Predicting Node Failures in an Ultra-large-scale Cloud Computing Platform: an AIOps Solution

YANGGUANG LI and ZHEN MING (JACK) JIANG, York University, Canada
HENG LI and AHMED E. HASSAN, Queen's University, Canada
CHENG HE and RUIRUI HUANG, Alibaba Group, China
ZHENGDA ZENG, MIAN WANG, and PINAN CHEN, Alibaba Group, China

Many software services are nowadays hosted on cloud computing platforms, like Amazon EC2, due to many benefits like reduced operational costs. However, node failures in these platforms can impact the availability of their hosted services and potentially lead to large financial losses. Predicting node failures before they actually occur is crucial as it enables DevOps engineers to minimize their impact by performing preventative actions. However, such predictions are hard due to many challenges like the enormous size of the monitoring data and the complexity of the failure symptoms. AIOps, a recently introduced approach in DevOps, leverages data analytics and machine learning (ML) to improve the quality of computing platforms in a cost-effective manner. However, the successful adoption of such AIOps solutions requires much more than a top-performing ML model. Instead, AIOps solutions must be trustable, interpretable, maintainable, scalable, and evaluated in context. To cope with these challenges, in this paper we report our process of building an AIOps solution for predicting node failures for an ultra-large-scale cloud computing platform at Alibaba. We expect our experiences to be of value to researchers and practitioners, who are interested in building and maintaining AIOps solutions for large-scale cloud computing platforms.

CCS Concepts: • **Computing methodologies** → **Machine learning**; • **Software and its engineering** → *Software verification and validation*; *Operational analysis*.

Additional Key Words and Phrases: AIOps, cloud computing, failure prediction, ultra-large-scale platforms

ACM Reference Format:
Yangguang Li, Zhen Ming (Jack) Jiang, Heng Li, Ahmed E. Hassan, Cheng He, Ruirui Huang, Zhengda Zeng, Mian Wang, and Pinan Chen. 2020. Predicting Node Failures in an Ultra-large-scale Cloud Computing Platform: an AIOps Solution. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2020), 24 pages. https://doi.org/10.1145/3385187

1 INTRODUCTION

Cloud computing is now ubiquitous. IDG [20] reports that 73% of enterprises leverage the cloud in their IT infrastructure. On the one hand, cloud computing provides benefits like lower infrastructure costs and high elasticity. On the other hand, failures in the cloud are difficult to detect and can be quite costly when they occur. Such failures are estimated to

Authors' addresses: Yangguang Li, liyg525@gmail.com; Zhen Ming (Jack) Jiang, zmjiang@cse.yorku.ca, York University, Department of Electrical Engineering & Computer Science, Toronto, ON, Canada; Heng Li, hengli@cs.queensu.ca; Ahmed E. Hassan, ahmed@cs.queensu.ca, Queen's University, Software Analysis and Intelligence Lab (SAIL), Kingston, ON, Canada; Cheng He, hecheng.hc@alibaba-inc.com; Ruirui Huang, ruirui.huang@alibaba-inc.com, Alibaba Group, Hangzhou, Zhejiang, China; Zhengda Zeng, dave.zzd@alibaba-inc.com; Mian Wang, eefee.wm@alibaba-inc.com; Pinan Chen, pin-an.cpa@alibaba-inc.com, Alibaba Group, Hangzhou, Zhejiang, China.

Manuscript submitted to ACM

cost $700 billion yearly [30]. To ensure the quality of services, companies set up complex and elaborate monitoring infrastructures, which generate an enormous amount of data to track systems' health. However, it is very challenging to effectively transform such monitoring data into actionable insights due to the size and complexity of such data.

To cope with this challenge, AIOps (Artificial Intelligence for IT Operations) leverages data analytics and machine learning (ML) techniques to assist DevOps engineers to improve the quality of computing platforms in a cost-effective manner. Gartner [34] projects that by 2022, 40% of global enterprises will have strategically implemented AIOps solutions to support their IT operations. Recent AIOps research efforts at top software engineering and other venues (e.g., [13, 19, 23–25, 29]) have demonstrated the benefits of AIOps solutions in real-world settings. However, the successful adoption of such AIOps solutions requires much more than a top-performing ML model. Instead, AIOps solutions must be:

- **Trustable [12]**: AIOps solutions must incorporate years of field-tested engineer-trusted domain expertise into their ML models, instead of simply employing sophisticated models on raw data.
- **Interpretable [31]**: AIOps solutions need to be interpretable even if at the cost of lower performance. Such interpretable models enable DevOps engineers to reason about model recommendations, to gain upper management support for following such recommendations, and, more importantly, enabling DevOps engineers to improve the status quo (e.g., by improving and optimizing their monitoring solutions).
- **Maintainable [36]**: AIOps solutions need to require minimal maintenance and fine-tuning, since DevOps engineers are usually not ML experts, who are already overcommitted on many company-wide ML-initiatives.
- **Scalable [35]**: AIOps solutions need to be scalable and efficient, as they must analyze the monitoring data from thousands to millions of nodes.
- **In-context Evaluation [24]**: AIOps solutions must be evaluated in a context that resembles their actual production usage. Traditional ML evaluation techniques (e.g., cross-validation) are rarely applicable, as they do not consider the real-life peculiarities.

In this paper, we report our process in building an AIOps solution for SystemX, an ultra-large-scale cloud computing platform at Alibaba. SystemX is deployed over tens of thousands of connected nodes across Alibaba's data centers and used by millions of users every day. Each compositing node has different configurations (e.g., CPU, memory, virtual machines) and installs a variety of software applications (e.g., databases and machine learning platforms). Our AIOps solution predicts potential node failures in the near future and leaves DevOps engineers enough time to react to the potential node failures. Once our solution predicts a node failure in the near future, DevOps engineers would perform preventative actions (e.g., live migration) to minimize the impact of node failures on the production system.

Our AIOps solution is a pipeline that leverages readily-available alerts to provide actionable suggestions for DevOps engineers. Such alerts are already used and trusted by DevOps engineers and capture years of their experience with the cloud platform. The alerts are generated by sifting through terabytes of raw monitoring data using rules derived from years of domain expertise and experience from prior issues. Our solution, which is a data analytics pipeline combining innovative feature engineering and data sampling techniques with carefully tuned machine learning models, is interpretable and easy to maintain. Experiments show that our solution performs well in a production-like context. Our AIOps solution is already adopted and used in practice to analyze the monitoring data from tens of thousands of nodes of SystemX's deployment.

The main contributions of this paper are:

(1) This is the first work to discuss and evaluate the needed criteria for the successful adoption of AIOps solutions.

(2) Experiments show that our AIOps solution, which is a data analytics pipeline combining random forest-based modeling with innovative feature engineering and data sampling techniques, outperforms a state-of-the-art AIOps solution [24] in terms of prediction performance while requiring significantly less computational power.
(3) Given the extreme skew of our data, with node failures being extremely rare, we proposed a novel oversampling technique that leverages data from failed nodes prior to their failure as well as multiple samples from normal nodes. This technique improves the performance of our approach and the prior state-of-the-art approach.
(4) We proposed a novel evaluation technique, which examines the effectiveness of our AIOps solutions in a production-like usage context instead of using traditional performance evaluation techniques for ML models.
(5) The lessons that we learned from building and deploying this AIOps pipeline will be of great value for researchers and practitioners, who are interested in ensuring the adoption of their AIOps solutions.

Paper Organization. The rest of the paper is organized as follows: Section 2 provides background information and presents related works. Section 3 presents an exploratory study of the characteristics of the alert data. Section 4 provides an overview of our AIOps solution. Sections 5, 6 and 7 explain our implementation details and present our evaluation results. Section 8 presents the lessons learned and Section 9 discusses the threats to validity. Section 10 concludes the paper.

2 BACKGROUND AND RELATED WORKS

This section first provides some background information about SystemX, its monitoring infrastructure and its monitoring data (Section 2.1). Then we discuss two areas of prior work related to our problem context: prior research in the area of AIOps (Section 2.2) and in evaluating the performance impact of various modeling decisions (Section 2.3).

2.1 SystemX

Since failures can happen at different levels (hardware, operating system, container or VM, and application levels), cloud service providers (e.g., Microsoft [32] and IBM [45]) usually install a variety of probes to ensure Quality of Service (QoS). Some probes passively collect resource usage data (e.g., CPU and memory) or performance measures (e.g., response time and throughput), whereas others proactively check the health of the system by periodically performing heartbeats [16] or sanity checks (e.g., [2]). SystemX is a large-scale cloud computing platform at Alibaba used by millions of users every day. Its deployment spans over tens of thousands of nodes. The size of these raw monitoring data is extremely large (in the range of tens of terabytes every day).

SystemX has already supported a range of different fault tolerance techniques to ensure extremely high levels of reliability and robustness. Although node failures are extremely rare (e.g., less than one in thousands of nodes fails daily), such failures still severely impact end-users whenever they occur. In order to catch early warning signs of these failures, there are additional applications that scan through the monitoring data and proactively generate alerts. Some of these alerts are based on thresholds on the collected system behavior data (e.g., CPU higher than some pre-determined threshold) or error logs (e.g., certain types of errors appear too frequently within sometime range). The goal of such alerts is to provide early warnings, so that preventative actions (e.g., VM migration [7] or software rejuvenation [3]) can be performed to mitigate or minimize the impact of such failures. Other systems have similar alert mechanisms, but they might use different terms like incident reports [9, 10, 27, 28] or tickets [44, 45].

Effectively leveraging such alert data is a major challenge for the DevOps engineers at SystemX. On the one hand, these alerts, which were generated by customized rules based on their own domain expertise and years of experience

from prior issues, contain helpful hints for potential node failures. On the other hand, alerts can be generated not only from the nodes that would fail in the near future, but also from healthy nodes, as some of the issues reported by the alerts can be transient (e.g., network congestions or low memory availability) and can be recovered afterward. However, the very high volume and the intensity of these alerts make it challenging for DevOps engineers to manually distinguish between transient issues and the issues that would cause node failures.

Hence, the goal of this paper is to leverage AI techniques to predicts node failures in the near future such that DevOps engineers can take preventative actions to minimize the impact of node failures on the production system. We do not distinguish the types of node failures nor do we concern about the root causes of node failures in this paper. Once the potential failures are predicted, DevOps engineers would first perform stress testing that tests the suspected node with synthetic load, in order to validate if the node is indeed failing. Then, DevOps engineers would perform live migration to move the jobs from the failed node to a healthy one which has the same configuration. During the live migration, the complete running status of the virtual machines on the failed node is saved and reloaded in the healthy node, so the system can provide identical services after the migration without impacting the experience of the end-users. In the future, our node failure detection solution can be combined with various self-healing strategies [9, 10, 27, 28] to automatically mitigate the impact of node failures (e.g., through automated live migration and automated management of node repairing process).

2.2 Prior Research Efforts in AIOps

Here we discuss prior AIOps solutions in the context of the aforementioned five criteria for the successful adoption of AIOps solutions.

Trustable. Many of the prior AIOps solutions analyze the raw monitoring data for problem detection [13, 19, 23–25] and issue diagnosis [29]. He et al. proposed Log3C [19] that automatically identifies problems in a cloud-based system using logs. Lin et al. [25] mined previously-reported issues to discover a sudden surge of new issues (a.k.a., emerging issues) which could impact the user experience. Lim et al. [23] clustered existing performance metrics to discover recurrent and unknown issues. Lin et al. proposed a deep learning-based approach, MING [24], which leverages temporal (e.g., CPU and memory) and spatial (e.g., rack locations) data to predict node failures in the cloud. El-Sayed et al. [13] learned from the trace data to predict job failures in large-scale cloud computing platforms. Luo et al. [29] correlated alerts with various resource usage metrics to discover temporal dependency and to infer causal relations between them.

In contrast, we believe that the knowledge, experience, and trust captured in alerts are extremely valuable and should be leveraged over raw data. Prior works [9, 10, 27, 28] attempt to improve existing alert mechanisms (e.g., using rules encoded in KPIs [9, 10] or pre-defined thresholds [44, 45]). Xue et al. [44, 45] proposed a technique to filter out false alerts, while Ding et al. [9, 10, 27, 28] mined historical alerts to suggest self-healing strategies for problematic nodes. In contrast, we leverage the alerts to predict node failures.

Interpretable. Many prior AIOps solutions are built using black-box models (e.g., [24, 44, 45]). Very few prior solutions use interpretable models (e.g., random forest [13] or association rules [27, 28]). None of the prior works actually interpreted the ML models. In this paper, we report on our experience in interpreting the ML models to better understand and improve the existing monitoring and alerts infrastructure (see Section 8 - Interpretability).

Maintainable and Scalable. Most of the prior AIOps solutions [19, 23, 24, 24, 25, 29] are scalable, as they have already been used in production on a large volume of monitoring data from thousands or even millions of nodes. However,

few works discussed the maintenance issues when deploying the systems in the field except [27, 28] who particularly pointed out that the tuning of ML models is very challenging. In this paper, we compare the effort and the computational costs for tuning our ML models.

In-context Evaluation. Only two prior works used non-traditional evaluation techniques to ensure that their models are evaluated in a realistic context. Instead of leveraging cross-validation, Lin et al. [24] proposed a time-based evaluation technique to evaluate their AIOps solution in context. El-Sayed et al. [13] considered the "just-in-time" value of their prediction on job failures for parallel processing systems. A prediction that is correct yet done long after a job has started is of little value. We use a time-based evaluation technique as in [24], and similar to [13], we only considered predictions as valid when they are "just-in-time" for DevOps engineers to react to and not too early to cause financial losses. Different from [13], our "just-in-time" evaluation approach uses a time window to mark valid predictions (i.e., a node failure prediction is only valid when the node actually fails within the time window).

2.3 Prior Research on the Impact of Modeling Decisions

When building an AIOps solution, there will be different modeling decisions (e.g., ML algorithms and configuration tuning) that need to be made. Here we discuss prior empirical studies on the performance impact of the modeling decisions.

Feature Engineering and Data Sampling. Prior works note that feature engineering and data sampling play important roles in ML models for various software engineering tasks (e.g., performance modeling [46] and defect prediction [1, 38, 47]). Since performance metrics may sometimes be incomplete due to various transient issues, Xue et al. [46] proposed an approach, which automatically fills in the missing data. To compensate for the small amount of buggy data, Agrawal et al.[1] proposed a data re-sampling technique, called SMOTUNED (an automatically-tuned version of SMOTE). They observed that using SMOTUNED for data pre-processing significantly improves the performance of their defect prediction models. Zhang et al. [47] observed that using summation to aggregate software features usually hinders the performance of defect prediction models. In this work, we propose an AIOps solution that combines random forest-based models with carefully-designed feature engineering and data sampling.

Parameter Tuning. Prior works demonstrate that the performance of ML models can vary significantly based on their configuration settings [40, 42, 43]. Tantithamthavorne et al. [40, 42] observed that using automated parameter optimization techniques can significantly improve the performance of defect prediction models. The performance of ML models built with certain ML algorithms (e.g., random forest and support vector machines) are more stable across different configurations compared to other algorithms (e.g., neural network and decision trees). Wang et al. [43] observed that tuning the hyper-parameter settings for deep learning-based models, although very time-consuming, can significantly impact the model performance for defect prediction. In this paper, we evaluate the performance of ML models under a range of configuration settings (i.e., different ML algorithms and sampling approaches) in the context of detecting node failures.

3 EXPLORATORY STUDY

We first conducted an exploratory study to understand the characteristics of the provided alert data. This data is collected from tens of thousands of nodes of SystemX's deployment during a period of six months (2018-07-01–2018-12-31).

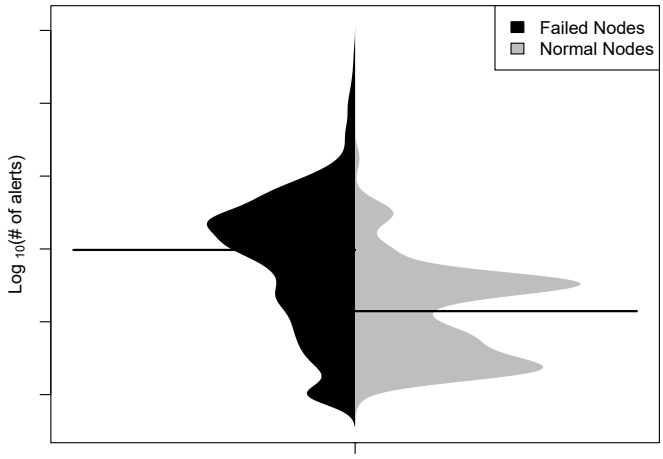

Fig. 1. Visually comparing the distributions of the alerts in failed vs. normal nodes; y-axis values are hidden for confidentiality reasons.

Finding 1: Alert data encapsulates years of field-tested and trusted domain expertise. Alerts for SystemX report problems from different sources (e.g., kernels, drivers, software applications, and hardware components). Each alert contains a timestamp, a verbosity level (`critical`, `unrecoverable`, and `recoverable`), and a textual message describing the error. In terms of severity, the `critical` level alerts are the highest, whereas the `recoverable` level alerts are the lowest. `Critical` level alerts usually refer to issues that are very likely to lead to severe issues if no immediate actions are taken. SystemX has already supported a set of different self-healing strategies (e.g., task migration or load balancing) to improve robustness and to minimize the impact on end users. `Recoverable` level alerts refer to issues that can be addressed by these self-healing strategies, whereas the `Unrecoverable` level alerts refer to issues that cannot. One example of such textual message, `corrected MCE error count exceeded threshold: 4800 times in 24 hours`) informs the DevOps engineers of a particular error *(MCE error)* that occurred too frequently in the past 24 hours. The message content has been modified for confidentiality reasons.

Finding 2: The distribution of alerts is statistically different between the failed and normal nodes. Figure 1 shows a beanplot visualizing the differences in the distribution of alerts between failed and normal nodes. In order to quantify the differences, we applied two non-parametric statistical techniques: Wilcoxon Rank Sum (WRS) and Cliff's Delta (CD). WRS assesses if the two distributions are statistically different. CD [21] calculates the strength of the differences (a.k.a., effect size). Our results show that there is a clear statistical difference (i.e., p-value < 0.05) between the two types of nodes with a *large* effect size (i.e., $\delta = 0.509$). We also calculated the WRS and CD to measure the differences in the distribution of each type of alert between failed and normal nodes. We observed that most of the alert types have statistically significantly different distributions between failed and normal nodes, and that the effect sizes of the differences range from *trivial* to *large*.

Finding 3: Alerts are not always strong indicators of a node failure. We examined the timing of the alerts for failed nodes to determine whether alerts are good indicators of future node failures. For each failed node, we measured the time gap (a.k.a., elapsed time) between the first generated alert and the node failure, as well as the time gap between the last generated alert and the node failure. Figure 2 visualizes the distributions of these two time gaps. In general, the

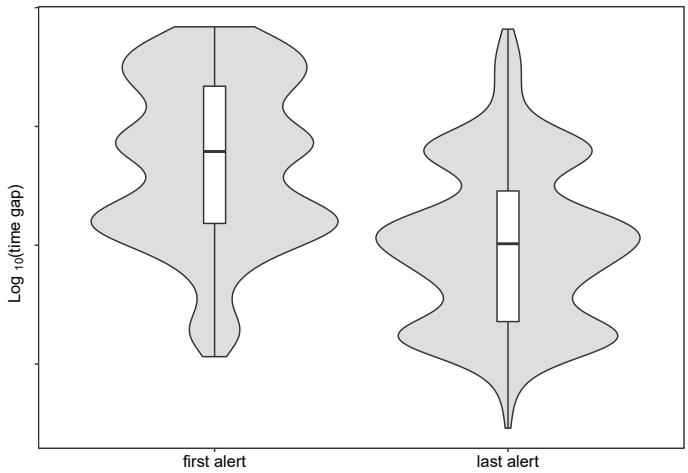

Fig. 2. Distribution of the time gap between the first/last alert and the node failure event; y-axis values are hidden for confidentiality reasons.

first alert occurs about 60 days before the node failure, and the last alert occurs almost right before the node failure. We further sampled some individual nodes to study the frequency of alerts over time, and we observed that there are generally many more alerts close to the node failure time. However, the exact number of alerts close to the failure event (e.g., within 24 hours before the failure) varies significantly among the failed nodes.

Summary of Exploratory Study

Alerts can be used to distinguish between the normal and the failed nodes, as the number of the alerts is significantly higher in the failed nodes than the normal nodes. However, as alerts can happen anytime, only using the alert counts cannot effectively predict node failures. Sophisticated AIOps solution is needed to extract and model various characteristics of the monitoring data.

4 AN OVERVIEW OF OUR APPROACH

This section provides an overview of our approach to building an AIOps solution for predicting node failures. As shown in Figure 3, our AIOps pipeline is divided into three phases: (1) feature engineering (Section 5) processes the monitoring data and produces useful features for the ML models, (2) model training (Section 6) trains the ML models based on the features, and (3) model evaluation (Section 7) examines the effectiveness of the trained ML models in a production-like usage context.

There are four major challenges *([C01]–[C04])* that we faced during our process of building a successful AIOps solution, including the challenges of complex data format, data skewness, hyperparameter tuning for deep learning models, and the valid evaluation of the AIOps solution. Such challenges are not only existing in our application context, but also applied to other scenarios where a similar AIOps solution is needed. We detail our techniques to overcome these challenges in the next three sections.

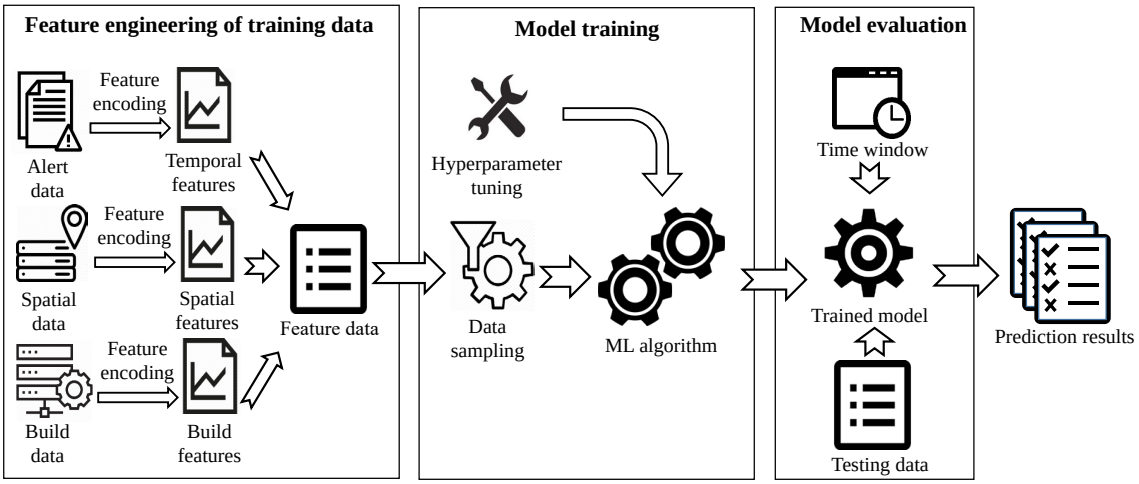

Fig. 3. An overview of our AIOps pipeline.

5 FEATURE ENGINEERING

Our AIOps solution leverages the early warnings (i.e., alert data) of a node to predict node failures. In addition to the alert data, we also included spatial data (e.g., the location of a node) when building our AIOps solutions, as recent work suggests that spatial data can be useful in analyzing and predicting node failures [24, 45]. Furthermore, we incorporated the information about the software and the hardware configuration of a node (i.e., build data), as node failures can be caused by drivers or kernel bugs. The process of extracting and transforming the raw data into features, which are inputs to the ML algorithms, is called *feature engineering*. Feature engineering is generally considered as a key step in a successful ML project, as the performance of an ML model is highly dependent on its features [1, 11].

Feature engineering is always related to the used ML models. In this work, we experimented with three ML algorithms[1] in our AIOps solution: LSTM [18], MING [24], and random forest [5]. In the feature engineering phase, the major challenge is to deal with complex data format, as they cannot be directly used in some of the ML model(s) under study. We need to convert such raw data into features that are suitable for each of the three studied ML algorithms.

[C01] Dealing with the Challenge of Complex Data Format

The raw data has different data formats, some of which are not supported by certain ML models out of the box. Hence, we performed various feature encoding techniques on them. For example, the alert data is a time series; the *cluster name*, a type of spatial data, is a categorical variable; and the *memory size*, a type of build data, is an ordinal variable. Depending on the types of variables and the requirements of each ML algorithm, we used different feature encoding approaches. For the random forest-based model, we converted the time-series data into temporal features using back-tracking. For the LSTM-based model, we encoded the categorical data into numerical data using a target encoding. For the ordinal data, we normalized them. Table 1 shows the resulting features after performing the feature encoding process. In total, we used 1,692 features across the different ML models. There are 1,675 temporal, 9 spatial, and 8 build features.

[1]More details about these ML algorithms are described in Section 6.

Table 1. Number of features for SystemX and their feature representation techniques.

Monitoring data/ Feature category	Variable types	# of features	Encoding technique	ML algorithm
alerts /temporal	time series	1,675	back-tracking	random forest
spatial	categorical	9	target encoding	LSTM
build	categorical	5	target encoding	LSTM
	ordinal	3	normalization	All

Table 2. An example of transforming alerts into temporal features.

Types	Time	00:00–01:00	01:00–02:00	02:00–03:00
Alerts	$Alert_A$	3	1	6
	$Alert_B$	2	5	7
Temporal Features	$Freq_{A_0}$	3	1	6
	$Freq_{B_0}$	2	5	7
	$Freq_{A_1}$	N/A	3	1
	$Freq_{A_1}$	N/A	2	5
	$Ratio_{A_1}$	N/A	0.33	6
	$Ratio_{B_1}$	N/A	2.5	1.4
	$Ratio_{A_2}$	N/A	N/A	0.33
	$Ratio_{B_2}$	N/A	N/A	2.5

All the monitoring data is streamed to a central repository every hour. Therefore, we created a *data point* for each node in each hour. The goal of our AIOps solution is to assist DevOps engineers in deciding whether a node will fail in the near future whenever an alert is generated from that node. Hence, we created a data point for a node in an hour only when there were alerts generated for that node in that hour. No data points were created once a node failed.

Converting time-series data into temporal features for random forest. Alerts, which provide DevOps engineers early warnings to potential node failures, are time-series data. The random forest-based model does not support the modeling of the time-series data out of the box. Hence, we converted the alerts into temporal features using a back-tracking technique. We decided to represent the alert data using two dimensions of temporal features: frequency and change ratio, which can highlight the differences between the normal and failed nodes. We now explain our temporal feature encoding technique using the example shown in Table 2. In this example, there are two types of generated alerts ($Alert_A$ and $Alert_B$) over a period of three hours (00:00–03:00) for this node.

- **Alert frequency.** In Section 3, we observe that there are significantly more alerts in the failed nodes compared to the normal ones, and that alerts happen at different time intervals prior to a node failure. Therefore, we use the *frequency* of alerts to capture the occurrence of different types of alerts over time. For each data point for a node, we calculate the frequency of alerts in each of the previous hours (a.k.a., backtracking). In this running example, we extracted four features along this dimension: $Freq_{A_0}$ and $Freq_{B_0}$ refer to the occurrences of $Alert_A$ and $Alert_B$ for the current hour, whereas $Freq_{A_1}$ and $Freq_{B_1}$ refer to the occurrences of $Alert_A$ and $Alert_B$ in the previous hour for this

Table 3. An example of transforming the spatial data (Room ID) into features.

Room ID	$Room_A$	$Room_B$
# of nodes	300	1,000
# of failed nodes	30	200
Feature representation	0.1	0.2

node. During the period of 01:00–02:00, there is one $Alert_A$. Hence, $Freq_{A_0}$ is one. In the previous hour (00:00–01:00), there are three $Alert_A$s. Hence, $Freq_{A_1}$ is three. We only back-track one hour for the alert counts in our running example for illustrative purposes. In our actual AIOps solution, we back-track the alert counts up to previous 168 hours (a.k.a., 7 days). If there are X types of alerts, there will be $168 \times X$ features along this dimension.

- **Alert change ratio.** Some alerts may be more verbose than the others. Hence, we would like to have a relative measure, the *change ratio of the alerts*, to capture the relative changes of the occurrences of alerts over time. This measure computes the ratio of the number of generated alerts between adjacent hours. In our running example, we extracted four features along this dimension: $Ratio_{A_1}$ and $Ratio_{B_1}$ computes the change ratio of $Alert_A$ and $Alert_B$ between the current and previous hour, whereas $Ratio_{A_2}$ and $Ratio_{B_2}$ computes the change ratio of $Alert_A$ and $Alert_B$ between the previous hour and the hour before the previous hour. There are three, one, and six $Alert_A$s during this three hour period for this node. Hence, during the period of 02:00–03:00, $Ratio_{A_1} = \frac{6}{1} = 6$ and $Ratio_{A_2} = \frac{1}{3} = 0.33$. Same as the frequency dimension, in our actual AIOps solution, we have back-tracked the alert counts up to 168 hours apart. If there are X types of alerts, there will be $167 \times X$ features along this dimension.

Encoding the categorical data for the LSTM model. All the spatial and parts of the build data are categorical data. The spatial data records the global dependency among different nodes. Examples of the spatial data can be the rack, cluster, room ID, and data center ID for each node. The build data provides information about the vendor, the hardware, and software configuration information about each node. Examples include the size of the memory and the manufacture of the node. In total, we have nine types of spatial data and five types of build data that are categorical variables. We performed a variable encoding technique, called target encoding, to convert such categorical variables into numerical variables. *Target encoding* encodes the categorical variables according to the distribution of the "target" across different values of the categorical variable. In our case, each value of a categorical feature is encoded as the rate of the associated node failures with a particular value for that categorical variable. We illustrate this technique using the example that is shown in Table 3. In the example, there are two values for the categorical variable *Room ID*: $Room_A$ and $Room_B$. As shown in Table 3, 30 out of 300 nodes in $Room_A$ failed, while 200 out of 1,000 nodes in $Room_B$ failed. Hence, the node failure rate is 0.1 for $Room_A$ and 0.2 for $Room_B$. This feature will be calculated hourly based on the number of failed nodes in that room at that time (i.e., only considering the failure rate in the past).

Normalizing the ordinal data for all the ML models. There are three types of spatial data (e.g., memory size), which are ordinal variables. Different ordinal variables have different ranges, thus we used the Min-Max approach to normalize them into the same range (0, 1) with the minimal ordinal value being encoded as 0 and the maximum ordinal value encoded as 1. Other in-between values are normalized based on their distance to the minimal and maximum values, using a linear transformation: $\frac{x-min}{max-min}$, where x is the value of a particular ordinal variable. For example, if there are three different memory sizes: 20 GB, 50 GB, and 100 GB, they will be normalized as 0, 0.375, and 1, respectively. When a value (e.g., in the future data) exceeds the Min-Max range, we allow the normalized value to be smaller than 0 or larger

than 1. As we describe in Section 7.1, our ML models should be re-trained periodically to avoid concept drift. Thus, the chance that value exceeds the Min-Max range is very small.

6 MODEL TRAINING

This section explains our various modeling decisions during the model training phase. We experimented with three ML algorithms in our AIOps solution: LSTM [18], MING [24], and random forest [5]. In order to conduct a fair comparison, all three ML algorithms leverage all three types of data (i.e., alert data, spatial data, and build data) as features, using the feature engineering techniques described in Section 5. We also proposed baseline models to understand the relative performance of the used ML algorithms. Below, we explain our chosen ML algorithms and our rationale for choosing them in our study.

- **Baseline models.** We design our baseline models based on the existence of each type of alert. For each type of alert, we build a baseline model that predicts a node failure if and only if there exists such type of alert. Our baseline models are derived from domain experts' experience of using the existence of certain alerts to identify potential node failures. The domain experts consider the existence of five types of alerts to predict node failures. We build baseline models for all five types of alerts: one baseline model for each type of alert.
- **LSTM** is a deep neural network-based ML algorithm commonly used for predicting time-series data [18]. We choose LSTM as it is natural to model the problem of predicting node failures using the temporal features (i.e., time-series features). For a fair comparison, we also included the spatial and build features by binding them with the temporal features in this paper. Among different variants of LSTM, we used the bi-directional version, as it is reported to yield the best performance in this context [17].
- **Random forest** is an ensemble-based classification algorithm [5]. Compared to the LSTM, which is considered as a black-box, the random forest is interpretable. An ML model is interpretable if we can understand its decision process and study the relative importance of features. The interpretability of ML models has already been used in prior research to understand phenomena (e.g., fault proneness [6]) or to improve processes (e.g., code review [37]). We choose the random forest algorithm because it's interpretable [39] and usually achieves the best performance among other traditional classifiers [15].
- **MING** [24] is a hybrid approach, which combines LSTM and random forest. Inside MING, there are two ML models: an LSTM model, which learns from the temporal features, and a random forest model, which learns from the spatial features. During prediction, MING combines the intermediate results from the internal LSTM and the random forest model using a Learning to Rank (LTR) model [26] and outputs one unified prediction outcome. We choose MING because it is the state-of-the-art approach for predicting node failures. Recent work reports that when predicting node failures inside the Microsoft Azure cloud, the ML models trained using the MING approach outperform the LSTM and the random forest-based models [24]. In this work, we replicate MING by learning an internal LSTM model using the temporal features and learning an internal random forest model using the spatial and build features (the build features are newly added in our study).

There are two major challenges in this phase: (1) *data skewness*: for a highly reliable system like SystemX, the chance that a node fails in the near future is extremely small. This poses challenges for building effective models; and (2) *hyperparameter tuning*: ML algorithms have many configuration parameters that can impact the performance of the ML models. It is challenging to effectively locate an optimal configuration setting within a limited amount of time.

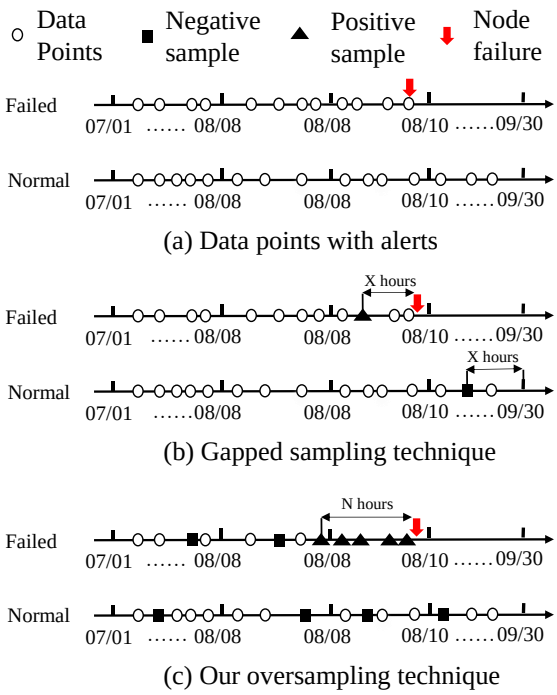

Fig. 4. An example for different data sampling techniques.

[C02] Dealing with the Challenge of Data Skewness

The failed and normal nodes are very imbalanced in our dataset. Even for the failed nodes, the healthy time period is much longer than the failing period, i.e., a node is always healthy before it fails. In other words, our dataset is highly skewed, which poses challenges for building models that can effectively predict node failures.

Traditionally, data re-balancing (e.g., oversampling or undersampling) approaches are used to deal with data skewness. Oversampling adds random repetitions of the minority data points while under-sampling randomly removes the majority data points. However, prior works [38, 39] note that neither sampling techniques increases the performance of classification models in terms of AUC (Area Under the Curve). In this work, we proposed a novel oversampling technique to not only select representative data points from both the failed and the normal nodes, but also significantly enrich the training samples related to node failures. We also implemented another data sampling technique, called *gapped sampling*, as it was previously proposed and used in a similar problem context (a.k.a., node failure prediction [24]). We intend to compare the resulting model performance with these two sampling techniques. We will illustrate both data sampling techniques using a running example in Figure 4(a) which shows all the data points (a.k.a., hourly periods containing alerts) for the normal and the failed node. A red arrow indicates the time of a node failure. **We only apply the data sampling techniques to the training data**.

Selecting the early failure indicators for node failures. The **gapped sampling** technique was originally introduced in [24] for selecting data points for both the failed and normal nodes in the Azure cloud. The idea behind the gap sampling technique is to select data points that closely resemble early indicators of node failures. As shown in Figure 4(b),

for each failed node, one positive sample (i.e., indicated by a solid triangle) is selected X hours before the node failure occurs. For each normal node, one negative sample (indicated by a solid square) is selected X hours before the end of our training period.

Capturing a complete picture for both the normal and failure nodes. We propose a novel **oversampling** technique that enriches data by capturing both the failing and normal behavior for the eventually failed nodes, as well as the behavior of normal nodes. Section 3 shows that alerts can be generated at any time before the failure of a node. Furthermore, different types of alerts may be generated at different data points. Only selecting one data point (as is done in the *gapped sampling* technique) might miss the early indicators for the failure of a node. Compared to the *gapped sampling* technique, our *oversampling* technique selects more positive samples from the failed nodes, which helps us better capture the failure symptoms. In addition, our technique selects more data points from the normal nodes, as normal nodes can also generate alerts. For each failed node, we consider an N-hour time window right before the time when the failure occurs. All the data points within this window are labeled as positive examples. In addition, for each failed node, one data point (i.e., a negative sample) is randomly selected every day (except for the N-hour window before the time of the failure). For each normal node, one data point (i.e., a negative sample) is randomly selected every day. Figure 4(c) illustrates the sampling results of our running example using our *oversampling* technique. In addition to the five positive samples, additional negative samples are also randomly selected outside the N-hour time window for the failed node.

We evaluated the performance of the three ML algorithms under the two different data sampling techniques. Each data sampling technique has one configuration: the gapped size X in the *gapped sampling* technique and the window size N in our *oversampling* technique. As described in Section 7.2, while training ML models, we also tuned these two configuration options.

[C03] Dealing with the Challenge of Hyperparameter Tuning

Deep neural network-based models like LSTM and MING contain many configuration parameters, which are collectively called hyperparameters. Studies [33, 43] show that the performance of these models can vary significantly based on the choice of the configuration parameters. We used a random search to tune the hyperparameters for LSTM, as prior work [4] shows that random search is more efficient in locating optimal hyperparameter settings than grid search in the hyperparameter space. As the optimal configurations may not be the same across different contexts, we performed this hyperparameter tuning process for LSTM under each data sampling technique. Since MING also uses LSTM, the same tuning process is performed. Different from the LSTM/MING-based models, the performance of the random forest-based models is generally stable across different configuration settings [40]. We just used the default configuration setting for the random forest-based models.

7 MODEL EVALUATION

One of the major challenges we faced in the model evaluation phase is how to evaluate the performance of ML models in a production-like context. We first discuss how we dealt with this challenge. Then we describe the experiments that we conducted (Section 7.1) and present the results of our evaluation (Section 7.2).

[C04] Dealing with the Challenge of in-Context Evaluation

In order to evaluate the ML models in a production-like setting, we developed a new evaluation technique. There are two main differences compared to the conventional evaluation techniques for ML models:

(1) Evaluating the prediction for each node: We evaluated the prediction performance for each node instead of individual prediction results. For each node and each hourly period containing alert(s), a prediction is made. Such individual predictions are considered together to evaluate the effectiveness of the predictions for a node, i.e., whether the predictions help DevOps engineers spot the failure of the node. If DevOps engineers took the advice from our AIOps solution, they would decommission a node whenever this node is predicted to be failing. There should not be any subsequent predictions when a node is taken offline. Hence, if a node is predicted to be failing at a particular time instance, we will not evaluate its predictions afterward even though this node may actually be healthy and generating alerts subsequently.
(2) Just-in-time prediction: The prediction of node failures should be made "just-in-time". On the one hand, predicting a node failure prematurely (e.g., several weeks ahead) would lead to financial losses for the company, as currently-healthy nodes will be mistakenly replaced with newly purchased ones. On the other hand, late predictions (e.g., ten seconds before the node failure) would be of no value, as DevOps engineers would not have time to react and perform mitigation actions. To cope with this requirement, we introduced a prediction time window. A node failure prediction will not be useful unless such a prediction is made within this time window. After consulting with the DevOps engineers at Alibaba, the desirable prediction window is between 2 and 72 hours, which is a good balance between the cost of predicting node failures too early and the effectiveness of our approach to predict node failures in advance. However, the prediction window may need to be adjusted for a different industrial setting.

7.1 Experiments

We experimented with our AIOps pipeline using the data collected from tens of thousands of nodes of SystemX's deployment during a period of six months (2018-07-01–2018-12-31). Our AIOps solution needs to be used continuously when it is deployed in production. Since cloud computing platforms are constantly evolving due to changes like user workload, feature updates, and hardware/software upgrades [14], the causes of node failures will likely change over time as well. ML models should be re-trained periodically to avoid concept drift. The production team plans to re-train the ML models every month. Hence, to simulate this context, we conducted three experiments, which evaluate the ML models under three consecutive evaluation months: October, November, and December, 2018. For each experiment, we used all the available data before the evaluation period as training data. For example, to evaluate the ML models under the month of October, we used all the monitoring data between July 1, 2018 and September 30, 2018.

7.2 Evaluation Results

For each of the three experiments, we trained the three ML algorithms under the two different data sampling techniques. For the LSTM and MING-based models, we performed hyperparameter tuning to select the optimal configuration parameters. In addition, we experimented with various gap sizes (X) and window sizes (N) in the two data sampling techniques (we provide the results for different gap sizes and window sizes in Section 8). We calculated the AUC to measure the performance of these ML models under different settings. AUC is insensitive to a classification threshold and it provides a good performance measure when the distribution of the predicted classes (e.g., failed nodes and normal nodes) is imbalanced [22]. Due to Non-Disclosure Agreement (NDA), we cannot report the detailed results for other performance measures (e.g., precision and recall). However, other performance measures yield similar trends as the AUC. For a particular ML model (ML algorithm + data sampling technique), we selected the ML model configuration with the best performance. In addition, we also kept track of the time taken to train and test the ML models.

Our oversampling technique performs better than the gapped sampling technique. Figure 5 shows the average AUC measures of the resulting ML models across three different models. All three ML models perform better under

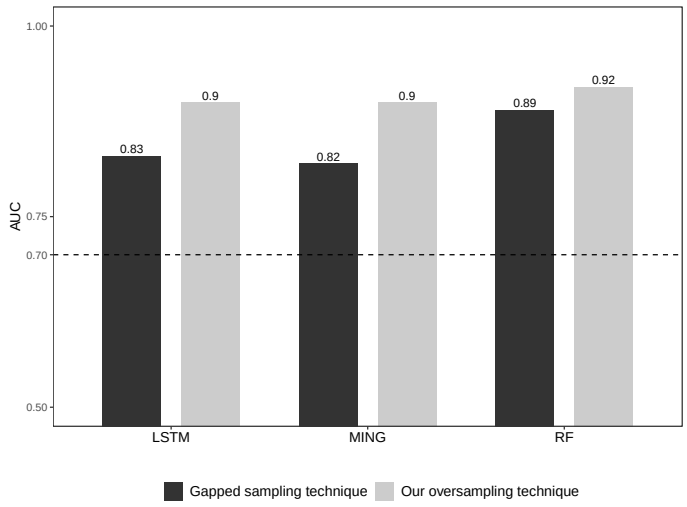

Fig. 5. The AUC measures for the different ML models. The dotted line shows the best performance of the baseline models that predict node failures based on the existence of each alert type.

our oversampling technique, with an AUC improvement of 3% to 10%. Compared to the gapped sampling technique, our oversampling technique enriches the training data related to node failures, thereby achieving better performance. Besides, all the models achieve better AUC than the best baseline model that simply predicts node failures based on the existence of each type of alert (our baseline models achieve an AUC of 0.50 to 0.70, compared to the AUC of 0.82 to 0.92 for the evaluated models).

Random forest combined with our oversampling technique is the best solution in terms of prediction performance. Among all the ML models, the ML models that are trained using random forest using our oversampling technique perform the best, whereas the ML models that are trained using MING or LSTM under the gapped sampling technique are the worst. The ML model trained using random forest under our oversampling technique achieves a 12% higher AUC than MING under the gapped sampling technique (i.e., the solution in [24]).

Random forest-based models are our best solution in terms of computational cost. When comparing different ML models (see Table 4), the training and the testing process of random forest-based models is much faster than its deep-learning counterpart. For example, under our oversampling technique, the training process of random forest is 33 times faster than MING and 72 times faster than LSTM, without considering the time taken for hyperparameter tuning. When comparing the testing process, the random forest-based model under the same oversampling technique is 75 and 146 times faster than MING and LSTM, respectively. Note that for LSTM/MING-based ML models, hyperparameter tuning is required. Hence, multiple training processes under different configurations need to be performed. The training process for MING/LSTM-based models will be much longer (e.g., hundreds or thousands of times longer). As the models need to be re-trained periodically in a real-production environment, using a random forest model can effectively reduce the computational cost. Therefore, among all the ML models, we chose a random forest model under our oversampling technique as our final AIOps solution.

Temporal features are most influential to the model performance. Table 5 shows the performance of the random forest model (under our oversampling approach), using all but excluding each type of feature. The performance of

Table 4. Training and testing time of different modeling approaches. All the experiments were conducted on a public cloud with a provision of 16 Intel Skylake (2.399 GHz) CPU cores and 177 GB memory.

	MING		LSTM		RF	
	Oversampling	**Gapped**	**Oversampling**	**Gapped**	**Oversampling**	**Gapped**
Training (sec)	3,159	1,202	6,875	1,358	94	40
Testing (sec)	151	208	293	349	2	2

Table 5. Performance (AUC) of the random forest model under our oversampling approach, excluding each type of feature.

All features	**Excluding temporal features**	**Excluding spatial features**	**Excluding build features**
0.92	0.60	0.89	0.89

the model is slightly impacted (i.e., with an AUC decrease of 3%) when the spatial features or the build features are excluded. In comparison, the performance of the model is significantly impacted (i.e., with an AUC decrease of 35%) when the temporal features are excluded.

Summary of Evaluation Results

The ML models perform better under our newly proposed oversampling technique than the gapped sampling technique. When comparing different ML models under the same sampling techniques, models trained using random forest are more efficient and more accurate than the deep learning-based models (LSTM and MING). Hence, we decided to use a random forest model combined with our oversampling technique as our AIOps solution.

8 DISCUSSION & LESSONS LEARNED

In this section, we present the learned lessons during the process of building, evaluating, and deploying our AIOps solution. We structure this discussion along the five criteria for the successful adoption of AIOps solutions.

Trustability. Experiments show that our AIOps solution outperforms the prior state-of-the-art AIOps solution [24] when predicting node failures in SystemX. Instead of processing tens of terabytes of the raw monitoring data, our solution leverages the existing domain expertise embedded in the alerts. DevOps engineers work with alerts every day and are constantly modifying the rules which generate these alerts. Our solution can easily incorporate changes in the alerts (e.g., the addition of new types of alerts or updating existing alerts) and integrates nicely with DevOps engineers' daily workflow. DevOps engineers from SystemX trust their field-tested alerts and in turn trust our solution and use it daily.

Interpretability. LSTM and MING-based models are black-box ML models, as they cannot explain the decision processes behind their predictions. Our AIOps solution is a random forest-based model, which is interpretable. We opened up the trained models and studied the importance scores of the various features that contributed to the prediction output. We leveraged such information to help us better understand and improve the monitoring infrastructure and software.

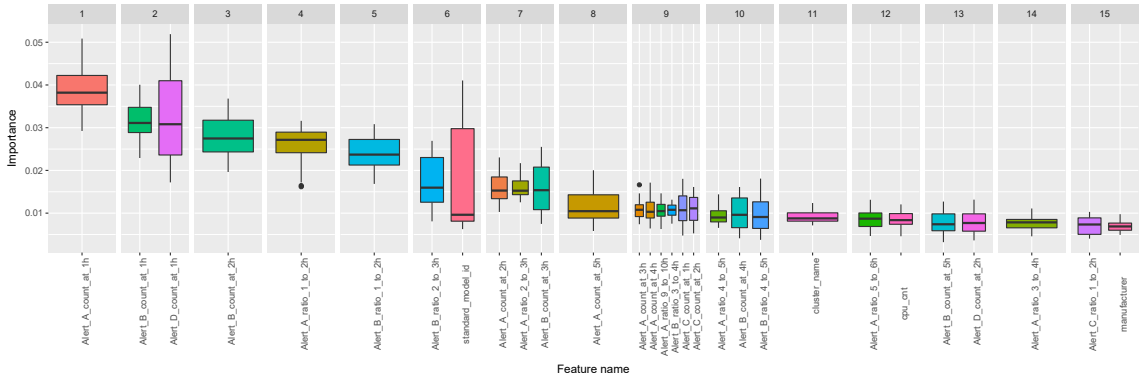

Fig. 6. The distribution of the feature importance scores in the random forest-based model with our oversampling approach. The features are grouped into statistically heterogeneous groups using the Scott-Knott ESD test.

Since the ML models are re-trained every month, the importance scores for each feature may vary over time. To address this issue, we aggregated the importance scores for each feature across different models trained from different time periods. For each of the three experiments described in Section 7.1, we first applied the bootstrap sampling technique on our training set. Then we built a random forest-based model and extracted the feature importance score for each feature. We repeated this process ten times. We aggregated these feature importance scores across different models and ranked them with the Scott-Knott Effect Size Difference (ESD) test [41]. Scott-Knott ESD test measures the magnitude of the differences between two measurements (feature importance scores in our case) and leverages a hierarchical clustering to partition the measurements into statistically distinct groups with a non-negligible difference. Then, we studied the top 15 most important groups of features. A model that is built using only the top 15 most important groups of features achieves an AUC that is only 2% lower than the full model. Figure 6 visualizes the results of our Scott-Knott ESD test (showing the top 15 groups of features).

[Model Understanding]: *The alert data plays the most important role when predicting node failures.* 25 out of 29 (86%) features from the top 15 groups are alert-related. For example, $Alert_A$ and $Alert_B$ are two of the most important features. Both $Alert_A$ and $Alert_B$ are unrecoverable issues. $Alert_A$ is an error reported by the kernel, whereas $Alert_B$ is an error reported from the applications, which perform self-checks on nodes. However, not all alerts are important in our model. For example, none of the features related to $Alert_X$ appear in the top 15 groups, even though $Alert_X$ is a `critical` level alert. However, since it appears almost right before the node failure, it does not play an important role in our prediction models. The feature importance scores of the alert types are consistent with the performance of the baseline models that consider the corresponding alert types: the alert types that achieve a better baseline model (e.g., $Alert_A$ and $Alert_B$) correspond to more important features in our model. Only one spatial feature (*cluster name*) and three build features (*standard model ID*, *CPU count*, and *manufacturer*) are important. Compared to the alert-related features, they are ranked considerably lower. After discussing with the DevOps engineers, we found that these features are related to some particular types of physical machines in certain clusters.

[Process Improvement]: *Selective streaming of the monitoring data.* The monitoring data from the individual nodes are currently aggregated and streamed to a central repository every hour. As SystemX is constantly growing with more nodes being added regularly, the monitoring infrastructure may be overwhelmed with the increasing amount of the data. Based on the above results, a more fine-grained data aggregation and streaming process can be used to alleviate

such pressure. First, the alerts, which process the data from the monitoring probes and the health-checking applications, seem to sufficiently capture the early warning signs of node failures. Hence the recorded data from the monitoring probes and the health-checking applications can be streamed over larger time intervals (e.g., every two hours) to ensure the scalability of the monitoring infrastructure. Besides, the monitoring data related to the unimportant alerts can be collected less frequently than the monitoring data related to the important alerts. Furthermore, for those particular types of machines located in certain clusters, which are shown to be more failure-prone than the others, smaller time intervals (e.g., every 15 minutes) might be needed to better monitor and ensure the QoS for these nodes.

Maintainability. All the modeling process (feature engineering, model training, and model evaluation) described in our approach can be automated, once the AIOps solution is deployed in the field. However, one of the major issues related to the maintainability of the ML models is related to configuration tuning [36]. Since the ML model needs to be retrained every month, having a robust (a.k.a., insensitive to various configuration settings) ML algorithm is crucial to the maintainability of the AIOps solution.

As explained in Section 6, we experimented with various hyperparameter settings for the LSTM and the MING-based models and found that the performance of these models can vary significantly based on the choice of the hyperparameters. Furthermore, the optimal hyperparameter settings vary depending on the training dataset and the choice of the oversampling techniques. Currently, we use a random search for hyperparameter tuning, as it is more efficient than the widely used grid search in locating the optimal hyperparameter settings [4]. This process is very time consuming and requires ML expertise. Random forest-based models, on the other hand, are considerably less sensitive to hyperparameter settings [40].

Scalability. Our AIOps solution is already adopted and used in practice to analyze alert data in SystemX. As shown in Section 7.2, compared to LSTM/MING-based models, random forest-based models are much faster during the training and the prediction process. Feature engineering is another resource-consuming phase, as it takes a while to encode the alerts into temporal features. One way to reduce the timing of this phase is to reduce the number of temporal features that need to be encoded. For example, we back-tracked the alert counts up to the previous 168 hours (7 days). However, after checking the top important features in our built random forest models, we observed that all the important alert-related features are between 1 to 10 hours before the failure. We subsequently built a random forest model using only the important features appeared in the top 15 groups. The performance of this new model is only slightly (i.e., 2%) lower than the original one. This gives us a good idea in terms of further improving the efficiency of our initial AIOps solution.

In-context Evaluation. Traditionally, the performance of ML models is evaluated based on the *correctness* of individual predictions on the testing datasets. However, after iterations of discussions with the DevOps engineers, we realized that such evaluation techniques would overestimate the performance of a model in the actual production context. For example, predicting a node failure multiple times (i.e., in different individual predictions) or predicting node failures without a time window will likely overestimate the performance. Therefore, we developed a context-aware evaluation technique, which calculates the aggregated prediction performance for each node, in a "just-in-time" manner. Using the traditional evaluation technique, the performance measures (e.g., AUC) would have been overestimated by up to 20%.

Difficult to transfer AIOps solutions to a different context. The monitoring infrastructure is highly system and company-dependent. Therefore, AIOps solutions may not be easily transferable to other systems. For example, Lin et al. [24] proposed the MING approach to predict node failures in the Microsoft Azure cloud. Their AIOps solution works nicely in that context, but performed poorly in ours. This is mainly due to the differences in the monitoring infrastructure, the availability of the monitoring dataset, and the problem context. First, the monitoring infrastructures and the types of

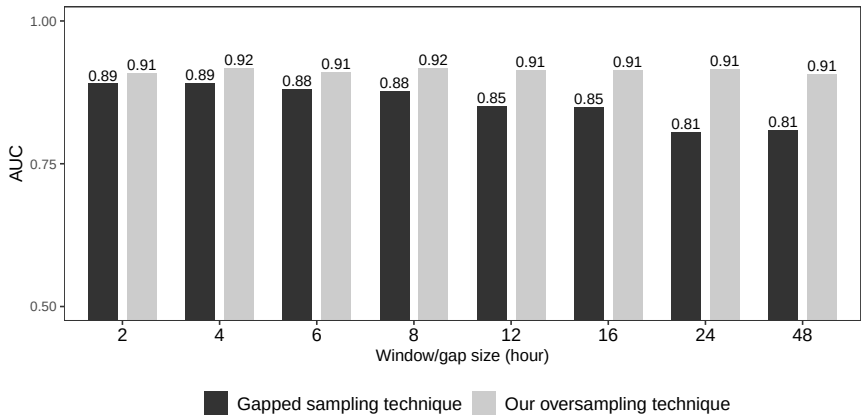

Fig. 7. The AUC measures for the random forest-based models with different sampling configurations.

available monitoring data are different. MING uses the raw system resource usage data (e.g., CPU, memory, and disk) in their prediction model, whereas we use alerts which already processed such data based on domain knowledge and experience from previously reported issues. Second, the problem context is not completely the same. Although both problems are about predicting node failures, their AIOps solution makes predictions in a fixed time interval. In our case, however, a prediction for a node is only made when there are new alerts generated for that particular node, as DevOps engineers need to know whether they need to react to a particular alert or not.

Impact of different window and gap sizes in the sampling approaches. One can configure the window size (N) and gap size (X) in our oversampling technique and the gapped sampling approach, respectively. Figures 7, 8, and 9 show the performance of the ML models using different window sizes in our oversampling approaches and different gap sizes in the gapped sampling approaches. Figures 7, 8, and 9 show that, for all three types of ML models, our oversampling technique always achieves better results than the gapped sampling approach under different configurations of the window and gap sizes. Besides, our oversampling approach is less sensitive to the window size than the gapped sampling approach is to the gap size. For example, for the random forest model under our oversampling technique, the AUC keeps stable at 0.91 and 0.92 for all the experimented window sizes (i.e., 2 to 48 hours). In comparison, for the random forest model under the gapped sampling technique, the AUC varies from 0.81 to 0.89 under different gap sizes (i.e., 2 to 48 hours). As our oversampling technique captures the complete picture before node failures, it outperforms the gapped sampling approach that only captures partial early indicators of node failures.

Why our solution fails to predict node failures in some cases. As discussed in Section 3, alerts happen not only on failed nodes but also on healthy nodes. In some cases, some failed nodes exhibit similar patterns of alerts as normal nodes, which causes it extremely difficult to distinguish such failed nodes from healthy nodes. Besides, we use a time-based evaluation that trains models on earlier data (e.g., the first three month data) and evaluate models on later data (e.g., the fourth-month data). As the hardware, software, and the environment is constantly evolving, the relationship between node failures and the explanatory features can be different between the model training time and the model evaluation time. For example, the types and volumes of alerts may evolve with the changing hardware and software. There are also cases where node failures cannot be captured by existing alerts. DevOps engineers hence need to add new types of alerts to better capture and warn such failures. In addition, the quality of the labels of node failures

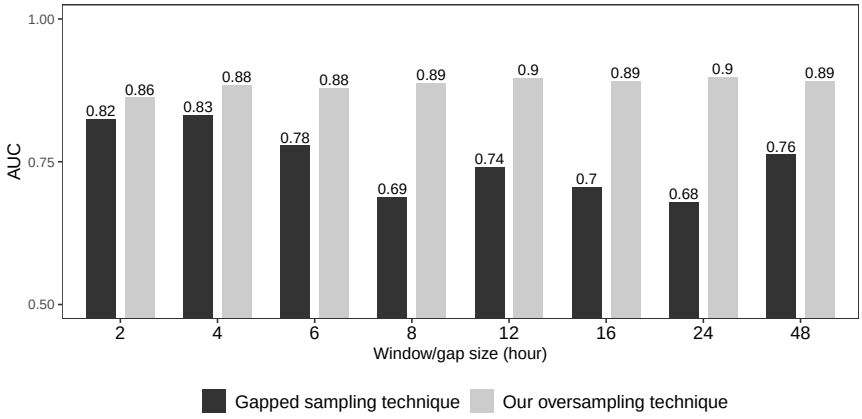

Fig. 8. The AUC measures for the LSTM-based models with different sampling configurations.

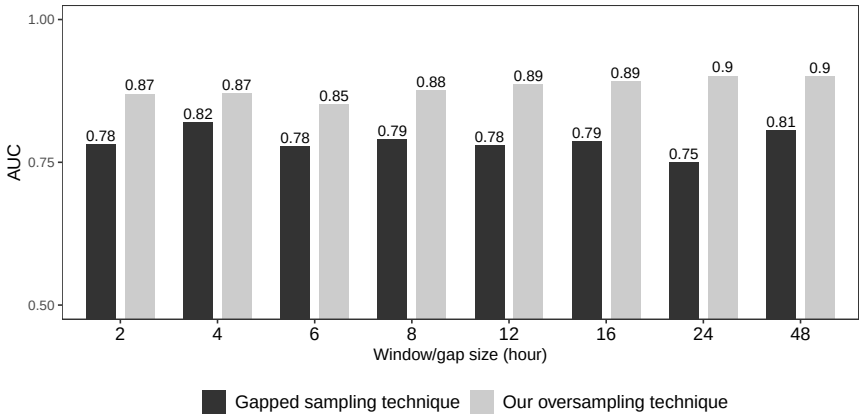

Fig. 9. The AUC measures for the MING-based models with different sampling configurations.

and the collected feature data can also impact the accuracy of our node failure predictions. For example, the volume of alerts may not be measured accurately when monitoring tool failures happen occasionally.

9 THREATS TO VALIDITY

9.1 External Validity

We demonstrated that other AIOps solutions (e.g., MING [24], which we replicated in our study) are not directly transferable to our context, as different systems have different monitoring infrastructures. The problem context could be different as well. Various aspects (e.g., feature engineering, data sampling, and evaluation techniques) of the ML models should be tailored towards the particular problem context. However, the experience and the lessons that we learned during this process are useful to other researchers and practitioners, when they develop their own AIOps solutions, especially when they are interested in ensuring successful adoption of their AIOps solutions in practice.

We showed that random forest-based models combining with innovative data sampling techniques outperforms deep learning models in our context. This also matches the findings from recent study on comparing automated recommendation results [8]. This may be due to the scarcity of positive samples (i.e., node failures) in the training data. Deep learning models require large amount of training data in order to perform well. However, the node failure events, although they are very costly if they occur, are very rare in practice.

This work uses one ultra-large-scale cloud computing platform as the case study system, as it is difficult to get access to research resources of multiple such platforms. Hence, our approaches and findings may have limitations in generalizing to other cloud computing platforms, as the symptoms and causes of node failures might be different for another different cloud computing platform. For example, our approaches may not apply to cloud computing platforms that do not produce alerting signals ahead of the node failure time. In addition, our approach is highly dependent on the granularity of the alerts. For example, if certain types of alerts are generated all the time, they may not be useful for our approach. Nevertheless, other researchers and practitioners can learn from our general approaches and findings to craft their own AIOps solution under a different business scenario. The paper presented the results of evaluating our AIOps solution on the platform for a period of six months. In fact, our solution has been applied on SystemX on a daily basis to help DevOps engineers operate the ultra-large-scale cloud computing platform.

9.2 Internal Validity

The performance of ML models is highly dependent on the various choices that are made during the modeling process. Therefore, we built a range of ML models with three ML algorithms (LSTM, MING, and random forest), two different data sampling (gapped vs. our oversampling) techniques, and configuration parameters (configuration parameters for the ML models and for the data sampling techniques). In our case study, the ML model built with random forest combined with our oversampling technique performs the best.

9.3 Construct Validity

Instead of using the traditional evaluation techniques for ML models, we developed a new approach, which better reflects the usage context of our models in a real-production setting. Under our new evaluation technique, no further prediction results will be evaluated when a failure prediction is made for a node, as DevOps engineers would have already replaced this node with new ones. Furthermore, predicting a node failure outside the prediction window (2–72 hours based on business requirement) is not considered as a successful prediction in our problem context, as too-early predictions would waste company's money to replace healthy nodes, and too-late predictions leave little or no time for DevOps engineers to react.

In this work, we evaluated the performance of our AIOps solution in a production-like environment instead of a real-production environment. However, we used the data from a real-production environment and we simulated the scenarios when our solution had been applied in the real-production environment. Theoretically, our evaluation can equivalently measure the performance of our AIOps solution when applied in a real-production environment.

10 CONCLUSIONS AND FUTURE WORK

This paper reports our process in building an AIOps solution for predicting node failures in an ultra-large-scale cloud computing platform. We experimented with three ML algorithms (LSTM, MING, and random forest) under two data sampling (gapped vs. our oversampling) techniques. We evaluated the performance of the resulting ML models in a production-like context. Results show that random forest combined with our oversampling technique yields the best

performance in terms of computational costs and predictive power. We discussed our learned lessons along with five key criteria for the successful adoption of AIOps solutions: (1) trustable, (2) interpretable, (3) maintainable, (4) scalable, and (5) in-context evaluation. The lessons that we learned from building and deploying our AIOps solution will be of great value for researchers and practitioners, who are interested in ensuring the adoption of their AIOps solutions. Our solution is already used on a daily basis for SystemX at Alibaba. Once our AIOps solution predicts a node failure in the near future, DevOps engineers would take preventative actions (e.g., live migration) to minimize the impact of node failures on the production system. In the future, in order to improve node resiliency, our AIOps solution can be combined with self-healing strategies to automatically mitigate the impact of node failures (e.g., through automated live migration and automated management of node repairing process).

ACKNOWLEDGMENTS

This work is partially funded by the Alibaba Innovative Research Program (AIR). We are grateful to Alibaba for providing access to their systems and data used in our study. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of Alibaba and/or its subsidiaries and affiliates. Our results do not reflect the quality of Alibaba's products.

REFERENCES

[1] Amritanshu Agrawal and Tim Menzies. 2018. Is "Better Data" Better Than "Better Data Miners"?: On the Benefits of Tuning SMOTE for Defect Prediction. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. 1050–1061.

[2] George Amvrosiadis, Alina Oprea, and Bianca Schroeder. 2012. Practical scrubbing: Getting to the bad sector at the right time. In *Proceedings of the 42th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 1–12.

[3] Alberto Avritzer, Andre Bondi, Michael Grottke, Kishor S. Trivedi, and Elaine J. Weyuker. 2006. Performance Assurance via Software Rejuvenation: Monitoring, Statistics and Algorithms. In *Proceedings of the 36th International Conference on Dependable Systems and Networks (DSN)*. 435–444.

[4] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research* 13, Feb (2012), 281–305.

[5] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[6] Marcelo Cataldo, Audris Mockus, Jeffrey A. Roberts, and James D. Herbsleb. 2009. Software Dependencies, Work Dependencies, and Their Impact on Failures. *IEEE Transactions on Software Engineering* 35, 6 (2009), 864–878.

[7] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live Migration of Virtual Machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation (NSDI)*. 273–286.

[8] Maurizio Ferrari Dacrema, Paolo Cremonesi, and Dietmar Jannach. 2019. Are We Really Making Much Progress? A Worrying Analysis of Recent Neural Recommendation Approaches. In *Proceedings of the 13th ACM Conference on Recommender Systems*. 101–109.

[9] Rui Ding, Qiang Fu, Jian-Guang Lou, Qingwei Lin, Dongmei Zhang, Jiajun Shen, and Tao Xie. 2012. Healing Online Service Systems via Mining Historical Issue Repositories. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 318–321.

[10] Rui Ding, Qiang Fu, Jian Guang Lou, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Mining Historical Issue Repositories to Heal Large-Scale Online Service Systems. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 311–322.

[11] Pedro Domingos. 2012. A Few Useful Things to Know About Machine Learning. *Commun. ACM* 55, 10 (2012), 78–87.

[12] Editorial. 2018. Towards trustable machine learning. *Nature Biomedical Engineering* 2 (2018), 709–710.

[13] Nosayba El-Sayed, Hongyu Zhu, and Bianca Schroeder. 2017. Learning from Failure Across Multiple Clusters: A Trace-Driven Approach to Understanding, Predicting, and Mitigating Job Terminations. In *Proceedings of the 37th International Conference on Distributed Computing Systems (ICDCS)*. 1333–1344.

[14] King Chun Foo, Zhen Ming (Jack) Jiang, Bram Adams, Ahmed E. Hassan, Ying Zou, and Parminder Flora. 2015. An Industrial Case Study on the Automated Detection of Performance Regressions in Heterogeneous Environments. In *Proceedings of the 37th International Conference on Software Engineering (ICSE SEIP)*. 159–168.

[15] Baljinder Ghotra, Shane McIntosh, and Ahmed E. Hassan. 2015. Revisiting the Impact of Classification Techniques on the Performance of Defect Prediction Models. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. 789–800.

[16] Mohamed G. Gouda and Tommy M. McGuire. 1998. Accelerated Heartbeat Protocols. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS)*. 202–209.

[17] Alex Graves and Jürgen Schmidhuber. 2005. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks* 18, 5-6 (2005), 602–610.

[18] K. Greff, R. K. Srivastava, J. Koutnik, B. R. Steunebrink, and J. Schmidhuber. 2017. LSTM: A Search Space Odyssey. *IEEE Transactions on Neural Networks and Learning Systems* 28, 10 (2017), 2222–2232.

[19] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2018. Identifying Impactful Service System Problems via Log Analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 60–70.

[20] IDG. 2018. 2018 Cloud Computing Survey. https://www.idg.com/tools-for-marketers/2018-cloud-computing-survey/. Last accessed 09/25/2019.

[21] Vigdis By Kampenes, Tore Dybå, Jo E. Hannay, and Dag I. K. Sjøberg. 2007. Systematic review: A systematic review of effect size in software engineering experiments. *Information Software Technology* 49, 11-12 (2007), 1073–1086.

[22] Max Kuhn and Kjell Johnson. 2013. *Applied predictive modeling*. Springer.

[23] Meng-Hui Lim, Jian-Guang Lou, Hongyu Zhang, Qiang Fu, Andrew Beng Jin Teoh, Qingwei Lin, Rui Ding, and Dongmei Zhang. 2014. Identifying Recurrent and Unknown Performance Issues. In *2014 IEEE International Conference on Data Mining (ICDM)*. 320–329.

[24] Qingwei Lin, Ken Hsieh, Yingnong Dang, Hongyu Zhang, Kaixin Sui, Yong Xu, Jian-Guang Lou, Chenggang Li, Youjiang Wu, Randolph Yao, et al. 2018. Predicting Node failure in cloud service systems. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 480–490.

[25] Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, and Dongmei Zhang. 2016. iDice: Problem Identification for Emerging Issues. In *IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 214–224.

[26] Tie-Yan Liu. 2009. Learning to Rank for Information Retrieval. *Foundations and Trends® in Information Retrieval* 3, 3 (2009), 225–331.

[27] Jian-Guang Lou, Qingwei Lin, Rui Ding, Qiang Fu, Dongmei Zhang, and Tao Xie. 2013. Software Analytics for Incident Management of Online Services: An Experience Report. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 475–485.

[28] Jian-Guang Lou, Qingwei Lin, Rui Ding, Qiang Fu, Dongmei Zhang, and Tao Xie. 2017. Experience report on applying software analytics in incident management of online service. *Automated Software Engineering* 24, 4 (2017), 905–941.

[29] Chen Luo, Jian-Guang Lou, Qingwei Lin, Qiang Fu, Rui Ding, Dongmei Zhang, and Zhe Wang. 2014. Correlating Events with Time Series for Incident Diagnosis. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 1583–1592.

[30] Matthias Machowinski. [n.d.]. How predictive maintenance can eliminate downtime. https://technology.ihs.com/572369/businesses-losing-700-billion-a-year-to-it-downtime-says-ihs. Last accessed 04/17/2019.

[31] Christoph Molnar. 2019. *Interpretable Machine Learning*. https://christophm.github.io/interpretable-ml-book/.

[32] Vinod Nair, Ameya Raul, Shwetabh Khanduja, Vikas Bahirwani, Qihong Shao, Sundararajan Sellamanickam, Sathiya Keerthi, Steve Herbert, and Sudheer Dhulipalla. 2015. Learning a Hierarchical Monitoring System for Detecting and Diagnosing Service Issues. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 2029–2038.

[33] Vivek Nair, Zhe Yu, Tim Menzies, Norbert Siegmund, and Sven Apel. 2018. Finding Faster Configurations using FLASH. *IEEE Transactions on Software Engineering (Early Access)* (2018).

[34] Pankaj Prasad and Charley Rich. 2018. Market Guide for AIOps Platforms. https://www.gartner.com/doc/3892967/market-guide-aiops-platforms. Last accessed 04/17/2019.

[35] Anand Rajaraman and Jeffrey David Ullman. 2011. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA.

[36] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden Technical Debt in Machine Learning Systems. In *Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS)*. 2503–2511.

[37] Junji Shimagaki, Yasutaka Kamei, Shane McIntosh, Ahmed E. Hassan, and Naoyasu Ubayashi. 2016. A Study of the Quality-impacting Practices of Modern Code Review at Sony Mobile. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE)*. 212–221.

[38] Chakkrit Tantithamthavorn and Ahmed E. Hassan. 2018. An Experience Report on Defect Modelling in Practice: Pitfalls and Challenges. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 286–295.

[39] C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto. 2018. The Impact of Class Rebalancing Techniques on the Performance and Interpretation of Defect Prediction Models. *IEEE Transactions on Software Engineering (Early Access)* (2018).

[40] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. 2016. Automated Parameter Optimization of Classification Techniques for Defect Prediction Models. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. 321–332.

[41] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. 2017. An Empirical Comparison of Model Validation Techniques for Defect Prediction Models. *IEEE Transactions on Software Engineering* 43, 1 (Jan 2017), 1–18.

[42] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. 2018. The Impact of Automated Parameter Optimization on Defect Prediction Models. *IEEE Transactions on Software Engineering* (2018).

[43] Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. 2018. Deep Semantic Feature Learning for Software Defect Prediction. *IEEE Transactions on Software Engineering (Early Access)* (2018).

[44] J. Xue, R. Birke, L. Y. Chen, and E. Smirni. 2016. Managing Data Center Tickets: Prediction and Active Sizing. In *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 335–346.

[45] Ji Xue, Robert Birke, Lydia Y. Chen, and Evgenia Smirni. 2018. Spatial–Temporal Prediction Models for Active Ticket Managing in Data Centers. *IEEE Transactions on Network and Service Management* 15, 1 (2018), 39–52.

[46] Ji Xue, Bin Nie, and Evgenia Smirni. 2017. Fill-in the gaps: Spatial-temporal models for missing data. In *Proceedings of the 13th International Conference on Network and Service Management (CNSM)*. 1–9.

[47] Feng Zhang, Ahmed E. Hassan, Shane McIntosh, and Ying Zou. 2017. The Use of Summation to Aggregate Software Metrics Hinders the Performance of Defect Prediction Models. *IEEE Transactions on Software Engineering* 43, 5 (2017), 476–491.