Deep Learning or Classical Machine Learning? An Empirical Study on Log-Based Anomaly Detection

Boxi Yu
boxiyu@link.cuhk.edu.cn
School of Data Science, The Chinese University of Hong Kong, Shenzhen (CUHK-Shenzhen), China

Jiayi Yao
jiayiyao@link.cuhk.edu.cn
School of Data Science, The Chinese University of Hong Kong, Shenzhen (CUHK-Shenzhen), China

Qiuai Fu
fuqiuai@huawei.com
Huawei Cloud Computing Technologies CO., LTD.
China

Zhiqing Zhong
cheehingchung@gmail.com
School of Data Science, The Chinese University of Hong Kong, Shenzhen (CUHK-Shenzhen), China

Haotian Xie
haotianxie@link.cuhk.edu.cn
School of Data Science, The Chinese University of Hong Kong, Shenzhen (CUHK-Shenzhen), China

Yaoliang Wu
wuyaoliang1@huawei.com
Huawei Cloud Computing Technologies CO., LTD.
China

Yuchi Ma
mayuchi1@huawei.com
Huawei Cloud Computing Technologies CO., LTD.
China

Pinjia He*
hepinjia@cuhk.edu.cn
School of Data Science, The Chinese University of Hong Kong, Shenzhen (CUHK-Shenzhen), China

ABSTRACT

While deep learning (DL) has emerged as a powerful technique, its benefits must be carefully considered in relation to computational costs. Specifically, although DL methods have achieved strong performance in log anomaly detection, they often require extended time for log preprocessing, model training, and model inference, hindering their adoption in online distributed cloud systems that require rapid deployment of log anomaly detection service.

This paper investigates the superiority of DL methods compared to simpler techniques in log anomaly detection. We evaluate basic algorithms (e.g., KNN, SLFN) and DL approaches (e.g., CNN) on five public log anomaly detection datasets (e.g., HDFS). Our findings demonstrate that simple algorithms outperform DL methods in both time efficiency and accuracy. For instance, on the Thunderbird dataset, the K-nearest neighbor algorithm trains 1,000 times faster than NeuralLog while achieving a higher F1-Score by 0.0625. We also identify three factors contributing to this phenomenon, which are: (1) redundant log preprocessing strategies, (2) dataset simplicity, and (3) the nature of binary classification in log anomaly detection. To assess the necessity of DL, we propose LightAD, an architecture that optimizes training time, inference time, and performance score. With automated hyper-parameter tuning, LightAD allows fair comparisons among log anomaly detection models, enabling engineers to evaluate the suitability of complex DL methods.

Our findings serve as a cautionary tale for the log anomaly detection community, highlighting the need to critically analyze datasets and research tasks before adopting DL approaches. Researchers proposing computationally expensive models should benchmark their work against lightweight algorithms to ensure a comprehensive evaluation.

KEYWORDS

Log analysis, anomaly detection, dataset, empirical study

ACM Reference Format:
Boxi Yu, Jiayi Yao, Qiuai Fu, Zhiqing Zhong, Haotian Xie, Yaoliang Wu, Yuchi Ma, and Pinjia He. 2024. Deep Learning or Classical Machine Learning? An Empirical Study on Log-Based Anomaly Detection. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3623308

1 INTRODUCTION

Over the past decade, many online large-scale software systems have been developed to enhance our daily lives, including search engines, social media platforms, and machine translation systems. These online software systems must be available on a 24/7 basis, as any downtime may result in user dissatisfaction and significant revenue loss, especially for large-scale distributed systems designed to serve millions of users. In fact, according to "Downtime, Outages and Failures - Understanding Their True Costs" [22], a critical application failure can result in loss of application service and data, costing companies up to $300,000 per hour in web application downtime or $5,600 per minute. Well-known companies have suffered significant revenue losses due to large-scale web service downtime.

*Pinjia He is the corresponding author.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3623308

For example, Amazon reported a loss of $2,646,501 in revenue during a 13-minute downtime episode [74], while Facebook suffered a loss of approximately $426,607 in revenue during a 19-minute site-wide outage [73].

System logs are crucial in the development and maintenance of modern software, and are widely used to detect anomalies in cloud systems. However, as the volume of logs continues to grow, classical log analysis approaches that rely on manual inspection have become inefficient and time-consuming. For example, some modern cloud systems can produce 50 gigabytes (around 200 million lines) of logs per hour [60]. To address this challenge, many automatic log analysis methods have been proposed, including classical machine learning (ML)-based methods [9, 10, 15, 47–49, 51, 80] and DL methods [21, 24, 52, 56, 62, 85]. Some DL methods claim to outperform classical methods on datasets such as BGL and Thunderbird [63]. Log data needs to be transformed into vectors before it can be fed into a model. To this end, various log vectorization methods have been proposed, including static code analysis [80], log parsers [31, 75], and neural representations [42] using BERT [19]. However, it is unclear whether these methods are necessary, as no analysis has compared them with a simple baseline method that eliminates variables in the log and tokenizes the log into a set of word tokens.

Despite the satisfactory performance of SOTA DL methods, the data preprocessing, training, and inference processes often consume a significant amount of time and require high-demand hardware. Taking inspiration from Fu and Menzies' suggestion [29] that "it is a good practice to explore simple and fast techniques before applying DL methods on SE tasks", we aim to investigate whether simple methods can achieve comparable performance to DL methods in log anomaly detection. This leads to our first research question:

RQ1: Do DL methods have advantages over simple ones on log anomaly detection?

To answer the research question of whether DL methods have advantages over simple ones on log anomaly detection tasks, we conducted experiments on five popular log anomaly detection datasets: HDFS [80], BGL, Spirit, ThunderBird, and Liberty [63]. Some DL methods have claimed to achieve excellent performance on these datasets. However, our study found that simple methods can achieve the best performance on each of the five log datasets in terms of evaluation metrics and time efficiency. We try to explain why simple log vectorization methods and simple ML-Based models can easily beat the intricate DL models. Therefore, we ask our second research question:

RQ2: Why do DL methods not outperform simple ones?

Our investigation has uncovered three fundamental reasons why basic log vectorization techniques and simple machine learning-based models can surpass complex DL models with ease. First, the redundancy of log preprocessing strategies employed by some approaches, such as grouping multiple lines of log messages to generate a session window, hampers the effectiveness of DL models. Our analysis of supercomputer log datasets such as BGL, Thunderbird, and Spirit [63] indicates that these datasets are labeled in a line-by-line manner, making redundant preprocessing unnecessary. Second, the simplicity of log datasets can lead to varying degrees of data leakage caused by recurring log patterns among the datasets, which aligns with the finding in [43]. Our findings reveal that direct log sequence/message matching can achieve comparable or even better results than SOTA deep methods on HDFS and Spirit. Additionally, after eliminating the recurring log sequences in HDFS, simple log anomaly detection methods can still achieve relatively high F_1-Score (0.9406-0.9410) compared to deep methods (0.8017-0.9244). Third, the innate nature of binary classification in log anomaly detection may also contribute to the superiority of simple binary classification methods over DL models specifically engineered for log anomaly detection. To further explore the underlying reasons for this phenomenon, we provide a detailed examination of binary classification in log anomaly detection and highlight the advantages of utilizing simple binary classification techniques. Our research findings underscore the importance of leveraging simple and effective approaches for log anomaly detection.

In our analysis of both RQ1 and RQ2, we observed that intricate DL methods failed to surpass their simpler counterparts across all five public log datasets. This raises a critical question about the necessity of employing costly DL methods in log anomaly detection, thereby giving our third research question:

RQ3:When should we use DL methods?

We carefully examined recent research papers on DL anomaly detection methods [42, 52] and discovered that their claims of the superiority of DL methods may not always be valid due to three factors: (1) weak baseline comparison, (2) potential ineffective hyper-parameter tuning, and (3) neglect of time efficiency. In this paper, we propose an approach named LightAD that optimizes both performance and time efficiency. With automated hyper-parameter tuning using Bayesian optimization, LightAD provides a fair comparison of various log anomaly detection methods.

On the deduplicated HDFS, LightAD consistently recommends the use of simple methods, regardless of the prioritized objective, such as F_1-Score or inference time. Our experimental results caution the log anomaly detection community to carefully evaluate new innovations before applying them, especially for cloud web applications serving millions of users. Before deploying new and expensive processes, it is important to compare them with simpler and faster alternatives. Moving forward, we are interested in testing LightAD on additional datasets to determine whether complex log grouping or DL methods are necessary for log anomaly detection. We believe LightAD can serve as a strong baseline for future log studies.

There are four main contributions of this work:

- We show that complex DL methods and the related log preprocessing techniques do not necessarily have advantages over simple methods in both accuracy performance and time efficiency on the current log anomaly detection benchmarks.
- We summarize three reasons why DL methods do not outperform simple methods: (1) redundant log data preprocessing strategies, (2) simplicity of current log benchmarks, and (3) the innate nature of binary classification in log anomaly detection.
- We propose LightAD, a framework that automatically tunes the hyper-parameters to optimize objectives leveraging both accuracy and efficiency, and show its practical ability to verify the necessity of DL methods.

- We give a cautionary tale that critical analysis should be conducted on the tasks before applying costly AI models.

2 BACKGROUND

2.1 Log-based Anomaly Detection Workflow

The overall framework of log anomaly detection is presented in Fig. 1, which includes two mainstream methods: methods utilizing log parsing and methods without parsing. Log anomaly detection methods with log parsing comprise four steps: log collection, log parsing, log grouping, and feature extraction. On the other hand, log anomaly detection methods without log parsing consist of three steps: log collection, preprocessing, and neural representation before anomaly detection. In this section, we will introduce the components of these two mainstream methods.

Log Collection. Logs serve as historical records of runtime information in software systems. In modern large-scale distributed systems, logs play a crucial role in diagnosing systems, but the overwhelming volume of logs can be daunting. Thus, log collection is the first step of automatic log anomaly detection systems.

Log Parsing. Raw log data are semi-structured and need log parsing tools to be parsed into structured data that enable downstream log analysis tasks, including usage analysis, anomaly detection, duplicate issue identification, performance modeling, and failure diagnosis [89]. Some works [61, 80] use static analysis techniques to extract event templates from the source code. For closed-source software, several data-driven methods have been adopted to parse logs, such as SLCT [75], LogCluster [76], IPLoM [53], LKE [28], Spell [20], and Drain [31].

Log Vectorization. As log data are natural language sentences, they need to be processed into vectors before being fed into the models. There are two categories of log vectorization, one is log vectorization with log parsing, and the other is without log parsing. For vectorization with log parsing, log grouping is the first step. There are three log grouping methods, including fixed-window, sliding-window [43], and session window [42, 80]. Session windows, which are frequently adopted by classical ML-based methods, are based on identifiers. For instance, HDFS [80] employs aggregate log by "blk_id" to generate session window which counts the events. For DL log anomaly detection methods, they will convert the logs into three main types of vectors [43], which are sequential vectors, quantitative vectors, and semantic vectors. For example, DeepLog [21] assigns each log event with an index, and then generates sequential vectors with a certain window size. The sequential vectors record the execution path of the log event. The quantitative vectors are similar to the log count vectors, which record the occurrence of each log event in a log window. The semantic vectors use a language model to represent the semantic meaning of log events and convert the log windows into semantic vectors.

Inspired by the bag-of-words model, *Term Frequency / Inverse Document Frequency* (TF-IDF), a well-established heuristic in information retrieval [64, 69], is always employed as a term weighting method to calculate the importance of words in log events. For log vectorization without log parsing, NeurlLog [42] is proposed to encode log by using BERT [19] to generate semantic vectors and feed the vectors into a Transformer-based classifier [77].

2.2 Representative Approaches

To choose the representative approaches, we reviewed a series of experience reports [16, 33], empirical study [43], surveys [32, 39], and the SOTA methods [21, 42, 52, 85] on log anomaly detection. Various methods are being utilized for log anomaly detection, consisting of simple ML-Based models and heavily weighted DL models. We consider the performance of these models on several log anomaly detection benchmarks and choose three top-notch DL models with the best $F1$ *score*, i.e., CNN [52], NeuralLog [42], LogRobust [85] and compare them with some commonly used simple models. For simple models, we adopt K-nearest neighbor (KNN), single hidden layer feed-forward neural network (SLFN), and decision tree (DT) for log anomaly detection.

2.2.1 Classical ML-based Methods. **K-nearest neighbor (KNN)** [27] is one of the most fundamental and simple classification methods. In our implementation, the training phase of KNN is to store the feature vectors and class labels of the log data. For the inference phase, the prediction of a log vector is assigned the label which aligns the majority of the labels among the nearest k training samples to that query point.

SLFN. A multilayer perceptron (MLP) is a class of fully connected feedforward neural network, which utilizes back-propagation for training. Single hidden layer feed-forward neural network (SLFN) is the most naive MLP, which only contains a single hidden layer.

Decision Tree. Decision tree (DT) is like a decision support tool that uses a tree structure with several branches to illustrate the predicted state for each instance. Decision tree was first adopted to diagnose failures in large internet sites by Chen et al. [15].

2.2.2 Deep Learning Methods. **CNN.** Lu et al. [52] propose an approach for log anomaly detection by leveraging Convolutional Neural Networks (CNN). The CNN-based model can learn event relationships automatically and achieve high accuracy. The deep CNN-based model consists of *logkey2vec* embeddings, three 1D convolutional layers, dropout layer and max-pooling.

LogRobust. To tackle the challenge that log data often contain unseen instances due to the update of log statements, Zhang et al. [85] propose LogRobust to extract the semantic information of the log data. Furthermore, LogRobust incorporates the attention mechanism [5] into the Bi-LSTM model to assign different weights to log events.

NeuralLog. As the first approach without log parsing, NeuralLog [42] preprocesses the log messages into a set of words. Then it uses WordPiece tokenization [70, 79] to handle the OOV words, and uses a pre-trained BERT [19] to obtain the semantic meaning of log messages. To better understand the semantics of logs, NeuralLog adopts the transformer [77] model rather than RNN-based models.

Two main factors contribute to SVM, DT, and SLFN being considered simpler than CNN [52], LogRobust [85] (based on LSTM), and NeuralLog [42] (based on BERT). **(1) Architecture complexity**: KNN, DT and SLFN have relatively simple architectures compared to CNNs, LSTMs, and BERT. KNN only involves a distance metric and the selection of k nearest neighbors to make a prediction. DT uses a decision tree structure to recursively partition the input feature space into smaller regions. SLFN only contains one hidden

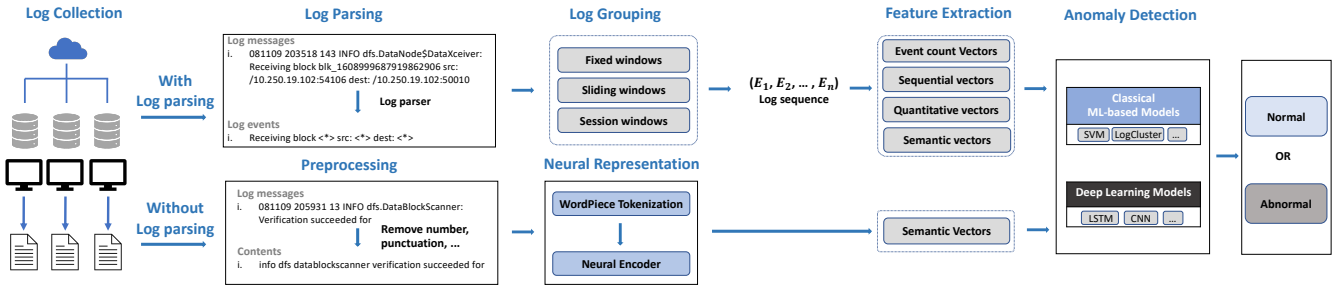

Figure 1: Overview of log anomaly detection workflow

layer, whose network depth is much shallower than the DL architecture. **(2) Computation cost:** KNN, DT, and SLFN are relatively computationally efficient, especially during the inference phase. In contrast, CNNs, LSTMs, and BERT are computationally expensive due to their complex architectures and large number of parameters.

3 EXPERIMENTAL SETUP

3.1 Research Questions

In this study, we aim to demonstrate that DL methods may not always offer advantages over other approaches in log anomaly detection. To accomplish this, we investigate three research questions:

- **RQ1** Do DL methods have advantages over simple ones on log anomaly detection?
- **RQ2** Why do intricate anomaly detection methods not outperform simple ones? outperform simple ones?
- **RQ3** When do we need really DL methods?

3.2 Experimental Environments

The experiments are run on a Linux workstation with a 192GB Memory, and GeForce RTX A6000 GPU. The Linux workstation is running 64-bit Red Hat 4.8.5-28.

3.3 Dataset

Based on our review of experience reports [16, 33], empirical studies [43], and surveys [32, 39], we found that most of the log anomaly datasets only contains binary labels, namely, normal and abnormal. As a result, log anomaly detection is treated as a binary classification task. In this paper, we evaluate log anomaly detection methods using five commonly used public datasets: HDFS [80], Blue Gene/L (BGL), Thunderbird, Liberty, and Spirit [63].

The HDFS dataset, collected by Xu et al. [80] on the Amazon EC2 platform, consists of 11,175,629 log messages. The logs are segmented into log sequences using block IDs and assigned a ground truth label of *normal* or *abnormal*. The BGL, Thunderbird, Liberty, and Spirit datasets are message-wise log datasets collected from supercomputers, and their statistical information is provided in Table 1. The logs contain alert and non-alert messages, which are distinguished by alert category tags. In the first column of the log, '-' denotes non-alert messages while the rest indicate alert messages. The alert and non-alert messages are considered *abnormal* and *normal*, respectively. Given the substantial size of the Liberty, Spirit, and Thunderbird datasets, consisting of over 200 million messages [63], we opted to selectively include a modest subset of 3.7%, 3.9%, and 4.7% respectively, employing a chronological selection strategy. For HDFS datasets, we randomly split them into training and testing sets with a ratio of 8:2. As for the supercomputer log datasets, we split them into training and testing sets with a ratio of 8:2 in a chronological manner.

Table 1: Descriptive Statistics of log datasets

Dataset	Category	Label Granularity	#Messages	#Anomalies
HDFS	Distributed system	block-wise	11,175,629	16,838
Blue Gene /L	Supercomputer	message-wise	4,747,963	348,460
Thunderbird	Supercomputer	message-wise	10,000,000	353,794
Spirit	Supercomputer	message-wise	7,983,345	768,142
Liberty	Supercomputer	message-wise	10,000,000	3,256,972

3.4 Evaluated Models

In our experiments, we focus on revisiting the current log-based anomaly detection methods; therefore, we evaluate several representative classical ML-based methods and DL methods. For simple methods, we evaluate DT [15] SLFN [34], and KNN [27]. For DL Based methods, we evaluate CNN [52], LogRobust [85], and NeuralLog [42]. We evaluate DT, LogRobust, NeuralLog, and CNN by using the open-sourced implementation [1, 2, 42]. For KNN and SLFN, we implement them by ourselves. or DL methods with the parser, we used Drain [31], a widely adopted parser with good performance for log preprocessing. For simple methods, we preprocess the log by tokenizing the log and removing the digits, instead of adopting the parser. Specifically, we construct event frequency vectors for block-wise dataset by grouping the logs with the block ID. We evaluate each model five times and take their average results.

3.5 Evaluation Metrics

To measure the effectiveness of the anomaly detection methods, we use the Precision, Recall, and F_1-Score metrics, represented by the abbreviations P, R, F1, respectively. We characterize their outcome as True Positive (TP), True Negative (TN), False Positive (FP), False Negative (FN). TP represents the number of abnormal log sequences that are correctly detected by the anomaly detection methods. TN represents the number of normal log sequences that are correctly detected by the anomaly detection methods. FP represents the number of log sequences that are mistakenly identified as anomalies.

FN represents the abnormal log sequences that are not detected by the anomaly detection methods. The evaluation metrics are defined as below:

- *Precision*: the ratio between number of correctly detected anomalies and number of detected anomalies: $Prec = \frac{TP}{TP+FP}$
- *Recall*: the ratio between number of correctly detected anomalies and total number of anomalies: $Recall = \frac{TP}{TP+FN}$
- *F1 score*: the harmonic mean between *Precision* and *Recall*: $F1\ score = \frac{2*Prec*Recall}{Prec+Recall}$

4 RESULTS AND DISCUSSIONS

4.1 RQ1: Do DL Methods Have Advantages Over Simple Ones on Log Anomaly Detection?

To address this question, we utilize simple anomaly detection models (KNN, DT, SLFN) and compare them with three SOTA DL methods (CNN [52], LogRobust [85], NeuralLog [42]) in terms of both accuracy and runtime efficiency. For the simple approaches, we utilize different preprocessing strategies for block-wise and message-wise log datasets.

We process the block-wise dataset (HDFS [80]) by extracting the tokens from each text log message (excluding header) split by spaces and removing any tokens containing digits. We group the log messages into log sequences using blk_{id} and encode them with the event frequency. The entire preprocessing workflow is illustrated in Fig. 2.

For the message-wise datasets (BGL, Thunderbird, Spirit, and Liberty), our anomaly detection method is even simpler. Fig. 3 illustrates the approach, where we begin by tokenizing the log messages using the same way as in the HDFS dataset. However, we do not convert the tokenized log messages into numerical vectors. Rather, we calculate the Jaccard distance to measure the distance between log messages:

$$Jaccard(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}, \quad (1)$$

where A and B each represent the set of a single tokenized log message. We then pair each tokenized log message in the testing set with its nearest log message in the training set. The predicted label of a log message in the testing set is aligned with that of the paired log message in the training set. To ensure fairness when comparing our simple methods with the DL methods, we set the window size and stride to 10 in the evaluation stage, as in the DL methods.

For the DL methods, we follow the preprocessing methods and hyper-parameters settings in the original paper. We empirically tune the parameters by grid search with cross-validation approach if the hyper-parameters are not provided in the paper or the method hasn't been tested on a specific dataset. For the simple methods, we also tune the hyper-parameters by grid search with cross-validation. Specifically, we set the number of neighbors as 1 in KNN.

4.1.1 Comparison on Accuracy. Table 2 displays the performance scores of both deep methods (CNN [52], LogRobust [85], NeuralLog [42]) and simple methods (KNN, DT, SLFN). Due to the incompatibility of DT and SLFN with the data format derived from our specific preprocessing methods for message-wise datasets, these techniques were not evaluated on these datasets. Consequently, their corresponding evaluation metrics are The highest scores are indicated in bold font. Overall, it is evident that the simple methods outperform the complex ones. Across all five datasets, there is at least one simple method that can achieve the highest precision rate, recall rate, and F_1-Score. Only on liberty, NeuralLog can achieve the perfect score, drawing with KNN.

Fig. 4 compares the performance delta between the best simple methods and the best deep methods. Liberty is not included in Fig. 4 for the best deep method (NeuralLog) and KNN can both achieve perfect scores. Simple methods beat the deep ones on all the other four datasets with respect to all the evaluation metrics. Note that the highest precision, recall, or F_1-Score may not come from the same model.

To show the superiority of the classical ML methods, we compare them with DL methods on a more challenging benchmark, i.e., we only use 1% of the data in the train dataset for training the model. The results are shown in Table 3, where we can see that many DL methods' performance is largely undermined. For instance, CNN achieved a mere 0.0440 F_1-Score on Spirit, while LogRobust [85] achieved only 0.5579 F_1-Score on BGL. In contrast, KNN demonstrated a range of F_1-Scores from 0.9135 to 1.000. These findings indicate that the simpler methods exhibit greater robustness when faced with the challenges presented by this benchmark.

Table 2: Overall Performance on five public datasets

Dataset		CNN	LogRobust	NeuralLog	KNN (N=1)	DT	SLFN
HDFS	P	0.9840	0.9858	0.9942	0.9986	**0.9988**	0.9962
	R	0.9895	0.9890	0.9970	**1.0000**	0.9991	0.9962
	F1	0.9867	0.9874	0.9956	0.9988	**0.9990**	0.9962
BGL	P	0.5988	0.6120	0.9493	**0.9747**	N/A	N/A
	R	0.9749	0.9749	0.9796	**0.9858**	N/A	N/A
	F1	0.7419	0.7520	0.9641	**0.9802**	N/A	N/A
Thunder-bird	P	0.9680	0.9613	0.9922	**0.9939**	N/A	N/A
	R	0.8785	0.9025	0.8833	**1.0000**	N/A	N/A
	F1	0.9211	0.9310	0.9345	**0.9970**	N/A	N/A
Spirit	P	-	-	0.9770	**1.0000**	N/A	N/A
	R	-	-	0.9414	**1.0000**	N/A	N/A
	F1	-	-	0.9587	**1.0000**	N/A	N/A
Liberty	P	0.9999	0.9985	**1.0000**	**1.0000**	N/A	N/A
	R	0.9999	0.9985	**1.0000**	**1.0000**	N/A	N/A
	F1	0.9936	0.9932	**1.0000**	**1.0000**	N/A	N/A

'-' denotes "out of memory" error, N/A denotes not applicable.

4.1.2 Comparison on Time Efficiency. Table 4 illustrates both the training time and inference time of all the methods we evaluate. We use bold fonts to indicate the least runtime. In general, simple methods run faster than deep methods on all five datasets in terms of both training time and inference time. On HDFS, KNN has the shortest training time (e.g., 3,225X faster than NeuralLog) while DT has the shortest inference time (e.g., 185X faster than NeuralLog). In the case of the other four supercomputer datasets, we also see a significant efficiency advantage with our KNN approach during both the training and inference stages. For instance, KNN's training time on BGL is 1, 278X faster than NeuralLog, and KNN's inference time on BGL is 23X faster than NeuralLog. Even though our experiment is conducted with 192GB RAM, CNN and LogRobust still cause "out of memory" error on Spirit, indicating deep methods' high demand for hardware.

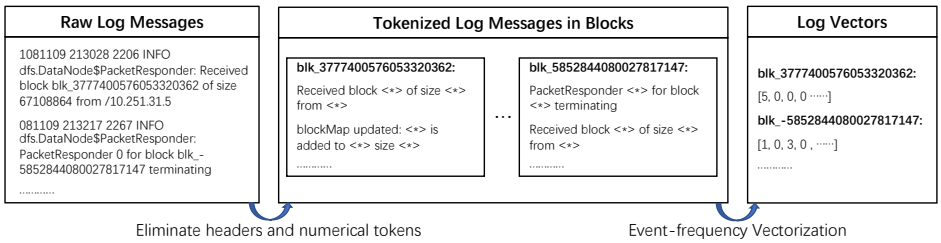

Figure 2: Preprocessing of block-wise datasets for simple methods

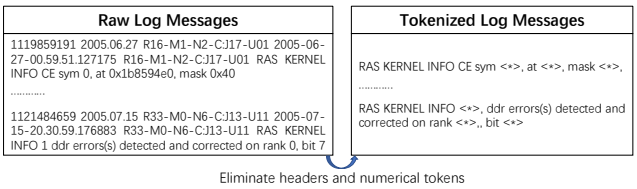

Figure 3: Preprocessing of message-wise datasets for simple methods

Table 3: Overall Performance on five public datasets (1% training dataset)

Dataset		CNN	LogRobust	NeuralLog	KNN (N=1)	DT	SLFN
HDFS	P	0.8782	0.9104	0.9387	0.9824	**0.9796**	0.9537
	R	0.7601	0.8498	0.8246	0.9713	**0.9917**	0.8032
	F1	0.8177	0.8747	0.8766	0.9766	**0.9856**	0.8714
BGL	P	0.3792	0.4869	0.7903	**0.9935**	N/A	N/A
	R	0.9028	0.7661	0.9324	**0.9754**	N/A	N/A
	F1	0.5055	0.5579	0.8258	**0.9844**	N/A	N/A
Thunder-bird	P	0.4435	0. 9492	**0.9997**	0.9912	N/A	N/A
	R	0.8559	0.8277	0.8790	**0.9048**	N/A	N/A
	F1	0.5728	0.8836	0.9354	**0.9455**	N/A	N/A
Spirit	P	0.0979	0.0307	0.4039	**0.9193**	N/A	N/A
	R	0.0651	0.0368	0.3126	**0.9078**	N/A	N/A
	F1	0.0440	0.0321	0.3430	**0.9135**	N/A	N/A
Liberty	P	0.9479	0.9746	**1.0000**	**1.0000**	N/A	N/A
	R	0.9844	0.9895	**1.0000**	**1.0000**	N/A	N/A
	F1	0.9657	0.9817	**1.0000**	**1.0000**	N/A	N/A

N/A denotes not applicable.

Table 4: Training and Inference Efficiency on five public datasets (seconds)

Dataset		CNN	LogRobust	NeuralLog	KNN (N=1)	DT	SLFN
HDFS	Train	74.7774	41.4481	6,044.0218	1.8742	2.3986	29.8954
	Infer	2.7748	2.2360	55.4187	2.3104	**0.2994**	0.4844
BGL	Train	75.0809	63.3461	5,188.3536	**4.0601**	N/A	N/A
	Infer	3.2555	3.2807	72.9399	**3.1934**	N/A	N/A
Thunder-bird	Train	581.3999	592.0321	10,218.6227	**10.2145**	N/A	N/A
	Infer	30.8276	29.6045	85.5081	**10.4084**	N/A	N/A
Spirit	Train	-	-	6,626.9311	**7.5750**	N/A	N/A
	Infer	-	-	110.3898	**2.1295**	N/A	N/A
Liberty	Train	440.0547	473.2985	10,187.34908	**9.6261**	N/A	N/A
	Infer	26.0857	30.0932	94.7462	**4.3901**	N/A	N/A

'-' denotes "out of memory" error, N/A denotes not applicable.

Table 5 compares the runtime of different log preprocessing strategies. The least preprocessing time is marked in bold. CNN and LogRobust both utilize a log parser to generate log templates from which log semantics is extracted. NeuralLog adopts a neural representation method to preprocess raw logs into log vectors without

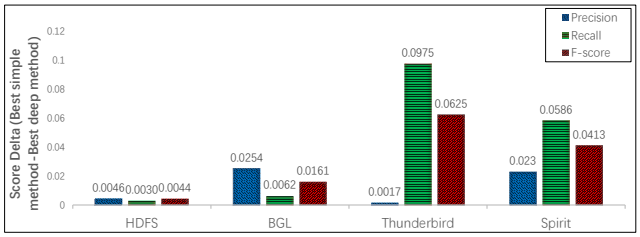

Figure 4: Comparison of performance scores between the best simple method and the best deep method

parsing. Our naive preprocessing has the least runtime on all the five datasets we use. Our naive preprocessing strategies for simple methods run $4X$ to $12X$ faster than the neural representation on the five log datasets, and $13.6X$ to $19.3X$ faster than the log parser with semantics extraction on the four log datasets, where the time for preprocessing Spirit via the log parser is omitted due to "out of memory" error.

RQ1 Answer: DL methods are neither better in performance scores nor runtime efficiency than simple methods on log anomaly detection.

Table 5: Preprocessing Efficiency on five public datasets (seconds)

Method	HDFS	BGL	Thunderbird	Spirit	Liberty
Parser+Semantics	980.2582	434.9437	1,114.0291	-	1,244.7097
Neural representation	289.1004	205.1231	323.4426	568.4599	484.8715
Ours	**71.8927**	**23.7540**	**75.1366**	**47.0459**	**64.4417**

'-' denotes "out of memory" error

4.2 RQ2: Why Do Intricate Anomaly Detection Methods Not Outperform Simple Ones?

To gain insight into the reasons behind the comparable, and oftentimes superior, performance of simplistic binary classifiers over intricately designed DL methods in the context of log-based anomaly detection, we propose three potential explanations: (1) the utilization of redundant log preprocessing strategies, (2) the relatively straightforward nature of the datasets, and (3) the intrinsic characteristics of binary classification tasks.

4.2.1 Redundant Log Preprocessing Strategies. As shown in RQ1, complicated log preprocessing methods are much slower than our simple preprocessing methods. Furthermore, deep methods based

on complex log preprocessing do not show any advantage over their simple counterparts in terms of both performance score and runtime efficiency. In addition to the uncalled-for log vectorization methods, log grouping strategies should be carefully examined in accordance with the granularity of labels. For HDFS dataset [80], most anomaly detection methods aggregate logs with the identifier, which is reasonable for the fact that Xu et al. [80] labeled the dataset on the block level. However, the reason why the recent line of research adopts session window grouping on supercomputer datasets is unclear. According to the original paper [63] that introduces the supercomputer datasets, the labeling procedure is performed on log message level with a combination of regular expressions and system administrators' intervention. Therefore, it is reasonable for us to challenge the validity of window-wise log grouping in anomaly detection. As shown in Table 2, KNN with our simple log preprocessing strategies outperforms the DL methods, especially for the message-wise datasets. For instance, on Thunderbird, the F_1-Score of KNN is 0.0625 higher than the best DL method NeuralLog. This could support our claim that message-wise datasets shouldn't be aggregated by windows. As for NeuralLog, its performance disparity with simple methods remains relatively stable on different datasets (F-score difference: 0-0.0625). However, to further show that window aggregation is unnecessary, we re-run NeuralLog with a window size of 1 on BGL, Thunderbird, and Spirit. Liberty is not used for NeuralLog can already achieve perfect score with window size 10. Fig. 5 shows the score delta of NeuralLog when setting its window size to 1 and 10, respectively. The F_1-Scores on the three datasets are improved by 0.0007, 0.0401, and 0.0077, which demonstrate that the session window is useless. Although avoiding grouping improves the performance of NeuralLog, its F_1-Scores on the three datasets are still lower than KNN.

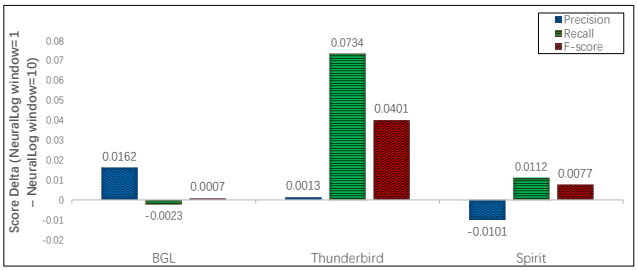

Figure 5: Comparison of NeuralLog performance scores when setting window to 1 and 10

4.2.2 Simplicity of the Datasets. The overwhelming extent of data leakage might be a potential reason why simple methods can achieve extremely high scores. Note that different preprocessing strategies may lead to different extent of data leakage. Thus, for all five datasets, we first preprocess the data with our naive strategy and log parser, respectively. Then we record the number of test data that have/haven't appeared in the training set in the second and third column in Table 6. We further measure the extent of data leakage with the percentage of leaked data in the fourth column. Except for BGL, data leakage on the other four datasets all exceeds 98%. To show how data leakage can make anomaly detection easier, we adopt a very naive matching algorithm: (1) if the test data has an identical match in the training set, it should have the same label as the matched training data, (2) otherwise, we randomly predict the label with uniform distribution (p=0.5). As shown in the last column of Table 6, the naive matching algorithm can achieve an F_1-Score of 0.9968 with both our naive preprocessing and log parser on HDFS. Combining with Table 2, we can see naive match can outperform all the intricate deep models! The naive match method achieves a perfect score on the Liberty dataset, just like NeuralLog, and slightly outperforms CNN and LogRobust. While the naive match falls short in achieving satisfactory scores on BGL, Thunderbird, and Spirit, the presence of a substantial amount of leaked data in the test dataset might still aid in anomaly detection.

Table 6: Extent of Data leakage and results with naive match

Dataset		# Unique	# Duplicate	% Duplicate	F-score by naive match
HDFS	ours	46	114,967	0.9996	0.9968
	parser	46	114,967	0.9996	0.9968
BGL	ours	426,475	523,118	0.5509	0.2222
	parser	425,749	516,950	0.5484	0.2244
Thunder-bird	ours	10,791	1,989,209	0.9946	0.8726
	parser	37,095	1,957,932	0.9814	0.6659
Spirit	ours	295	1,596,374	0.9998	0.8926
	parser	4,537	1,587,210	0.9971	0.3512
Liberty	ours	1	1,999,999	1.0000	1.0000
	parser	58	1,989,720	1.0000	1.0000

To further investigate whether data leakage is a major contributor to the high anomaly detection accuracy, we eliminate the repetitive log sequences in the HDFS dataset. We use the ground truth template to perform the elimination process, and the deduplicated HDFS only 589 log vectors. Then we run all the methods again on the dataset without data leakage. However, we are unable to perform the same elimination process on the four supercomputer datasets in that the anomaly detection should be conducted on the log message level, and eliminating duplicate log messages seems to be far divorced from real-world scenarios.

Table 7 displays the results obtained by various methods on the original HDFS, deduplicated HDFS, and deduplicated HDFS with Bayesian tuner. The highest scores are highlighted in bold. Notably, all methods experience a decline in performance when duplicate log sequences are eliminated. This decline is more pronounced in the case of deep methods compared to simpler ones. While the F_1-Scores for all three simple methods remain above 0.9, deep methods achieve an F_1-Score of approximately 0.8. It's worth mentioning that we adhere to the hyperparameters outlined in RQ1. However, different methods may exhibit varying sensitivity to their respective hyperparameters. Consequently, we fine-tune the hyperparameters for each method using Bayesian optimization [72]. Tables 8 and 9 provide details on the hyperparameters and their tuning ranges for simple methods and deep methods, respectively. As demonstrated in Table 7, Bayesian tuning significantly enhances the F_1-Scores for all methods. All simple methods achieve an F_1-Score of approximately 0.94. Among the deep methods, CNN experiences the most substantial improvement, increasing its F_1-Score from 0.8011 to 0.9244, while reducing the F_1-Score gap with the best simple method (i.e., DT) from 0.1389 to 0.0166. However, LogRobust and NeuralLog only achieve F_1-Scores of 0.8017 and 0.8464, respectively.

In summary, simple methods continue to outperform deep methods on deduplicated log dataset.

Table 7: Comparison of results on original hdfs and deduplicated hdfs

Dataset		CNN	LogRobust	NeuralLog	KNN	DT	SLFN
Original HDFS	P	0.9840	0.9858	0.9971	0.9986	**0.9988**	0.9962
	R	0.9895	0.9890	0.9984	**1.0000**	0.9991	0.9962
	F1	0.9867	0.9874	0.9977	0.9988	**0.9990**	0.9962
Deduplicated HDFS	P	0.6690	0.6644	0.6847	0.8975	**0.9029**	0.8871
	R	**1.0000**	**1.0000**	0.9707	0.9263	0.9816	1.0000
	F1	0.8011	0.7979	0.8005	0.9105	**0.9400**	0.9399
Deduplicated HDFS with Bayesian Tuner	P	**0.9009**	0.6710	0.8218	0.8945	0.8967	0.8930
	R	0.9542	**0.9975**	0.8818	0.9922	0.9920	0.9947
	F1	0.9244	0.8017	0.8464	0.9406	**0.9410**	0.9407

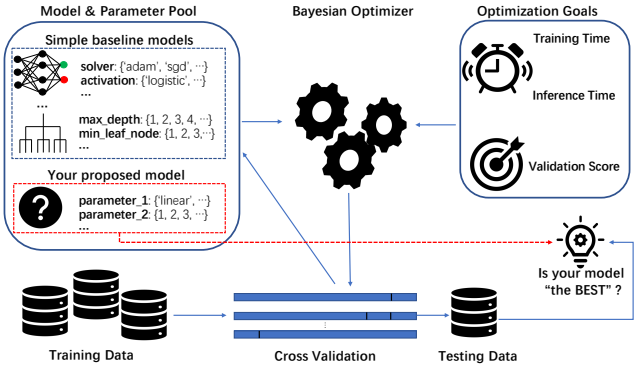

Figure 6: Workflow of LightAD

4.2.3 Innate Nature of Binary Classification in Log Anomaly Detection. The task of binary classification can invariably be conceptualized as the partitioning of a multi-dimensional space into two distinct segments, achieved through the use of one or more hyperplanes. A pertinent question arises in log anomaly detection: Can binary classification inherently be addressed through the application of simplistic methods? Simple non-parametric methods (e.g., KNN, DT) should always have good results as long as the data characteristics of train data do not deviate much from the test data. According to Huang et al. [34], SLFN with a non-constant bounded activation function (e.g., sigmoid) can form disjoint decision regions with arbitrary shapes in a multi-dimensional space. This conclusion can be further generalized to certain non-bounded activation functions [34]. In the context of binary classification in log anomaly detection, the log data, owing to the inherent patterns exhibited in the logging systems, may invariably encompass recurring log data within the corpus. As a result, simple methods such as KNN, DT, and SLFN are considered potential candidate methods for effectively addressing the binary classification task of log anomaly detection. Therefore, it is unnecessary to adopt intricate time-consuming deep methods before the attempt to adopt time-consuming DL methods. Moreover, the intricate network may sometimes backfire on a simple binary classification task. As shown in Table 7, even with optimal hyper-parameters, the F_1-Score of LogRobust is only 0.8017, much worse than the average F_1-Score of 0.94 for the simple methods on deduplicated HDFS. LogRobust leverages the sequential information of log event occurrence by adopting LSTM architecture. However, the order of log event occurrence might be misleading for anomaly detection since HDFS is labeled with respect to event count vectors [80].

RQ2 Answer: Intricate log anomaly detection methods cannot outperform the simple methods due to (1) redundant log preprocessing strategies, (2) simplicity of dataset, (3) the innate nature of binary classification in log anomaly detection.

4.3 RQ3: When Do We Really Need DL Methods?

In the field of log-based anomaly detection, most research [42, 52, 85] do not compare the proposed models with their naive counterparts appropriately.

In the context of log-based anomaly detection, or in a wider context of binary classification tasks, complex models might always be inferior to their naive counterparts. Bear in mind that the inferiority may come from both accuracy and efficiency. However, there are several reasons why most research [52, 85] claim their proposed models are better than the naive baseline models: (1) ignored baseline models, (2) potential ineffective hyper-parameter tuning, (3) overlook in time efficiency.

Many naive binary classifiers are not included in the comparison when a new model is proposed. LogRobust [85] and NeuralLog [42] compare themselves with only two simple methods, LR [9] and SVM [47]. None of the simple methods (KNN, DT, SLFN) we adopt is included in their comparison. CNN [52] doesn't include any naive binary classifiers in the paper.

Ineffective parameter tuning may provide a false conclusion that the proposed model is better than the naive ones. For instance, Le et al. [42] claim that NeuralLog can achieve an F_1-Score of 0.98 while SVM can only achieve an F_1-Score of 0.96. Note that they employ Loglizer [2] for SVM implementation. Loglizer's SVM is implemented with sklearn.LinearSVC [66] which only supports linear kernels. The F_1-Score on HDFS can be easily boosted to 0.99 by adopting a polynomial kernel with sklearn.svm.SVC.

In log anomaly detection, both training time and inference time are important. Due to the huge size of log data [60], it takes a long time for the model to retrain the model with the mountains of data. Besides, long inference time may lead to more revenue lost if the anomalies of the software systems have not been reported in time. [23]. Recent DL studies rarely provide an analysis of time efficiency with the most efficient classical approaches (e.g., DT, KNN, or SLFN).

To determine whether a proposed model is really worth using, we propose LightAD, an automated framework to compare the proposed anomaly detection model with naive baseline models. Fig. 6 shows the workflow of LightAD. We first prepare the set of naive baseline models and their corresponding hyper-parameter spaces. The hyper-parameter space for the proposed model should also be prepared. For each model, we optimize its hyper-parameters with respect to ModelGain which leverages three aspects of the model performance: (1) accuracy, (2) training time per log sequence and (3) inference time per log sequence. We formalize ModelGain as the following:

Table 8: Parameter Pool for simple methods

Models	Parameters	Default	Tuning Range	Description
KNN	n_neighbors	5	[1,10]	Number of the nearest neighbors used for classification.
	metric	"euclidean"	["euclidean", "manhattan", "cosine"]	Metric to use for distance computation.
DT	criterion	"gini"	["gini", "entropy", "log_loss"]	The function to measure the quality of a split.
	splitter	"best"	["best", "random"]	The strategy used to choose the split at each node.
	max_depth	None	None ∪ [5, 70]	Maximum depth of the tree.
	min_samples_split	2	[2, 5]	Minimum number of samples required to split an internal node.
	min_samples_leaf	1	[1, 5]	Minimum number of samples required to be at a leaf node.
SLFN	hidden neurons	100	[10, 200]	Number of neurons in the hidden layer.
	activation	"relu"	["identity", "relu", "logistic", "tanh"]	Activation function for the hidden layer.
	alpha	1e-4	[1e-6, 1e-2]	Strength of the L2 regularization term.
	tol	1e-4	[1e-6, 1e-2]	Tolerance for the optimization.
	max_iter	200	[20, 400]	Maximum number of iterations.

Table 9: Parameter Pool for deep methods

Models	Parameters	Default	Tuning Range	Description
CNN	hidden_size	128	[1, 256]	Number of neurons in the hidden layer.
	embedding_dim	32	[1, 64]	The embedding size for log events.
	epoches	5	[1, 20]	Maximum number of iterations.
	learning_rate	0.05	[1e-3, 1e-1]	The learning rate during training phase.
	batch_size	1,024	[32, 2048]	Number of samples that will be propagated.
LogRobust	hidden_size	128	[1, 256]	Number of neurons in the hidden layer.
	embedding_dim	32	[1, 128]	The embedding size for log events.
	epoches	5	[1, 20]	Maximum number of iterations.
	learning_rate	1e-2	[1e-3, 1e-1]	The learning rate during training phase.
	batch_size	1,024	[16, 2,048]	Number of samples that will be propagated.
	num_layers	2	[2, 12]	Number of layers in the fully connected neural network.
	num_directions	2	[1, 2]	Number of directions used in LSTM
NeuralLog	ff_dim	2,048	[1536, 2560]	Size of the feed-forward network.
	epoches	20	[5, 50]	Maximum number of iterations.
	batch_size	64	[1, 64]	Number of samples that will be propagated.
	init_lr	3e-4	[1e-4, 1e-1]	Initial learning rate.
	num_heads	12	[2, 6]	Number of attention heads.
	dropout	1e-1	[1e-2, 1e-1]	The dropout rate during training phase.
	val_ratio	0.1	[0.1,0.4]	The ratio of validation set in train set.

$$ModelGain = \lambda_1 * \frac{F - \tilde{F_0}}{F_0} + \lambda_2 * \frac{T_{train} - \tilde{T_0}}{T_0} + \lambda_3 * \frac{T_{infer} - \tilde{T_1}}{T_1} \quad (2)$$

where F stands for F_1-Score, T_{train} and T_{test} are the training and inference time per log sequence respectively. λ_1, λ_2 and λ_3 are three positive constants, indicating the relative importance of model accuracy, training efficiency, and inference efficiency. F_0, $\tilde{F_0}$, T_0, $\tilde{T_0}$, T_1 and $\tilde{T_1}$ are predetermined normalization constants to make the aggregation of metrics with different units reasonable. Note that F_0 should be positive and T_0, T_1 should be negative since higher F_1-Score and shorter training and inference time are preferred. For each model, its hyper-parameters are optimized with Bayesian Tuner [72]. Finally, we calculate the ModelGain for each model on the test dataset. The model with the highest ModelGain is the best model. If the best model happens to be the proposed model, then we conclude that the proposed model is valuable. Otherwise, the proposed model is inferior to the best simple model. The main steps of LightAD are described in Fig. 7.

Table 10 shows the output model under different priorities. The highest scores under different priorities are marked in bold. The first column indicates the objective we want to prioritize upon which, in the second column, we determine the weights for each objective. The normalization constants are set as: $F_0 = 0.2$, $\tilde{F_0} = 0.8$, $T_0 = -0.03$, $\tilde{T_0} = 0$, $T_1 = -0.002$ and $\tilde{T_1} = 0$. We only include CNN as "the proposed model" in that CNN has the best performance on the deduplicated HDFS dataset as in Table 7. As shown in the last column of Table 10, the optimal models under different priorities are all simple methods. According to LightAD, KNN is recommended for prioritizing training time, while DT is the preference for the other three priority settings: accuracy, inference time, and a balanced priority among all λ_i for $i \in \{1, 2, 3\}$.

RQ3 Answer: We propose LightAD, an automated framework to compare the proposed anomaly detection model with simple baseline models. On the deduplicated HDFS dataset, LightAD suggests simple methods are always superior to deep methods under different priorities.

Table 10: The optimal models output by LightAD with different priorities on deduplicated HDFS

Priority	$(\lambda_1, \lambda_2, \lambda_3)$	Model	ModelGain	Optimal Model
Accuracy	(1.0, 0.0, 0.0)	CNN	0.5904	DT
		KNN	0.7000	
		DT	**0.7030**	
		SLFN	0.6824	
Train	(0.5, 0.5, 0.0)	CNN	0.2593	KNN
		KNN	**0.3514**	
		DT	0.3499	
		SLFN	0.3377	
Inference	(0.5, 0.0, 0.5)	CNN	0.1896	DT
		KNN	0.3076	
		DT	**0.3462**	
		SLFN	0.3377	
Balanced	(0.3, 0.3, 0.3)	CNN	0.1065	DT
		KNN	0.1850	
		DT	**0.2121**	
		SLFN	0.1984	

1. Let S denote the set of simple models (e.g., DT, KNN), k denote of proposed model (e.g., CNN, LogRobust) and M denote $S \cup \{k\}$. For each $m \in M$, $Space_m$ denotes the corresponding hyper-parameter space in which these hyper-parameters need to be tuned. $|Space_m|$ denotes the dimension of $Space_m$.
2. LightAD initially runs each m for $N_m = 10 * |Space_m|$ times. Each run is denoted as $run_{i \in N_m}$ and the corresponding hyper-parameters are contained in set H_m.
3. Each run_i is scored with a certain optimization objective. In the case of our LightAD, the objective is to maximize ModelGain, leveraging three aspects of the model: accuracy, train time, and infer time. The set of scores is kept in $SCORES(m)$.
4. LightAD adopts Bayesian Tuner which builds a surrogate function $\mathcal{S}_m$ (e.g., Gaussian Process) based on $SCORES(m)$ and H_m. The goal of $\mathcal{S}_m$ is to approximate $\mathcal{F}_m = \mathcal{F}(Score|m, Space_m)$. $h_{new} = argmax\mathcal{S}_m$ and ModelGain(h_{new}) are added into H_m and $SCORES(m)$, respectively.
5. Step 3 is repeated until $max_{iter} = 10 * |Space_m|^2$ is reached.
6. For each $m \in M$, $BESTPARAMS(m) = argmaxSCORES(m)$.
7. Return the model with the highest score on the test dataset and its corresponding optimal hyper-parameters. If the proposed model k is exactly the best model, then it is considered as valuable. Otherwise, k is considered as redundant.

Figure 7: Procedure LightAD: strive to find the best model that maximizes ModelGain on training and validation data and check whether the proposed model is valuable

4.4 Threats to Validity

External Validity. With the advancement of cloud computing, a wide range of cloud software has emerged along with their corresponding logging systems. This paper shows that simple and fast methods can outperform expensive cost DL methods on five public log datasets. This is due to the lack of available log anomaly detection datasets. For instance, Le et al. [43] study four public log datasets in their study, and He et al. [33] include two datasets. Our study contains all the log datasets of these works and adds one more log dataset, i.e., Liberty [63]. However, all the existing public log datasets are old. HDFS was collected in 2009 [80] while the other four supercomputer datasets were generated between 2005 and 2007 [63]. In the future, we will collect log datasets on various software systems to validate the effectiveness of LightAD.

Internal Validity. There are two main threats to internal validity. The first threat is that we only focus on supervised log anomaly detection models (e.g., LogRobust [85]) whereas unsupervised models (e.g., DeepLog [21]) are not included. This could be a potential threat for us to generalize our findings to the entire log anomaly detection community. The second threat is the incomplete use of some log datasets. Due to the large data size of Thunderbird, Liberty, and Spirit, we only utilize specific segments of these datasets. Even though the segmentation of huge datasets is commonly used in previous works [42, 43], different selections of the log datasets could affect the results to some extent.

5 RELATED WORK

5.1 Empirical Studies of Classical Machine Learning Over Deep Learning

Despite the remarkable performance of DL in many areas, it can be computationally expensive and requires a GPU for parallel computation. There are some research works [12, 29, 40, 54, 57] that show that DL methods are slower and less accurate than classical methods. For example, Cao et al. [12] showed that LSTM-based DL is $1,000X$ slower despite having lesser accuracy in system anomaly detection. Menzies et al. [57] demonstrated that combining clustering algorithms with local classification algorithms achieved comparable performance to DL models while running hundreds of times faster. Inspired by these empirical studies, we investigated log anomaly detection and showed that with simple data preprocessing and parameter tuning, classical machine learning methods can outperform DL methods and take much less time to train the model.

5.2 Log-based Anomaly Detection

Log-based anomaly detection is the task of identifying the system's anomalous patterns that do not conform to the expected behaviors on normal log data. The log-based anomaly detection can be divided into supervised learning [9, 47, 52] and unsupervised learning [24, 80]. Additionally, it can be divided into classical ML-Based log anomaly detection methods [15, 80] and DL methods [24, 52, 62]. Recently, NeuralLog [42] can achieve the best performance on four public log dataset (HDFS [80], BGL, Thunderbird, Spirit [63]) without log parsing. This paper shows that we can achieve the highest performance on five public datasets (HDFS, BGL, Spirit, Thunderbird, Liberty [63]) by using simple methods without intricate log preprocessing.

5.3 Automated Log Analysis

Automated log analysis aims to enable effective and efficient usage of software-intensity systems, consisting of four aspects: (1) automate and assist log writing, (2) log compression, (3) log parsing, (4) log mining.

Logging. Event logging is an essential method for recording textual and numeric data of software systems, and its implementation often depends on an empirical process during the development phase [65]. He et al. [32] identified three main concerns of developers regarding event logging, namely diagnosability, maintenance, and performance. Several works have tackled the challenges of automated logging, including where to log [14, 35, 86], what to log [46, 46, 78], and how to log [13, 38, 50].

Log Compression. After collecting logs, they are typically stored on the hard disk for further diagnosis or auditing of sensitive operations. As the scale of distributed systems continues to grow, the storage of accumulated logs can become a burden on the storage system. He et al.[32] identified three categories of log compression techniques: bucket-based compression[6, 26], dictionary-based compression [18, 67], and statistics-based compression [8, 30]. Recently, some works have made progress in achieving both storage cost and time efficiency [25, 82].

Log Parsing. Log parsing is a critical component of automated log analysis, which converts semi-structured logs into structured data. Log parsing methods can be classified into two groups: offline log parsing approaches [28, 53, 75] and online log parsing approaches [17, 20, 31]. To address the problem of log message format identification, Messaoudi et al.[58] proposed the MoLFI approach, which formulates it as a multi-objective problem. Some works have focused on designing guidelines[37, 89] to evaluate current log parsing methods comprehensively.

Automated Log Mining. Automated log mining involves automatically exploring and analyzing large volumes of log data, with the goal of extracting useful information from the systems and predicting future trends to take early actions. To analyze logs automatically, textual logs must first be transformed into appropriate data formats. Several works have focused on extracting features from logs [4, 7]. There are various topics within log mining, including failure prediction [68], failure diagnosis [71, 87, 88], anomaly detection [80, 81, 85], log-based slicing [59], log visualization [55], and root cause analysis [44].

Our work focuses on anomaly detection, a typical and widely-explored log mining task.

5.4 Empirical Studies on Logs

Empirical studies are essential in log analysis, providing valuable insights into various aspects of academic and industrial practices. He et al. [32] conducted a survey covering the entire lifecycle of log analysis, including logging, log compression, log parsing, and automated log mining. Other empirical studies have focused on specific domains of log analysis, such as logging [45, 83, 84], log-based anomaly detection [16, 33], software monitoring [11], cloud log forensics [36], and cyber security applications [41].

Le et al. [43] conducted an empirical study on recent DL models for log-based anomaly detection and found that the performance of existing models was largely undermined by various factors, including training data selection strategies, different dataset characteristics, and early detection capability, indicating that the problem of log-based anomaly detection is still unsolved.

In contrast to previous studies, our research demonstrates that complex methods such as DL and intricate log preprocessing do not necessarily outperform simple methods like KNN, SVM, and DT. Additionally, our study includes five public log datasets, one more than the study conducted by Le et al [43].

6 CONCLUSION

In this paper, we demonstrate that simple models can outperform complex log vectorization methods and resource-intensive models, achieving superior results with reduced computation time. We then analyze why sophisticated DL methods fail to surpass these simpler techniques. To assist researchers in determining whether DL methods are necessary, we propose LightAD, an architecture that optimizes three objectives using Bayesian optimization while considering both performance metrics and time efficiency. This approach automatically fine-tunes hyper-parameters for different models, facilitating a fair comparison among various techniques.

Employing LightAD, we establish that simple methods can outperform DL techniques on five public log datasets in terms of both performance metrics and time efficiency. Based on our findings, we recommend that researchers explore simpler approaches for software engineering tasks before delving into DL methods. At a minimum, these simpler methods can serve as a baseline against more complex techniques.

Moving forward, we plan to collect additional log datasets from widely-used open-source software (e.g., Apache Kafka) and evaluate LightAD on new log benchmarks. Furthermore, we aim to investigate how machine learning and log vectorization techniques can be effectively applied in the log analysis domain.

7 DATA AVAILABILITY

The codes and data of this paper can be found at [3].

ACKNOWLEDGMENTS

We thank the anonymous ICSE reviewers for their valuable feedback on the earlier draft of this paper. This paper was supported by the National Natural Science Foundation of China (No. 62102340), and Shenzhen Science and Technology Program.

REFERENCES

[1] 2022. A deep learning-based log analysis toolkit for automated anomaly detection. Retrieved April 30, 2022 from https://github.com/logpai/deep-loglizer
[2] 2022. A machine learning-based log analysis toolkit for automated anomaly detection. Retrieved April 30, 2022 from https://github.com/logpai/loglizer
[3] 2023. A toolkit for light automated log anomaly detection. https://github.com/BoxiYu/LightAD
[4] Hen Amar, Lingfeng Bao, Nimrod Busany, David Lo, and Shahar Maoz. 2018. Using finite-state models for log differencing. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 49–59.
[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).
[6] Raju Balakrishnan and Ramendra K Sahoo. 2006. Lossless compression for large scale cluster logs. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 7–pp.
[7] Lingfeng Bao, Nimrod Busany, David Lo, and Shahar Maoz. 2019. Statistical log differencing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 851–862.
[8] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. 2011. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 267–277.

[9] Peter Bodik, Moises Goldszmidt, Armando Fox, Dawn B Woodard, and Hans Andersen. 2010. Fingerprinting the datacenter: automated classification of performance crises. In *Proceedings of the 5th European conference on Computer systems*. 111–124.

[10] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. 2000. LOF: identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 93–104.

[11] Jeanderson Candido, Maurício Aniche, and Arie van Deursen. 2019. Contemporary software monitoring: A systematic literature review. *arXiv e-prints* (2019), arXiv–1912.

[12] Qing Cao and Haoran Niu. 2022. Higher-order Markov Graph based Bug Detection in Cloud-based Deployments. In *2022 IEEE International Performance, Computing, and Communications Conference (IPCCC)*. IEEE, 153–160.

[13] Boyuan Chen et al. 2017. Characterizing logging practices in java-based open source software projects–a replication study in apache software foundation. *Empirical Software Engineering* 22, 1 (2017), 330–374.

[14] Boyuan Chen and Zhen Ming Jiang. 2017. Characterizing and detecting anti-patterns in the logging code. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 71–81.

[15] Mike Chen, Alice X Zheng, Jim Lloyd, Michael I Jordan, and Eric Brewer. 2004. Failure diagnosis using decision trees. In *International Conference on Autonomic Computing, 2004. Proceedings.* IEEE, 36–43.

[16] Zhuangbin Chen, Jinyang Liu, Wenwei Gu, Yuxin Su, and Michael R Lyu. 2021. Experience Report: Deep Learning-based System Log Analysis for Anomaly Detection. *arXiv preprint arXiv:2107.05908* (2021).

[17] Hetong Dai, Heng Li, Che Shao Chen, Weiyi Shang, and Tse-Hsun Chen. 2020. Logram: Efficient log parsing using n-gram dictionaries. *IEEE Transactions on Software Engineering* (2020).

[18] Sebastian Deorowicz and Szymon Grabowski. 2008. Sub-atomic field processing for improved web log compression. In *2008 International Conference on" Modern Problems of Radio Engineering, Telecommunications and Computer Science"(TCSET)*. IEEE, 551–556.

[19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[20] Min Du and Feifei Li. 2016. Spell: Streaming parsing of system event logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, 859–864.

[21] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 1285–1298.

[22] Facebook. 2019. Downtime, outages and failures - understanding their true costs. http://www.evolven.com/blog/downtime-outages-and-failures-understanding-their-true-costs.html

[23] Facebook. 2019. Facebook loses $24,420 a minute during outages. https://www.theatlantic.com/technology/archive/2014/10/facebook-is-losing-24420-per-minute/382054/

[24] Amir Farzad and T Aaron Gulliver. 2020. Unsupervised log message anomaly detection. *ICT Express* 6, 3 (2020), 229–237.

[25] Peng Fei, Zhou Li, Zhiying Wang, Xiao Yu, Ding Li, and Kangkook Jee. 2021. {SEAL}: Storage-efficient causality analysis on enterprise logs with query-friendly compression. In *30th USENIX Security Symposium (USENIX Security 21)*. 2987–3004.

[26] Bo Feng, Chentao Wu, and Jie Li. 2016. MLC: an efficient multi-level log compression method for cloud backup systems. In *2016 IEEE Trustcom/BigDataSE/ISPA*. IEEE, 1358–1365.

[27] Evelyn Fix and Joseph Lawson Hodges. 1989. Discriminatory analysis. Nonparametric discrimination: Consistency properties. *International Statistical Review/Revue Internationale de Statistique* 57, 3 (1989), 238–247.

[28] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution anomaly detection in distributed systems through unstructured log analysis. In *2009 ninth IEEE international conference on data mining*. IEEE, 149–158.

[29] Wei Fu and Tim Menzies. 2017. Easy over hard: A case study on deep learning. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 49–60.

[30] Kimmo Hätönen, Jean François Boulicaut, Mika Klemettinen, Markus Miettinen, and Cyrille Masson. 2003. Comprehensive log compression with frequent patterns. In *International Conference on Data Warehousing and Knowledge Discovery*. Springer, 360–370.

[31] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. 2017. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE international conference on web services (ICWS)*. IEEE, 33–40.

[32] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R Lyu. 2021. A survey on automated log analysis for reliability engineering. *ACM Computing Surveys (CSUR)* 54, 6 (2021), 1–37.

[33] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. 2016. Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th international symposium on software reliability engineering (ISSRE)*. IEEE, 207–218.

[34] Guang-Bin Huang, Yan-Qiu Chen, and Haroon A Babri. 2000. Classification ability of single hidden layer feedforward neural networks. *IEEE transactions on neural networks* 11, 3 (2000), 799–801.

[35] Zhen Ming Jiang, Ahmed E Hassan, Gilbert Hamann, and Parminder Flora. 2008. Automatic identification of load testing problems. In *2008 IEEE International Conference on Software Maintenance*. IEEE, 307–316.

[36] Suleman Khan, Abdullah Gani, Ainuddin Wahid Abdul Wahab, Mustapha Aminu Bagiwa, Muhammad Shiraz, Samee U Khan, Rajkumar Buyya, and Albert Y Zomaya. 2016. Cloud log forensics: foundations, state of the art, and future directions. *ACM Computing Surveys (CSUR)* 49, 1 (2016), 1–42.

[37] Zanis Ali Khan, Donghwan Shin, Domenico Bianculli, and Lionel Briand. 2022. Guidelines for Assessing the Accuracy of Log Message Template Identification Techniques. In *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*. ACM.

[38] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *European conference on object-oriented programming*. Springer, 220–242.

[39] Max Landauer, Sebastian Onder, Florian Skopik, and Markus Wurzenberger. 2022. Deep Learning for Anomaly Detection in Log Data: A Survey. *arXiv preprint arXiv:2207.03820* (2022).

[40] Max Landauer, Sebastian Onder, Florian Skopik, and Markus Wurzenberger. 2023. Deep learning for anomaly detection in log data: A survey. *Machine Learning with Applications* 12 (2023), 100470.

[41] Max Landauer, Florian Skopik, Markus Wurzenberger, and Andreas Rauber. 2020. System log clustering approaches for cyber security applications: A survey. *Computers & Security* 92 (2020), 101739.

[42] Van-Hoang Le and Hongyu Zhang. 2021. Log-based anomaly detection without log parsing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 492–504.

[43] Van Hoang Le and Hongyu Zhang. 2022. Log-based Anomaly Detection with Deep Learning: How Far Are We? *arXiv preprint arXiv:2202.04301* (2022).

[44] Zeyan Li, Junjie Chen, Rui Jiao, Nengwen Zhao, Zhijun Wang, Shuwei Zhang, Yanjun Wu, Long Jiang, Leiqin Yan, Zikai Wang, et al. 2021. Practical root cause localization for microservice systems via trace analysis. In *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*. IEEE, 1–10.

[45] Zhenhao Li, Tse-Hsun Peter Chen, Jinqiu Yang, and Weiyi Shang. 2021. Studying duplicate logging statements and their relationships with code clones. *IEEE Transactions on Software Engineering* (2021).

[46] Zhenhao Li, Heng Li, Tse-Hsun Peter Chen, and Weiyi Shang. 2021. Deeplv: Suggesting log levels using ordinal based neural networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1461–1472.

[47] Yinglung Liang, Yanyong Zhang, Hui Xiong, and Ramendra Sahoo. 2007. Failure prediction in ibm bluegene/l event logs. In *Seventh IEEE International Conference on Data Mining (ICDM 2007)*. IEEE, 583–588.

[48] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. 2016. Log clustering based problem identification for online service systems. In *Proceedings of the 38th International Conference on Software Engineering Companion*. 102–111.

[49] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. 2008. Isolation forest. In *2008 eighth ieee international conference on data mining*. IEEE, 413–422.

[50] Joshua Lockerman, Jose M Faleiro, Juno Kim, Soham Sankaran, Daniel J Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. 2018. The {FuzzyLog}: A Partially Ordered Shared Log. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 357–372.

[51] Jian-Guang Lou, Qiang Fu, Shenqi Yang, Ye Xu, and Jiang Li. 2010. Mining invariants from console logs for system problem detection. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*.

[52] Siyang Lu, Xiang Wei, Yandong Li, and Liqiang Wang. 2018. Detecting anomaly in big data system logs using convolutional neural network. In *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*. IEEE, 151–158.

[53] Adetokunbo AO Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. 2009. Clustering event logs using iterative partitioning. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1255–1264.

[54] Mika Mäntylä, Martín Varela, and Shayan Hashemi. 2022. Pinpointing anomaly events in logs from stability testing–n-grams vs. deep-learning. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 285–292.

[55] Shahar Maoz and David Harel. 2011. On tracing reactive systems. *Software & Systems Modeling* 10, 4 (2011), 447–468.

[56] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, et al. 2019. LogAnomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs.. In *IJCAI*, Vol. 19. 4739–4745.

[57] Tim Menzies, Suvodeep Majumder, Nikhila Balaji, Katie Brey, and Wei Fu. 2018. 500+ times faster than deep learning:(a case study exploring faster methods for text mining stackoverflow). In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 554–563.

[58] Salma Messaoudi, Annibale Panichella, Domenico Bianculli, Lionel Briand, and Raimondas Sasnauskas. 2018. A search-based approach for accurate identification of log message formats. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 167–16710.

[59] Salma Messaoudi, Donghwan Shin, Annibale Panichella, Domenico Bianculli, and Lionel C Briand. 2021. Log-based slicing for system-level test cases. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 517–528.

[60] Haibo Mi, Huaimin Wang, Yangfan Zhou, Michael Rung-Tsong Lyu, and Hua Cai. 2013. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems* 24, 6 (2013), 1245–1255.

[61] Meiyappan Nagappan, Kesheng Wu, and Mladen A Vouk. 2009. Efficiently extracting operational profiles from execution logs using suffix arrays. In *2009 20th International Symposium on Software Reliability Engineering*. IEEE, 41–50.

[62] Sasho Nedelkoski, Jasmin Bogatinovski, Alexander Acker, Jorge Cardoso, and Odej Kao. 2020. Self-attentive classification-based anomaly detection in unstructured logs. In *2020 IEEE International Conference on Data Mining (ICDM)*. IEEE, 1196–1201.

[63] Adam Oliner and Jon Stearley. 2007. What supercomputers say: A study of five system logs. In *37th annual IEEE/IFIP international conference on dependable systems and networks (DSN'07)*. IEEE, 575–584.

[64] Kishore Papineni. 2001. Why inverse document frequency?. In *Second Meeting of the North American Chapter of the Association for Computational Linguistics*.

[65] Antonio Pecchia, Marcello Cinque, Gabriella Carrozza, and Domenico Cotroneo. 2015. Industry practices and event logging: Assessment of a critical software development process. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 169–178.

[66] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of machine learning research* 12, Oct (2011), 2825–2830.

[67] Balázs Rácz and András Lukács. 2004. High density compression of log files. In *Data Compression Conference, 2004. Proceedings. DCC 2004.* IEEE, 557.

[68] Barbara Russo, Giancarlo Succi, and Witold Pedrycz. 2015. Mining system logs to learn error predictors: a case study of a telemetry system. *Empirical Software Engineering* 20, 4 (2015), 879–927.

[69] Gerard Salton and Christopher Buckley. 1988. Term-weighting approaches in automatic text retrieval. *Information processing & management* 24, 5 (1988), 513–523.

[70] Mike Schuster and Kaisuke Nakajima. 2012. Japanese and korean voice search. In *2012 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE, 5149–5152.

[71] Weiyi Shang, Zhen Ming Jiang, Hadi Hemmati, Brain Adams, Ahmed E Hassan, and Patrick Martin. 2013. Assisting developers of big data analytics applications when deploying on hadoop clouds. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 402–411.

[72] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems* 25 (2012).

[73] StatusCake Team. 2020. The Most Expensive Website Downtime Periods in History. https://www.statuscake.com/the-most-expensive-website-downtime-periods-in-history/

[74] UpGuard. 2019. The cost of downtime at the world's biggest online retailer. Retrieved September 1, 2020 from https://www.upguard.com/blog/the-cost-of-downtime-at-the-worlds-biggest-online-retailer

[75] Risto Vaarandi. 2003. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003)(IEEE Cat. No. 03EX764)*. Ieee, 119–126.

[76] Risto Vaarandi and Mauno Pihelgas. 2015. Logcluster-a data clustering and pattern mining algorithm for event logs. In *2015 11th International conference on network and service management (CNSM)*. IEEE, 1–7.

[77] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[78] Zehao Wang, Haoxiang Zhang, Tse-Hsun Chen, and Shaowei Wang. 2021. Would you like a quick peek? providing logging support to monitor data processing in big data applications. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 516–526.

[79] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[80] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 117–132.

[81] Lin Yang, Junjie Chen, Zan Wang, Weijing Wang, Jiajun Jiang, Xuyuan Dong, and Wenbin Zhang. 2021. Semi-supervised log-based anomaly detection via probabilistic label estimation. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1448–1460.

[82] Kundi Yao, Mohammed Sayagh, Weiyi Shang, and Ahmed E Hassan. 2021. Improving State-of-the-art Compression Techniques for Log Management Tools. *IEEE Transactions on Software Engineering* (2021).

[83] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be conservative: Enhancing failure diagnosis with proactive logging. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 293–306.

[84] Haonan Zhang, Yiming Tang, Maxime Lamothe, Heng Li, and Weiyi Shang. 2022. Studying logging practice in test code. *Empirical Software Engineering* 27, 4 (2022), 1–45.

[85] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. 2019. Robust log-based anomaly detection on unstable log data. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 807–817.

[86] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 565–581.

[87] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2018. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering* 47, 2 (2018), 243–260.

[88] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 683–694.

[89] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R Lyu. 2019. Tools and benchmarks for automated log parsing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 121–130.