I J C T A, 9(15), 2016, pp. 7291-7305

Auditing System Application Logs Using Improved Iplom Algorithm

Murugan Vaiyapuri* and G. K. Sandhia*

ABSTRACT

The importance of event logs, as a source of information in systems and network management cannot be compromise. With the ever increasing size and complexity of today's event logs, the task of analyzing event logs has become difficult to carry out manually. For this reason recent research has focused on the automatic analysis of these log files. IPLoM (Iterative Partitioning Log Mining), a algorithm for the mining of clusters from event logs. Through a Three step hierarchical partitioning process IPLoM partitions log data into its respective clusters. In its 4th and final stage IPLoM produces cluster descriptions or line formats for each of the clusters produced. Unlike other similar algorithms IPLoM is not based on the Apriori algorithm and it is able to find clusters in data whether or not its instances appear frequently. Ambiguous Token Delimiters is one of the problem present in the IPLoM algorithm. Using the additional step in the IPLoM algorithm this problem can be overcome. This additional step is added along with fourth step. This improved IPLoM algorithm can be avoided the Ambiguous Token Delimiters problem.

Index Terms: Algorithms, experimentation, event log mining, fault management, clustering

1. INTRODUCTION

THE goal of autonomic computing as espoused by IBM's senior vice president of research, Paul Horn in March 2001 can be defined as the goal of building self-managing computing systems [1]. The four key concepts of self-management in autonomic computing are self-configuration, self-optimization, self-healing, and self-protection. Given the increasing complexity of computing infrastructure which is stretching to its limits the human capability to manage it, the goal of autonomic computing is a desirable one. However, it is a long-term goal, which must first start with the building of computing systems, which can automatically gather and analyze information about their states

To support decisions made by human administrators [1]. Event logs generated by applications that run on a system.

Consist of independent lines of text data, which contain information that pertains to events that occur within a system. This makes them an important source of information to system administrators in fault management and for intrusion detection and prevention. With regard to autonomic systems, these two tasks are important cornerstones for self-healing and self-protection, respectively. Therefore, as we move toward the goal of building systems that are capable of self-healing and self-protection, an important step would be to build systems that are capable of automatically analyzing the contents of their log files, in addition to measured system metrics [2], [3], to provide useful information to the system administrators.

The goal of autonomic computing as espoused by IBM's senior vice president of research, Paul Horn in March 2001 can be defined as the goal of building self-managing computing systems[5] . The four key concepts of self-management in autonomic computing are self-configuration, self-optimization, self-healing, and self-protection. Given the increasing complexity of computing infrastructure which is stretching to its limits the human capability to manage it, the goal of autonomic computing is a desirable one. However, it

* MTech CSE, SRM University, Chennai and Assistant Professor, SRM University

is a long-term goal, which must first start with the building of computing systems, which can automatically gather and analyze information about their states to support decisions made by human.

A basic task in automatic analysis of log files is message type extraction [4], [5], [6], and [7]. Extraction of message types makes it possible to abstract the unstructured content of event logs, which constitutes a key challenge to achieving fully automatic analysis of system logs. Message type descriptions are the templates on which the individual unstructured messages in any event log are built. Message types, once found, are useful in several ways:

- Compression. Message types can abstract the con-tents of system logs. We can therefore use them to obtain more concise and compact representations of log entries. This leads to memory and space savings.
- Indexing. Each unique message type can be as-signed an Identifier Index (ID), which in turn can be used to index historical system logs leading to faster searches. In [8], the authors demonstrated how message types can be used for log size reduction and indexing of the contents of event logs.
- Model building. The building of computational models on the log data, which usually requires the input of structured data, can be facilitated by the initial extraction of message type information. Message types are used to impose structure on the unstructured messages in the log data before they are used as input into the model building algorithm. In [9], [10], the authors demonstrate how message types can be used to extract measured metrics used for building computational models from event logs. The authors were able to use their computed models to detect faults and execution anomalies using the contents of system logs.
- Visualization. Visualization is an important component of the analysis of large data sets. Visualization of the contents of systems logs can be made more meaningful to a human observer by using message types as a feature of the visualization. For the visualization to be meaningful to a human

Observer, the message types must be interpretable. This fact provides a strong incentive for the production of message types that have meaning to a human observer.

To give an example of what message types are, consider this line of code:

sprintf(message, Connection from %s port %d, ipaddress, port number);

in a C program could produce the following log entries:

“Connection from 192.168.10.6 port 25” “Connection from 192.168.10.6 port 80” “Connection from 192.168.10.7 port 25” “Connection from 192.168.10.8 port 21.”

These four log entries would form a cluster (group) or event type in the event log and can be represented by the message type description (or line format):

“Connection from * port *.”

```
2005-06-03-15.42.50.823719 R02-M1-N0-C:J12-U11 RAS KERNEL INFO instruction cache parity error corrected
2005-06-03-15.42.50.982731 R02-M1-N0-C:J12-U11 RAS KERNEL INFO instruction cache parity error corrected
2005-06-06-22.41.37.357738 R20-M0-NA-C:J15-U11 RAS KERNEL INFO generating core.3740
2005-06-06-22.41.37.392258 R20-M0-NA-C:J17-U11 RAS KERNEL INFO generating core.3612
2005-06-11-19.20.25.104537 R30-M0-N9-C:J16-U01 RAS KERNEL FATAL data TLB error interrupt
2005-06-11-19.20.25.393590 R30-M0-N9-C:J16-U01 RAS KERNEL FATAL data TLB error interrupt
2005-07-01-17.52.23.557949 R22-M0-NA-C:J05-U01 RAS KERNEL INFO 458720 double-hummer alignment exceptions
2005-07-01-17.52.23.584839 R22-M0-NA-C:J03-U01 RAS KERNEL INFO 458720 double-hummer alignment exceptions
```

Figure 1: An example system log file. Each line represents an event.

The wildcards "*" represent message variables. We will adopt this representation in the rest of our work. Determining what constitutes a message type might not always be as simple as this example might suggest. Consider the following messages produced by the same print statement. "Link 1 is up," "Link 1 is down," "Link 3 is down," "Link 4 is up." The most logical message type description here is "Link * is *," however from a analysis standpoint having two descriptions "Link * is up" and "Link * is down" maybe preferable. There may also be other cases where messages produced by different print statements could form single logical message types. However, for the most part, message types will usually correspond to messages pro-duced by the same print statement, so we retain our representation for simplicity.

The goal of message type extraction is to find the representations of the message types that exist in a log file. This problem is well attested to in the literature but there is as yet no standard approach to the problem [9]. Techniques for automatically mining these line patterns from event logs have been based on the Apriori algorithm [11] for frequent item sets from data, e.g., Simple Log File Clustering Tool (SLCT) [12] and Loghound [13], or other line pattern discovery techniques like Teiresias [14] designed for other domains [7]. SLCT, Loghound, and Teiresias as algorithms are aimed toward the discovery of frequent textual patterns.

In this paper, we introduce Iterative Partitioning LogMining (IPLoM), a novel algorithm for the mining of eventtype patterns from event logs. Unlike previous algorithms, IPLoM is not primed toward the finding of only frequent textual patterns, but instead IPLoM's aim is to find all possible patterns. IPLoM works through a 3-step partitioning process, which partitions a log file into its respective clusters. In a fourth and final stage, the algorithm produces

A cluster description for each leaf partition of the log file. These cluster descriptions then become event type patterns discovered by the algorithm. IPLoM is able to find clusters in the data irrespective of the frequency of its instances and it scales gracefully in face of long message type patterns and it produces message type descriptions at a level of abstraction, which is preferred by a human observer. In our experiments, we compared the outputs of IPLoM, SLCT, Loghound, and Teiresias on seven different event log files, making up over 1 million log events, against message types produced manually on the event log files by our Faculty's tech support group. Results demonstrate that IPLoM consistently outperforms the other algorithms. It was able, in the best case, to produce approximately 70 percent of the manually produced message types compared to 36 percent for the best existing algorithm.

The rest of this paper is organized as follows: Section 2 discusses previous work in event type pattern mining and categorization. Section 3 outlines the proposed algorithm and the methodology to evaluate its performance. Section 4 describes the results whereas Section 5 presents the conclusion and the future work.

2. BACKGROUND AND PREVIOUS WORK

We begin this section by first defining some of the terminology used in this paper. We then discuss previous related work in the area of event log clustering and message type extraction.

2.1. Definitions

- Event log. A text-based audit trail of events that occur within the system or application processes on a computer system (Fig. 1).
- Event. An independent line of text within an event log which details a single occurrence on the system, (Fig. 2). An event typically contains not only a message but other fields of information like a Date, Source, and Tag as defined in thesyslog Request for Comment (RFC) [15]. For message type extraction, we are only interested in the message field of the event. This is why events are sometimes referred to in

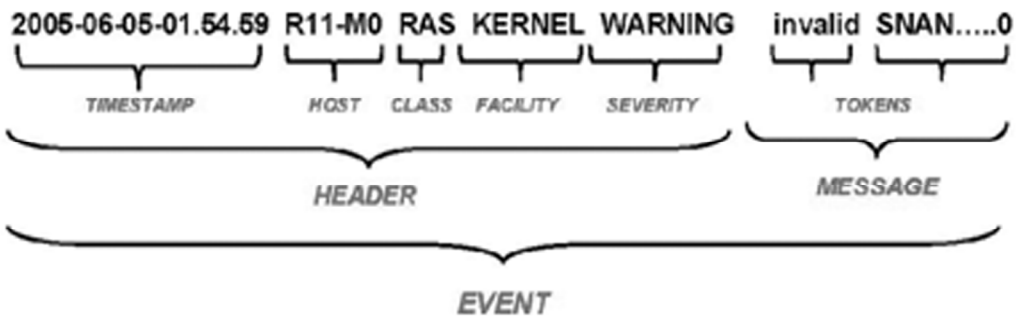

Figure 2: An example system log event.

Figure 2: An example system log event.

the literature as messages. In Fig. 2, the first five fields (delimited by whitespace) represent the Timestamp, Host, Class,

Which may contain more than 1 word up to a maximum value provided by the user. With both SLCT and Loghound, lines that do not match any of the frequent patterns discovered are classified as outliers.

SLCT and Loghound have received considerable attention and have been used in the implementation of the Sisyphus Log Data Mining toolkit [22], as part of the LogView log visualization tool [23] and in online failure prediction [24].

A comparison of SLCT against a bioinformatics pattern discovery algorithm developed by IBM called Teiresias [14] is carried out in [7]. Teiresias was designed to discover all patterns of at least a given specificity and support in categorical data.

In our work, we introduce IPLoM, a novel log-clustering algorithm. IPLoM works differently from the other clustering algorithms described above as it is not based on the Apriori algorithm and does not explicitly try to find line formats. The algorithm works by creating a hierarchical partitioning of the log data. The leaf nodes of this hierarchical partitioning of the data are considered clusters of the log data and they are used to find the cluster descriptions or line formats that define each cluster. Our experiments demonstrate that IPLoM outperforms SLCT, Loghound, and Teiresias when they are evaluated on the same data sets.

3. METHODOLOGY

In this section, we first give a detailed description of our proposed algorithm and our methodology for testing its performance against those of previous algorithms.

3.1. The IPLoM Algorithm

The IPLoM algorithm is designed as a log data clustering algorithm. It works by iteratively partitioning a set of log messages used as training exemplars. At each step of the partitioning process, the resultant partitions come closer to containing only log messages which are produced by the same line format. At the end of the partitioning process, the algorithm attempts to discover the line formats that produced the lines in each partition. These discovered partitions and line formats are the output of the algorithm.

Our main assumptions on the kind of event logs that IPLoM is suited for are the following:

1. The events in the log contain at least one field that is an unstructured natural language description of the event. These descriptions, which we call "messages," illustrated in Fig. 2, would naturally be produced by a set of "print" statements in a program source code.
2. The exact structure of these "messages" is unknown or not well documented.

Our work is therefore relevant to any log file where these assumptions are true, not just log files that meet the format of Fig. 2. Some application event logs are well structured and well documented, e.g., WebSphere. In such cases, message type extraction may not be necessary. For example, in [26], the authors provide a use case of process mining in web services using WebSphere, while in [27], the authors propose a tool for visualizing web services behavior by mining the contents of event logs. Process mining refers to the analysis of event logs as a way of monitoring adherence to business process rules. The analysis can be carried out without message type extraction due to the structured nature of the WebSphere logs, which utilize the Common Base Event model [21] for event representation.

An outline of the four steps of IPLoM is given in Fig. 3. The algorithm is designed to discover all possible line formats in the initial set of log messages and does not require a support threshold like SLCT or Loghound. As it may be sometimes required to find only line formats that have a support that exceeds a certain threshold, a file prune function (Algorithm 1) is incorporated into the algorithm. By removing the partitions that fall below the threshold value at the end of each partitioning step, we are able to produce only line formats that meet the desired support threshold at the end of the algorithm. The use of the file prune function is however optional. The following sections describe each step of the algorithm in more detail.

Algorithm 1. File_Prune Function: Prunes the partitions produced using the file support threshold.

Input: Collection C½& of log file partitions.

Real number F S as file support threshold. {Range for F S is assumed to be between 0 _ 1.}

Output: Collection C½& of log file partitions with support greater than F S.

1: for every partition in C do

2: Supp¼ $\frac{\neq \textit{Lines in Partition}}{\neq \textit{Lines in Collection}}$

3: if Supp< F S then

4: Delete partition from C½&

5: end if

6: end for

7: Return(C)

3.2. Step 1: Partition by Event Size

The first step of the partitioning process works on the assumption that log messages that have the same message type description are likely to have the same event size. For this reason, IPLoM's first step (Fig. 4) uses the event size heuristic to partition the log messages. By partition, we mean nonoverlapping groupings of the messages. It can be intuitively concluded that all the instances of this cluster, e.g., "Connection from255.255.255.255" and "Connection from 0.0.0.0" would alsocontain the same number of tokens. By partitioning our data first by event size, we are taking advantage of the property of most cluster instances of having the same event size. Therefore, the resultant partitions of this heuristic are likely to contain the instances of the different clusters, which have the same event size.

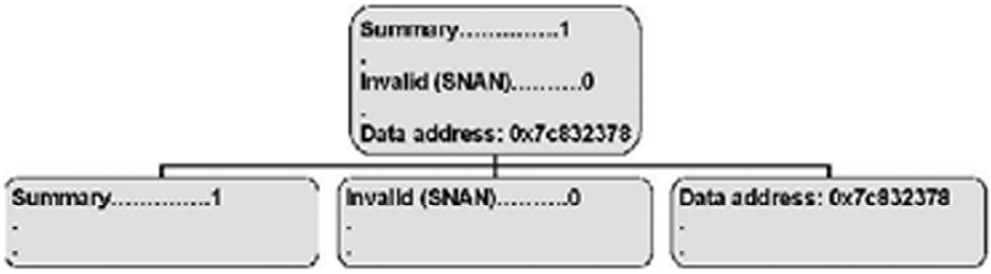

Figure 4: IPLoM Step 1: partition by event size. Separates the messages into partitions based on their event size.

Figure 5: IPLoM Step 2: partition by token position. Selects the token position with the least number of unique values, token position 2 in this example. Then, it separates the messages into partitions-based unique token values, i.e., "plb" and "address:", in the token position.

Sometimes, it is possible that clusters with events of variable size exist in the event log. This scenario is explained in more detail in Section 4.7.3.

Since IPLoM assumes that messages belonging to the same cluster should have the same number of tokens or event size, this step of the algorithm would separate such clusters. This does not occur too often, and variable size message types can still be found by post processing IPLoM's results. The process of finding variable size message types can be computationally expensive. Nevertheless, performing this process on the templates produced by IPLoM rather than on the complete log would require less computation.

3.3. Step 2: Partition by Token Position

At this point, each partition of the log data contains log messages, which are of the same size and can therefore be viewed as n-tuples, with n being the event size of the log messages in the partition. This step of the algorithm works on the assumption that the column with the least number of variables (unique words) is likely to contain words, which are constant in that position of the message type descriptions that produced them. Our heuristic is therefore to find the token position with the least number of unique values and further split each partition using the unique values in this token position, i.e., each resultant partition will contain only one of those unique values in the token position discovered, as can be seen in the example outlined in Fig. 5. A pseudo code description of this step of the partitioning process is given in Algorithm 2.

Algorithm 2. IPLoM Step 2: Selects the token position with the lowest cardinality and then separates the lines in the partition based on the unique values in the token position. Backtracks on partitions with lines that fall below the partition support threshold.

Input: Collection of log file partitions from Step-1.

Real number P ST as partition support threshold. {Range for P ST is assumed to be between 0_1.}

Output: Collection of log file partitions derived at Step-2 C In.

1: for every log file partition do {Assume lines in each partition have same event size.}

2: Determine token position P with lowest cardinality with respect to set of unique tokens.

3: Create a partition for each token value in the set of unique tokens that appear in position P .

4: Separate contents of partition based on unique token values in token position P . into separate partitions.

5: end for

6: for each partition derived at Step-2 do {}

7: if P SR < P S then

8: Add lines from partition to Outlier partition

9: end if

10: end for

11: File_Prune() {Input is the collection of newly createdpartitions}

12: Return() {Output is collection of pruned new partitions}

The memory requirement of unique token counting is a potential concern with the algorithm. While the problem of unique token counting is not specific to IPLoM, we believe IPLoM has an advantage in this respect. Since IPLoM partitions the database, only the contents of the partition being handled need be stored in memory. This greatly reduces the memory requirements of the algorithm. More-over, other workarounds can be implemented to further reduce the memory requirements. For example, in this Step 2 of the algorithm, by determining an upper bound (UB) on the lowest token count in Step 1, we can drastically reduce the memory requirements of this step, further counts of unique tokens in any token position that exceeds the upper bound can be eliminated. However, in this work, our aim is to make a proof of concept so we left the implementation of such code optimization techniques for future work.

Despite the fact that we use the token position with the least number of unique tokens, it is still possible that some of the values in the token position might actually be variables in the original message type descriptions. While an error of this type may have little effect on Recall, it could adversely affect Precision. To mitigate the effects of this error, a partition support ratio (PSR) for each partition produced could be introduced.

This partition has event size equal to 4. We need to select two token positions to perform the search for bijection on. The first token position has one unique token, {Command}. The second token position has two unique tokens, {has, failed}. The third token position has three unique tokens, {completed, been, on}. While the fourth token position has three unique tokens, {successfully, aborted, starting}. We notice in this example that token count 3 appears most frequently, twice, once in position 3 and once in position 4. The heuristic would therefore select token positions 3 and 4 in this example.

Algorithm 3. IPLoM Step 3: Selects the two token positions and then separates the lines in the partition based on the relational mappings of unique values in the token positions. Backtracks on partitions with lines that fall below the partition support threshold.

To summarize the steps of the heuristic, we first determine the number of unique tokens in each token position of a partition. We then determine the most frequently occurring token count among all the token positions. This value must be greater than 1. The token count that occurs most frequently is likely indicative of the number of message types that exist in the partition. If this is true, then a bijective relationship should exist between the tokens in the token positions that have this token count.

Despite the fact that we use the token position withthe least number of unique tokens, it is still possible thatsome of the values in the token position might actually bevariables in the original message type descriptions. Whilean error of this type may have little effect on Recall, it couldadversely affect Precision

Input: Collection of partitions from Step 2. {Partitions of event size 1 or 2 are not processed here}

Real number CT as cluster goodness threshold. {Range for CT is assumed to be between 0_1.}

Output: Collection of partitions derived at Step-3.

1: for every log file partition do

2: if CGR >¼CT then {See (2)}

3: Add partition to collection of output partitions

4: Move to next partition.

5: end if

6: Determine token positions using heuristic as P1 and P 2. {Heuristic is explained in the text. We assumetoken position P1 occurs before P2.}

7: Determine mappings of unique token values P1 in

respect of token values in P2 and vice versa.

8: if mapping is 1_1 then

9: Create partitions for event lines that meet each 1_1 relationship.

10: else if mapping is 1_M or M_1 then

11: Determine variable state of M side of relationship.

12: if variable state of M side is CONST ANT then

13: Create partitions for event lines that meet relationship.

14: else {variable state of M side is V ARIABLE}

15: Create new partitions for unique tokens in M side of the relationship.

16: end if

17: else {mapping is M_M}

18: All lines with meet M_M relationships are placed in one partition.

19: end if

20: end for

21: for each partition derived at Step-3 do {}

22: if P SR < P S then

23: Add lines from partition to Outlier partition

24: end if

25: end for

26: File_Prune() {Input is the collection of newly createdpartitions}

27: Return() {Output is collection of pruned new partitions}

Once the most frequently occurring token count value is determined, the token positions chosen will be the first two token positions, which have a token count value equivalent to the most frequently occurring token count.

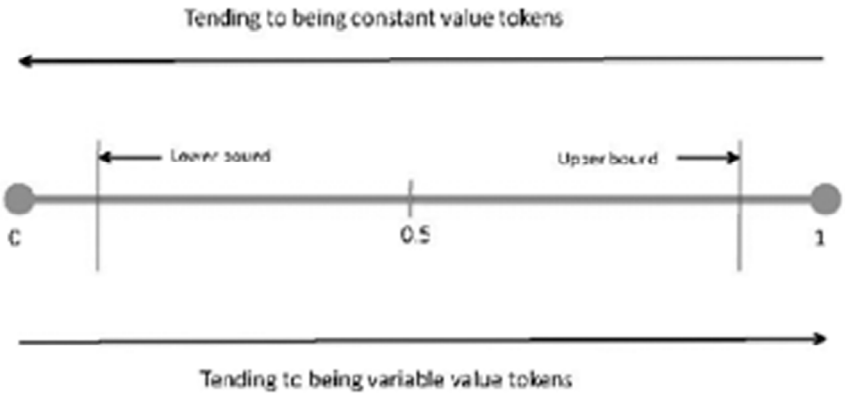

Figure 6: Deciding on how to treat 1-M and M-1 relationships. This procedure is implemented in the Get_Rank_Position function.

A bijective function is a 1-1 relation that is both injective and surjective. When a bijection exists between two elements in the sets of tokens, this usually implies that a strong relationship exists between them and log messages that have these token values in the corresponding token positions are separated into a new partition.

Privilegedand imprecise in position 2 also, respectively,contain the tokens instruction and exception in position 3 and vice versa. Consider the event messages given in Fig. 6 below to illustrate 1-M, M-1, and M-M relationships. If token positions 2 and 3 are chosen by the heuristic, we would have a 1-M relationship with tokens speeds, 3,552 and 3,311 as all lines that contain the tokenspeeds in position 2 have either tokens 3,552 or 3,311 in position 3, a M-1 relationship will be the reverse of this scenario. On the other hand, if token positions 3 and 4 are chosen by the heuristic, we would have a M-M relationship.

It is obvious that no discernible relationship can be found with the tokens in the chosen positions. Token 3,552 (inposition 3) maps to tokens 3,552 (in position 4) and 3,534. Onthe other hand, token 3,311 also maps to token 3,534, this makes it impossible to split these messages using their token relationships. It is a scenario like this that we refer to as a M-M relationship.

In the case of 1-M and M-1 relations, the M side of the relation could represent variable values (so we are dealing with only one message type description) or constant values (so each value actually represents a different message type description). The diagram in Fig. 8 describes the simple heuristic that we developed to deal with this problem. Using the ratio between the number of unique values in the set and the number of lines that have these values in the correspond-ing token position in the partition, and two threshold values, a decision is made on whether to treat the M side as consisting of constant values or variable values.

Before partitions are passed through the partitioning process of Step 3 of the algorithm, they are evaluated to determine if they already form good clusters. To do this, a cluster goodness ratio threshold (CGT) is introduced into the algorithm. The cluster goodness ratio (CGR) is the ratio of the number of token positions that have only one unique value to the event size of the lines in the partition, according to (2).

In the example in Fig. 7, the partition to be split has four token positions. Of these four, the first and second have only one unique value, i.e., “Program” and “Interrupt,” respectively. Therefore, the CGR for this partition will be $\frac{2}{4}$. Partitions that have a value higher than the CGT are considered good clusters and are not partitioned any further in this step. Just as in Step 2 the PSR can be used to backtrack on the partitioning at the end of Step 3.

CGR ¼

#T okenP ositionsW ithOneUniqueT okenInP artition : δ2Þ

Figure 7: IPLoM Step 4: discover message type descriptions. If the cardinality of the unique token values in a token position is equal to 1, then that token position is represented by that token value in the template. Else, we represent the token position with an "*".

3.4. Step 4: Discover Message Type Descriptions (Line Formats) from Each Partition

In this step of the algorithm, partitioning is complete and we assume that each partition represents a cluster, i.e., every log message in the partition was produced using the same line format. A message type description or line format consists of a line of text where constant values are represented literally and variable values are represented using wild-cards. This is done by counting the number of unique tokens in each token position of a partition. If a token position has only one value then it is considered a constant value in the line format, while if it is more than one then it is considered a variable. This process is illustrated in Fig. 9.

3.5. Algorithm Parameters

In this section, we give a brief overview of the parameters/ thresholds used by IPLoM. The fact that IPLoM has several parameters, which can be used to tune its performance, provides flexibility for the system administrators since this gives them the option of using their expert knowledge when they see it necessary

- File support threshold (FST). Ranges between [0,1]. It reduces the number of clusters produced by IPLoM. Any cluster whose instances have a support value less than this threshold is discarded. The higher this value is set to, the fewer the number of clusters that will be produced. This parameter is similar to the supportthreshold defined for SLCT and Loghound.
- Partition support threshold. Ranges between [0,1]. It is essentially a threshold that controls backtracking. Based on our experiments, the guideline is to set this parameter to very low values, i.e., <0:05, for optimum performance.
- Upper_bound and Lower_bound. Ranges between [0,1].

  M side of relationships in Step 2. Lower_Boundshould usually take values <0:5 while Upper_Bound takes values >0:5.
- Cluster goodness threshold. Ranges between [0,1]. It is used to avoid further partitioning. Its optimal setting should lie in the range of 0.3-0.6.

4 RESULTS

Our goal in the design of IPLoM was threefold. The first was to design an algorithm that is able to find all message types that may exist in a given log file. The second was to give every message type an equal chance of being found irrespective of the frequency of its instances in the data. Our third was to design an algorithm that will produce message types at an abstraction level preferred by a human observer. We therefore begin our discussion in this section by first describing the setup of our experiments in Section 4.1 and then providing

results that show how these goals have been met using a default scenario for running the algorithm, i.e., when we want to find all message types in Section 4.2. We also provide results on resource consumption (CPU and Memory) for the SLCT, Loghound and IPLoM in Section 4.2For SLCT and Loghound, this support value can be specified either as a percentage of the number of events in the event log or as an absolute value. For this reason, we run two sets of experiments using support values specified as percentages and as absolute values. In either case, we set these support values low because intuitively this allows for finding most of the clusters in the data, which is one of our goals. In Section 4.5, we present results of parameter sensitivity analysis. In Section 4.6, we present a case study testing IPLoM on the logs from one of the world's fastest supercomputers. In Section 4.7, we discuss our analysis of the performance limits of IPLoM.

4.1. Experimental Setting

All our experiments were run on an iMac7 desktop computer running Mac OS X 10.5.6. The machine has an Intel Core 2 Duo processor with a speed of 2.4 GHz and 2 GB of memory. In order to evaluate the performance of IPLoM, we selected open source implementations of algorithms previously used in system/ application log data mining. For this reason, SLCT, Loghound, and Teiresias were selected. We therefore tested the four algorithms against seven log data sets, which we compiled from different sources, Table 1 gives an overview of the data sets used. The message types in Table 1 were derived manually. The HPC log file is a publicly available data set collected on high performance clusters at the Los Alamos National Laboratory in New Mexico, USA [28]. The Access, Error, System, and Rewrite data sets were collected on our faculty network at Dalhousie, while the Syslog and Windows files were collected on servers owned by a large ISP working with our research group. Due to privacy issues, we are not able to make the Dalhousie and ISP data available to the public.

$$\text{F-Measure} = \frac{2\_\text{Precision}\_\text{Recall}}{\frac{1}{4}\text{Precision}^{p}\text{Recall}}^{5}{}^{p}$$

The parameter values used in running the experiments that produced our baseline results and the results in Sections 4.3 and 4.4 are provided in Table 2. The seed value for SLCT and Loghound is a seed for a random number generator used by the algorithms, all other parameter values for SLCT and Loghound are left at their default values. The parameters for Teiresias were also chosen to achieve the lowest support value allowed by the algorithm. The IPLoM parameters were all set empirically except in the case of the cluster goodness threshold and the partition support threshold.

In setting the cluster goodness threshold, we ran IPLoM on the HPC file while varying this value. The parameter was then set to the value (0.34) that gave the best result and was kept constant for the other files used in our experiments. On the other hand, the partition support threshold was set to 0 to provide a baseline performance. Such a setting for the performance threshold implies that no backtracking was done during partitioning.

It is pertinent to note that we were unable to test the Teiresias algorithm against all our data sets. This was due to its inability to scale to the size of our data sets. This is a problem that is attested to in [7], too. Thus, in this work, Teiresias could only be tested against the Syslog data set. The memory consumption results were obtained by monitoring the processes for each algorithm by using the Unixps and grep utilities.

4.2. Baseline Experiments

The result of our default evaluation of SLCT, Loghound, and IPLoM are shown in Fig. 10. The graphs in Fig. 10 visualize these results for Recall, Precision, and F-Measure metrics for all the algorithms using the

Table 1
Log Data Event Size Statistics

Name	Min	Max	Avg.
HIC	1	95	30.7
Syslog	1	25	4.57
Windows	2	82	22.38
Access	3	13	5.0
Error	1	41	9.12
System	1	11	2.97
Rewrite	3	14	10.1

TABLE 4
Algorithm Performance Based on Cluster Event Size

Event Size Range	No. of Clusters	Percentage Retrieved (%)			
		SLCT	Loghound	Loghound-2	IPL.M
1–10	316	12.97	13.29	49.68	53.80
11–20	142	7.04	9.15	35.92	49.30
> 21	68	15.15	16.67	77.27	51.52

“IR” evaluation method. Since one of our aims is to find all message types that may exist in a log file, we run this set of experiments with the lowest file support threshold possible, which is an absolute support value of 1. SLCT and Loghound would not work efficiently with an absolute support value of 1, so we run them with 2 instead. An absolute support value of 1 means every line/word will be considered frequent andthe result of the algorithms will be reduced to the case of finding unique lines for SLCT or the case of finding all possible templates for Loghound. Both situations are notvalidated desirable. Since Teiresias worked only on the Syslog data-set, its results are not included in our analysis. Utilizing the parameter values listed in Table 2, Teiresias produced a Recall performance of 0.1, a Precision performance of 0.04, which led to an F-Measure performance of 0.06 using the IR evaluation method. By providing “IR” evaluations that compare the results of the algorithms with manually produced results, we evaluate how well we have met the third design goal, which was to design an algorithm that will produce message types at an abstraction level preferred by a human observer.

Another cardinal goal in the design of IPLoM is theability to discover clusters in event logs irrespective of howfrequently its instances appear in the data. The performanceof the algorithms using this evaluation criterion is outlinedin Table 5. The results show a reduction in performance forall the algorithms for clusters with a few instances;however, IPLoM’s performance was more resilient.

For fairness in our comparison, we provide two different evaluations for Loghound. Loghound being a frequent item set mining algorithm is unable to detect variable parts of a message type when they occur at the tail end of a message type description. For example, if we intend to find the message type description “Error code: *,” it is possible for Loghound to find the message type description “Error code:” without the trailing variable at the end. In such a situation, we need to try that how come the user will get the error code as above as follows.

We however state that we provide these results for information purposes only. It is our belief that considering message type descriptions where the number of trailing variables cannot be assessed is detrimental to our goal of ensuring that we find only meaningful message types at an abstraction level preferred by a human observer. This interpretation means that we can no longer distinguish an instance of

the first message type from an instance of the second or third message types. So even though we have presented Loghound-2 results where the trailing variables are not used, we believe that inpractice one needs to use the trailing variables to distinguish the aforementioned differences. However, we wanted to give the benefit of doubt to Loghound for a fair comparison.

The average IR F-Measure performance across the data sets, at this default support level, is 0.07, 0.04, 0.10, and 0.46 for SLCT, Loghound, Loghound-2, and IPLoM, respectively. However, as stated in [29], in cases where data sets have relatively long patterns or low minimum support thresholds are been used, a priori-based algorithms incur nontrivial computational cost during candidate generation. The event size statistics for our data sets are outlined in Table 3, which shows the HPC file as having the largest maximum and average event sizeThis was however not a problem for SLCT (as it generates only 1 item sets). In terms of performance based on event size, Table 4 shows consistent performance from IPLoM irrespective of event size, while SLCT and Loghound seem to suffer for midsize clusters. Evaluations of Loghound considering the trailing variable problem shows Loghound achieving its best results for message types with a large event size and achieving results which are comparable to IPLoM in the other categories.

Another cardinal goal in the design of IPLoM is the ability to discover clusters in event logs irrespective of how frequently itsnstances appear in the data. The performance of the algorithms using this evaluation criterion is outlined in Table 5. The results show a reduction in performance for all the algorithms for clusters with a few instances; however, IPLoM's performance was more resilient.

The resource consumption results for SLCT, Loghound, and IPLoM are presented in Tables 6, 7, and 8. The tablesresults classification accuracy.The IPLoM algorithm is a lightweight

5. CONCLUSION AND FUTURE WORK

Due to the size and complexity of sources of information used by system administrators in fault management, it has become imperative to find ways to manage these sources of information automatically. Application logs are one such source. We present our work on designing a novel algorithm for message type extraction from log files, IPLoM. So far, there is no standard approach to tackling this problem in the literature [9]. Message types are semantic groupings of system log messages. They are important to system administrators, as they aid their understanding of the contents of log files. Administrators become familiar with message types over time and through experience. Our work provides a way of finding these message types automatically. In conjunction with the other fields in an event (host names, severity), message types can be used for more detailed analysis of log files.

Through a 3-step hierarchical partitioning process, IPLoM partitions log data into its respective clusters. In its fourth and final stage, IPLoM produces message type descriptions or line formats for each of the clusters produced. IPLoM is able to find message clusters whether or not its instances are frequent. We demonstrate that IPLoM produces cluster descriptions, which match human judgment more closely when compared to SLCT, Log-hound, and Teiresias. It is also shown that IPLoM demonstrated statistically significantly better performance than either SLCT or Loghound on six of the seven different data sets tested. These results however do not imply that SLCT and Loghound are not useful tools for event log analysis. They can still be useful for log analysis that involves other fields in an event. However, our results show that a specialized algorithm such as IPLoM can significantly improve the abstraction level of the unstructured message types extracted from the data.

Message types are fundamental units in any application log file. Determining what message types can be produced by an application accurately and efficiently is therefore a fundamental step in the automatic analysis of log files. Message types, once determined, which simplifies further processing steps like visualization or mathematical modeling, but also a way of labeling the individual terms (distinct word and position pairs) in the data.

Future work on IPLoM will involve using the information derived from the results of IPLoM in other automatic log analysis tasks which help with fault management. We also intend to implement an optimized version of IPLoM in a low level programming language such as C/C++, and make it publicly available on our website.[3] Lastly, our future work will continue on the integration of machine learning techniques and information retrieval with message type clustering in order to study automation of fault management and troubleshooting for computer systems.

REFERENCES

[1] J.O. Kephart and D.M. Chess, "The Vision of Autonomic Computing," Computer, vol. 36, no. 1, pp. 41-50, Jan. 2003.

[2] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, Indexing, Clustering, and Retrieving System History," Proc. 20th ACM Symp. Operating Systems Principles,

pp. 105-118, 2005.

[3] M. Jiang, M.A. Munawar, T. Reidemeister, and P.A. Ward, "Dependency-Aware Fault Diagnosis with Metric-Correlation Models in Enterprise Software Systems," Proc. Sixth Int'l Conf.Network and Service Management, pp. 137-141, 2010.

[4] M. Klemettinen, "A Knowledge Discovery Methodology for Telecommunications Network Alarm Databases," PhD dissertation, Univ. of Helsinki, 1999.

[5] S. Ma and J. Hellerstein, "Mining Partially Periodic Event Patterns with Unknown Periods," Proc. 16th Int'l Conf. Data Eng., pp. 205-214, 2000.

[6] Q. Zheng, K. Xu, W. Lv, and S. Ma, "Intelligent Search for Correlated Alarm from Database Containing Noise Data," Proc.Eighth IEEE/IFIP Network Operations and Management Symp.,

pp. 405-419, 2002.

[7] J. Stearley, "Towards Informatic Analysis of Syslogs," Proc. IEEEInt'l Conf. Cluster Computing, pp. 309-318, 2004.

[8] A. Makanju, A.N. Zincir-Heywood, and E.E. Milios, "Storage and Retrieval of System Log Events Using a Structured Schema Based on Message Type Transformation," Proc. 26th ACM Symp. AppliedComputing (SAC), pp. 525-531, Mar. 2011.

[9] W. Xu, L. Huang, A. Fox, D. Patterson, and M.I. Jordan, "Detecting Large-Scale System Problems by Mining Console Logs," SOSP '09:Proc. ACM SIGOPS 22nd Symp. Operating Systems Principles,

pp. 117-132, 2009.

[10] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis," Proc. Ninth IEEE Int'l Conf. Data Mining (ICDM '09),

pp. 149-158, Dec. 2009.

[11] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Associa-tion Rules," Proc. 20th Int'l Conf. Very Large Data Bases (VLDB),

J.B. Bocca, M. Jarke, and C. Zaniolo, eds., pp. 487-499, 1994.

[12] R. Vaarandi, "A Data Clustering Algorithm for Mining Patterns from Event Logs," Proc. IEEE Workshop IP Operations and Manage-ment, pp. 119-126, 2003.

[13] R. Vaarandi, "A Breadth-First Algorithm for Mining Frequent Patterns from Event Logs," Proc. IFIP Int'l Conf. Intelligence inComm. Systems, vol. 3283, pp. 293-308, 2004.

[14] I. Rigoutsos and A. Floratos, "Combinatorial Pattern Discovery in Biological Sequences: The TEIRESIAS Algorithm," Bioinformatics, vol. 14, pp. 55-67, 1998.

[15] C. Lonvick, "The BSD Syslog Protocol," RFC3164, Aug. 2001.

[16] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan, "Automatic Subspace Clustering of High Dimensional for Data Mining Applications," Proc. ACM SIGMOD Int'l Conf. Managementof Data, 1998.

[17] S. Guha, R. Rastogi, and K. Shim, "CURE: An Efficient Clustering Algorithm for Large Databases," Proc. ACM SIGMOD Int'l Conf.Management of Data, pp. 73-84, 1998.

[18] S. Goil, H. Nagesh, and A. Choudhary, "MAFIA: Efficient and Scalable Subspace Clustering for Very Large Data Sets," technical report, Northwestern Univ., 1999.

[19] J.H. Bellec and M.T. Kechadi, "CUFRES: Clustering using Fuzzy Representative Event Selection for the Fault Recognition Problem in Telecommunications Networks," Proc. ACM First PhD Workshopin CIKM, pp. 55-62, 2007.

[20] T. Li, F. Liang, S. Ma, and W. Peng, “An Integrated Framework on Mining Log Files for Computing System Management,” Proc.ACM SIGKDD Int’l Conf. Knowledge Discovery in Data Mining (KDD ’05), pp. 776-781, 2005.

[21] B. Topol, “Automating Problem Determination: A First Step Toward Self Healing Computing Systems,” IBM White Paper, http://www-106.ibm.com/developerworks/autonomic/library/ ac-summary/ac-prob.html, Oct. 2003.

[22] J. Stearley, “Sisyphus Log Data Mining Toolkit,” http://www. cs.sandia.gov/sisyphus, Jan. 2009.

[23] A. Makanju, S. Brooks, N. Zincir-Heywood, and E.E. Milios, “Logview: Visualizing Event Log Clusters,” Proc. Sixth Ann. Conf.Privacy, Security and Trust (PST), pp. 99-108, Oct. 2008.