Towards Observability for Production Machine Learning Pipelines

Shreya Shankar
UC Berkeley
shreyashankar@berkeley.edu

Aditya G. Parameswaran
UC Berkeley
adityagp@berkeley.edu

ABSTRACT

Software organizations are increasingly incorporating machine learning (ML) into their product offerings, driving a need for new data management tools. Many of these tools facilitate the initial development of ML applications, but sustaining these applications post-deployment is difficult due to lack of real-time feedback (i.e., labels) for predictions and silent failures that could occur at any component of the ML pipeline (e.g., data distribution shift or anomalous features). We propose a new type of data management system that offers end-to-end *observability*, or visibility into complex system behavior, for deployed ML pipelines through assisted (1) detection, (2) diagnosis, and (3) reaction to ML-related bugs. We describe new research challenges and suggest preliminary solution ideas in all three aspects. Finally, we introduce an example architecture for a "bolt-on" ML observability system, or one that wraps around existing tools in the stack.

PVLDB Reference Format:
Shreya Shankar and Aditya G. Parameswaran. Towards Observability for Production Machine Learning Pipelines. PVLDB, 15(13): 4015 - 4022, 2022.
doi:10.14778/3565838.3565853

1 INTRODUCTION

Organizations are devoting increasingly more resources towards developing and deploying applications powered by machine learning (ML). ML applications rely on pipelines that span many heterogeneous stages or *components*, such as feature generation and model training, requiring specialized data management tools. Most work in data management for ML concentrates on specific components, e.g., preprocessing [24, 63] or model training [77, 78, 47, 22]. Some industry solutions have also garnered adoption by handling data management issues that stem from model experimentation [82, 10].

However, there are many unaddressed challenges in *sustaining* ML pipelines once built: maintaining, debugging, and improving them after the initial deployment. Various best practices for "production ML" and failure case studies highlight the dire need for ML sustainability [11, 70]. We posit that for sustainability, ML practitioners should be able to (1) detect, (2) diagnose, and (3) react to bugs post-deployment. Compared to traditional software systems, which typically only break when there are infrastructure issues, ML pipelines can also fail unpredictably due to data issues—and therefore are uniquely challenging to sustain in all three aspects.

Bug Detection: Hard Due to Feedback Delays. It is well-known that data distributions change or shift over time, causing model performance to drop [74, 58]. Detecting performance drops post-deployment is challenging due to lack of "ground-truth" data: in many production ML systems, feedback on predictions, or labels, can arrive at a later time. Furthermore, in many pipelines, only a few labels arrive (e.g., labelers manually annotate some predictions, or only a handful of predicted outputs are displayed to the user). As a result, practitioners are unable to monitor simple ML metrics such as accuracy in real-time. As an alternative, end-to-end ML frameworks such as TFX [48] and Sagemaker [37] monitor internal pipeline state or health via distance metrics [44] over distributions of ML features and outputs over time. These proxies often produce too many false positives and thus do not accurately determine when models are underperforming, as we will discuss further in Section 2.

Bug Diagnosis: Hard Due to Pipeline Complexity. Even if a failure is confidently detected, the complex, highly intertwined nature of components in the ML pipeline makes it hard to understand where bugs could lie. For production ML pipelines, "changing anything changes everything (CACE)," causing predictions to vary unpredictably [70]. Consequently, production ML uniquely suffers from silent pipeline bugs, such as corrupted or stale subsets of features. Practitioners painstakingly enumerate and maintain (i.e., tune) data quality constraints for component inputs and outputs [12, 69], motivating automatic specification and maintenance of precise constraints at the component level.

Bug Fixes: Hard Due to No Obviously Correct Answers. Even if users can successfully pinpoint all pipeline bugs, there can be many ways to bring model performance back up to a desirable level, and effectiveness depends on the nature of the data or task. For example, different components, when fixed, can cause different magnitudes of improvement in ML performance. Users often have no sense of what to fix first, relative to resource and time costs.

ML Observability. The challenges outlined above motivate the need for *observability* [73], or "*better visibility into understanding the complex behavior of software using telemetry collected ... at run time*" [32], tailored for ML pipelines. Observability encompasses more than just monitoring predefined metrics for holistic system health—it also allows users to ask questions about how systems historically behaved or perform "needle-in-a-haystack" queries [43].

Contributions. In this paper, we discuss unaddressed research challenges in ML observability as a call-to-arms for the database community to contribute to this nascent research direction. We propose the concept of a "bolt-on" observability system for ML pipelines—one that does not require users to rewrite all their code to use a specific framework. ML application developers assemble their pipelines in an ad-hoc manner employing a myriad of tools along the way, and our bolt-on observability system must interoperate with such heterogeneous pipelines. For example, practitioners may

Proceedings of the VLDB Endowment, Vol. 15, No. 13 ISSN 2150-8097.
doi:10.14778/3565838.3565853

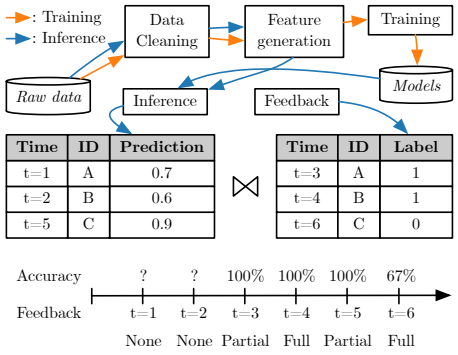

Figure 1: A generic end-to-end ML pipeline. Feedback comes with delay, impacting real-time accuracy.

use a Hive metastore to catalog raw data [75], Deequ for data validation [69], and Weights & Biases for experiment tracking [10].

For our bolt-on observability system to address bug detection, diagnosis, and fixing needs, we propose a three-pronged approach:
(1) Monitoring approximations of top-line, i.e., business-critical, ML metrics to alert users of ML performance drops even when there may not be real-time labels. In Section 3.3, we propose automated techniques that rely on lightweight proxies to bin predictions and estimate metrics based on importance weighting, drawing on the approximate query processing and streaming literature.
(2) Given ML performance drops, identifying issues in inputs and outputs for each component in the pipeline to aid diagnosis. In Section 3.4, we propose logging fine-grained information across provenance snapshots, automatically specifying and tuning data quality constraints, and adversarially learning differences between training and live data to track distribution shift.
(3) Tracing ML bugs back to silent data and engineering-related issues (i.e., pipeline bugs). In Section 3.5, we describe tracking feedback delays and column-wide error scores across dataflow graphs to assist practitioners in repairing broken components.

In Section 3, we present a roadmap (Figure 2) of challenges and preliminary solutions—*detecting* drops in holistic ML metrics (e.g., accuracy), *diagnosing* them by tracking point-in-time, component-level issues, and *reacting* to the drops by analyzing cross-time, cross-component issues. In Section 4, we discuss an example of a bolt-on ML observability system architecture and introduce our vision for MLTRACE, a lightweight bolt-on ML observability tool, which has already received preliminary interest from practitioners with over 400 GitHub stars (`github.com/loglabs/mltrace`).

2 BACKGROUND

We discuss prior work in data management for ML pipelines and current end-to-end ML pipeline frameworks.

ETL, Assertions, and Experiment Tracking. Input data for ML models is typically constructed through a series of ETL workloads. Faulty predictions can stem from such workloads, such as incorrectly performing missing value imputation [68]. Tools like Dagger [63] and mlinspect [24] help practitioners detect data-related bugs in ML preprocessing components. Our focus is instead on bugs that arise post-deployment. Other tools [82, 77, 10] focus on experiment tracking, one of the biggest pain points in generating models for production ML pipelines. However, none of these tools determine *if* and *why* production pipelines are failing.

Data Quality Assertions. Other work [69, 2, 48, 30] proposes libraries of assertions to be embedded in ML code, however, they provide no guidance for which assertions to embed. Additionally, results of these tests must be externally logged with a separate service for users to query post-hoc. While data quality assertions are certainly valuable for catching egregious issues (e.g., negative values for columns that should be positive), ML pipeline performance can drop over time without failing user-embedded assertions.

Detecting data shift. Many papers in the ML literature discuss how various forms of data shift cause model performance to degrade [74, 50, 40]. To address such shift problems, the ML community has proposed monitoring distance metrics across distributions of features and predictions [61]. However, with thousands of features and seasonal changes in data, such methods may not correctly flag shift, might trigger too many alarms and cause alert "fatigue" [12]. Thus, there is a need for higher-precision methods that detect, diagnose, and react to data shift. Some research in post-deployment ML debugging focuses on finding slices (i.e., predicates) where models perform poorly [59, 65], but these methods require labels, which are not always available.

Unresolved Observability Challenges in Existing Tools. End-to-end frameworks such as Sagemaker [37] and TFX [48] provide logging at the component level but only support primitive monitoring based on user-specified metrics and similarly do not help address data shift. These frameworks also force their users to rewrite their pipeline using their DSLs. To avoid having users perform a cumbersome rewrite, some proprietary tools only monitor features and predictions. Other declarative frameworks [64, 49] allow users to declaratively specify end-to-end ML pipelines without supporting the identification of deployment bugs.

3 RESEARCH CHALLENGES

Before we describe our research challenges, we first introduce key definitions and an example ML pipeline (Figure 1).

3.1 ML Pipeline Preliminaries

3.1.1 Definitions. An ML pipeline involves multiple data processing components, leading to one or more ML models that provide *predictions* for a specific *task*. A *metric* is a measure of success for an ML pipeline, such as prediction accuracy. A *tuple* is an individual feature vector used to generate predictions. A *live* prediction is a prediction made after deployment, as opposed to predictions made during training. The consumers of predictions provide *feedback*, or some data that indicates the quality of a prediction (e.g., item selection for recommendations, correctness for binary classification). *Labels*, or "ground-truth" for predictions, are derived from feedback. Finally, we refer to groups of tuples, defined on conjunctions of predicates on features, as *buckets*.

3.1.2 Example ML Pipeline. Using data from the New York City Taxi and Limousine Coalition [3], our ML task is to predict whether a rider will give their driver a tip > 20% of the fare. Predictions are probabilities (i.e., floats between 0 and 1). Each tuple in the dataset (Yellow Trips) represents a single ride, with 17 attributes.

Our ML pipeline includes five components, as described by the rectangular boxes in Figure 1. We have two sub-pipelines—training and inference—that share the cleaning and feature generation components. The pipeline includes one model, an `sklearn` random

	Detect	Diagnose	React
Driving Question	**How** is the pipeline doing in real-time?	**Which** components have errors at a given time?	**What** component fixes help address errors?
Objective	Track **holistic** ML metrics (e.g., accuracy)	Identify **point-in-time component-level** issues	Assist in fixing **cross-component cross-time** issues
	Missing labels (Section 3.3.1)	Logging and provenance (Section 3.4.1)	Unexpected feedback delays (Section 3.5.1)
Challenges	Delayed labels (Section 3.3.2)	Auto-tuned data quality constraints (Section 3.4.2)	Pipeline-wide errors (Section 3.5.2)
		Natural data shift (Section 3.4.3)	

Figure 2: Breakdown of research challenges

forest classifier. The ML pipeline is evaluated on accuracy, or the fraction of correct predictions.

3.1.3 Formalizing Distribution Shift. In deployment settings, real-time accuracy is often difficult to measure due to feedback delays, so practitioners monitor changes, or shifts, in distributions of features and predictions. ML practitioners have introduced any number of types of shifts, such as concept, data, covariate, label, subpopulation, prior probability, and low-data shifts, among others—and these definitions often conflict [74, 50, 40, 67, 80]. If Y is the label space and X is the feature or covariate space (e.g., location of ride, number of passengers), we note that all of the aforementioned shift definitions boil down to *at least one* of the two shift scenarios:

Concept shift: $P(Y|X)$ changes; $P(Y)$ changes but $P(X)$ doesn't
Covariate shift: $P(X)$ and $P(Y)$ change but $P(Y|X)$ doesn't

A concrete example of concept shift is a recession: riders tip less, changing $P(Y)$ but not $P(X)$. A concrete example of covariate shift is New Year's Eve: the number of taxi rides will be relatively higher near Times Square in New York, changing $P(X)$ and $P(Y)$ as a result, even though the nature of a taxi ride that results in a high tip does not change, i.e., $P(Y|X)$. There's ML literature on learning under these natural shifts [21, 41]; however, our research challenges focus on the *unexpected* combinations of shifts that arise in production.

The rationale for tracking $P(Y)$ and $P(X)$ over time in production pipelines is that significant changes in these values can indicate new data quality or engineering bugs that need to be fixed. However, methods to flag changes in distributions, as mentioned in Section 2, cause too many false positive alerts. For example, practitioners compute the K-S test statistic between training and live tuples for *each feature* to approximate how $P(X)$ has changed, often yielding thousands of measures, which can be confusing to navigate. Additionally, on large datasets, p-values can go to zero even without actual significance [38], further exacerbating alert fatigue.

3.2 Research Roadmap

As shown in Figure 2, we employ a three-pronged framework of detecting (Section 3.3), diagnosing (Section 3.4), and reacting (Section 3.5) to bugs in ML pipelines after deployment:

Detection. This prong answers the question *how is the deployed ML pipeline doing in real-time*, with a focus on performance measures such as accuracy. There are two challenges in estimating performance. First, the lack of labels, which happens soon after deployment (Section 3.3.1), and second, labels are available but arbitrarily delayed (Section 3.3.2). In the latter case, estimating real-time performance requires a join across the out-of-sync label and prediction streams, which is difficult at scale.

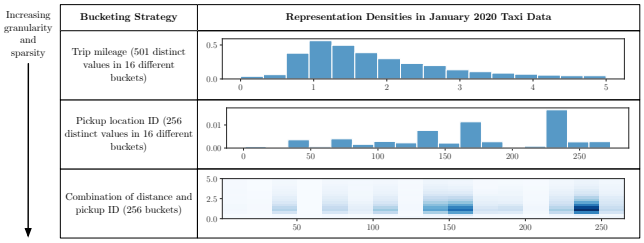

Figure 3: Bucketing strategies, normalized to show bucket density. As buckets become more finer-grained, they also become sparse.

Diagnosis. Given a drop in performance, diagnosis answers the question *which components of the pipeline are potential sources of errors*, with a focus on a single component and a single point in time. To make sense of errors in individual components, we need to log intermediate inputs/outputs and provenance (Section 3.4.1). With this logging in place, we should automatically specify and tune data validation constraints to address spectrum of pipeline bugs, from hard and soft constraint violations (Section 3.4.2), to data shift (Section 3.4.3).

Reaction. With errors in individual components identified, reaction answers the question *what fixes to the pipeline can help address errors*, across components and time. To help fix the pipeline, we need to both make sure that any sources of label lag are addressed (Section 3.5.1)—to ensure that we have better estimates of performance measures, and that the cross-component and cross-time issues are addressed (Section 3.5.2).

3.3 Detecting ML Performance Issues

Post-deployment, the starting point for identifying issues is monitoring drops in ML metrics such as accuracy. This becomes challenging when labels or predictions are delayed or absent. Moreover, delays may not be uniform across buckets (e.g., a power outage in East Village might prevent taxicab meter information from being uploaded). As shown in Figure 1, predictions and feedback arrive at different timestamps and are joined on some identifier. At every timestamp, ML pipelines can move between three feedback scenarios: no feedback, partial feedback, and full feedback. There are two key challenges: first, the lack of labels (impacting the partial and no feedback settings), and second, arbitrary label delays (impacting the partial and full feedback settings). We discuss both of these challenges in turn. We focus on cumulative accuracy, which is the easier case; there are additional challenges in computing accuracy on sliding windows, which we cover in our technical report [72].

3.3.1 The Lack of Labels. After deployment, it is common to either have no labels or only a subset of predictions labeled. These labels may arrive in batches at a later date, possibly after human review, motivating us to still find ways to estimate real-time performance without them. To estimate cumulative accuracy, we may use importance weighting (IW) techniques [74]. We can identify buckets based on input feature combinations, determine the training accuracy for each bucket, and weight these accuracies based on the number of points in each bucket in the live (post-deployment, unlabeled) data. Consider neighborhood as a bucketing strategy: if the training set had FiDi and Midtown accuracies of 80% and 50% respectively and we have 100 FiDi and 500 Midtown live predictions, we can estimate an accuracy of $0.8 \times 100 + 0.5 \times 500 = 55\%$.

There are multiple competing objectives in determining which bucketing strategy would lead to best estimates of accuracy, among the $O(n!)$ possible bucketing schemes, where n is the number of features. Figure 3 illustrates three bucketing schemes. The first couple have representation in each bucket, which gives us some confidence in per-bucket accuracy. However, the last one has some empty buckets: if a live tuple were to be assigned to such a bucket, we would not have an accuracy estimate for it. Finer-grained bucketing schemes may capture patterns not found in coarse-grained ones but could also be more sparse, which can impact the correctness of our accuracy estimates. Moreover, finer-grained bucketing schemes would occupy more space than coarse-grained ones. Beyond *(i) Space* and *(ii) Sparsity*, there are other objectives we need to consider. *(iii) Variance:* buckets should have high variance in training accuracies, *(iv) Predictiveness:* the training accuracy for each bucket should be predictive of actual accuracies for live data in that bucket. Balancing these objectives is non-trivial. We may take inspiration from stratified sampling [56] in Approximate Query Processing (AQP) [6, 5], and also ensemble schemes [36].

Finally, we may gain additional benefits from changing the bucketing strategy in response to live data. To do so, we must efficiently identify buckets in high-dimensional, changing data streams with a reference dataset in mind (i.e., the training set). A starting point could be to extend streaming clustering algorithms that are explicitly robust to changing data distributions [51].

3.3.2 Label Delays. Labels are often delayed in arbitrary ways. So, estimating accuracy, which requires a join between prediction and label/feedback streams, is challenging to do at scale, since keeping both streams in memory is impossible. One option is to uniformly subsample both streams, since the usual problems with sampling over joins [15, 29, 27] don't apply when each prediction tuple joins with precisely one feedback tuple. Since we do not know the size of the streams, one can apply reservoir sampling [7] on both streams using a shared hash function on the common identifier. However, this approach is wasteful—once the pair of prediction and feedback tuples are received, they no longer need to be in memory. Moreover, the quality of the estimate degrades over time since we are maintaining a fixed size sample over growing streams. Ideally we want to maintain both a reservoir (for prediction tuples without feedback) plus partial aggregates (for prediction tuples with feedback). Joined tuples can make way for new slots in the reservoir. However, doing so while respecting reservoir sampling guarantees of each tuple having the same probability of being sampled is non-trivial. For example, the sudden arrival of many feedback tuples can cause multiple vacant slots in the reservoir, increasing the probability for the next prediction tuple to be included in the reservoir.

Beyond labels, another way to approximate ML pipeline performance is to directly monitor changes in an important business metric (e.g., user satisfaction, click-through-rate, revenue). Sometimes the ML metric (e.g., model accuracy) does not align with a business metric (e.g., user satisfaction, revenue), requiring users to rethink the ML objective or discard the model altogther. An engineering challenge is to provide integrations with other system components that are not directly part of the ML pipeline—e.g., sales tools that record metrics like daily active users. To help users understand the effectiveness of their ML models, we can show correlations between ML metrics and business metrics over time.

3.4 Diagnosing ML Performance Issues

After detecting an ML performance drop, we next need to diagnose it by identifying *which* components have bugs at that time. We focus on bugs that arise *after* a pipeline deployment (i.e., related to a mismatch in data between training and serving). There is a spectrum of data-centric ML production bugs [31, 45]: *hard* → *soft* → *drift*, from most to least time-sensitive. Hard errors, such as some data sources failing to ingest and resulting in missing feature values, need immediate attention. Soft errors, such as features having anomalous means, require more tedious manual investigation because of false positives: many columns can deviate significantly while only a few are responsible for pipeline performance drops. Hard and soft errors are both forms of *pipeline errors*, which are often addressed by engineering changes to pipeline components. Finally, bugs can result from natural data drift, causing model performance to slowly decrease; nevertheless, they require attention. Unlike pipeline errors, data drift occurs naturally as data evolves and models no longer faithfully capture the underlying relationships. We discuss the challenges addressing pipeline errors and drift errors next, after discussing a prerequisite: logging and provenance.

3.4.1 Logging and Provenance. Logging at the component level helps us uncover whether the output of a given pipeline component has an error. Then, to trace errors across components, we additionally need provenance. There is extensive work on logging and provenance, e.g., [26, 53, 16, 54, 20], and for ML and data science pipelines [14, 8, 52, 26, 17, 46, 82]. Some approaches require using a specific end-to-end ML framework [8] or logging API [14, 26, 46, 82]. Others instrument the AST or bytecode to capture lineage [24, 25, 17] or employ error-prone static analysis [52], forcing users to remain in a particular language or ML framework. We instead propose a simple bolt-on approach: users annotate pipeline components (e.g., with decorators for Python) with pointers to inputs and outputs (e.g., dataframe variables) that automatically get logged to an observability store.

3.4.2 Tracking Pipeline Errors via Auto-tuned Data Integrity Constraints. ML pipeline errors are typically caught by data validation constraints [12, 24, 68, 4, 57]. For example, Schelter et al. [68] defines 25 different types of single-column ML-specific constraints, and two constraints on column pairs, each requiring tediously setting thresholds per column (or pair). The long-term maintenance of these constraints is also a headache: this includes dealing with alert fatigue or missed constraint violations, and tuning thresholds for each constraint. Automating creation and maintenance for these constraints while preserving high precision (i.e., all violations correspond to bugs) and recall (i.e., all bugs are caught by violations) is an important challenge. Existing solutions suggest basic automatic constraints such as type checks and set membership for categorical columns; although they have decent precision, recall is low [48].

We propose auto-generating suites of input and output validation constraints and auto-tuning them over time to maximize precision and recall of violations. Constraints should be *explainable* and thus more actionable for users, rather than seemingly random float-valued column bounds. A solution idea is to learn an autoregressive model [81] that predicts the likelihood of a column's value for a tuple given values of the other columns; then we can aggregate likelihood scores for each column over sliding windows of post-deployment tuples. One approach is to fit an

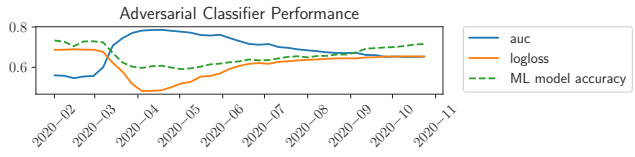

Figure 4: AUC and log loss from the adversarial classifier trained to separate a training data sample and reservoir sample of live tuples.

autoregressive function for each column to model its distribution at time t given the joint distribution of other columns at time t, as well as tuples corresponding to times $< t$. However, this does not scale to more than a handful of columns, so we can use masking techniques to train a single model for all columns [23, 18, 81]. Another challenge is that most off-the-shelf autoregressive models are trained explicitly to predict the next value or token, not to learn the *distributions*, or density functions, of columns. Thus, we need to discretize inputs and outputs so the model learns a distribution of buckets. A simple bucketization strategy can be derived from the CDF (e.g., quantiles), but this will fail as distributions change over time. Alternatively, recent progress in language modeling suggests a different bucketization strategy—numbers can be discretized into digits [79]. To turn the density estimation model(s) into a suite of constraints, users can define a common threshold for columns, e.g., tuples should have an aggregated column likelihood score above 80%. We can also fine-tune the threshold based on the size of the data (i.e., number of columns and tuples) to find a good trade-off between precision and recall of alerts. Further challenges include efficiently maintaining these autoregressive models and their data (e.g., fine-tuning, running across provenance snapshots).

3.4.3 Addressing Natural Data Drift. Data integrity checks do not flag slower, longer-term distribution shift. For instance, a recession could cause riders to tip less across the population, changing $P(Y)$ but not $P(X)$. To approximately compute shifts in $P(X)$ and $P(Y)$, existing work proposes tracking metrics like KL divergence and KS tests [61] between sliding windows in live inference data and train datasets (i.e., for train-serve skew as described in Breck et al. [12]). There are two problems with this approach: (1) it requires the inference and training data to be kept in memory, and (2) it doesn't work well when there are many tuples—p-values go to zero even if shifts aren't significant enough to warrant a retrain, as discussed in Section 3.1.3.

To solve (1), the memory issue, we can leverage a reservoir of live tuples (as in Section 3.3), but it is impractical to keep the entire training set in memory. We can keep a materialized sample of the training set in-memory. To solve (2), the p-value issue, we can draw inspiration from adversarial validation, a method to determine whether train and test sets are drawn from the same distribution [19]. Adversarial validation trains a binary classifier F to predict whether a tuple d came from either the train or test dataset. If F converges to $\sim 50\%$ AUC [39], then one can assume the datasets are similar [55]. We can extend this to track shift: we train F to predict whether d comes from the training sample or the reservoir sample of live data (as in Section 3.3), and log the AUC. However, adapting this method to the streaming setting is computationally challenging because we would need to train a new classifier F every time we log an AUC, and computing AUC requires multiple passes through the data.

One insight is that users only care about how the AUC changes over time, as an increasing AUC indicates that live data is diverging from training set data. As a proxy, we can log $F(d)$'s *loss* over time, which can be computed in a single pass. To avoid frequently retraining $F(d)$ from scratch, every time we get a new tuple in the reservoir sample of live data, we can sample d from the reservoir with $p = 0.5$ and the training set with $p = 0.5$; then, we can fine-tune $F(d)$ on d with stochastic gradient descent. Here, the intuition is that decreases in loss are coupled with increases in AUC, as shown in Figure 4. As loss decreases, it becomes easier to separate the training and live data, indicating distribution shift. The onset of distribution shift as flagged by the adversarial classifier aligns with the beginning of the ML model accuracy drop (late March 2020). The features highly weighted in $F(d)$ are also the ones most likely to be responsible for the shift, further aiding diagnosis.

3.5 Reacting to Bugs in ML Pipelines

Once bugs are isolated, they need to be fixed. Slower distribution shift can be fixed by a retrain, but silent pipeline errors require immediate engineering attention. These pipeline errors require careful analysis across components and time: the challenge is to determine which pipeline errors that, upon fixing, will have the largest positive impact on ML accuracy. Unlike traditional data repair problems [4, 34, 62, 42, 9], where the focus is on cleaning a snapshot of the data, here we want to point users to pipeline errors that, if addressed, can best improve future prediction quality. We discuss two such pipeline errors: reacting to label feedback delays and repairing broken pipeline components.

3.5.1 Reacting to Feedback Delays. Knowing how the distribution of feedback delays changes over time can uncover pipeline errors and enable quick response. Assuming the distribution of label delay is unknown and nonstationary (i.e., we cannot train a separate model to predict which predictions won't have feedback), a challenge lies in identifying groups of tuples that have similar feedback delay times to understand patterns. Most streaming clustering algorithms may not produce groups described with only a few predicates [66]. For debugging purposes, users also care about how these clusters of delayed tuples change over time, or anomalies in delays.

Overall, we want to pick predicate combinations that "cover" all of the tuples that have severe label delays. This is analogous to frequent itemsets [35]; recent work has extended it to work in an approximate setting, while optimizing for metrics like coverage [28]. Unlike that setting, here, we cannot materialize a sample upfront and operate on it; instead, we must operate on a stream directly, and determine what predicate combinations may have high coverage "on the fly". For this, we can draw on incremental maintenance for frequent itemsets [76], however this work focuses on updating itemsets given the addition of new tuples. In our setting some prediction tuples that are missing feedback may have their feedback arrive a bit later than expected. Therefore, we will need to both add and remove tuples and thereby update the counts of the current frequent itemsets during incremental maintenance.

3.5.2 Assisted Repair of Broken Components. Besides feedback, there are two types of pipeline errors that cause performance drops: data staleness (no change) and corruption (unexpected change). A staleness example is if the pipeline to regenerate rider-related features (e.g., historical average tip) broke, forcing reads of old

feature values. A corruption example is if an engineer changed geographical features to read from a better maps API, but the API returned distances in kilometers instead of miles.

A key insight for both types of pipeline errors is that they are caused by *columns*, because columns tend to be outputs of pipeline logic (e.g., creating features) [12]. Quantitative data cleaning techniques from the statistics and database literature typically define units of data to be cleaned as subsets of tuples, not columns [4, 13, 60]. Column-level changes are hard to catch—in our corruption example, a few anomalous trip distances isn't unusual, but all of them suddenly increasing is. We could leverage functional dependency (FD) discovery techniques to identify which columns most violate FDs [33, 62]. However, these are hard to apply in a noisy multivariate setting and consistently tune for production pipelines, especially without prior specifications from users.

To assist repairing broken components, our research question is: what (component, column set) pairs best explain an ML performance drop? We can leverage the auto-tuned constraints from Section 3.4.2 to identify fuzzy changes in behavior at the column level. First, we determine individual column error scores for each component based on its historical behavior; then, we group columns by statistical correlations. Finally, we rank the (component, column set) error scores based on pipeline behavior by aggregating them across the dataflow graph. We discuss all three steps in turn.

To measure column error, we can adapt statistical anomaly detection techniques to track how our auto-tuned data integrity constraints behave over time. Concretely, since we have learned a representation of $\Pr[X]$ for any column X, we fix some scalar estimate of $\Pr[X]$ (e.g., $\mu(\Pr[X])$ or $p25(\Pr[X])$) and derive a time-based anomaly score (e.g., how many standard deviations $\mu(\Pr[X])$ at a given time is away from its rolling, 7-day mean). Columns with an anomaly score ≥ 3 may indicate errors. Note that this approach flags corruption bugs but not staleness bugs, which will have anomaly scores of 0. To formulate staleness as an anomaly detection problem, we derive the time-based anomaly score from a different scalar estimate of $\Pr[X]$, such as the variance of recent means of the column. Consequently, zero-valued historical variances and large-magnitude anomaly scores indicate staleness. We leave further implementation details to the tech report [72]. A practical challenge is that columns are highly correlated, but we can group co-erroneous columns based on correlation. One option is to use probabilistic graphical models [62], but a simpler, non-ML idea is to sample a covariance matrix to represent "distances" between features and apply spectral clustering.

For the last step—ranking errors across all pipeline components—we track error across the dataflow graph. Each node represents a (column group, component) pair. Two nodes share an edge if their corresponding components share an edge in the ML pipeline graph. Each node is initially labeled with its error score ζ. Intermediate nodes with small ζs should deprioritize errors earlier in the pipeline, as our goal is to find nodes that cause high ζ's later in the pipeline. For our ranking algorithm, we can conceptually traverse from the predictions node (i.e., the final component) backwards through all paths $p \in P$ in the graph; at every node i, we record $\phi_i = \max_{p \in P} \prod_j \zeta_j$ for all the nodes j visited so far (including the current node i). Nodes with high ϕ require immediate attention.

4 SYSTEM

A bolt-on ML observability system must be able to compute and store (1) history of and (2) interactions between components, requiring logging state at component runtime. Data and model integrity checks (e.g., expected number of nulls, model assertions [30]) can be programmed as constraints. Metric computation (e.g., approximate accuracy) can run as triggers.

Interface Layer. Users should be able to view real-time pipeline performance (i.e., accuracy) and query fine-grained data summaries, traces for outputs, and other information in component logs.

Execution Layer. The execution layer, which wraps around a component, must be able to run trigger computation such as importance-weighting, executing and fine-tuning data quality constraints, drift detectors, time-based anomaly detection, and other methods described in Section 3. Additionally, the execution layer must identify component dependencies to track provenance for predictions.

Storage Layer. We must store at least three types of data: pointers to inputs and outputs, ML metrics monitored across consecutive runs of the same component (Section 3.3), and logs capturing fine-grained state (provenance, data validation results, and drift detection as described in Section 3.4) every time a component is run. Additionally, the system must keep samples of training sets and live inference tuples in-memory for the execution layer to use while computing fine-grained information (e.g., K-S test results, adversarial classifier weights).

MLTRACE Abstractions. Our bolt-on ML observability system, MLTRACE, will eventually have the following functionality: (1) a library of functions that can support predefined computation before or after component runs for metric calculation or any relevant alerts, triggers, or constraints; (2) automatic logging of inputs, outputs, and metadata at the component run level; and (3) an interface for users to ask arbitrary post-hoc queries about their pipelines. Our current prototype has preliminary approaches for (2) and (3) and we are working on populating our library (1). We provide declarative, client-facing abstractions for users to specify components and the metrics and tests they would like to compute at every run of the component. The current prototype of MLTRACE is publicly available on Github [71] and PyPI [1]. Additional details can be found in our technical report [72].

5 CONCLUSION

We proposed new research challenges in ML observability through a taxonomy of detecting, diagnosing, and reacting to ML bugs, helping make sense of performance with incomplete information, identify potential issues in ML pipeline components, and trace the source of errors. We discussed a high-level architecture of a bolt-on ML observability system, and introduced our prototype, MLTRACE. We call on the database community to contribute to the vision of ML observability by alleviating the many the data management concerns that come with production ML.

ACKNOWLEDGEMENTS

We acknowledge support from grants IIS-2129008, IIS-1940759, and IIS-1940757 awarded by the National Science Foundation, funds from the Alfred P. Sloan Foundation, as well as EPIC lab sponsors: Adobe, Microsoft, Google, and Sigma Computing.

REFERENCES

[1] mltrace.

[2] Welcome to great expectations.

[3] Tlc trip record data, 2020.

[4] Ziawasch Abedjan et al. Detecting data errors: Where are we and what needs to be done? *Proc. VLDB Endow.*, 9(12):993–1004, August 2016.

[5] Swarup Acharya, Phillip B Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. The aqua approximate query answering system. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 574–576, 1999.

[6] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42, 2013.

[7] Charu C Aggarwal. On biased reservoir sampling in the presence of stream evolution. In *Proceedings of the 32nd international conference on Very large data bases*, pages 607–618. Citeseer, 2006.

[8] Pulkit Agrawal, Rajat Arya, Aanchal Bindal, Sandeep Bhatia, Anupriya Gagneja, Joseph Godlewski, Yucheng Low, Timothy Muss, Mudit Manu Paliwal, Sethu Raman, et al. Data platform for machine learning. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1803–1816, 2019.

[9] Leopoldo Bertossi, Solmaz Kolahi, and Laks V. S. Lakshmanan. Data cleaning and query answering with matching dependencies and matching functions. In *Proceedings of the 14th International Conference on Database Theory*, ICDT '11, page 268–279, New York, NY, USA, 2011. Association for Computing Machinery.

[10] Lukas Biewald. Tracking with weights and biases `www.wandb.com/`, 2020.

[11] Eric Breck et al. The ml test score: A rubric for ml production readiness and technical debt reduction. In *Big Data'17*, 2017.

[12] Eric Breck, Marty Zinkevich, Neoklis Polyzotis, Steven Whang, and Sudip Roy. Data validation for machine learning. In *Proceedings of SysML*, 2019.

[13] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3), jul 2009.

[14] Adriane Chapman, Paolo Missier, Giulia Simonelli, and Riccardo Torlone. Capturing and querying fine-grained provenance of preprocessing pipelines in data science. *Proceedings of the VLDB Endowment*, 14(4):507–520, 2020.

[15] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. On random sampling over joins. *SIGMOD Rec.*, 28(2):263–274, jun 1999.

[16] James Cheney, Laura Chiticariu, Wang-Chiew Tan, et al. Provenance in databases: Why, how, and where. *Foundations and Trends® in Databases*, 1(4):379–474, 2009.

[17] Fernando Chirigati, Rémi Rampin, Dennis Shasha, and Juliana Freire. Reprozip: Computational reproducibility with ease. In *Proceedings of the 2016 international conference on management of data*, pages 2085–2088, 2016.

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[19] Carl McBride Ellis. What is adversarial validation?, Jul 2021.

[20] Juliana Freire and Claudio T Silva. Making computations and publications reproducible with vistrails. *Computing in Science & Engineering*, 14(4):18–25, 2012.

[21] João Gama, Indrundefined Žliobaitundefined, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM Comput. Surv.*, 46(4), mar 2014.

[22] Rolando Garcia et al. Hindsight logging for model training. In *VLDB'21*, 2021.

[23] Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. Made: Masked autoencoder for distribution estimation. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 881–889, Lille, France, 07–09 Jul 2015. PMLR.

[24] Stefan Grafberger, Shubha Guha, Julia Stoyanovich, and Sebastian Schelter. Mlinspect: A data distribution debugger for machine learning pipelines. In *SIGMOD'21*, 2021.

[25] Philip J Guo and Margo I Seltzer. Burrito: Wrapping your lab notebook in computational infrastructure. 2012.

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

[27] Dawei Huang, Dong Young Yoon, Seth Pettie, and Barzan Mozafari. Joins on samples: A theoretical guide for practitioners. *Proc. VLDB Endow.*, 13(4):547–560, dec 2019.

[28] Manas Joglekar, Hector Garcia-Molina, and Aditya Parameswaran. Interactive data exploration with smart drill-down. *IEEE Transactions on Knowledge and Data Engineering*, 31(1):46–60, 2017.

[29] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 631–646, New York, NY, USA, 2016. Association for Computing Machinery.

[30] Daniel Kang, Deepti Raghavan, Peter Bailis, and Matei A. Zaharia. Model assertions for monitoring and improving ml models. *ArXiv*, abs/2003.01668, 2020.

[31] Speaker: Partha Kanuparthy, Partha Kanuparthy, Meta, Speaker: Animesh Dalakoti, Animesh Dalakoti, Speaker: Kunal Bhalla, and Kunal Bhalla. Ml monitoring & observability @meta scale, May 2022.

[32] Suman Karumuri, Franco Solleza, Stan Zdonik, and Nesime Tatbul. Towards observability data management at scale. *ACM SIGMOD Record*, 49(4):18–23, 2021.

[33] Solmaz Kolahi and Laks V. S. Lakshmanan. On approximating optimum repairs for functional dependency violations. In *Proceedings of the 12th International Conference on Database Theory*, ICDT '09, page 53–62, New York, NY, USA, 2009. Association for Computing Machinery.

[34] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J. Franklin, and Ken Goldberg. Activeclean: Interactive data cleaning for statistical modeling. *Proc. VLDB Endow.*, 9(12):948–959, aug 2016.

[35] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive data sets*. Cambridge university press, 2020.

[36] Xi Liang, Stavros Sintos, Zechao Shang, and Sanjay Krishnan. *Combining Aggregation and Sampling (Nearly) Optimally for Approximate Query Processing*, page 1129–1141. Association for Computing Machinery, New York, NY, USA, 2021.

[37] Edo Liberty et al. Elastic machine learning algorithms in amazon sagemaker. pages 731–737, 06 2020.

[38] Mingfeng Lin, Henry Lucas, and Galit Shmueli. Too big to fail: Large samples and the p-value problem. *Information Systems Research*, 24:906–917, 12 2013.

[39] Charles X Ling, Jin Huang, Harry Zhang, et al. Auc: a statistically consistent and more discriminating measure than accuracy. In *Ijcai*, volume 3, pages 519–524, 2003.

[40] Zachary Lipton, Yu-Xiang Wang, and Alexander Smola. Detecting and correcting for label shift with black box predictors. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 3122–3130. PMLR, 10–15 Jul 2018.

[41] Jie Lu, Anjin Liu, Fan Dong, Feng Gu, João Gama, and Guangquan Zhang. Learning under concept drift: A review. *IEEE Transactions on Knowledge and Data Engineering*, 31:2346–2363, 2019.

[42] Yuyu Luo, Chengliang Chai, Xuedi Qin, Nan Tang, and Guoliang Li. Visclean: Interactive cleaning for progressive visualization. *Proc. VLDB Endow.*, 13(12):2821–2824, aug 2020.

[43] Charity Majors. Observability: A manifesto, Jul 2021.

[44] Frank J Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 46(253):68–78, 1951.

[45] Mihir Mathur. Full-spectrum ml model monitoring at lyft, Jun 2022.

[46] Hui Miao, Amit Chavan, and Amol Deshpande. Provdb: Lifecycle management of collaborative analysis workflows. In *HILDA'17*, 2017.

[47] Hui Miao, Ang Li, Larry Davis, and Amol Deshpande. Towards unified data and lifecycle management for deep learning. In *ICDE'17*, 2017.

[48] Akshay Naresh Modi et al. Tfx: A tensorflow-based production-scale machine learning platform. In *KDD 2017*, 2017.

[49] Piero Molino, Yaroslav Dudin, and Sai Sumanth Miryala. Ludwig: a type-based declarative deep learning toolbox, 2019.

[50] Jose G. Moreno-Torres, Troy Raeder, Rocío Alaiz-Rodríguez, Nitesh V. Chawla, and Francisco Herrera. A unifying view on dataset shift in classification. *Pattern Recognition*, 45(1):521–530, 2012.

[51] Maryam Mousavi, Azuraliza Abu Bakar, and Mohammadmahdi Vakilian. Data stream clustering algorithms: A review. In *SOCO 2015*, 2015.

[52] Mohammad Hossein Namaki, Avrilia Floratou, Fotis Psallidas, Subru Krishnan, Ashvin Agrawal, Yinghui Wu, Yiwen Zhu, and Markus Weimer. Vamsa: Automated provenance tracking in data science scripts. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1542–1551, 2020.

[53] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R Pocock, Anil Wipat, et al. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[54] Sarah Oppold and Melanie Herschel. Provenance-based explanations: are they useful? In *Proceedings of the 14th International Workshop on the Theory and Practice of Provenance*, pages 1–4, 2022.

[55] Jing Pan, Vincent Pham, Mohan Dorairaj, Huigang Chen, and Jeong-Yoon Lee. Adversarial validation approach to concept drift problem in automated machine learning systems. *ArXiv*, abs/2004.03045, 2020.

[56] Van L Parsons. Stratified sampling. *Wiley StatsRef: Statistics Reference Online*, pages 1–11, 2014.

[57] Clément Pit-Claudel, Zelda E. Mariet, Rachael Harding, and Samuel Madden. Outlier detection in heterogeneous datasets using automatic tuple expansion. 2016.

[58] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. Data management challenges in production machine learning. In *SIGMOD '17*, 2017.

[59] Neoklis Polyzotis, Steven Whang, Tim Klas Kraska, and Yeounoh Chung. Slice finder: Automated data slicing for model validation. In *Proceedings of the IEEE Int' Conf. on Data Engineering (ICDE), 2019*, 2019.

[60] Nataliya Prokoshyna, Jaroslaw Szlichta, Fei Chiang, Renée J. Miller, and Divesh Srivastava. Combining quantitative and logical data cleaning. *Proc. VLDB Endow.*, 9(4):300–311, dec 2015.

[61] Stephan Rabanser, Stephan Günnemann, and Zachary Chase Lipton. Failing loudly: An empirical study of methods for detecting dataset shift. In *NeurIPS*, 2019.

[62] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. Holoclean: Holistic data repairs with probabilistic inference. *Proc. VLDB Endow.*, 10(11):1190–1201, aug 2017.

[63] E. Rezig et al. Dagger: A data (not code) debugger. In *CIDR*, 2020.

[64] Christopher Ré, Feng Niu, Pallavi Gudipati, and Charles Srisuwananukorn. Overton: A data system for monitoring and improving machine-learned products. In *CIDR*, 2020.

[65] Svetlana Sagadeeva and Matthias Boehm. *SliceLine: Fast, Linear-Algebra-Based Slice Finding for ML Model Debugging*, page 2290–2299. Association for Computing Machinery, New York, NY, USA, 2021.

[66] Sandhya Saisubramanian, Sainyam Galhotra, and Shlomo Zilberstein. *Balancing the Tradeoff Between Clustering Value and Interpretability*, page 351–357. Association for Computing Machinery, New York, NY, USA, 2020.

[67] Shibani Santurkar, Dimitris Tsipras, and Aleksander Madry. Breeds: Benchmarks for subpopulation shift. *arXiv: Computer Vision and Pattern Recognition*, 2021.

[68] Sebastian Schelter et al. Automating large-scale data quality verification. In *PVLDB'18*, 2018.

[69] Sebastian Schelter, Philipp Schmidt, Tammo Rukat, Mario Kiessling, Andrey Taptunov, F. Biessmann, and Dustin Lange. Deequ - data quality validation for machine learning pipelines. 2018.

[70] D. Sculley et al. Hidden technical debt in ml systems. In *NIPS*, 2015.

[71] Shreya Shankar. mltrace: Coarse-grained lineage and tracing for machine learning pipelines.

[72] Shreya Shankar and Aditya Parameswaran. Towards observability for machine learning pipelines, 2021.

[73] Cindy Sridharan. *Distributed Systems Observability: A Guide to Building Robust Systems*. O'Reilly Media, 2018.

[74] Masashi Sugiyama et al. Covariate shift adaptation by importance weighted cross validation. In *JMLR*, 2007.

[75] Ashish Thusoo et al. Hive: A warehousing solution over a map-reduce framework. In *VLDB*, 2009.

[76] Mohamed Anis Bach Tobji, Boutheina Ben Yaghlane, and Khaled Mellouli. Incremental maintenance of frequent itemsets in evidential databases. In *ECSQARU*, 2009.

[77] Manasi Vartak. Modeldb: a system for machine learning model management. In *HILDA '16*, 2016.

[78] Manasi Vartak et al. Mistique: A system to store and query model intermediates for model diagnosis. In *SIGMOD '18*, 2018.

[79] Eric Wallace, Yizhong Wang, Sujian Li, Sameer Singh, and Matt Gardner. Do nlp models know numbers? probing numeracy in embeddings. In *EMNLP*, 2019.

[80] Olivia Wiles, Sven Gowal, Florian Stimberg, Sylvestre-Alvise Rebuffi, Ira Ktena, Krishnamurthy Dvijotham, and Ali Taylan Cemgil. A fine-grained analysis on distribution shift. *ArXiv*, abs/2110.11328, 2021.

[81] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. Deep unsupervised cardinality estimation. *Proc. VLDB Endow.*, 13(3):279–292, nov 2019.

[82] M. Zaharia et al. Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull.*, 41:39–45, 2018.