata (training vs. end-to-end). Our work here is based on a specific framework (MLMD [3] which is part of TFX [5, 15]) which automatically records the end-to-end provenance using a graph-based model. We build on this framework and introduce a method to segment large provenance graphs into smaller model-centric graphs. However, our analysis and this decomposition is orthogonal from the specific representation of complete provenance, and the same methodology could apply to other systems. Moreover, our final contribution with respect to pruning redundant graphlets has not been explored in previous work to the best of our knowledge.

Understanding ML workflows. While our work is, to the best of our knowledge, the first large-scale study of production ML pipelines, a few prior papers have conducted empirical studies on ML and data science (DS) workflows. Lee et al. [33] aim to understand how ML developers iterate on models by studying ML workflows generated by novice and intermediate ML developers on Kaggle-style tasks with static datasets and no model deployment. Our study is fundamentally different in the nature of the ML tasks studied—with an emphasis on production ML and pipelines that are run over a long period. Some works in the HCI community study ML/DS workflows by interviewing ML developers and data scientists [11, 28, 53]. Another related area of work is anecdotal reports and retrospectives by industry ML practitioners [44, 46, 48] that shed light on real-world ML practices and challenges. We complement this line of work with quantitative insights from a corpus of production ML pipelines.

2 PRELIMINARIES

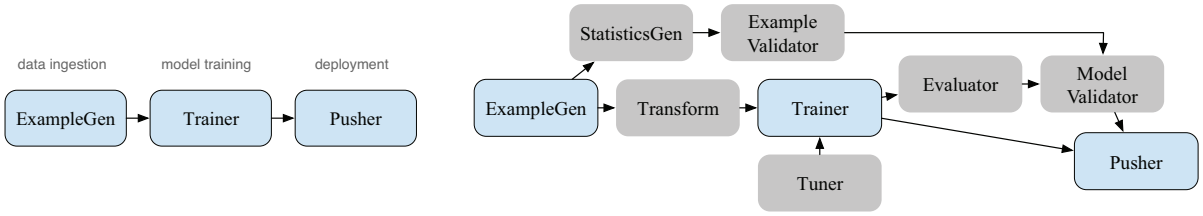
Our corpus comprises TensorFlow Extended (TFX) [15] pipelines. TFX is an end-to-end platform for production ML used by product teams across Google. The platform has been recently open-sourced and has been adopted by major organizations outside Google as well [30]. TFX provides a set of operators that can be strung together into a pipeline that conceptually accepts data as input and produces a model as output. (The actual topology of the pipeline can get much more complicated in practice, as we describe later.) Even though our discussion here is centered around TFX (to match the actual corpus), we note that the concepts that we introduce are present in other end-to-end ML frameworks discussed previously.

2.1 Basic Concepts

We use the term ***pipeline*** to represent a graph of operators that are connected in a producer/consumer fashion. Figure 1(a) shows a simple TFX pipeline comprising three operators: *ExampleGen*, which imports data in a format suitable for training; *Trainer*, which uses the imported data to train a model; and *Pusher*, which takes the generated model and deploys it for inference in some external service (e.g., TensorFlow.Serving). The topology of the pipeline graph corresponds to the input/output relationships between operators. Moreover, we assume that these relationships are "type-checked", e.g., *ExampleGen* outputs data in a format that is understood by the *Trainer*, when the pipeline is authored. The specifics of pipeline authoring are orthogonal to our work.

In itself, a pipeline encodes only coarse-grained dependencies between its operators. In contrast, a pipeline ***trace*** records the fine-grained relationship between individual executions of the operators and the provenance of their input/output artifacts when the pipeline runs in production. Formally, a trace is a directed acyclic graph of *execution* nodes and *artifact* nodes, with edges linking execution nodes to their input/output artifacts. Figure 2(a) shows a sample trace of the pipeline in Figure 1(a), where the nodes are arranged horizontally, left to right, in increasing order of the finish time for executions and creation time for artifacts. In this trace, the *ExampleGen* operator has executed three times, producing one *data span* artifact per execution. A *data span* artifact corresponds to a chunk of data whose semantics depend on the pipeline. For instance, if the pipeline trains models to recommend videos to users, then a single data span might correspond to previous user interactions with the video service over the period of a single day. Continuing with the example, the *Trainer* operator has executed two times, each time reading the last two data spans in the pipeline and producing the corresponding model. We note that this pattern is quite common in practice: there is a data ingestion process (here, the *ExampleGen* operator) that produces outputs at a fine level of granularity (e.g., each span corresponds to a single day of data) and the model training process reassembles the data into coarser granularity (e.g., a rolling window of the last two days). The example trace concludes with the *Pusher* operator, which executes once for the first model but not for the second model, which means that the second model was not deployed. The latter case is not uncommon in production and can be attributed to several reasons, e.g., the model does not have better performance than the previous model or fails certain validation checks, or the deployment mechanism throttles model pushes to avoid overloading.

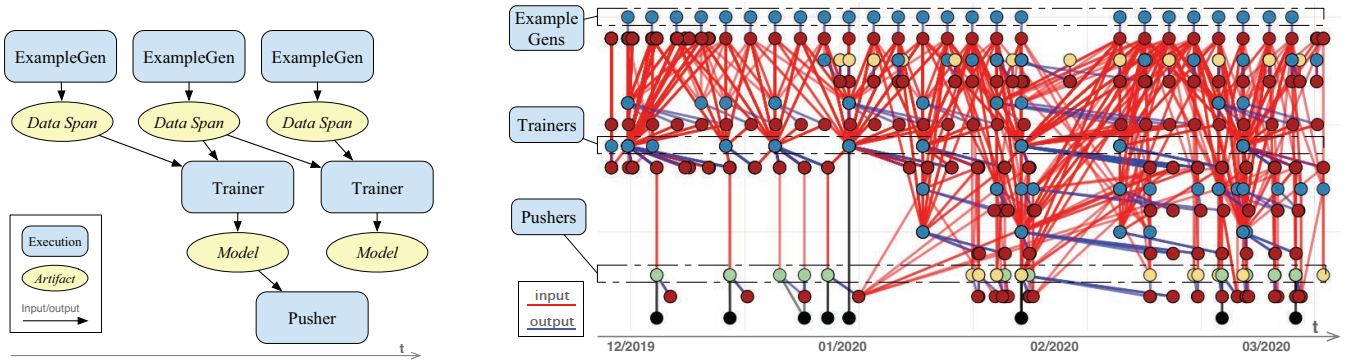
Note that a trace does not reflect any information about the orchestration of the pipeline operators. For instance, the example trace might be generated through a serial execution of operators, or by asynchronous scheduling where several operators with overlapping inputs can run at the same time. Moreover, a pipeline may be triggered periodically (e.g., by ingesting the newest span of data



(a) A simple TFX pipeline comprising three operators: *ExampleGen*, *Trainer*, and *Pusher*. Edges denote the input/output dependencies.

(b) A more typical TFX pipeline that comprises additional operators for data preprocessing, data and model validation, and tuning. Shaded operators correspond to the additional functionality compared to (a).

Figure 1: Examples of TFX pipelines



(a) A simple trace for the pipeline in Fig. 1(a).

(b) A real-world trace from our corpus, using operators shown in Fig. 1(b).

Figure 2: Examples of pipeline traces. The left-to-right order preserves the time of artifact generation.

every hour and triggering new runs of the operators) or manually (e.g., a model developer reruns the pipeline after making changes to the input data or training code). All we assume is that some external system is responsible for scheduling operators, and the trace will grow over time with every run of the pipeline to contain the full history of operator executions and generated artifacts.

Up to this point, we used a simple example comprising the most elemental steps of an ML production pipeline: data ingestion, training, and deployment. In practice, pipelines are more complicated. First, there are several other operators that correspond to important stages and safety checks in production ML. Figure 1(b) shows a more typical pipeline that expands on the simple pipeline of Figure 1(a) by including operators for data analysis and validation, data pre-processing, and model analysis and validation. Pipeline authors may also introduce custom operators depending on their ML task. Second, these operators can be wired in different topologies, e.g., a Trainer operator might generate an initial model that is then distilled through a separate Trainer operator to produce the final model, or the data-validation operator might block the execution of downstream operators if the data contains any errors. Last, the configuration of the pipeline may change over time. For instance, the pipeline might start with training a single "production" model, then be augmented over time with more Trainer operators that correspond to "experiment" models (some of which might become production models eventually).

This pipeline-level complexity carries over to the recorded traces. Moreover, it is common to have long-running production pipelines that continuously ingest data spans and output "fresh" models, resulting in traces that continuously grow over time. Add to that the fact that executions often share artifacts (e.g., through the rolling-window mechanism described in Figure 2(a)), and it becomes clear that traces can easily become large graphs with complicated structure that are challenging to analyze. Figure 2(b) shows one such large real-world trace; our corpus has pipelines with as many as **6900 nodes**. This observation motivates our proposal to analyze traces through lower-complexity or finer-grained *model graphlets* that essentially "unnest" each trace graph with respect to the generated models—more on this in Section 4.

2.2 Corpus of Traces for Analysis

The main contribution of this paper is the first-ever analysis of a corpus of production ML pipelines. The corpus comprises the traces of 3000 TFX pipelines (each with up to 6900 nodes) deployed at Google over a period of four months. We focused on pipelines that generated at least one trained model and had at least one model deployed outside the pipeline. These are the production pipelines whose models support downstream applications. The resulting pipelines correspond to hundreds of teams that span product areas (e.g., advertising, video recommendations, app recommendations, and maps), ML tasks (e.g., single-/multi-label classification, regression, and ranking), and model architectures (linear and deep models of varying complexity). In total, the collected traces contain 7.7M execution nodes and 20M artifact nodes.

The traces were collected using the ML Metadata [3] framework (MLMD), which powers the management of metadata and provenance in TFX [16]. Similar to our earlier definition, MLMD models a pipeline trace as a graph of *Execution* and *Artifact* nodes with their input/output relationships. We note that the recorded MLMD traces

also carry *Context* nodes to represent the grouping of *Artifact*s and *Execution*s, but this information is not used in our analysis. MLMD records additional metadata per node (e.g., start and completion time of *Execution*s, creation time of *Artifact*s), which we use to glean information about the triggering cadence of pipelines. Moreover, the corpus records high-level information for each data span artifact, comprising the features present in the span, their types, and statistics for different feature types (e.g., the unique values for a categorical feature, or the mean and standard deviation for a numerical feature). It's worth mentioning that both TFX and MLMD are open sourced and similar traces can be derived by TFX users.

3 PIPELINE-LEVEL ANALYSIS

We begin with an analysis of the corpus at the level of entire traces, aiming to understand higher-level characteristics of ML pipelines, such as the structural properties of the corresponding graphs, common data types and transformations, common ML model architectures, and the cost of different operators in addition to training. Section 4 will present a finer-grained analysis at the level of sub-traces, to understand the behavior of ML pipelines from the perspective of individual models.

3.1 Pipeline Lifespan and Activity

We first examine two aspects of ML pipelines: their typical lifespan and model training frequency. We define the lifespan of a pipeline as the count of days between the timestamps of the newest and the oldest nodes in its trace. Hence, the lifespan is an indication of how long the pipeline is active. The distribution in Figure 3(a) shows that, on average, pipelines are active for 36 days, with some being active for the entire span of our corpus (130 days). The longer lifespans correspond to continuous pipelines that generate a stream of models based on a stream of incoming data and are common within product groups that are heavy users of ML. Figure 3(d) shows the distribution of lifespan broken down by model type, where "DNN" refers to all deep learning models, "Linear" refers to all generalized linear models, and "Rest" contains all other models, including tree-based models. Interestingly, the pipelines with linear models live longer than the ones with DNN models.

Figure 3(b) shows the frequency distribution of the average number of trained models in the trace, per day. The majority of pipelines generate one model per day but there is also a wide spread of cadences, with some pipelines training close to 1000 models per day! Upon closer investigation, these are continuous pipelines that have a fast stream of incoming data and the ability to quickly produce updated models. As in the previous figure, these pipelines correlate well with product teams that are heavy users of ML. In addition, in Figure 3(e), we find that the cadence of pipelines using DNN are more diverse than the other methods, which reflect a wide adoption of it in different problems.

Overall, pipelines can have a large lifespan and generate models at a high rate. In turn, the accumulated state of these pipelines can become complex. For instance, the trace of the more complicated pipelines can have up to 6953 (artifact or execution) nodes. Tools to efficiently query or summarize these complex traces can become indispensable for humans to debug or manage these pipelines.

3.2 Pipeline Complexity

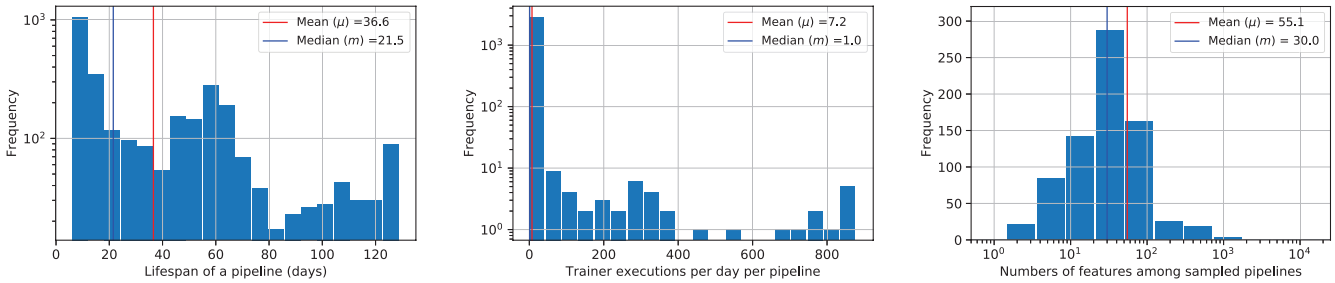
Next, we analyze the complexity of ML pipelines. Specifically, we examine three aspects: 1) the shape of the input data; 2) the typical transformations for training; 3) the diversity of model architectures.

Input Data Shape. Figure 3(c) and Figure 3(f) show the distributions of the feature count in the input data. We use "feature" to refer to a column in the data (e.g., the 'video-id' column) and "domain" for the set of values that this feature has in the input examples. We compute the distribution as follows: we identify the data span nodes in all traces, and use the associated MLMD metadata for each data span (Section 2.2) to retrieve the number of features. As shown, the vast majority of the pipelines utilize up to 100 features. However, higher feature counts do appear and there are extreme cases of tens of thousands of features in a pipeline.

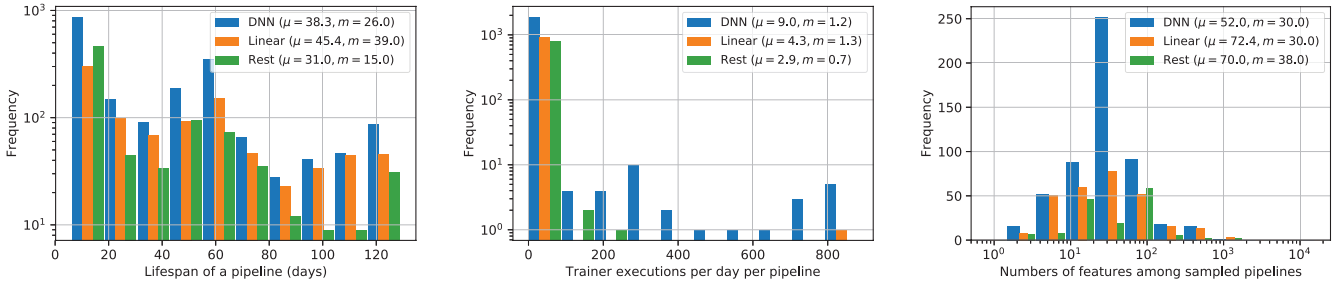
We also examine the composition of features in each pipeline in terms of numerical (e.g., length of a video) and categorical/sparse features (e.g., id of a video, query text). Note that this distinction does not necessarily reflect the type used to encode the feature. For instance, a 'video id' feature might be encoded as a number but treated as a sparse/categorical feature in training as well as in our breakdown. We find that the features of each pipeline are equally distributed among these two types: on average, 53% of the features are categorical. Moreover, we analyzed the domain of the categorical features and found that each categorical feature has, on average, **10.6 million** unique values in its domain. For pipeline with DNN models, the average is 13.6 millions, while for the ones having Linear models, it is larger than 20 million. This large domain size indicates high data complexity and thus increased cost and complexity for data transformations prior to training. For instance, we discuss below how such categorical features are typically embedded in vocabularies prior to training.

Feature Transformation. The additional metadata that we collect in our corpus (see Section 2.2) provides a glimpse into the types of transformations applied to the raw data before training. These transformations are often applied in two stages: *1)* deriving necessary statistics for the transformation; *2)* applying the transformation using the statistics. As an example, consider the common z-score transform for numerical features: the first stage computes the mean and standard deviation of the feature values, and the second stage uses these two metrics to normalize each feature value. In general, the second stage is embarrassingly parallel and can thus easily scale to large datasets. The first analysis stage, however, is much more expensive as it requires potentially expensive reductions over the data (e.g., sorting a large space of video ids based on frequency). We thus focus on this analysis stage in what follows, since that is where we see opportunities for data processing-related optimizations.

Figure 4 shows the different types of analyses applied in the first stage of feature transformations. The "vocabulary" analyzer performs the aforementioned truncation of a sparse feature into a smaller numerical domain. As an example, given a feature that contains text tokens (e.g., words), this analysis computes the top-K tokens based on frequency and then maps the features to the numerical domain $[0, K]$. The other types correspond to more straightforward analyses over numerical features (e.g., min, max, and so on). Finally, "custom" refers to a black-box analysis (essentially, a UDF) that is pipeline-specific and tailored to the business logic of the



(a) Distribution of pipeline span. (b) Distribution of trained models per day. (c) Distribution of the number of features.



(d) Distribution of pipeline span. (e) Distribution of trained models per day. (f) Distribution of the number of features.

Figure 3: Pipeline Activity and Data Complexity Analysis Results.

corresponding ML task. Pipeline authors resort to these UDF-style analyses when their features require more complex handling than what TFX offers out of the box.

Figure 4 shows two views of the same data. At the top, we show the percentage of pipelines that reference each analysis at least once, which indicates the relevance of these analyses across the pipelines in our corpus. The bottom view shows the total usage of these analyses across all traces and indicates the frequency of their usage in production. We observe that custom analyses are also used in several pipelines, although their total usage is much lower in the bottom chart. Our hypothesis is that such UDF-based analyses are more relevant for experimental pipelines that have a short lifespan and less activity, whereas canonical analyses provide good coverage for "steady-state" pipelines.

Both views in Figure 4 confirm the prominence of vocabulary computations over categorical features, which provides an interesting opportunity for the data management community, in two

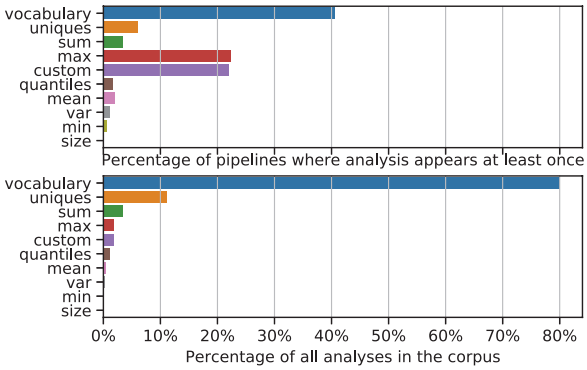


Figure 4: Analyzer Usage

respects. First, this computation is a top-K query over an aggregation of the data where K can be very large, e.g., values of K from hundreds of thousands to millions are not uncommon. It is interesting to consider how these queries can be optimized for different representations of the data. Second, the choice of K has non-trivial implications on model quality and performance. A higher K implies better coverage of "important" sparse values, which in some cases results in significant improvements in accuracy. At the same time, since this mapping becomes part of the model, a higher K implies larger model sizes (to store the mapping) and perhaps higher processing time (to perform the mapping). To make an informed choice, the model developer needs to understand the tradeoffs for different values of K in terms of these dimensions, and this, in turn, introduces an interesting data analysis problem that may be amenable to techniques from data management (e.g., approximate query answering for this class of queries).

Model Diversity. We next examine the type of models used across pipelines, which helps characterize the diversity of training methods in production. Moreover, the choice of model architecture affects the selection of other operators and hence other characteristics of the pipeline. Figure 5 shows the usage of different architectures as a percentage of all models in our corpus. As we can see, 64% of them use deep neural networks (DNNs), with an additional 2% that use a combination of DNNs with linear models. A smaller percentage of pipelines employ linear models and tree-based methods. A small fraction of the models deal with specialized tasks requiring ensemble models or custom methods that we lump in the "other" slice.

It is interesting to relate this breakdown to recent papers that embed ML techniques inside database management systems and thus aim to jointly optimize data access and model training [24,

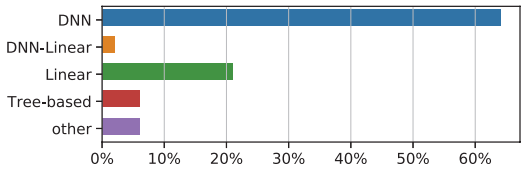


Figure 5: Percentage of Trainer runs with each model type

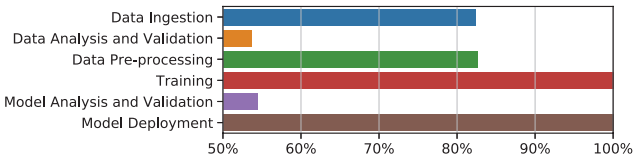


Figure 6: Percentage of pipelines with different operators.

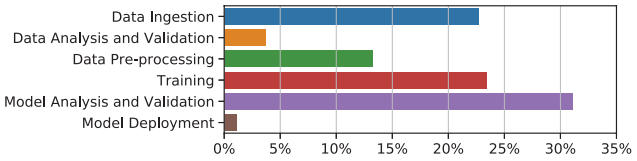


Figure 7: Compute cost of different operators.

29, 32, 40]. A focus on a specific class of models (e.g., linear or DNN models) is still relevant in practice and can cover a significant fraction of production ML workloads. However, this would leave on the table a much bigger fraction of pipelines outside of the selected class that could also benefit from optimized systems.

3.3 Resource Consumption

Finally, we turn our attention to the composition of pipelines in terms of the different operators and their corresponding resource footprints.

Figure 6 shows the different types of TFX operators present in our traces and the corresponding percentage of pipelines using these operators. Here we group them in terms of their high-level functionality in the pipeline: data ingestion; data analysis and validation; data pre-processing; training; model analysis and validation; and model deployment. Perhaps unsurprisingly, the most common operators include data ingestion, data pre-processing, training, and deployment, with training and deployment in 100% of the pipelines since our corpus focuses on ML pipelines that support production applications. It is also worth noting that about half of the pipelines employ data- and model-validation operators, which essentially block the deployment of the trained models if there are errors in the data or if the model metrics are not sufficiently good, respectively [18, 21]. These operators act as safety checks and are common when trained models used in downstream production systems.

Figure 7 shows a different breakdown of operators based on their resource usage. For each group, the figure shows the total compute cost as a percentage of the overall compute cost by all groups. Perhaps surprisingly, we see that the training operators account for less than a third of the total computation. This finding goes against the conventional picture that ML is mostly about the training algorithm. In contrast, our results indicate that production ML involves more steps that are also more costly, e.g., the data/model analysis and validation operators account for ~35% of the total compute

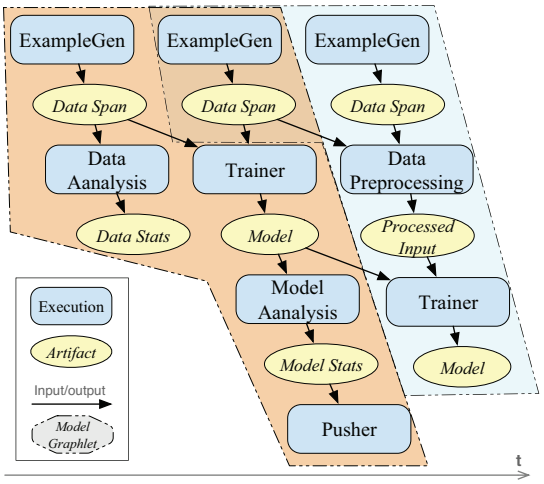


Figure 8: A model graphlet example

cost of the pipelines and are more expensive than training. Note that like data analysis and validation, model analysis and validation also boils down to traditional data processing operations, since it involves the computation of model metrics (e.g., average loss, area-under-the-curve) on slices of the input data, i.e., group-by queries with a model-driven aggregation per group. The figure also shows a significant cost for data ingestion (~22%) as TFX initiates a "hermetic" copy of each data span from the external data source, along with shuffling/partitioning of the examples to different splits (training/testing/eval) for the downstream model.

Besides showing the cost of operators beyond training, the breakdown in Figure 7 reveals one more point: pipeline failures can be costly in terms of resources. Specifically, ML pipelines have several failure points corresponding to the different operators, e.g., the pipeline may stop because the data contains errors, or because the training code has faults, or because model validation failed. In turn, each failure point may occur after the successful completion of upstream operators, several of which can be costly based on our analysis. In other words, failures are not cheap and there is an upside to preventing them or dealing with them proactively (more on this in Section 5). Moreover, artifact caching and reuse, when feasible, can cut down significantly on the cost of different stages. We can use the costs in Figure 7 (in conjunction with failure probabilities) to determine optimized materialization policies [50].

4 FINE-GRAINED GRAPHLET ANALYSIS

Next we dial up the resolution of the analysis to investigate fine-grained characteristics within each pipeline, and in particular, the cadence and characteristics of model training and deployment—which, as stated earlier, can happen several times within a trace.

4.1 Model Graphlets

Recall that pipelines can be continuous, in that they consume a stream of incoming data spans to train and produce fresh models. Hence, a single pipeline trace may contain multiple executions of the Trainer operator (and associated downstream and upstream operators) on overlapping inputs. For instance, when using a rolling-window on the input data spans to update models, a data span might

contribute to several models; when using warm-start during training, a generated model artifact might be used as an input to other Trainer executions. As a result, the pipeline trace is intricately tied to the complexity of ML in practice, and it is not uncommon for the trace to have a single large connected component that encompasses all nodes, which in turn makes it difficult to examine the finer-grained characteristics of the pipeline (e.g., Figure 2).

The issues of inter-connectedness and size of provenance graphs have similarly emerged in different domains, wherein techniques such as user views, segmentation, and aggregation have been explored to transform the graphs to usable or interpretable ones [9, 17, 36, 37]. We adopt a similar approach but we leverage the semantics of production-ML operators and connections between them. Specifically, we segment the trace graph into several subgraphs, so that each subgraph represents a single (logical) end-to-end pipeline run related to an individual model. We provide some intuition for this definition in the context of an example below. We refer to such a subgraph as a *model graphlet*, or graphlet for short. The segmentation works as follows: given a Trainer execution node n, the corresponding graphlet g is a subgraph of the pipeline trace that comprises the following nodes, and the corresponding edges:

(a) all ancestor executions of n (and their input/output artifacts);
(b) all data-analysis/-validation executions performed on data spans from the previous step (and their input/output artifacts); and
(c) all descendant executions of n (and their input/output artifacts) that are not on paths to other Trainer executions.

At the highest level, rule a captures all ancestor executions that directly influence n along with their artifacts; rule b recognizes the fact that there may be data analysis/validation executions performed on data spans that are input to n or one of its ancestor executions, but are not themselves ancestors of n, and therefore includes any such data analysis/validation executions; rule c adds descendent executions of n, post-training, that are not counted in the graphlets of other Trainer executions—this is checked by ensuring that they are not on paths to such Trainers.

Example. Figure 8 shows the graphlets extracted on a sample trace. Intuitively, each graphlet corresponds to a model and captures the subgraph of the trace that is relevant for the generation (rule a) and deployment (rule c) of the model. Note that the first model is used to warmstart the Trainer for the second model, yet per rule c this edge is a "cut" between the two graphlets. The reason is that we aim to create smaller graphs of bounded complexity, so that we can analyze the corpus at the granularity of individual models. The graphlets also include data-analysis artifacts (rule b), so that we can analyze data-related metrics (re-use and similarity) across graphlets.

From this point onward, we recast our analysis on the set of graphlets that we extract from the corpus. In total, this gives rise to 450,000 graphlets.

4.2 Data Change across Graphlets

We now shift our attention to the evolution of input data across graphlets in the same pipeline. Beyond the immediate implications of data characteristics on ML model performance [43], by understanding the reuse and similarity of data across graphlets, we can glean insights related to ML data management, e.g., to what extent materialization of intermediate data transformations and incremental computation can be useful for ML pipelines. The analysis that follows uses the notion of *consecutive graphlets*. Two graphlets g and g' from the same pipeline are said to be consecutive iff the corresponding trainer executions are adjacent in chronological order.

4.2.1 Reuse. We quantify data reuse with the Jaccard similarity between the data spans of consecutive graphlets: $|\mathcal{I}(g) \cap \mathcal{I}(g')| / |\mathcal{I}(g) \cup \mathcal{I}(g')|$, where $\mathcal{I}(g)$ is the set of input data spans in g.

The first row in Table 1 shows the histogram of the similarity values in the corpus. The high value at (0.75, 1]—57% of all pairs of consecutive graphlets—is due to the fact that many pipelines train multiple parallel models on the same inputs for A/B testing. Moreover, consecutive graphlets can correspond to retrainings on the same data after the pipeline author changes other details of the process, e.g., the training algorithm or feature transformations. The mean similarity of .647 indicates that, on average, graphlets share two thirds of the data spans. Moreover, several consecutive graphlets have a $> 80\%$ similarity of their inputs. These results point to interesting optimization opportunities on data preparation for training. Specifically, we can leverage this overlap and employ techniques from incremental data computation and view maintenance to efficiently perform the data preparation steps for a new training run. Indeed, most of the data analysis operators in Section 3.2 lend well to incremental view maintenance techniques [31].

4.2.2 Dataset Similarity. Next, we examine whether span contents come from the same data distribution, which unlocks further opportunities for reuse and optimization across graphlets. For instance, if two consecutive graphlets have data inputs with the same distribution, then some aspects of data preparation can be reused even if the data spans are different, e.g., the second graphlet can reuse the vocabularies of the first graphlet over the same categorical features (see also Section 3.2). One "extreme" optimization is to skip training altogether, given that the data distribution is the same!

Measuring data similarity between consecutive graphs requires special considerations due to the fact that a graphlet can contain multiple data spans, and we only have summary statistics for each data span. We elaborate on the data similarity metric in the technical report [51].

The second row in Table 1 shows the quartiles of data-similarity values over all pairs of consecutive graphlets in the corpus. Similar to Jaccard similarity (first row), dataset similarity is bimodal at the first and last quantiles. However, the trend is reversed, due to the fact that data spans common in both sets may not be paired up in to ordinal position matching. As explained above, this is by design to account for cases of sequential data visitation in training. The third row in Table 1 shows the quantiles of data similarity when the latter is averaged over all graphlets in the same pipeline. Compared

	[0, 0.25]	(0.25, 0.5]	(0.5, 0.75]	(0.75, 1]	μ
Jaccard	30.2%	8.2%	4.4%	57.3%	0.647
Dataset	89.7%	0.3%	0.1%	9.9%	0.101
Avg Dataset	87.3%	5%	3.1%	4.6%	0.092

Table 1: Similarity metrics for consecutive model graphlets; percentage of consecutive graphlet pairs in each similarity range, along with the mean similarity.

to the second row, we observe a drop in the higher quantiles, which indicates that pipelines with a large number of graphlets have more dissimilar pairs. Put differently, long-running pipelines belonging to power users have higher data volatility, motivating the need for data validation to safeguard against data errors and drift.

4.3 Model Retraining and Deployment

In this section, we use the graphlet decomposition to examine the relationship between model (re)training and deployment in the pipelines. As our analysis shows below, the conclusion is that (a) there are many trained models that are not deployed, and (b) very likely they correspond to wasted computation, which in turn introduces an interesting optimization opportunity.

4.3.1 Training vs. Deployment. A model *push* marks the deployment of a trained model to a downstream service, thus making the model "visible" outside of the pipeline. In TFX, a push is performed via the execution of the Pusher operator. Note that a newly generated model may remain unpushed (e.g., due to failure to validate, or downstream throttling), in which case downstream services will keep using the last-pushed model. In this sense, a model push causes a model "refresh" for the downstream services.

Figure 9(a) shows the distributions of this time gap (in hours) for the two classes, while Figure 9(b) shows the cumulative view of the same distributions, both in log scale on the x axis. Immediately, we see that the two distributions have the same shape, but the mean is upshifted by ∼15 hours for the time between pushed graphlets. This is corroborated by the CDFs, which show that the time between 80% of all graphlet pairs is less than the median value for the time between pushed graphlets. These trends can have two possible explanations: 1) pushed graphlets tend to run longer than unpushed graphlets, or 2) pushed and unpushed graphlets have similar run times but are interleaved, thus widening the time gap between pushes. The second explanation has important implications for system optimization, motivating further effort to unearth the cause.

To further investigate the discrepancy between model training cadence and model push cadence, we plot the distribution of the number of graphlets between consecutive model pushes in Figure 9(c). We see that very few pipelines have no intervening unpushed graphlets between consecutive pushes, while most pipelines have between 1 to 10 unpushed graphlets between consecutive model pushes, with an average of ∼3. Furthermore, Figure 9(d) shows that unpushed graphlets have higher training costs than pushed ones overall. These two facts together confirm that the second explanation presented above, namely the interleaving of pushed and unpushed graphlets, is indeed the cause for the difference in distribution observed in Figure 9(a).

4.3.2 Model Freshness vs. Wasted Computation. The conclusion that pushed graphlets are interleaved with unpushed graphlets is a cause for concern for two reasons: 1) if the application requires model pushes to keep up with the ingestion of new data, then a mismatch between the cadences of model training and model pushing implies an "unhealthy" pipeline; 2) if the application is tolerant of unpushed models, model training runs that do not lead to model pushes can be skipped to minimize wasted computation. In either case, the unpushed graphlets are potentially wasted computation.

	μ_{pushed}	$\mu_{unpushed}$	μ
Input data similarity	0.109	0.099	0.101
Code match	0.838	0.846	0.845

Table 2: Model push vs. data drift and code change.

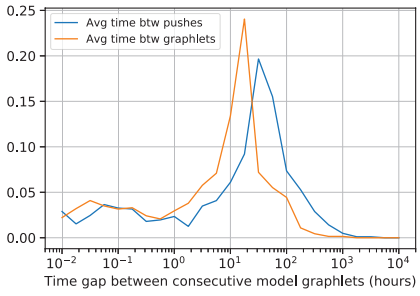
While we do not have the requisite telemetry to precisely characterize when unpushed graphlets are indeed wasteful (e.g., unpushed models might be transitively useful by warmstarting models that do get pushed), our analysis results present quantitative evidence that the total amount of wasted computation is high. First, we observe that approximately 80% of all graphlets are unpushed. If none of the graphlets were used for warmstarting and there were no overlaps between graphlets, 80% of graphlets would account for ∼80% of overall computation since pushed and unpushed graphlets have similar CPU usage. To account for warmstarting and potential overlaps between graphlets, we can simply 1) remove graphlets belonging to pipelines with warmstarting, which account for 9% of total graphlets, and 2) remove all computation cost for operators that could potentially overlap, which accounts for ∼ 60% of overall computation resource consumption, computed by removing the warmstarting pipelines from the results in Figure 7. Even with these generous assumptions the waste is still at > 30%.

There is clearly an interesting question in whether we can identify the root cause of this inefficiency. However, further analysis shows that it is unlikely to find simple explanations. (We develop a more involved solution for the problem in Section 5.) Specifically, suppose that we attempt to explain the inefficiency through the following hypotheses: 1) rate-limited push: model pushes are configured to be at least a certain time apart, and models are trained faster than the predetermined push rate; 2) model types: certain model types can be more error prone and fail due to non-determinism with all else held equal; 3) data drift: shift in data distribution prevents models from passing validation checks; 4) code change: updating the code for the Trainer introduces system or model bugs.

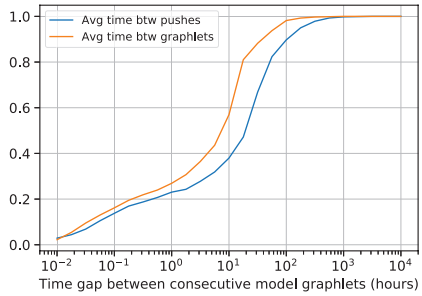
To check (1), we compare the distribution of the model training time shown in Figure 9(e) with the distribution of the time gap between graphlets shown in Figure 9. While the average time gap between pushed graphlets is ∼40 hours, the mean model training time is 168 hours, making it highly unlikely that models are trained faster than the allowed push rate. This eliminates 1) as the cause of wasted computation.

To check (2), we observe from Figure 9(f) that the likelihoods of graphlets being pushed for different model types is highly variable. Moreover, all model types have a likelihood < 0.6, indicating that no single one is the culprit. However, it is unclear whether the low likelihood for some model types is due to the model type being error prone or other confounding factors.

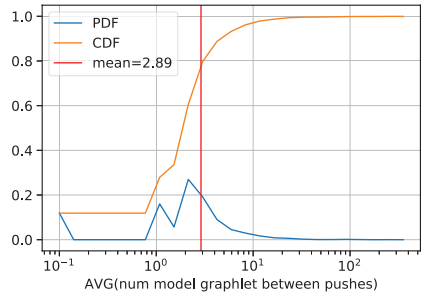
Finally, to check (3) and (4), we compare the means for input data similarity, as measured by the metric introduced earlier, and code match (where 1 indicates match and 0 indicates no match) with the immediate preceding graphlet for the pushed and unpushed graphlets shown in Table 2. Overall, between consecutive graphlets, the input data similarity is 0.101, and the code stays the same 84.5% of the time. For both measures, we observe no significant difference between the pushed and unpushed groups. These findings suggest that code change and data drift are not major causes of a graphlet not pushing a model.



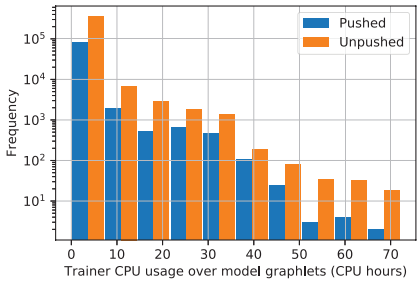
(a) PDF of the average time between consecutive model graphlets.



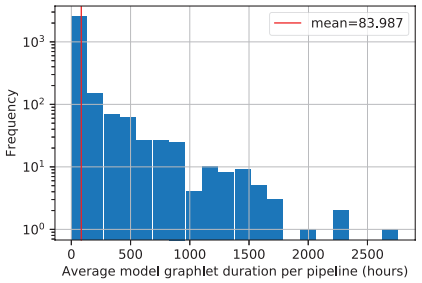
(b) CDF of the average time between consecutive model graphlets.



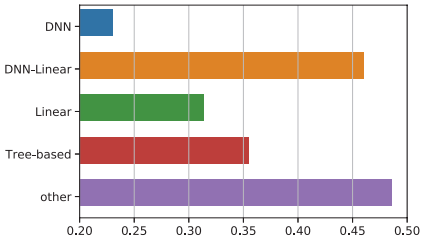
(c) Distribution of number of graphlets between pushes.



(d) CPU usage for the trainer.



(e) Model graphlets duration in hours.



(f) Model type vs. likelihood of pushes.

Figure 9: Model graphlet analysis.

5 MITIGATING WASTED COMPUTATION

Our analysis in Section 4 revealed that unpushed graphlets represent a significant chunk of the total pipeline computation, even though they are not likely to have an observable impact on subsequent models or downstream services. In turn, this raises a new optimization opportunity: predict whether a graphlet will not lead to a pushed model and then adjust its execution in order to conserve compute cost. For instance, the pipeline scheduler may choose to down-prioritize or stall such graphlets until the pipeline owner intervenes and fixes the underlying issue(s).

In this section we design a decision function for this prediction task. Clearly, the decision function has to balance between different types of errors and their effects. A false negative, i.e., skipping a graphlet that would have resulted in a model push, compromises model freshness, while a false positive, i.e., running a model graphlet that does not result in a model push, contributes to wasted computation. We discuss later a method to explore the tradeoff between the two error types and show that we can eliminate nearly half of the wasted computation.

Data. To study this problem, we form a dataset by filtering the previous corpus to only include pipelines that do not warmstart model training with previous versions of the model. Unpushed graphlets are useful if they help warmstart subsequent model training (see Section 4.3) and so we should not consider them as wasted computation. This leaves us with 2827 pipelines containing 420k graphlets in total. The dataset contains 80% unpushed graphlets and 20% pushed graphlets. To account for class imbalance, we use balanced accuracy to measure the fitness of decision functions.

5.1 Problem Statement

Let $\mathcal{G} = \{g\}$ be a set of model graphlets and $\mathcal{Y} : \mathcal{G} \rightarrow \{0, 1\}$ be the indicator function that evaluates to 1 for pushed graphlets and 0 otherwise. The objective of the waste mitigation problem is to find

$$\underset{\hat{\mathcal{Y}} \in \mathcal{H}}{\operatorname{argmin}} \sum_{g \in \mathcal{G}} L_m\left(\mathcal{Y}(g) \cdot (1 - \hat{\mathcal{Y}}(g))\right) + L_w\left(\hat{\mathcal{Y}}(g) \cdot (1 - \mathcal{Y}(g))\right) \quad (1)$$

where $\mathcal{H}$ is the space of decision functions that we will explore to find $\hat{\mathcal{Y}}$, an approximation for $\mathcal{Y}$, L_m is the loss function for model freshness that depends only on false negatives, and L_w is the loss function for wasted computation that depends only on false positives. Intuitively, model freshness is only affected by false negatives, where pushed graphlets are incorrectly predicted as `unpushed` and therefore prevented from updating the externally visible model; on the other hand, wasted computation can only be incurred by false positives where an unpushed graphlet was erroneously run without resulting in a refreshed model downstream.

The loss functions can be designed to prioritize either cost saving or model freshness. However, it is difficult, in practice, to determine the tradeoff *apriori* without getting a sense of the complexity of the dataset and the decision function. To overcome this challenge, one can use a single loss function L for both L_m and L_w to weigh the two types of errors equally and allow $\hat{\mathcal{Y}}$ to be a real-valued function that assigns likelihoods to the two labels, namely `pushed` and `unpushed`. Once $\hat{\mathcal{Y}}$ is found, the tradeoff between compute cost and model freshness can then be made post-hoc by setting a specific threshold on $\hat{\mathcal{Y}}$ to produce a binary decision function. Another convenient consequence of setting $L_m = L_w$ is that the problem becomes a standard binary classification problem that can be solved using standard approaches.

Limitations of Simple Heuristics. The analyses from Section 3 and 4 surface numerous candidate signals that can be used to solve the binary classification problem. We experimented with a few simple heuristics for solving Eq. 1 derived from our findings, e.g., model type, input overlap, and code match. The best handcrafted heuristic (model type) achieved a balanced accuracy of 0.6. The large search space of heuristics, on top of the low performance of heuristics we handcrafted, motivates the machine learning approach to automatically utilize complex signals for decision making.

5.2 Machine Learning Based Approach

We approximate the decision function $\hat{\mathcal{Y}}$ by training a supervised ML model. Each graphlet in the training data is labeled as `pushed` or `unpushed` as described above. For each graphlet, we create features based on its structure and associated metadata, mostly relying on the insights discussed in Section 3 and 4. We also introduce features involving the immediately preceding graphlets in order to capture temporal signals such as data-span similarity.

5.2.1 Features. We partition graphlet features into four categories described below. The first two categories (shape and model information) are features extracted from the graphlet itself. The remaining two categories (input data and code change) are history-based, i.e., they are derived by comparing a graphlet with a window of graphlets that immediately precede it. A distinct feature is created for each ordinal position in the window, e.g., for a window size of three, `code_change_1`, `code_change_2`, and `code_change_3`, are created to indicate whether the code in graphlet g has changed compared to those that are 1, 2, or 3 graphlets prior to g.

Graphlet shape. Shape features include the count of executions corresponding to each operator, as well as the average input and output count for each execution. We partition the operators into *pre-trainer* operators that can execute without the output of the Trainer, the Trainer, and *post-trainer* operators that validate the output of the Trainer for safety and quality. For example, the graphlet in orange in Figure 8 has 2 ExampleGen with on average 1 output, 1 Trainer with on average 2 input and 1 output, 1 Data Analysis and 1 Model analysis both with average input and output count both being 1, and 1 Pusher with on average 1 input. Note that to obtain the execution count for a particular operator, we need to run the graphlet up to that operator, incurring computation overhead to obtain these features. Recall that the cost to run the post-trainer operators is ~30% of the pre-trainer operators, as shown in Figure 7.

Model information. We include the model type, e.g. Linear, DNN, etc., as well as the model architecture for graphlets containing DNN models, as one-hot encoded features. Figure 9(f) indicates that model type and model pushes are correlated.

Input data. Input data-related features include both overlap computed using the input Jaccard similarity (Section 4.2.1), and dataset similarity (Section 4.2.2), between a graphlet g and the graphlets preceding g as history-based features.

Code change. A binary feature for whether the code versions for the Trainer operator match between a graphlet g and the graphlets preceding g, as history-based features.

We also experimented with pipeline-level features inspired by the findings in Section 3, such as the data transformations, average time between graphlets and average graphlet duration in a pipeline, but found no significant improvement in model performance.

	Model	Balanced Acc.	Feature Cost
Random Forest	RF:Input	0.737	0.31
	RF:Input+Pre	0.801 (+9%)	0.53 (+71%)
	RF:Input+Pre+Trainer	0.818 (+2%)	0.77 (+ 45%)
	RF:Validation	0.948 (+16%)	1.00 (+30%)
Ablation	RF:Input	0.737	0.31
	RF:History	0.738	0.77
	RF:Shape	0.680	0.77
	RF:Model-Type	0.592	0.77

Table 3: Balanced accuracy for all model variants. The feature cost column indicates the compute cost to obtain the necessary features (rescaled to $[0, 1]$ with RF:Validation = 1).

5.2.2 ML Model Training and Testing. We split pipelines in the corpus at random into a training and a test set, such that the total number of graphlets belonging to pipelines in the training set is ~80% of the total number of graphlets in the entire corpus, and the distribution of pushed and unpushed graphlets are roughly the same in the training and test sets.

We experimented with a large variety of models including DNNs and Gradient Boosted Decision Trees, as well as more interpretable models, such as Logistic Regression and Random Forest, using the Scikit-learn library [41], and found that Random Forest performed comparably with the more complex models explored by the AutoML tool. We report our results from using the Random Forest model below and discuss lessons learned from the model next.

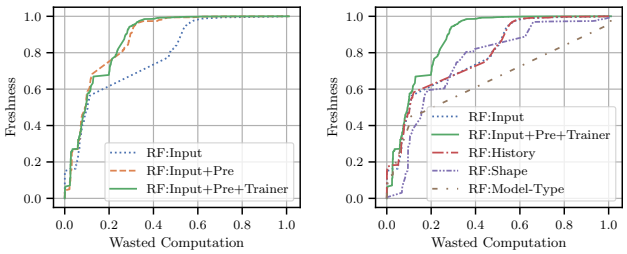
5.3 Evaluation

We evaluate the models on both model performance and their ability to generate execution policies for reducing wasted computation.

5.3.1 Classification performance. We present the results for four variants of the Random Forest model:
RF:Input has all of the features except the graphlet shape features;
RF:Input+Pre has all of the features in RF:Input plus the graphlet shape features for pre-trainer operators;
RF:Input+Pre+Trainer has all of the features in RF:Input+Pre plus graphlet shape features for Trainers;
RF:Validation has all of the features in RF:Input+Pre+Trainer plus shape features for post-trainer operators in the graphlet.

The four variants can be thought of as incrementally revealing more features about the shape of the graphlets as more operators are executed. They represent points in the pipeline execution that the system can intervene and abort execution to minimize wasted computation. For example, with RF:Input, the system can decide to abort as soon as the data is ingested; with RF:Input+Pre, the system has the option to abort right before model training. Note that RF:Validation is rather impractical in term of reducing wasted computation, as it requires all operators in the graphlet to be run to obtain the features. Also, the execution of validation operators can be highly correlated with whether the graphlet is pushed. We include RF:Validation as a proxy for an upper bound on prediction performance given complete (i.e., oracular) information.

Table 3 shows the model accuracy and the computation cost for the features used in each model, with the cost rescaled to $[0, 1]$ by



(a) Model freshness vs. wasted computation curves for Random Forest models.

(b) Model freshness vs. wasted computation curves for the ablation study.

Figure 10: Evaluation Results.

dividing by the maximum cost attained by RF:Validation. Adding additional information leads to better model accuracies, however, the gain in accuracy does not grow linearly with the compute cost. Notably, from RF:Input+Pre to RF:Input+Pre+Trainer, the difference between the two models corresponds to shape features for the Trainer operator, which incurs a 45% increase in computation cost, for a measly 2% lift in model accuracy. All of the models significantly outperform the heuristics presented in Section 5.1, suggesting that there are complex interactions between signals that are hard to capture with heuristics looking at one signal at a time.

5.3.2 System Performance Improvement. We now examine how the trained classifier can help balance the tradeoff discussed earlier: skipping unpushed graphlets reduces wasted computation, but skipping pushed graphlets compromises model freshness in downstream applications.

Figure 10(a) depicts this empirical tradeoff between model freshness (y-axis) and wasted computation (x-axis) for the trained classifier. We compute this curve by doing a parameter sweep of the binary-classifier's threshold. Each threshold value results in specific false positive and true positive rates over the evaluation dataset, which we translate to values for wasted computation and model freshness respectively. As an example, consider the rates (0.65, 1) for a specific threshold value, which corresponds to a decision function that classifies correctly all the pushed graphlets and mis-classifies 65% of the unpushed graphlets (based again on our evaluation dataset). Since all pushed graphlets are identified correctly, model freshness is equal to 100%. Moreover, we can aggregate the total compute cost of the mis-classified unpushed graphlets, which in this case sums up to 70% of the total compute cost for these graphlets (equivalently, we recover 30% of the wasted computation). Hence, we map the decision function to point (0.7, 1) in Figure 10(a).

We highlight two important takeaways from Figure 10(a). First, we can eliminate 50% of all wasted computation without sacrificing model freshness at all. For a large-scale ML system like TFX, this amounts to a great deal of computation resources saved without affecting end-user experience. Second, model freshness does not drop substantially if we cut computation even further from 50% to 60%. However, following that, model freshness drops dramatically from 0.4 to 0, implying that we cannot hope to do much better than removing 60% of the wasted compute without significant sacrifice on model freshness. While RF:Input is cheaper than RF:Input+Pre and RF:Input+Pre+Trainer as shown in Table 3, it is also a much worse execution policy. It can eliminate at most 30% of the wasted compute before model freshness suffers from serious decline. On the other hand, RF:Input+Pre and RF:Input+Pre+Trainer can easily eliminate 50% of the wasted computation with no impact on freshness. Thus, for a frugal user who is tolerant of model staleness, RF:Input might be the preferred model. A user who has strict model freshness requirements might prefer RF:Input+Pre to help them minimize waste without sacrificing model quality. The negligible improvement in prediction performance by RF:Input+Pre+Trainer compared to RF:Input+Pre does not justify the 45% computation overhead to obtain the features. Thus, RF:Input+Pre+Trainer, despite leading in prediction performance, is not as effective from a cost saving perspective.

5.3.3 Feature Ablation Study. To better understand the impact of the different groups of features introduced in Section 5.2, we conduct a feature ablation study, where Random Forest models are trained and evaluated with a subset of the features. The balanced accuracies for the ablated models are shown in the last four rows of Table 3. RF:History contains both the input data features and code change features. RF:Shape contains only the counts for the operators excluding validators. Of the four groups of features, RF:Input, while being the cheapest in terms of cost to obtain feature values, achieves the best performing model. Interestingly, adding the code features to the input features, as is done in RF:History, has no effect on the model performance. RF:Shape performs significantly worse than RF:Input but better than RF:Model-Type, which achieved the same performance as the simple heuristic involving model type.

Figure 10(b) shows the freshness vs. wasted computation curves for models in the feature ablation study. We use RF:Input+Pre+Trainer as the baseline since it is the best performing Random Forest model. Of all the feature classes, model features are the least informative by a long shot. The fact that RF:Input and RF:History have identical performance serves to validate that code changes are uncorrelated with model pushes. The interweaving of the RF:Input and RF:Shape curves suggest that these two groups of features are predictive for different subset of graphlets. The ablation study shows that no single group of features captures most of the accuracy gains, suggesting that there exist complex interactions between the different groups of features driving the performance of the best models.

6 CONCLUSIONS

To our knowledge, we presented the first ever large-scale analysis of production ML pipelines, based on a large corpus of pipelines from Google. Our analysis demonstrates the high complexity of these pipelines and the importance of operators other than training, in particular operators related to data preprocessing and analytics. Furthermore, we analyzed the cadence and characteristics of the generated models and characterized their relationship with respect to the input training data. The overall analysis revealed several points where techniques from data management (e.g., incremental view computation, or approximate query answers) can optimize different steps of these pipelines. Moreover, as an example of new optimization problems uncovered by our study, we identified the problem of wasted computation for models that are trained in these pipelines but not deployed to downstream services. To this end, we proposed and evaluated a solution to pro-actively identify such cases and thus avoid wasting resources without compromising the freshness of actually deployed models.

REFERENCES

[1] accessed 2020-09. *Kubeflow.* https://www.kubeflow.org.

[2] accessed 2020-09. *Metaflow.* http://metaflow.org.

[3] accessed 2020-09. *ML Metadata.* https://www.tensorflow.org/tfx/guide/mlmd.

[4] accessed 2020-09. *Pachyderm.* https://www.pachyderm.io.

[5] accessed 2020-09. *Towards ML Engineering: A Brief History Of TensorFlow Extended (TFX).* https://blog.tensorflow.org/2020/09/brief-history-of-tensorflow-extended-tfx.html.

[6] accessed 2020-11. *Machine Learning Library (MLlib) Guide.* https://spark.apache.org/docs/latest/ml-pipeline.html.

[7] accessed 2020-11. *sklearn.pipeline.Pipeline.* https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html.

[8] accessed 2020-11. *What are Azure Machine Learning pipelines?* https://docs.microsoft.com/en-us/azure/machine-learning/concept-ml-pipelines.

[9] Rui Abreu, Dave Archer, Erin Chapman, James Cheney, Hoda Eldardiry, and Adrià Gascón. 2016. Provenance Segmentation. In *8th USENIX Workshop on the Theory and Practice of Provenance, TaPP 2016, Washington, D.C., USA, June 8-9, 2016*, Sarah Cohen Boulakia (Ed.). USENIX Association. https://www.usenix.org/conference/tapp16/workshop-program/presentation/abreu

[10] Eleanor Ainy, Pierre Bourhis, Susan B. Davidson, Daniel Deutch, and Tova Milo. 2015. Approximated Summarization of Data Provenance. In *Proceedings of the 24th ACM International Conference on Information and Knowledge Management, CIKM 2015, Melbourne, VIC, Australia, October 19 - 23, 2015*, James Bailey, Alistair Moffat, Charu C. Aggarwal, Maarten de Rijke, Ravi Kumar, Vanessa Murdock, Timos K. Sellis, and Jeffrey Xu Yu (Eds.). ACM, 483–492. https://doi.org/10.1145/2806416.2806429

[11] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 291–300.

[12] Yael Amsterdamer, Susan B. Davidson, Daniel Deutch, Tova Milo, Julia Stoyanovich, and Val Tannen. 2011. Putting Lipstick on Pig: Enabling Database-style Workflow Provenance. *Proc. VLDB Endow.* 5, 4 (2011), 346–357. https://doi.org/10.14778/2095686.2095693

[13] Zhuowei Bao, Susan B. Davidson, Sanjeev Khanna, and Sudeepa Roy. 2010. An optimal labeling scheme for workflow provenance using skeleton labels. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, Ahmed K. Elmagarmid and Divyakant Agrawal (Eds.). ACM, 711–722. https://doi.org/10.1145/1807167.1807244

[14] Louis Bavoil, Steven P. Callahan, Carlos Eduardo Scheidegger, Huy T. Vo, Patricia Crossno, Cláudio T. Silva, and Juliana Freire. 2005. VisTrails: Enabling Interactive Multiple-View Visualizations. In *16th IEEE Visualization Conference, IEEE Vis 2005, Minneapolis, MN, USA, October 23-28, 2005, Proceedings*. IEEE Computer Society, 135–142. https://doi.org/10.1109/VISUAL.2005.1532788

[15] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, et al. 2017. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1387–1395.

[16] Denis Baylor, Kevin Haas, Konstantinos Katsiapis, Sammy Leong, Rose Liu, Clemens Mewald, Hui Miao, Neoklis Polyzotis, Mitchell Trott, and Martin Zinkevich. 2019. Continuous Training for Production ML in the TensorFlow Extended (TFX) Platform. In *2019 USENIX Conference on Operational Machine Learning, OpML 2019, Santa Clara, CA, USA, May 20, 2019*, Bharath Ramsundar and Nisha Talagala (Eds.). USENIX Association, 51–53. https://www.usenix.org/conference/opml19/presentation/baylor

[17] Olivier Biton, Sarah Cohen Boulakia, Susan B. Davidson, and Carmem S. Hara. 2008. Querying and Managing Provenance through User Views in Scientific Workflows. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen (Eds.). IEEE Computer Society, 1072–1081. https://doi.org/10.1109/ICDE.2008.4497516

[18] Eric Breck, Neoklis Polyzotis, Sudip Roy, Steven Whang, and Martin Zinkevich. 2019. Data Validation for Machine Learning. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*, Ameet Talwalkar, Virginia Smith, and Matei Zaharia (Eds.). mlsys.org. https://proceedings.mlsys.org/book/267.pdf

[19] Eugen Cepoi and Liping Peng. 2020. Runway - Model Lifecycle Management at Netflix. USENIX Association.

[20] James Cheney, Laura Chiticariu, and Wang Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Found. Trends Databases* 1, 4 (2009), 379–474. https://doi.org/10.1561/1900000006

[21] Mike Dreves, Gene Huang, Zhuo Peng, Neoklis Polyzotis, Evan Rosen, and Paul Suganthan C. G. 2020. From Data to Models and Back. In *Proceedings of the Fourth Workshop on Data Management for End-To-End Machine Learning, In conjunction with the 2020 ACM SIGMOD/PODS Conference, DEEM@SIGMOD 2020, Portland, OR, USA, June 14, 2020*, Sebastian Schelter, Steven Whang, and Julia Stoyanovich (Eds.). ACM, 1:1–1:4. https://doi.org/10.1145/3399579.3399868

[22] Juliana Freire, David Koop, Emanuele Santos, and Cláudio T. Silva. 2008. Provenance for Computational Tasks: A Survey. *Comput. Sci. Eng.* 10, 3 (2008), 11–21. https://doi.org/10.1109/MCSE.2008.79

[23] Alon Y. Halevy, Flip Korn, Natalya Fridman Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. 2016. Goods: Organizing Google's Datasets. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 795–806. https://doi.org/10.1145/2882903.2903730

[24] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib Analytics Library or MAD Skills, the SQL. *Proc. VLDB Endow.* 5, 12 (2012), 1700–1711. https://doi.org/10.14778/2367502.2367510

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

[26] David A Holland, Uri Jacob Braun, Diana Maclean, Kiran-Kumar Muniswamy-Reddy, and Margo I Seltzer. 2008. Choosing a data model and query language for provenance. In *Proceedings of the 2nd International Provenance and Annotation Workshop (IPAW'08)*. Springer.

[27] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd D. Millstein, and Tyson Condie. 2015. Titian: Data Provenance Support in Spark. *Proc. VLDB Endow.* 9, 3 (2015), 216–227. https://doi.org/10.14778/2850583.2850595

[28] Sean Kandel, Andreas Paepcke, Joseph M Hellerstein, and Jeffrey Heer. 2012. Enterprise data analysis and visualization: An interview study. *IEEE Transactions on Visualization and Computer Graphics* 18, 12 (2012), 2917–2926.

[29] Konstantinos Karanasos, Matteo Interlandi, Doris Xin, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Supun Nakandal, Subru Krishnan, Markus Weimer, et al. 2019. Extending relational query processing with ML inference. *arXiv preprint arXiv:1911.00231* (2019).

[30] Konstantinos Katsiapis and Kevin Haas. 2019. Towards ML Engineering with TensorFlow Extended (TFX). In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis (Eds.). ACM, 3182. https://doi.org/10.1145/3292500.3340408

[31] Christoph Koch, Daniel Lupei, and Val Tannen. 2016. Incremental View Maintenance For Collection Programming. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (San Francisco, California, USA) *(PODS '16)*. Association for Computing Machinery, New York, NY, USA, 75–90. https://doi.org/10.1145/2902251.2902286

[32] Arun Kumar, Robert McCann, Jeffrey F. Naughton, and Jignesh M. Patel. 2015. Model Selection Management Systems: The Next Frontier of Advanced Analytics. *SIGMOD Rec.* 44, 4 (2015), 17–22. https://doi.org/10.1145/2935694.2935698

[33] Angela Lee, Doris Xin, Doris Lee, and Aditya Parameswaran. 2020. Demystifying a Dark Art: Understanding Real-World Machine Learning Model Development. *Workshop on Human-in-the-Loop Data Analytics (HILDA) at the ACM SIGMOD International Conference on Management of Data* (2020).

[34] Edo Liberty, Zohar Karnin, Bing Xiang, Laurence Rouesnel, Baris Coskun, Ramesh Nallapati, Julio Delgado, Amir Sadoughi, Yury Astashonok, Piali Das, Can Balioglu, Saswata Chakravarty, Madhav Jha, Philip Gautier, David Arpin, Tim Januschowski, Valentin Flunkert, Yuyang Wang, Jan Gasthaus, Lorenzo Stella, Syama Rangapuram, David Salinas, Sebastian Schelter, and Alex Smola. 2020. Elastic Machine Learning Algorithms in Amazon SageMaker. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 731–737. https://doi.org/10.1145/3318464.3386126

[35] Hui Miao and Amol Deshpande. 2018. ProvDB: Provenance-enabled Lifecycle Management of Collaborative Data Analysis Workflows. *IEEE Data Eng. Bull.* 41, 4 (2018), 26–38. http://sites.computer.org/debull/A18dec/p26.pdf

[36] Hui Miao and Amol Deshpande. 2019. Understanding Data Science Lifecycle Provenance via Graph Segmentation and Summarization. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 1710–1713. https://doi.org/10.1109/ICDE.2019.00179

[37] Luc Moreau. 2015. Aggregation by Provenance Types: A Technique for Summarising Provenance Graphs. In *Proceedings Graphs as Models, GaM@ETAPS 2015, London, UK, 11-12 April 2015 (EPTCS, Vol. 181)*, Arend Rensink and Eduardo Zambon (Eds.). 129–144. https://doi.org/10.4204/EPTCS.181.9

[38] Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, Beth Plale, Yogesh Simmhan, Eric G. Stephan, and Jan Van den Bussche. 2011. The Open Provenance Model core specification (v1.1). *Future Gener. Comput. Syst.* 27, 6 (2011), 743–756.

https://doi.org/10.1016/j.future.2010.07.005

[39] Luc Moreau, Paolo Missier, Khalid Belhajjame, Reza B'Far, James Cheney, Sam Coppens, Stephen Cresswell, Yolanda Gil, Paul Groth, Graham Klyne, et al. 2013. Prov-dm: The prov data model.

[40] Dan Olteanu. 2020. The Relational Data Borg is Learning. *Proc. VLDB Endow.* 13, 12 (2020), 3502–3515. http://www.vldb.org/pvldb/vol13/p3502-olteanu.pdf

[41] 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 E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[42] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2017. noWorkflow: a Tool for Collecting, Analyzing, and Managing Provenance from Python Scripts. *Proc. VLDB Endow.* 10, 12 (2017), 1841–1844. https://doi.org/10.14778/3137765.3137789

[43] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2017. Data management challenges in production machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1723–1726.

[44] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2018. Data lifecycle challenges in production machine learning: a survey. *ACM SIGMOD Record* 47, 2 (2018), 17–28.

[45] Lukas Rupprecht, James C. Davis, Constantine Arnold, Yaniv Gur, and Deepavali Bhagwat. 2020. Improving Reproducibility of Data Science Pipelines through Transparent Provenance Capture. *Proc. VLDB Endow.* 13, 12 (2020), 3354–3368. http://www.vldb.org/pvldb/vol13/p3354-rupprecht.pdf

[46] Sebastian Schelter, Felix Biessmann, Tim Januschowski, David Salinas, Stephan Seufert, Gyuri Szarvas, Manasi Vartak, Samuel Madden, Hui Miao, Amol Deshpande, et al. 2018. On Challenges in Machine Learning Model Management. *IEEE Data Eng. Bull.* 41, 4 (2018), 5–15.

[47] Sebastian Schelter, Joos-Hendrik Boese, Johannes Kirschnick, Thoralf Klein, and Stephan Seufert. 2017. Automatically tracking metadata and provenance of machine learning experiments. In *Machine Learning Systems workshop at NIPS*.

[48] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. In *Advances in neural information processing systems*. 2503–2511.

[49] Manasi Vartak and Samuel Madden. 2018. MODELDB: Opportunities and Challenges in Managing Machine Learning Models. *IEEE Data Eng. Bull.* 41, 4 (2018), 16–25. http://sites.computer.org/debull/A18dec/p16.pdf

[50] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya Parameswaran. [n.d.]. Helix: Holistic Optimization for Accelerating Iterative Machine Learning. *Proceedings of the VLDB Endowment* 12, 4 ([n. d.]).

[51] Doris Xin, Hui Miao, Aditya Parameswaran, and Neoklis Polyzotis. 2021. Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities. arXiv:2103.16007 [cs.DB]

[52] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. 2018. Accelerating the Machine Learning Lifecycle with MLflow. *IEEE Data Eng. Bull.* 41, 4 (2018), 39–45.

[53] Amy X Zhang, Michael Muller, and Dakuo Wang. 2020. How do data science workers collaborate? roles, workflows, and tools. *Proceedings of the ACM on Human-Computer Interaction* 4, CSCW1 (2020), 1–23.