Loader: A Log Anomaly Detector Based on Transformer

Tong Xiao, Zhe Quan, Zhi-Jie Wang, *Member, IEEE,* Yuquan Le, Yunfei Du, Xiangke Liao, Kenli Li, *Senior Member, IEEE,* and Keqin Li, *Fellow, IEEE*

Abstract—Detecting anomalies in logs is crucial for service and system management, since logs are widely used to record the runtime status, and are often the only data available for postmortem analysis. Since anomalies are usually rare in real-world services and systems, a common and feasible practice is to mine or learn normal patterns from logs, and deem the breakers as anomalies. As log sequences are a kind of time series data, RNN (Recurrent Neural Network) and its variants have been extensively employed to capture the normal patterns. Nevertheless, the sequential nature of RNN and its variants makes them hard to parallelize and capture long-term dependencies, which may hinder their performance. To address this issue, in this paper we propose Loader, a novel semi-supervised **lo**g **a**nomaly **d**etector based on Transform**er**, because the Transformer architecture eschews recurrence and is able to draw global dependencies. Loader leverages the Transformer encoder to capture normal patterns from normal log sequences. When detecting, it gives a set of candidate log templates, that may appear after the input log substring under normal conditions. If the template of the actual next log message is not within the candidate set, this implies an anomaly. Previous similar methods select the most possible k log templates all the time, the performance is sensitive to k, and it is nontrivial to pick a proper k. To alleviate this, we design a more flexible and robust 'top-p' algorithm, which determines the candidate set based on the cumulative probability of the most possible log templates. Extensive experiments are conducted based on three public log datasets, the experimental results validate the effectiveness and competitiveness of our approach.

Index Terms—Log anomaly detection, log analysis, Transformer, service and system management.

1 Introduction

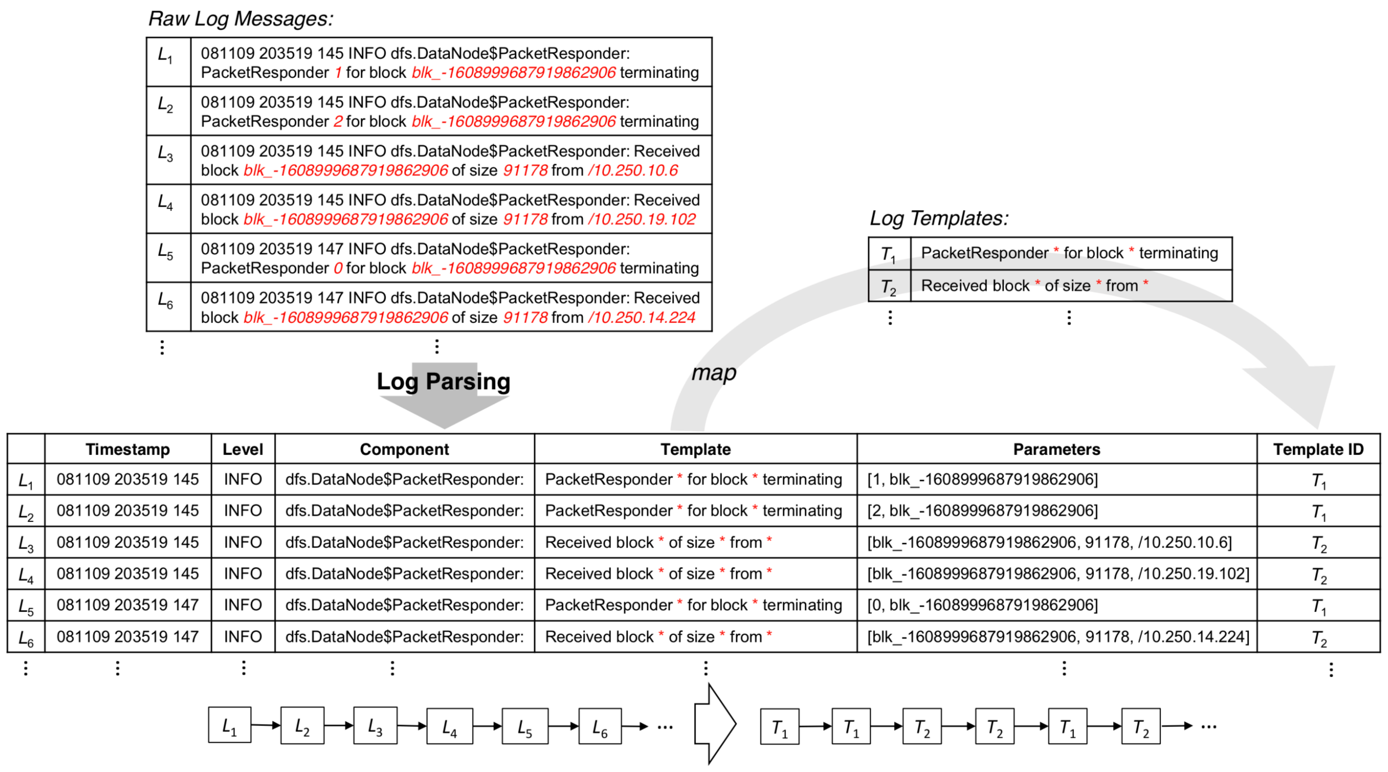
With the rapid development of cloud computing and big data, more and more traditional applications and systems are migrated to cloud platforms and accessed via the Internet as web services. As the services and systems are becoming increasingly complex and large-scale, they are more vulnerable to various bugs and malicious attacks, making service and system management a challenging task. Especially for giant cloud service providers like Amazon, Microsoft, Google, Alibaba, and Huawei, who often provide services for millions of users all around the world on a 24×7 basis, high RAS (Reliability, Availability, and Serviceability) is critical, since a subtle issue may lead to serious consequences and tremendous economic losses [1], [2], [3], [4].

Logs have always played an important role in service and system management [5], [6], since they are widely used by programs to record significant events and runtime information. When an exception or error occurs, support engineers and operators tend to dive into the logs to look for clues, as logs are often the only data at hand for postmortem analysis. Unfortunately, complex and large-scale services and systems often consist of numerous components, and produce tons of logs every day. This makes it cumbersome, error-prone, and even infeasible to analyze the logs manually in time. As a result, automated log analysis is imperative and has drawn extensive attention from both academia and industry [7], [8], [9], [10]. And one of the critical tasks is to detect anomalies in logs automatically and timely, *i.e.*, automated log anomaly detection.

There have been a lot of efforts devoted to automated log anomaly detection, employing various techniques from traditional machine learning [7], [9] to deep learning [11], [12]. For most of them, a prerequisite is log parsing, which aims to transform raw textual logs into a structured format, as illustrated in Fig. 1. With the help of log parsing, we can map massive logs to a small set of log templates. Then a sequence of log messages (*i.e.*, a log sequence) can be represented by a list of templates (template IDs), as shown in the bottom part of Fig. 1. The aim of log anomaly detection is to uncover anomalous patterns or behaviors in log sequences, the input is log sequences, rather than individual log messages. **Note:** *For simplicity, in the rest of this paper, when it is about log anomaly detection, if we say a* ***log message****, we mean its template (template ID); and a* ***log sequence*** *refers to the list of templates (template IDs) of a sequence of log messages.*

Generally, we can classify automated log anomaly detection methods into three categories: supervised, unsupervised, and semi-supervised methods. Supervised methods need both positive (anomalous) and negative (normal) samples (log sequences), together with their labels to train a

- *Tong Xiao, Zhe Quan, Yuquan Le, and Kenli Li are with the College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan 410082, China. E-mail: {xiaotong18, quanzhe, leyuquan, lkl}@hnu.edu.cn.*
- *Zhi-Jie Wang is with the College of Computer Science, Chongqing University, Chongqing 400044, China. E-mail: cszjwang@cqu.edu.cn.*
- *Yunfei Du is with Huawei Technologies Co., Ltd., Shenzhen, Guangdong 518129, China. E-mail: duyunfei5@huawei.com.*
- *Xiangke Liao is with the College of Computer, National University of Defense Technology, Changsha, Hunan 410073, China. E-mail: xkliao@nudt.edu.cn.*
- *Keqin Li is with the Department of Computer Science, State University of New York, New Paltz, NY 12561, USA. E-mail: lik@newpaltz.edu.*



```
L1  081109 203519 145 INFO dfs.DataNode$PacketResponder: PacketResponder 1 for block blk_-1608999687919862906 terminating
L2  081109 203519 145 INFO dfs.DataNode$PacketResponder: PacketResponder 2 for block blk_-1608999687919862906 terminating
L3  081109 203519 145 INFO dfs.DataNode$PacketResponder: Received block blk_-1608999687919862906 of size 91178 from /10.250.10.6
L4  081109 203519 145 INFO dfs.DataNode$PacketResponder: Received block blk_-1608999687919862906 of size 91178 from /10.250.19.102
L5  081109 203519 147 INFO dfs.DataNode$PacketResponder: PacketResponder 0 for block blk_-1608999687919862906 terminating
L6  081109 203519 147 INFO dfs.DataNode$PacketResponder: Received block blk_-1608999687919862906 of size 91178 from /10.250.14.224
⋮
```

T_1	PacketResponder * for block * terminating
T_2	Received block * of size * from *
⋮	⋮

	Timestamp	Level	Component	Template	Parameters	Template ID
L_1	081109 203519 145	INFO	dfs.DataNode$PacketResponder:	PacketResponder * for block * terminating	[1, blk_-1608999687919862906]	T_1
L_2	081109 203519 145	INFO	dfs.DataNode$PacketResponder:	PacketResponder * for block * terminating	[2, blk_-1608999687919862906]	T_1
L_3	081109 203519 145	INFO	dfs.DataNode$PacketResponder:	Received block * of size * from *	[blk_-1608999687919862906, 91178, /10.250.10.6]	T_2
L_4	081109 203519 145	INFO	dfs.DataNode$PacketResponder:	Received block * of size * from *	[blk_-1608999687919862906, 91178, /10.250.19.102]	T_2
L_5	081109 203519 147	INFO	dfs.DataNode$PacketResponder:	PacketResponder * for block * terminating	[0, blk_-1608999687919862906]	T_1
L_6	081109 203519 147	INFO	dfs.DataNode$PacketResponder:	Received block * of size * from *	[blk_-1608999687919862906, 91178, /10.250.14.224]	T_2
⋮	⋮	⋮	⋮	⋮	⋮	⋮

$L_1 \rightarrow L_2 \rightarrow L_3 \rightarrow L_4 \rightarrow L_5 \rightarrow L_6 \rightarrow \cdots \Rightarrow T_1 \rightarrow T_1 \rightarrow T_2 \rightarrow T_2 \rightarrow T_1 \rightarrow T_2 \rightarrow \cdots$

Fig. 1. Illustration of log parsing.

model, *e.g.*, decision tree [13], support vector machine (SVM) [14], logistic regression [15], and Long Short-Term Memory (LSTM) [16]. Since anomalies are usually rare in real-world environments, there often exists an extreme imbalance between positive and negative samples, and it is not easy to collect enough positive (anomalous) samples for training. These would limit the application of supervised methods. On the contrary, unsupervised methods do not need labeled samples. They directly process all samples to figure out outliers, through techniques such as principal component analysis (PCA) [7] and invariants mining [17]. But they suffer lower performance than supervised ones [9]. Semi-supervised methods try to incorporate the merits of both supervised and unsupervised methods, by only requiring partial samples to be labeled. A common practice is to learn normal patterns from normal log sequences, then make judgments according to whether the given log sequences obey the patterns or not. For example, DeepLog [11] and LogAnomaly [12] employ LSTM to capture latent patterns from normal log sequences, and try to predict the most possible log templates that could appear after the input log substring under normal conditions. If the template of the actual next log message is not among the most possible templates, then it is anomalous.

Considering that a service or system would work in normal state most of the time, an overwhelming majority of its logs would be normal, it is relatively easy to obtain abundant log sequences that can be labeled as normal. This makes it more practical and appealing to adopt a semi-supervised method in log anomaly detection.

As logs are generated over time, log sequences, which consist of log messages in time order, are essentially a kind of time series data. Therefore, it can be easily understood that, RNN (Recurrent Neural Network) and its variants, including LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit), have been widely applied to log anomaly detection [11], [12], [16], [18], [19]. However, in an RNN model, each step relies on its previous step, which makes the model hard to parallelize and capture long-term dependencies, as illustrated in Fig. 2a.

To alleviate the above issue, in this paper, we put forward a novel semi-supervised log anomaly detection approach, which leverages the Transformer encoder to replace RNN and its variants. Since the Transformer architecture eschews recurrence by self-attention and positional encoding, and is able to draw global dependencies [20], as illustrated in Fig. 2b. In recent years, we have witnessed the great success of the Transformer architecture in various fields, where RNN and its variants had played important roles, including natural language processing (NLP) [21], computer vision (CV) [22], speech applications [23], *etc.* We dub our approach as Loader (**Log** **a**nomaly **d**etector based on Transform**er**).

Specifically, we borrow the general idea behind DeepLog and LogAnomaly which utilize LSTM, *i.e.*, we train Loader only using normal log sequences to capture normal patterns; and when detecting, Loader provides a set of candidate log templates, that are most likely to appear after the input log substring under normal conditions. If the template of the actual next log message is not within the candidate set, then we say there is an anomaly. Take the log sequence in Fig. 1 as an instance, and suppose we have a substring of the first three log messages ($L_1 \rightarrow L_2 \rightarrow L_3$), which can be represented by $T_1 \rightarrow T_1 \rightarrow T_2$. After inputting this log substring into Loader, it will give a set of candidate log

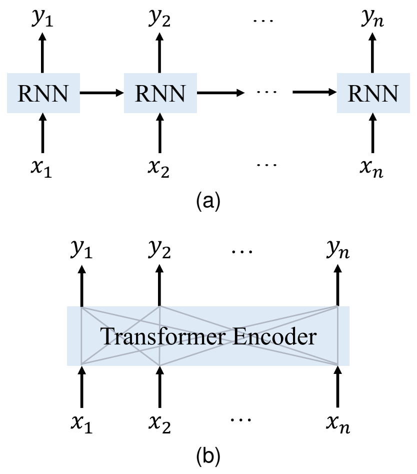


Fig. 2. A simple illustration of (a) RNN vs. (b) the Transformer encoder. $(x_1, x_2, \ldots, x_n)$ is the input sequence, $(y_1, y_2, \ldots, y_n)$ is the output sequence.

templates $\mathcal{C}$, that are most likely to appear after the input log substring under normal conditions. Here we assume $\mathcal{C} = \{T_3, T_4\}$, then in this case, because the actual next log message is L_4 (T_2) and $T_2 \notin \mathcal{C}$, we assert it to be anomalous. But unlike DeepLog or LogAnomaly, we do not identify the set of candidate log templates via a simple yet coarse-grained 'top-k' algorithm, which selects the most possible k (*e.g.*, 5) log templates all the time. We design a more flexible and robust algorithm, which selects a minimal set of log templates, whose probabilities are larger than the others, and the cumulative probability is large enough, *i.e.*, exceeds a threshold (*e.g.*, 0.9). We call this algorithm 'top-p', where p stands for "probability".

To summarize, this paper makes the following contributions:

- We propose a novel semi-supervised log anomaly detection approach, Loader, which leverages the Transformer encoder instead of the commonly used RNN and its variants, to detect anomalies in logs.
- We design a more flexible and robust algorithm, 'top-p', to determine the set of candidate log templates, that are most likely to appear after the input log substring under normal conditions, instead of selecting a fixed number of templates.
- Extensive experiments are conducted based on three public log datasets to validate our approach. The results show that, our approach Loader is better than or competitive with other unsupervised and semi-supervised log anomaly detection methods.

The rest of this paper is organized as follows. In Section 2, we introduce the related work, including log parsing and log anomaly detection. Section 3 covers our approach Loader, including the overall workflow and design details. The experimental settings, results, and analyses are given in Section 4. Finally, we conclude our paper in Section 5.

2 Related Work

2.1 Log Parsing

Log parsing is the basis of various automated log analysis tasks, including log anomaly detection, failure diagnosis, failure prediction, *etc.*, and has attracted a lot of interest.

Since logs are printed by logging statements in the source code, there is an attempt to parsing logs via source code analysis [7]. It is not trivial to analyze the source code of a large program. What is worse, a great number of programs are not open-source. These would hinder the application of such log parsing methods. Thus, data-driven log parsing methods are predominant. The key idea behind this kind of methods is that, the more frequent a part (*e.g.*, a word, a token, a substring, or an n-gram) appears in historical logs, the more likely it is to be a constant part. The main techniques adopted include frequent pattern mining [24], [25], [26], clustering [27], [28], longest common subsequence [29], [30], iterative partitioning [31], [32], and parse tree [33], [34]. In [35], the authors presented a comprehensive study on log parsing, by implementing and evaluating 13 log parsers on 16 benchmark datasets.

2.2 Log Anomaly Detection

As mentioned earlier, there are three types of log anomaly detection methods according to the data needed for training, *i.e.*, supervised, unsupervised, and semi-supervised methods.

2.2.1 *Supervised Methods*

In [9], the authors explored three supervised methods employing traditional machine learning techniques, *i.e.*, logistic regression, decision tree, and support vector machine (SVM). In [16], the authors proposed LogRobust, which adopts an attention-based Bidirectional Long Short-Term Memory (Bi-LSTM) network to detect anomalies in unstable log data. HitAnomaly [36] claims to be the first one that utilizes the Transformer model in log anomaly detection. It encodes log sequences and parameter values via a hierarchical Transformer structure. All these methods directly treat log anomaly detection as a binary classification task, and need labeled positive (anomalous) and negative (normal) samples (log sequences) to train the model.

2.2.2 *Unsupervised Methods*

In [7], the authors applied principal component analysis (PCA) to detecting anomalies in console logs. They create a message count vector from each log sequence, and utilize PCA to generate a normal subspace and an abnormal one from the matrix composed of these message count vectors. Then they treat those far away from the normal subspace as anomalies. Reference [17] tried to mine program invariants from log message groups, which reflect the inherent linear relationships in program workflows. If a log sequence violates any of the invariants, it is regarded as anomalous. This method also manipulates the matrix composed of the message count vectors, and utilizes singular value decomposition (SVD) to estimate the invariants. These two methods are also investigated in [9].

2.2.3 *Semi-supervised Methods*

DeepLog [11] is the first one to treat log anomaly detection as a sequence prediction problem, and the general ideas behind DeepLog and our approach Loader are similar. The main differences are that, DeepLog utilizes LSTM to capture normal execution patterns, and adopts the 'top-k' algorithm to determine the candidate set. Similar to DeepLog, LogAnomaly [12] also utilizes LSTM to learn normal patterns. But it leverages a template2Vec method to represent each log template with a vector which captures the semantic and syntax information of the log template, instead of an integer index widely used by prior works, including DeepLog. In addition, LogAnomaly also uses LSTM to capture the quantitative relationships of logs. LogBERT [37] borrows ideas from BERT [21], and leverages the Transformer encoder instead of RNN to capture patterns from normal log sequences, to resolve the drawbacks of RNN-based models. Unlike DeepLog, LogAnomaly, or our approach Loader, which try to predict the next log message of the input log substring under normal conditions, LogBERT randomly masks some log messages within the whole log sequence, and tries to predict them using the rest log messages. PLELog [19] is another semi-supervised method, but it is quite different from the aforementioned three methods as well as our approach Loader. It still treats log anomaly detection as a binary classification problem, so it needs both positive (anomalous) and negative (normal) samples (log sequences) for training. But it only requires partial normal samples to be labeled, and estimates the labels of the rest through clustering. If a cluster contains a known normal log sequence, then all log sequences in this cluster would be labeled as normal. Log sequences in other clusters are labeled as anomalous. These labeled samples are then used to train an attention-based GRU network for anomaly detection. In [38], a clustering-based method named LogCluster is proposed to identify problems in online service systems. It first clusters normal log sequences which come from lab environment, and chooses the centroid of each cluster as a representative log sequence. Then, for an incoming log sequence, LogCluster identifies its state (normal or anomalous) by computing its distances to all representative log sequences. If the minimum distance exceeds a threshold, then this log sequence would be reported as anomalous.

3 Our Approach

3.1 Overview

Fig. 3 shows the overall workflow of training Loader and using it to detect anomalies in logs. First of all, we need to parse raw log messages and separate them into log sequences. Then, for each log sequence, we apply a sliding window with step size 1 to it for substring processing. At each sliding step, with the substring of log messages falling into the sliding window (*i.e.*, the input log substring), we first encode them through an embedding layer to get their embeddings, and then add their embeddings with the positional encodings to get their input representations; afterwards, we pass these input representations through the Transformer encoder, to get a new output representation for each log message; finally, the mean of these output representations is used to predict a set of log templates, that may appear after the input log substring under normal conditions. The prediction is accomplished by a linear layer followed by a softmax layer, and the output is a probability distribution over all log templates, which indicates how likely each template would appear after the input log substring. When training, only normal log sequences are used, and the actual log message following the input log substring is used as label to calculate a cross-entropy loss with the output probability distribution, which is then used for backpropagation to optimize the model. When detecting, we choose a set of candidate log templates according to their probabilities given by the output probability distribution, and check if the actual next log message of the input log substring is within the candidate set or not, if yes, we treat it as normal, otherwise we deem it as anomalous. In the end, a log sequence would be reported as anomalous if at least one log message in it is detected as anomalous.

Next, we address the key details throughout the workflow, including critical steps or components in data preprocessing and the Loader model.

3.2 Data Preprocessing

In this section, we introduce how to preprocess raw log messages into separate log sequences, including two steps: log parsing (Section 3.2.1) and separation (Section 3.2.2).

3.2.1 Log Parsing

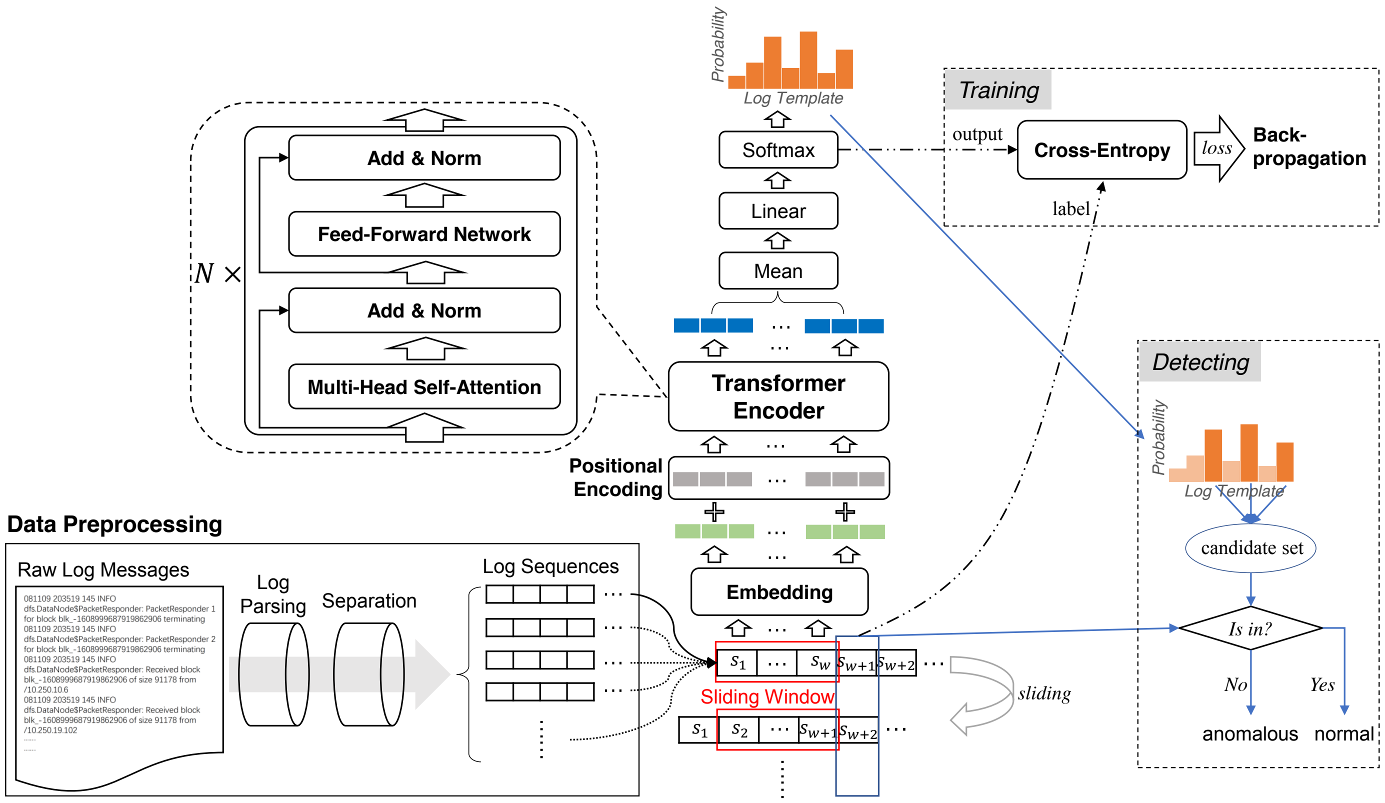
The goal of log parsing is to transform raw textual logs into a structured format, as illustrated in Fig. 1. It is common that each log message begins with several auxiliary fields, such as the "Timestamp", "Level", and "Component" columns in Fig. 1, followed by the content field. As the auxiliary fields often have fixed formats and are easy to extract, the core of log parsing is to distinguish the constant parts from the variable parts in the content field, and get a log template together with optional parameters for each log message. The log template is composed of the constant parts, while the variable parts corresponding to parameters are replaced by some wildcards (*e.g.*, * in the "Template" column in Fig. 1).

A log template actually refers to the invariant part of the string parameter of a log printing statement in the source code, so the number of log templates is finite for a specific service/system. Consequently, we can map the possibly endless log messages generated over time to a restricted set of templates, which can facilitate our model to learn latent patterns from log sequences.

3.2.2 Separation

Logs produced by different tasks or processes, within a service/system or across different services/systems, are usually mixed up, so we need to separate them into different log sequences for better anomaly detection. Some logs have certain kind of identifiers recorded to ease this separation. For example, in the HDFS log dataset used in our experiments which we will detail later, each log message has a *block_id* indicating the block operated, so we can easily separate these log messages according to this identifier.

On the other hand, it is a common practice to detect anomalies in logs, which have a specified number of log messages or are generated over a specified time period. This also require us to separate a large set of logs into different log sequences.



```
081109 203519 145 INFO
dfs.DataNode$PacketResponder: PacketResponder 1
for block blk_-1608999687919862906 terminating
081109 203519 145 INFO
dfs.DataNode$PacketResponder: PacketResponder 2
for block blk_-1608999687919862906 terminating
081109 203519 145 INFO
dfs.DataNode$PacketResponder: Received block
blk_-1608999687919862906 of size 91178 from
/10.250.10.6
081109 203519 145 INFO
dfs.DataNode$PacketResponder: Received block
blk_-1608999687919862906 of size 91178 from
/10.250.19.102
......
......
```

Fig. 3. Overall workflow of Loader.

3.3 Loader

In this section, we detail how Loader computes the output probability distribution from the input log substring (Section 3.3.1 ∼ Section 3.3.3), how to train Loader (Section 3.3.4), and how to make a judgment according to the output probability distribution (Section 3.3.5).

3.3.1 *Input Encoding*

The input to Loader at each sliding step is a substring of log messages falling into the sliding window, *i.e.*, the input log substring. Assume the window size is w and the step size is 1, then for a log sequence $S = [s_1, s_2, s_3, \ldots]$, the input log substring for each sliding step would be:

$$\begin{aligned} &[s_1, s_2, \ldots, s_w], \\ &[s_2, s_3, \ldots, s_{w+1}], \\ &\qquad\vdots \\ &[s_i, s_{i+1}, \ldots, s_{w+i-1}], \\ &\qquad\vdots \end{aligned}$$

If we define the set of all log templates as $\mathcal{T} = \{T_1, T_2, \ldots, T_t\}$ (t is the total number of log templates), then for any $i \in \mathbb{N}^+$, $s_i \in \mathcal{T}$. In the embedding layer, an embedding matrix $M \in \mathbb{R}^{t\times d}$ (d is the embedding size) is used to encode all these templates, whose ith row is the embedding of T_i ($1 \leqslant i \leqslant t$), and is denoted as $\boldsymbol{e}_i$ ($\boldsymbol{e}_i \in \mathbb{R}^d$). Then we can represent each log message s_i with an embedding $\boldsymbol{e}_j$ if $s_i = T_j$. The embedding matrix M is composed of the template vectors, which are output by our log parsing method LPV [28], and encode the semantic and syntactic information of log templates.

Since the Transformer encoder has no recurrence or convolution structure, the order information of the input log substring is injected through positional encoding. The positional encodings have the same size d as the embeddings from the embedding layer, and are calculated as follows (the same as in [20]):

$$\begin{aligned} PE_{(p,2i)} &= \sin(p/10000^{2i/d}), \\ PE_{(p,2i+1)} &= \cos(p/10000^{2i/d}), \end{aligned} \tag{1}$$

where p is the position index in the input log substring ($0 \leqslant p < w$), $2i$ and $2i+1$ are even and odd indexes of the encodings' dimensions respectively ($0 \leqslant 2i,\ 2i+1 < d$).

Afterwards, for the log messages in the input log substring, we add their embeddings from the embedding layer with the positional encodings in a position-wise manner, to get their input representations, which will be fed to the Transformer encoder. For the sake of efficient computing, these input representations are packed into a matrix $X_{in} \in \mathbb{R}^{w\times d}$.

3.3.2 *The Transformer Encoder*

Loader leverages the Transformer encoder to capture normal patterns in normal log sequences. The Transformer encoder consists of $N(N \geqslant 1)$ identical layers, each of which has two sub-layers, *i.e.*, a multi-head self-attention mechanism and a fully connected feed-forward network.

The multi-head self-attention is to apply multiple self-attention functions to its input in parallel, each of which is called a head, then concatenate all heads' outputs and project to the same d-dimensional space as the input. Suppose there are h heads and the input to this sub-layer is

$X_{atn} \in \mathbb{R}^{w\times d}$, then for the ith ($1 \leqslant i \leqslant h$) head, its output $head_i$ is calculated as follows:

$$head_i = \mathrm{Attention}(X_{atn}W_i^Q,\ X_{atn}W_i^K,\ X_{atn}W_i^V), \quad (2)$$

where $W_i^Q \in \mathbb{R}^{d\times d_q}$, $W_i^K \in \mathbb{R}^{d\times d_k}$, $W_i^V \in \mathbb{R}^{d\times d_v}$ are projection matrices which are trainable, and $d_q = d_k = d_v = d/h$; $\mathrm{Attention}(\cdot)$ is the scaled dot-product attention introduced in [20], which is defined as follows:

$$\mathrm{Attention}(Q, K, V) = \mathrm{Softmax}(\frac{QK^{\mathrm{T}}}{\sqrt{d_k}})V, \quad (3)$$

where d_k is the size of the row vectors of Q and K, the $\mathrm{Softmax}(\cdot)$ function is used to calculate the weights for the row vectors of V and is defined as:

$$\mathrm{Softmax}(\boldsymbol{z}) = \frac{e^{\boldsymbol{z}}}{\sum_i e^{\boldsymbol{z}_i}}, \quad (4)$$

where $\boldsymbol{z}$ is a row vector. This ensures that the weights calculated from each row vector add up to 1. It can be seen from the above that, each row of X_{atn} attends to all of its rows, including the row itself. The final output of the multi-head self-attention is obtained as follows:

$$\mathrm{MultiHead_h}(X_{atn}) = \mathrm{Concat}([head_1, \dots, head_h])W^o, \quad (5)$$

where $\mathrm{Concat}(\cdot)$ concatenates all $head_i$'s ($1 \leqslant i \leqslant h$) horizontally to get a matrix $H \in \mathbb{R}^{w\times hd_v}$, and $W^o \in \mathbb{R}^{hd_v\times d}$ is another projection matrix which is also trainable.

The fully connected feed-forward network consists of two linear transformations, between which there is a ReLU activation. It is applied to each position separately and identically. Suppose the input to this sub-layer is $X_{ffn} \in \mathbb{R}^{w\times d}$, then it can be defined as follows:

$$\mathrm{FFN}(X_{ffn}) = \mathrm{ReLU}(X_{ffn}W_1 + \boldsymbol{b}_1)\cdot W_2 + \boldsymbol{b}_2, \quad (6)$$

where $W_1 \in \mathbb{R}^{d\times d_{ff}}$ and $W_2 \in \mathbb{R}^{d_{ff}\times d}$ are weight matrices, $\boldsymbol{b}_1 \in \mathbb{R}^{d_{ff}}$ and $\boldsymbol{b}_2 \in \mathbb{R}^{d}$ are biases, for the two linear transformations respectively, and all are trainable; d_{ff} is a hyperparameter which defines the dimension of the inner layer; $\mathrm{ReLU}(\cdot)$ is a commonly used activation function defined as:

$$\mathrm{ReLU}(x) = \begin{cases} x, & x > 0; \\ 0, & x \leqslant 0. \end{cases} \quad (7)$$

In addition, around each sub-layer (*i.e.*, multi-head self-attention or feed-forward network), there is a residual connection [39] followed by layer normalization [40]. So the output of each sub-layer is $\mathrm{LayerNorm}(X + \mathrm{SubLayer}(X))$, where X is the input to the sub-layer, $\mathrm{SubLayer}(\cdot)$ is either $\mathrm{MultiHead_h}(\cdot)$ or $\mathrm{FFN}(\cdot)$ defined above, and $\mathrm{LayerNorm}(\cdot)$ is the layer normalization function.

3.3.3 Output Decoding

The output of the Transformer encoder is a matrix $X_{out} \in \mathbb{R}^{w\times d}$, each row of which is the output representation of a log message in the input log substring. Since each log message has attended to all log messages in the input log substring (including itself), as mentioned earlier, we believe its output representation has encoded the context information of the whole input log substring. To ensure no bias, we use the mean of these output representations, *i.e.*, the mean row vector of X_{out} (denoted as $\mathrm{mean}(X_{out}, 0) \in \mathbb{R}^d$), and utilize a linear layer with a softmax followed to produce a probability distribution over all log templates. The output probability distribution gives the probability of each log template to appear after the input log substring, under normal conditions.

Formally, the whole process of output decoding can be described as follows:

$$prob\_dist = \mathrm{Softmax}(\mathrm{mean}(X_{out}, 0)\cdot W + \boldsymbol{b}), \quad (8)$$

where $W \in \mathbb{R}^{d\times t}$ and $\boldsymbol{b} \in \mathbb{R}^t$ are trainable weight matrix and bias respectively, t is the total number of log templates, $prob\_dist \in \mathbb{R}^t$ and $\mathrm{sum}(prob\_dist) = 1$.

3.3.4 Training

When training, we use the actual log message following the input log substring as label, and compute the cross-entropy loss between its one-hot representation and $prob\_dist$ from output decoding. The one-hot representation of the ith template T_i is a sparse vector with length t, in which only the ith element is 1 and the others are 0's (t is the total number of log templates). Assume the label is s_{w+i}, then the loss $\mathcal{L}$ is computed as follows:

$$\mathcal{L} = \mathrm{CrossEntropy}(\mathrm{one\_hot}(s_{w+i}),\ prob\_dist), \quad (9)$$

where $\mathrm{one\_hot}(\cdot)$ stands for the one-hot representation, and $\mathrm{CrossEntropy}(\cdot)$ is the function to compute the cross-entropy loss, which is defined as follows:

$$\mathrm{CrossEntropy}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_i \mathbf{y}_i \log \hat{\mathbf{y}}_i. \quad (10)$$

The loss is then used for backpropagation to update the trainable parameters of Loader.

3.3.5 Detecting

On detecting, we first identify a set of candidate log templates $\mathcal{C}$ according to $prob\_dist$ from output decoding, then check whether the actual next log message s_{next} of the input log substring is within the candidate set or not. If $s_{next} \in \mathcal{C}$, we treat it as normal, otherwise we treat it as anomalous.

Algorithm 1: The 'top-k' Algorithm

```
Input: $prob\_dist$,
       number of candidate log templates $k$,
       set of all log templates $\mathcal{T} = \{T_1, T_2, \dots, T_t\}$
Output: set of candidate log templates $\mathcal{C}$
1 initialize $\mathcal{C}$ to an empty set
2 for $j = 1$ to $k$ do
3     $i \leftarrow$ index of the maximum value in $prob\_dist$
4     add $T_{i+1}$ to $\mathcal{C}$
5     delete $T_{i+1}$ from $\mathcal{T}$
6     delete $prob\_dist[i]$
7 end
8 return $\mathcal{C}$
```

Previous similar works, such as DeepLog [11] and LogAnomaly [12], simply select a fixed number of candidate log templates with higher probabilities, all the time. We call it the 'top-k' algorithm, and describe the process in Algorithm 1. However, we observe that the number of distinct log templates, which can appear after a given log

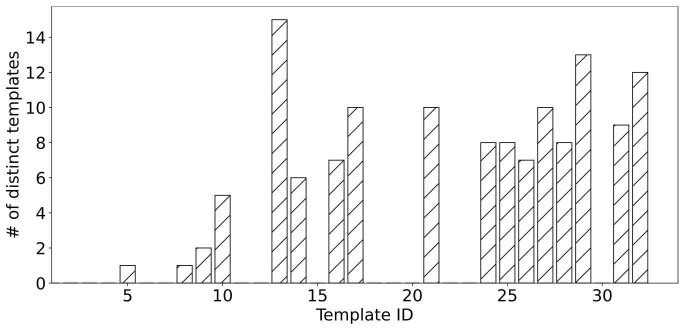


Fig. 4. Numbers of distinct log templates appearing after each log template, in the training set of the HDFS log dataset.

template under normal conditions, often differs for different log templates. This can be verified by Fig. 4, where the horizontal axis represents different log templates, and the vertical axis is the number of distinct log templates, which appear after a specific log template under normal conditions. The numbers used to draw Fig. 4 are obtained based on the training set (only containing normal log sequences) of the HDFS log dataset used in our experiments which we will detail later.

Algorithm 2: The 'top-p' Algorithm

```
Input: $prob\_dist$,
       probability threshold $\lambda_p$,
       set of all log templates $\mathcal{T} = \{T_1, T_2, \ldots, T_t\}$
Output: set of candidate log templates $\mathcal{C}$
1  initialize $\mathcal{C}$ to an empty set
2  initialize cumulative probability $cum\_prob$ to 0
3  while $cum\_prob < \lambda_p$ do
4      $i \leftarrow$ index of the maximum value in $prob\_dist$
5      add $T_{i+1}$ to $\mathcal{C}$
6      delete $T_{i+1}$ from $\mathcal{T}$
7      $cum\_prob \leftarrow cum\_prob + prob\_dist[i]$
8      delete $prob\_dist[i]$
9  end
10 return $\mathcal{C}$
```

In this paper, we propose a more flexible and robust way to select the candidate log templates, and name it the 'top-p' algorithm. As described in Algorithm 2, the number of candidate log templates selected is determined by a probability threshold λ_p $(0 < \lambda_p < 1)$ rather than fixed. Specifically, we iteratively pick out a log template with the maximum probability at each iteration, until the cumulative probability of the selected log templates is no less than λ_p. So the number of candidate log templates may vary with different $prob\_dist$'s. We will validate the effectiveness of this algorithm in our experiments.

4 Evaluation

In this section, we first give the experimental settings, including datasets, baselines, evaluation metrics, implementation details, and experimental environment (Section 4.1). Then, we present the effectiveness comparison between Loader and all baseline methods (Section 4.2). Next, we compare the efficiency of Loader with that of the RNN-based baseline methods (Section 4.3). Subsequently, we confirm the effectiveness of the 'top-p' algorithm vs. the 'top-k' algorithm (Section 4.4). After that, we investigate the influence of each model parameter, on the effectiveness of Loader (Section 4.5). Finally, we give a simple illustration to show how Loader detects anomalies (Section 4.6).

TABLE 1
Some Statistics about the Datasets

Dataset	# of log messages	# of anomalies	# of templates*
HDFS	11,175,629	16,838	32
BGL	4,747,963	348,460	424
Thunderbird	10,000,000	353,794	1,255

* The number of log templates returned by our log parsing method LPV [28].

4.1 Experimental Settings

4.1.1 Datasets

We evaluate our approach Loader based on three public log datasets, *i.e.*, the HDFS log dataset [7], the Blue Gene/L log dataset [41], and the Thunderbird log dataset [41] (denoted by *HDFS*, *BGL*, and *Thunderbird* respectively)[1].

► *HDFS*: This dataset is a set of HDFS (Hadoop Distributed File System) logs generated by running sample Hadoop map-reduce jobs on more than 200 Amazon EC2 nodes. There are 11,175,629 log messages in total. A *block_id* is recorded in each log message to indicate a block operation, according to which we can separate the log messages into different log sequences. Each log sequence reflects the execution path of a block. Every block was manually labeled as normal or anomalous by Hadoop experts, and among the total 575,061 blocks, 16,838 blocks are anomalous.

► *BGL*: This dataset was collected from the Blue Gene/L supercomputer deployed at Lawrence Livermore National Labs (LLNL). It contains 4,747,963 log messages, among which 348,460 were tagged as alerts by the system experts. Unlike *HDFS*, there are no identifiers recorded in the log messages which can be used to separate them. Following previous works [9], [12], we utilize a sliding time window to slice these log messages into log sequences, according to their timestamps. If there exists one or more alerts in a log sequence, then it would be labeled as anomalous, otherwise as normal.

► *Thunderbird*: This dataset was collected from Thunderbird, a supercomputer installed at Sandia National Labs (SNL). It has more than 200 million log messages. We use the first 10 million for evaluation, because one of the baseline methods cannot finish in reasonable time with more log messages, in our experimental environment. The log format of this dataset is similar to that of *BGL*, and alerts were also tagged by the system experts. Among the first 10 million log messages, 353,794 are alerts. We utilize a sliding time window to slice the log messages into log sequences, just as what we do with *BGL*.

Some statistics about these three datasets are given in Table 1. Following previous works [9], [12], for each dataset,

1. Available at https://github.com/logpai/loghub

TABLE 2
Parameter Settings of Loader

	Parameter	Description	Value
Model Parameters	w	sliding window size	10
	N	number of layers in the Transformer encoder	2
	h	number of heads in multi-head self-attention	2
	d_{ff}	inner layer dimension of feed-forward network	128
Training Parameters	$epochs$	training epochs	20
	$batch\_size$	training batch size	2048
	lr	learning rate	0.001
Detecting Parameters	k	number of candidates in the 'top-k' algorithm	1, 2, 3, $\cdots$, 10 (*HDFS*) 1, 2, 3, $\cdots$, 10, 20, 30, 40, 50 (*BGL*, *Thunderbird*)
	λ_p	probability threshold in the 'top-p' algorithm	0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, **0.95**, 0.99, 0.999 (*HDFS*) 0.9, 0.99, 0.999, 0.9999, 0.99999, **0.999999** (*BGL*, *Thunderbird*)

we use the front 80% normal log sequences for training, and the rest for testing. Because in practice, we can only learn from historical logs produced earlier.

4.1.2 Baselines

As supervised log anomaly detection methods need both normal and anomalous log sequences together with their labels for training, for fair comparison, we choose the following six unsupervised and semi-supervised log anomaly detection methods as baselines.

▶ PCA [7]: This approach builds a message count vector from each log sequence, and applies principal component analysis (PCA) on the matrix composed of these message count vectors, to construct a normal subspace and an abnormal one. Then those far away from the normal subspace are reported as anomalies.

▶ Invariants Mining (IM) [17]: Like PCA, this approach also operates on the matrix composed of the message count vectors. It employs singular value decomposition (SVD) to mine invariants which can be satisfied by most of the vectors, and then treats those violating one or more invariants as anomalies.

▶ LogCluster [38]: This approach clusters normal log sequences into different clusters, and then treats those too far away from these clusters as anomalies. In this approach, each log sequence is turned into a vector according to the distinct log templates it contains.

▶ DeepLog [11]: DeepLog employs an LSTM model to learn normal patterns from normal log sequences. It then tries to predict the next log message of each input log substring. If the actual next log message is among the top k candidates, then it's treated as normal, otherwise as anomalous. It works in a streaming manner with a sliding window, just as Loader.

▶ LogAnomaly [12]: Similar to DeepLog, LogAnomaly also tries to learn normal patterns from normal log sequences and predict the next log message of each input log substring. But it employs two LSTM models, one for sequential patterns and the other one for quantitative patterns.

▶ LogBERT [37]: Like Loader, LogBERT also leverages the Transformer encoder to model normal log sequences. But it does not try to predict the next log message of each log substring, and is not designed to work in a streaming manner. It randomly masks some log messages in the log sequence, and then tries to predict these masked ones.

Among the above six methods, the first two (*i.e.*, PCA and IM) are unsupervised methods, and the other four are semi-supervised. More details about these baselines are given in Section 2.

4.1.3 Evaluation Metrics

Log anomaly detection is essentially a binary classification task (anomalous or normal). Following previous works [9], [11], [12], [37], we use three commonly used metrics (*i.e.*, $Precision$, $Recall$, and $F\text{-}measure$) to measure the effectiveness of Loader and the baseline methods. $Precision = \frac{TP}{TP+FP}$, $Recall = \frac{TP}{TP+FN}$, and $F\text{-}measure = \frac{2\times Precision\times Recall}{Precision+Recall}$, where TP stands for the number of log sequences which are judged as anomalous and labeled as anomalous as well, FP is the number of log sequences which are judged as anomalous but labeled as normal, and FN refers to the number of log sequences which are judged as normal but labeled as anomalous.

4.1.4 Implementation Details

We implemented Loader with Python 3.9.7 and TensorFlow 2.7.0. By default, the Transformer encoder consists of two layers. In each layer, the multi-head self-attention has two heads, and the inner layer dimension of the feed-forward network is 128. The default sliding window size is 10. The Adam algorithm [42] with learning rate 0.001 is adopted to optimize the model. We implemented both the 'top-k' algorithm and the 'top-p' algorithm for detecting, in order for better comparison. The parameter settings of Loader are summarized in Table 2. For each detecting parameter, we give the value range we explored on each dataset, which is also applied to DeepLog and LogAnomaly for fair comparison, because these three methods share the same detecting strategy. The numbers in bold (0.95 and 0.999999) are the default values of the probability threshold for Loader on each dataset. As to log parsing, we employ LPV, a state-of-the-art log parsing method which has been published in one of our previous works [28]. With the help of LPV, we can get a template vector for each log template, which is used in the embedding layer, and the embedding size is 24.

For baselines PCA, IM, and LogCluster, we employed a popular implementation provided by the LogPAI team[2]. For baselines DeepLog and LogAnomaly, we adopted a popular

2. https://github.com/logpai/loglizer

TABLE 3
Effectiveness Comparison between Loader and the Baselines on *HDFS*, *BGL*, and *Thunderbird*

Method	*HDFS*			*BGL*			*Thunderbird*		
	Precision	*Recall*	*F-measure*	*Precision*	*Recall*	*F-measure*	*Precision*	*Recall*	*F-measure*
PCA	**0.999***	0.442	0.613	**0.964***	0.022	0.043	**0.985**	0.156	0.269
IM	**0.969**	**0.918**	**0.943**	**0.957**	0.292	0.448	0.749	0.100	0.177
LogCluster	0.123	0.546	0.200	**0.959**	0.251	0.398	**0.997***	**0.948**	**0.972**
LogBERT	**0.985**	0.683	0.807	0.857	0.803	0.829	**0.996**	**0.932**	**0.963**
DeepLog ('top-k')	**0.935**	**0.964**	**0.949**	0.846	**0.997***	**0.915**	0.836	**1.000***	**0.911**
DeepLog ('top-p')	**0.928**	**0.994**	**0.960**	0.850	**0.990**	**0.915**	**0.969**	**0.999**	**0.984**
LogAnomaly ('top-k')	**0.975**	**0.937**	**0.955**	0.848	**0.994**	**0.915**	0.843	**1.000***	**0.915**
LogAnomaly ('top-p')	**0.980**	**0.973**	**0.977**	0.848	**0.995**	**0.916**	**0.946**	**1.000***	**0.972**
Loader ('top-k')	**0.946**	**0.987**	**0.966**	0.871	**0.986**	**0.925**	0.833	**1.000***	**0.909**
Loader ('top-p')	**0.959**	**0.998***	**0.978***	**0.900**	**0.968**	**0.933***	**0.982**	**1.000***	**0.991***

open-source implementation[3], and plugged in our 'top-p' algorithm. LogBERT is shared online[4] by its authors, and we made use of it as is.

4.1.5 Experimental Environment

We conducted all experiments on a Linux server equipped with an Intel(R) Xeon(R) CPU E5-2678 v3 @ 2.50GHz, 192GB DDR4 memory, and 4 NVIDIA GeForce GTX 1080 Ti GPUs. The Linux distribution version is Ubuntu 16.04.7 LTS.

4.2 Effectiveness Comparison with Baselines

Table 3 shows the effectiveness comparison between Loader and all baseline methods on *HDFS*, *BGL*, and *Thunderbird* respectively, in which we put the numbers $\geqslant 0.9$ in bold, and the maximum number of each column is marked with *. For DeepLog, LogAnomaly, and Loader, we experimented with the 'top-k' algorithm and the 'top-p' algorithm respectively. And for either 'top-k' or 'top-p', we tested with each value in the value range given in Table 2 for the corresponding detecting parameter, and reported the best result (having the highest $F\text{-}measure$).

It can be seen from Table 3 that, our approach Loader achieves the highest $F\text{-}measure$ on each dataset, when adopting the 'top-p' algorithm. This validates the effectiveness and competitiveness of our approach.

DeepLog, LogAnomaly, and our approach Loader outperform the other four baselines (*i.e.*, PCA, IM, LogCluster, and LogBERT) in terms of $Recall$ on each dataset, no matter adopting 'top-k' or 'top-p'. We think the reason is as follows: DeepLog, LogAnomaly, and Loader all try to predict the next log message of the input log substring in current sliding window, under normal conditions. The sliding window moves forward step by step, to process every log message of a log sequence (except the first w ones). If any log message is not within the corresponding candidate set, then the log sequence would be flagged as anomalous. So DeepLog, LogAnomaly, and Loader are likely to detect more anomalies, which brings higher $Recall$. But, as a side effect, this may cause more false positives and degrades the $Precision$, which is verified by the results in Table 3. Specifically, DeepLog, LogAnomaly, and Loader do not beat all the other four baselines in terms of $Precision$, on any dataset. Nevertheless, these three methods perform better than, or equally with (only for LogAnomaly on *Thunderbird*) the other four baselines in terms of $F\text{-}measure$ on each dataset, when adopting the 'top-p' algorithm. When the 'top-k' algorithm is adopted, they still have better $F\text{-}measure$ on two datasets (*i.e.*, *HDFS* and *BGL*). The above confirms the effectiveness and superiority of the detecting strategy behind these three methods and our 'top-p' algorithm.

3. https://github.com/donglee-afar/logdeep
4. https://github.com/HelenGuohx/logbert

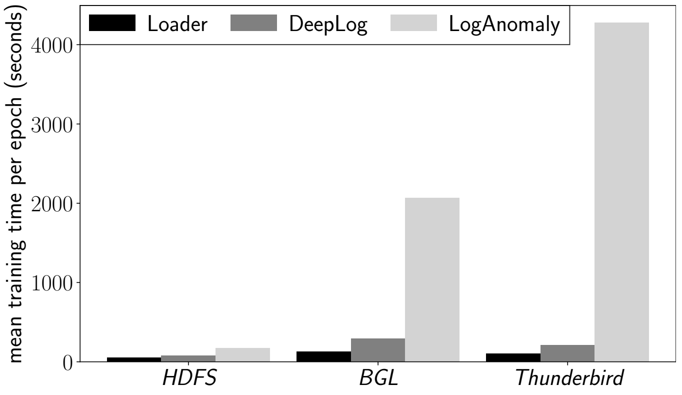


Fig. 5. Mean training time per epoch of Loader, DeepLog, and LogAnomaly, on *HDFS*, *BGL*, and *Thunderbird*.

DeepLog, LogAnomaly, and our approach Loader have competitive effectiveness with each other, but Loader has better $F\text{-}measure$ than the other two on each dataset, when adopting the 'top-p' algorithm. When 'top-k' is adopted, Loader has better $F\text{-}measure$ than the other two on *HDFS* and *BGL*, and is a little lower than them on *Thunderbird*. On the other hand, LogAnomaly has better $Precision$ than DeepLog and Loader on *HDFS*, both DeepLog and LogAnomaly have better $Recall$ than Loader on *BGL*, no matter whether 'top-k' or 'top-p' is adopted. These indicate that, DeepLog, LogAnomaly, and Loader possess different advantages and capabilities, on different datasets.

4.3 Efficiency Comparison with RNN-based Methods

To demonstrate the efficiency superiority of our approach Loader over RNN-based methods, we compare the mean training time per epoch of Loader with that of DeepLog and LogAnomaly, on each dataset. The results are shown in Fig.

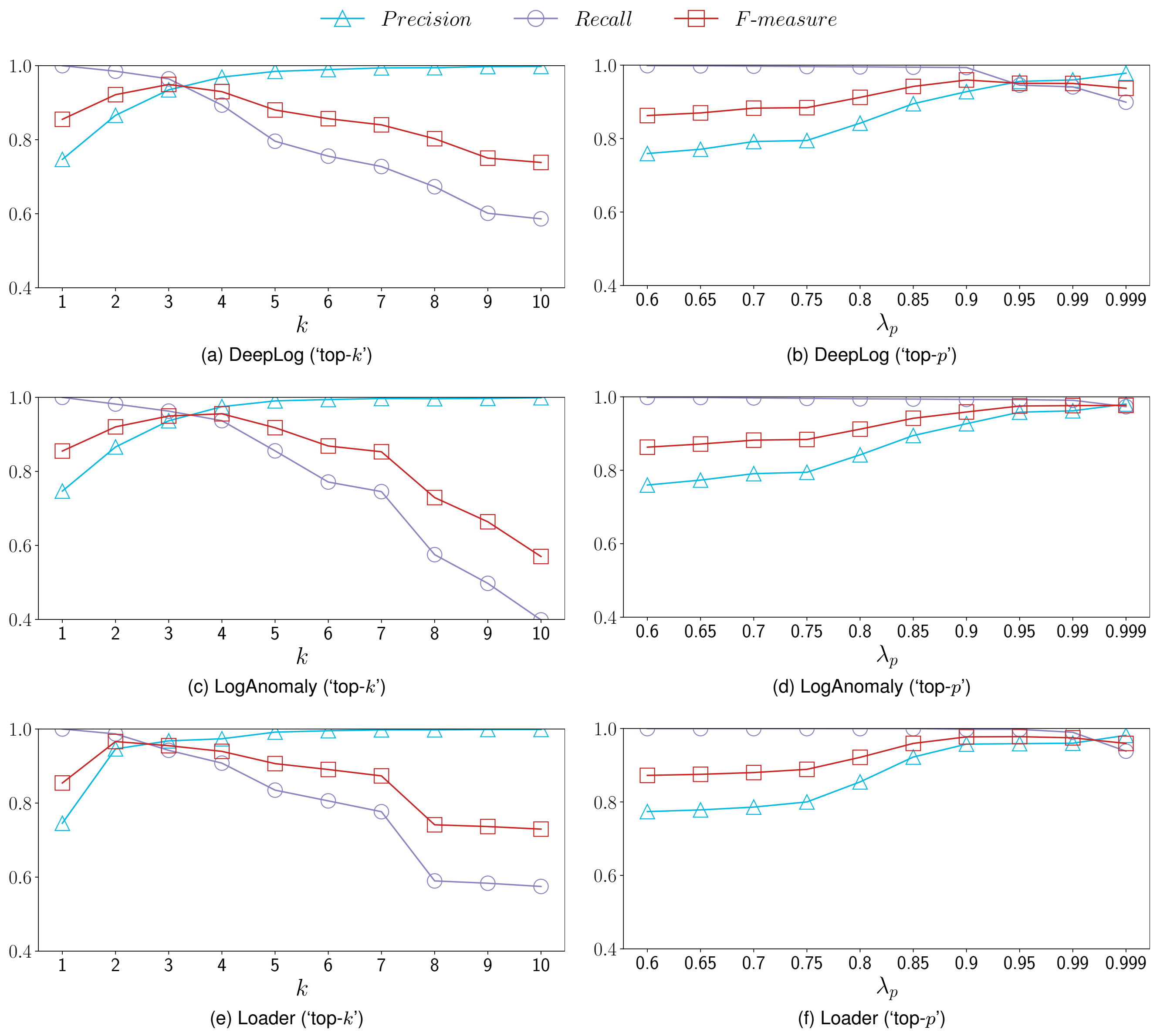


Fig. 6. Effectiveness variations of DeepLog, LogAnomaly, and Loader on *HDFS*, when varying the value of the detecting parameter of the 'top-k' algorithm and the 'top-p' algorithm respectively.

5, from which we can see that, Loader needs much less time to finish a training epoch than DeepLog and LogAnomaly, on each of the three log datasets. This benefits from the Transformer encoder utilized by Loader, which has no recurrence and is more parallelizable than RNN and its variants. On the other hand, as LogAnomaly employs two LSTM models, one for sequential patterns and the other for quantitative patterns, so it is reasonable that LogAnomaly needs more training time than DeepLog.

4.4 Effectiveness of 'top-p' vs. 'top-k'

From Table 3, we can see that, DeepLog, LogAnomaly, and Loader may achieve better or equal $F\text{-}measure$, if adopting the 'top-p' algorithm rather than the 'top-k' algorithm. To further demonstrate the superiority of our proposed 'top-p' algorithm, we plot the effectiveness variations of these three methods adopting 'top-k' and 'top-p' respectively, when changing the value of the corresponding detecting parameter, *i.e.*, k of 'top-k' and λ_p of 'top-p'. The results are given in Fig. 6.

It can be observed from Fig. 6 that, for any of the three methods (DeepLog, LogAnomaly, and Loader), when adopting the 'top-k' algorithm, with k increases from 1 to 10, the $Precision$ increases rapidly at the beginning and then keeps approaching 1.0 when $k \geqslant 7$, while the $Recall$ keeps dropping, causing the $F\text{-}measure$ to first increase and then decrease. On the other hand, when adopting the 'top-p' algorithm, with λ_p increases, the $Precision$ grows slowly to approximate to 1.0, while the $Recall$ drops slightly, leading the $F\text{-}measure$ to increase slowly.

Overall, when adopting the 'top-k' algorithm, the effectiveness is sensitive to parameter k, and is quite different when k takes different values. When adopting the 'top-p' algorithm, however, the effectiveness does not change much when varying λ_p, which makes it easier to choose a proper value for λ_p to achieve sound effectiveness. Thus, we can

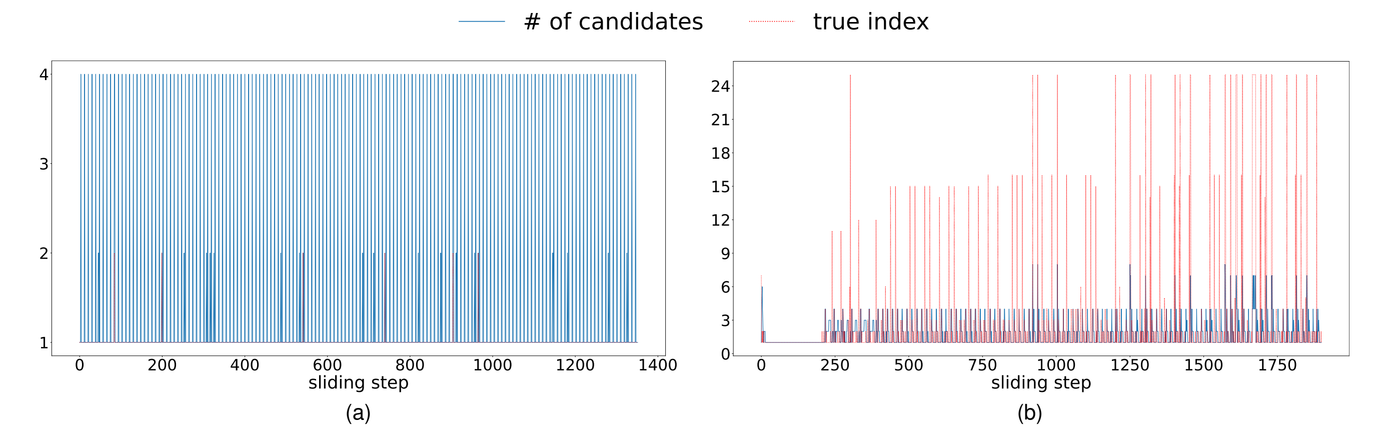


Fig. 7. The number of candidate log templates determined by the 'top-p' algorithm (`# of candidates`) vs. the position index of the actual next log message in the ordered log template list which is sorted in descending order according to the output probability distribution (`true index`), in each sliding step. (a) Testing with the first 150 normal log sequences from the testing set of *HDFS*. (b) Testing with the first 150 anomalous log sequences from the testing set of *HDFS*.

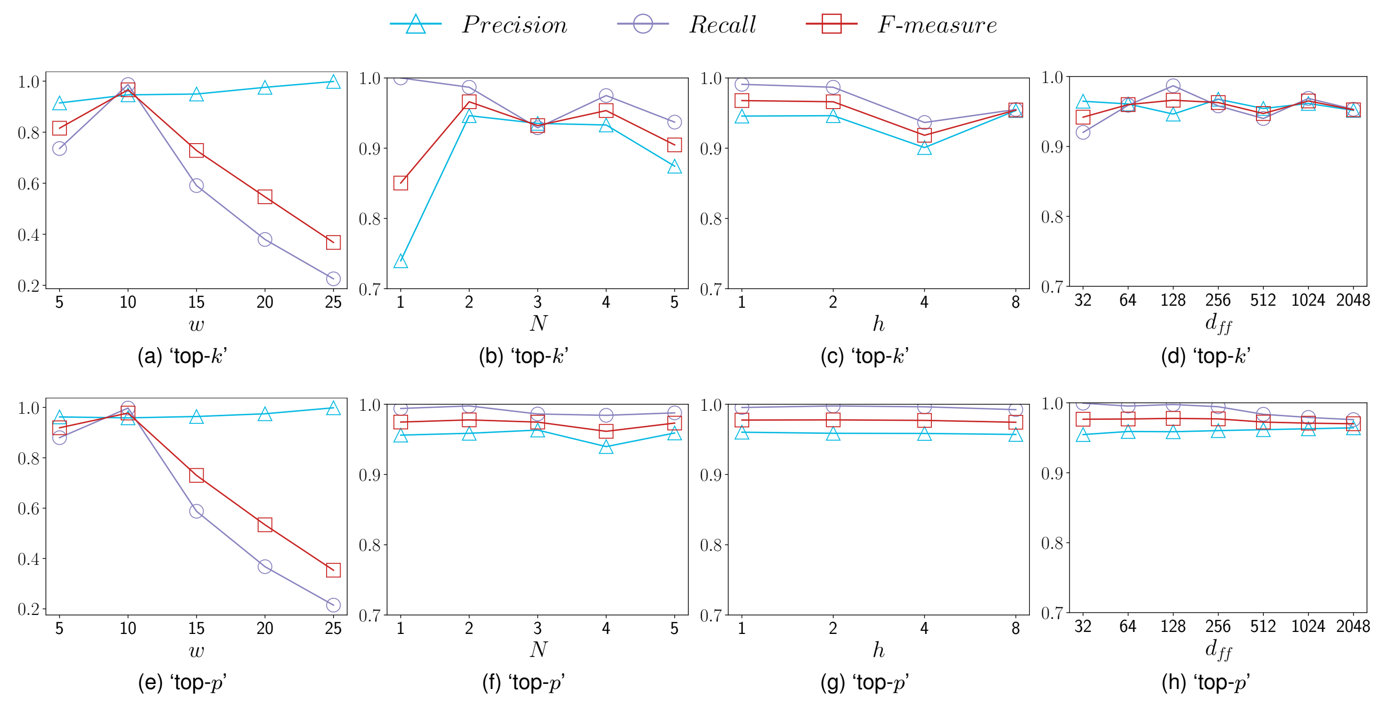


Fig. 8. Effectiveness variations of Loader on *HDFS*, when varying the value of each model parameter.

say that our proposed 'top-p' algorithm is more flexible and robust than the 'top-k' algorithm.

The reason why the 'top-p' algorithm performs better is that, unlike the 'top-k' algorithm, which selects a fixed number (*e.g.*, 5) of candidate log templates all the time, 'top-p' may select different numbers of candidate log templates, when encountering different input log substrings. To illustrate this, we tested Loader with the first 150 normal and anomalous log sequences from the testing set of *HDFS* respectively. In each sliding step, we counted the number of candidate log templates determined by 'top-p', and recorded the position index of the actual next log message in the ordered log template list, which is sorted in descending order according to the output probability distribution. The results are shown in Fig. 7, from which we can see that, the number of candidate log templates selected by the 'top-p' algorithm changes frequently at different sliding steps. This feature may enable Loader to detect more anomalies, and avoid producing too many false positives at the same time.

4.5 Influences of Different Model Parameters

In this section, we explore how each of the model parameters (*i.e.*, w, N, h, and d_{ff} in Table 2) affects the effectiveness of Loader, through a series of experiments on *HDFS*.

We varied the value of one model parameter at a time, while keeping the other model parameters with their values listed in Table 2. The training parameters were kept with their values in Table 2 all the time. For the detecting parameters, we tested with each value within the value range given in Table 2 and reported the best result (having the highest F-$measure$), for the 'top-k' algorithm and the 'top-p' algorithm respectively, to better measure the influence of each model parameter. The effectiveness variations when

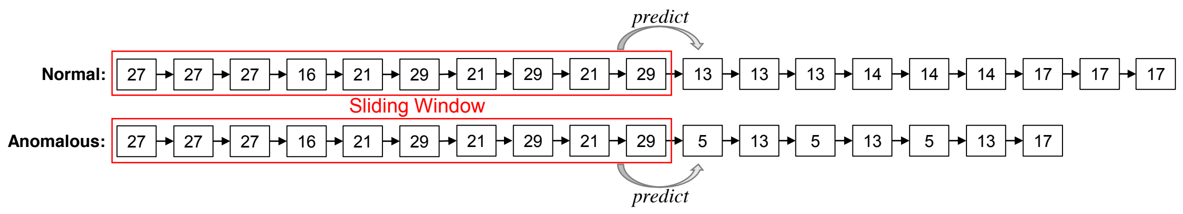

predict
Normal: 27 27 27 16 21 29 21 29 21 29 13 13 13 14 14 14 17 17 17
Sliding Window
Anomalous: 27 27 27 16 21 29 21 29 21 29 5 13 5 13 5 13 17
predict

Fig. 9. A normal log sequence and an anomalous one from the testing set of *HDFS*. The first ten log messages of them are identical. The sliding window size $w = 10$.

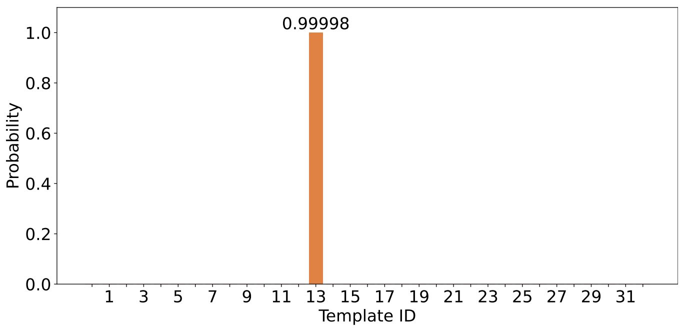


Fig. 10. The output probability distribution over all log templates, when feeding Loader with the first ten log messages of the log sequences in Fig. 9.

varying different model parameters are given in Fig. 8, where the subfigures in the first row (Figs. 8a $\sim$ 8d) are about Loader adopting the 'top-k' algorithm, and the second row (Figs. 8e $\sim$ 8h) about 'top-p'. Each column of Fig. 8 corresponds to a model parameter. Note that the two subfigures in the first column (corresponding to the model parameter w) have a different value range in the y-axis from the others.

It can be seen from Fig. 8 that, Loader is particularly sensitive to the sliding window size (w), no matter adopting the 'top-k' algorithm or the 'top-p' algorithm. Specifically, as w increases, the $Precision$ keeps increasing slowly, while the $Recall$ first grows and then drops, causing the $F\text{-}measure$ to first increase and then decrease. This means Loader tends to judge a log sequence as normal (causing more false negatives), when the sliding window size becomes larger. After analyzing the lengths of all log sequences in *HDFS* (the length of a log sequence is the number of log messages in it), we find the mean length is 19. We infer that, a proper sliding window size should be less than the mean length, so that not too many log sequences need padding to fill the sliding window to make Loader work, and only require Loader to perform prediction once, *i.e.*, predicting the last log message. On the other hand, the sliding window size should be neither too small nor too large, as too few log messages could not convey enough information, while too many log messages could bring too much noise, both would harm the performance of Loader. For the remaining three parameters (N, h, and d_{ff}), the effectiveness of Loader fluctuates with different parameter values, when the 'top-k' algorithm is adopted, whereas keeps relatively stable when the 'top-p' algorithm is adopted. This means the number of layers (N), the number of heads in multi-head self-attention (h), and the inner layer dimension of feed-forward network (d_{ff}) do affect the learning ability of Loader, but the effectiveness can be improved via selecting a proper set of candidate log templates at each sliding step. This also confirms the flexibility and robustness of our proposed 'top-p' algorithm.

4.6 A Simple Illustration of Detecting

To better understand how Loader detects anomalies, we picked out a normal log sequence and an anomalous one from the testing set of *HDFS*, which are shown in Fig. 9 (each number represents the template ID of a log message). The first ten log messages of these two log sequences keep identical, until the 11th. Because the sliding window size is 10 (the default value), we fed the first ten log messages into Loader, and checked the output probability distribution, which is visualized in Fig. 10. We can see that the template with ID 13 has a probability of more than 0.9999, so the candidate set $\mathcal{C}$ contains only one element since $\lambda_p = 0.95$ (the default value), *i.e.*, $\mathcal{C} = \{13\}$. As a result, the second log sequence in Fig. 9 would be reported as anomalous by Loader, because $5 \notin \mathcal{C}$. In practice, we could terminate the detecting process of the second log sequence at this moment. For the first log sequence, however, we cannot say it is normal at this moment, and need to go on checking the following log messages.

5 Conclusion

Log anomaly detection is a critical task in service and system management. Especially for today's complex and large-scale services and systems, it becomes difficult or even infeasible to manually detect anomalies from massive logs in time. Thus, automated log anomaly detection becomes imperative. In this paper, we present a novel semi-supervised log anomaly detection approach, Loader, which leverages the Transformer encoder, instead of the commonly used RNN and its variants, to capture patterns from normal log sequences. Loader only needs normal log sequences for training. When detecting, it gives a candidate set of the most possible log templates, that may appear after the input log substring under normal conditions. If the template of the actual next log message is not in the candidate set, then it implies an anomaly. What's more, we design a more flexible and robust 'top-p' algorithm to determine the candidate set in detecting, rather than the 'top-k' algorithm adopted by previous similar methods. Extensive experiments have been conducted based on three public log datasets, to validate the effectiveness of Loader. The results confirm that, our approach Loader is better than or competitive with other unsupervised and semi-supervised log anomaly detection approaches.

References

[1] S. Huang, C. Fung, K. Wang, P. Pei, Z. Luan, and D. Qian, "Using recurrent neural networks toward black-box system anomaly prediction," in *Proc. IWQoS*, 2016, pp. 1–10.

[2] Q. Lin, K. Hsieh, Y. Dang, H. Zhang, K. Sui, Y. Xu, J.-G. Lou, C. Li, Y. Wu, R. Yao, M. Chintalapati, and D. Zhang, "Predicting node failure in cloud service systems," in *Proc. ESEC/FSE*, 2018, pp. 480–490.

[3] L. Wang, N. Zhao, J. Chen, P. Li, W. Zhang, and K. Sui, "Root-cause metric location for microservice systems via log anomaly detection," in *Proc. ICWS*, 2020, pp. 142–150.

[4] J. Gao, H. Wang, and H. Shen, "Task failure prediction in cloud data centers using deep learning," *IEEE Trans. Serv. Comput.*, vol. 15, no. 3, pp. 1411–1422, 2022.

[5] M. Cinque, R. D. Corte, and A. Pecchia, "Microservices monitoring with event logs and black box execution tracing," *IEEE Trans. Serv. Comput.*, vol. 15, no. 1, pp. 294–307, 2022.

[6] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang, and C. He, "Latent error prediction and fault localization for microservice applications by learning from system trace logs," in *Proc. ESEC/FSE*, 2019, pp. 683–694.

[7] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proc. SOSP*, 2009, pp. 117–132.

[8] K. Zhang, J. Xu, M. R. Min, G. Jiang, K. Pelechrinis, and H. Zhang, "Automated it system failure prediction: A deep learning approach," in *Proc. Big Data*, 2016, pp. 1291–1300.

[9] S. He, J. Zhu, P. He, and M. R. Lyu, "Experience report: System log analysis for anomaly detection," in *Proc. ISSRE*, 2016, pp. 207–218.

[10] S. He, P. He, Z. Chen, T. Yang, Y. Su, and M. R. Lyu, "A survey on automated log analysis for reliability engineering," *ACM Comput. Surv.*, vol. 54, no. 6, 2021.

[11] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proc. CCS*, 2017, pp. 1285–1298.

[12] W. Meng, Y. Liu, Y. Zhu, S. Zhang, D. Pei, Y. Liu, Y. Chen, R. Zhang, S. Tao, P. Sun, and R. Zhou, "Loganomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs," in *Proc. IJCAI*, 2019, pp. 4739–4745.

[13] M. Chen, A. Zheng, J. Lloyd, M. Jordan, and E. Brewer, "Failure diagnosis using decision trees," in *Proc. ICAC*, 2004, pp. 36–43.

[14] Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo, "Failure prediction in ibm bluegene/l event logs," in *Proc. ICDM*, 2007, pp. 583–588.

[15] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, "Fingerprinting the datacenter: Automated classification of performance crises," in *Proc. EuroSys*, 2010, pp. 111–124.

[16] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, J. Chen, X. He, R. Yao, J.-G. Lou, M. Chintalapati, F. Shen, and D. Zhang, "Robust log-based anomaly detection on unstable log data," in *Proc. ESEC/FSE*, 2019, pp. 807–817.

[17] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, "Mining invariants from console logs for system problem detection," in *Proc. USENIX ATC*, 2010, pp. 1–14.

[18] Z. Wang, Z. Chen, J. Ni, H. Liu, H. Chen, and J. Tang, "Multi-scale one-class recurrent neural networks for discrete event sequence anomaly detection," in *Proc. KDD*, 2021, pp. 3726–3734.

[19] L. Yang, J. Chen, Z. Wang, W. Wang, J. Jiang, X. Dong, and W. Zhang, "Semi-supervised log-based anomaly detection via probabilistic label estimation," in *Proc. ICSE*, 2021, pp. 1448–1460.

[20] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. NeurIPS*, 2017, pp. 6000–6010.

[21] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proc. NAACL-HLT*, 2019, pp. 4171–4186.

[22] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," in *Proc. ICLR*, 2021.

[23] S. Karita, N. Chen, T. Hayashi, T. Hori, H. Inaguma, Z. Jiang, M. Someki, N. E. Y. Soplin, R. Yamamoto, X. Wang, S. Watanabe, T. Yoshimura, and W. Zhang, "A comparative study on transformer vs rnn in speech applications," in *Proc. ASRU*, 2019, pp. 449–456.

[24] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *Proc. IPOM*, 2003, pp. 119–126.

[25] R. Vaarandi and M. Pihelgas, "Logcluster - a data clustering and pattern mining algorithm for event logs," in *Proc. CNSM*, 2015, pp. 1–7.

[26] H. Dai, H. Li, C.-S. Chen, W. Shang, and T.-H. Chen, "Logram: Efficient log parsing using n-gram dictionaries," *IEEE Trans. Softw. Eng.*, vol. 48, no. 3, pp. 879–892, 2022.

[27] L. Tang and T. Li, "Logtree: A framework for generating system events from raw textual logs," in *Proc. ICDM*, 2010, pp. 491–500.

[28] T. Xiao, Z. Quan, Z.-J. Wang, K. Zhao, and X. Liao, "Lpv: A log parser based on vectorization for offline and online log parsing," in *Proc. ICDM*, 2020, pp. 1346–1351.

[29] M. Du and F. Li, "Spell: Streaming parsing of system event logs," in *Proc. ICDM*, 2016, pp. 859–864.

[30] ——, "Spell: Online streaming parsing of large unstructured system logs," *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 11, pp. 2213–2227, 2019.

[31] A. A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "Clustering event logs using iterative partitioning," in *Proc. KDD*, 2009, pp. 1255–1264.

[32] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "A lightweight algorithm for message type extraction in system application logs," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 11, pp. 1921–1936, 2012.

[33] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *Proc. ICWS*, 2017, pp. 33–40.

[34] A. Vervaet, R. Chiky, and M. Callau-Zori, "Ustep: Unfixed search tree for efficient log parsing," in *Proc. ICDM*, 2021, pp. 659–668.

[35] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and benchmarks for automated log parsing," in *Proc. ICSE-SEIP*, 2019, pp. 121–130.

[36] S. Huang, Y. Liu, C. Fung, R. He, Y. Zhao, H. Yang, and Z. Luan, "Hitanomaly: Hierarchical transformers for anomaly detection in system log," *IEEE Trans. Netw. Serv. Manag.*, vol. 17, no. 4, pp. 2064–2076, 2020.

[37] H. Guo, S. Yuan, and X. Wu, "Logbert: Log anomaly detection via bert," in *Proc. IJCNN*, 2021, pp. 1–8.

[38] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen, "Log clustering based problem identification for online service systems," in *Proc. ICSE-C*, 2016, pp. 102–111.

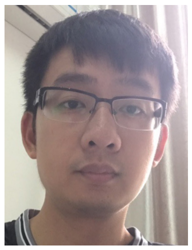
[39] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. CVPR*, 2016, pp. 770–778.

[40] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[41] A. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *Proc. DSN*, 2007, pp. 575–584.

[42] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. ICLR (Poster)*, 2015.

Tong Xiao received the bachelor's degree in software engineering from the North University of China, Taiyuan, China, and the master's degree in computer science from the National University of Defense Technology, Changsha, China. He is currently a PhD student at the College of Computer Science and Electronic Engineering, Hunan University, Changsha, China. His research interests include automated log analysis, anomaly detection, failure prediction, data mining, and deep learning.



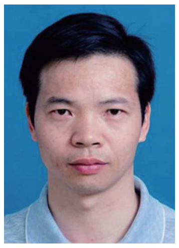
Zhe Quan received the PhD degree in computer science from the University de Picardie Jules Verne, France. He is currently a professor at the College of Computer Science and Electronic Engineering, Hunan University (HNU), Changsha, China. His main research interests include parallel and high-performance computing, artificial intelligence, data mining, and machine learning. He has published a set of research papers in venues such as IEEE TPDS, AAAI, IJCAI, ICDM, ICSC, and BIBM.

Zhi-Jie Wang (Member, IEEE) received the PhD degree in computer science from the Shanghai Jiao Tong University, Shanghai, China. He is currently an associate professor at the College of Computer Science, Chongqing University (CQU), Chongqing, China. His current research interests include artificial intelligence, machine learning, data mining, and databases. He has published a set of research papers in these fields including IEEE TPDS, IEEE TKDE, IEEE TMM, IEEE TALSP, IEEE TCSS, AAAI, IJCAI, and ICDM. He is also a member of ACM and CCF.

Yuquan Le is currently a PhD student at Hunan University. He received a BS degree from Nanchang University and an MS degree from Hunan University. His current research interests include Natural Language Processing, Legal Artificial Intelligence, data mining, and machine learning. He has published research papers in venues such as IJCAI, ECAI, TASLP, CIKM, ICASSP, AACL, and PAKDD.

Yunfei Du received the BS degree from the Beijing Institute of Technology, Beijing, China, and the PhD degree from the National University of Defense Technology, Changsha, China. He is now the Chief Architecturer for cluster computing in Huawei Technologies Co., Ltd. His research interests include parallel and distributed systems, fault tolerance, and scientific computing. He has published a set of research papers in venues such as IEEE TPDS, AAAI, PACT, and ICCAD.

Xiangke Liao received the BS degree from the Department of Computer Science and Technology, Tsinghua University, Beijing, China, in 1985, and the MS degree from the National University of Defense Technology, Changsha, China, in 1988. He is a Full Professor of the College of Computer, National University of Defense Technology. His research interests include parallel and distributed computing, high-performance computer systems, operating systems, cloud computing, and networked embedded systems.



Kenli Li (Senior Member, IEEE) received the PhD degree in computer science from the Huazhong University of Science and Technology, Wuhan, China, in 2003. He was a Visiting Scholar with the University of Illinois at Urbana-Champaign, Champaign, IL, USA, from 2004 to 2005. He is currently a Full Professor of the College of Computer Science and Electronic Engineering, Hunan University, Changsha, China. He has served on the editorial boards of the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Computers*, the *IEEE Transactions on Sustainable Computing*, and the *IEEE Transactions on Industrial Informatics*. His current research interests include parallel computing, cloud computing, big data computing, and neural computing.

Keqin Li (Fellow, IEEE) is a SUNY Distinguished Professor of Computer Science with the State University of New York. He is also a National Distinguished Professor with Hunan University, China. His current research interests include cloud computing, fog computing and mobile edge computing, energy-efficient computing and communication, embedded systems and cyber-physical systems, heterogeneous computing systems, big data computing, high-performance computing, CPU-GPU hybrid and cooperative computing, computer architectures and systems, computer networking, machine learning, intelligent and soft computing. He has authored or coauthored over 900 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He holds nearly 70 patents announced or authorized by the Chinese National Intellectual Property Administration. He is among the world's top 5 most influential scientists in parallel and distributed computing in terms of both single-year impact and career-long impact based on a composite indicator of Scopus citation database. He has chaired many international conferences. He is currently an associate editor of the *ACM Computing Surveys* and the *CCF Transactions on High Performance Computing*. He has served on the editorial boards of the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Computers*, the *IEEE Transactions on Cloud Computing*, the *IEEE Transactions on Services Computing*, and the *IEEE Transactions on Sustainable Computing*. He is an AAAS Fellow, an IEEE Fellow, and an AAIA Fellow. He is also a Member of Academia Europaea (Academician of the Academy of Europe).