Automatic and Precise Data Validation for Machine Learning

Shreya Shankar*
University of California, Berkeley
shreyashankar@berkeley.edu

Labib Fawaz
Meta
labibfawaz@meta.com

Karl Gyllstrom
Meta
gylls@meta.com

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

ABSTRACT

Machine learning (ML) models in production pipelines are frequently retrained on the latest partitions of large, continually-growing datasets. Due to engineering bugs, partitions in such datasets almost always have some corrupted features; thus, it's critical to find data issues and block retraining before downstream ML accuracy decreases. However, current ML data validation methods are difficult to operationalize: they yield too many false positive alerts, require manual tuning, or are infeasible at scale. In this paper, we present an automatic, precise, and scalable data validation system for ML pipelines, employing a simple idea that we call a Partition Summarization (PS) approach to data validation: each timestamp-based partition of data is summarized with data quality metrics, and summaries are compared to detect corrupted partitions. We demonstrate how to adapt PS for any data validation method in a robust manner and evaluate several adaptations—which by themselves provide limited precision. Finally, we present GATE, our data validation method that leverages these adaptations, giving a 2.1× average improvement in precision over the baseline from prior work on a case study within our large tech company.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**.

KEYWORDS

machine learning; data validation

ACM Reference Format:
Shreya Shankar*, Labib Fawaz, Karl Gyllstrom, and Aditya Parameswaran. 2023. Automatic and Precise Data Validation for Machine Learning. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management (CIKM '23), October 21–25, 2023, Birmingham, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3583780.3614786

1 INTRODUCTION

Errors in input data can negatively impact machine learning (ML) performance, motivating data validation for ML pipelines [6, 8, 25]. Breck et al. [8] state that "the importance of this problem is hard to overstate, especially for production pipelines," where ML models are frequently retrained on the latest partitions of data—including the underlying input features as well as the corresponding predictions [49]. For instance, suppose the audio doesn't work for the newest release of a social media app. Since audio-related features for users who updated their apps would get corrupted, the performance of a news feed ranking model might suffer. Not only would the corrupted partition (i.e., day) of data result in low ML accuracy, but any ML model retrained on this partition would also be corrupted. It's thus crucial to detect data issues before retraining models, but on the flip side, if we falsely alert that there is corrupted data and prevent retraining, production model snapshots quickly become stale. In this paper, we therefore focus on the problem of **automatically and precisely validating data to detect issues in production ML pipelines, *before* models are retrained.**

Method	No Tuning Needed	Handles Correlated Features	Tracks Multiple Metrics	High Precision and Recall
Schelter et al. [39]	✗	✗	✓	✗
Breck et al. [8]	✗	✗	✓	✗
Rabanser et al. [33]	✓	✗	✗	✗
Redyuk et al. [36]	✓	✗	✓	✗
GATE	✓	✓	✓	✓

Table 1: Data Validation for ML Methods.

Detecting data errors in production ML pipelines before they cause downstream accuracy drops is hard at our scale (i.e., a large tech company with hundreds of ML pipelines; details omitted for anonymity). Our data partitions are several petabytes, with tens of thousands of features (i.e., input data to the model) and multiple retraining jobs per model per week. Given so many features, it is almost always the case that engineering bugs outside the model developers' control corrupt some of them—especially when, at many organizations, feature pipelines are separated from calls to ML models [42, 49]. Additionally, it's unclear how much corruption actually impacts model accuracy [32]. This corruption tolerance is often different for each pipeline, requiring pipelines to have on-call engineers that investigate whether production model snapshots are broken and should be rolled back to earlier versions [42]. A data validation system should alert on-call engineers to gate, or block, the promotion of a model snapshot to production, without requiring constant pipeline babysitting.

Prior ML data validation techniques are too manual and coarse-grained. Prior work on ML data validation either overly relies on manual input or doesn't flag most errors. Schelter et al. [39] propose a variety of general-purpose large-scale data validation techniques; however, these techniques aren't operationally scalable because they require engineers to enumerate and fine-tune constraints for each feature. Breck et al. [8] propose *schema validation* techniques for production ML pipelines, where each tuple is checked against a schema, consisting of type and null value checks and loosely-defined bounds (e.g., non-negativity for an age-related feature) inferred from a training set. Schema validation is necessary but not sufficient; our ML pipelines have schema validation but still experience many data corruption issues.

Prior drift detection techniques are too imprecise and don't scale. Another approach is to employ drift detection techniques from the ML literature. A method popularized by Rabanser et al.

*Work performed during an internship at Meta

CIKM '23, October 21–25, 2023, Birmingham, United Kingdom

ACM ISBN 979-8-4007-0124-5/23/10.
https://doi.org/10.1145/3583780.3614786

[33] computes the Kolmogorov-Smirnov test [26] for each feature between their current and historical distributions, and other work has tried different statistical tests [2, 25], but triggering alerts based on distributional differences at scale can be computationally infeasible, as historical data can be too large or blocked for privacy reasons. A simple solution is to keep a rolling sample of historical data to compare the current partition of data to, but prior work finds that this solution leads to too many false positive alerts [8, 14, 39, 40, 45].

Building an automatic, precise, and scalable data validation system is hard. Our work composes a number of simple but powerful insights to build a performant and useful data validation tool.

First, we note that false positive drift alerts are often tied to expected temporal patterns—for example, day-vs-night or weekday-vs-weekend—these, by themselves, are not true instances of ML data errors, but are incorrectly identified by prior approaches because they simply compare the current batch of data with another batch (either all of the prior data, or a rolling window). We propose a new straightforward approach called *Partition Summarization (PS)*, where we compare a summary of the current partition of data to a number of historical summaries. For example, the partition corresponding to Monday is compared not just with Sunday, or a coarse-grained aggregate for the last week, but (say) individual partitions for each day of the week—which includes both weekends and weekdays. Storing summaries of partitions for use in later computation has been used in other data management contexts, e.g., [24, 28], and we apply this idea to ML data validation.

Second, while the PS approach makes sense, it isn't exactly clear how to summarize a partition in an effective manner for validating it relative to previous partitions. Prior literature offers a number of statistical measures to place in the summaries (e.g., column mean, max), but we find that naively adapting and normalizing (e.g., min-max) these statistics, like Redyuk et al. [36] propose for general anomaly detection in datasets, still leads to false positive alerts for ML-related data validation. To solve this problem, we leverage anomaly and change point detection techniques [3, 7] and adapt them to be robust to slower, expected drift over time (e.g., age-related features are monotonically increasing).

Finally, even if we can effectively summarize a partition, we still have an issue of many false positive alerts, because ML datasets can have many correlated features, often because of preprocessing techniques like one-hot encoding. If each partition summary consists of several measures for each feature column, simply applying a dimensionality reduction technique (e.g., PCA [31]) doesn't meet operational requirements because alerts should be actionable. An alert such as "Cluster No. 12 drifted" isn't useful to an on-call engineer. As a result, we propose a scalable, robust-to-noise strategy to group features into disjoint clusters based on their correlations.

All together, we present GATE, a data validation method that improves upon adaptations of aforementioned methods (Table 1) by employing the following steps: *(i)* clustering correlated features based on partition summaries, *(ii)* computing statistical measures of data quality for features in a robust manner, and *(iii)* monitoring cluster-wide aggregate statistics over time.

Outline. To the best of our knowledge, we are the first to propose an automatic, high-precision, and operationally feasible data validation system for ML pipelines. Our system focuses on validating partitions of data for ML pipelines to detect issues before models are retrained. Our contributions are the following:

- We give an overview of our ML pipelines and enumerate business requirements for an automatic data validation system (Section 2),
- We describe a general framework for adapting existing data validation methods to the Partition Summarization (PS) framework, but these methods do not meet our requirements—namely, they still result in too many false positive alerts (Sections 3 and 4),
- We introduce GATE, our new, high-precision and high-recall automatic data validation technique that employs the PS framework (Section 5), and
- We discuss takeaways from a case study on some of our ML pipelines (Section 6). We compare our data validation methods to prior work and discuss preferable techniques for different use cases. Finally, we mention an example of using our system during an on-call rotation to debug an ML pipeline performance drop.

2 BACKGROUND

In this section, we provide background on production ML pipelines, requirements for a practical data validation solution, and a short discussion of existing data validation methods.

2.1 ML Pipelines

A *production ML pipeline* consists of one or more ML models that continuously produce predictions, and are periodically retrained on the latest partition(s) of data. Figure 1 depicts an example production ML pipeline. Traffic to the pipeline is routed to either the main model model A or one of many experimental models (e.g., model B). Suppose an input tuple (① in Figure 1) is routed to model A. Then, ② Model A will compute a prediction (e.g., probability of user action). This prediction is returned to an end user or application and ③ logged to a table with all features and predictions. Separately, every several hours or days (the cadence differs for different models), ④ retraining jobs are launched to retrain each model on the latest partition(s) of tuples—with the exception of a small hold-out set of tuples used for validation. Each retraining job verifies that the newly trained model snapshot achieves good performance (i.e., low loss) on the hold-out set, and if ⑤ this *model validation* step passes, the new model snapshot replaces its corresponding old production snapshot. ML engineers can include additional model-specific checks, like unit tests on specific tuples [22].

While model validation is necessary to prevent bad models from going to production, it is not sufficient. Breck et al. [8] discuss how bugs in code are a common source of production data errors and that the quality of ML predictions drastically suffers if features are corrupted. Thus, it is imperative to also perform *data validation*, in addition to model validation, especially in the continual setting where data is constantly fed back to the model for retraining.

2.2 Data Validation Requirements

Our data validation goal is to determine whether there are enough "invalid" tuples in the latest partitions of training datasets to cause a downstream model performance drop if the retrained model snapshot is pushed to production. Trivially, we could *gate*, or block, all retrained models being pushed to production, but then the model snapshot in production would quickly get stale. Thus, it is important to *precisely* identify when and how the data is corrupted.

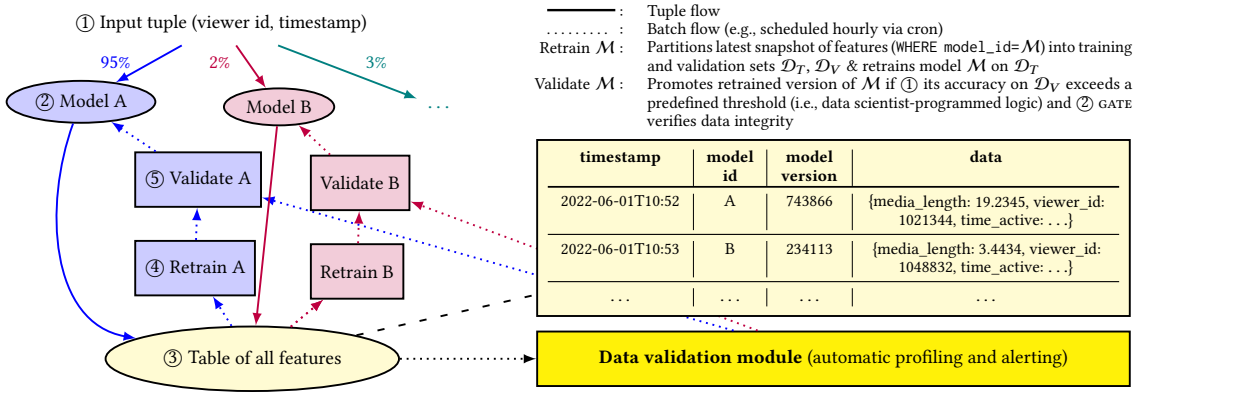


Figure 1: An ML pipeline for a single prediction task, consisting of a main model and many experimental models. If the latest snapshot of features is corrupted, even with model-specific validation, the resulting model version will have poor performance.

For this purpose, prior work recommends employing schema validation and other constraint checking techniques [8, 39]. However, even though these techniques were implemented at our company, they were insufficient, since ML engineers had to enumerate features and constraints for their pipelines, which was too hard to sustain over time as data changed and teams experienced natural turnover. To build a better data validation system, we first interviewed ML engineers to collect the following requirements:

Support for many correlated features. Pipelines consist of datasets that can have tens of thousands of features, many of which are highly correlated and contribute unequally to overall ML model performance. A data validation solution that treats features independently and identically will trigger false positive alerts. While some organizations may be able to weight alerts on features by their importance scores [37], this approach is not generalizable and computationally feasible at scale. Not all models have clearly defined feature importance scores (e.g., in deep learning), and because models are frequently retrained, feature importances change often.

Automated alerts that don't lead to fatigue. Engineers explicitly require a minimum recall (otherwise the system will be useless) and implicitly require a minimum precision (otherwise they will not pay attention to system alerts). In previous efforts to build a data validation tool within the company, engineers had to manually tune thresholds for firing alerts, eventually abandoning the tool. Moreover, ML engineers are often responsible for ML pipelines that consist of features that they don't create (due to organizational turnover), and it's impractical to enumerate and carefully tune constraints for features they may not have context about.

Alert debugging assistance. Alerts should be interpretable for engineers to act on. Since an engineering bug typically breaks a single feature (e.g., a typo corrupts a string-valued feature), alerts should map to a broken feature or set of features [43]. As such, monitoring uninterpretable low-rank representations of features (e.g., from PCA) is not useful.

2.3 Data Validation for ML: Existing Measures

A data validation *method* takes some data quality statistic(s) and triggers an alert if some condition is satisfied.

2.3.1 ML-Relevant Data Quality Statistics. For each feature and prediction column, the following statistics are commonly measured for data cleaning and validation: completeness, mean, standard deviation, number of unique values, number of frequent values, and the count of the most frequently-occurring value [17, 39]. For ML, it's also useful to track how the density of a column changes over time, via histograms or empirical cumulative density functions (eCDFs) [8, 25, 33].

2.3.2 Alert Mechanisms. Several data validation for ML tools separately monitor each column and alert whether the current partition's statistics are anomalous (i.e., outside several deviations from the mean, fail a two-sample statistical test), which can be computationally expensive, require manual tuning, and cause false positive alerts at scale [5, 26, 33, 39]. Another alert mechanism approach is to use anomaly detection models: data quality measures can be concatenated across all columns and fed into a model (e.g., k-nearest neighbors) to predict whether an alert should be triggered [7]. However, such anomaly detection models typically treat features independently and identically [36]—which makes it challenging to correctly trigger alerts without causing alert fatigue in the ML setting, where certain features dominate importance.

Looking Ahead. The rest of the paper introduces our data validation problem setup, the Partition Summarization (PS) framework that creates an automated alert mechanism out of any data quality statistic that can run quickly and cheaply on data of our scale, and GATE, an anomaly detection algorithm with an interpretable clustering component that results in automatic, precise, and scalable data validation.

3 PROBLEM

In this section, we introduce our problem statement and discuss evaluation metrics for data validation methods.

3.1 Formalization

Consider a dataset $\mathcal{D}$ of timestamp-ordered partitions[1] $\mathcal{D} = \{D^1, D^2, \ldots, D^t\}$ where $\mathcal{D}$ has features (i.e., columns) $F_1, F_2, \ldots, F_n$, and F_j^i represents the multiset of values for feature j in the ith partition of data, i.e., D^i. At every timestamp t, an ML model—which we may not know and therefore treat as a black box—is retrained on the partitions until that point, i.e., $\bigcup_{i<t} D^i$. Although the absolute performance of the model can be measured by different metrics, such as accuracy or loss, we generically define an ML *performance drop* as a degradation in the metric of choice compared to a rolling baseline, e.g., 5% drop in accuracy compared to the 7-day rolling average accuracy. Typically, an ML engineer sets this definition and

[1] In this paper, each partition represents a day (i.e., 24 hours) of data, but other breakdowns are possible (e.g., a partition per hour).

threshold. A data partition D^i is *corrupted* if a model retrained on D^i experiences a performance drop at the following timestamp, i.e., $i+1$. We denote $\mathcal{D}$'s corruptions with $y^i \in \{0,1\}$, where $y^i = 1$ if and only if D^i is corrupted.

We then define a *data validation method* to be a function that returns a score indicating the likelihood of a corruption for D^t, given historical partitions $D^1, \ldots, D^{t-1}$. More formally, a data validation method f is defined as $f\left(D^t \mid D^1, D^2, \ldots, D^{t-1}\right)$. If the arguments are clear from the context, we will use the short notation $f(t)$ for a data validation method. Practically, $f(t)$ should be inexpensive to compute—especially at scale, but how do we evaluate its performance, i.e., if $f(t) \approx y^t$? We discuss evaluation metrics next.

3.2 Evaluation Metrics

Typically, data validation methods produce alerts, where an alert is triggered when $f(t)$ exceeds some threshold (e.g., 50%). Given alert threshold τ, we let $a_\tau^t = \mathbb{I}[f(t) \geq \tau]$. We define precision and recall at t:

$$P(t,\tau) = \frac{\sum_{i=1}^t y^i \cdot a_\tau^i}{\sum_{i=1}^t a_\tau^i} \quad R(t,\tau) = \frac{\sum_{i=1}^t y^i \cdot a_\tau^i}{\sum_{i=1}^t y^i}$$

A data validation method may generate at most t distinct scores, one corresponding to each $f(i)$ such that $i \in \{1,\ldots,t\}$, giving us at most t thresholds $\tau_1 \ldots \tau_t$ to choose from. Specifically, we are interested in **precision@0.9**, or the precision at the threshold τ that gives 90% recall. This metric is of primary interest, since a method that cannot recall most failures does not meet the bar for deployment. We also consider another metric—**average precision (AP)**—that takes into account the overall shape of the precision-recall (P-R) curve across thresholds [35]. AP is the weighted mean of precisions at each threshold, where the weight is the increase in recall from the previous threshold. Equation (1) shows the definition of AP at t, if $P(t,\tau_i)$ and $R(t,\tau_i)$ are the precision and recall at the ith threshold τ_i:

$$AP(t) = \sum_{i=1}^{t} \left(R(t,\tau_i) - R(t,\tau_{i-1})\right) P(t,\tau_i) \tag{1}$$

AP captures the overall predictive power of a method, or how the method performs when various thresholds of scores are chosen to fire alerts. As such, AP gives us a holistic, unbiased estimate of a method's performance.

4 PARTITION SUMMARIZATION

Existing data validation setups [8, 25, 39] typically apply some statistical measure Q (e.g., completeness, mean) to each feature F_j at time t and analyze how $Q\left(F_j^t\right)$ compares to $Q\left(F_j^{t-1}\right)$, the statistic for the previous day. Or, they compare $Q\left(F_j^t\right)$ to $Q\left(\cup_{i=1}^{t-1} F_j^i\right)$, i.e., the statistic computed on the union of all previous values[2]. Neither setup accounts for expected temporal patterns, e.g., weekends behaving differently compared to weekdays.

The Partition Summarization (PS) approach to data validation that we introduce, involves first computing one or more statistical summaries, Q, (e.g., mean) for each feature F_j for each partition D^i for $i \leq t$, i.e., $Q\left(F_j^i\right)$. Once these Q's are computed and stored per partition, we then combine them in various ways to compute $f(t)$. Data validation methods can differ in the choice of statistics for partition summaries (i.e., Q) and how to combine these statistics to produce the overall data quality scores (i.e., f). We describe a general adaptation of existing data validation methods to the PS setting in Section 4.1 and give specific adaptations in Section 4.2. We place common notation in Table 2.

4.1 General Adaptation

As we described in Section 3, in the PS framework, we decompose data validation into two steps: one, where we compute summaries Q for features $F_1, F_2, \ldots, F_n$ across partitions $D^1, D^2, \ldots, D^t$; and second, we combine these summaries to get an overall measure of data quality $f(t)$ at each time step t. As a step towards computing $f(t)$, we aggregate each feature's difference between Q and its rolling average $\overline{Q}$. In practice, we need to normalize the differences between Q and $\overline{Q}$ before aggregating them. Formally, given Q and F_i, we define the rolling average $\overline{Q}$ over the last k days as:

$$\overline{Q}(F_i,t) = \frac{1}{k}\sum_{j=1}^{k} Q\left(F_i^{t-j}\right) \tag{2}$$

Once we have rolling average $\overline{Q}$ per feature, we can normalize the Qs per feature as $\widetilde{Q}$. The normalized $\widetilde{Q}$s can be computed using different normalization techniques, such as percent difference (PD) or z-score:

$$\widetilde{Q}_{PD}(F_i,t) = \frac{Q\left(F_i^t\right) - \overline{Q}(F_i,t)}{\overline{Q}(F_i,t)} \tag{3}$$

$$\widetilde{Q}_z(F_i,t) = \frac{Q\left(F_i^t\right) - \overline{Q}(F_i,t)}{\sigma\left(\left[Q\left(F_i^{t-1}\right),\ldots,Q\left(F_i^{t-k}\right)\right]\right)} \tag{4}$$

where σ is a function that calculates the standard deviation. Finally, f can be computed by aggregating the $\widetilde{Q}$'s for each feature, producing a scalar score:

$$f(t) = \frac{1}{n}\sum_{i=1}^{n} \widetilde{Q}(F_i,t)$$

We can trigger an alert if $f(t)$ exceeds a threshold—which can be determined by keeping track of f for a few timestamps and computing if the current value is an outlier. In practice, any scalar outlier detection mechanism can be used (e.g., multiple standard deviations ≥ the mean, ≥ than 95th percentile). An alert threshold chosen this way is robust to temporal variation (i.e., a percentile threshold will maintain its meaning and significance over time). In the following, we slightly abuse notation to adapt our approach to various data quality metrics: briefly, $\overline{Q}$ indicates a rolling average measure, $\widetilde{Q}$ indicates a normalized version (using $\overline{Q}$) for the current time step relative to others, and $f(t)$ aggregates $\widetilde{Q}$ across features.

4.2 Adaptations of Existing Approaches

We pick three categories of data quality measures to adapt to the PS setting: (1) monitoring the percent drop of completeness for each feature, (2) monitoring z-scores of any scalar statistic (e.g., completeness, mean) for each feature, and (3) monitoring p-values from two-sample statistical tests (e.g., Kolmogorov-Smirnov [26]) for each feature. For each of the methods, we set $k = 7$, or one week,

[2]Here and elsewhere, when we use ∪, we are referring to the multiset union rather than the set union.

Symbol	Description
D^t	Partition of data at timestamp t
y	Binary labels representing whether partitions are corrupted, $y^t = 1$ if D^t is corrupted and 0 otherwise
F	A feature (i.e., column) in the dataset; F_j represents the jth feature
f	Data validation method, $f : D^t \to \mathbb{R}$
Q	Data quality statistic (e.g., mean, completeness), $Q : F^t \to \mathbb{R}$
$\overline{Q}$	Average of the most recent statistical measures Q for a feature, defined in Equation (2)
$\widetilde{Q}_{PD}$	Percent difference between Q and its rolling average $\overline{Q}$, defined in Equation (3)
$\widetilde{Q}_z$	z-score difference (i.e., number of standard deviations) between Q and its rolling average $\overline{Q}$, defined in Equation (4)
C	Completeness of a feature for a partition (i.e., the fraction of non-null values), $C : F^t \to [0, 1]$
G^t	GATE's clustering assignment at time t, where $G^t(F)$ represents F's cluster and $\lvert G^t \rvert = v$ is the number of distinct clusters

Table 2: Notation Table

in computing $\overline{Q}$ and $\widetilde{Q}$. We chose $k = 7$ because of typical organizational considerations (e.g., on-call rotation lengths are based on the week, meetings are often weekly), but this parameter can vary.

4.2.1 δ-Completeness Drop. The completeness drop method creates a score at time t based on the number of features that have experienced a completeness drop (i.e., $\widetilde{C}$) $\geq \delta$ with respect to their rolling 7-day average completeness. The score is weighted by each feature's rolling average completeness (i.e., $\overline{C}$). Equation (5) shows the δ-completeness drop score for time t:

$$\widetilde{C}_{PD}(F_i, t, \delta) = \begin{cases} \widetilde{C}_{PD}(F_i, t) \times \overline{C}(F_i, t) & \text{if } \widetilde{C}_{PD}(F_i, t) \geq \delta \\ 0 & \text{otherwise} \end{cases}$$

$$f_C(t, \delta) = \frac{1}{n} \sum_{i=1}^{n} \left| \widetilde{C}_{PD}(F_i, t, \delta) \right| \tag{5}$$

We weight each feature exceeding δ by its rolling average $\overline{C}$ because we only want features to contribute to f_C when they deviate significantly *and* typically have large completeness values.

4.2.2 z-Score Anomaly Detection. In this method, we fix a scalar statistic (e.g., completeness or mean) and create a score at time t based on the fraction of features that have a z-score outside some cutoff τ. z-scores are computed using 7-day rolling means and standard deviations of the statistic. Given statistic Q, the anomaly detection score for time t is shown in Equation (6):

$$\widetilde{Q}_z(F_i, t, \tau) = \begin{cases} \left| \widetilde{Q}_z(F_i, t) \right| & \text{if } \left| \widetilde{Q}_z(F_i, t) \right| \geq \tau \\ 0 & \text{otherwise} \end{cases}$$

$$f_z(t, \tau) = \frac{1}{n} \sum_{i=1}^{n} \widetilde{Q}_z(F_i, t, \tau) \tag{6}$$

Intuitively, we zero the contributing z-score for features with small z-scores (i.e., $< \tau$) because, given the tens of thousands of features we must monitor, many features are likely to have normal z-scores (i.e., less than 3). We want the "most anomalous" features to significantly alter the resulting f_z score—for example, one feature's z-score being 15 is more alarm-worthy than 15 features each having a z-score of 1. Before adding the intermediate $\widetilde{Q}_z(F_i, t, \tau)$ step (i.e., the indicator for $\geq \tau$), this method performed very poorly.

4.2.3 Two-Sample Statistical Tests. To consider entire *distributions* of each feature, we evaluated p-values based on three different statistical measures: Kolmogorov-Smirnov (KS), Wasserstein-1 (wass) or Earth-Mover's Distance, and DTS [13, 26, 46]. At a high level, the KS measure computes the largest difference between two eCDFs at a single value, the Wasserstein-1 measure computes the entire area of difference between two eCDFs, and the DTS measure weights the Wasserstein-1 measure by variance of the combined eCDFs (denoted as $\hat{D}$ in Equation (9)). For our implementation, the eCDFs consist of 99 percentiles, or the 1st percentile to 99th percentile. If $\mathbf{eCDF}(F_i, t) \in \mathbb{R}^{99}$ represents an eCDF of feature F_i at time t, the two-sample test statistics are shown in Equations (7) to (9):

$$\mathbf{diff}(F_i, t) = \left| \mathbf{eCDF}(F_i, t) - \frac{1}{7} \left(\sum_{j=1}^{7} \mathbf{eCDF}(F_i, t - j) \right) \right|$$

$$h_{\text{KS}}(F_i, t) = \max \mathbf{diff}(F_i, t) \tag{7}$$

$$h_{\text{wass}}(F_i, t) = \sum \mathbf{diff}(F_i, t) \tag{8}$$

$$\hat{D}(F_i, t) = \sigma \left(\mathbf{eCDF}(F_i, t), \frac{1}{7} \left(\sum_{j=1}^{7} \mathbf{eCDF}(F_i, t - j) \right) \right)^2$$

$$h_{\text{DTS}}(F_i, t) = \frac{\sum \mathbf{diff}(F_i, t)}{\hat{D}(F_i, t) \times \left(1 - \hat{D}(F_i, t) \right)} \tag{9}$$

Note that $\mathbf{diff}(F_i, t) \in \mathbb{R}^{99}$ represents an element-wise absolute value difference between the eCDF at time t and the 7-day rolling average eCDF. To convert a statistical measure to a p-value, we bootstrap estimates of the measure while randomly partitioning the current and historical average eCDFs and compute the fraction of bootstrapped values that don't exceed the original measure.

To convert the p-values to a scalar score for t, we compute the fraction of features with a p-value smaller than a significance level α, typically set to 0.05. If $p(F_i, t)$ is a p-value from a two-sample statistical test measure at time t for feature F_i, the resulting score for this measure at time t is shown in Equation (10):

$$\widetilde{Q}_{\text{twosamp}}(F_i, t, \alpha) = \mathbb{I}[p(F_i, t) \leq \alpha]$$

$$f_{\text{twosamp}}(t) = \frac{1}{n} \sum_{1=1}^{n} \widetilde{Q}_{\text{twosamp}}(F_i, t, \alpha) \tag{10}$$

5 GATE

In this section, we describe GATE, our technique that results in fewer false positives than the methods from Section 4.1, while being a bit more involved to implement. GATE can be broken down into the following components (as depicted in Figure 2):

Decorrelation. We apply spectral decomposition to correlations between features. This allows us to map clusters back to original features for debugging (unlike PCA). The decorrelation step accepts input partitions D^i where $i \leq t$ and outputs $G^t \in \mathbb{R}^n$, where each feature is represented by only one cluster. We denote the number of unique clusters $\lvert G^t \rvert = v$. For speed and scalability, we apply this clustering method to partition summaries instead of entire underlying datasets. (Section 5.1)

Anomaly Matrix Creation. We compute 6 statistical measures for each feature and derive their z-scores for normalization purposes. We use $G^t \in \mathbb{R}^n$ to cluster and average the z-scores. This

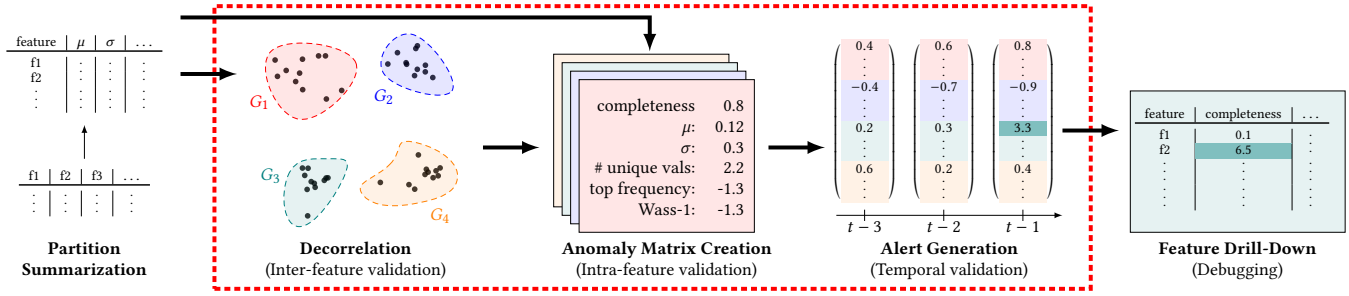


Figure 2: Architecture of our Partition Summarization (PS) approach to data validation for ML. The red box represents a general data validation technique. We found that existing techniques did not achieve high precision and recall in identifying our ML performance drops—mainly, there were too many false positive alerts. Our technique, GATE, clusters correlated features and triggers an alert if an entire group's summary statistics are anomalous.

step returns $X(t) \in \mathbb{R}^{t \times 6v}$, where each row in $X(t)$ represents a measure of data quality for that timestamp. (Section 5.2)

Alert Generation. We compute the average distance from $X_t(t)$ (the tth row in X) to a fraction of the closest previous rows and trigger an alert if the average distance exceeds some threshold. We can filter on recent timestamps if desired. This alert generation step accepts $X(t) \in \mathbb{R}^{t \times 6v}$ from the previous step, or the concatenated anomaly matrices from previous partitions, and outputs a corruption score. We normalize statistics to be z-scores, with respect to recent partition summaries, before comparing them. (Section 5.3)

Optional Feature Drill-Down. While this step is not critical for alerting, ML engineers want to understand why an alert was triggered. We find the most anomalous clusters and "drill-down" into their features and corresponding anomalous statistics. (Section 5.4)

5.1 Decorrelation

In this clustering step, we clustered information from the partition summaries, not the raw data itself, as it's computationally impractical to perform clustering on raw data. We leveraged a graph theoretic approach: nodes were features, and edge distances corresponded to the correlations between endpoint nodes (i.e., features) [47]. First, for each of our 6 summary statistics (described further in Section 5.2), we computed a covariance matrix across features using our summary table. To determine the number of clusters $|G^t|$, we ran PCA on each covariance matrix to determine the number of components $v_1, \ldots, v_6$ that would explain 95% of variance. We set $|G^t| = \max v_1, \ldots, v_6$. Then we summed the absolute values of the covariance matrices to yield the graph edge matrix. Finally, we ran a spectral clustering algorithm to partition the graph into cluster assignments G^t, where each feature corresponded to one cluster. We denote the cluster for a feature F as $G^t(F)$.

There are a number of clustering methods in the literature [50]; here, we picked spectral clustering for its simplicity and application to the graph setting. While spectral clustering has some drawbacks—it doesn't work with noisy data and has a high runtime complexity [18]—our technique doesn't face these problems because our summaries are aggregations and are thus less likely to be noisy, and the size of the covariance matrix (i.e., graph) is based on the number of features, not the number of tuples in the dataset.

5.2 Anomaly Matrix Creation

The goal of this step is to turn D^t into a vector of statistics to compare against previous partitions' vectors. For our algorithm, we used the following six statistics: (1) completeness, (2) mean, (3) standard deviation, (4) number of unique values, approximated via sketches (5) top frequency (i.e., count of the most frequently-occurring value divided by total count) and (6) Wasserstein-1 (i.e., Earth-Mover's) distance between consecutive partitions' eCDFs. To normalize statistics, we turn each statistic into a z-score. Then, we reduce dimensionality by averaging z-scores across clusters. The clustering step (Section 5.1) weights features unequally—based on correlations—for the alert generation step (Section 5.3).

More formally, given clustering assignment G^t from Section 5.1, $|G^t| = v$ distinct clusters, and statistics $Q_1, Q_2, \ldots, Q_6$, at time t, we average the normalized statistics $\widetilde{Q}_z$ within clusters to get Equation (11):

$$\begin{aligned}
\widetilde{\mathbf{Q}}_z(F_i, t) &= \left[\left|\widetilde{Q1}_z(F_i, t)\right|, \ldots, \left|\widetilde{Q6}_z(F_i, t)\right|\right] && \in \mathbb{R}^6_+ \\
\widetilde{\mathbf{Q}}_G(j, t) &= \frac{\sum_{i=1}^{n} \mathbb{I}\left[G^t(F_i) = j\right] \cdot \widetilde{\mathbf{Q}}_z(F_i, t)}{\sum_{i=1}^{n} \mathbb{I}\left[G^t(F_i) = j\right]} && \in \mathbb{R}^6_+ \\
\mathbf{x}(t) &= \left[\widetilde{\mathbf{Q}}_G(1, t)\ \widetilde{\mathbf{Q}}_G(2, t)\ \ldots\ \widetilde{\mathbf{Q}}_G(v, t)\right] && \in \mathbb{R}^{6v}_+ \qquad (11)
\end{aligned}$$

Note that $\mathbb{I}\left[G^t(F) = j\right]$ is a binary function that returns 1 if F is in the jth cluster and 0 otherwise, as determined by G^t. Now, we compute $X(t)$, a matrix representing $\mathbf{x}$ for current and historical partitions. Each row in $X(t)$ corresponds to a partition, and columns represent concatenated, normalized statistics vectors (across features and time).

$$X(t) = [\mathbf{x}(1), \mathbf{x}(2), \ldots, \mathbf{x}(t-1), \mathbf{x}(t)] \qquad \in \mathbb{R}^{t \times 6v}_+ \quad (12)$$

Note that the same clustering, i.e., G^t, must be applied to each partition, or it won't make sense to compare rows in $X(t)$.

5.3 Alert Generation

Given $X(t) \in \mathbb{R}^{t \times 6v}$ from the previous step, we compare rows to determine whether the data at time t should trigger an alert. Intuitively, each row, i.e., $X_i(t)$, represents the quality of D^i. We fix some neighbor fraction f, and for each row $i \in t$, we compute the average distance from $X_t(t)$ to the closest $\lfloor f \times t \rfloor$ non-anomalous

neighbors, or preceding rows, according to Equation (13):

$$\mathbb{I}_y(t) = \begin{cases} 1 & \text{if } y^t = 1 \\ \infty & \text{otherwise} \end{cases}$$
$$\mathbf{d}(t) = \left[\mathbb{I}_y(i) \cdot \|X_t(t) - X_i(t)\|_p \text{ for } i \in [1, \ldots, t-1]\right] \quad \in \mathbb{R}_+^{t-1}$$
$$f_{\text{GATE}}(t) = \frac{1}{\lfloor f \times t \rfloor} \sum \min_{\lfloor f \times t \rfloor} \mathbf{d}(t) \quad \in \mathbb{R}_+ \tag{13}$$

We experimented with setting $p = 1$ (Manhattan distance) and $p = 2$ (Euclidean distance) for p in Equation (13). Similar to the adapted methods described in Section 4.1, we trigger an alert if $f_{\text{GATE}}(t)$ exceeds a threshold, which can be determined by keeping track of f_{GATE} for a few timestamps and determining if the current value of f_{GATE} is an outlier. Our alert threshold is interpretable: one can multiply a "maximum allowed" z-score by the number of clusters (i.e., v) to get an estimate of how close two rows in X^t can be to each other.

5.4 Debugging: Feature Drill-Down

Although the drill-down component is not a step in our automated data validation approach, GATE is designed to help practitioners debug alerts by drilling into relevant clusters' features. First, we rank the clusters by their corresponding $\widetilde{\mathbf{Q}}_G$ in $\mathbf{x}$ (Equation (11)). Then, for top clusters, we identify the features in each cluster using G^t and similarly rank features by the magnitude of their $\widetilde{\mathbf{Q}}_z$. The top-ranked features are the most anomalously behaving features, by definition of z-score (i.e., larger absolute value z-scores are more standard deviations away from the mean), and can be presented to ML engineers. Another useful debugging strategy is to compare $X_t(t)$ to anomalous partitions, or $X_i(t)$ for $i < t$ where $y^i = 1$. If $X_t(t)$ is a seasonal anomaly, such as Thanksgiving and Christmas, the neighboring anomalous $X_i(t)$ may also correspond to a holiday, since each partition is independently summarized. We give an anecdote of the drill-down component in Section 6.3.2.

6 CASE STUDY

In this section, we report a case study on two ML pipelines powering large and business-critical recommender systems in our company. First, we present precision@0.9 and AP numbers for PS methods described in Sections 4 and 5, as well as their runtimes. We choose parameter values based on business considerations (e.g., normalizing statistics with respect to rolling 7-day aggregations of data). Then, we discuss the usability of our techniques.

6.1 Setup

6.1.1 Dataset Descriptions. Each of the two datasets in our case study included one month of partitions (i.e., $D^1, D^2, \ldots, D^t$ where $t \approx 30$) and tens of thousands of features (i.e., $F_1, F_2, \ldots, F_n$ where $n \geq 20,000$). A month of data includes temporal shifts, like weekend vs weekdays. We denote the specific datasets as $\mathcal{D}_1$ and $\mathcal{D}_2$ with feature sets $\mathcal{F}_1$ and $\mathcal{F}_2$ respectively. $\mathcal{D}_1$'s minimum date partition is June 5, 2022 and maximum date partition is July 5, 2022. $\mathcal{D}_2$'s minimum date partition is July 15, 2022 and maximum date partition is August 15, 2022. Both datasets $\mathcal{D}_1$ and $\mathcal{D}_2$ share some (but not all) features—concretely, $\mathcal{F}_1 \neq \mathcal{F}_2$ and $\mathcal{F}_1 \cap \mathcal{F}_2 \neq \emptyset$. For privacy and legal reasons, we cannot disclose details of the features, but the datasets include a mix of integer, float, and categorical features.

As discussed in Section 2.1, models in the ML pipelines are frequently and automatically trained on fresh views of $\mathcal{D}$ and chained together to make final float-valued predictions, which represent the probability of a user clicking on a media recommendation (binary classification). We collaborated directly with on-call engineers to identify the types of ML performance drops they want the data validation system to flag. We found three categories:

(1) **Increased model loss**: Whether the (normalized) cross-entropy loss on live predictions for an ML model (trained on $\mathcal{D}$) increased by some predefined threshold from its rolling 7-day average cross-entropy loss
(2) **Uncalibrated predictions:** Whether the error for a calibration function (fit on ML predictions to align them with true events) is larger than a predefined threshold
(3) **Label shift:** Whether the number of clicks (i.e., positive labels) decreased by some predefined threshold from its rolling 7-day average click rate

For each $\mathcal{D}$, we unioned all occurrences of the three failure events to get ground truth labels. $\mathcal{D}_1$ and $\mathcal{D}_2$ have significantly different failure rates (i.e., fraction of positives), as shown in Table 3. $\mathcal{D}_2$ has an unusually high failure rate, so it will be easier for a method to achieve good precision and recall.

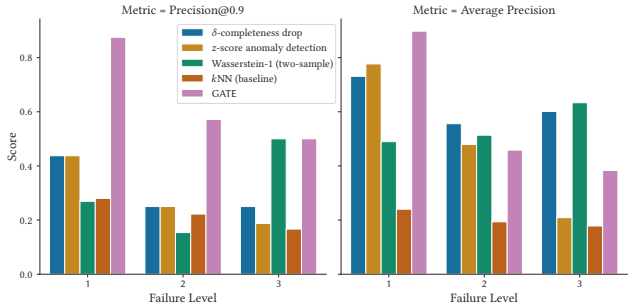
Finally, as mentioned in Section 2, ML teams at our organization care about the tunability of data validation methods, or the ability to adapt the method to different magnitudes of failures (i.e., model performance drops for the ML pipeline). As such, we came up with three different *failure levels* for each dataset or ML pipeline, as shown in Table 3. Thresholds for each of the definitions increase such that the number of failures in FL_i is $\geq$ the number of failures in FL_{i+1}. These FLs were validated by an on-call ML engineer.

FL	Definition	$\mathcal{D}_1$	$\mathcal{D}_2$
1	($\uparrow$ cross-entropy $\geq$ 0.02) $\cup$ (calibration error $\geq$ 0.3) $\cup$ (label shift $\geq$ 0.3)	0.33	0.59
2	($\uparrow$ cross-entropy $\geq$ 0.04) $\cup$ (calibration error $\geq$ 0.4) $\cup$ (label shift $\geq$ 0.4)	0.23	0.47
3	($\uparrow$ cross-entropy $\geq$ 0.06) $\cup$ (calibration error $\geq$ 0.5) $\cup$ (label shift $\geq$ 0.5)	0.17	0.44

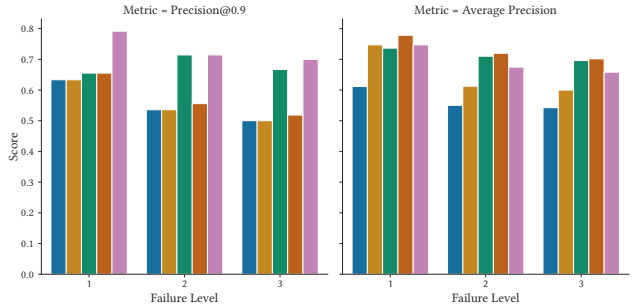
Table 3: Fraction of positive labels (i.e., failures) for each failure level (FL) in $\mathcal{D}_1$ and $\mathcal{D}_2$.

6.1.2 Data Validation Methods. We evaluated a k-nearest neighbor baseline against our adaptations described in Section 4 (δ-completeness drop, z-score anomaly detection, Wasserstein-1 two-sample statistical tests) and GATE. Our baseline comes from Redyuk et al. [36], which targets validation for general data ingestion (as opposed to ML data validation) by creating a vector of statistics for each time step and performing a k-nearest neighbor algorithm against historical vectors to label the current time step's vector as anomalous or acceptable. We use the 7 statistics from the paper: completeness, approximate count of distinctive values, ratio of the most frequent value, maximum, mean, minimum, and standard deviation. For each feature, we compute these statistics and concatenate the results to get the larger vector used in the k-nearest neighbors algorithm. We call this method kNN (baseline) in Figure 3.

6.1.3 Architectural Setup. Due to privacy and legal considerations, we discuss only high-level details. Raw data (i.e., $\mathcal{D}$) for each dataset is stored in a data warehouse system. Upon arrival of a new partition, a PS job is launched: in this job, both ML model performance (e.g., accuracy or loss) and summary statistics are computed for that



(a) Results for $\mathcal{D}_1$, where there are only a few failures. Most ML pipelines we've observed have similarly low failure counts.



(b) Results for $\mathcal{D}_2$, where there are many failures (> 50%). Our methods achieve higher precision@0.9 and comparable AP to the baseline [36].

Figure 3: Case study results across datasets and failure levels. The legend is shown in Figure 3a.

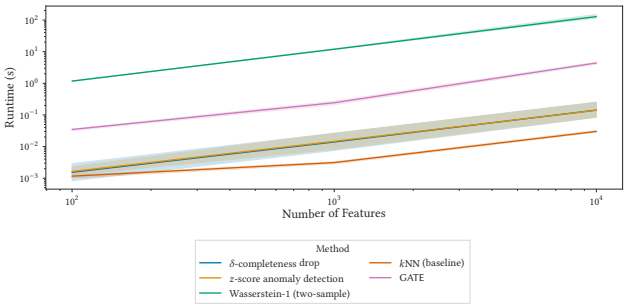


Figure 4: Runtimes of computing a single partition's score, using three trials. The δ-completion drop and z-score anomaly detection methods have similar runtimes.

partition and logged to a summary store. The job's queries are written in PrestoSQL [41]. Each data validation method reads from the same summary store to compare summaries and trigger alerts. The data validation methods are implemented in Python, Pandas, and `scikit-learn`. These jobs run for every partition summary, and the evaluation metrics are computed with `scikit-learn` functions.

6.2 Performance Results

For $\mathcal{D}_1$, our best method gave a 2.9× average improvement in precision@0.9 and 2.3× average improvement in AP over the baseline. For $\mathcal{D}_2$, where there were more failures, our best method gave a 1.3× improvement in precision@0.9 while maintaining 0.9× the AP as the baseline. We observed a reasonable runtime of approximately 5 seconds (50 seconds with the offline clustering step). In Figure 3, we present results for each method listed in Section 6.1. In Figure 4, we show the runtimes to compute a single partition's score for each method. We discuss how to select between methods for different ML pipeline settings in Section 6.3.1.

6.2.1 Precision@0.9. In $\mathcal{D}_1$, GATE significantly outperformed other methods in precision@0.9. For FLs 1 and 2 in $\mathcal{D}_1$, GATE achieved approximately 2× the precision@0.9 as other methods. For FL 3, GATE's precision@0.9 was matched by the Wasserstein-1 two-sample test. In $\mathcal{D}_2$, GATE achieved the highest precision@0.9 in all three FLs. The Wasserstein-1 method tied GATE's performance for FL 2 and came close for FL 3. In $\mathcal{D}_2$, where there were more failures, GATE's improvement in precision@0.9 was not as pronounced; still, it had the highest precision@0.9 across all failure levels. GATE has high precision due to its clustering step, which prevents correlated features from triggering many false positive alarms. The Wasserstein-1 two-sample test achieves high precision when there are fewer failures (i.e., higher FLs) because in the case of a severe failure, a large fraction of records are corrupted, significantly perturbing entire distributions of features. Density measures (e.g., Wasserstein-1 distance) can capture this shift more precisely than point statistics (e.g., mean), which aren't robust to outliers.

6.2.2 Average Precision (AP). While AP is a less important metric than precision@0.9 for us, we found that our methods' APs were better than the baseline in $\mathcal{D}_1$ and nearly matched the baseline in $\mathcal{D}_2$. In $\mathcal{D}_2$, where there were significantly more failures, the baseline from Redyuk et al. [36] achieved the highest AP for all FLs, but GATE achieved 90% of that, and the Wasserstein-1 two-sample test achieved 97% of that. Still, we found our methods' APs acceptable in this setting, since most ML pipelines don't experience such a high failure rate (where it is easier to achieve good precision).

6.2.3 Runtime. The baseline is the fastest method because there is no time-based normalization. The two-sample tests (highest AP) have the highest runtime because they require bootstrapping to yield p-values. They are not practical for more than $O(10,000)$ features. GATE has the second-largest runtime, but its runtime is only a few seconds for $O(10,000)$ features. Although GATE includes the Wasserstein-1 distance measure, it is faster than the two-sample tests because it does not bootstrap a p-value.

6.3 Discussion

Here, we discuss methods to choose for different ML pipeline settings and GATE's feature drill-down component.

6.3.1 Best PS Methods for Different Settings. If a data validation system needs to have high precision@0.9, we learned that different settings have different choices of best PS methods:

Low failure rates. When pipelines have frequently-corrupted data, the corruptions are, by definition, less anomalous—rendering anomaly detection techniques like z-score computation useless. The Wasserstein-1 test (i.e., capturing the distribution of a feature) alone is the best single metric to use, but it is computationally expensive as it requires bootstrapping to get a p-value. GATE has higher precision, includes the Wasserstein-1 distance, and is orders of magnitude less computationally expensive (Figure 4). Most of our ML pipelines had relatively low failure rates, making GATE a good overall choice. However, in pipelines with high failure rates, especially if AP matters more than precision@0.9, Redyuk et al. [36]'s method might be preferable.

Many correlated features. GATE significantly outperforms other methods in terms of mitigating false positives when ML pipelines have many correlated features because of GATE's clustering component (Section 5.1), which only triggers an alert when an entire

group of correlated features is anomalous. In our collaborations with other teams, we observed that many ML pipelines had hundreds, if not thousands, of correlated features. As it is the case in many other organizations [42], data scientists frequently create correlated features based on session data—for example, there might be binary features for watching a video after 1 second, 2 seconds, 5 seconds, and more. However, in pipelines with a few features, a simple, fast method like δ-completion drop might be preferable.

First week of deployment. In the beginning of deployment, nearest-neighbor methods generated alerts in a seemingly random way because there wasn't enough data for comparing partitions. Since most pipelines' failure rates are typically less than 1 in 7, it took at least a week for GATE to start producing meaningful alerts. In the first week, other methods (e.g., δ-completeness drop) might be more useful to trigger alerts, since the alert criteria is typically more meaningful (e.g., drop of 30%, z-score ≥ 3, p-value < 0.05).

High-cardinality features. GATE worked well with features that spanned a large distribution of values. In settings with only binary or low-cardinality (i.e., spanning only a few distinct values) features, data quality metrics like top-k values or number of unique values might, on their own, precisely flag anomalies. However, with real-valued or other high-cardinality features, such metrics might fluctuate even if the distribution remains similar. In practice, many of our ML pipelines have several high-cardinality features, such as user and content embeddings [10, 30].

6.3.2 On-Call Experiences. In this case study, GATE reduced alert fatigue during on-call rotations by having fewer false positive alerts while maintaining acceptable recall requirements. Although GATE was primarily used to anticipate ML performance drops before they showed up in performance dashboards, the feature drill-down component (Section 5.4) could also be used during on-call rotations to diagnose why there was a performance drop. In one scenario, an ML pipeline's performance had significantly dropped, and several days had passed without identifying the root cause. Since there were more than 10,000 features in this model, most of the features had at least one anomalous data quality statistic, making it impossible to figure out the group of broken features. Upon looking at the feature names in the most anomalous cluster identified by GATE, it was clear that there was a sound-related bug, since many of the features were related to audio and had anomalous Wasserstein-1 values.

7 RELATED WORK

GATE combines insights from anomaly detection, clustering, data cleaning, and data validation.

Anomaly Detection and Clustering for Large Datasets. Chandola et al. [9] enumerate challenges of adapting the notion of an anomaly as the underlying distribution of data changes over time. While estimation models (e.g., z-score, Median Absolute Deviation) based on a threshold are commonly used to estimate point outliers [7, 27], our problem is more similar to finding a subsequence of outliers. We draw inspration from dissimilarity approaches like k-nearest neighbors [4, 7, 34], but our challenge is to handle correlated features. Several papers discuss anomaly detection in the light of high-dimensional data [19, 21, 38]. Methods like principal component analysis (PCA) produce low-rank representations of the data, which yield alerts interpretable at the tuple level, not the feature or column level (a requirement given by our engineers in Section 2.2). Zhang et al. [52] introduce clustering large datasets using partitions of data, which we apply to data validation. We summarize partitions and cluster the summaries.

Data Cleaning for ML. ML training sets require clean data [11, 16, 23, 48]. Most cleaning tools require manual input, such as verifying tuples predicted to be outliers, limiting their scalability [12, 23, 44, 48, 51]. Abedjan et al. [1] discuss four categories of data errors when cleaning large-scale datasets: outliers, duplicates, rule violations, and pattern violations. The measures used in our algorithm span these categories. Overall, data cleaning methods don't apply to our setting because we don't want to clean all corrupted tuples (our pipelines already have high-performing ML models). Rather, we simply wish to block the retraining of models on corrupted data. Moreover, at our scale, the challenge is in identifying corruptions that actually cause model performance drops, not any corruption.

Data Validation for ML Pipelines. Data in ML pipelines must be validated, otherwise model performance can suffer [8, 32, 39]. Several research projects and open-source software libraries try to propose solutions or investigate the problem. Biessmann et al. [6] discuss four dimensions of data validation (DV): correctness, consistency, completeness, and statistical properties, and many DV methods monitor these statistics. Existing work—such as Deequ [40], TFX [5], DataSentinel [45], and DaQL [14]—defines constraints for pipeline inputs and outputs, however users must specify constraint values (e.g., completeness bounds). Data "linting" tools typically perform type checks, duplicate detection, and outlier detection based on a fixed number of standard deviations away from the mean but do not tie directly to downstream ML model performance [14, 20]. A broader survey of DV tools [15] finds that most tools require a "gold standard" of data—which often doesn't exist in most production settings. Lwakatare et al. [25] find that, in practice, most engineering teams ignore data validation alerts. For instance, two-sample statistical tests based on differences between distributions (e.g., Wasserstein) commonly trigger many false positive alerts [26, 29, 46]. Lwakatare et al. [25] say that the alerts did not provide actionable feedback and required too much manual maintenance . Redyuk et al. [36] introduce an automated k-nearest neighbors approach to find anomalous partitions of data in a general data ingestion context. Our method is specific to ML pipelines.

8 CONCLUSION

In this paper, we discussed automatically validating data before downstream ML model performance drops occur. We described the Partition Summarization (PS) approach to data validation, where summaries of timestamped partitions are compared to determine anomalous partitions. We introduced a general adaptation for existing data validation methods to the PS setting and GATE, our method that produces high-precision alerts without manual tuning from engineers. Finally, we discussed our learnings from implementing automatic data validation in production ML pipelines.

Acknowledgments. We thank the anonymous reviewers for their valuable feedback. We acknowledge support from grants DGE-2243822, IIS-2129008, IIS-1940759, and IIS-1940757 awarded by the National Science Foundation, an NDSEG Fellowship, funds from the Alfred P. Sloan Foundation, as well as EPIC lab sponsors: G-Research, Adobe, Microsoft, Google, and Sigma Computing. The content is solely the responsibility of the authors and does not necessarily represent the official views of the funding agencies and organizations.

REFERENCES

[1] Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. 2016. Detecting Data Errors: Where Are We and What Needs to Be Done? *Proc. VLDB Endow.* 9, 12 (aug 2016), 993–1004. https://doi.org/10.14778/2994509.2994518

[2] Samuel Ackerman, Eitan Farchi, Orna Raz, Marcel Zalmanovici, and Parijat Dube. 2020. Detection of data drift and outliers affecting machine learning model performance over time. *arXiv preprint arXiv:2012.09258* (2020).

[3] Samaneh Aminikhanghahi and Diane J Cook. 2017. A survey of methods for time series change point detection. *Knowledge and information systems* 51, 2 (2017), 339–367.

[4] Fabrizio Angiulli and Clara Pizzuti. 2002. Fast Outlier Detection in High Dimensional Spaces. In *PKDD*.

[5] 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.

[6] Felix Biessmann, Jacek R. Golebiowski, Tammo Rukat, Dustin Lange, and Philipp Schmidt. 2021. Automated Data Validation in Machine Learning Systems. *IEEE Data Eng. Bull.* 44 (2021), 51–65.

[7] Ane Blázquez-García, Angel Conde, Usue Mori, and Jose A Lozano. 2021. A review on outlier/anomaly detection in time series data. *ACM Computing Surveys (CSUR)* 54, 3 (2021), 1–33.

[8] Eric Breck, Marty Zinkevich, Neoklis Polyzotis, Steven Whang, and Sudip Roy. 2019. Data Validation for Machine Learning. In *Proceedings of SysML*. https://mlsys.org/Conferences/2019/doc/2019/167.pdf

[9] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly Detection: A Survey. *ACM Comput. Surv.* 41, 3, Article 15 (jul 2009), 58 pages. https://doi.org/10.1145/1541880.1541882

[10] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. 2016. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*. 7–10.

[11] Xu Chu, Ihab F Ilyas, Sanjay Krishnan, and Jiannan Wang. 2016. Data cleaning: Overview and emerging challenges. In *Proceedings of the 2016 international conference on management of data*. 2201–2206.

[12] Xu Chu, John Morcos, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Nan Tang, and Yin Ye. 2015. KATARA: A Data Cleaning System Powered by Knowledge Bases and Crowdsourcing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1247–1261. https://doi.org/10.1145/2723372.2749431

[13] Connor Dowd. 2020. A New ECDF Two-Sample Test Statistic. *arXiv: Methodology* (2020).

[14] Lisa Ehrlinger, Verena Haunschmid, Davide Palazzini, and Christian Lettner. 2019. A DaQL to Monitor Data Quality in Machine Learning Applications. In *DEXA*.

[15] Lisa Ehrlinger and Wolfram Wöß. 2022. A survey of data quality measurement and monitoring tools. *Frontiers in big data* (2022), 28.

[16] Venkat Gudivada, Amy Apon, and Junhua Ding. 2017. Data quality considerations for big data and machine learning: Going beyond data cleaning and transformations. *International Journal on Advances in Software* 10, 1 (2017), 1–20.

[17] Joseph M. Hellerstein. 2008. Quantitative Data Cleaning for Large Databases.

[18] Ellen Hohma, Christian M. M. Frey, Anna Beer, and Thomas Seidl. 2022. SCAR: Spectral Clustering Accelerated and Robustified. *Proc. VLDB Endow.* 15, 11 (sep 2022), 3031–3044. https://doi.org/10.14778/3551793.3551850

[19] Ling Huang, XuanLong Nguyen, Minos Garofalakis, Michael Jordan, Anthony Joseph, and Nina Taft. 2006. In-network PCA and anomaly detection. *Advances in neural information processing systems* 19 (2006).

[20] Nick Hynes, D. Sculley, and Michael Terry. 2017. The Data Linter: Lightweight Automated Sanity Checking for ML Data Sets. http://learningsys.org/nips17/assets/papers/paper_19.pdf

[21] Firuz Kamalov and Ho Hon Leung. 2020. Outlier Detection in High Dimensional Data. *Journal of Information & Knowledge Management* 19, 01 (mar 2020), 2040013. https://doi.org/10.1142/s0219649220400134

[22] Daniel Kang, Deepti Raghavan, Peter Bailis, and Matei Zaharia. 2020. Model Assertions for Monitoring and Improving ML Models. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 481–496. https://proceedings.mlsys.org/paper/2020/file/a2557a7b2e94197ff767970b67041697-Paper.pdf

[23] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J Franklin, and Ken Goldberg. 2016. Activeclean: Interactive data cleaning for statistical modeling. *Proceedings of the VLDB Endowment* 9, 12 (2016), 948–959.

[24] Kaiyu Li and Guoliang Li. 2018. Approximate Query Processing: What is New and Where to Go? *Data Science and Engineering* 3 (2018), 379 – 397.

[25] Lucy Ellen Lwakatare, Ellinor Rånge, Ivica Crnkovic, and Jan Bosch. 2021. On the experiences of adopting automated data validation in an industrial machine learning project. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 248–257.

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

[27] Saeed Mehrang, Elina Helander, Misha Pavel, Angela Chieh, and Ilkka Korhonen. 2015. Outlier detection in weight time series of connected scales. In *2015 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, 1489–1496.

[28] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Very Large Data Bases Conference*.

[29] Alfred Müller. 1997. Integral Probability Metrics and Their Generating Classes of Functions. *Advances in Applied Probability* 29, 2 (1997), 429–443. http://www.jstor.org/stable/1428011

[30] Shimei Pan and Tao Ding. 2019. Social media-based user embedding: A literature review. *arXiv preprint arXiv:1907.00725* (2019).

[31] Matthew Partridge and Rafael A Calvo. 1998. Fast dimensionality reduction and simple PCA. *Intelligent data analysis* 2, 3 (1998), 203–214.

[32] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2018. Data Lifecycle Challenges in Production Machine Learning: A Survey. *SIGMOD Rec.* 47, 2 (dec 2018), 17–28. https://doi.org/10.1145/3299887.3299891

[33] Stephan Rabanser, Stephan Günnemann, and Zachary Lipton. 2019. Failing loudly: An empirical study of methods for detecting dataset shift. *Advances in Neural Information Processing Systems* 32 (2019).

[34] Sridhar Ramaswamy, Rajeev Rastogi, and Kyuseok Shim. 2000. Efficient Algorithms for Mining Outliers from Large Data Sets. *SIGMOD Rec.* 29, 2 (may 2000), 427–438. https://doi.org/10.1145/335191.335437

[35] Zhu M Recall. 2004. Precision and average precision. *Department of Statistics and Actuarial Science. University of Waterloo, Waterloo* (2004).

[36] Sergey Redyuk, Zoi Kaoudi, Volker Markl, and Sebastian Schelter. 2021. Automating Data Quality Validation for Dynamic Data Ingestion. In *EDBT*.

[37] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. Model-agnostic interpretability of machine learning. *arXiv preprint arXiv:1606.05386* (2016).

[38] Peter J Rousseeuw and Mia Hubert. 2011. Robust statistics for outlier detection. *Wiley interdisciplinary reviews: Data mining and knowledge discovery* 1, 1 (2011), 73–79.

[39] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. 2018. Automating Large-Scale Data Quality Verification. *Proc. VLDB Endow.* 11, 12 (aug 2018), 1781–1794. https://doi.org/10.14778/3229863.3229867

[40] Sebastian Schelter, Philipp Schmidt, Tammo Rukat, Mario Kiessling, Andrey Taptunov, Felix Biessmann, and Dustin Lange. 2018. DEEQU - Data Quality Validation for Machine Learning Pipelines.

[41] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. 2019. Presto: SQL on everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1802–1813.

[42] Shreya Shankar, Rolando Garcia, Joseph M. Hellerstein, and Aditya G. Parameswaran. 2022. Operationalizing Machine Learning: An Interview Study. https://doi.org/10.48550/ARXIV.2209.09125

[43] Shreya Shankar and Aditya Parameswaran. 2021. Towards Observability for Machine Learning Pipelines. *arXiv preprint arXiv:2108.13557* (2021).

[44] Michael Stonebraker and Ihab F. Ilyas. 2018. Data Integration: The Current Status and the Way Forward. *IEEE Data Eng. Bull.* 41 (2018), 3–9.

[45] Arun Swami, Sriram Vasudevan, and Joojay Huyn. 2020. Data Sentinel: A Declarative Production-Scale Data Validation Platform. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1579–1590. https://doi.org/10.1109/ICDE48307.2020.00140

[46] SS Vallender. 1974. Calculation of the Wasserstein distance between probability distributions on the line. *Theory of Probability & Its Applications* 18, 4 (1974), 784–786.

[47] Scott White and Padhraic Smyth. 2005. A spectral clustering approach to finding communities in graphs. In *Proceedings of the 2005 SIAM international conference on data mining*. SIAM, 274–285.

[48] Weiyuan Wu, Lampros Flokas, Eugene Wu, and Jiannan Wang. 2020. Complaint-driven training data debugging for query 2.0. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1317–1334.

[49] 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* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2639–2652. https://doi.org/10.1145/3448016.3457566

[50] Rui Xu and Donald Wunsch. 2005. Survey of clustering algorithms. *IEEE Transactions on neural networks* 16, 3 (2005), 645–678.

[51] Mohamed Yakout, Ahmed K. Elmagarmid, Jennifer Neville, Mourad Ouzzani, and Ihab F. Ilyas. 2011. Guided Data Repair. *Proc. VLDB Endow.* 4, 5 (feb 2011), 279–289. https://doi.org/10.14778/1952376.1952378

[52] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. 1996. BIRCH: An Efficient Data Clustering Method for Very Large Databases. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data* (Montreal, Quebec, Canada) *(SIGMOD '96)*. Association for Computing Machinery, New York, NY, USA, 103–114. https://doi.org/10.1145/233269.233324