LPV: A Log Parsing Framework Based on Vectorization

Tong Xiao*, Zhe Quan*, Zhi-Jie Wang, *Member, IEEE*, Kaiqi Zhao, Xiangke Liao, Huang Huang, Yunfei Du, and Kenli Li, *Senior Member, IEEE*

***Abstract*—Logs are pervasive in modern computing systems, and are valuable to service and system management. Nevertheless, with the rapidly growing size and complexity of computing systems, the log volume is exploding, which makes automatic log analysis imperative. Generally, in automatic log analysis, the first and fundamental step is log parsing, to which a lot of effort has been devoted. However, in most existing log parsing methods, log messages are merely treated as plain text. In natural language processing (NLP) area, it is a common practice to represent words and sentences with vectors, then the similarity between two words or sentences can be measured by the distance between their vectors. Inspired by these, we put forward a novel log parsing framework, named LPV (Log Parser based on Vectorization), which performs log parsing by converting log messages and log templates into vectors, with the help of a vectorization method in NLP. LPV incorporates offline and online log parsing. In the offline log parsing, the central idea is to first represent log messages with vectors, so that the similarity between two log messages can be measured by the distance between their vectors, then we cluster log messages via clustering the vectors, and finally we extract log templates from the resultant clusters. By the end of the offline log parsing, each log template is assigned with an average vector, so that in the online log parsing, the similarity between an incoming log message and each log template can also be measured by the distance between their vectors. Extensive experiments have been conducted based on several public log datasets to evaluate LPV with three different vectorization methods. The results demonstrate that, with a proper vectorization method, LPV performs competitive with state-of-the-art log parsing methods, in both effectiveness and efficiency.**

***Index Terms*—Log parsing, log template extraction, log analysis, vectorization, service and system management.**

This paper is an extended version of our conference paper which has been published in the proceedings of the 20th IEEE International Conference on Data Mining (ICDM 2020) [1]. This work was supported in part by the National Key R&D Program of China (No. 2018YFB0204302), and in part by the NSFC (No. 61972425, U1811264). *(Corresponding author: Zhi-Jie Wang.)*

Tong Xiao, Zhe Quan, Huang Huang, and Kenli Li are with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China (e-mail: {xiaotong18, quanzhe, huanghuang, lkl}@hnu.edu.cn).

Zhi-Jie Wang is with the College of Computer Science and the Ministry of Education Key Laboratory of Dependable Service Computing in Cyber Physical Society, Chongqing University, Chongqing 400044, China (e-mail: cszjwang@cqu.edu.cn).

Kaiqi Zhao is with the School of Computer Science, University of Auckland, Auckland 1010, New Zealand (e-mail: kaiqi.zhao@auckland.ac.nz).

Xiangke Liao is with the College of Computer, National University of Defense Technology, Changsha 410073, China (e-mail: xkliao@nudt.edu.cn).

Yunfei Du is with the Huawei Technologies Co., Ltd., Shenzhen 518129, China (e-mail: duyunfei5@huawei.com).

*The first two authors contributed equally to this work.

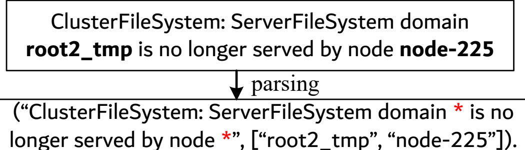


Fig. 1. Illustration of parsing a log message. The parsing result is represented by a tuple consisting of a log template and a list of variables.

I. Introduction

LOGS are common in modern computing systems while they are very valuable resources. There exists rich information about the resident system in logs, and logs play a very important role in the whole lifecycle of computing systems [2], [3]. Traditionally, system designers and developers employ logging mechanism to record significant events happened in the system, *e.g.*, the start/stop of a service, system state changes, *etc.* Meanwhile, system administrators and maintainers heavily rely on logs to understand system runtime status, when they detect and diagnose anomalies and failures, such as software/hardware exceptions, warnings, and errors [4]–[6]. On the other hand, with the growing scale and complexity of computing systems, the log volume is exploding, which makes it cumbersome, tedious, error-prone, and even impractical to manually deal with the huge amount of logs and mine useful information from them [7], [8], especially for high-performance computing (HPC) and cloud computing systems [9]–[13]. Thus, automatic log analysis is in urgent need, because it can free us from the above dilemmas and ease many management tasks, such as anomaly detection, failure prediction, and root cause analysis of service and system issues [14]–[22].

In typical automatic log analysis, the first and fundamental step is log parsing, which aims to parse unstructured logs into structured data [23], as logs are usually unstructured plain text output by the "`printf`" or similar functions in C language and the equivalents in other programming languages [24]–[26]. Fig. 1 gives an illustration of log parsing, where the log message in the upper box can be split into two parts: the constant and the variable (the variable part is the text in boldface). By log parsing we aim to obtain a log template, which keeps the constant part literally while the variable part is replaced with some wildcards (see the * in red color in Fig. 1). The final result of parsing a log message is a log

0000–0000/00$00.00 © 2021 IEEE

template, together with a list of variables which are replaced by wildcards in the log template.

However, it is often nontrivial to distinguish the constant part from the variable in a log message. In former literatures and past practice, there roughly exist three types of log parsing methods. The first type requires domain experts to go over logs and offer rules such as regular expressions, which is cumbersome, tedious, labor-intensive, and error-prone. The second type dives into the source code to reach the corresponding "print" statements of log messages [27], which can gain accurate log templates theoretically, but requires the source code to be available. The third type employs some heuristics or machine learning and data mining techniques to figure out the constant part automatically, which needs little domain knowledge and does not need access to the source code [3], [24], [25], [28]–[31]. In a nutshell, the third type of methods make decisions based on the fact that, the constant part appears in all log messages output by the same "print" statement, so the subsequence contained in more log messages is more likely to be the constant part. Thus, it is natural to treat log parsing as a *clustering problem*, followed by extracting the longest common subsequence (LCS) [32] of tokens from each cluster.

In natural language processing (NLP) area, it is a common practice to represent words and sentences with vectors, then the similarity between two words or sentences can be measured by the distance between two vectors. A log message is a string of tokens and can also be deemed as a sentence. The *bag-of-words* model [33] is a simple yet popular way to represent texts with vectors, but it is not trivial to apply this model to converting log messages into vectors, because the length (number of tokens) of a log message is usually small, while the vocabulary size (number of unique tokens) of logs is likely very large [34], [35], leading to the *Curse of Dimensionality* [30], [31] and high-dimensional sparse vectors. In recent years, distributed representations of words (or word embeddings) with a modest or low dimensionality, which are trained on large corpora, have greatly benefited various NLP tasks. For example, *word2vec* [36], [37] can learn high-quality word embeddings, which capture syntactic and semantic word relationships well; *ELMo*, furthermore, can learn contextualized word representations, which model syntax, semantics, and polysemy at the same time [38]. On the other hand, current state-of-the-art log anomaly detection approaches (*e.g.*, LogAnomaly [20], LogRobust [39], HitAnomaly [21]) propose to represent each log template with a semantic vector based on the distributed representations of its tokens, instead of simply using a template index, to better detect anomalies. But currently they rely on an existing log parsing method (*e.g.*, FT-tree [40], Drain [41]) to get log templates first, and then represent each log template with a vector by encoding each token with a distributed representation. One can easily understand that it would be much convenient for log anomaly detection if the log parsing method can return log templates and the corresponding vectors together.

Inspired by the above, we put forward a novel log parsing framework in this paper, whose basis is vectorization, *i.e.*, representing tokens, log messages, as well as log templates with vectors, utilizing a vectorization method in NLP. In [1], we proposed a log parser based on vectorization which is named LPV, the vectorization method used there is *word2vec*. This work extends LPV to a general framework, which can support any vectorization method. For consistency, we also dub this framework as LPV (Log Parser based on Vectorization), which still incorporates offline and online log parsing. Briefly, as for the offline log parsing, we first convert each textual log message into a vector, with the help of a vectorization method, *e.g.*, *bag-of-words*, *word2vec*, *ELMo*, *etc.* Then, we cluster the entire log messages via clustering their corresponding vectors. Next, we extract one or more log templates from each cluster of log messages, and merge similar log templates into a single one to get fine-tuned log templates. In the end, we assign each log template with an average vector, which is computed based on the vectors of log messages sharing this template. As for the online log parsing, we first convert an incoming log message into a vector, just as in the offline log parsing. Then we calculate the distance between this vector and every log template's vector. Subsequently, we can greatly accelerate the log parsing by matching this incoming log message only with a few closest log templates (in terms of the vector distance) generated in the offline log parsing.

To summarize, this paper makes the following contributions:

- In the offline log parsing, our framework represents log messages with vectors, and clusters log messages via clustering their corresponding vectors. This enables us to make full use of abundant history logs to achieve high effectiveness.
- Our framework also supports online log parsing. We assign each log template with an average vector, so that the similarity between an incoming log message and a log template can also be measured by the distance between two vectors. This idea can accelerate online log parsing by matching each incoming log message only with the most possible log templates.
- We have implemented the framework using three different vectorization methods, and conducted empirical study on several public log datasets, which are widely used by previous studies for evaluation. The experimental results demonstrate that, with a proper vectorization method, LPV performs competitive with state-of-the-art log parsing methods, in both effectiveness and efficiency.

The rest of this paper is organized as follows. Section II introduces the related work. Section III presents our framework LPV, including the overall workflow and details of each phase. In Section IV, we first evaluate the performance of LPV by comparing with state-of-the-art log parsing methods, then we measure the impacts of different parameters. Finally, we conclude our paper in Section V.

II. Related Work

Log parsing is particularly important for automatic log analysis, and has attracted a lot of interest from both academia and industry in past years. Existing log parsing methods roughly fall into three categories, *i.e.*, the rule-based, the source code-based, and the data-driven methods [42].

A. Rule-based and Source Code-based Log Parsing Methods

The rule-based log parsing methods heavily rely on domain experts to establish handcrafted rules, in forms like regular expressions. Although straightforward, this kind of methods require thorough understanding of the log formats, and a lot of manual efforts are needed to build different rules for different types of logs. Currently, a vast majority of industrial log management and analysis software systems offer interfaces for end users, to provide their customized log parsing rules, *e.g.*, Splunk, ELK, Logentries, *etc.*

The idea behind the source code-based log parsing methods is somewhat intuitive. If we can locate the "print" statement of a log message in the source code, we could definitely get its template. Xu *et al.* [27] employed such a method to extract log templates for system problem detection. This kind of methods bear two drawbacks: 1) the source code is not always available, especially for commercial software, which limits the application scope; 2) different programming languages may have different logging grammars, and need different tricks to get the templates.

B. Data-driven Log Parsing Methods

A lot of research efforts have been poured into the data-driven log parsing. Our method LPV also belongs to this category. The main techniques adopted by this kind of methods are frequent pattern mining and clustering.

Frequent pattern mining in log parsing tries to figure out the frequent words and patterns in a log dataset. The basic observation is that, more frequent words are more likely to be the constant. For example, in SyslogDigest [43], a sequence of words with high frequencies are considered as a message template. This method gets inspiration from signature abstraction in spam detection, and learns templates by constructing a tree structure for each message type to describe the template hierarchy, according to the frequencies of word combinations. Similarly, FT-tree [40] also maintains a tree structure for each message type, but it utilizes the frequencies of words rather than that of word combinations, so it supports incremental learning of templates. STE (statistical template extraction) [44] gives higher scores for template words than parameters based on statistics, then determines each word in a message to be a template word or parameter via score clustering. Logram [45] generates n-gram dictionaries to keep the frequency of each unique n-gram in a log dataset, then identifies dynamically generated tokens from low-appearing n-grams and replaces them with wildcards to generate log templates. Another technique line works as follows, first all frequent words in a log dataset are identified, then different clusters of log messages are formed according to the frequent words contained in each log message, and finally a log pattern (*i.e.*, a log template) is created for each cluster. Log parsing methods employing this technique include SLCT [46], LogHound [47] and LogCluster [3].

Clustering in log parsing is to cluster log messages corresponding to different templates into different groups, by employing clustering algorithms or some kind of heuristics. For example, LogTree [30] makes use of the format and structural information of log messages in the clustering process, via building semi-structural log messages and giving different importance to different levels, and various clustering algorithms can be plugged into it to generate system events. LogSig [31] tries to partition all log messages into k groups based on the term pairs generated for each log message, and constructs the message signature for each message group. IPLoM [28], [29] iteratively partitions the whole log dataset into respective clusters through three steps (*i.e.*, partition by token count, partition by token position, and partition by search for bijection, respectively), and finally produces a message type description (*i.e.*, a log template) for each cluster. In [48], clusters are formed in an online manner according to the log similarities between each arriving message and each existing cluster. In brief, if the highest log similarity is no less than a predefined threshold, the message is assigned to the cluster with the highest similarity, otherwise a new cluster is created from the message. Similarly, LogMine [49] defines a distance between two log messages or patterns, and inserts each log message or pattern into an existing cluster if the distance between it and the representative of the cluster is less than a threshold, or creates a new cluster otherwise. HELO [50], unlike the above methods, consists of an offline clustering process and an online one. In the offline clustering process, HELO recursively finds a split column to divide log messages into different clusters, until all clusters are stable, and then identifies a group template for each cluster. In the online clustering process, the group templates from the offline process can be adapted according to incoming log messages. Our method, LPV, initially proposed in [1], incorporates both offline and online log parsing. The offline log parsing clusters log messages by converting them into vectors, with the help of *word2vec*, and then extracts log templates from the clusters. The online log parsing converts each incoming log message into a vector, and identifies its template from the templates generated in the offline log parsing through vector distance. In this journal version, we extend LPV to a general framework which can support any vectorization method, and take *bag-of-words* and *ELMo* as two additional examples. In addition, extensive experiments were conducted to illustrate how the parameters of LPV affect its effectiveness.

There are also other techniques used by the data-driven log parsing methods. For example, MoLFI [51] formulates log parsing as a multi-objective optimization problem, *i.e.*, maximizing the number of log messages matched by each template and the specificity of each template to a particular type of event at the same time, and employs a multi-objective genetic algorithm (NSGA-II [52]) to tackle the problem. Spell [24], [25] utilizes an LCS-based approach, and achieves nearly linear time complexity for most incoming log messages by pre-filtering. Drain [41] utilizes a fixed-depth parse tree to accelerate the log group search process, by log message length, preceding tokens, and token similarity successively.

All in all, in most existing log parsing methods, log messages are merely treated as pure strings, and string matching or string distance is the basis to accomplish log parsing.

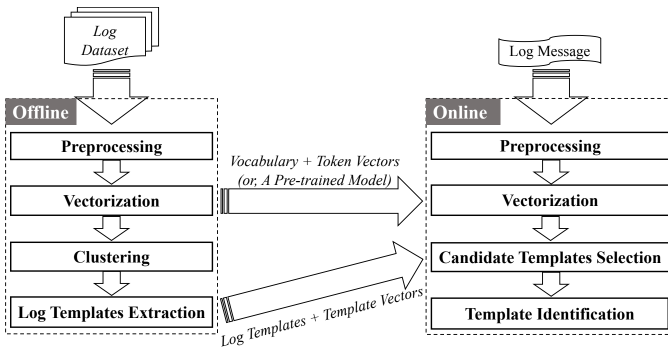


Fig. 2. Overall framework of LPV.

III. Methodology

In this section, we first give an overview of LPV, and then cover its implementation details.

A. Overview of LPV

The core idea of LPV is to represent log messages as well as log templates with vectors, so that the similarity between two log messages or between a log message and a log template can be measured by the distance between their corresponding vectors.

Fig. 2 gives the overall framework of LPV, which consists of an offline log parsing process and an online one. In the offline log parsing, we first divide the entire log dataset into separate clusters, by first representing log messages with vectors utilizing some kind of vectorization method, and then clustering the resultant vector set via a clustering algorithm. Afterwards, we extract one or more log templates from each cluster of log messages, and merge similar templates into a single one to get the final fine-tuned templates. Finally, we assign each log template with a **template vector**, which is defined as an average vector of the vectors of all log messages sharing the same log template. The online log parsing fully leverages the outputs of the offline log parsing. Specifically, for each incoming log message, we first convert it into a vector, just as in the offline log parsing. Then, we calculate the distance between this vector and each template vector, and pick out the *n* closest template vectors (the *n* corresponding templates are chosen as **candidate templates**). Finally, we match the incoming log message with each candidate template, and select the entirely matched one as the template of the incoming log message, if such a candidate template exists; otherwise, if there does not exist such a candidate template, the log message itself is regarded as its template, and the actual template can be extracted in later offline log parsing.

B. The Offline Log Parsing

To the offline log parsing, the input is an existing log dataset collected over a period, and the outputs are: (i) a vocabulary together with the vectors of all tokens, for vectorization methods like *bag-of-words* and *word2vec*, or a pre-trained model, for vectorization methods like *ELMo*; and (ii) a set of log templates together with their corresponding template vectors. Next, we explain each phase of the offline log parsing, including Preprocessing, Vectorization, Clustering, and Log Templates Extraction, as shown in Fig. 2.

TABLE I
Summary of Two Supercomputer Log Datasets (*BGL* and *HPC*)[a]

Dataset	Period	$\#\{OLM\}$	$\#\{ULM\}$	$\#\{USLM\}$
BGL	215 days	4,747,963	358,353	773
HPC	9 years	433,490	9,965	156

[a]CFDR Data, https://www.usenix.org/cfdr-data

1) Preprocessing: The aim of this phase is to reduce the dataset size and vocabulary size, which is accomplished through two operations: duplicates removal and common variables substitution.

► *Duplicates removal.* There often exist lots of repeated log messages in a log dataset with a long time span. Note that, here we only take the *content* field of each log message into consideration, and exclude other fields like the timestamp, the component name, *etc.* See Table I for an example, in which *BGL* is a log dataset collected from a BlueGene/L supercomputer located at Lawrence Livermore National Labs (LLNL) and *HPC* is a log dataset collected from an HPC system located at Los Alamos National Laboratory (LANL), $\#\{OLM\}$, $\#\{ULM\}$, and $\#\{USLM\}$ refer to the total number of original log messages, the number of unique log messages after all repeated log messages have been removed, and the number of unique log messages after common variables substitution and further eliminating identical ones, respectively. We can see from the 3rd and 4th columns (*i.e.*, $\#\{OLM\}$ and $\#\{ULM\}$) that, more than 90% of the log messages in *BGL* and *HPC* are repeated. So by removing the repeated log messages, we can dramatically reduce the dataset size and avoid a large number of repeated manipulations.

► *Common variables substitution.* It is observed that the huge number of variables (*e.g.*, IP addresses, the node IDs of a large-scale system, usernames, numbers, *etc.*) is a main cause to the large vocabulary size of a log dataset. Although we cannot identify all variables, or we would have accomplished log parsing, we do know some most possible formats of variables, such as common variables like IP addresses, numbers, *etc.* Therefore, in LPV we utilize some regular expressions to substitute those common variables with a few special tokens, *e.g.*, substituting all IP addresses with a special token "`$$IPADDR$$`". After these substitutions, a lot of log messages could be the same, and duplicates can be removed again to get unique substituted log messages, since these duplicates definitely have the same log template. In this way, we not only avoid a large vocabulary size, but also reduce the dataset size further (refer to the last column of Table I), benefiting to the rest processing. It is worth noting that, (i) we do not carry out common variables substitution in-place, the original log messages stay as is; and (ii) we will extract log templates from the original log messages.

To help understanding the Preprocessing phase, Fig. 3 gives a simple illustration. It can be easily understood from the figure that, by recording the indices of original log messages and unique log messages, LPV enables us to trace original log

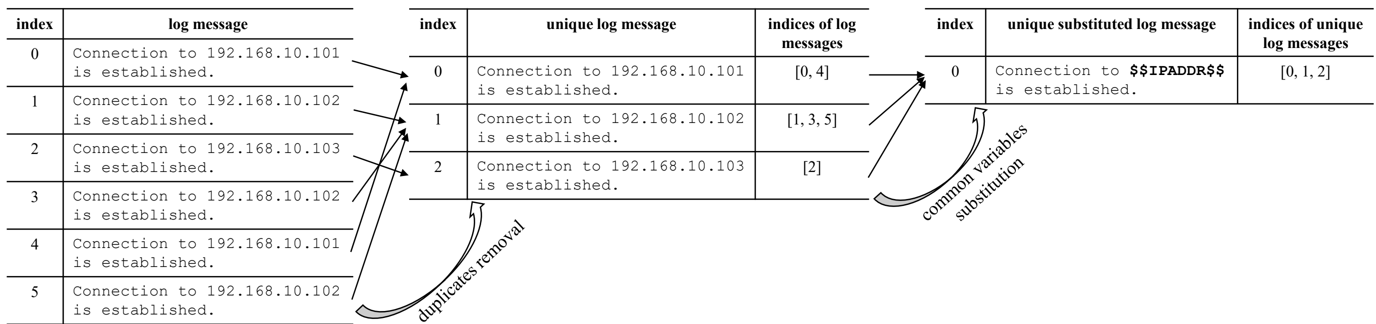


Fig. 3. A simple illustration of the Preprocessing phase of the offline log parsing.

messages throughout the offline log parsing.

2) Vectorization: In this phase, we try to represent each unique substituted log message with a vector by two steps: *Word2Vector* and *Log2Vector*. For log messages that have been preprocessed into the same unique substituted log message, they share the same log template, and thus can be represented by an identical vector.

► *Word2Vector.* In this step we map each unique token to a vector, utilizing some kind of vectorization method. Here we take three vectorization methods as instances, *i.e.*, *bag-of-words*, *word2vec*, and *ELMo*. Specifically, we first split each unique substituted log message into a list of tokens by whitespaces and punctuation marks including "=", "(", ")", "[", "]", ":", "?", *etc.* Then, we build a vocabulary of these tokens and sort them by frequency in descending order. Note that punctuation marks are also treated as individual tokens, as they always keep constant across all the log messages output by the same "print" statement. Finally, (1) for *bag-of-words*, we encode each token using a one-hot scheme, *i.e.*, representing each token with a sparse vector, whose length is the vocabulary size, and all of its elements are 0's except the one at the position corresponding to the token, which is set to 1. For example, assuming there are four tokens in a vocabulary, then they can be encoded as follows:

1st token: [1, 0, 0, 0]
2nd token: [0, 1, 0, 0]
3rd token: [0, 0, 1, 0]
4th token: [0, 0, 0, 1]

(2) For *word2vec*, we operate as in our previous work [1]. We adopt the Skip-gram model architecture described in [37], which is to predict the surrounding words of each given word (the number of words to predict, to the left and right of the given word, is called *window size*). We employ negative sampling as an alternative to the hierarchical softmax, which selects several negative samples for each data sample. But we disable the subsampling of high-frequency tokens, because we think they are more likely to be constant in log messages. In addition, when training, we constrain the pair of input and label tokens within a unique substituted log message, since there exist no strong correlations between the tail tokens of a unique substituted log message and the head tokens of the next one. (3) For *ELMo*, we use the TensorFlow version[1] to train a 2-layer bidirectional language model (biLM) on a new corpus (*i.e.*, the set of unique substituted log messages from a log dataset), and dump biLM embeddings for the entire set of unique substituted log messages to a single file. Here each layer of the biLM is a bidirectional LSTM. Then, we select the top layer representation (LSTM output) of each token as its vector.

[1]https://github.com/allenai/bilm-tf

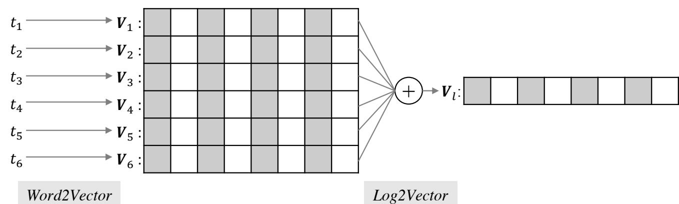


Fig. 4. A simple illustration of the Vectorization phase of the offline log parsing.

► *Log2Vector.* In this step we compute a vector for each unique substituted log message, by summing up all its tokens' vectors. Generally, for a unique substituted log message l which consists of n tokens $[t_1, t_2, \ldots, t_n]$, assuming the vectors of these tokens are $\mathbf{V}_1, \mathbf{V}_2, \ldots, \mathbf{V}_n$ respectively, then the vector of l (denoted by $\mathbf{V}_l$) is computed as follows:

$$\mathbf{V}_l = \sum_{i=1}^{n} \mathbf{V}_i \tag{1}$$

It can be easily seen that, each token in a unique substituted log message l makes a contribution to l's vector $\mathbf{V}_l$.

Fig. 4 shows the process of converting a simple unique substituted log message l, which consists of 6 tokens (*i.e.*, $[t_1, t_2, t_3, t_4, t_5, t_6]$), into a vector $\mathbf{V}_l$. By utilizing a proper vectorization method, we aim to achieve the goal that, vectors of unique substituted log messages sharing the same template are close in the vector space. In addition, by summing up the vectors of all tokens in a unique substituted log message, all unique substituted log messages' vectors have the same dimension. These features could be convenient to the following Clustering phase.

3) Clustering: After representing each unique substituted log message with a vector, we can calculate a distance between any two, and cluster these unique substituted log messages based on the distance, as shown by the simple illustration in Fig. 5. Since we often do not know the number of log templates in a log dataset in advance, so it is more desirable

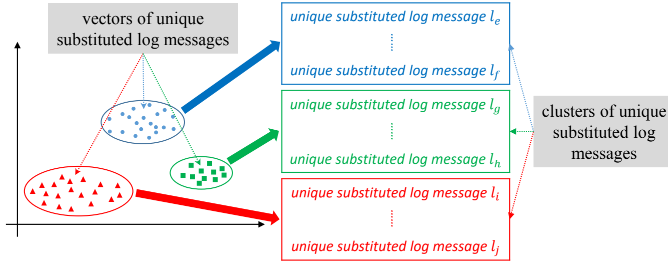


Fig. 5. A simple illustration of the Clustering phase of the offline log parsing.

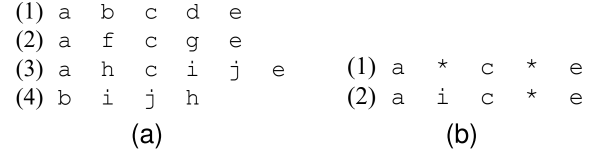


Fig. 6. Examples for merge. (a) Intra-cluster Merge; (b) Inter-cluster Merge.

to employ a clustering algorithm that does not need to specify the number of clusters. In view of this, we adopt the Complete-Linkage clustering method [53] and use Euclidean distance in our implementation. The hierarchical/agglomerative clustering process will be terminated when the minimum distance between any two clusters exceeds a threshold τ_d. We would like to point out that, other clustering algorithms and distance types may also be plugged into our framework.

4) Log Templates Extraction: Log templates will be extracted through three steps in this phase, *i.e.*, Partition, Intra-cluster Merge, and Inter-cluster Merge. Next, we address them successively.

► *Partition.* For every cluster of unique substituted log messages formed in the Clustering phase, we generate a candidate log template for each of its members in this step. Specifically, for each special token in a unique substituted log message, we first extract all common variables previously substituted by it, since we can trace the original log messages, just as mentioned in the Preprocessing phase (cf. Section III-B1). If all the variables are identical, then this special token is replaced with the unique variable value; otherwise, it is replaced with the wildcard. Taking Fig. 3 as an instance, because the IP addresses substituted by "`$$IPADDR$$`" in the unique substituted log message "`Connection to $$IPADDR$$ is established.`" are not identical, the candidate log template would be "`Connection to * is established.`" if we choose "$\star$" as the wildcard; on the other hand, if all the IP addresses are the same, say "`127.0.0.1`", then we treat it as a constant, and the candidate log template would be "`Connection to 127.0.0.1 is established.`". The reason why we call this step "Partition" is that, there will be a candidate log template for each member of a cluster after this step, which likes partitioning a cluster with multiple members into multiple smaller ones.

This step brings the following two benefits, which can improve the parsing effectiveness:

- Possible mistakes made by the previous common variables substitution can be corrected, *i.e.*, constants which have common variables' formats and are substituted by special tokens are restored to their original contents.
- If the distance threshold τ_d in the Clustering phase (cf. Section III-B3) is too big, then there may be some unique substituted log messages within a cluster which do not share the same log template, this step generates different candidate log templates for them and mitigates the influence of a too big τ_d.

► *Intra-cluster Merge.* In this step we try to merge similar candidate log templates within a cluster into a single one, through the following six substeps:

- First, we split each candidate log template in a cluster into tokens by whitespaces, equal signs ("="), *etc.* We say the tokens at the same position of each candidate log template form a column, *e.g.*, the first tokens of all candidate log templates in the cluster form the first column. Here we assume the number of columns of these candidate log templates to be n.
- Second, for each column, *e.g.*, the ith column ($1 \leqslant i \leqslant n$), we find out the most frequent token t_i and its frequency f_i (*i.e.*, the number of occurrences) from the tokens in this column. If there are multiple such tokens, any one can be selected.
- Third, we first figure out all unique values and their respective frequencies from the list $[f_1, f_2, \ldots, f_n]$ obtained in the previous substep. Suppose that there are m unique values, denoted by $uf_1, uf_2, \ldots, uf_m$. Then we iterate from the most frequent unique value to the least frequent which are bigger than 1. In each iteration (suppose the unique value being iterated is $uf_j > 1$ ($1 \leqslant j \leqslant m$)), we count the number of columns which satisfy $f_i \geqslant uf_j$, and check the ratio of the count result (*i.e.*, the number of satisfied columns) over the total number of columns (*i.e.*, n). If the ratio is no less than a threshold τ_r, then we stop the iteration and select the most frequent token t_i of each satisfied column as a constant (go from the first column to the last successively), and proceed to the fourth substep. If there are no constants found after the whole iteration, then it means there is no need to merge candidate log templates in this cluster, and we quit the Intra-cluster Merge step for this cluster.
- Fourth, we pick out all candidate log templates, each of which contains the selected constants successively. This means that we do not require the constants to be in some fixed columns, but only to keep the relative order, *i.e.*, a constant selected from a column must appear before constants from following columns.
- Fifth, we get a merged log template by concatenating all constants in order, and inserting a wildcard between two constants, if there is at least one token between them in any of the candidate log templates picked out.
- Finally, for the remaining candidate log templates within the cluster, we repeat the above operations to merge them recursively.

Think of the four lines in Fig. 6a as a cluster, and let $\tau_r = 0.5$, the Intra-cluster Merge of these lines goes as follows:

- First, each of the four lines is split by whitespaces, and the number of columns is 6 (determined by line (3)).
- Second, for the 1st column, the most frequent token t_1 is `a` and its frequency $f_1 = 3$. Similarly, t_2 is `b` and $f_2 = 1$, t_3 is `c` and $f_3 = 3$, t_4 is `d` and $f_4 = 1$, t_5 is `e` and $f_5 = 2$, t_6 is `e` and $f_6 = 1$. As for the 2nd or 4th column, since the frequency of each token is 1, so any token can be selected for each of these two columns.
- Third, now we get $[f_1, f_2, f_3, f_4, f_5, f_6] = [3, 1, 3, 1, 2, 1] \Rightarrow uf_1 = 1, uf_2 = 3, uf_3 = 2$, it is obvious that the most frequent unique value bigger than 1 is 3, and the number of columns satisfying $f_i \geqslant 3$ is 2 (f_1 and f_3). Because $2/6 < \tau_r$, we iterate to the less frequent unique value, *i.e.*, 2, and the number of columns satisfying $f_i \geqslant 2$ is 3 (f_1, f_3, and f_5). Since $3/6 = \tau_r$, so t_1, t_3, t_5, *i.e.*, `a`, `c`, `e` are selected as constants.
- Fourth, lines (1), (2) and (3) are picked out, as they contain `a`, `c`, `e` successively.
- Fifth, now we can get the result "`a` $\star$ `c` $\star$ `e`" if "$\star$" is chosen as the wildcard, because there is one token between `a` and `c`, and one or two token(s) between `c` and `e`, in the three lines picked out.
- Finally, since only line (4) is left, so we do not need to merge it recursively.

► *Inter-cluster Merge.* If the distance threshold τ_d in the Clustering phase (cf. Section III-B3) is too small, then some unique substituted log messages sharing the same log template may be clustered into different clusters, and we also need to merge their candidate log templates into a single one. Briefly, for each candidate log template in a cluster C_i, we merge it with the ones in the closest cluster C_j (C_j can be easily identified from the clustering result). The Inter-cluster Merge is much simpler than the Intra-cluster Merge, since in this step two candidate log templates are merged only if they have the same length and match at each position. And two candidate log templates match at one position if and only if their tokens at this position are identical, or one of the tokens is the wildcard. As for the result template, the value at each position is set as follows:

- If the tokens of the two candidate log templates at this position are identical, then the token is set as the result template's value at this position.
- Otherwise, one of the two tokens at this position should be the wildcard, then the result template's value at this position is set to the wildcard.

For example, assume the two lines in Fig. 6b are two candidate log templates belonging to two closest clusters, then the Inter-cluster Merge result of them would be "`a` $\star$ `c` $\star$ `e`" if "$\star$" is the wildcard.

Once the log templates are extracted, for each log template, we can get a set of unique log messages sharing this template, because we can trace them through the unique substituted log messages, as described in the Preprocessing phase (cf. Section III-B1). Then we compute a template vector for each log template. In our implementation, the template vector (denoted by **TV**) of a log template T is defined as follows:

$$\mathbf{TV} = \frac{1}{|\Theta|} \sum_{l \in \Theta} \mathbf{V}_l \tag{2}$$

where Θ is the set of unique log messages sharing this template T, $|\Theta|$ is the number of elements in Θ, and $\mathbf{V}_l$ is the vector of l. Recall that, unique log messages preprocessed into the same unique substituted log message have an identical vector.

C. The Online Log Parsing

After the offline log parsing is finished, its outputs (*i.e.*, a vocabulary together with the vectors of all tokens or a pre-trained model, and all log templates together with their template vectors) can be leveraged by the online log parsing. Without loss of generality, assume that there are N log templates ($T_1, T_2, \ldots, T_N$) generated in the offline log parsing, and their template vectors are $\mathbf{TV}_1, \mathbf{TV}_2, \ldots, \mathbf{TV}_N$ respectively, the online log parsing is performed through the following four phases: Preprocessing, Vectorization, Candidate Templates Selection, and Template Identification, as shown in Fig. 2. Specifically, for each incoming log message l, the online log parsing is done as follows:

- **Preprocessing**. This phase is similar to the Preprocessing phase of the offline log parsing (cf. Section III-B1), but we only need to perform common variables substitution here, since there is only one log message and duplicates removal is not needed.
- **Vectorization**. In this phase, the substituted log message is first split into tokens. Then, for vectorization methods like *bag-of-words* and *word2vec*, the vector of each token is obtained by looking up the vocabulary together with all tokens' vectors from the offline log parsing; for vectorization methods like *ELMo*, the pre-trained model from the offline log parsing is used to compute the vectors for all tokens on the fly. In this process, new tokens which are unseen in the offline log parsing are mapped to a special token `UNK`. For *bag-of-words*, it is treated just like other tokens; for *word2vec* and *ELMo*, it is initialized with a zero vector. Finally, this log message's vector $\mathbf{V}_l$ is calculated by summing up the vectors of all its tokens according to equation (1), just as in the offline log parsing.
- **Candidate Templates Selection**. In this phase, we calculate the distance between the log message's vector $\mathbf{V}_l$ and each template vector $\mathbf{TV}_i (1 \leqslant i \leqslant N)$, and find out the n closest template vectors to $\mathbf{V}_l$. Note that the distance type used here should be the same as the one used in the Clustering phase of the offline log parsing, *i.e.*, Euclidean distance in our implementation (cf. Section III-B3). Suppose the n closest template vectors to $\mathbf{V}_l$ are $\mathbf{TV}_{i1}, \mathbf{TV}_{i2}, \ldots, \mathbf{TV}_{in}$, then their corresponding templates $T_{i1}, T_{i2}, \ldots, T_{in}$ are called the **top n candidate templates**.
- **Template Identification**. In this phase, we try to identify the incoming log message l's template from the top n candidate templates. This is accomplished by matching

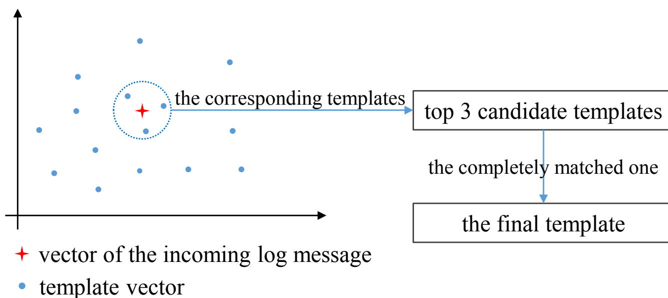


Fig. 7. A simple illustration of the Candidate Templates Selection and Template Identification phases of the online log parsing.

TABLE II
THE THREE LOG DATASETS USED FOR EVALUATION

Dataset	$\#$ *log messages*	$\#$ *ground truth templates*
BGL	4,747,963	394
HPC	433,490	105
HDFS	11,175,629	29

l with each candidate template, and selecting the completely matched one as the result. Here, we think a log message and a log template are completely matched, if and only if they become identical after replacing each wildcard in the log template with some character(s). For example, suppose the line (1) in Fig. 6a is an incoming log message, and the line (1) in Fig. 6b is one of the top n candidate templates, then they are completely matched, since they become identical after replacing the first wildcard with `b` and the second wildcard with `d` in the candidate template. On the other hand, if the incoming log message does not match with any one of the top n candidate templates, then the log message itself is treated as its template and the actual template can be generated in later offline log parsing.

Fig. 7 gives a simple illustration of the last two phases of the online log parsing. Note that, a small n like 3 or 5 is enough, as will be shown later in our experiments. This indicates that, the online log parsing can be dramatically accelerated by matching each incoming log message only with a few candidate templates.

IV. EVALUATION

In this section, we evaluate the effectiveness and efficiency of LPV employing three different vectorization methods respectively, *i.e.*, *bag-of-words*, *word2vec*, and *ELMo*, by comparing with state-of-the-art log parsing methods. For simplicity, we denote LPV employing *bag-of-words* as $\text{LPV}_{[\text{BoW}]}$, LPV employing *word2vec* as $\text{LPV}_{[\text{W2V}]}$, and LPV employing *ELMo* as $\text{LPV}_{[\text{ELMo}]}$, respectively. In addition, we measure how each of the parameters impacts the effectiveness by taking $\text{LPV}_{[\text{W2V}]}$ as an instance.

A. Experimental Settings

1) Log datasets: In addition to the two supercomputer log datasets (*BGL* and *HPC*) mentioned in Table I, we also use another dataset of HDFS logs (denoted by *HDFS*), which were generated by a Hadoop cluster set up on 203 EC2 nodes, through running sample Hadoop map-reduce jobs for 48 hours [27]. Some key features about these three log datasets are given in Table II, in which $\#$ *log messages* is the total number of original log messages, and $\#$ *ground truth templates* is the number of ground truth templates. The ground truth templates of *BGL* and *HPC* are manually extracted and provided online[2] by the authors of IPLoM, and that of *HDFS* are provided online[3] by the LogPAI team. Moreover, for each of these log datasets, 2,000 log messages (together with everyone's ground truth template) are provided online[3] by the LogPAI team, which are randomly sampled from the entire log dataset yet retain the key properties such as event redundancy and event variety [42].

2) Baselines: We use three state-of-the-art log parsing methods (*i.e.*, Drain [41], Spell [25], and Logram [45]) as the baselines. Drain utilizes a fixed-depth parse tree to guide log template search. It traverses the tree by log message length and preceding tokens to a leaf node, and selects the final template by token similarity, or creates a new template if no suitable one is found. Spell makes use of the following idea: for log messages output by the same "print" statement, the constants often take a major part, and the longest common subsequence (LCS) of tokens is very likely to be the log template. To improve efficiency, Spell employs an inverted list and a prefix tree to avoid matching with all existing strings. Logram leverages n-gram dictionaries to parse log messages efficiently. It first generates an n-gram dictionary which records the frequency of each unique n-gram in the log dataset (n=2,3). Then, for each log message, if the frequency of a 3-gram in it is less than a threshold, then this 3-gram may contain dynamically generated tokens and is further transformed into 2-grams. Afterward, for these 2-grams, if the frequency of a 2-gram is less than another threshold, it is considered to contain dynamically generated tokens. Finally, dynamically generated tokens are identified from those low-appearing 2-grams, and are replaced with wildcards to generate the log template.

3) Evaluation metrics: We use four metrics (*i.e.*, *Accuracy*, *Precision*, *Recall*, and *F-measure*) to measure the effectiveness of LPV's offline log parsing. *Accuracy* is defined as the ratio of original log messages which are correctly parsed; *Precision* is the ratio of generated log templates which are the same as the ground truth; *Recall* is the ratio of ground truth templates which are correctly figured out; and *F-measure* is defined as follows:

$$F\text{-}measure = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

Additionally, we use the time cost of parsing each entire log dataset to measure the efficiency of LPV's offline log parsing.

Because LPV's online log parsing is based upon the outputs of its offline log parsing, so the effectiveness of LPV's online log parsing is expected to be the same as that of its offline log parsing. We verify this by testing whether LPV's online

[2] https://web.cs.dal.ca/~makanju/iplom/
[3] https://github.com/logpai/logparser

TABLE III
DEFAULT VALUES OF LPV[BoW]'S PARAMETERS

Parameter	τ_d	τ_r
Default Value	1.5	0.5

TABLE IV
DEFAULT VALUES OF LPV[W2V]'S PARAMETERS

Parameter	τ_d	τ_r	*ES*	*WS*	*epochs*	*NNS*
Default Value	1.2	0.5	24	5	110	25

log parsing is capable of retaining the effectiveness of its offline log parsing, which is accomplished by picking out the 2,000 log messages provided by the LogPAI team (mentioned in Section IV-A1) from each log dataset and checking the percentage of log messages whose true templates are among the top 3 candidates. In addition, we measure the efficiency of LPV's online log parsing by the time cost of parsing these 2,000 log messages online.

4) Parameter setting: There are no parameters in LPV's online log parsing. In the offline log parsing, there are two thresholds: the threshold τ_d in the Clustering phase (cf. Section III-B3), which refers to the minimum inter-cluster distance allowed; and the threshold τ_r in the Intra-cluster Merge of the Log Templates Extraction phase (cf. Section III-B4), which refers to the allowed minimum value in terms of $\frac{\#\ constants}{\#\ columns}$ to merge candidate log templates in a cluster, $\#\ constants$ is the number of constants selected and $\#\ columns$ is the number of columns in the cluster. In addition, for *word2vec*, we have the following 4 key parameters: the embedding size (denoted by *ES*), the window size (denoted by *WS*), the number of epochs to train (denoted by *epochs*), and the number of negative samples per training example (denoted by *NNS*); for *ELMo*, we have the following 3 key parameters: the LSTM hidden state size (denoted by *LSTMHS*), the LSTM output size (denoted by *LSTMOS*), and the number of epochs to train (denoted by *epochs*).

The default values of these parameters are given in Tables III, IV, and V. The way we get these default values is as follows: for each group of parameters corresponding to a vectorization method, we first give each parameter an empirical value, then we tune one parameter at a time by varying its value while keeping the others unchanged, after this we set the parameter to the value which leads to the best effectiveness, and go on with another parameter. We remark that, since we aim to get a unified value for each parameter across all datasets, so we did not choose the parameter values that lead to the best effectiveness for each dataset individually, but the ones which lead to outstanding effectiveness on all datasets. We will explore how the parameters impact the effectiveness in Section IV-D, by taking LPV[W2V] as an example.

5) Experimental environment: We conducted our experiments on a Linux Server, which is equipped with a 24-core Intel(R) Xeon(R) CPU E5-2678 v3 @ 2.50GHz, 128GB DDR4 RAM @ 2400 MHz, and NVIDIA GeForce GTX 1080 Ti 11G GPUs. The Linux distribution is Ubuntu 16.04.5 LTS

TABLE V
DEFAULT VALUES OF LPV[ELMo]'S PARAMETERS

Parameter	τ_d	τ_r	*LSTMHS*	*LSTMOS*	*epochs*
Default Value	20	0.5	96	24	600

64-bit. We implement LPV with Python 3.5.2 and TensorFlow 1.10.1 with GPU support.

B. Effectiveness

As mentioned earlier, LPV's online log parsing is expected to have the same effectiveness as its offline log parsing. For ease of presentation, we first evaluate the effectiveness of LPV's offline log parsing, by comparing with three state-of-the-art log parsing methods (*i.e.*, Drain, Spell, and Logram). Then, we verify whether LPV's online log parsing retains the effectiveness of its offline log parsing.

1) The offline log parsing: Fig. 8 presents the effectiveness comparison among Drain, Spell, Logram, and LPV employing three different vectorization methods (*i.e.*, *bag-of-words*, *word2vec*, and *ELMo*) on the three log datasets (*BGL*, *HPC*, and *HDFS*). To avoid bias possibly caused by the randomness of the vectors from *word2vec* and *ELMo*, each metric value used to plot a bar of LPV[W2V] or LPV[ELMo] is the average of 5 repeated runs, and the maximum and minimum values in these runs are marked out by two short horizontal lines at the top of the bar. The reason why the accuracies of Logram reported here have a large margin with those in the original paper is that, for each log dataset, the original paper only used the 2,000 log messages provided by the LogPAI team (mentioned in Section IV-A1) to evaluate the accuracy of Logram, while we used the entire log dataset.

We can observe that, even with a simple vectorization method, LPV[BoW] achieves a competitive performance. It outperforms Drain, Spell, and Logram in three metrics (*Accuracy*, *Recall*, and *F-measure*) on *BGL*, in all four metrics on *HPC*, and in three metrics (*Accuracy*, *Precision*, and *F-measure*) on *HDFS*. What's more, LPV[BoW] achieves the best *Recall* on *BGL*, and the best *Accuracy*, *Precision*, and *F-measure* on *HDFS*. These indicate that, log templates of these three log datasets can be effectively distinguished just by the tokens contained in each template.

On the whole, LPV[W2V] has the best effectiveness on *BGL*: it achieves the best *Accuracy*, *Precision*, and *F-measure*, and its *Recall* is the second best. On *HPC*, LPV[W2V] outperforms Drain, Spell, and Logram in all four metrics, and is on par with LPV[BoW] and LPV[ELMo]. On *HDFS*, LPV[W2V] has the best *Accuracy*, and is competitive with the best ones in the other three metrics.

As to LPV[ELMo], its effectiveness is not so attractive on *BGL*, but it is still competitive with Spell and is much better than Drain and Logram in all four metrics. On *HPC*, LPV[ELMo] has obvious fluctuations in *Precision*, *Recall*, and *F-measure*. Nevertheless, the averages are better than the others, and the lower bounds are still better than the corresponding metric values of Drain, Spell, and Logram. On *HDFS*, LPV[ELMo] has the best *Accuracy*, *Precision*, *F-measure* and the second

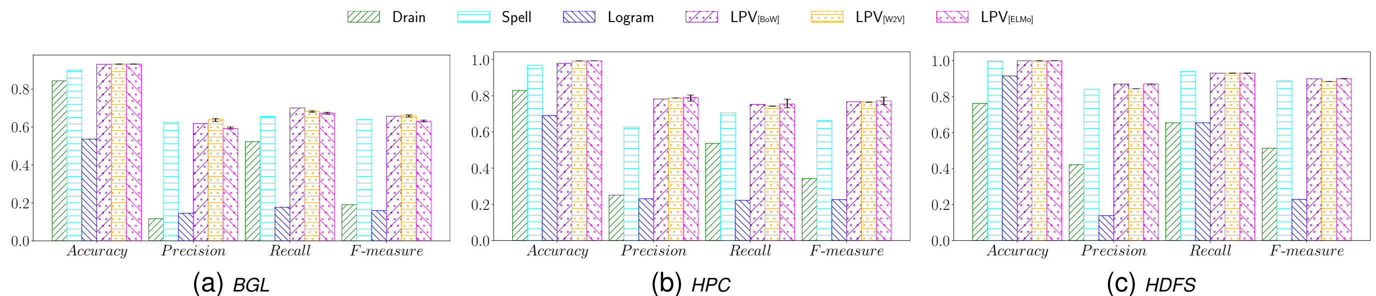


Fig. 8. Effectiveness comparison among Drain, Spell, Logram, and LPV employing three different vectorization methods, on the three log datasets.

TABLE VI
EFFECTIVENESS VERIFICATION OF LPV'S ONLINE LOG PARSING

		BGL	*HPC*	*HDFS*
LPV[BoW]	No Match	206	96	0
	Top 1	1794(100%)	1903(99.95%)	2000 (100%)
	Top 3	1794(100%)	1904(100%)	2000 (100%)
LPV[W2V]	No Match	199	84	0
	Top 1	1793 (99.56%)	1915 (99.95%)	2000 (100%)
	Top 3	1801 (100%)	1916 (100%)	2000 (100%)
LPV[ELMo]	No Match	202	84	0
	Top 1	1790 (99.56%)	1913 (99.84%)	2000 (100%)
	Top 3	1798 (100%)	1916 (100%)	2000 (100%)

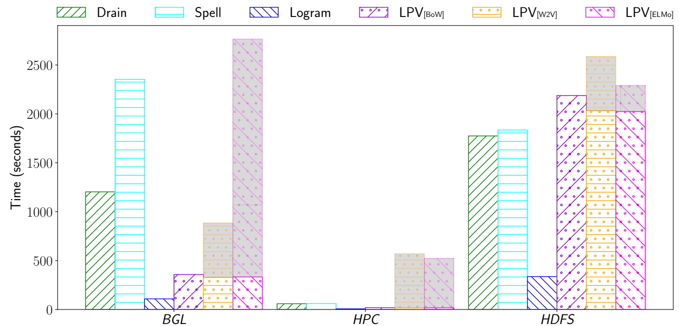


Fig. 9. Time cost comparison of parsing each entire log dataset among Drain, Spell, Logram, and LPV employing three different vectorization methods. The upper bars with shadow stand for the model training time.

best $Recall$. *ELMo* uses a complicated language model, and the results here make us to believe that the three log datasets are not large enough to train the model well.

All in all, the effectiveness results aforementioned validate the feasibility and superiority of our framework to employ vectorization methods in offline log parsing.

2) The online log parsing: To verify whether LPV's online log parsing retains the effectiveness of its offline log parsing, for each log dataset, we operate as follows: first, we pick out the 2,000 log messages provided by the LogPAI team (mentioned in Section IV-A1) from the dataset, and perform offline log parsing with the rest log messages; then, we perform online log parsing with the 2,000 log messages, and check for each log message whether its ground truth template is among the top 3 candidates. Table VI gives the results of LPV[BoW], LPV[W2V], and LPV[ELMo]. In this table, 'No Match' stands for the number of log messages, whose ground truth templates are not generated in the offline log parsing; and 'Top n'(n=1, or 3) refers to the number (and percentage) of log messages, for each of which the ground truth template is among the top n candidates.

We can see that, for LPV[BoW], LPV[W2V], and LPV[ELMo] on each of the three log datasets, except for the log messages whose ground truth templates are not generated in the offline log parsing, the others' ground truth templates are all among the top 3 candidates, and for more than 99% of them, the ground truth template is exactly the top 1 candidate.

Therefore, it can be asserted that, LPV's online log parsing completely retains the effectiveness of its offline log parsing, and often needs to match the incoming log message only with the top 1 or top 3 candidate template(s), instead of matching with all existing log templates, which is beneficial to improving the parsing efficiency. What's more, the high percentage of 'Top 1' (more than 99%) indicates that, it is feasible in online log parsing to measure the similarity between an incoming log message and a log template by the distance between their corresponding vectors.

C. Efficiency

1) The offline log parsing: Fig. 9 gives the time cost comparison of parsing each entire log dataset among Drain, Spell, Logram, and LPV employing three different vectorization methods (*i.e.*, *bag-of-words*, *word2vec*, and *ELMo*). Each number used to plot a bar is the average time cost of 5 repeated runs. And for LPV[W2V] and LPV[ELMo], the model training time on each dataset is marked out by the upper bar with shadow.

We can see that Logram is the fastest on all the three log datasets, which owes to its n-gram dictionaries kept in memory. But its relatively low effectiveness ($Accuracy$, $Precision$, $Recall$, and $F\text{-}measure$) on each log dataset (cf. Fig. 8) indicates that, simply using the frequencies of n-grams may not perform well when parsing large log datasets. LPV[BoW] is much faster than Drain and Spell on *BGL* and *HPC*, and is a little bit slower than them on *HDFS*. As to LPV[W2V] and LPV[ELMo], if the model training time is excluded, which is possible if there are token vectors off the shelf provided by others, just like in the NLP area, then they can be even faster than LPV[BoW]. All these show the competitive efficiency of LPV's offline log parsing.

2) The online log parsing: For LPV, we operate as in Section IV-B2. For Drain, Spell, and Logram, we run them with the 2,000 log messages provided by the LogPAI team (mentioned in Section IV-A1) as input for each log dataset.

TABLE VII
TIME COST COMPARISON OF PARSING 2,000 LOG MESSAGES ONLINE AMONG DRAIN, SPELL, LOGRAM, AND LPV EMPLOYING THREE DIFFERENT VECTORIZATION METHODS (UNIT: SECONDS)

	BGL	*HPC*	*HDFS*
Drain	0.2765	0.2544	0.2805
Spell	0.3200	0.2770	0.3094
Logram	*0.0719*	*0.0402*	*0.0890*
LPV[BoW]	1.3183	0.2624	0.3734
LPV[W2V]	**0.2365**	**0.1334**	**0.2593**
LPV[ELMo]	10.7195	3.8558	4.3452

Table VII gives the time cost comparison of parsing these 2,000 log messages online among Drain, Spell, Logram, and LPV employing three different vectorization methods (*i.e.*, *bag-of-words*, *word2vec*, and *ELMo*), and each number in the table is the average time cost of 10 repeated runs.

It can be seen that, Logram is again the fastest on all the three log datasets. Except for Logram, on each of the three log datasets, LPV[W2V] is the fastest, and its average time cost of parsing a log message is less than 0.2 milliseconds. LPV[BoW] is competitive with Drain and Spell on *HPC* and *HDFS*, but is slower than them on *BGL*. For the *bag-of-words* model, the vector dimension is the vocabulary size, which is in hundreds for *HPC* and *HDFS*, while in thousands for *BGL*. The lower efficiency of LPV[BoW] to that of LPV[W2V] demonstrates the advantage of employing lower-dimensional vectors in log parsing. LPV[ELMo] is much slower than the others, because for each incoming log message, it computes representations for the tokens of the log message on the fly, using the pre-trained model from the offline log parsing, which is rather computationally expensive.

D. Impacts of Different Parameters

To evaluate how each of the parameters influences the effectiveness of LPV, we take LPV[W2V] as an instance, and vary one parameter while keeping the others with the default values listed in Table IV, since LPV[W2V] has outstanding effectiveness and higher efficiency than LPV[BoW] and LPV[ELMo] on all the three log datasets. The value changes of the four metrics ($Accuracy$, $Precision$, $Recall$, and $F\text{-}measure$) on the three log datasets (*BGL, HPC,* and *HDFS*) when varying different parameters are shown in Figs. 10, 12∼16. Again, to avoid bias possibly caused by the randomness of the vectors, all metric values used to plot these figures are the average of 5 repeated runs.

1) Impact of τ_d: Fig. 10 shows the effect when varying parameter τ_d. It can be seen that, on *HDFS*, the performance keeps stable; while on *BGL* and *HPC*, the performance is not so good when τ_d is too small or too big. Intuitively, a small τ_d may partition log messages having the same log template into different clusters, while a large τ_d may partition log messages having different templates into a single cluster, both do harm to the performance. However, the fluctuation of the performance is somewhat mild, which validates the effective design of the three steps (*i.e.*, Partition, Intra-cluster Merge, and Inter-cluster Merge) in the Log Templates Extraction phase (cf. Section

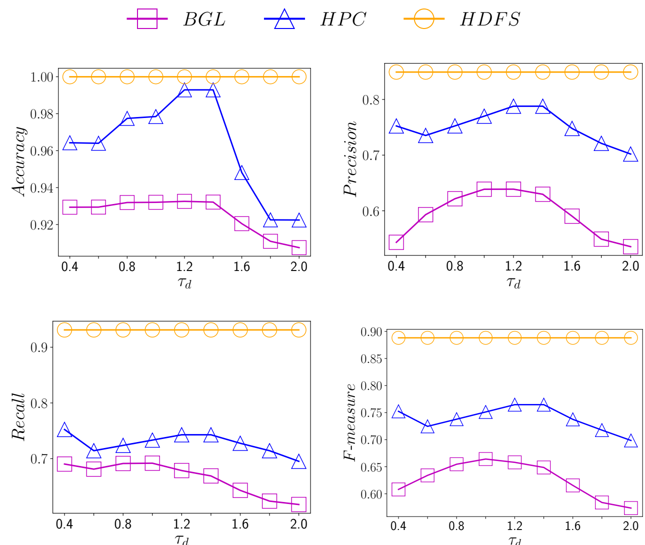


Fig. 10. Impact of τ_d on LPV[W2V]'s effectiveness.

III-B4). To explore this deeper, Fig. 11 shows the number of clusters changed in various stages throughout the offline log parsing, with different τ_d's, where $\#\ initial\ clusters$ is the number of clusters formed after the Clustering phase, $\#\ clusters\ partitioned$ and $\#\ clusters\ merged$ are the number of clusters "partitioned into smaller clusters" and the number of clusters merged into larger clusters (including Intra-cluster Merge and Inter-cluster Merge) respectively, and $\#\ final\ clusters$ is the number of final clusters, also the number of templates generated, since a log template is extracted from each of the final clusters. We can observe that the Log Templates Extraction phase keeps the number of final clusters from changing sharply, weakening the influence of τ_d to some extent. Especially for *HDFS*, the number of final clusters keeps the same when varying τ_d, which leads to a stable performance.

2) Impact of τ_r: From Fig. 12, we can see that LPV[W2V]'s effectiveness is sensitive to parameter τ_r. As mentioned in Section IV-A4, τ_r is the lower bound of $\frac{\#\ constants}{\#\ columns}$ to merge candidate log templates in a cluster. If it is too small, then less tokens in a log template have to be constant, causing more log messages to share the same log template, which is too general. If it is too big, then more tokens in a log template have to be constant, causing less log messages to have the same log template, which is too specific. Both cases lead to poor effectiveness. This correlation is not so strong on *HDFS*, this is because there are only 29 ground truth templates in *HDFS*, and it is relatively easy to distinguish these templates.

3) Impact of ES: Fig. 13 shows the impact of parameter ES. We can see that, the performance is stable on the whole except when ES is too small, in which case the vectors are not able to express and distinguish different tokens effectively, especially for *BGL* and *HPC*. But if ES is too big, the performance may slightly drop down, this is because a larger embedding size introduces more randomness, thus more training is needed to get stable vectors. Nevertheless, from the figure we can see that, it is feasible to represent tokens in a large

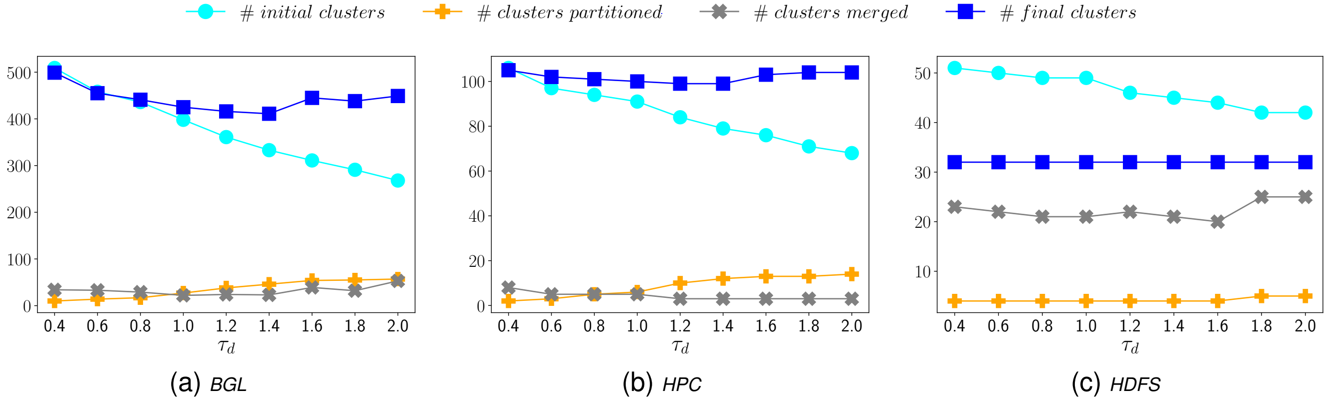


Fig. 11. Variation of the numbers of clusters in different stages when varying τ_d.

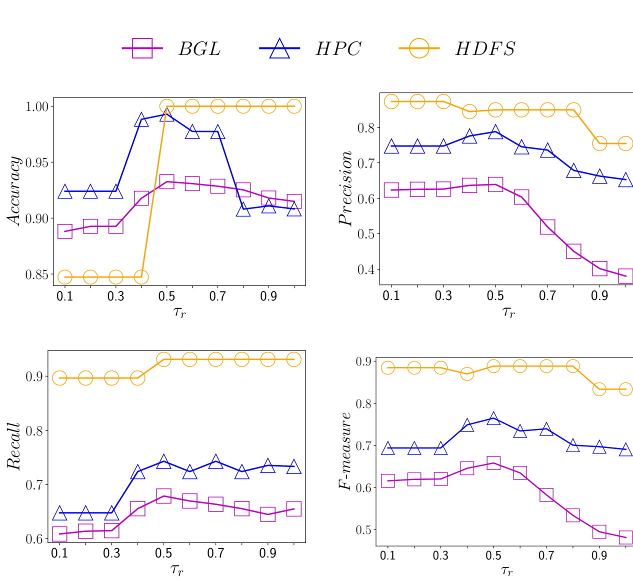


Fig. 12. Impact of τ_r on LPV[W2V]'s effectiveness.

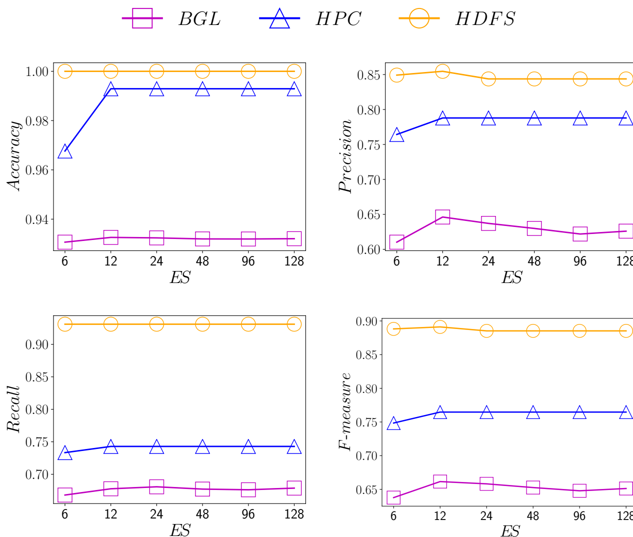


Fig. 13. Impact of *ES* on LPV[W2V]'s effectiveness.

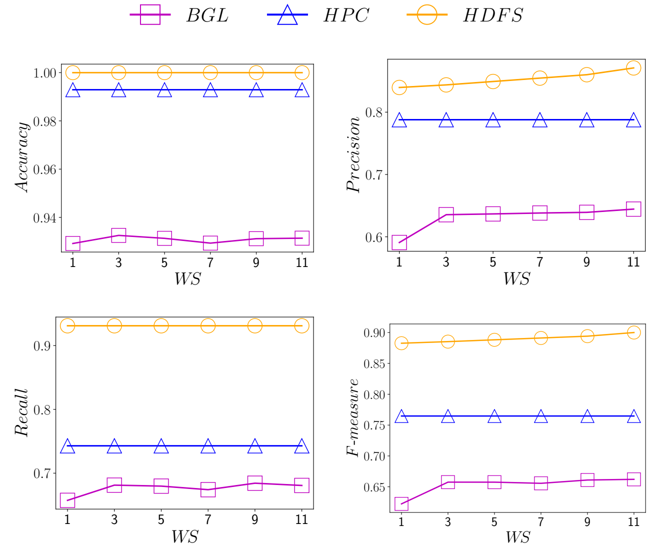


Fig. 14. Impact of *WS* on LPV[W2V]'s effectiveness.

log dataset with low-dimensional vectors, which is beneficial to the parsing efficiency, because lower-dimensional vector arithmetic is less computationally expensive. In addition, this confirms the validity of the operations in the Preprocessing phase of the offline log parsing to reduce the size of the dataset and vocabulary (cf. Section III-B1).

4) Impact of WS: For parameter *WS*, a small value degrades the performance, as shown in Fig. 14. It can be easily understood that, a larger window size can capture syntactic and semantic word relationships better, since more neighbors are utilized to encode each token into a vector; and a small window size may lose some significant context information, harming the quality of the vectors. But the impact of *WS* on the effectiveness is somewhat subtle, meaning that there are not very strong relationships between neighbor tokens in a log message. Indeed, log messages are commonly free texts without any mandatory schemas or grammar rules, since developers can almost output any strings into log files. This is also confirmed by the fine effectiveness of LPV[BoW].

5) Impact of epochs: As to parameter *epochs*, we find that if it is too small, then the performance is not so good, as shown in Fig. 15. This is mainly due to that, the syntactic

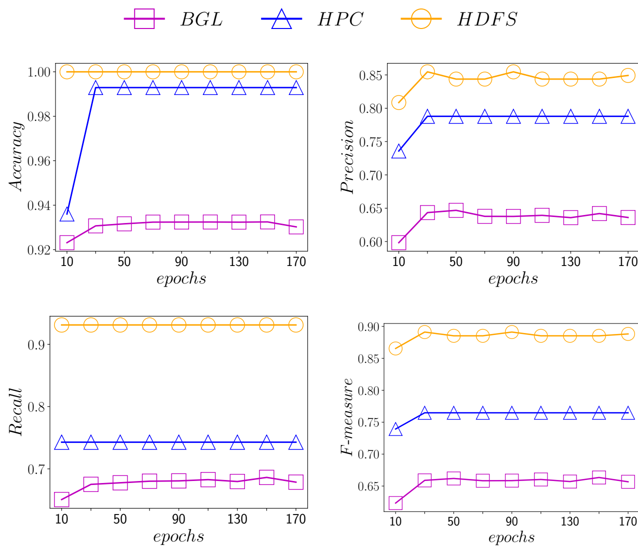


Fig. 15. Impact of *epochs* on LPV[W2V]'s effectiveness.

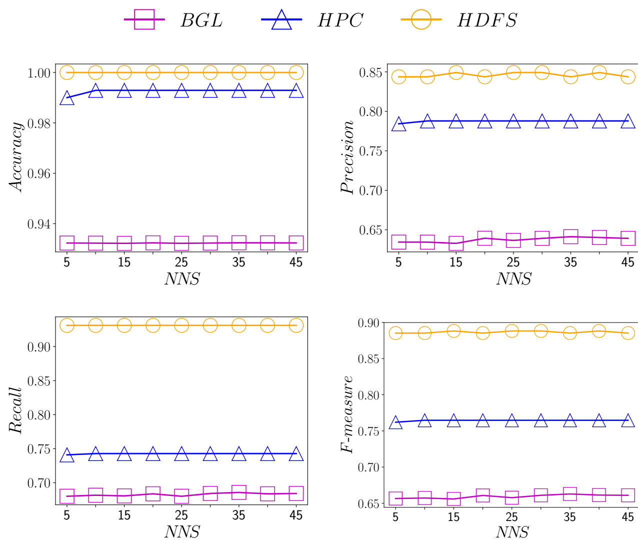


Fig. 16. Impact of *NNS* on LPV[W2V]'s effectiveness.

and semantic word relationships may not be well captured by the tokens' vectors, when the model is without sufficient training. But a bigger *epochs* does not necessarily lead to better performance, since the performance would be stable after enough epochs of training.

6) Impact of NNS: Fig. 16 indicates that parameter *NNS* has little impact on LPV[W2V]'s effectiveness, which validates the use of negative sampling in *word2vec*, and indicates that a small *NNS* is enough to get good effectiveness on the three log datasets.

The impacts of the four key parameters of *word2vec* (*ES*, *WS*, *epochs*, and *NNS*) on the effectiveness of LPV[W2V] verify that, the quality of token vectors does have impact on the result of this log parsing method based on vectorization, which indicates that we can get better effectiveness through higher-quality vectors.

It should be noted that, *Accuracy* is defined based on the number of original log messages, while *Precision*, *Recall*, and *F-measure* are defined based on the number of log templates. Therefore, they do not necessarily follow the same trend, because the number of log messages sharing a log template may differ greatly for different log templates.

V. CONCLUSION

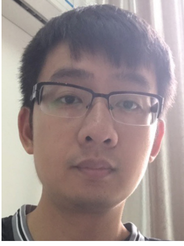
In this paper, we proposed a novel log parsing framework based on vectorization, which is called LPV, and implemented with three different vectorization methods (*i.e.*, *bag-of-words*, *word2vec*, and *ELMo*). Different from prior log parsing methods, LPV represents log messages and log templates with vectors, so that the similarity between two log messages or between a log message and a log template can be measured by the distance between two vectors. LPV incorporates offline and online log parsing. More specifically, in the offline log parsing, LPV clusters log messages via clustering their corresponding vectors, and then extracts log templates from the resultant clusters; in the online log parsing, LPV picks out several closest (most possible) log templates for each incoming log message in terms of vector distance, and accelerates log parsing by matching the incoming log message only with these selected templates, instead of with all existing log templates. Since the online log parsing is based upon the outputs of the offline log parsing, so LPV enables us to make full use of history logs to achieve high effectiveness in the offline log parsing, and ensure high efficiency in the online log parsing at the same time. We have conducted comprehensive experiments based on three public log datasets, which have been widely used by previous works for evaluation, and the results validate the feasibility and competitiveness of our framework.

REFERENCES

[1] 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*. IEEE, 2020, pp. 1346–1351.
[2] A. Oliner, A. Ganapathi, and W. Xu, "Advances and challenges in log analysis," *Commun. ACM*, vol. 55, no. 2, pp. 55–61, 2012.
[3] R. Vaarandi and M. Pihelgas, "Logcluster - A data clustering and pattern mining algorithm for event logs," in *Proc. CNSM*. IEEE Computer Society, 2015, pp. 1–7.
[4] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Online system problem detection by mining patterns of console logs," in *Proc. ICDM*. IEEE, 2009, pp. 588–597.
[5] K. Nagaraj, C. Killian, and J. Neville, "Structured comparative analysis of systems logs to diagnose performance problems," in *Proc. NSDI*. USENIX Association, 2012, pp. 353–366.
[6] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with csight," in *Proc. ICSE*. ACM, 2014, pp. 468–479.
[7] S. He, J. Zhu, P. He, and M. R. Lyu, "Experience report: System log analysis for anomaly detection," in *Proc. ISSRE*. IEEE Computer Society, 2016, pp. 207–218.
[8] C. Bertero, M. Roy, C. Sauvanaud, and G. Tredan, "Experience report: Log mining using natural language processing and application to anomaly detection," in *Proc. ISSRE*. IEEE Computer Society, 2017, pp. 351–360.
[9] A. Borghesi, A. Bartolini, M. Lombardi, M. Milano, and L. Benini, "Anomaly detection using autoencoders in high performance computing systems," in *Proc. AAAI*. AAAI Press, 2019, pp. 9428–9433.
[10] A. J. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *Proc. DSN*. IEEE Computer Society, 2007, pp. 575–584.
[11] L. Yu, Z. Zheng, Z. Lan, T. Jones, J. M. Brandt, and A. C. Gentile, "Filtering log data: Finding the needles in the haystack," in *Proc. DSN*. IEEE Computer Society, 2012, pp. 1–12.

[12] H. Mi, H. Wang, Y. Zhou, M. R.-T. Lyu, and H. Cai, "Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 6, pp. 1245–1255, 2013.
[13] X. Xu, L. Zhu, I. Weber, L. Bass, and D. Sun, "Pod-diagnosis: Error diagnosis of sporadic operations on cloud applications," in *Proc. DSN*. IEEE Computer Society, 2014, pp. 252–263.
[14] Q. Fu, J. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *Proc. ICDM*. IEEE Computer Society, 2009, pp. 149–158.
[15] J. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, "Mining invariants from console logs for system problem detection," in *Proc. USENIX ATC*. USENIX Association, 2010, pp. 1–14.
[16] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proc. CCS*. ACM, 2017, pp. 1285–1298.
[17] A. Das, F. Mueller, P. Hargrove, E. Roman, and S. B. Baden, "Doomsday: Predicting which node will fail when on supercomputers," in *Proc. SC*. IEEE, 2018, pp. 9:1–9:14.
[18] Y. Xu, K. Sui, R. Yao, H. Zhang, Q. Lin, Y. Dang, P. Li, K. Jiang, W. Zhang, J. Lou, M. Chintalapati, and D. Zhang, "Improving service availability of cloud systems by predicting disk error," in *Proc. USENIX ATC*. USENIX Association, 2018, pp. 481–494.
[19] 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*. IEEE, 2016, pp. 1291–1300.
[20] 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*. International Joint Conferences on Artificial Intelligence Organization, 2019, pp. 4739–4745.
[21] 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.
[22] C. Zhang, X. Wang, H. Zhang, H. Zhang, and P. Han, "Log sequence anomaly detection based on local information extraction and globally sparse transformer model," *IEEE Trans. Netw. Serv. Manag.*, vol. 18, no. 4, pp. 4119–4133, 2021.
[23] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, "Towards automated log parsing for large-scale log data analysis," *IEEE Trans. Dependable Secure Comput.*, vol. 15, no. 6, pp. 931–944, 2018.
[24] M. Du and F. Li, "Spell: Streaming parsing of system event logs," in *Proc. ICDM*. IEEE Computer Society, 2016, pp. 859–864.
[25] ——, "Spell: Online streaming parsing of large unstructured system logs," *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 11, pp. 2213–2227, 2019.
[26] P. He, J. Zhu, S. He, J. Li, and M. R. Lyu, "An evaluation study on log parsing and its use in log mining," in *Proc. DSN*. IEEE Computer Society, 2016, pp. 654–661.
[27] W. Xu, L. Huang, A. Fox, D. A. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proc. SOSP*. ACM, 2009, pp. 117–132.
[28] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "Clustering event logs using iterative partitioning," in *Proc. ACM SIGKDD*. ACM, 2009, pp. 1255–1264.
[29] ——, "A lightweight algorithm for message type extraction in system application logs," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 11, pp. 1921–1936, 2012.
[30] L. Tang and T. Li, "Logtree: A framework for generating system events from raw textual logs," in *Proc. ICDM*. IEEE Computer Society, 2010, pp. 491–500.
[31] L. Tang, T. Li, and C. Perng, "Logsig: generating system events from raw textual logs," in *Proc. CIKM*. ACM, 2011, pp. 785–794.
[32] K. Bringmann and M. Künnemann, "Multivariate fine-grained complexity of longest common subsequence," in *Proc. SODA*. SIAM, 2018, pp. 1216–1235.
[33] Y. Zhang, R. Jin, and Z. Zhou, "Understanding bag-of-words model: a statistical framework," *Int. J. Machine Learning & Cybernetics*, vol. 1, no. 1-4, pp. 43–52, 2010.
[34] J. Stearley, "Towards informatic analysis of syslogs," in *Proc. CLUSTER*. IEEE Computer Society, 2004, pp. 309–318.
[35] M. Aharon, G. Barash, I. Cohen, and E. Mordechai, "One graph is worth a thousand logs: Uncovering hidden structures in massive system event logs," in *Proc. ECML PKDD*. Springer, 2009, pp. 227–243.
[36] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Proc. ICLR Workshop*, 2013.
[37] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. NeurIPS*, 2013, pp. 3111–3119.
[38] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," in *Proc. NAACL*, 2018.
[39] 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*. ACM, 2019, pp. 807–817.
[40] S. Zhang, W. Meng, J. Bu, S. Yang, Y. Liu, D. Pei, J. Xu, Y. Chen, H. Dong, X. Qu, and L. Song, "Syslog processing for switch failure diagnosis and prediction in datacenter networks," in *Proc. IWQoS*. IEEE, 2017, pp. 1–10.
[41] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *Proc. ICWS*. IEEE, 2017, pp. 33–40.
[42] 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.
[43] T. Qiu, Z. Ge, D. Pei, J. Wang, and J. Xu, "What happened in my network: Mining network events from router syslogs," in *Proc. IMC*. ACM, 2010, pp. 472–484.
[44] T. Kimura, K. Ishibashi, T. Mori, H. Sawada, T. Toyono, K. Nishimatsu, A. Watanabe, A. Shimoda, and K. Shiomoto, "Spatio-temporal factorization of log data for understanding network events," in *Proc. INFOCOM*. IEEE, 2014, pp. 610–618.
[45] 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, 2020.
[46] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *Proc. IPOM*. IEEE, 2003, pp. 119–126.
[47] ——, "A breadth-first algorithm for mining frequent patterns from event logs," in *Proc. INTELLCOMM*. Springer, 2004, pp. 293–308.
[48] T. Kimura, A. Watanabe, T. Toyono, and K. Ishibashi, "Proactive failure detection learning generation patterns of large-scale network logs," in *Proc. CNSM*. IEEE, 2015, pp. 8–14.
[49] H. Hamooni, B. Debnath, J. Xu, H. Zhang, G. Jiang, and A. Mueen, "Logmine: Fast pattern recognition for log analytics," in *Proc. CIKM*. ACM, 2016, pp. 1573–1582.
[50] A. Gainaru, F. Cappello, S. Trausan-Matu, and B. Kramer, "Event log mining tool for large scale HPC systems," in *Proc. Euro-Par*. Springer, 2011, pp. 52–64.
[51] S. Messaoudi, A. Panichella, D. Bianculli, L. Briand, and R. Sasnauskas, "A search-based approach for accurate identification of log message formats," in *Proc. ICPC*. ACM, 2018, pp. 167–177.
[52] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, 2002.
[53] A. Großwendt and H. Röglin, "Improved analysis of complete-linkage clustering," *Algorithmica*, vol. 78, no. 4, pp. 1131–1150, 2017.

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 working toward the PhD degree at the College of Computer Science and Electronic Engineering, Hunan University, Changsha, China. His research interests include log analysis, anomaly detection, failure prediction, data mining, and deep learning.



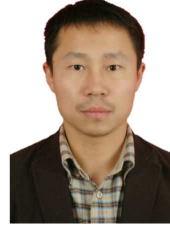
Zhe Quan received the PhD degree in computer science from the University de Picardie Jules Verne, France. He is currently a professor at the College of Computer Science and Electronic Engineering, Hunan University (HNU), Changsha, China. Before joining HNU, he worked at the National University of Defense Technology, Changsha, China, and worked as a postdoctoral research fellow at the Berkeley and Livermore Lab of the University of California, United States. His main research interests include machine learning, artificial intelligence, parallel and high-performance computing, *etc.* He has published a set of research papers in venues such as IEEE TPDS, AAAI, IJCAI, ICSC, BIBM, *etc.*

Zhi-Jie Wang 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 data mining, artificial intelligence, databases, machine learning, *etc.* He has published a set of research papers in these fields including IEEE TKDE, IEEE TPDS, IEEE TMM, IEEE TALSP, IEEE TCSS, IJCAI, AAAI, ICDM *etc.* He is a member of IEEE, ACM, and CCF.

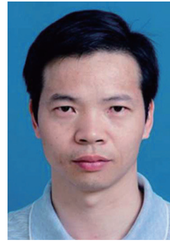
Kaiqi Zhao received the PhD degree from the School of Computer Science and Engineering, Nanyang Technological University, Singapore. He is currently an assistant professor at the School of Computer Science, the University of Auckland, Auckland, New Zealand. Prior to that, he worked as a Research Fellow at the Singtel Cognitive and Aritificial Intelligence Lab for Enterprise, Nanyang Technological University, Singapore. He has published over 20 research papers in venues including SIGMOD, VLDB, ICDE, KDD, EDBT, AAAI, CIKM, VLDB Journal, *etc.* His current research interests include spatio-temporal data mining, mining social media, text mining and knowledge discovery, and machine learning.

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.

Huang Huang received the BSc degree from the School of Information Science and Engineering, Harbin Institute of Technology, in 2008, the MSc degree from the College of Engineering, Shantou University, in 2011, and the PhD degree from the College of Computer, National University of Defense Technology, under the supervision of Prof. Y. T. Lu. He is currently a post-doctor at the College of Computer Science and Electronic Engineering, Hunan University. His research interests include large-scale distributed storage, parallel I/O optimization, fault information collection and prediction.



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 focus on 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, ICCAD, *etc.*



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 Computer Science and Electronic Engineering with Hunan University, Changsha, China, and also the Deputy Director of the National Supercomputing Center, Changsha. 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.