Information Processing and Management 58 (2021) 102540

Contents lists available at ScienceDirect

Information Processing and Management

journal homepage: www.elsevier.com/locate/ipm

QLLog: A log anomaly detection method based on Q-learning algorithm

Xiaoyu Duan [a], Shi Ying [a,*], Wanli Yuan [a], Hailong Cheng [a], Xiang Yin [b]

[a] School of Computer Science, Wuhan University, Wuhan 430072, China
[b] Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China

ARTICLE INFO

Keywords:
Log anomaly detection
Q-learning
Reinforcement learning
Data analysis

QLLog: detect anomalies and rank severity

ABSTRACT

Most of the existing log anomaly detection methods suffer from scalability and numerous false positives. Besides, they cannot rank the severity level of abnormal events. This paper proposes a log anomaly detection based on Q-learning, namely QLLog, which can detect multiple types of system anomalies and rank the severity level of abnormal events. We first build a mathematical model of log anomaly detection, proving that log anomaly detection is a sequential decision problem. Second, we use the Q-learning algorithm to build the core of the anomaly detection model. This allows QLLog to automatically learn directed acyclic graph log patterns from normal execution and adjust the training model according to the reward value. Then, QLLog combines the advantages of the Q-learning algorithm and the specially designed rules to detect anomalies when log patterns deviate from the model trained from log data under normal execution. Besides, we provide a feedback mechanism and build an abnormal level table. Therefore, QLLog can adapt to new log states and log patterns. Experiments on real datasets show that the method can quickly and effectively detect system anomalies. Compared with the state of the art, QLLog can detect numerous real problems with high accuracy 95%, and its scalability outperforms other existing log-based anomaly detection methods.

1. Introduction

System logs record system states and application behaviors. Operation & maintenance personnel (i.e., ops) usually analyze system logs to locate the fault (Astekin, Zengin, & Sözer, 2019; Zhang et al., 2018). Besides, such log data is universally available in nearly all computer systems and is a valuable resource for understanding system status. Therefore, system logs are an essential data source for performance monitoring and anomaly detection.

Traditional anomaly detection methods are usually based on standard mining methodologies (Du, Li, Zheng, & Srikumar, 2017). There are many defects in traditional anomaly detection methods. For example, (1) they need domain knowledge, (2) they are difficult to identify the unknown faults or only identify a single type of abnormal log. Besides, (3) they cannot incrementally update the anomaly detection model in an online fashion. Therefore, traditional anomaly detection methods are no longer workable. With the popularization of machine learning and deep learning technology, scholars begin to study the application of deep learning and machine learning methods in log anomaly detection.

Existing anomaly detection methods are usually based on LSTM (Ding, Ma, Gao, Ma, & Tan, 2019; Du et al., 2017; Luo, Liu, & Gao, 2017), SVM (Scholkopf & Smola, 2018; Turkoz, Kim, Son, Jeong, & Elsayed, 2020; Zhang & Sivasubramaniam, 2008), clustering (Hasan, Orgun, & Schwitter, 2019; Lin, Zhang, Lou, Zhang, & Chen, 2016; Yuan, Li, Yao, & Zhang, 2017), PCA (Xu,

* Corresponding author.
E-mail address: yingshi@whu.edu.cn (S. Ying).

https://doi.org/10.1016/j.ipm.2021.102540
Received 23 March 2020; Received in revised form 16 December 2020; Accepted 29 January 2021
Available online 9 February 2021

Huang, Fox, Patterson, & Jordan, 2010), Isolated forest (Liu, Ting, & Zhou, 2009), and data mining techniques (Lou, Fu, Yang, Xu, & Li, 2010; Ye, Li, Adjeroh, & Iyengar, 2017). Although these methods are successful in particular scenarios, it still has many disadvantages. For example, both clustering and Isolated forest methods produce many false positives. PCA is not good at detecting the too regular normal pattern in the dataset. Mining techniques can only detect execution sequence anomalies. LSTM model is difficult to update. Overall, existing anomaly detection methods suffer from scalability, a large number of false positives, and only focus on a single type of log anomaly. Besides, most of the existing approaches do not rank the level of log abnormal. If the anomaly detection method can rank the level of log abnormal, it will improve the detection efficiency.

The research goal and the practical significance of this paper are to find an efficient log anomaly detection method, maintain stable operation of the system. (1) It can perform anomaly detection for multiple types of anomalies to reduce the false positives, not limited to a single type of fault; (2) It can incrementally update the anomaly detection model to adapt to the new log states and log patterns; (3) It can apply to several real datasets, not affected by domain knowledge. Therefore, we propose an anomaly detection approach, namely QLLog (log anomaly detection based on Q-Learning). This research differs from existing works. For example, QLLog provides the probability of system anomalies and reports high-level anomalies to ops to improve their efficiency in handling anomalies. QLLog can detect more log anomalies than existing methods because it can detect both the point anomalies and the execution sequence anomalies, but most of the existing methods only detect the point anomalies or the execution sequence anomalies. Besides, according to the feedback report, QLLog can automatically update the anomaly detection model and not retrain it.

QLLog uses the Q-learning algorithm to build the core part of the anomaly detection model. Q-Learning algorithm is a typical reinforcement learning method based on value function. We first build a mathematical model for log anomaly detection. Second, we prove that the log anomaly detection problem is a typical sequential decision problem and is remarkably consistent with the Markov decision process (MDP). We use Q-Learning to solve the Q value of each log execution path and use the Q value to determine whether or not the log execution sequence is abnormal. Besides, we build an abnormal level table and calculate the log point abnormal probability by using the longest common subsequence. We evaluated QLLog on different real log datasets with over 1000 thousand log entries. The results show that QLLog has high accuracy and efficiency.

The primary contributions of this paper are as follows:

- We propose an unsupervised anomaly detection method for unbalanced log datasets with high scalability and high precision, namely QLLog.
- QLLog can detect multiple types of log anomalies, including log point anomalies and log execution anomalies.
- QLLog provides a feedback mechanism to update the detection model.
- QLLog provides a dynamically updated abnormal level table to reduce false positives.

The rest of this paper is organized as follows: Section 2 presents related work on log anomaly detection. Section 3 introduces the basic structure and the pre-processing of HDFS log data. Section 4 presents the problem description of log anomaly detection. Section 5 presents the QLLog and introduces the anomaly detection process, respectively. We evaluated the performance of QLLog in Section 6. Finally, Section 7 presents the final remarks.

2. Related work

Anomaly detection has long played an important role in various fields such as internet security (Kwon et al., 2019; Wu et al., 2012), software security (Haddadpajouh, Dehghantanha, Khayami, & Choo, 2018; Jahromi et al., 2020), web security (Javed, Burnap, & Rana, 2019), social network (Hasan et al., 2019; Kaur & Singh, 2017), and big data (Mudassar, Ko, & Mukhopadhyay, 2018; Yu et al., 2016). Anomalies can be caused by errors in the data but sometimes indicate a new, previously unknown, underlying process. As early as 1987, Enderlein and Hawkins (1987) have defined the anomalies. They pointed out that the anomaly data are the data that deviate from other observation data seriously (i.e., abnormal instances are markedly different from normal instances). Therefore, anomaly detection is also called outlier detection.

Lou et al. (2010) pointed out that there is a constant linear relationship in the system. They proposed an unstructured log analysis technique (IM) for anomaly detection, with a novel algorithm to automatically discover program invariants in logs. This approach first mines small invariants that could be satisfied by most vectors and then treats those vectors that do not satisfy these invariants as abnormal execution sessions.

Ren, Fu, Zhan, and Zhou (2012) proposed Logmaster. Logmaster first builds a general-purpose event correlation mining system. Second, Logmaster uses the Apriori-LIS algorithm to mine event correlations in a single node and multiple nodes. Third, they design an innovative abstraction, Failure Correlation Graphs (FCG), to represent correlations of failure events. Finally, Logmaster detects anomalies by using ECG.

Lin et al. (2016) proposed LogCluster, an approach that clusters the logs to ease log-based problem identification. LogCluster facilitates problem identification by clustering similar log sequences and retrieving recurrent issues. The operation of LogCluster can be divided into two stages: the construction stage and the production stage. LogCluster first clusters log sequence to construct an initial knowledge base. Second, LogCluster analyzes the log sequences and checks if the clusters can be found in the knowledge base. In the end, examines a small number of representative log sequences from the clusters that are previously unseen. LogCluster improves the effectiveness, but the strategy is too coarse-grained.

Du et al. (2017) proposed DeepLog, a deep neural network model utilizing Long Short-Term Memory (LSTM), to model a system log as a natural language sequence. DeepLog first uses the execution sequence of normal logs in the training stage to train a log key

Table 1
Log parsing results.

ID	Log template
1	WARN dfs FSNamesystem BLOCK NameSystem addStoredBlock Redundant addStoredBlock request received for on size
2	INFO dfs FSNamesystem BLOCK NameSystem addStoredBlock blockMap updated is added to size
3	INFO dfs FSNamesystem BLOCK NameSystem.allocateBlock
4	INFO dfs DataNode PacketResponder PacketResponder for block terminating
5	INFO dfs DataNode PacketResponder Received block of size from

anomaly detection model and constructs system execution workflow models for diagnosis purposes. In the detection stage, DeepLog compares the predicted results of the log key anomaly detection model with the actual log results to determine whether the system is abnormal. Papers that models log sequences as natural language sequences also include (Das, Mueller, Siegel, & Vishnu, 2018; Marchi et al., 2015; Meng et al., 2019). These methods have good detection results, but the disadvantage is that it is difficult to update the model.

Xu et al. (2010) proposed an automatic methodology for mining console logs using a combination of program analysis. This method first combines log parsing and text mining with source code analysis to extract structure from the console logs. Second, it extracts features from the structured information to detect anomalous patterns in the logs using Principal Component Analysis (PCA). Finally, they use a decision tree to distill the results of PCA-based anomaly detection to a format readily understandable. The disadvantage of this method is that source code is not always accessible in practice, and the detection efficiency and accuracy are greatly affected by dimensions.

Watanabe, Otsuka, Sonoda, Kikuchi, and Matsumoto (2012) developed a method for predicting upcoming failures to handle them before failures occurrence. This method first classifies log entries from components based on similarities between log entries. It classifies the logs automatically, even if the formats of observed logs are frequently changed. Next, it identifies a log pattern (a set of log types), which is highly related to a failure occurrence by Bayesian inference. Besides, it also predicts the occurrence of failures by detecting the log pattern recorded in the log pattern dictionary.

3. Log parser

Log datasets are composed of log entries, and each log entry includes three parts (Zhu et al., 2019): timestamp, log type, and log event. Timestamp records when the system generated the log entry, log type records the rough characteristics of the log entry (e.g., “INFO” or “WARN”), log event is the core part of the log entry, records the behavior of system and application. Each word in log entries can be categorized into either log parameter word or log keyword. Log keywords are used to describe the semantic information of system events. Log parameter words usually appear with the form, such as file path, IP address, Mac address, and hostname. Log anomaly detection is to determine whether the log event is abnormal or not. Therefore, even if the log type is “INFO”, it still might be an abnormal log entry. Note that we do not have to define or detect any anomalies when parsing logs, and the goal of the log parser is only parsing each unstructured log entry into a structured log entry. Anomalies are defined and detected by the detection model. The raw log data and classification log data are shown in the top of Fig. 1:

The log entries in Fig. 1 include two log types, three log subtypes, and five log events. As seen from multiple examples of log entry shown in Fig. 1, there are several parameter words, such as symbol, block_id, file path. Log anomaly detection methods typically apply a two-step procedure (Du & Li, 2016; He, Zhu, Zheng, & Lyu, 2017; Zhang et al., 2018). First, a log parser is used to parse log entries to the structured log. Second, anomaly detection is performed. Therefore, in our work, we first parse unstructured, free-text log entries into a structured representation, i.e., extract log templates from log entries and then match the log entries to the templates. For example, file path and timestamp in the third raw log entry are the parameter words that vary from one log entry to another. Therefore, they are not a part of a log template, whereas the rest, sketches out the event and hence is a template that summarizes this and other similar log entries. The log templates of each raw log entry are shown in Table 1.

As illustrated in Table 1, we set a unique ID to different log templates, so one log template (log entry) is a counterpart of one template ID. For example, the template ID of the first line in Table 1 is 1.

In HDFS dataset, we set the **block_id** as identifiers for a particular execution sequence. Each dataset has unique identifiers (e.g. the identifier of OpenStack log dataset is **instance_id**). These identifiers can group log entries together or untangle log entries produced by concurrent processes to separate, single-thread sequential sequences. For example, the second, fourth and fifth log entries in Fig. 1 can be divided into the same group (log execution sequence) because they have the same **block_id** (*blk*_8356640976087556977). Log execution sequence dataset are shown in Fig. 2.

Each line in Fig. 2 is a log execution sequence, and each number (ID) in the execution sequence is a log entry. Each log entry in log execution sequence has the same **block_id**. Fig. 2 contains eight different log execution sequences.

4. Problem description

For the log execution sequence dataset, each ID is not only a log entry but also a log state. We divide states into two categories: abnormal state and normal state. The normal state is the log entry that records the normal log event. Therefore, the abnormal state is the log entry that records the abnormal event. Besides, there is a special case of the abnormal state. The log event is normal, but it is abnormal if it is in an error position of the log execution sequence.

No.	Raw log data
1	081109 204917 35 WARN dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: Redundant addStoredBlock request received for blk_3888635850409849568 on 10.251.107.227:50010 size 67108864
2	081110 103340 34 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.251.71.16:50010 is added to blk_-8356640976087556977 size 67108864
3	081110 103340 34 INFO dfs.FSNamesystem: BLOCK* NameSystem.allocateBlock: /user/root/rand/_temporary/_task_200811101024_0001_m_000153_0/part-00153. blk_-4642050192821612252
4	081110 103340 7159 INFO dfs.DataNode$PacketResponder: PacketResponder 1 for block blk_-8356640976087556977 terminating
5	081110 103340 7159 INFO dfs.DataNode$PacketResponder: Received block blk_-8356640976087556977 of size 67108864 from /10.251.71.16

Log classification

Log time stamp	081109 204917 35, 081110 103340 34, 081110 103340 7159
Log type	INFO, WARN
Log subtypes	dfs.FSNamesystem, dfs.DataNode$PacketResponder
Log event	BLOCK* NameSystem.addStoredBlock: Redundant addStoredBlock request received for **blk_3888635850409849568** on 10.251.107.227:50010 size 67108864
	BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.251.71.16:50010 is added to **blk_-8356640976087556977** size 67108864
	BLOCK* NameSystem.allocateBlock: /user/root/rand/_temporary/_task_200811101024_0001_m_000153_0/part-00153. **blk_-4642050192821612252**
	PacketResponder 1 for block **blk_-8356640976087556977** terminating
	Received block **blk_-8356640976087556977** of size 67108864 from /10.251.71.16

Fig. 1. Log structure.

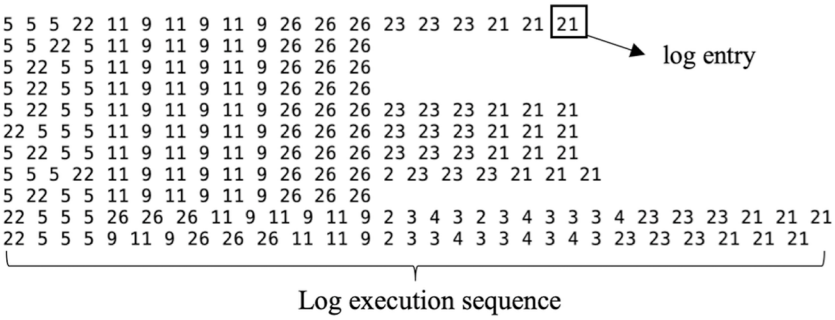


Fig. 2. Log execution sequence set.

We classify states according to the position in the execution sequence. We define the first state in the log execution sequence as the initial state, the state at the end of the log sequence is the final state, and the state between the initial state and the final state is an intermediate state. For example, in the first sequence of Fig. 2, the initial state is 5, and the final state is 21. Initial state 5 and final state 21 appear three times in a row. We define the initial state 5 and final state 21 have three self-cycles. We use the directed graph to show the process of the state transition. The directed graph is shown in Fig. 3.

As seen in Fig. 3, the states with one circle are initial state (i.e., 5, 22) or intermediate state (i.e., 9, 11, 2, 3, 4), the states with double circle are final state (i.e., 26, 21).

Consider the following log execution sequence $L = (S, E)$, S is the set of log states (S has $n \in \mathbb{N}$ log states), we set $S = \{s_1, s_2, \ldots, s_n\}$, and $\forall\ s_i \in S$ is one log state (log entry). We divide the log state set into a normal state set and an abnormal state set, therefore, S should be divided into two parts, $S_n = \{x | \forall x \in S\}$ is the normal state set, $S_a = \{x | \forall x \in S\}$ is the abnormal state set, obviously $S = S_n \cup S_a$.

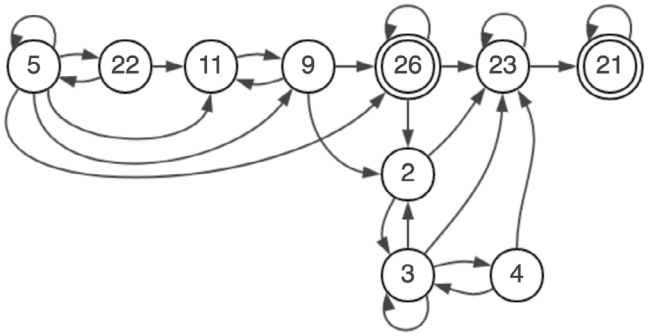


Fig. 3. Log state digraph.

$E = S \times S$ is the set of all log execution paths. Each path $e_{xy} \in E$, $\forall x, y \in S$, state x to state y is a one-way connection. We use $\xi_{xy} = \{0, 1\}$ to represent the state of each path, if $\xi_{xy} = 1$, the path e_{xy} is a normal execution path; otherwise, $\xi_{xy} = 0$, the path e_{xy} is an abnormal execution path. Therefore, we can divide E into two parts according to the above conditions: $E_n = \left\{e_{xy} | e_{xy} \in E, \xi_{xy} = 1, \forall x, y \in S\right\}$ is the set of all normal execution paths. $E_a = \left\{e_{xy} | e_{xy} \in E, \xi_{xy} = 0, \forall x, y \in S\right\}$ is the set of all abnormal execution paths, obviously, $E = E_n \cup E_a$.

We can use $H = \left\langle e_{x_1,y_1}, \ldots, e_{x_n,y_n} \right\rangle (n \in \mathbb{N})$ to represent the log execution sequence. For example, given a log sequence $L = [5, 22, 5, 26, 26]$, it may be expressed as $H = \left\langle e_{5,22}, e_{22,5}, e_{5,26}, e_{26,26} \right\rangle$. Therefore, the log execution sequence set is $ExP = \left\{H_1, H_2, \ldots, H_m\right\} (m \in \mathbb{N})$, and should satisfy the Formula (1).

$$|ExP| = \sum_{n=0}^{|E_n|} P\left(E_n, n\right) \tag{1}$$

where $P\left(E_n, n\right)$ is the number of permutations, which are composed of n execution paths selected from the execution path set E.

Assuming a path between any state x and any state y in the execution sequence, we can use $R_{xy} \in Z$ to represent the reward from state x to state y. Let $R_{xy} = \{0, 1, 100\}$ be the reward. If y is a final state, the reward of the path E_{xy} is 100; if y is not a final state, the reward is 1; if the path E_{xy} is an abnormal path, the reward is 0.

Markov decision processes (MDPs) (Huang et al., 2014; Tong, Chen, Deng, Li, & Li, 2019) are widely used to provide a mathematical framework for modeling decision making to solve stochastic sequential decision problems, in situations where an outcome is partly random and partly under the control of the decision-maker. Based on the above description, log anomaly detection is a typical sequential decision. The goal of finitely satisfying the Markov property in log anomaly detection is to form a state transition matrix that consists of each possibility of transitioning to the next state from a given state. This process is entirely consistent with the Markov decision process.

Reinforcement learning model (e.g., Q-learning) consists of an agent, a state, and a set of actions for each state. Q-learning (Christopher & Watkins, 1992; Clifton & Laber, 2020) stores all the information in a Q table, which represents the Q-value between two states. Using the policy declared in MDPs and performing an action, the agent can move from one state to another. Performing an action in a given state provides a reward for the agent. Therefore, in the log anomaly detection scenario, we can take the QLLog as an agent. Each log entry in the log execution sequence is the state S. The action space is the set E, each execution path e_{xy} in the set E is an action, and the reward is R_{xy}. In our case, limited by training data, S is a finite set of states and E is a finite set of actions. In fact, we can describe the log execution sequence as several different strategies. In the detection stage, we determine whether the current sequence is met the strategies getting from the training stage or not. If it is met, it is a normal log sequence. Otherwise, it is an abnormal log sequence.

5. QLLog architecture and overview

QLLog is an anomaly detection method that can detect two types of anomalies and provide the severity level of anomalies. We can divide the structure of QLLog into two stages, three modules, and the core module is the anomaly detection model. Fig. 4 shows the structure of QLLog.

Training stage. In this stage, the training data for QLLog are log entries from the normal system execution path. We first parse log entries to log sequence (see Section 3 for more details). Second, we use the Q-learning reinforcement learning algorithm to build the Q table. Finally, we create an abnormal level table in the anomaly detection model. The abnormal level table is empty in the training stage, but it is dynamically updated in the detection stage. Furthermore, it is necessary for QLLog to update the anomaly detection model incrementally because the training data may not cover all possible normal states.

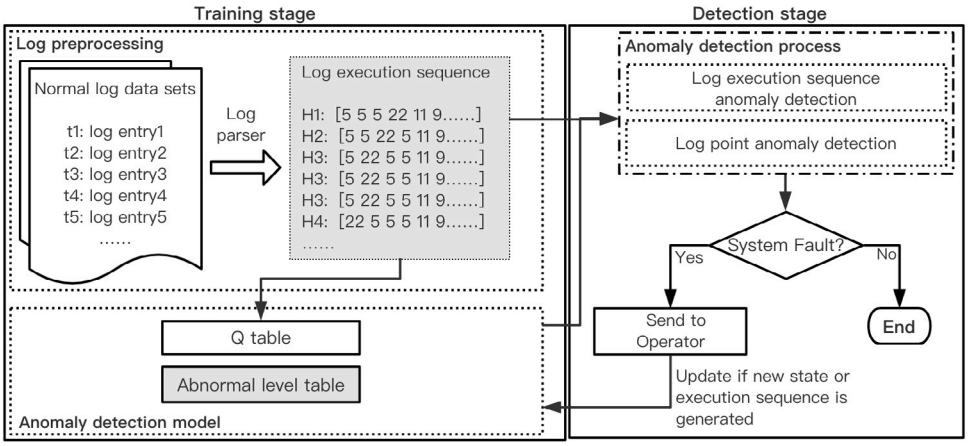


Fig. 4. The architecture of QLLog.

Detection stage. The goal of this stage is to detect two types of log anomalies, including log point anomalies and log execution anomalies. QLLog uses the Q table and abnormal level table to determine whether the log state is abnormal, then it uses the Q table and the designed special rules to determine whether the log execution sequence is abnormal. For both the point anomalies and the execution sequence anomalies, QLLog can get the feedback reports from ops and use them to update the anomaly detection model.

5.1. Anomaly detection model

In this section, we describe how to build the anomaly detection model. Our method first scans the training data to derive a list of log states in the order of their ID, and records the frequency of each state. The number of ID determines the row and column in Q table (i.e., the number of rows and columns are the same as the number of ID). The first column is the start state and the first row is the arrival state. We store the value of each execution path e_{xy} (i.e. Q value) in the table, and the default value is 0. Then, we find out all the final states of the log sequence set and collect the states of each position in the sequence. For example, the first position in the sequence, including state: 5, 22.

We define the state of the agent is a triple, $State = \langle S, \mathcal{R}, \mathcal{E} \rangle$, where $S = \{x | \forall x \in S\}$ is the current log state, $\mathcal{R} = R_{xy} = \{0, 1, 100\}$ is the immediate reward received after transitioning from the current log state to the next log state, and $\mathcal{E} = E = \{e_{xy} | e_{xy} \in E, \xi_{xy} = 0 \cup 1, \forall x, y \in S\}$ is the action.

In the training stage, the input is the log state of the current log sequence at time t, the output is a log state at time $t + 1$. QLLog adjusts the action according to the back-feed reward. For example, a log sequence: $L = [5, 22, 5, 26]$, given the first state 5, Q-learning outputs a log state s_i, $s_i \in S$, if $s_i = 22$ is same as the next state in log sequence, and s_i is the final state, the reward is 100; if s_i is same as the next state in log sequence, but s_i is not the final state, the reward is 1; if s_i is different from the next state in log sequence, the reward is 0. Then, take actual state 22 as the next input (i.e. the second log state in the log sequence). Fig. 5 summarizes the training process.

The log state transition process can be described as follows: if the current log sequence is $L = [5, 22, 5, 26]$, the agent initial state and the reward can be represented as $State = \langle \{5\}, \{\emptyset\}, \{\emptyset\} \rangle$. Q-learning outputs a log state 5, the agent state is updated to $State = \langle \{5\}, \{\langle 5, 0 \rangle\}, \{\emptyset\} \rangle$, state 5 and state 22 are different, so the reward in the triple is 0. Take 22 as the next input, the output is log state 5, the state is immediately updated to $State = \langle \{22\}, \{\langle 5, 1 \rangle\}, \{\langle e_{5,22} \rangle\} \rangle$. When the log state is the last state in log sequence, the agent state is $State = \langle \{26\}, \emptyset, \{\langle e_{5,22}, e_{22,5}, e_{5,26} \rangle\} \rangle$. Then, we get a complete execution sequence $H = \langle e_{5,22}, e_{22,5}, e_{5,26} \rangle$. In the process of state transition, Q value is updated synchronously, but if the reward of e_{xy} is 0, the Q value of the e_{xy} will not be updated. The update formula is shown in Formula (2).

$$\begin{aligned} \text{New } Q\left(x, e_{xy}\right) = & Q\left(x, e_{xy}\right) + \alpha [\ R\left(x, e_{xy}\right) + \\ & \gamma \max_{e'_{xy}} Q'\left(x', e'_{xy}\right) - Q\left(x, e_{xy}\right)\ ] \end{aligned} \tag{2}$$

where New$Q(x, e_{xy})$ is the new Q value, $Q(x, e_{xy})$ is the Q value of the current state, α is the learning rate. We can regard the learning rate as a measure of discarding the old value and generating the new value. If the learning rate is 1, it abandons the old value entirely. The learning rate chosen in this article is 1. $R(x, e_{xy})$ is the reward. γ is the discount factor which is a constant that satisfies $\gamma \in [0, 1]$. $\max_{e'_{xy}} Q'\left(x', e'_{xy}\right) - Q(x, e_{xy})$ is the execution path that can maximize Q value.

Q table is updated until the entire log execution sequence dataset has been processed. After training is done, the anomaly detection model generates a Q table with a fixed Q value. Fig. 6 shows the pseudo-code of the Q value update.

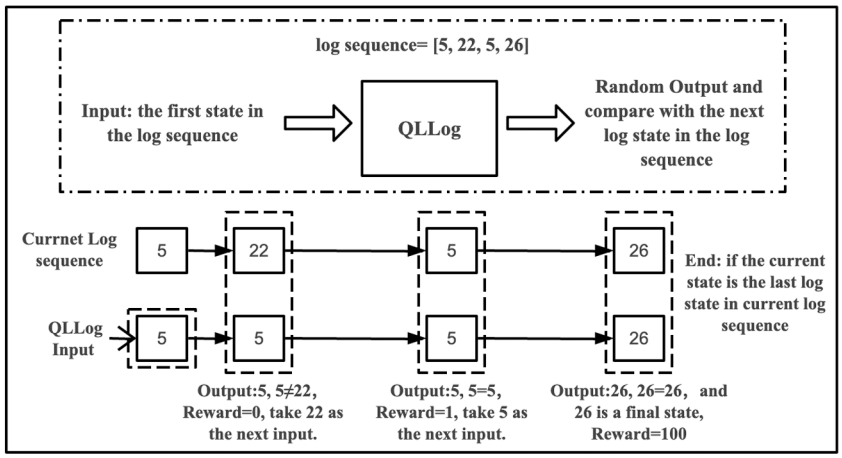


Fig. 5. Model training process.

Input: Log execution sequence set ExP
Output: Q value
Initialize Q value arbitrarily // Q=0
Repeat (for each episode): // Each log execution sequence is an episode
Initialize x
Repeat (for each step of episode):
Enter action e_{xy} based on the current log state x_i of the log execution sequence
Take action e_{xy}, observe R, x'

$$NewQ(x, e_{xy}) = Q(x, e_{xy}) + \alpha[R(x, e_{xy}) + \gamma maxQ'(x', e_{xy}') - Q(x, e_{xy})]$$

If $x' = x_{i+1}$
$x \leftarrow x'$;
Else
$x \leftarrow x_{i+1}$
until x is terminal state of log execution sequence

Fig. 6. The pseudo-code of the Q value update.

5.2. Log anomaly detection

System anomalies can be classified into three groups: point anomalies, execution sequence anomalies, and collective or group anomalies. This paper focuses on detecting two types of log anomalies: log point anomalies and log execution sequence anomalies. Log point anomalies often represent an irregularity or deviation that happens randomly. Log execution sequence anomalies represent each log entry in the log sequence is normal, but the execution order is abnormal. In this section, we introduce how to detect anomalies.

Given a log execution sequence, QLLog first detects the input state by using the Q table. If the log state in the Q table, the log state is normal. Otherwise, it is an abnormal log state. Then, QLLog determines whether the current abnormal state exists in the abnormal level table. If it is not in the abnormal level table, QLLog calculates the similarity between the log template of the current state and other normal log templates using the longest common subsequence. To improve the efficiency of the calculation, we use the dynamic programming method.

Let $M = \langle m_1, m_2, \ldots, m_i \rangle$ and $Z = \langle z_1, z_2, \ldots, z_j \rangle$ be the template of two log entries, where M is the template of abnormal log entry, Z is a normal log entry, $C[i, j]$ is the length of the longest common subsequence. Normally, the abnormal log differs greatly from the normal log, so the abnormal probability of log state is Formula (3):

$$P(abnormal) = \frac{C[i, j]}{M_i} \tag{3}$$

The higher the value of $P(abnormal)$, the lower the abnormal probability is. After getting the abnormal probability, QLLog sends the abnormal state and the abnormal probability to the ops. Finally, it updates the abnormal level table according to the report

Table 2
Anomalies and their levels in HDFS dataset.

Anomaly description	Frequency	Level
Namenode not updated after deleting block	4297	10
Write exception client give up	3225	10
Received block that does not belong to any file	1240	7
Delete a block that no longer exists on data node	724	5
Receive block exception	89	1
Replication monitor timedout	45	0

of ops. Only all the log states are normal, QLLog checks the log execution sequence. Given a log sequence $L = [5, 22, 5, 26]$, the detection rules and steps are as follows:

- QLLog first determines whether the log execution path is normal or not. For example, when the log state is moved from 5 to 22, we check the Q value of $e_{5,22}$ in the Q table. Only the log path $e_{5,22}$ exists in the Q table and the Q value is larger than the threshold k, the log execution path is normal. Otherwise, it is an abnormal path.
- If all log execution paths are normal, QLLog checks the last log state in the log sequence. If the last state is a final state, it is a normal log sequence. Otherwise, it is an abnormal log sequence.
- QLLog checks whether the state is in the right position. For example, in the training stage, the position of final state 26 is often sixth or seventh in the log sequence, but it is the fourth in the current log sequence, then the log sequence is abnormal.

When the current log sequence satisfies all the above conditions, the log sequence is normal. Otherwise, QLLog sends the abnormal log sequence to ops. Note that, whatever the type of anomaly, it must check whether the anomaly has been recorded in the abnormal level table before sending it to ops. If the level of the current anomaly is 0, the anomaly will not be sent. Otherwise, QLLog sends the anomaly to ops. The anomaly detection model will update according to the reports.

5.3. Update abnormal level table

Because of the operating system's update, new types of log entries (new log state) can emerge. Therefore, QLLog provides a feedback mechanism to update the abnormal level table. The abnormal level table consists of two parts: anomaly instance (include anomaly state and anomaly sequence) and level. The abnormal level table is empty in the training stage, which is dynamically updated according to the reports of ops in the detection stage. In this table, each false positive instance and the true anomaly correspond to a specific level, but one level corresponds to multiple anomalies. There are ten kinds of severity levels in this table. i.e., levels range from 1 to 10. Level 10 is the highest. As shown in Table 2, we list 6 anomalies with their frequency and levels, which are in the HDFS dataset.

In our case, the instances in the abnormal level table are sorted in descending order according to the level. For each newly arrived anomaly, it is inserted into the appropriate position in the table (to ensure the high-level anomalies are in front of the low-level anomalies) according to the level (provided by the ops). Such insertions can be done incrementally. Besides, we adjust the level according to the frequency of the anomalies during the update process. We believe that the higher the anomaly frequency, the more quickly it should be solved, except for the level 0 anomalies. For example, if ops report that the anomaly (state or sequence) is a false positive, QLLog inserts the instance in the abnormal level table and sets level 0. If ops report it is an abnormal state or sequence such as *"Namenode not updated after deleting block"* and set the level 7, which will update to a higher level 10 with the increase of its frequency. For high-level anomalies, the model takes priority to output them to help users solve the problem in time. For low-level anomalies such as level 0, the model ignores them.

6. Evaluation

In this section, we evaluate the performance of QLLog and compare it with these methods, such as LogCluster, DeepLog, IM and PCA. The platform used in Sections 6.2–6.6 are Python 3.7. All experiments use the same machine with an Intel (R) Xeon (R) Silver 2.20 GHz CPU, 128 G memory, and Windows 10 operating system.

6.1. Log datasets

We use distributed system log datasets as training sets and test sets, including HDFS dataset (Du et al., 2017; Xu, Huang, Fox, Patterson, & Jordan, 2009; Xu et al., 2010; Zhu et al., 2019) and OpenStack dataset (Du et al., 2017). These two datasets are labeled by domain experts.

HDFS dataset is generated through running Hadoop-based MapReduce jobs (such as distributed sort and text scan) on 203 Amazon's EC2 nodes. This dataset includes 11,197,954 log entries, about 2.9% are abnormal. Log entries are grouped into 575,060 log sequences by **block_id**. All anomalies are shown in Table 3.

OpenStack dataset is generated on one control node, one network node and eight compute nodes, including 155,508 log entries. Log entries are grouped into 5714 log sequences by **instance_id**, including 5544 normal log sequences and 170 abnormal log sequences. Three types of anomalies are injected at different execution points, which are shown in Table 3.

Table 3
Anomaly description of HDFS dataset and OpenStack dataset.

Dataset	Anomaly description
HDFS	Namenode not updated after deleting block Write exception client give up Write failed at beginning Replica immediately deleted Received block that does not belong to any file Redundant addStoredBlock Delete a block that no longer exists on data node Empty packet for block Receive block exception Replication monitor timedout Other anomalies
OpenStack	Neutron timeout during VM creation Libvirt error while destroying a VM Libvirt error during cleanup after destroying a VM

Table 4
Detail of the log datasets.

Dataset	Data size	Log entry	Log state	Log sequence
HDFS	1.54 GB	11,197,954	29	575,060
OpenStack	45.9 MB	155,508	40	5714

Details of the HDFS dataset and OpenStack dataset could be found in Du et al. (2017), He, Zhu, He, and Lyu (2016) and Xu et al. (2010). Zhu et al. (2019) have officially released these datasets in 2018, and we have made no modification to these datasets in this paper. These datasets have been downloaded over 22,470 times. Table 4 summarizes the two datasets.

6.2. Accuracy evaluation of anomaly detection

In this section, we evaluate the accuracy of the QLLog. A detection method's capability is usually assessed by the following three intuitive metrics: *Recall*, *Precision* and *F-measure*. These evaluation metrics are used in previous studies frequently (Bertero, Roy, Sauvanaud, & Tredan, 2017; He, Zhu, He, Li, & Lyu, 2016; Siddiqui et al., 2018). The definitions of the metrics are as follows:

$$Precision = \frac{TP}{TP + FP} \tag{4}$$

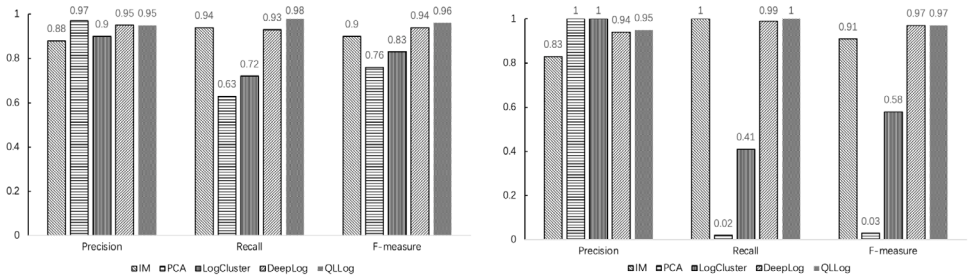
$$Recall = \frac{TP}{TP + FN} \tag{5}$$

$$F - measure = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} \tag{6}$$

True positives (TP) are anomalies that are accurately determined as such by the method. True negatives (TN) are normal instances that are accurately determined as normal instances. If the method determines that an instance (log state or log sequence) is an abnormal one, but it is actually normal, we label the outcome as a false positive (FP). False negatives (FN) are anomalies that are incorrectly missed by the detection method. *F-measure* is the harmonic mean of *Precision* and *Recall*.

We run five unsupervised anomaly detection methods such as QLLog, LogCluster, DeepLog, IM and PCA to detect the anomalies for the HDFS dataset and OpenStack dataset respectively. PCA and IM do not require training data. DeepLog, LogCluster and QLLog only need normal training data. We use the 10-fold cross-validation model to evaluate the five methods (leverage the front 90% as the training data, and the rest 10% as the testing data). The benefit of 10-fold cross-validation is that all logs are used for both training and validation. Fig. 7 shows the comparison using *Recall*, *Precision* and *F-measure*.

Fig. 7(a) shows the performance of these methods in the HDFS dataset. Fig. 7(b) shows the performance of these methods in the OpenStack dataset. Overall, QLLog has achieved the best accuracy among the five methods, having an averaged *F-measure* 0.96 on the HDFS dataset and 0.97 on the OpenStack dataset. LogCluster has high *Precision* but with low *Recall* in both two datasets, because its detection strategy is too coarse-grained. PCA has the worst *Recall* and *F-measure*, especially in the OpenStack dataset (almost all instances are detected as anomalies) because the PCA method detects anomalies by variance, but the log sequences in both HDFS and OpenStack datasets are regular. IM is good at mining data rules. Therefore, it shows reasonable performance in both the HDFS dataset and OpenStack dataset. DeepLog has the same *Precision* 0.95 with QLLog in HDFS dataset, but the *Recall* of QLLog is better than DeepLog.



(a) Accuracy on HDFS (b) Accuracy on OpenStack

Fig. 7. Evaluation on HDFS log and OpenStack log.

Table 5
Set up of HDFS log dataset (unit: sequence).

Dataset	Training data (if needed)	Test data	Log template
HDFS	Normal: 111 644 Abnormal: 0	Normal: 558,222 Abnormal: 16,838	29 40

Table 6
Number of FP and FN on HDFS dataset.

	PCA	IM	LogCluster	DeepLog	QLLog
FP	248	2028	1482	824	819
FN	5606	1253	4947	1058	265
TP	11 232	15 585	11 891	15 780	16 573
TN	502 153	556 194	556 740	557 398	557 403

6.3. *FN And FP evaluation of anomaly detection*

In this section, we only leverage the front 30% as the training data, and the 100% (575 060 log sequences) as the testing data, including 558 222 normal sequences and 16 838 abnormal sequences. Table 5 summarizes the dataset used in this case. Table 6 shows the test results.

As shown in Table 6, PCA achieves the fewest false positives, but at the price of more false negatives. LogCluster has quite a high rate of false negatives. Although IM can mine the relationship between logs, it still has the problem of high false positive rate and high false negative rate. Detection results of DeepLog and QLLog are good, especially QLLog. QLLog has few false positives and the fewest false negatives. Then, we take this a step further and interpret the results in terms of false positives and false negatives.

False Positive Case: When the map-reduce system distributes job configuration files to all the nodes, a few blocks are replicated 10 times instead of 3 times for the majority of blocks. We define this false positive as *"multiple replica (for task/job desc files)"*. It is a rare (only 372 in total log entries) and suspicious situation compared to the data accesses. However, it is normal by the system design (Xu et al., 2010). Many existing methods cannot solve this problem, such as PCA and DeepLog. Even if DeepLog can update the model by modifying the weight, this problem cannot be avoided. Nevertheless, in our case, we can record this suspicious situation in the abnormal level table and set level 0. Therefore, when the "anomaly" comes again, it is ignored (no warning). Another false positive is caused by the suspicious keyword "Exception" in the template. Some methods that use "Error" and "Exception" as classification features may classify such logs as anomalies. However, due to QLLog captured the relationship between logs during the training stage, it can avoid the false positive by using sequence anomaly detection.

False negative case: We found one type of anomaly that LogCluster and PCA did not find. For example, in a certain (relatively rare) path, when a block is deleted (due to temporary over-replication), the record on the namenode is not updated until the next write to the block, causing the file system to believe in a replica that no longer exists, which causes subsequent block deletion to fail. No log records the existence of this anomaly. Therefore, it is impossible to detect this anomaly only by detecting the abnormal log, but it is easy to catch the anomaly by using sequence anomaly detection.

However, QLLog does report some false negatives, which are common in some methods (PCA, IM and LogCluster) that do not consider timeout anomalies. For example, we found that one of the data nodes takes far longer to respond than all others do. This situation will not cause serious system problems, but this scenario slows down the entire system to the speed of the slowest responding node. Significantly, after a long write session, these blocks will eventually be correctly written. Therefore, these execution sequences are complete. In fact, if the sequence contains all the events of a given pattern, it should be marked as correct regardless of when the event occurs. Based on this fact, we failed to detect such timeout anomalies. Besides, as shown in Table 2, there are 45 timeout anomalies in the HDFS dataset. The frequency of timeout anomaly also proves that it is reasonable to divide the level of the anomaly by frequency. In real-time detection, the anomaly of timeout may be set to level 0 or level 1. Nevertheless, to ensure the fairness of the experiment, we record these 45 undetected anomalies.

Table 7
Running time with increasing data size (unit: second).

Method	5k	500k	1M	5M	10M	50M
QLLog	0.03	2.71	5.70	34.64	54.68	260.49
DeepLog	1.09	45.71	67.05	210.44	322.56	602.96
IM	0.97	19.32	25.99	172.27	285.18	1529.13

Table 8
Training time (unit: second).

Method	1%	5%	10%	50%
QLLog	43.26	205.11	400.59	2062.76
DeepLog	3782.15	>2 h	>2 h	>2 h

6.4. Scalability evaluation of anomaly detection

In this section, the scalability of these anomaly detection methods is evaluated on the HDFS dataset with varying sizes (from 5k to 50M). We chose the detection methods that *F-measure* are more than 0.9 for comparison. The results are shown in Table 7.

As shown in Table 7. Among these methods, QLLog has the best scalability. We can observe that QLLog, DeepLog and IM anomaly detection methods scale linearly as the data size increases. QLLog maintains a high speed for both small and large datasets. The average speed of QLLog (1.83×10^2 kb/s) is about 5 times faster than DeepLog (3.36×10^1 kb/s) and IM (2.83×10^1 kb/s). When the data size is less than 10M, the running time of IM is ideal, but when the data size is more than 10M, the running time of IM increases significantly. It is caused by IM brute force searching on a large dataset (unnecessary search for high-dimensional correlations). Therefore, to control the running time, it must be set the stopping criteria to control its brute force searching process on the large datasets. Significantly, even if the data size is increased to 10M, the running time of QLLog is still less than one minute. But the running time of DeepLog in 5M data size is already above two minutes. When the accuracy of the detection methods is almost the same (e.g., DeepLog and QLLog), the faster the detection speed, the more conducive it is to help the ops locate and deal with the anomaly in time. In addition to the above three methods, due to the low accuracy of LogCluster, we do not compare it in this section. But it is necessary to point out that the time complexity of LogCluster is $O(n^2)$. Clearly, LogCluster cannot handle large-scale datasets in an acceptable time (more than 1 h in dealing with 50M data size).

To further prove the advantages of QLLog, we calculate the training time of DeepLog and QLLog on different training data size. i.e., select 1%, 5%, 10% and 50% of the total dataset (11,197,954) as the training data. Table 8 shows the results.

As shown in Table 8. Although DeepLog points out that it only needs a small number (1%) datasets to obtain high accuracy, it still takes much time (3782.15 s) to train the model. Compared with DeepLog, QLLog needs more training data to ensure accuracy, but it can train a large amount of data in a short time. Besides, normal log data is universally available in nearly all computer systems. In summary, the results prove that the QLLog is very competitive compared to the state-of-art methods.

6.5. Security case studies

SYN flooding attacks are a common type of Distributed Denial-of-Service (DDoS) attack. SYN flooding attacks exploit the TCP's three-way handshake mechanism and its limitation in maintaining half-open connections. When a server receives a SYN request, it returns a SYN/ACK packet to the client. Until the SYN/ACK packet is acknowledged by the client, the connection remains in the half-open state for a period of up to the TCP connection timeout, which is typically set to 75 s. The server has built-in its system memory a backlog queue to maintain all half-open connections. Since this backlog queue is finite, once the backlog queue limit is reached, all connection requests will be dropped. If a SYN request is spoofed, the victim server will never receive the final ACK packet to complete the three-way handshake. Flooding spoofed SYN requests can easily exhaust the victim server's backlog queue, causing all the incoming SYN requests to be dropped.

QLLog can utilize the inherent TCP SYNFIN pairs' behavior to construct the execution sequence (i.e., one appearance of a SYN packet results in the eventual return of a FIN packet). Besides, since DDoS attack traffic uses randomly spoofed source IP addresses to disguise their true identities, the variation of new IP addresses can serve as another fundamental feature (state) of DDoS attacks. To detect SYN flooding attacks, we set a fixed time window for QLLog in the detection stage. In each time window, if the number of the new state (i.e., the new source IP address) is over the threshold and the execution sequences are incomplete, QLLog will report the SYN flooding attacks. To test the performance of QLLog on SYN flooding attacks, we used the DDoS attack data generated by the TFN2K tool. The detection accuracy of QLLog is over 90%.

6.6. Limitation

We found some types of log datasets in which could not be detected anomalies quickly and accurately. For example, the Blue Gene/L supercomputer system dataset (Du et al., 2017; Liang, Zhang, Sivasubramaniam, Jette, & Sahoo, 2006) contains 4 747 963 log entries and 385 log types (log states). There are 348 698 (7%) log entries labeled as anomalies. As the number of log states increases in Q-learning, the Q table and abnormal level table become too large to store and learn efficiently. Besides, QLLog may send numerous false positives to ops, and reduce detection efficiency. Therefore, we need to select the log dataset that has a few log states, such as HDFS and OpenStack log datasets, when we apply OLLog to the real operation process of the log dataset.

7. Conclusions

This paper proposes a log anomaly detection method based on Q-learning, namely QLLog, which can detect multiple types of system anomalies, rank the severity level of abnormal events, and update the detection model. It performs anomaly detection not only at per log sequence but also at per log entry. By incorporating ops feedback report, QLLog supports online update to its anomaly detection model. Hence it is able to incorporate and adapt to the new log state and log sequence. Experiments on real datasets show the effectiveness of QLLog. Compared with the state of the art, QLLog can detect anomalies from log datasets with high accuracy and outperform other existing log-based anomaly detection methods. In addition, for end-users, such as operators, integrators, and developers, it is necessary to detect anomalies in time to reduce and avoid losses. Due to the humanized feedback mechanism and accurate detection results of QLLog, end-users can use it to locate high-level anomalies easily and accurately as a real-world requirement. It is greatly improving work efficiency to reduce the severe repercussions of anomalies on system performance and operating costs. Our future work is to consider but is not limited to combining Q-learning with the LSTM (LeCun, Bengio, & Hinton, 2015) or RNN (Luo, Liu, & Gao, 2017) neural network to resolve the limitation of QLLog.

CRediT authorship contribution statement

Xiaoyu Duan: Conceptualization, Methodology, Writing - original draft preparation, Writing - reviewing & editing. **Shi Ying:** Supervision, Project administration. **Wanli Yuan:** Software, Data curation, Visualization, Reviewing. **Hailong Cheng:** Data curation, Visualization, Validation, Reviewing. **Xiang Yin:** Reviewing.

Acknowledgments

The work was supported by National Natural Science Foundation of China under grant NO. 61672392 and NO. 61373038, the National Key Research and Development Program of China under grant NO. 2016YFC1202204.

References

Astekin, M., Zengin, H., & Sözer, H. (2019). DILAF: A framework for distributed analysis of large-scale system logs for anomaly detection. *Software - Practice and Experience*, *49*(2), 153–170. http://dx.doi.org/10.1002/spe.2653.

Bertero, C., Roy, M., Sauvanaud, C., & Tredan, G. (2017). Experience report: Log mining using natural language processing and application to anomaly detection. In *2017 IEEE 28th international symposium on software reliability engineering (ISSRE)* (pp. 351–360). IEEE, http://dx.doi.org/10.1109/ISSRE.2017.43.

Christopher, P., & Watkins, L. (1992). Q-learning. *Machine Learning*, *8*, 279–292.

Clifton, J., & Laber, E. (2020). Q-learning: Theory and applications. *Annual Review of Statistics and Its Application*, *7*, 279–301. http://dx.doi.org/10.1146/annurev-statistics-031219-041220.

Das, A., Mueller, F., Siegel, C., & Vishnu, A. (2018). Desh: deep learning for system health prediction of lead times to failure in HPC. In *International symposium* (pp. 40–51). http://dx.doi.org/10.1145/3208040.3208051.

Ding, N., Ma, H., Gao, H., Ma, Y., & Tan, G. (2019). Real-time anomaly detection based on long short-term memory and gaussian mixture model. *Computers & Electrical Engineering*, *79*, Article 106458. http://dx.doi.org/10.1016/j.compeleceng.2019.106458.

Du, M., & Li, F. (2016). Spell: Streaming parsing of system event logs. In *2016 IEEE 16th international conference on data mining (ICDM)* (pp. 859–864). IEEE, http://dx.doi.org/10.1109/ICDM.2016.0103.

Du, M., Li, F., Zheng, G., & Srikumar, V. (2017). Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the ACM conference on computer and communications security* (pp. 1285–1298). ACM Press, http://dx.doi.org/10.1145/3133956.3134015.

Enderlein, G., & Hawkins, D. M. (1987). Identification of outliers. *Biometrical Journal - BIOM J*, *29*, 198. http://dx.doi.org/10.1002/bimj.4710290215.

Haddadpajouh, H., Dehghantanha, A., Khayami, R., & Choo, K.-K. R. (2018). A deep recurrent neural network based approach for internet of things malware threat hunting. *Future Generation Computer Systems*, *85*, 88–96. http://dx.doi.org/10.1016/j.future.2018.03.007.

Hasan, M., Orgun, M. A., & Schwitter, R. (2019). Real-time event detection from the twitter data stream using the twitternews+ framework. *Information Processing & Management*, *56*(3), 1146–1165. http://dx.doi.org/10.1016/j.ipm.2018.03.001.

He, P., Zhu, J., He, S., Li, J., & Lyu, M. R. (2016). An evaluation study on log parsing and its use in log mining. In *2016 46th annual IEEE/IFIP international conference on dependable systems and networks (DSN)* (pp. 654–661). IEEE, http://dx.doi.org/10.1109/DSN.2016.66.

He, S., Zhu, J., He, P., & Lyu, M. R. (2016). Experience report: System log analysis for anomaly detection. In *2016 IEEE 27th international symposium on software reliability engineering (ISSRE)* (pp. 207–218). http://dx.doi.org/10.1109/ISSRE.2016.21.

He, P., Zhu, J., Zheng, Z., & Lyu, M. R. (2017). Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE international conference on web services (ICWS)* (pp. 33–40). IEEE, http://dx.doi.org/10.1109/ICWS.2017.13.

Huang, C., Zhou, S., Xu, J., Niu, Z., Zhang, R., & Cui, S. (2014). Markov decision process. In *Signal processing for cognitive radios* (pp. 207–268). John Wiley & Sons, Ltd, http://dx.doi.org/10.1002/9781118824818.ch7, Ch. 7.

Jahromi, A. N., Hashemi, S., Dehghantanha, A., Choo, K.-K. R., Karimipour, H., Newton, D. E., et al. (2020). An improved two-hidden-layer extreme learning machine for malware hunting. *Computers & Security*, *89*, Article 101655. http://dx.doi.org/10.1016/j.cose.2019.101655.

Javed, A., Burnap, P., & Rana, O. (2019). Prediction of drive-by download attacks on twitter. *Information Processing & Management*, *56*(3), 1133–1145. http://dx.doi.org/10.1016/j.ipm.2018.02.003.

Kaur, R., & Singh, S. (2017). A comparative analysis of structural graph metrics to identify anomalies in online social networks. *Computers & Electrical Engineering*, *57*, 294–310. http://dx.doi.org/10.1016/j.compeleceng.2016.11.018.

Kwon, D., Kim, H., Kim, J., Suh, S., Kim, I., & Kim, K. (2019). A survey of deep learning-based network anomaly detection. *Cluster Computing*, *22*, 949–961. http://dx.doi.org/10.1007/s10586-017-1117-8.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, *521*, 436–444. http://dx.doi.org/10.1038/nature14539.

Liang, Y., Zhang, Y., Sivasubramaniam, A., Jette, M., & Sahoo, R. (2006). Bluegene/l failure analysis and prediction models. In *Proceedings of the international conference on dependable systems and networks* (pp. 425–434). http://dx.doi.org/10.1109/DSN.2006.18.

Lin, Q., Zhang, H., Lou, J.-G., Zhang, Y., & Chen, X. (2016). Log clustering based problem identification for online service systems. In *Proceedings of the 38th international conference on software engineering companion* (pp. 102–111). Association for Computing Machinery, http://dx.doi.org/10.1145/2889160.2889232.

Liu, F. T., Ting, K., & Zhou, Z.-H. (2009). Isolation forest. In *Proceedings of the 2008 eighth IEEE international conference on data mining* (pp. 413–422). IEEE Computer Society, http://dx.doi.org/10.1109/ICDM.2008.17.

Lou, J.-G., Fu, Q., Yang, S., Xu, Y., & Li, J. (2010). Mining invariants from console logs for system problem detection. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference* (pp. 231–244). USA: USENIX Association, http://dx.doi.org/10.5555/1855840.1855864.

Luo, W., Liu, W., & Gao, S. (2017). Remembering history with convolutional LSTM for anomaly detection. In *2017 IEEE international conference on multimedia and expo (ICME)* (pp. 439–444). http://dx.doi.org/10.1109/ICME.2017.8019325.

Luo, W., Liu, W., & Gao, S. (2017). A revisit of sparse coding based anomaly detection in stacked rnn framework. In *2017 IEEE international conference on computer vision (ICCV)* (pp. 341–349). http://dx.doi.org/10.1109/ICCV.2017.45.

Marchi, E., Vesperini, F., Weninger, F., Eyben, F., Squartini, S., & Schuller, B. (2015). Non-linear prediction with lstm recurrent neural networks for acoustic novelty detection. In *IJCNN 2015* (pp. 1–7). http://dx.doi.org/10.1109/IJCNN.2015.7280757.

Meng, W., Liu, Y., Zhu, Y., Zhang, S., Pei, D., Liu, Y., et al. (2019). Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs. (pp. 4739–4745). http://dx.doi.org/10.24963/ijcai.2019/658.

Mudassar, B., Ko, J., & Mukhopadhyay, S. (2018). An unsupervised anomalous event detection framework with class aware source separation. In *ICASSP 2018 - 2018 IEEE international conference on acoustics, speech and signal processing (ICASSP)* (pp. 2671–2675). http://dx.doi.org/10.1109/ICASSP.2018.8462309.

Ren, R., Fu, X., Zhan, J., & Zhou, W. (2012). Logmaster: Mining event correlations in logs of large-scale cluster systems. In *Proceedings of the IEEE symposium on reliable distributed systems* (pp. 71–80). http://dx.doi.org/10.1109/SRDS.2012.40.

Scholkopf, B., & Smola, A. J. (2018). *Learning with kernels: Support vector machines, regularization, optimization, and beyond*. MIT Press, http://dx.doi.org/10.7551/mitpress/4175.001.0001.

Siddiqui, A., Fern, A., Dietterich, T., Wright, R., Theriault, A., & Archer, D. (2018). Feedback-guided anomaly discovery via online optimization. In *KDD '18, Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining* (pp. 2200–2209). http://dx.doi.org/10.1145/3219819.3220083.

Tong, Z., Chen, H., Deng, X., Li, K., & Li, K. (2019). A scheduling scheme in the cloud computing environment using deep Q-learning. *Information Sciences*, *512*, 1170–1191. http://dx.doi.org/10.1016/j.ins.2019.10.035.

Turkoz, M., Kim, S., Son, Y., Jeong, M. K., & Elsayed, E. A. (2020). Generalized support vector data description for anomaly detection. *Pattern Recognition*, *100*, Article 107119. http://dx.doi.org/10.1016/j.patcog.2019.107119.

Watanabe, Y., Otsuka, H., Sonoda, M., Kikuchi, S., & Matsumoto, Y. (2012). Online failure prediction in cloud datacenters by real-time message pattern learning. In *4th IEEE international conference on cloud computing technology and science proceedings* (pp. 504–511). http://dx.doi.org/10.1109/CloudCom.2012.6427566.

Wu, X., Turner, D., Chen, C.-C., Maltz, D. A., Yang, X., Yuan, L., et al. (2012). Netpilot: Automating datacenter network failure mitigation. In *Proceedings of the ACM SIGCOMM 2012 conference on applications, technologies, architectures, and protocols for computer communication - SIGCOMM '12* (p. 419). ACM Press, http://dx.doi.org/10.1145/2342356.2342438.

Xu, W., Huang, L., Fox, A., Patterson, D., & Jordan, M. (2009). Online system problem detection by mining patterns of console logs. In *2009 Ninth IEEE international conference on data mining* (pp. 588–597). IEEE, [ISSN: 15504786] ISBN: 978-1-4244-5242-2, http://dx.doi.org/10.1109/ICDM.2009.19.

Xu, W., Huang, L., Fox, A., Patterson, D., & Jordan, M. I. (2010). Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on operating systems principles - SOSP '09* (pp. 117–131). ACM Press, http://dx.doi.org/10.1145/1629575.1629587.

Ye, Y., Li, T., Adjeroh, D., & Iyengar, S. (2017). A survey on malware detection using data mining techniques. *ACM Computing Surveys*, *50*, 1–40. http://dx.doi.org/10.1145/3073559.

Yu, X., Joshi, P., Xu, J., Jin, G., Zhang, H., & Jiang, G. (2016). Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. In *Proceedings of the twenty-first international conference on architectural support for programming languages and operating systems - ASPLOS '16* (pp. 489–502). ACM Press, http://dx.doi.org/10.1145/2872362.2872407, (2).

Yuan, G., Li, B., Yao, Y., & Zhang, S. (2017). A deep learning enabled subspace spectral ensemble clustering approach for web anomaly detection. In *2017 international joint conference on neural networks (IJCNN)* (pp. 3896–3903). http://dx.doi.org/10.1109/IJCNN.2017.7966347.

Zhang, Y., & Sivasubramaniam, A. (2008). Failure prediction in IBM bluegene/l event logs. In *2008 IEEE international symposium on parallel and distributed processing* (pp. 1–5). http://dx.doi.org/10.1109/IPDPS.2008.4536397.

Zhang, S., Zhang, Y., Chen, Y., Dong, H., Qu, X., Song, L., et al. (2018). Prefix: Switch failure prediction in datacenter networks. In *Proceedings of the ACM on measurement and analysis of computing systems, Vol. 2* (pp. 1–29). http://dx.doi.org/10.1145/3179405.

Zhu, J., He, S., Liu, J., He, P., Xie, Q., Zheng, Z., et al. (2019). Tools and benchmarks for automated log parsing. In *2019 IEEE/ACM 41st international conference on software engineering: Software engineering in practice (ICSE-SEIP)* (pp. 121–130). IEEE, http://dx.doi.org/10.1109/ICSE-SEIP.2019.00021.