LPV: A Log Parser Based on Vectorization for Offline and Online Log Parsing

Tong Xiao[1], Zhe Quan[1,*], Zhi-Jie Wang[2,*], Kaiqi Zhao[3], and Xiangke Liao[4]
[1]College of Computer Science and Electronic Engineering, Hunan University, Changsha, China
[2]College of Computer Science, Chongqing University, Chongqing, China
[3]School of Computer Science, University of Auckland, Auckland, New Zealand
[4]College of Computer Science and Technology, National University of Defense Technology, Changsha, China
{xiaotong18, quanzhe}@hnu.edu.cn, cszjwang@yahoo.com, kaiqi.zhao@auckland.ac.nz, xkliao@nudt.edu.cn

***Abstract*—As the first and foremost step of typical automatic log analysis, log parsing has attracted a lot of interest. Most of existing studies treat log messages as pure strings and rely on string matching or string distance. In NLP, *word2vec* has shown very efficient and effective in representing words with low dimensional vectors. Inspired by this, in this paper we propose a novel method, called LPV (Log Parser based on Vectorization), for both offline and online log parsing. The central idea of our method in offline log parsing is to first convert log messages into vectors, and measure the similarity between two log messages by the distance between two vectors, then log messages can be clustered via clustering the vectors, and log templates can be extracted from the resulting clusters. For online log parsing, we also assign log templates with some kind of average vectors, so that the similarity between an incoming log message and each log template can also be measured by the distance between two vectors. We have conducted extensive experiments based on three widely used log datasets, and the results demonstrate that our proposed method LPV can achieve a competitive performance, compared against state-of-the-art log parsing methods.**

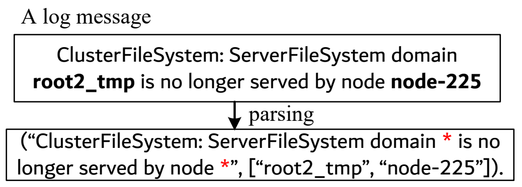
***Index Terms*—log parsing, log template extraction, vectorization, clustering**

I. Introduction

Logs contain rich information and play an important role in the life cycle of computing systems. However, with the increasing scale and complexity of modern computing systems, the volume of logs explodes, which makes it cumbersome, error-prone, and impractical to cope with the huge amount of logs and dig out useful information manually, especially for large-scale HPC and cloud systems. Therefore, automatic log analysis is urgently needed, since it can alleviate the above dilemmas and ease many tasks such as anomaly and failure detection, diagnosis, and prediction [1]–[4].

Typically, the first and foremost step of automatic log analysis is to parse unstructured logs into structured data, since logs are traditionally unstructured plain text produced by the "`printf`" or similar statements in C and equivalents in other programming languages [5]–[8]. As an example, consider the log message given in Fig. 1. It can be split into constant parts and variable parts (the texts in bold). The log parsing process usually needs to obtain a log template which can be achieved by replacing the variable parts with wildcards (the * in red color).

* The corresponding author.


A log message

ClusterFileSystem: ServerFileSystem domain **root2_tmp** is no longer served by node **node-225**

("ClusterFileSystem: ServerFileSystem domain * is no longer served by node *", ["root2_tmp", "node-225"]).

Fig. 1. Example of parsing a log message. The parsing result is represented by a tuple in the form of (log template, [list of variables]).

However, it is often difficult to determine which parts are constants (resp., variables) in a log message. In existing literatures, there are roughly three types of methods. The first one relies on domain experts to inspect logs and provide rules, which is labor-intensive and tedious. The second leverages source codes to obtain the "print" statement of a log message [8], which can get accurate log templates, but relies on the availability of source codes. The third type of methods [5], [6], [9]–[13] utilize machine learning and data mining techniques to figure out the constant parts automatically, which need little domain knowledge and no access to source codes.

It is possible that some researchers might have attempted to covert log messages into vectors and apply data clustering algorithms on the vectors to perform log parsing. However, there exists no literature yet that shows such a vector-based clustering method can work well for log parsing. This could be due to that, logs usually have short length while a likely large vocabulary size [14], thus it is difficult to perform log parsing via vector-based models such as *bag-of-words*, because of the *Curse of Dimensionality* [11], [12]. Hearteningly, we observed that the *word2vec* [15] in NLP provides an efficient and effective word embedding method to represent words with low-dimensional vectors, which capture syntactic and semantic word relationships well at the same time [16]. This presents a great opportunity to utilize vector-based clustering methods to perform log parsing.

Inspired by the above, in this paper we propose a novel method, dubbed as LPV (Log Parser based on Vectorization), for both offline and online log parsing. Generally, as for offline log parsing, LPV first converts textual logs into vectors with the help of *word2vec*. Then, it clusters the resulting set of vectors to get groups of logs, and extracts one or more templates from each group. Finally, it merges similar templates into a single one to obtain fine-tuned templates and their corresponding template vectors. As for online log parsing, LPV first converts each incoming log message into a vector just like in offline log parsing, and then accelerates log parsing by matching the incoming log message with only a few closest log templates, in terms of vector distance. We have conducted empirical study based on three widely used log datasets. The results show that LPV achieves a competitive performance, compared against state-of-the-art log parsing methods.

II. Related Work

Log parsing has attracted a lot of attention in past years [17]. Early works such as SLCT [18] and LogHound [19] leveraged the idea from the Apriori algorithm, which is used for mining frequent itemsets, to find frequent words and patterns in the log dataset. Their basic observation is that frequent words are more likely to be constant parts. Such methods first identify all frequent words in the log dataset, and then form different clusters according to the frequent words contained in each log message, and finally create a log pattern for each cluster. An improved approach, called LogCluster [13], was recently proposed to address shortcomings of SLCT. Nevertheless, these Apriori-based algorithms are not efficient enough.

In [9], [10], the authors proposed a lightweight algorithm called IPLoM, which first iteratively partitions the whole log dataset into respective clusters using three steps (*i.e.,* partition by token count, by token position, and by search for bijection, respectively). Then, the method generates a message type description for each cluster. IPLoM assumes that log messages having the same message type description should be having the same token count. This assumption may limit its applications.

In [11], the authors proposed LogTree, which utilizes the format and structural information of log messages via building semi-structural log messages and assigning different importance to different levels. LogTree can employ various clustering algorithms to generate log templates, but the number of clusters is a user-given parameter, which is often hard to determine. In [12], the authors developed logSig, which tries to partition all log messages into *k* groups based on the term pairs generated for each log message. Yet, the number of groups, *k*, must be specified by user, which is also difficult to determine.

Different from the above approaches, HELO [20] has an offline clustering process together with an online one. In the offline clustering process, HELO recursively finds a split column (a word position) and divides log messages into different clusters until all clusters are stable. In the online clustering process, the template set from the offline process can be adapted according to each incoming log message.

For open-source softwares, there is also effort to parse logs by leveraging source code analysis [8], as a log's schema is hidden in the log printing statement. Such a method is able to parse possible log messages accurately, but the access to source codes is a prerequisite.

With the increasing demand of online monitoring, several novel online log parsing methods have been proposed recently.

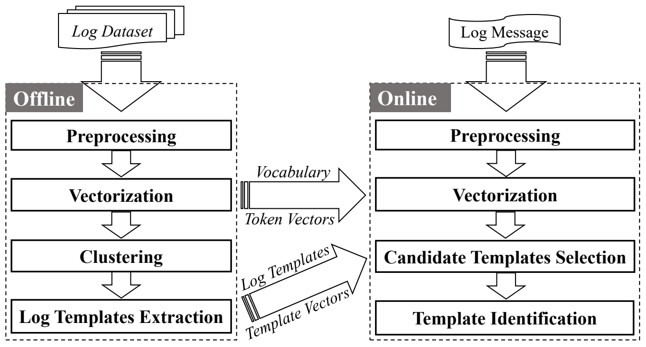


Fig. 2. Overall framework of LPV.

Spell [5], [6] is an online log parser which utilizes a longest common subsequence (LCS) based approach. Drain [21] is another online log parsing approach which utilizes a fixed depth parse tree to accelerate the log group search process.

In summary, almost all existing works merely treat log messages as strings and directly process or cluster them based on string matching or string distance.

III. Methodology

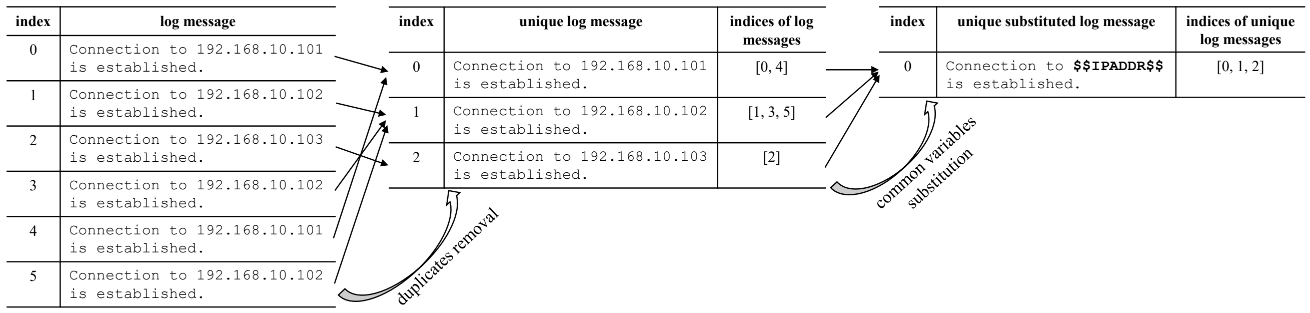
A. Overview of LPV

The core idea of our approach is to represent log messages and log templates with vectors and measure the similarity between two log messages as well as between a log message and a log template based on the distance between two vectors, which is thoroughly novel in the log parsing field.

Fig. 2 shows the overall framework of LPV, which consists of an offline component and an online one. The offline component divides the entire log dataset into separate clusters, by first converting log messages into vectors and then applying a clustering algorithm on the resulting vector set. Subsequently, it extracts one or more log templates from each cluster, and further merges similar templates into a single one, in order to get the final fine-tuned templates together with their corresponding vectors (called **template vectors**). The online component fully leverages the outputs of the offline component. For each incoming log message, the online component first converts it into a vector just like the offline component does. Then, it finds the top *n* closest template vectors (whose corresponding templates are chosen as **candidate templates**). Finally, it matches the log message with each candidate template, and selects the completely matched one (if exists) as the log message's template; otherwise, the log message itself is treated as a new log template and its actual template can be extracted in following offline log parsing.

B. Offline Log Parsing of LPV

The input of the offline log parsing is an existing log dataset, and the outputs are (i) the vocabulary together with all tokens' vectors, and (ii) a set of log templates together with their corresponding template vectors. In what follows, we address each phase of the offline log parsing shown in Fig. 2.



index	log message
0	`Connection to 192.168.10.101 is established.`
1	`Connection to 192.168.10.102 is established.`
2	`Connection to 192.168.10.103 is established.`
3	`Connection to 192.168.10.102 is established.`
4	`Connection to 192.168.10.101 is established.`
5	`Connection to 192.168.10.102 is established.`

index	unique log message	indices of log messages
0	`Connection to 192.168.10.101 is established.`	[0, 4]
1	`Connection to 192.168.10.102 is established.`	[1, 3, 5]
2	`Connection to 192.168.10.103 is established.`	[2]

index	unique substituted log message	indices of unique log messages
0	`Connection to $$IPADDR$$ is established.`	[0, 1, 2]

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

1) Preprocessing: In this phase, we aim to reduce the size of dataset and vocabulary by two operations: duplicates removal and common variables substitution (cf., Fig. 3).

► *Duplicates removal.* For a log dataset which spans a long period, it often has a lot of repeated log messages. Note that, here we only consider the *content* field of each log message. For example, the two supercomputer log datasets used by [5], [6], [10], [21] (https://www.usenix.org/cfdr-data), denoted by *BGL* and *HPC* respectively, have over 90% repeated log messages. By removing the duplicates, many repeated manipulations on identical log messages can be avoided.

► *Common variables substitution.* We observe that the huge amount of variables is a main reason that incurs the large vocabulary size of a log dataset. Though we cannot identify all the variables, we do know some most possible formats of variables, such as common variables like IP addresses, numbers, *etc.* Our method utilizes some regular expressions to substitute these common variables with a few special tokens. For example, in our implementation we substitute all IP addresses with a special token "`$$IPADDR$$`". After this kind of substitutions, a lot of log messages could be the same, and duplicates can be removed to get unique substituted log messages since these duplicated log messages definitely share the same template. It is worth noting that, (i) the common variables substitution is not carried out in-place, it makes no impact on the original log messages; and (ii) the final log templates are extracted from original log messages.

2) Vectorization: The goal of this phase is to convert each unique substituted log message into a vector, which is achieved through two steps: *Word2Vector* and *Log2Vector*.

► *Word2Vector.* This step is to map each unique token (or word) to a vector, with the help of *word2vec*. Specifically, we first split each unique substituted log message into tokens by whitespaces and punctuations including "`=`", "`(`", "`)`", "`[`", "`]`", "`:`", "`?`", *etc.* We also treat punctuations as individual tokens, since they always keep constant across all log messages produced by the same "print" statement. For *word2vec*, we adopt the Skip-gram model architecture described in [15], and utilize negative sampling which selects some negative samples for each data sample. But we disable subsampling of high-frequency tokens, since they are more likely to be constant parts. In addition, we constrain the pair of input and label tokens within a unique substituted log message, since there are no strong correlations between the tail tokens of a unique substituted log message and the head tokens of the next one.

► *Log2Vector.* This step is to calculate a vector for each unique substituted log message. Specifically, for a unique substituted log message l consisting of n tokens $t_1, t_2, \ldots, t_n$ (whose corresponding vectors are $\mathbf{V}_1, \mathbf{V}_2, \ldots, \mathbf{V}_n$ respectively), the vector $\mathbf{V}_l$ is computed as: $\mathbf{V}_l = \sum_{i=1}^{n} \mathbf{V}_i$. This step ensures that the vectors of unique substituted log messages having the same template are close in vector space.

3) Clustering: Once all of the unique substituted log messages have been converted into vectors, we can compute the semantic distance between two log messages, based on which we can cluster the log messages. Since the number of log templates in a log dataset is usually unknown in advance, it is more desirable to apply a clustering algorithm that does not need the number of clusters as a user-given parameter. In our implementation, we adopt the Complete-Linkage clustering method and use Euclidean distance. The clustering process will terminate when the minimum distance between two clusters exceeds a threshold τ_d.

4) Log Templates Extraction: This phase consists of three steps: Partition, Intra-cluster Merge, and Inter-cluster Merge.

► *Partition.* For each cluster of unique substituted log messages, this step generates a candidate log template for each member of it. Specifically, for each special token in a unique substituted log message, we extract all common variables previously substituted by it. If all the variables are identical, then we replace this special token with the unique variable value; otherwise, we replace it with the wildcard. Taking Fig. 3 as an example, since the IP addresses substituted by "`$$IPADDR$$`" are not identical, the candidate log template would be "`Connection to * is established.`" if "$\star$" is chosen as the wildcard. We remark that, since there may be multiple candidate log templates for a single cluster after this step, which likes partitioning the cluster into multiple ones, therefore, we call this step "Partition".

The Partition step has two advantages: (i) The possible mistakes made by the previous common variables substitution will be corrected. (ii) If the clustering distance threshold τ_d is too big, then there may be some unique substituted log messages in a cluster which do not have the same log template,

```
(1) a  b  c  d  e
(2) a  f  c  g  e
(3) a  h  c  i  j  e      (1) a  *  c  *  e
(4) b  i  j  h            (2) a  i  c  *  e
         (a)                       (b)
```

Fig. 4. Example of merge. (a) Intra-cluster Merge; (b) Inter-cluster Merge.

this step assigns them with different candidate log templates.

▶ *Intra-cluster Merge.* This step tries to merge similar candidate log templates within a cluster into a single one. Specifically, it can be done by 6 substeps. **First**, each candidate log template is split into columns by whitespaces, equal signs, *etc.* Without loss of generality, we assume that there are at most n columns. **Second**, for each ith column $(1 \leq i \leq n)$, we figure out the most frequent token t_i and its frequency f_i. **Third**, we find the unique values $uf_1, uf_2, \ldots, uf_m$ (suppose that there are m unique values) and their frequencies in the list $[f_1, f_2, \ldots, f_n]$. We then iterate from the most frequent unique value to the least frequent that are bigger than 1. In each iteration (suppose the unique value being iterated is $uf_j > 1$ $(1 \leq j \leq m)$), we count the number of columns which satisfies $f_i \geq uf_j$, and check the ratio of the "count result" to the total number of columns 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. If there is no constant found after the whole iteration, then we terminate the Intra-cluster Merge step. Otherwise, we proceed to the fourth substep. **Fourth**, we pick out all candidate log templates which contain the selected constants in a sequential order. Note that constants are not required to be in fixed columns. **Fifth**, we get the result log template by concatenating all constants and inserting a wildcard for the unsatisfied column(s) of the candidate log templates picked out. **Finally**, for the left candidate log templates within the cluster, we merge them recursively.

Take the four lines in Fig. 4(a) as an example, and let $\tau_r = 0.5$. **First**, each line is split into columns by whitespaces, and there are at most 6 columns. **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 and 4th columns, since the frequencies of all tokens in each column are equal to 1, so any token can be chosen for these two columns. **Third**, since $[f_1, f_2, f_3, f_4, f_5, f_6] = [3, 1, 3, 1, 2, 1] \Rightarrow uf_1 = 1, uf_2 = 3, uf_3 = 2$, it is clear that the most frequent unique value bigger than 1 is 3, and the number of columns satisfying $f_i \geq 3$ is 2 (f_1 and f_3). Since $2/6 < \tau_r$, we iterate to the less frequent unique value 2, and the number of columns satisfying $f_i \geq 2$ 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, since they contain a, c, e sequentially. **Fifth**, now we can get result "a $\star$ c $\star$ e" if "$\star$" is chosen as the wildcard, since there is at least one unsatisfied column between a and c, as well as c and e. **Finally**, since there is only line (4) left, so there is no need to merge it recursively.

▶ *Inter-cluster Merge.* If the clustering distance threshold τ_d is too small, it is possible that some unique substituted log messages having the same template would be assigned to different clusters. To this, we need to merge their candidate log templates into a single one as well. In brief, for each candidate log template in a cluster C_i, we merge it with the ones in cluster C_j satisfying that, the distance between C_i and C_j is less than the distance between C_i and any other cluster $C_k (k \neq j)$. In this step, two candidate log templates are merged only when they have the same length and match in each column. Here, we say two candidate log templates match in one column if and only if their values in this column are equal, or one of them is the wildcard. As for the result template, we set the value of each column as follows: (i) If the values of the two candidate log templates in this column are equal, then the value is set as the result template's value in this column. (ii) Otherwise, one of the two values in this column should be the wildcard. Then, the result template's value in this column is set to the wildcard. For example, the two lines shown in Fig. 4(b) are to be merged into "a $\star$ c $\star$ e", where "$\star$" is the wildcard.

In the end of the Log Templates Extraction phase, we also get the corresponding template vectors. In our implementation, a template vector is defined as the average of the vectors of all unique log messages having the same log template.

C. Online Log Parsing of LPV

Once the offline log parsing is completed, its outputs can be used in the online log parsing. We perform the online log parsing 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 **Preprocessing** and **Vectorization** phases are almost the same as those in the offline log parsing, except that duplicates removal and *Word2Vector* are not needed. Then, in the **Candidate Templates Selection** phase, we calculate the distance between the log message l's vector and each template vector, and figure out the top n closest ones, whose corresponding templates are called the **top n candidate templates**. For the **Template Identification** phase, we attempt to identify the incoming log message l's template from the top n candidate templates. This is achieved by matching l with each candidate template, and the completely matched one is the result. Note that, we say a log message and a log template are completely matched if and only if they can be totally identical after replacing wildcards in the log template with some characters. For example, if line (1) in Fig. 4(a) is an incoming log message, and line (1) in Fig. 4(b) is one of the top n candidate templates, then they are completely matched. On the other hand, if none of the top n candidate templates matches the incoming log message, the log message itself is regarded as its template and the actual template can be generated in following offline log parsing.

Empirically, a small n like 3 or 5 is enough. Thus our method can speed up online log parsing by matching each incoming log message with only a few candidate templates.

TABLE I
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

IV. EXPERIMENTAL EVALUATION

A. Experimental Settings

1) Log datasets: We used the two supercomputer log datasets (*BGL* and *HPC*) mentioned in Section III-B1. In addition, we used another dataset of HDFS logs [6], [8], [21] (denoted by *HDFS*). Table I gives the numbers of log messages and ground truth templates of each dataset. For *BGL* and *HPC*, the ground truth templates are extracted and provided online[1] by the authors of IPLoM. For *HDFS*, the ground truth templates are provided online[2] by the LogPAI team. Furthermore, for each log dataset, the LogPAI team provides 2,000 log messages together with every one's ground truth template, which are scattered in the entire log dataset.

2) Baselines: Drain and Spell are two state-of-the-art log parsing methods [6], [21]. Drain utilizes a fixed depth parse tree to guide log template search. The tree is searched by log message length and preceding tokens to a leaf node, and the final template is selected by token similarity, or a new template is created if no suitable one is found. Spell adopts the following idea: for the log messages produced by the same "print" statement, the constant parts often take a majority, and the longest common subsequence of tokens is 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.

3) Evaluation metrics: For LPV's offline log parsing, we use four metrics ($Accuracy$, $Precision$, $Recall$, and $F\text{-}measure$) to measure the effectiveness. $Accuracy$ is defined as the ratio of correctly parsed log messages in the entire log dataset. $Precision$ means among all log templates generated, how many (the ratio) are the same as the ground truth, and $Recall$ is among all ground truth templates, how many (the ratio) are correctly figured out. $F\text{-}measure$ is defined as: $F\text{-}measure = \frac{2\times Precision \times Recall}{Precision + Recall}$. In addition, we use the time cost of parsing each entire log dataset to measure the efficiency. For LPV's online log parsing, since it is based on the outputs of the offline log parsing, so it's expected that the effectiveness of LPV's online log parsing is the same as that of the offline log parsing. We verify this by testing whether LPV's online log parsing is able to retain the effectiveness of its offline log parsing. And the efficiency is measured by the time cost of online parsing the 2,000 log messages provided by the LogPAI team.

4) Parameter setting: There are two thresholds in LPV's offline log parsing: τ_d and τ_r. In addition, we have the following four key parameters of *word2vec*: the embedding size (ES), window size (WS), number of epochs to train ($epochs$), and number of negative samples per training example (NNS). The default values of the six parameters are given in Table II.

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

TABLE II
PARAMETER SETTING OF LPV.

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

TABLE III
EFFECTIVENESS COMPARISON BETWEEN LPV AND TWO STATE-OF-THE-ART LOG PARSING METHODS ON THREE LOG DATASETS.

Dataset	Method	Accuracy	Precision	Recall	F-measure
BGL	Drain	0.8435	0.1166	0.5227	0.1907
	Spell	0.9002	0.6247	0.6561	0.6400
	LPV	**0.9326**	**0.6388**	**0.6787**	**0.6582**
HPC	Drain	0.8288	0.2511	0.5370	0.3422
	Spell	0.9692	0.6271	0.7048	0.6637
	LPV	**0.9929**	**0.7879**	**0.7429**	**0.7647**
HDFS	Drain	0.7630	0.4222	0.6552	0.5135
	Spell	0.9994	0.8421	**0.9412**	**0.8889**
	LPV	**0.9999**	**0.8492**	0.9310	0.8882

5) Experimental environment: All of our experiments were conducted on a Linux server equipped with a 24-core Intel(R) Xeon(R) CPU E5-2678 v3 @ 2.50GHz, 128GB RAM and NVIDIA GeForce GTX 1080 Ti GPU, running 64-bit Ubuntu 16.04.5 LTS. We implement LPV in Python 3.5.2 and TensorFlow 1.10.1 with GPU support.

B. Effectiveness

1) Offline log parsing: Table III shows the effectiveness comparison results among Drain, Spell, and LPV on the three log datasets. To avoid bias possibly caused by the randomness of the vectors, the metric values of LPV are the average of 5 repeated runs. We can observe that our method LPV outperforms Drain and Spell in all four metrics on *BGL* and *HPC*. And on *HDFS*, LPV outperforms the competitors in 2 out of 4 metrics (*i.e., $Accuracy$* and $Precision$), and performs essentially close to the best in the other 2 metrics. These validate that our attempt to apply such a vector-based clustering method to offline log parsing is feasible and promising.

2) Online log parsing: For each log dataset, we first picked out the 2,000 log messages provided by the LogPAI team and performed offline log parsing with the rest. Then, we performed online log parsing with these 2,000 log messages and checked for each whether its template is among the top 3 candidate templates. The result is given in Table IV, where, $No\ Match$ refers to the number of log messages whose templates are not correctly generated in the offline log parsing, and $Top\ n$ ($n = 1, 3$) is the number of log messages whose templates are among the top *n* candidate templates. One can see that, except for the log messages whose templates are not correctly generated in the offline log parsing, the others' templates are all among the top 3 candidate templates, and more than 99% are the top 1 candidate. Therefore, we can say that, LPV's online log parsing completely retains

TABLE IV
EFFECTIVENESS VERIFICATION OF LPV'S ONLINE LOG PARSING.

	BGL	*HPC*	*HDFS*
No Match	199	84	0
Top 1	1793 (99.56%)	1915 (99.95%)	2000 (100%)
Top 3	1801 (100%)	1916 (100%)	2000 (100%)

TABLE V
TIME COST COMPARISON OF PARSING EACH ENTIRE LOG DATASET (UNIT: SECONDS).

	BGL	*HPC*	*HDFS*
Spell	2352.897	60.436	1835.019
Drain	1202.446	58.754	**1774.236**
LPV	**327.376**	**16.312**	2033.449

the effectiveness of its offline log parsing, and needs only a few string matchings by matching with only top 1 or top 3 candidate template(s), instead of matching with all existing log templates, which is beneficial to improving efficiency. In addition, the percentage of $Top\ 1$ (more than 99%) indicates that it's 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) Offline log parsing: Table V shows the time cost comparison of parsing each entire log dataset among Spell, Drain, and LPV, in which each number is the average time cost of 5 repeated runs. It can be seen that LPV is much faster than the other two methods on *BGL* and *HPC*, and is competitive with them on *HDFS*. These show LPV's competitive efficiency in offline log parsing. Note that, as for LPV, the *word2vec* model training can be done in advance, within several minutes.

2) Online log parsing: For our method LPV, we operate as in Section IV-B2. For Drain and Spell, we run them with the 2,000 log messages provided by the LogPAI team as input. Table VI shows the time cost comparison of parsing these 2,000 log messages online among Spell, Drain, and LPV, in which each number is the average time cost of 10 repeated runs. We can see that LPV is faster than the other two methods on all three log datasets, and the average time cost of parsing a log message is less than 0.15 milliseconds. These confirm the superior efficiency of our method in online log parsing.

V. CONCLUSION

In this paper, we propose LPV, a novel log parsing method based on vectorization. Different from prior log parsing methods, our method converts log messages as well as log templates into vectors, and measures the similarity between two log messages or between a log message and a log template by the distance between two vectors. LPV supports both offline and online log parsing, and the outputs of the offline log parsing are fully leveraged by the online log parsing, so it can make full use of history logs to get good effectiveness, and ensure good efficiency at the same time. Experimental results on public log datasets have validated the feasibility and competitiveness of our method.

TABLE VI
TIME COST COMPARISON OF ONLINE PARSING 2,000 LOG MESSAGES (UNIT: SECONDS).

	BGL	*HPC*	*HDFS*
Spell	0.3200	0.2770	0.3094
Drain	0.2765	0.2544	0.2805
LPV	**0.2365**	**0.1334**	**0.2593**

ACKNOWLEDGMENT

This work is supported in part by the National Key R&D Program of China (No. 2018YFB0204101), the NSFC (No. 61972425, U1811264), and the Key R&D Program of Changsha (No. kq1901023).

REFERENCES

[1] Q. Fu, J. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *ICDM*, 2009, pp. 149–158.
[2] J. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, "Mining invariants from console logs for system problem detection," in *USENIX ATC*, 2010, pp. 1–14.
[3] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *CCS*, 2017, pp. 1285–1298.
[4] A. Das, F. Mueller, P. Hargrove, E. Roman, and S. B. Baden, "Doomsday: predicting which node will fail when on supercomputers," in *SC*, 2018, pp. 9:1–9:14.
[5] M. Du and F. Li, "Spell: Streaming parsing of system event logs," in *ICDM*, 2016, pp. 859–864.
[6] ——, "Spell: Online streaming parsing of large unstructured system logs," *IEEE TKDE*, vol. 31, no. 11, pp. 2213–2227, 2019.
[7] 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 *DSN 2016*, 2016, pp. 654–661.
[8] W. Xu, L. Huang, A. Fox, D. A. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *SOSP*, 2009, pp. 117–132.
[9] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "Clustering event logs using iterative partitioning," in *ACM SIGKDD*, 2009, pp. 1255–1264.
[10] ——, "A lightweight algorithm for message type extraction in system application logs," *IEEE TKDE*, vol. 24, no. 11, pp. 1921–1936, 2012.
[11] L. Tang and T. Li, "Logtree: A framework for generating system events from raw textual logs," in *ICDM*, 2010, pp. 491–500.
[12] L. Tang, T. Li, and C. Perng, "Logsig: generating system events from raw textual logs," in *CIKM*, 2011, pp. 785–794.
[13] R. Vaarandi and M. Pihelgas, "Logcluster - A data clustering and pattern mining algorithm for event logs," in *CNSM*, 2015, pp. 1–7.
[14] J. Stearley, "Towards informatic analysis of syslogs," in *CLUSTER*, 2004, pp. 309–318.
[15] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *NIPS*, 2013, pp. 3111–3119.
[16] Z. Quan, Z. Wang, Y. Le, B. Yao, K. Li, and J. Yin, "An efficient framework for sentence similarity modeling," *IEEE ACM Trans. Audio Speech Lang. Process.*, vol. 27, no. 4, pp. 853–865, 2019.
[17] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and benchmarks for automated log parsing," in *ICSE (SEIP)*, 2019, pp. 121–130.
[18] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *IPOM*, 2003, pp. 119–126.
[19] ——, "A breadth-first algorithm for mining frequent patterns from event logs," in *INTELLCOMM*, 2004, pp. 293–308.
[20] A. Gainaru, F. Cappello, S. Trausan-Matu, and B. Kramer, "Event log mining tool for large scale HPC systems," in *Euro-Par*, 2011, pp. 52–64.
[21] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *ICWS*, 2017, pp. 33–40.