An Algorithm for Message Type Discovery in Unstructured Log Data

Daniel Tovarňák [a]

Institute of Computer Science, Masaryk University, Brno, Czech Republic

Keywords: Log Abstraction, Message Type Discovery, Log Management, Logging, Unstructured Data.

Abstract: Log message abstraction is a common way of dealing with the unstructured nature of log data. It refers to the separation of static and dynamic part of the log message, so that both parts can be accessed independently, allowing the message to be abstracted into a more structured representation. To facilitate this task, so-called message types and the corresponding matching patterns must be first discovered, and only after that can be this pattern-set used to pattern-match individual log messages in order to extract dynamic information and impose some structure on them. Because the manual discovery of message types is a tiresome and error-prone process, we have focused our research on data mining algorithms that are able to discover message types in already generated log data. Since we have identified several deficiencies of the existing algorithms, which are limiting their capabilities, we propose a novel algorithm for message type discovery addressing these deficiencies.

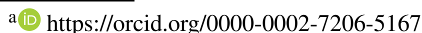
1 INTRODUCTION

Log data analysis is pervasively used for a plethora of mission-critical tasks, e.g. *debugging, security monitoring, forensic analysis, and network management*. Even though this is the case, log data are often considered low-grade due to their predominantly unstructured nature and their somewhat inherent free form. This is what makes log analysis, and log data processing in general, a challenging task, attracting a considerable focus from academia and industry alike.

The matter at hand stems from the way log data are generated. Logging is a *programming practice*. It is used by software developers to communicate information outside the scope of a program in order to trace its execution. It is usually done in an ad-hoc manner and the most important information of log entries – *log messages* – predominantly take the form of relatively short natural-language messages mixed with run-time context variables. Although such log messages are human-readable, the task of automatic log analysis is seriously hardened, since it is challenging to obtain the communicated information in machine-readable forms and data structures.

Unstructured log messages and their transformation into suitable representations is a particular problem that has attracted a considerable amount of research over the years. In literature, such a task is usually referred to as *log abstraction* (Nagappan and Vouk, 2010; Jiang et al., 2008) or *message type transformation* (Makanju et al., 2012). Simply put, log abstraction is the separation of static and dynamic part of the log message so that both parts can be accessed independently. The static part is referred to as *message type*, which essentially corresponds to the parameterized log message in a logging statement, and the dynamic part corresponds to the actual *logging variables* and their values. In order to facilitate the abstraction, regular expression parsing/matching is usually used. In such a case, each message type corresponds to a unique regex pattern.

To better understand the concept, consider a simple example from Table 1 on the next page. From the observed log messages, one can infer two message types described as follows: `[User * logged *]` and `[Service * started]`. In this commonly used notation, the wildcard symbol `[*]` denotes variable parts of the log message. Thus, in our example the log message `[User Jack logged in]` corresponds to the message type `[User * logged *]` with logging variables `[$1=Jack, $2=in]`. Note that since the users are oblivious to the corresponding source code they can easily assume that the log messages were generated via logging statement comparable to `[LOG.info("User {} logged {}", user, action)]` even though this may not be the case. To technically facilitate the actual separation, message types are typically represented by corresponding regular expression patterns, which are then in turn used to parse the log messages.

[a] 

Tovarňák, D.
An Algorithm for Message Type Discovery in Unstructured Log Data.
DOI: 10.5220/0007919806650676
In *Proceedings of the 14th International Conference on Software Technologies (ICSOFT 2019)*, pages 665-676
ISBN: 978-989-758-379-7
Copyright © 2019 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

Table 1: Message type discovery.

Log Messages		Message Types		Regular Expressions
User Jack logged in User John logged out Service sshd started User Bob logged in Service httpd started User Ruth logged out	⇒	User * logged * : [$1, $2] Service * started : [$1]	⇒	User (\w+) logged (\w+) Service (\w+) started

2 RELATED WORK

[This paper is a revised version of a text appearing in the author's dissertation thesis. For a further information on logging, log abstraction, and log data normalization, please see (Tovarnak, 2017).]

Manual discovery of message types and the corresponding matching patterns based on empirical observations can be a challenging task. Although it may be perfectly possible for a small number of message types, as the number grows, automated approaches for message type discovery can rapidly become the only feasible option. Log data generated by a single application or software project can contain hundreds of unique message types since their logging code can contain hundreds of unique logging statements.

When put bluntly, the goal of message type discovery is to find all the *„templates"* that correspond to the parameterized log messages used in logging code. In essence, this goal can be approached in two ways, either via source code analysis or by utilizing data mining techniques on historical log data. At this point, it should be noted again that the focus lies on log messages alone and it is assumed that they are already extracted from log entries, regardless their representation. In literature, several works belonging to these two orthogonal groups of approaches can be found with a general focus leaning strongly towards data mining. In this paper, our focus is also predominantly aimed at the data mining approaches since we are interested in covering the cases in which the source code generating the log data in question is not available. Proprietary hardware appliances and closed-source applications are good examples of such cases.

2.1 Source Code Analysis

As it is apparent from the title, the approaches based on source code analysis assume a certain structure of the logging statements in the source code and the message types are inferred based on the analysis of those statements. Apart from the works discussed below, we have been able to identify a number of works that, although they analyze logging code, e.g. (Yuan et al., 2012), or (Chen and (Jack) Jiang, 2017), they do so for the purpose of characterizing logging practices of the developers, not for the purposes of message type discovery and log abstraction.

(Xu et al., 2009) proposed a general approach to problem detection via the analysis of console logs and discuss their experience with the extraction of message types from source code via static analysis. The workflow of their general approach can be summarized as follows – parse the log entries and structure them via the discovered message types; construct feature vectors from the extracted information; apply anomaly detection methods to mine the feature vectors; visualize results in a decision tree. The static logging code analysis is achieved via abstract syntax tree (AST) traversal. The authors report that by using a source code as a reference to understand the structure of console logs, they were able to parse logs accurately and extract identifiers (message types) and state variables that usually ignored due to difficulties in log parsing. On the other hand, they also observe that it was impossible for them to correctly handle all special cases inherently present in real-world application source code-bases.

Prior to their work characterizing logging practices, (Yuan et al., 2010) proposed SherLog – a tool for diagnosing errors exhibited during production run failures. SherLog can analyze logs from a failed production run and source code to automatically generate useful control-flow and data-flow information to help engineers diagnose the error without reproducing the error or making any assumption on log semantics. The authors implemented LogParser, a general mechanism analyzing source code and its AST to automatically generate regex-based patterns in order to parse log messages and extract variable values. The only required input of LogParser are the names of the used logging functions and a position of the string-formatting argument within them. The source code is then scanned, and AST traversal is used to extract and build corresponding regular expressions together with inverse mapping to source code position. This information is then used to analyze failure logs and infer execu-

tion paths and run-time values that are useful for the developers to understand the root cause of the error.

2.2 Log Data Mining

Source code generating log data is not always available for analysis, e.g. in the case of network devices, closed-source applications, or software appliances. Thus, similarly to many others, we have focused our attention on approaches that discover message types in historical log data via data mining techniques. Generally speaking, the existing works in this area can be roughly divided into two groups. The first group uses specialized algorithms that are exploiting the nature of log data, especially in terms of word frequencies. The most commonly used mining technique is cluster analysis, where the message types are constructed from the discovered log message clusters. A less common group of approaches uses custom heuristics to directly induce the message types, e.g. via natural language processing. We have been also able to find works that combine these approaches. The major focus of this paper is on the former group of approaches.

Many of the log data mining approaches for message type discovery utilize some variant of cluster analysis. Simply put, clustering is a data mining task of dividing a set of objects into groups – clusters – so that the objects from the same cluster are similar to each other more than the objects in other clusters. In many applications, the notion of a cluster is not precisely defined and it is difficult to decide what constitutes a cluster until described via a corresponding algorithm, model, or concept (Tan et al., 2005). In the context of message type discovery, the objects that are subjected to the cluster analysis are log messages. The found clusters of log message are then considered to constitute the message types. For example, given the log messages listed in Table 1, the clustering techniques are able to automatically discover two clusters that represent the corresponding message types, which would have to be otherwise inferred empirically/manually.

The approaches used in this area take advantage of the observation that although the log messages are free-form, they are generated by a limited set of fixed logging statements and thus the generated log messages are likely to form clusters with respect to variable positions. However, as originally discussed in (Vaarandi, 2003) and further revisited in (Makanju et al., 2012), traditional clustering algorithms are not suitable for log message clustering. Provided each log message is considered to be a data point, and its individual words, or tokens, are referred to as attributes, the reasons can be paraphrased as follows.

- The log messages (message types) do not have a fixed number of attributes.
- The data point attributes, i.e. the individual words or tokens of each log message, are categorical. Most conventional clustering algorithms are designed for numerical attributes.
- Traditional clustering algorithms also tend to ignore the order of attributes. In the context of log messages (message types), the attribute order is important.
- Log data are high-dimensional, and although clustering algorithms for high-dimensional data have been designed, they do not cope well with data with different attribute types.

For these reasons, several algorithms and techniques for automatic clustering and/or categorization of log message with the goal of message type discovery have been developed, e.g. (Vaarandi, 2003; Nagappan and Vouk, 2010; Makanju et al., 2012; Fahad et al., 2014). Moreover, several works have emerged that do not use cluster analysis per se, yet, still take into account both the facts stated above and also exploit the same characteristics inherent to log data. The most prominent characteristic, first observed by Vaarandi in (Vaarandi, 2003), is the different frequency of words/tokens representing static part of the message, i.e. message type, and the tokens representing variable part of the log message. (Vaarandi, 2004) is an example of an approach that is not clustering per se – it uses an Apriori-based algorithm that models message type discovery as a task of frequent item-set mining.

We have been also able to identify several works that rely largely on domain knowledge and the nature of the targeted data set. For example, (Taerat et al., 2011) classifies individual log message tokens based on their type before clustering. In particular, it leverages a dictionary of English words for classification. After this classification, a meta-clustering step is performed. The work presented in (Jiang et al., 2008) represents a similar case, which is based on explicit language and log message structure heuristics. Although such approaches are certainly usable for some cases, they cannot be considered general-purpose. On the other hand, *optional* domain knowledge provided by the log analyst generally results in an increased accuracy of the message discovery approaches, e.g. as shown in (Tang et al., 2011), or (Vaarandi and Pihelgas, 2015). For example, via corresponding regular expressions, it is possible to indicate that an IP address, e-mail, and MAC address should be always considered to be variables. Moreover, it is quite common for the discovery algorithm to be executed multiple times by the log analysts with different parameters to yield the best results for the targeted data set. In fact, user

review is a fairly common (and important) part of message type discovery. Additionally, (He et al., 2016) reports that, in some cases, the removal of evident variables, e.g. IP addresses, can further improve the discovery accuracy. However, such preprocessing is not considered in this paper.

In the following paragraphs, we will present the core principles of the most commonly cited techniques and algorithms for message type discovery and then further discuss their characteristics and address their potential limitations in the context of our goals. Note that a first common step of every algorithm is the *tokenization* of log messages since usually, the input is in the form of plain-text lines, yet, the clustering algorithms work with individual words, also referred to as tokens. Most commonly, the messages are tokenized using inter-word spaces, unless we indicate otherwise. Also, note that we will use a basic wildcard notation to denote message types, i.e. `[Service * started]` will denote a message type with one variable position in the middle.

3 STUDIED APPROACHES

Simple Logfile Clustering Tool

In his seminal work (Vaarandi, 2003) Risto Vaarandi presented Simple Logfile Clustering Tool (SLCT) that uses a density-based method for clustering. The first step of the algorithm is similar to popular Apriori algorithm for mining frequent itemsets over transactional databases, then, however, takes a different approach and exploits two important observations about data sets consisting of log messages: *majority of the words occur only a few times in the data set, and there are many strong correlations between words that occur frequently*. Informally, the algorithm employed by SLCT can be described as follows.

1. Discover frequent $(position, word)$ pairs in the data set, e.g. $(1, Service)$. A pair is considered frequent if it occurs at least N times in the data set, where N is the user-specified support threshold.
2. For each log message, extract all frequent $(position, word)$ pairs it contains, for example $\{(1, Service), (3, started)\}$, and calculate the number of occurrences of such extracted combinations.
3. Combinations that occur at least N (support threshold) times in the data set are selected as clusters and then converted to an internal representation of message types – the missing word positions are replaced by wildcards, e.g. $\{Service, *, started\}$. Since by default, a space is used for message tokenization, the final message type can be translated to `[Service * started]`.

Observe, that due to the way SLCT works, it is unable to create wildcards after the last word in the combination. This means that if we consider combination $\{(1, User), (3, logged)\}$ to be selected as a cluster, SLCT is not able to convert it to a proper message type of `[User * logged *]`.

LogHound

LogHound (Vaarandi, 2004) is another clustering tool created by the same author as SLCT. It exploits the same observations about log message data sets, yet it transposes the clustering task to the problem of frequent item-set mining. The LogHound algorithm can be considered to be a generalization of Apriori and it mirrors it closely. In order to mine message types, LogHound views m-th log message in data set as a transaction (m, X), where $X = p_1, ..., p_k$ and $p_1, ..., p_k$ are $(word, position)$ pairs. Frequent itemsets are then converted to message types (Vaarandi, 2008). Similarly to SLCT, LogHound is unable to detect wildcards in the place of the last word of the message type and does not consider multiple delimiters.

Jiang, et al.

The authors of (Jiang et al., 2008) propose an automated approach for abstracting execution logs to execution events. The proposed approach is not used for message type discovery per se, i.e. with the goal of using the message types for pattern matching later, but rather for summarization and categorization of log messages within a data-set. Nevertheless, a basic heuristic-based algorithm with several clustering steps is proposed that in fact results in the discovery of message types.

1. The *anonymize* step uses heuristics to recognize variable positions in log messages and replaces them with generic tokens. Heuristics can be added or removed from this step based on domain knowledge. The heuristics used in the paper consist of assignment pairs like `[word=value]` and phrases like `[is[are|was|were] value]`.
2. The *tokenize* step separates the anonymized log messages into different groups (bins) according to the number of words and number of generic tokens in each log line. This means that log line with two words and one generic token would be put into a bin with the name $(2, 1)$.
3. The *categorize* step compares anonymized log messages in each bin, and each anonymized log message is assigned its unique id. Identical

anonymized messages are assigned identical ids and represent the same execution event.

4. The *reconcile* step re-examines all the execution events (message types) and merges such execution events that meet the following requirements: they belong to the same bin, differ from each other by one word at the same position, and there exists at least a predefined number of such execution events.

It can be seen that the proposed approach strongly depends on the used heuristics, and the fine-tuning possibilities are limited to the *reconcile* step. We have also observed that in some cases, the algorithm can produce overlapping message types.

Log Key Extraction (LKE)

The authors of (Fu et al., 2009) focus on log analysis for automated problem diagnosis. They propose a technique to detect anomalies based on finite-state automata, including work-flow errors and low-performance anomalies from system logs. Similarly to others, a need for log message classification (abstraction) is recognized as a necessary step. Thus, a technique to extract *log keys* from log messages is proposed. Note that this is not message type discovery per se since the authors are not interested in the dynamic content, but rather only in the static portion of the log message. Therefore, instead of message types, the technique discovers log keys, i.e. log messages without parameters, which are then used to classify log messages, without extracting the variable content. In a nutshell, the technique works as follows.

1. Parameters are erased based on empirical rules, e.g. numbers, URIs, and IP addresses. Such raw log keys are then tokenized into words using a space as a separator.
2. An initial connectivity-based clustering is performed using weighted word-edit distance to measure the similarity of the tokenized raw log keys.
3. The initial clusters are repeatedly partitioned using a heuristic-based splitting procedure based on dynamic positions' cardinalities.
4. Log keys are extracted from the final clusters by considering only common positions of the raw log keys.

The discovered log keys are then used for classification. The first step described above is repeated for each log message and the corresponding log key is determined by finding a minimal weighted edit distance. Note that although this technique cannot be directly used to discover message types later usable for pattern matching, it is able to induce a clustering on the targeted data set.

Nagappan-Vouk Algorithm

The Nagappan-Vouk algorithm presented in (Nagappan and Vouk, 2010) is inspired by SLCT, yet it does not form clusters of individual words (tokens), but of their respective frequencies. As the authors point out, that way, the clusters are not being discovered across log messages, but rather within them. This rather straightforward algorithm can be described as follows.

1. Build a frequency table that contains the number of word occurrences in a particular position in the log message. Hence, the rows in the table are the words, and the columns are the positions in each log message (see Table 2).
2. For each log message, retrieve the corresponding frequencies of the words the message consist of. Then, find frequency Z that occurs the most times within the log message – if there is a tie, pick the lowest frequency.
3. All the words of the log message that have frequencies greater or equal to Z are then considered to be the static parts of the message. The rest are variable parameters. Thus, a message type is created.
4. The discovered message types are saved into an appropriate structure (e.g. hash-table) in order to avoid duplicates. After the whole data set is iterated, the found message types are returned.

Considering frequency table shown in Table 2 below, the log message `[Service httpd started]` corresponds to word frequencies of $\{5,2,5\}$, respectively. The most common frequency Z is equal to 5, therefore, message type internally represented as $\{Service, *, started\}$ can be inferred. After the execution of the algorithm, a single message type `[Service * started]` will be constructed from the internal representation since the tokenization is based on a space delimiter. Although quite elegant, the algorithm does not provide any means for fine-tuning and produces overlapping message types.

Table 2: Example of a frequency table for Nagappan-Vouk Algorithm.

	1	**2**	**3**
Service	5	0	0
httpd	0	2	0
sshd	0	2	0
started	0	0	5
xinetd	0	1	0

Iterative Partitioning Log Mining

Iterative Partitioning Log Mining (IPLoM) (Makanju et al., 2012) is a technique based on hierarchical clus-

tering. In the first three steps the algorithm iteratively (recursively) partitions the data set and in the final step it attempts to discover the corresponding message types. The four steps can be summarized as follows.

1. *Partition the data set by a token count.* After this step the log messages in the resulting partitions will have the same length, token-wise, forming *n*-tuples within the partitions.
2. *Partition by token position.* For each partition calculate the cardinality of the tokens present at each position (the position can be viewed as a column in the *n*-tuples). Pick the position with the smallest cardinality and split the partition by unique values on that position.
3. *Partition by search for bijection.* First, rule out partitions that already form good clusters (using *cluster goodness threshold*). For each remaining partition, find the most occurring cardinality among the positions and pick first two positions that have this cardinality. Search for bijections (*1-1* relation) among unique tokens on these two positions and split the partition by the found bijections. If there are *1-M* and *M-1* relations, determine the split based on upper and lower bound thresholds. Ignore, or attempt to further split *M-M* relations.
4. The resulting partitions are considered to be the discovered message types. Positions with cardinality equal to 1 are considered to be constant, the remaining positions are variable parameters.

IPLoM demonstrated statistically significantly better performance than either SLCT or Loghound on six of the seven different data sets tested in (Makanju et al., 2012), and it achieved outstanding accuracy. Whilst having a good performance, IPLoM tokenizes log messages using space as a fixed delimiter and does not support multi-word variables, which limits its capabilities, as also confirmed by the authors. Overall, the algorithm can be parameterized using up to 4 bounded parameters.

logSig

In (Tang et al., 2011) the authors propose a message signature-based algorithm *logSig* to generate system events from textual log messages. By searching the most representative message signatures (message types), logSig categorizes log messages into a set of event types. The algorithm is not typical cluster analysis per se, but rather classification since it tries to find *k* message signatures (message types) to match all given messages as much as possible, where *k* is specified by the user, which is not very suitable for our purposes. The algorithm itself consists of three steps.

The first step is to separate every log message into all the possible pairs of words preserving order, e.g. $(Service, httpd)$, $(Service, started)$, $(httpd, started)$. The second step is to find *k* groups of log messages using *local search* optimization strategy such that each group share as many common pairs as possible. The last step is to construct message signatures (message types) based on identified common pairs for each message group. The authors evaluate the accuracy of the algorithm and show that the incorporation of domain knowledge led to increased accuracy for all the tested data sets, except one.

Baler Tool

Baler tool (Taerat et al., 2011) takes a slightly different approach than others since it classifies log message tokens based on their semantics, whilst leveraging English dictionary. The classified messages are then (meta)-clustered in steps, using two defined similarity functions based on Levenshtein edit distance. In the final step, message types are extracted from the discovered clusters by finding a *"longest common sub-sequence"* of the tokens and wildcards. Note that similar approaches rely on the existence of dictionary for the language the log messages are written in, provided natural language is used at all. On the other hand, Baler merges overlapping clusters and also supports multi-word variable positions.

LogCluster

LogCluster (Vaarandi and Pihelgas, 2015) by Vaarandi and Pihelgas addresses several shortcomings identified in their previous work and uses an algorithm that is an evolution of SLCT and LogHound. The algorithm does not consider token positions and after the candidate clusters are generated, it utilizes two additional heuristics to join and aggregate the candidates in order to deal with overlapping clusters and multi-word variables. LogCluster generates special aggregated message types, e.g. `[User with name *{1,3} (created|deleted)]`, where the $*\{m,n\}$ wildcard matches *m* to *n* words and the expression in parentheses can match only the listed tokens. To control the heuristics, the algorithm takes up to 5 parameters excluding regular expression used for tokenization.

Whilst we consider LogCluster to be currently the most advanced algorithm for log message clustering, it is our experience that it requires additional fine-tuning to provide the expected results, and the optimal values are not easy to find. Additionally, the aggregated message types are sometimes not suitable for direct conversion to corresponding matching patterns. Finally, although internally, the algorithm uses regular

expressions for tokenization, i.e. it supports multiple delimiters, these delimiters are then stripped from the resulting message types, rendering this functionality unusable.

4 IDENTIFIED DEFICIENCIES

Apart from the general comments provided above, we have identified several deficiencies of the existing algorithms that either limit their accuracy or usability for log abstraction. In some cases, only some of the approaches are affected, yet, there is no algorithm avoiding all of them, thus, providing a room for improvement.

Generation of Overlapping Message Types

Cluster/message type overlapping is a characteristic typical for frequency-based algorithms (*SLCT*, *LogHound*, *Nagappan-Vouk*, *Jiang, et al.*). This rather common situation occurs when the algorithm reports message type that fully or partially matches some other reported message type, e.g. `[Service httpd *]` and `[Service * started]`.

This is unsuitable for the purposes of log abstraction since one log message can adhere to two different message types, introducing unwanted unambiguity. The desirable outcome would be a single reported message type, i.e. `[Service * *]` in this case, created by merging of the overlapping clusters. This can be remedied via an advanced post-processing of the detected message types.

No Support for Multiple Token Delimiters

All of the studied approaches rely on delimiter-based tokenization for separating individual words (tokens) of the log messages, yet, only a single character can be used in most cases, which can be a limiting factor. The commonly used delimiter is space for obvious reasons. For example, the algorithms are unable to discover message type `[Login - user=* succ=*]` since they cannot use both the space `[␣]`, and equals sign `[=]` to tokenize the message. Instead, they are only able to report message type `[Login - * *]`, which is not accurate.

Note that this is not as straightforward to fix in the existing algorithms as it may appear and it would likely result in non-trivial alterations – the challenge of supporting multiple delimiters is not in the tokenization phase per se, but rather in the phase of message type construction/generation. This is also the case of *LogCluster*, which, even though using regular expressions for tokenization, omits the delimiters from the generated output, rendering it useless. Note that in some cases, it can be impossible to discover proper message types in a single execution even when the algorithm supports multiple delimiters for tokenization. In such situations, a manual intervention is needed.

Complicated Parameterization

In one way or the other, all of the presented approaches rely on the variability present in the targeted log message data sets. Moreover, it is our experience that the domain knowledge of the user is usually the main deciding factor of what constitutes an acceptable message type or their group. This means that the log analysts must be able to somehow control the sensitivity of the message type discovery algorithm depending on the targeted data set. This can be also traced in the works discussed above – in the evaluation phase, the respective algorithms were often fine-tuned to provide the best results for the data sets of interest.

While some of the proposed approaches do not support parameterization at all, for the others, except *logSig*, we deem the fine-tuning to be rather complicated and unintuitive. From our point of view, a single bounded parameter controlling sensitivity would be the most intuitive. Since the users usually react either to overly general or to overly specific message types, the parameter should ideally control granularity of the output, i.e. at one end of the parameter's interval, many individual message types would be discovered and at the opposite end, fewer, more general message types, would be returned.

No Support for Multi-word Variables

Majority of the above-mentioned approaches rely on the position of tokens within the log message. This results in the inability to properly detect variable parameters consisting of multiple words. Also, for *SLCT* and *LogHound* this results in the inability of detecting message types with wildcards at the last position. Consider data set that consists of variations of the following two messages: `[Service httpd started]` and `[Service foo bar started]`. Since they are of different length with variables at different positions, they would be eventually recognized as two distinct types, i.e. `[Service * started]` and `[Service * * started]`.

In some cases, this can be considered to be a correct behavior, yet, usually, this is not optimal. The approach to cope with this can be relatively straightforward, yet it requires additional iteration over the found message types. For example, the detected message types can be grouped using some kind of heuristic, e.g. *Levenshtein edit distance* used on whole words as

proposed in (Makanju et al., 2012). *Baler*, and *LogCluster* are the only algorithms to support multi-word variables.

5 PROPOSED ALGORITHM

In order to address the above-mentioned deficiencies, we have decided to design a new message type discovery algorithm combining different techniques used in the studied approaches. We refer to the algorithm as to the *Extended Nagappan-Vouk* (*ENG*) since it is based on the original idea of frequency table and intra-message word frequency. Other than that, the algorithm is significantly improved in order to support multiple delimiters for tokenization, support multi-word variables, report distinct message types with no overlapping, and finally, the algorithm can be parameterized via a single parameter controlling the sensitivity of the algorithm and the granularity of the reported message types.

Algorithm 1 (on the next page) takes three parameters – log message lines, i.e. the targeted data set; a list of delimiters used for tokenization; and a word frequency threshold represented by a percentile value between 0 and 100, inclusive. The algorithm consists of 5 steps – tokenization; frequency table generation; generation of base message types; creation of merged message types, i.e. elimination of overlapping clusters; and detection of multi-word variables.

First, the log messages are tokenized using the provided delimiters into a list of $(position, word)$ pairs and delimiters. It is important for the delimiters to be a part of the tokenized line, otherwise the algorithm would not be able to reconstruct proper message types back. Note that it is sufficient for the algorithm to work only with a set of *unique* log message lines, which is enforced in the beginning of the processing in order to improve an overall practical performance. After the tokenization step, a frequency table is built, taking only the tokenized words into account.

In the third step, a base message type is built for each tokenized log message, which is then inserted into a set of message types in order to eliminate duplicates. This step extends the core idea of the original algorithm. However, instead of most occurring frequency within the log message, the method of q-th percentile is used to discover variable parts of the message.

A word at a given position is a variable and it is replaced by a wildcard, if its frequency is lower than the q-th percentile of the frequencies present in the current tokenized line. This method allows us to define what a variable position means and parameterize the sensitivity of the variable discovery. For $q = 0$, i.e. the minimum frequency, the algorithm reports more words to be static, while for $q = 100$, i.e. the maximum frequency, it reports more words to be variable positions. After the base message types are generated, two post-processing steps are executed.

First, overlapping message types are eliminated in favor of generating more general message types we refer to as merge-types, which are strictly distinct. The base message types are partitioned by their length, including delimiters and wildcards, and reverse-sorted so that the most general message types are first. Each partition of the same length is then recursively iterated, trying to merge the base message types. Two message types can be merged if they exactly match each other. Variable positions always compare to equal to any word or other variable position. For example, message types $[foo, \sqcup, \star]$ and $[\star, \sqcup, bar]$ will result in merge-type $[\star, \sqcup, \star]$, corresponding to the internal representations of `[foo *]`, `[* bar]`, and `[* *]` respectively. The merging process continues until the partition becomes stable, i.e. there are no other base message types that can be merged. In the second post-processing step, we aggregate wildcard sequences with the aim of supporting multi-word variable positions. For each (merged) message type, wildcard chains are detected, i.e. sequences of wildcards that are delimited via a single space.

For example, message type $[foo, \star, \sqcup, \star, \sqcup, \star]$ will become $[foo, \sqcup, \star], [3]$ indicating that there are 3 space-separated words on the first variable position. By iterating all the message types, a temporary table is created, containing all the aggregate message types that differ only by the number of space-separated words on the respective variable positions, i.e. so-called wildcard lengths are calculated. Then, the temporary table is iterated and for each aggregate message type, all the monotonic sequences of lengths are detected. This means that, for example, for message type $[foo, \sqcup, \star], [[1, 3, 4]]$, the first variable position is detected to be consisting of either 1 word, or 3-4 words. Based on such monotonic ranges, i.e. wildcard cardinalities, all the possible aggregate message types are generated. Since in our example the message type has only one variable position, the cardinalities $[[(1, 1), (3, 4)]]$ will result in the algorithm reporting two possible message types `[foo %{1,1}]` and `[foo %{3,4}]` – the result of a Cartesian product of wildcard cardinalities for each respective variable position.

5.1 Output Example

Listing 1 (page after next) shows an example of message type discovery results of the described algorithm – its Python implementation, to be precise. Notice that

Algorithm 1: Message type discovery (ENG).

Input: log message *lines* to be clustered, *delimiters* used for tokenization, *percentile* threshold
Output: message types set *mt_set*

```
1:  function DISCOVERMESSAGETYPES(lines, delimiters, percentile)
2:      tokenized_lines ← [do tokenize(line, delimiters) for line in unique(lines)]
3:      freq_tab ← BUILDFREQUENCYTAB(tokenized_lines)
4:      mt_set1 ← BUILDMESSAGETYPES(tokenized_lines, freq_tab, percentile)
5:      mt_set2 ← BUILDSUPERTYPES(mt_set1)
6:      mt_set ← BUILDAGGTYPES(mt_set2)
7:      return mt_set
8:  function BUILDFREQUENCYTAB(tokenized_lines)
9:      freq_tab ← new HASHMAP(Pair → Integer)
10:     for tokenized in tokenized_lines do
11:         for x in tokenized do          ▷ tokenized line is a list of (position, word) pairs and delimiters
12:             if isWord(x) then          ▷ consider only words, not delimiters
13:                 freq_tab[x]++          ▷ increment frequency of this (position, word) pair
14:     return freq_tab
15: function BUILDMESSAGETYPES(tokenized_lines, freq_tab, percentile)
16:     mt_set ← {}                        ▷ mt_set will contain only unique message types
17:     for tokenized in tokenized_lines do
18:         frequencies ← [do freq_tab[x] for x in tokenized if isWord(x)]
19:         threshold ← qthPercentile(frequencies, percentile)
20:         message_type ← []
21:         for x in tokenized do
22:             if isWord(x) then
23:                 if freq_tab[x] < threshold then
24:                     message_type ≪ ⋆    ▷ substitute word with a wildcard
25:                 else
26:                     message_type ≪ x.word
27:             else
28:                 message_type ≪ x
29:         mt_set ≪ message_type          ▷ message_type is a list of words, wildcards, and delimiters
30:     return mt_set
31: function BUILDSUPERTYPES(mt_set)
32:     mrgt_set ← {}
33:     for partition in partitionByMsgTypeLength(mt_set) do
34:         mtypes ← reverse(sort(partition))   ▷ wildcard > word > delimiter
35:         merged_partition ← {}
36:         while mtypes ≠ ∅ do
37:             merge_found ← FALSE
38:             mt1 ← pop(mtypes)
39:             for mt2 in mtypes do
40:                 success, mtx ← tryToMerge(mt1, mt2)
41:                 if success then
42:                     remove(mt2, mtypes)
43:                     push(mtx, mtypes)
44:                     merge_found ← TRUE
45:                     break
46:             if not merge_found then
47:                 merged_partition ≪ mt1
48:         mrgt_set ≪ merged_partition
49:     return flatten(mrgt_set)           ▷ flatten the set of sets into a single set
50: function BUILDAGGTYPES(mt_set)
51:     temp_tab ← new HASHMAP(List → List)
52:     for message_type in mt_set do
53:         mt, len_list ← mergeWildcardChains(message_type)   ▷ mt is shortened message type
54:         temp_tab[mt] ≪ len_list        ▷ len_list contains lengths of merged wildcard chains in mt
55:     agg_set ← {}
56:     for mt, len_lists in temp_tab do   ▷ len_lists is a list of lists
57:         crd ← []
58:         for e in zip(len_lists) do     ▷ each element e is a list of possible lengths of each wildcard in mt
59:             crd ← [do (min(q), max(q)) for q in mnSeq(e)]   ▷ detect monotonic sequences of lengths
60:         for f in cartesian(crd) do     ▷ generate all possible combinations of wildcard cardinalities for mt
61:             agmt ← (mt, f)             ▷ agmt is a message type paired with a list of its wildcard cardinalities
62:             agg_set ≪ agmt
63:     return agg_set
```

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

```
Start processing (xor) Jen=user           Service sshd:22 started
User John logged out                      Start processing (xor) Daniel=user
User Bob logged in                        User Ruth logged out
Start processing (xor) Thomas=user        Start processing (xor) Tom Sawyer=user
Service httpd:8080 started                Start processing (nor) Root=user
```

```
# delimiters='\s', percentile=30:
User %{1:1} logged out
User Bob logged in
Start processing (xor) %{1:2}
Start processing (nor) Root=user
Service %{1:1} started
```

```
# delimiters='\s:=\(\)', percentile=65:


User %{1:1} logged %{1:1}
Start processing (%{1:1}) %{1:2}=%{1:1}
Service %{1:1}:%{1:1} started
```

Listing 1: Message type discovery results for a simple data set, using different parameters.

different parameters result in message types of different granularity. Also, note that the shown data set is not an excerpt, but an actual input for the algorithm. This illustrates that the discovery can be performed even on a relatively small historical sample.

6 ACCURACY EVALUATION

The accuracy evaluation of message type discovery is a typical task that can be achieved via classic information retrieval techniques for clustering evaluation (Manning et al., 2008). Assuming the discovery does not produce overlapping message types, each discovered message type induces a *strict cluster* of log messages in the original data set, i.e. each log message corresponds to exactly one message type. Note that this is naturally true regardless if clustering-based methods were used for the discovery or not. Provided there is a second clustering available, which is representing the *ground truth*, or *gold standard*, it is possible to calculate a number of external criteria that evaluate how well the discovered clustering matches the gold standard classes (message types).

Similarly to many others, we use *F-measure (F_1 score)* as the external criterion to be used for message type discovery accuracy evaluation. F-measure is a harmonic mean of *precision* and *recall*, other common external criteria, which can be calculated as follows (Manning et al., 2008). Let us view the task of clustering as a series of decisions, one for each of the $\frac{N\cdot(N-1)}{2}$ pairs of items (documents, log messages) in the data set. We want to assign two items to the same cluster if and only if they are similar, e.g. when two log messages belong to the same message type. The ground truth data set is used to determine if the decision is correct or not. A true-positive (*TP*) decision assigns two similar items to the same cluster, a true-negative (*TN*) decision assigns two dissimilar items to different clusters. There are two types of errors we can commit. A false-positive (*FP*) decision assigns two dissimilar documents to the same cluster. A false-negative (*FN*) decision assigns two similar documents to different clusters. *Precision* is the fraction $\frac{TP}{TP+FP}$ of all the decisions to cluster two items together and the decisions that do so correctly. *Recall* is the fraction $\frac{TP}{TP+FN}$ of all the decisions to cluster two items that should have been made and the decisions that were actually made. F-measure can be calculated from the precision and recall as follows.

$$F_1 = 2\cdot\frac{Precision\cdot Recall}{Precision + Recall}$$

Thanks to the work of He et al. (He et al., 2016) we have been able to evaluate the accuracy of our algorithm on externally provided heterogeneous log message data sets and accompanying ground truths, which should add to the evaluation validity. Moreover, we were able to compare the algorithm's accuracy with accuracies reported for some other algorithms for message type discovery. In their evaluation study, the authors used five real-life log message data sets ranging from supercomputers (BGL and HPC), through distributed systems (HDFS and Zookeeper), to standalone desktop software (Proxifier), in order to evaluate accuracy of four different message type discovery algorithms (*SLCT*, *IPLoM*, *LKE*, and *logSig*).

The data sets were randomly sampled for 2000 log messages from each dataset in order to shorten the running times of some of the more computationally intensive algorithms. The ground truth was created manually by the authors. The reported results (F-measures) are summarized in Table 3. Note that in the evaluation study, the authors also report results for preprocessed log messages, e.g. with removed IP addresses, yet we do not consider such preprocessing in our evaluation.

We were able to re-run the experiments on the sampled data sets, which are available online[1]. We

[1] https://github.com/logpai/logparser

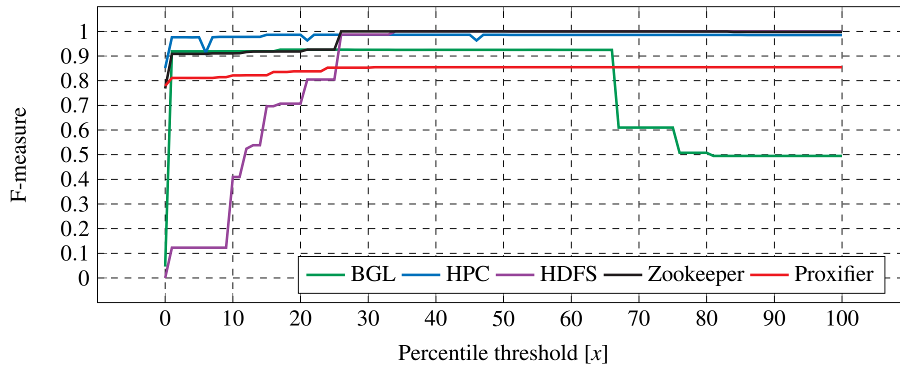


Figure 1: F-measure of *ENG* with respect to percentile threshold $q = x$ and *default delimiters*.

Table 3: F-measures for algortihms evaluated in (He et al., 2016).

	BGL	HPC	HDFS	Zookeeer	Proxifier	*AVG*
SLCT	0.61	0.81	0.86	0.92	0.89	*0.818*
IPLoM	0.99	0.64	0.99	0.94	0.90	*0.892*
LKE	0.67	0.17	0.57	0.78	0.81	*0.600*
LogSig	0.26	0.77	0.91	0.96	0.84	*0.748*

have implemented the presented Extended Nagappan-Vouk (*ENG*) algorithm in Python, and the prototype implementation was executed with several different settings in order to demonstrate the effects of different parameterizations and also to demonstrate the benefits of the algorithm's support for multiple delimiters. The accuracy results for five different parameterizations are shown in Table 4.

Table 4: F-measures for Extended Nagappan-Vouk algorithm and its different parameterizations.

	BGL	HPC	HDFS	Zookeeer	Proxifier	*AVG*
(1)	0.8556	0.8778	1.0000	0.7882	0.8162	*0.86756*
(2)	0.9251	0.9861	1.0000	0.9999	0.8547	*0.95316*
(3)	0.9191	0.9861	0.6965	0.9182	0.8220	*0.86838*
(4)	0.4949	0.9856	1.0000	0.9979	0.8547	*0.86662*
(*)	0.9985	0.9861	1.0000	0.9999	1.0000	*0.99690*

The first parameterization *(1)* considers the default percentile threshold ($q = 50$) with space serving as a delimiter. The second parameterization *(2)* also uses the default percentile, yet, now, multiple delimiters are used for tokenization in their default form (`[,;:=[]()␣]`). The third and fourth parameterization *(3)*, *(4)* shows the sensitivity of different data sets to different percentile thresholds ($q = 15$, $q = 85$), again with default delimiters. Finally, the fifth parameterization *(*)* represents the best possible clustering results that can be achieved for the algorithm. The percentile $q = 50$ exhibits the best overall accuracy when each respective data set is tokenized by an individual and fine-tuned set of delimiters. Although from a subjective perspective these tokenizations do not produce the best possible message types, they produce the best clustering results. This is related to the fact that the ground truth message types are also not at their best, but again, this is subjective.

It can be seen that even for the default algorithm settings, i.e. $q = 50$ with default delimiters, the algorithm exhibits very high accuracy represented by F-measure. Moreover, when considering different sets of delimiters for each evaluated data set, the exhibited accuracy is truly superior.

Figure 1 (above) illustrates that each of the tested data sets exhibits different sensitivity to the percentile threshold parameter q. It should also demonstrate, why we have chosen $q = 50$ as a default threshold parameter – during our use of the *ENG* algorithm in practice, the range between $q = 40$ and $q = 60$ exhibited the best results for many other real-life data sets.

SCIENCE AND TECHNOLOGY PUBLICATIONS

7 CONCLUSIONS

In this paper, we have focused on the first tier of the log abstraction task, i.e. message type discovery. The research in this area is focused on automated approaches for message type discovery and two orthogonal groups of approaches can be identified. The message type discovery can be based either on source code analysis or on data mining techniques applied to already generated log data. Since the source code of the targeted application producing log data may not be always available for analysis, we have turned our attention towards approaches that discover message types from historical log data via data mining techniques. We have studied the existing algorithms and identified several deficiencies, mainly in terms of their limited capabilities and practical usability for our goals. By combining the principles of several of these algorithms, we have been able to design a message type discovery algorithm with the ability to generate non-overlapping message types, use multiple token delimiters, use a single bounded parameter for fine-tuning, and support multi-word variables.

An accuracy evaluation based on five real-world data sets with externally provided ground truth shown that the proposed algorithm exhibits a superior accuracy. In the best case, the algorithm exhibited an average *F-measure* of 0.9969. In the future, we plan to deal with the problem of incremental discovery and message type evolution, which is somewhat inherent to the logging domain. We plan to address this by altering the algorithm so that it can work in an online manner, i.e. the message types would be discovered on-the-fly and the pattern-set for regex matching would be also updated dynamically.

ACKNOWLEDGEMENTS

The publication of this paper and the follow-up research was supported by the ERDF „CyberSecurity, CyberCrime and Critical Information Infrastructures Center of Excellence“ (No. CZ.02.1.01/0.0/0.0/16_019/0000822).

REFERENCES

Chen, B. and (Jack) Jiang, Z. M. (2017). Characterizing logging practices in java-based open source software projects – a replication study in apache software foundation. *Empirical Software Engineering*, 22(1):330–374.

Fahad, A., Alshatri, N., Tari, Z., Alamri, A., Khalil, I., Zomaya, A. Y., Foufou, S., and Bouras, A. (2014). A survey of clustering algorithms for big data: Taxonomy and empirical analysis. *IEEE Transactions on Emerging Topics in Computing*, 2(3):267–279.

Fu, Q., Lou, J.-G., Wang, Y., and Li, J. (2009). Execution anomaly detection in distributed systems through unstructured log analysis. In *International conference on Data Mining (full paper)*. IEEE.

He, P., Zhu, J., He, S., Li, J., and Lyu, M. R. (2016). An evaluation study on log parsing and its use in log mining. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 654–661.

Jiang, Z. M., Hassan, A. E., Flora, P., and Hamann, G. (2008). Abstracting execution logs to execution events for enterprise applications (short paper). In *2008 The Eighth International Conference on Quality Software*, pages 181–186.

Makanju, A., Zincir-Heywood, A. N., and Milios, E. E. (2012). A lightweight algorithm for message type extraction in system application logs. *IEEE Transactions on Knowledge and Data Engineering*, 24(11):1921–1936.

Manning, C. D., Raghavan, P., and Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.

Nagappan, M. and Vouk, M. A. (2010). Abstracting log lines to log event types for mining software system logs. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 114–117.

Taerat, N., Brandt, J., Gentile, A., Wong, M., and Leangsuksun, C. (2011). Baler: deterministic, lossless log message clustering tool. *Computer Science - Research and Development*, 26(3):285.

Tan, P.-N., Steinbach, M., and Kumar, V. (2005). *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc.

Tang, L., Li, T., and Perng, C.-S. (2011). Logsig: Generating system events from raw textual logs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM ’11, pages 785–794. ACM.

Tovarnak, D. (2017). Normalization of Unstructured Log Data into Streams of Structured Event Objects [online]. *Dissertation thesis*, Masaryk University, Faculty of Informatics, Brno. Available from <https://is.muni.cz/th/rjfzq/> [cit. 2019-05-10].

Vaarandi, R. (2003). A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations Management*, IPOM ’03, pages 119–126.

Vaarandi, R. (2004). *A Breadth-First Algorithm for Mining Frequent Patterns from Event Logs*, pages 293–308. Springer Berlin Heidelberg.

Vaarandi, R. (2008). Mining event logs with slct and loghound. In *NOMS 2008 - 2008 IEEE Network Operations and Management Symposium*, pages 1071–1074.

Vaarandi, R. and Pihelgas, M. (2015). Logcluster - a data clustering and pattern mining algorithm for event logs. In *Proceedings of the 2015 11th International Conference on Network and Service Management (CNSM)*, CNSM ’15, pages 1–7. IEEE Computer Society.

Xu, W., Huang, L., Fox, A., Patterson, D., and Jordan, M. I. (2009). Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP ’09, pages 117–132. ACM.

Yuan, D., Mai, H., Xiong, W., Tan, L., Zhou, Y., and Pasupathy, S. (2010). Sherlog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 143–154. ACM.

Yuan, D., Park, S., and Zhou, Y. (2012). Characterizing logging practices in open-source software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE ’12, pages 102–112. IEEE Press.