CAPRI: A Tool for Mining Complex Line Patterns in Large Log Data

Farhana Zulkernine, Patrick Martin,
Wendy Powley, Sima Soltani
School of Computing, Queen's University
Kingston, ON, Canada K7L 3N6
{farhana, martin, wendy, soltani}@cs.queensu.ca

Serge Mankovskii[*], Mark Addleman[#]
[*]CA Labs, Toronto, ON, Canada
[#]CA Technologies Inc., San Francisco, USA
[*]serge.mankovski@ca.com,
[#]mark.addleman@ca.com

ABSTRACT
Log files provide important information for troubleshooting complex systems. However, the structure and contents of the log data and messages vary widely. For automated processing, it is necessary to first understand the layout and the structure of the data, which becomes very challenging when a massive amount of data and messages are reported by different system components in the same log file. Existing approaches apply supervised mining techniques and return frequent patterns only for single line messages. We present CAPRI (type-CAsted Pattern and Rule mIner), which uses a novel pattern mining algorithm to efficiently mine structural line patterns from semi-structured multi-line log messages. It discovers line patterns in a type-casted format; categorizes all data lines; identifies frequent, rare and interesting line patterns, and uses unsupervised learning and incremental mining techniques. It also mines association rules to identify the contextual relationship between two successive line patterns. In addition, CAPRI lists the frequent term and value patterns given the minimum support thresholds. The line and term pattern information can be applied in the next stage to categorize and reformat multi-line data, extract variables from the messages, and discover further correlation among messages for troubleshooting complex systems. To evaluate our approach, we present a comparative study of our tool against some of the existing popular open-source research tools using three different layouts of log data including a complex multi-line log file from the z/OS mainframe system.

Categories and Subject Descriptors
I.5.3 [**Clustering**] and I.5.4 [**Applications**]: Algorithms and text processing tool for line, term and value pattern mining.

General Terms
Algorithms, Design, Experimentation.

Keywords
Line, term, value pattern mining, type-casting, association rule.

BigMine'13, August 11-14 2013, Chicago, IL, USA Copyright is held by the owner/author(s).Publication rights licensed to ACM.
ACM 978-1-4503-2324-6/13/08…$15.00.

1. INTRODUCTION
Log data analysis is crucial to retrieve knowledge regarding the system status when troubleshooting performance issues. With the evolution of various sensors and system monitoring tools, we come across many different log files with varying data layouts. The files are often very large and contain a mixture of many different pieces of information. The analysis of the data, therefore, has become more challenging than ever and involves multiple steps based on the information requirements and data contents. The first step in a complex log data analysis process naturally leads to discovering line patterns and understanding frequent data values in the log files to enable further processing in an automated or semi-automated manner. The application of line pattern mining has recently been quite prevalent in IDS (Intrusion Detection System) alert filtration [11][9] and event categorization for software maintenance and error diagnosis [3][5][7]. Other application areas include autonomic systems management [4][12], decision support systems [13]; anomaly detection by identifying abnormal system behavior [3]; fault detection by comparing with predefined fault characteristics [10], and prediction of error situations using techniques such as case-based reasoning, sequence mining or statistical approaches [3].

The work described here is a part of a larger research effort [14], which is aimed at providing a decision support framework for database administrators (DBA) to help troubleshoot problems in complex real world systems such as the z/OS mainframe DB2. Fig.1 shows an example of complex multi-line log messages from a z/OS mainframe DB2 JES (Job Entry Subsystem) master log file where many different monitoring agents report events in various formats [14]. The DBAs often have to inspect such large files manually for troubleshooting purposes. A mandatory step in automating the analysis of complex log files is to first understand the structural aspect of the line patterns in the data, which should include both frequent and rare line patterns; the contextual relationship between different line patterns, and frequent value patterns. This knowledge can then be used to automate the next step of data processing and knowledge discovery such as reformatting, content analysis, message classification, variable identification, and correlation analysis as necessary for specific application domains [3].

Although frequent itemset mining and text data mining have been popular areas of research in the last decade, a literature study reveals that most of the existing data mining approaches for semi-structured text data such as log files apply supervised learning methods [3] for job-specific limited size log files. Some of the more recent works demonstrate application of unsupervised learning methods [11]. However, the common assumptions in the existing approaches are that the messages are single line with a

```
1   J E S 2  J O B  L O G  --  S Y S T E M  Z P 0 1  --  N O D E  U S I L 0 P 0 1
2   19.00.25 STC10799 ---- SATURDAY,  18 DEC 2010 ----
3   03.58.05 STC09048   DSNJ110E  !SS11 LAST COPY 1 ACTIVE LOG DATA
          SET IS 95 PERCENT FULL
4   12.46.21 STC10799  DSNP010I  !PB11 END OF MESSAGES.  744
5     744  CONNECTION-ID=SERVER
6     744  CORRELATION-ID=PB1BTC008
7   12.17.58 STC10799  DSNV512I  !PB11 DSNVMON - AGENT 1:  771
8     744  LUW-ID=ODCA4760.EC77.C737D7D90AA5=181319
9     771  NAME     ST A   REQ ID          AUTHID   PLAN     ASID TOKEN
10    771  ----     -- -   --- --          ------   ----     ---- -----
11    771  SERVER   RA * 10879 PB1DIA025 PB1ADM DISTSERV 0070 67702
12    771  LONG 9728K VLONG 436K 64BIT 28784K
```

Figure 1. Data type A- z/OS mainframe DB2 JES master log file containing multi-line messages

time stamp and the formats of the parameters are known from heuristics. In this research, we address the challenges in processing complex multi-line messages in large log data.

The JES log file in z/OS mainframe is used by the DBAs as an important reference. We use the JES log file in Fig. 1 containing multi-line messages as a running example in the paper. The end of the first line of a multi-line message contains a line number that appears in the beginning of the subsequent lines of the same message, for example lines 4-8 excluding line 7. Additional complexity arises from the facts that a) day and time are expressed in separate messages (date in line 2 and time as *hh:mm:ss* in the beginning of each message); b) messages can be in table format (lines 9-11); c) the parameters can be of various formats, and d) the lines may be ordered incorrectly as lines 7 and 8 due to multiple reporting agents writing concurrently in the same file.

Existing research tools such as SLCT [10], LogHound [11], and IPLoM [7] have the following shortcomings: they fail to identify rare line patterns, they cannot process large multi-line log data, and they do not retrieve the frequent term patterns or the contextual relationship between commonly occurring pairs of line patterns. Our research is, therefore, motivated towards resolving these shortcomings.

We propose CAPRI (type-CAsted Pattern and Rule mIner), a novel algorithm and tool [2] that makes a minimal number of passes over semi-structured log data to detect line patterns using unsupervised clustering and a type-casting technique [ref. Sec. 3.1]. It has the following features:

a) Processes both single and multi-line messages.
b) Generates a compact list of frequent, rare, and interesting line patterns expressed in a type-casted format.
c) Generates a list of frequent term and value patterns given the minimum support values.
d) Generates rules that identify contextual relationships between subsequent line patterns.
e) Performs incremental mining for big data and uses a bitmap matrix to compute the support counts in a single pass over the data making efficient use of system resources.

The next section of the paper describes the related work. Definitions of some key concepts are presented in Section 3. Section 4 presents our approach in detail. We evaluate CAPRI against existing popular line pattern mining tools. Experimental results are provided in Section 5 to demonstrate the discovered line, term and value patterns and the rules from a set of test data. Section 6 concludes the paper summarizing the contributions and our future work plan.

2. RELATED WORK

Frequent pattern and association rule mining for semi-structured text data has been explored by many researchers [3]. Classification and clustering techniques for mining frequent patterns have been used for troubleshooting systems [4], anomaly detection [12], and software maintenance [5]. Most of these approaches apply supervised learning based on labelled training data, which is effective when there are a limited number of message formats that can be labelled. In complex systems and the big data paradigm this is infeasible. Hellerstein et al. [4] propose a framework to mine event bursts, periodic and dependent event patterns, and event attributes and correlations from logs that can be used to take corrective actions for systems management. Xu et al. [12] demonstrate a console log mining approach where message patterns are extracted by analyzing source code which is generally not accessible. In the anomaly detection paradigm, Fu et al. [3] propose an algorithm to categorize log messages by a generated text key without application specific knowledge. The approach assumes single line log messages each having a timestamp, a thread ID and a request ID. Jiang et al. [5] propose a clone detection based log abstraction technique using heuristics to identify parameters from constant terms, and thereby, understand the structures of log messages.

Some of the more recent line pattern mining approaches use unsupervised learning techniques but assume only single line messages with timestamps. Applications of these approaches can be found in categorizing and filtering important messages from IDS event logs [11][9], which contain a huge number of alerts. Vaarandi [10] proposes SLCT (Simple Log Clustering Tool) and LogHound tools to find only frequent message clusters in log data. Stearly [8] proposes the Sisyphus toolkit combining Teiresius and other components, which enables automated generation of maximal message patterns for categorization and grouping of time-correlated messages and interactive reviewing of the results. The approach requires some knowledge about the data and generates a separate output for the outliers. Both Teiresius and LogHound require that the data reside completely in memory. SLCT avoids the memory problem by requiring multiple passes over the data. Makanju et al. [7] propose IPLoM (Iterative Partitioning Log Mining), which uses a repeated data partitioning approach to first create clusters with messages having equal number of data items in each line and then repartitioning the clusters based on other criteria.

All of the above approaches apply value-based clustering. We move up a level in abstraction by introducing a type-casting mechanism and generating a more condensed set of type-casted line patterns using unsupervised learning and incremental mining techniques. CAPRI also generates a list of frequent term and value patterns and rules that state how the patterns are correlated in multi-line log messages.

3. DEFINITIONS

3.1 The Type-casting Technique

We identify 3 types of characters: alphabetic characters (a-z, A-Z), numeric characters or numbers (0-9) and symbols (anything other than the above two types). Literals refer to any combination of the above character types. The *Type Casting Technique* uses the type and the number of characters to map a word (space delimited data) to a type-casted format but leaves the symbols intact because they typically stand out in the pattern. We convert a

number to "i?" where "?" indicates the number of digits in that number. Similarly alphabetic literals are converted to "c?". Symbols are not modified. Therefore, "03.58.05 STC09048 DSNJ110E" in Fig. 1 line 3 is type-cast to "i2.i2.i2 c3i5 c4i3c1". This conversion allows us to represent data such as date and time, which do not have any frequent values, in a common format and thus capture additional semantic information about the data items.

3.2 Line, Term, and Value Patterns

A *Term* is a sequence of characters (alphabetic, numeric or symbol or any combination of these types) that is delimited by a space, or an end of line character depending on the position of the sequence in a line of data. A *Term Pattern* denotes the pattern of a type-cast term such as "i2.i2.i2" for all timestamp values like "19.00.25" in Fig. 1. A *Value of a Term* denotes a concrete instance of the term as it appears in the data such as "19.00.25". A term or term pattern can have multiple values such as the time data. A *Value Pattern* is a sequence of frequent characters in a term and '?', where a frequent character remains as is in its position and an infrequent character is replaced by a "?". For example, "12.??.??" is a value pattern for the term "i2.i2.i2". A *Line* is a sequence of terms which are separated by one or more spaces and is terminated by a line feed in the text data. A *Line Pattern* is a line composed of its term patterns where frequent terms remain the same at their positions but infrequent terms get replaced by "*". Consecutive "*"s are merged into one "*" to provide a concise representation of the line pattern. An example line pattern for line 3 of Fig. 1 would be "i2.i2.i2 c3i5 *", considering that the terms after "c3i5" are not frequent. More examples are given in Sections 4 and 5.

3.3 Frequent Patterns and Support Values

The support value of a pattern P in a dataset D containing semi-structured text data is:

$$Support\ (P) = \frac{Number\ of\ data\ lines\ containing\ P}{Number\ of\ data\ lines\ |D|} \quad \ldots\ldots\ldots (1)$$

where $D = \{L_1, L_2, \ldots, L_n\}$ the set of all lines L_i in the data, and $i \in (1, n)$, n=the maximum number of lines in the data. $LT_i = \{T_{i1}, T_{i2,\ldots}, T_{ij,\ldots}, T_{im}\}$ is the set of all type-casted term patterns in line L_i, where, $j \in (1, m)$, m=the maximum number of terms in line i. If $Support\ (P) > \varepsilon$, where ε is a user defined minimum support threshold then P is called a frequent pattern.

A line pattern $Pl \subseteq LT_i$ and it is composed of one or more type-casted term patterns. In the context of semi-structured text data, the position of the term in the line must be considered in computing its support. Therefore, a term pattern $Pt_j = \langle T_{ij}, j \rangle$, where $T_{ij} \in LT_i$ represents a term at the j^{th} position of i^{th} data line. The support for value patterns are computed in the same way as the term patterns. Here $L_i = \{V_{i1}, V_{i2}, \ldots, V_{im}\}$ which represents actual values of the terms instead of type-casted terms and a value pattern $Pv_j = \langle T_{ij}, V_{ij}, j \rangle$. However, the absolute support counts are often used instead of fractions as we use in Sections 4 and 5.

3.4 Closed Frequent Pattern

A pattern $\propto$ is a *Closed Frequent Pattern (CFP)* in a dataset D if $\propto$ is frequent in D and there exists no proper super-pattern β such that $\propto \subset \beta$ and β has the same support as $\propto$ in D [3]. If a pattern is frequent, each of its sub-patterns is also frequent. CFPs restrict the number of sub-patterns of a frequent pattern without losing any information.

3.5 Association Rule Mining

Rules comprise two parts namely, a condition and an implication. Given a set of items $I = \{i_1, i_2, i_3, \ldots\}$, a rule is generally expressed as $\propto \Rightarrow \beta$, where $\propto \subset I$, $\beta \subset I$, and $\propto \cap \beta = \emptyset$. $\propto$ is called the *antecedent* and β is called the *consequent* of the rule. In information retrieval, *association rule mining* is very effective in discovering the knowledge about relationships among different information pieces and establishing correlations between different data dimensions i.e., data items in a data set. We take each pair of patterns that occur together and find which terms have the same values to explore the contextual relationship. Our rule, therefore, resembles $(p_x\text{-}p_y) \Rightarrow (l_{xi}=l_{yj})$. We also derive the support and confidence values to determine the acceptability of the rules.

3.6 Support and Confidence of Rules

In general the support for a rule $\propto \Rightarrow \beta$ is:

$$Support\ (\propto \Rightarrow \beta) = \frac{Number\ of\ data\ lines\ containing\ (\propto \cup \beta)}{Number\ of\ data\ lines\ |D|} \quad \ldots (2)$$

$$Confidence\ (\propto \Rightarrow \beta) = \frac{Support\ (\propto \cup \beta)}{Support(\propto)} \quad \ldots\ldots\ldots\ldots (3)$$

where $Support(\propto)$ is computed using equation (1). For each pair of lines l_x and l_y occurring together in the data where l_y follows l_x, we mine association rules between their corresponding line patterns p_x and p_y to understand the contextual relationships between the different terms of these line patterns. We express the rule as $(p_x\text{-}p_y) \Rightarrow (l_{xi}=l_{yj})$, which states that if the line pattern p_y, follows the line pattern p_x, then the value of the i^{th} term, l_{xi} of the former line l_x matches the value of the j^{th} term l_{yj} of the following line l_y. The support for $(p_x\text{-}p_y)$ is calculated by the ratio of the total number of times p_y follows p_x, and the total number of line pairs in the data, which is (|D|-1), |D| being the total number of lines in the data set. The confidence of a rule is calculated using Eq. 3. For example, if (p1-p2) is true 10 times in 101 lines of data in total, then the support of the rule is 0.1. If out of these 10 times, every time term p1.T1=p2.T2 then the confidence of the rule is 1.0. A small deviation (5%) in the confidence of a rule indicates possible incorrect ordering of lines or incorrect data formats, and helps in correcting the data.

4. OUR APPROACH – CAPRI

4.1 Overview

Given the fact that most event reports contain messages that have an initial header part with a timestamp and some other ID, followed by a body part containing the detailed message, we devise a technique that sorts type-casted data to generate clusters of similar line formats. The type-casting technique abstracts the variation in data but maintains the structural data pattern.

CAPRI uses *incremental mining* to handle large data files, mines *frequent line, term and value patterns* including the *rare* and *interesting* line patterns, and the *contextual relationships* between subsequent line patterns in the data. In this paper, we define a simple criterion for *interestingness*, which is the subsequent appearance of three or more symbols in a line pattern because up to two symbols (e.g. "::") are used in the text or in parameter values. In Fig.1 line 1 is rare as it appears only once in a file, and line 2 is rare and also interesting as it contains "---". CAPRI successfully identifies the line patterns, categorizes and labels each line of data with a line pattern ID and optionally rewrites labeled lines in an output file. In the following sections we describe the algorithm in detail focusing on its key features.

4.2 Pattern Extraction

4.2.1 Generating Type-Casted and Sorted Data

As the data is read, it is converted into the type-casted format, and both the original and converted data are stored in a vector array in memory. Fig. 2 shows the type-casted data of Fig. 1. The type-casted data of Fig. 2 in the vector array is then sorted as in Fig. 3.

```
1   i1 c1 c1 c1 i1 c1 c1 c1 c1 c1 -- c1 c1 c1 c1 c1 c1 c1 c1 i1 i1 -- c1 c1 c1
        c1 c1 c1 c1 c1 i1 c1 i1 i1
2   i2.i2.i2 c3i5 ---- c8, i2 c3 i4 ----
3   i2.i2.i2 c3i5 c4i3c1 !c2i2 c4 c4 i1 c6 c3 c4 c3 c2 i2 c7 c4
4   i2.i2.i2 c3i5 c4i3c1 !c2i2 c3 c2 c8. i3
5       i3 c10-c2=c6
6       i3 c11-c2=c2i1c3i3
7   i2.i2.i2 c3i5 c4i3c1 !c2i2 c7 - c5 i1: i3
8       i3 c3-c2=c4i4.c2i2.c1i3c1i1c1i2c2i1=i6
9       i3 c4 c2 c1 c3 c2 c6 c4 c4 c5
10      i3 ---- -- - --- -- ------- ---- ---- -----
11      i3 c6 c2 * i5 c2i1c3i3 c2i1c3 c8 i4 i5
12      i3 c4 i4c1 c5 i3c1 i2c3 i5c1
```

Figure 2. Type-casted line data corresponding to Fig. 1

```
1   i1 c1 c1 c1 i1 c1 c1 c1 c1 c1 -- c1 c1 c1 c1 c1 c1 c1 c1 i1 i1 -- c1 c1 c1
        c1 c1 c1 c1 c1 i1 c1 i1 i1
2   i2.i2.i2 c3i5 ---- c8, i2 c3 i4 ----
3   i2.i2.i2 c3i5 c4i3c1 !c2i2 c3 c2 c8. i3
4   i2.i2.i2 c3i5 c4i3c1 !c2i2 c4 c4 i1 c6 c3 c4 c3 c2 i2 c7 c4
5   i2.i2.i2 c3i5 c4i3c1 !c2i2 c7 - c5 i1: i3
6       i3 ---- -- - --- -- ------- ---- ---- -----
7       i3 c10-c2=c6
8       i3 c11-c2=c2i1c3i3
9       i3 c3-c2=c4i4.c2i2.c1i3c1i1c1i2c2i1=i6
10      i3 c4 c2 c1 c3 c2 c6 c4 c4 c5
11      i3 c4 i4c1 c5 i3c1 i2c3 i5c1
12      i3 c6 c2 * i5 c2i1c3i3 c2i1c3 c8 i4 i5
```

Figure 3. Sorted type-casted line vector of data in Fig. 2.

4.2.2 Bitmap Matrix Generation

A *bitmap matrix*, as shown in Fig. 4, is used in our algorithm to compute the support for discovering closed frequent patterns *in one pass* over the data in memory. It is created by comparing the terms t_{ij} and $t_{(i+1)j}$ of subsequent rows *i* and *(i+1)* in the j^{th} column of the sorted array (Fig. 3) as stated in Eq. (5). M_{ij} is the resulting 0 or 1 as shown in the bitmap matrix of Fig. 4.

$$M_{ij} = (t_{ij} == t_{(i+1)j}) \quad \text{…….. (5)}$$

For example, line 2 of the matrix in Fig. 4, 1100…0, is the result of the comparison of the terms in lines 2 and 3 in Fig. 3. Since two lines of the data generate one line of the bitmap matrix, the total number of lines in the matrix would be one less than that of the vector array. We use the last row of the matrix to store the result of the bit-multiplication as explained next.

4.2.3 Bit Multiplication and Counting Support

As stated in Eq. (6), starting with the first row of the bitmap matrix, we multiply the bits M_{ij} and $M_{(i+1)j}$ in the subsequent rows in the same column *j* using a logical "AND" operation and the result is stored in MM_j in the last row. Line 12 in Fig. 4 shows the result of multiplying lines 2 and 3.

$$MM_j = M_{ij} * M_{(i+1)j} \quad \text{…….. (6)}$$

$MM_j=1$ indicates that last 3 consecutive terms in *j* in Fig. 3 are the same. We use a one dimensional array to save the count of 1s in the multiplication results for each column of the matrix as shown by the last column labeled *Counter(CC)* in Fig. 4. The count is increased if the multiplication result MM_j is 1 as shown below.

	1 2 3 4 5 6 7 8 9	Counter (CC)
1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0000…
2	1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1100…
3	1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	2211…
4	1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	2211…
5	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0000…
6	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1000…
7	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	2000…
8	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	3000…
9	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	4000…
10	1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	5000…
11	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	
12	**1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0**	***MM_j***

Figure 4. Bitmap matrix for the data shown in Fig. 3. Bit multiplication result is temporarily stored in the last row *MM*, where $MM_j = M_{ij} * M_{(i+1)j}$. *CC* keeps the count of consecutive 1's in each column

$$\textit{If } MM_j==1 \textit{ then } CC_j \mathrel{+}= 1 \textit{ else checkSave()} \quad \text{…….. (7)}$$

The algorithm is shown in Fig. 5. In Fig. 4, the result of the multiplication of lines 2 and 3, is shown in the last row and CC_j is "1100…" as the previous multiplication result was all 0s. For lines 3 and 4, the multiplication result is "111100…". Therefore, CC_j is updated to "221100…". *CC_j=n indicates that (n+2) terms at column j are the same in consecutive rows in the sorted type-casted data*. In Fig. 4 line 4 when $MM_j=0$, $CC_1=2$ means 4 terms at column 1 are the same up to the next line (see Fig. 3 lines 2-5).

In Eq. (7), when $MM_j=0$ the counter is checked to see if its value exceeds the minimum line (*minLineSup*) and term support thresholds (*minTermSup*). If a minimum threshold is reached or exceeded, an appropriate save pattern operation is invoked (Fig. 5 lines 19-27) as described in the next section. Otherwise, the counter is reset (Fig. 5 line 26) to 0.

Algorithm: Extract Patterns

```
1   Sort type-casted data stored in the vector array
2   Create the Bitmap-Matrix where Mij = (tij == t(i+1)j)
3   Reset CCj = 0
4   saveFlag = 0
5   For each row of matrix until (totrows-2)
6       saveFlag = 0
7       For each column
8           Compute Bit Multiplication (MMj = Mij * M(i+1)j )
9           If (MMj == 1)
10              Increment bit counter CCj += Mij* M(i+1)j
11              If ( lastLine )
12                  If ( checkSave(CCj, Mij, MMj, minTermSup) >0)
13                      Save frequent term pattern
14                      Check and save frequent value pattern
15                  Endif
16                  saveFlag = (checkSave(CCj,Mij,MMj,minLineSup)>0)
17              Endif
18          Else
19              If ( checkSave(CCj, Mij, MMj, minTermSup) >0)
20                  Save frequent term pattern
21                  Check and save frequent value pattern
22              Endif
23              If (checkSave(CCj,Mij,MMj,minLineSup)>0)
24                  saveFlag = 1
25              else
26                  Reset CCj =0
27              Endif
28          Endif
29      Endfor
30      saveLabelLinePatterns
31  Endfor
32  saveLabelRareInterestingLinePatterns
```

Figure 5. Extracts line, term and value patterns.

Note that each line of the bitmap matrix is created from two lines of data. An additional check is made when $MM_j=1$ (Fig. 5 line 11-17) for the last row (lastline=*totrows-2* where totrows=total number of data lines) because if a column contains all 1's until the last line, MM_j is never 0 and the line pattern does not get saved (Fig. 5 line 11). Also the bit multiplication compares 3 consecutive lines or more. Therefore, the last two lines of data have to be checked separately for frequent 2 term or line patterns.

4.2.4 Saving Term, Value and Line Patterns

The algorithm maintains frequent line, term and value pattern lists in memory. The checkSave() function returns 1 (Fig.5 lines 12, 16, 19, 23) if the support count (CC_j+2) equals or exceeds the minimum threshold (for terms *minTermSup* and for lines *minLineSup*) to indicate that a frequent pattern must be saved. If the pattern does not exist already, it is added to the corresponding pattern list, otherwise only its support count is updated. A *frequent term pattern FTP* is represented as <*ft, tp, ts*> where *ft* is the type-casted term pattern, *tp* is its position in the line and *ts* is its support. For *FTP*s, the actual values are inspected further to find the frequent value patterns (Fig. 5 line 14).

A *frequent value pattern FVP* is represented as <*ft, tp, VP*> where *ft* is the frequent term pattern and *tp* is its position in the line. *VP* = {<fv_1,vs_1>, <fv_2,vs_2>,…, <fv_n,vs_n>} is the set of all value patterns listed as a pair <fv_i,vs_i>, fv_i being the value pattern and vs_i being its corresponding support. To check frequent value patterns, the process starts at the row of the vector array where the term match started; extracts all the different values for that term from the corresponding row of unmapped data into an array; sorts the values, and follows the same strategy as described above using the bitmap matrix and bit multiplication technique. The counter value CC_j here indicates the number of times a character appears in subsequent rows in the j^{th} column. If the character count (CC_j+2) equals or exceeds the *minValueSup* then a closed frequent value pattern is saved.

A *frequent line pattern FLP* is represented as <*pid, lp, ls*> where *pid* is the pattern ID, *lp* is the type-casted line pattern and *ls* is the support count. A line pattern is generated once all its terms have been examined and if flagged for saving by checkSave() based on the *minLineSup* threshold. Any term that equals or exceeds the *minLineSup* is included as is and the others are replaced by a "*". Consecutive "*"s are merged into one to minimize the length of the pattern for very long data lines.

For line and value patterns, we create *closed frequent patterns* {$cfp_1, cfp_2, \ldots$} in multiple iterations from an item set $I = \{i_1, i_2, .., i_n\}$ and their corresponding support values $\{is_1, is_2, \ldots, is_n\}$ until there exists no $is_t \geq minSup$ where $t=\{1..n\}$. In each iteration we find the $s_{min} \geq minSup$ and create $cfp_u = \{(j_1, j_2, \ldots) \mid (j_x \in I) \vee (j_x =$"*"$) \vee (j_x =$"?"$)\}$ and $\forall (j_x = i_y) \Rightarrow is_y \geq s_{min}\}$, where $x=\{1..n\}$, $y=\{1..n\}$ and $u=\{1,2,..\}$. In the next iteration is_y is set to 0 where $is_y = s_{min}$ so as to not include that item again. We use "?" in value patterns and "*" in line patterns to indicate any character or term. For log data, the order of the items is preserved in the generated patterns. Fig. 6 shows the algorithm. For line patterns we merge multiple consecutive "*" to minimize the length of the pattern for very long data lines. New line patterns are saved using the *findSaveLinePattern* procedure (Fig. 6 line 17) which checks against existing patterns for a) exact match, or b) closest matching super patterns when saving rare or interesting patterns (that have support less than *minLineSup*) using a binary search algorithm. The objective is to avoid creating redundant line patterns. The *assignLineLabel* procedure in Fig. 6 line 18 categorizes the data lines by assigning them pattern IDs.

In Fig. 4 line 4, CC_2=2. Given a *minTermSup*=3, the term *c3i5* will be saved as a frequent term = <*c3i5, 2, 4*> with support (CC_2 +2). From its sorted actual values the frequent value patterns <*c3i5, 2,* {<*STC10799*,*3*>, <*STC?????*,*4*>}> are saved given *minValueSup*=3. If *minLineSup*=3, the line patterns are also saved since terms 1-4 in line 4 Fig. 4 have support values 4, 4, 3, 3 corresponding to *CC*= 2211…. In the first iteration s_{min}=3 and the pattern "i2.i2.i2 c3i5 c4i3c1 !c2i2 *" is saved as <P1, "i2.i2.i2 c3i5 c4i3c1 !c2i2 *", 3>. In the next iteration, support counts of value 3 are reset to 0 and the new s_{min}=4, and another super pattern is saved as <P2, "i2:i2:i2 c3i5 *", 4>.

Algorithm: saveLabelLinePatterns

```
1   While (saveFlag=1)
2       CC^min = -1
3       linePattern = ""
4       saveFlag = 0
5       For each term in the line
6         If (term_support>=minLineSup)
7           If (term_support < CC^min) or (CC^min <0)
8             CC^min = term_support
9           Endif
10          linePattern = linePattern + term
11        Else
12          If linePattern does not end with "*"
13            linePattern = linePattern + "*"
14          Endif
15        Endif
16      Endfor
17      Pid = findSaveLinePattern(linePattern, CC^min)
18      assignLineLabel(current_linenum, CC^min, Pid)
19      For each term in the line
20        If (term_support== CC^min)
21          Reset term_support=0
22        Else if (term_support >CC^min)
23          saveFlag=1
24        Endif
25      Endfor
26      CC^min = 0
27  Endwhile
```

Figure 6. Creates and saves closed frequent line patterns.

4.2.5 Saving Rare and Interesting Line Patterns

The extract pattern algorithm makes a final pass over the data in memory to check for *Interesting patterns* and unlabeled data lines (Fig. 5 line 32). In this work we assert a line is interesting if it contains 3 or more successive symbols (up to two symbols are used in data as "::"). The *getMatchingPattern* in Fig. 7 first looks for an exact match from existing patterns and then from closest super-patterns. If a match is not found, it looks for a partial match from which to construct a pattern. In the case of interesting

Algorithm: saveLabelRareInterestingLinePatterns

```
1   For each line
2       If (noLineLabel(line)) or isInterestingPattern(pattern)
3         pat = getMatchingPattern(pattern)
4         If (noLineLabel(line)) or isNew(pat) or
                  (!isNew(pat) and (llabel!=pat.pid))
5           Pid = findSaveLinePattern (pattern, 1)
6           assignLineLabel(current_linenum, 1, Pid)
7         Endif
8       Endif
9   Endfor
```

Figure 7. Creates and saves rare and interesting line patterns and labels the corresponding data lines.

patterns, the matching pattern is augmented with the interesting tokens from the line. In Fig. 3, line 1 is a rare pattern as it occurs only once and lines 2 and 6 are interesting as they contain 3 or more consecutive symbols. Extract pattern categorizes line 2 under "i2.i2.i2 c3i5 *". In the final pass, algorithm in Fig. 7 identifies it as an interesting pattern and augments the above to create a new pattern "i2.i2.i2 c3i5 --- *" for line 2.

4.3 Find Pattern Correlation Rules

Once the data is labeled with pattern IDs (PIDs), another pass is made over the data to a) count the support for each pair of PIDs that appear together in a specific order and b) find the relationship between pairs of terms in the lines with regards to position and value. When stating the rule, p1-p3 means a pair of consecutive line patterns p1 and p3, where p1 and p3 represent PIDs and p3 follows p1 as shown in Fig. 8 lines 3 and 4. The support is calculated the way as explained in Section 3.6. In Fig. 9, p1-p3 has support 18.18% (=2/11). Terms in a line are expressed as T*n* where *n* is the term position in the line starting with 0. The last term is expressed as TL, because of the different sizes of the lines. CAPRI compares the values of each pair of terms of a pair of PIDs, and generates rules of the form p1-p3⇒TL=T0 for lines 3 and 4 of Fig. 8. Here the antecedent of the rule TL belongs to p1 and the consequent, T0, belongs to p3. The confidence of the rule is calculated using Eq. 4. The support of (p1-p3 and TL=T0) is 1/11 and the support of p1-p3 is 2/11. Therefore, confidence=50%.

For large data files, rules that have low support and confidence values can be filtered out. Lines that violate rules having high confidence values indicate possible error in the data. In Fig. 8, line 7 violates the p1-p3 rule and has an error in line order.

```
0  p4  1 J E S 2 J O B L O G -- S Y S T E M Z P 0 1 -- N O D E U S I L 0 P 0 1
1  p5  19.00.25 STC10799 ---- SATURDAY, 18 DEC 2010 ----
2  p1  03.58.05 STC09048 DSNJ110E !SS11 LAST COPY 1 ACTIVE LOG
       DATA SET IS 95 PERCENT FULL
3  p1  12.46.21 STC10799 DSNP010I !PB11 END OF MESSAGES. 744
4  p3  744 CONNECTION-ID=SERVER
5  p3  744 CORRELATION-ID=PB1BTC008
6  p1  12.17.58 STC10799 DSNV512I !PB11 DSNVMON - AGENT 1: 771
7  p3  744 LUW-ID=ODCA4760.EC77.C737D7D90AA5=181319
8  p3  771 NAME ST A REQ ID AUTHID PLAN ASID TOKEN
9  p6  771 ---- -- - --- -- ------- ---- ---- -----
10 p3  771 SERVER RA * 10879 PB1DIA025 PB1ADM DISTSERV 0070
       67702
11 p3  771 LONG 9728K VLONG 436K 64BIT 28784K
```

Figure 8. Data of Fig. 1 categorized with line pattern ID.

4.4 Tool Output

Optionally (if the user enters an output file name) CAPRI writes data labeled with PIDs as shown in Fig. 8 to facilitate subsequent data processing. The discovered line, term and value patterns including the rules are written out to a default text file called "LTVPatterns.txt" as shown in Fig. 9.

4.5 Implementation

The CAPRI tool and test data sets with some results are available online [2]. It was developed using Java 1.6.0_26 on a 64-bit Windows platform. A set of Java class files can be extracted from a downloadable compressed rar file. It is executed as follows from a command prompt using the Java 1.6 runtime libraries.

java Capri <inputfile> [outputfile]

```
Line patterns: (Total 6 patterns from 12 lines)
p4 : i1 c1 * : 1
p2 : i2.i2.i2 c3i5 * : 4
p5 : i2.i2.i2 c3i5 ---- * : 1
p1 : i2.i2.i2 c3i5 c4i3c1 !c2i2 * : 3
p3 : i3 * : 7
p6 : i3 ---- * : 1

Term patterns: (Total 5 patterns ordered by position) : Position in line : Value
patterns (Total 9 patterns)
i2.i2.i2      : pos 1    : 4    [1?.??.?? : 3, ??.??.?? : 4]
i3            : pos 1    : 7    [771 : 4, 744 : 3, 7?? : 7]
c3i5          : pos 2    : 4    [STC10799 : 3, STC????? : 4]
c4i3c1        : pos 3    : 3    [DSN??1?? : 3]
!c2i2         : pos 4    : 3    [!??11 : 3]

Rule list (having support>0.1 and confidence>0.5)

p3-p3 (cnt:3, sup:0.2727272727272727) => {T0=T0=0.6666666666666666}
p1-p3 (cnt:2, sup:0.18181818181818182) => {TL=T0=0.5}
Total 2 rules generated.
```

Figure 9. Extracted line, term and value patterns and rules for the data of Fig. 1 for minLineSup=25% (=3), minTermSup=25% and minValueSup=25%.

5. EXPERIMENTAL EVALUATION

We evaluate the effectiveness of CAPRI based on the following criteria using a machine having an Intel core i7 processor with a 2.67 GHz CPU and 4GB memory, running a 64-bit Windows 7 operating system:

I. Ability to categorize all lines including rare and interesting line patterns, and find frequent term and value patterns, and rules that state relationships between pairs of line patterns.
II. The effect of minimum support thresholds on the result of moderate size data files.
III. Comparative performance analysis with other similar tools for both small and large data sets of various types.

Data Type B: Distributed DB2 log file

```
1    2010-12-01-10.34.15.693000-300 I1F955          LEVEL: Event
2      PID    : 3740              TID  : 4148       PROC : DB2STOP.EXE
3      INSTANCE: DB2              NODE : 000
4      Information in this record is only valid at the time when this file was
5      created (see this record's time stamp)
6    2010-12-01-10.34.15.693000-300 I959F1671       LEVEL: Event
...
```

Data Type C: Squid access log file

```
1 1286536308.779    180 192.168.0.224 TCP_MISS/200 411 GET
    http://liveupdate.symanteclive update.com/minitri.flg -
    DIRECT/125.23.216.203 text/plain
2 1286536309.586    921 192.168.0.68 TCP_MISS/200 507 POST http://rcv-
    srv37.inplay.tubemogul.co...eiver/services - DIRECT/174.129.41.128
    application/xml
...
```

Figure 10. Data types

5.1 Test Data Types

We use three different data layouts A, B, and C; A is a z/OS mainframe DB2 JES master log data with multi-line messages (example shown in Fig. 1); B is a distributed DB2 log with paragraph style messages, and C is a Squid web access log file with single line messages. Samples of the B and C data sets are presented in Fig. 10. Data sets of Fig. 1 (type A), B, and C with some experimental results are also available online with the tool.

5.2 Pattern and Rule Discovery

Our small running example dataset A of Fig. 1 and the outputs from CAPRI in Fig. 8 and 9 demonstrate that CAPRI not only finds the closed frequent patterns, it also accurately categorizes the lines with the PIDs of the closest matching patterns. It finds

rare and interesting line patterns, all term and value patterns, and rules given a minimum support threshold.

5.3 Effect of Support Thresholds

In general, greater support values result in fewer and more general patterns. We define the test cases shown in Table 1 to examine the effects of different parameters on CAPRI's performance for each data type. We apply CAPRI on moderate size log files of type A, B and C. The results are shown in Table 2. For rule count, each consequent of the same antecedent is counted separately.

Table 1: Test-case parameters for CAPRI

Test case	minLineSup	minTermSup	minValueSup	minRuleSup	minRuleConf
1	30	20	10	5	80
2	20	20	20	10	90
3	40	25	20	7	95

Note: Support thresholds are in percentile

Table 2: Study of the effect of parameters on results

Data type	File Size in KB	Lines of data	Test case	Number of patterns			Number of rules
				Line	Term	Value	
A	106	1679	1	7	20	139	4
			2	13	20	46	1
			3	7	19	45	3
B	176	2826	1	88	14	29	69
			2	101	14	17	66
			3	76	11	13	66
C	16	113	1	9	15	55	16
			2	11	15	31	10
			3	7	13	29	10

5.3.1 Observations and Inferences

Data type A: Comparing rows A.1 and A.2 in Table 2, we see that a lower *minLineSup* results in more line patterns and consequently a lower support value for each rule. So, with a lower *minLineSup* and slightly higher *minRuleSup* threshold, only 1 rule is extracted leaving out an important rule. Therefore, higher *minLineSup* allow CAPRI to efficiently mine closed frequent patterns and rules from larger data sets. A higher *minValueSup* helps in reducing the many different value patterns extracted for date and line numbers and still maintains other important value patterns. Overall, A.3 gives the most concise and effective set of patterns and rules for the data set.

Data type B: This block style data layout includes many lines with 3 or more consecutive symbols in binary data, and rare line patterns due to the unstructured message in each block. For interesting patterns, CAPRI looks for partial match in super patterns and then adds terms up to the interesting term. Therefore, many similar interesting patterns are created and the total number of patterns is high. Many rules are also created for the same antecedent due to existence of blocks of binary data.

Data type C: This data file contains very similar short single line messages that differ mostly in the parameters. The rules are, therefore, helpful in identifying constant terms. Since CAPRI explores frequent values only for the frequent terms, row C.2 has more term patterns (for lower *minTermSup*) and consequently more value patterns than C.3 for the same *minValueSup* threshold.

5.4 Comparative Performance Analysis

We select two popular open source line pattern mining tools called the SLCT and LogHound [10][6] which have previously been used in research studies. Other tools such as IPLoM [7] and Teiresius [8] assume single line messages and require preprocessing of the data to extract only the message (description part without the date, time and other preceding parts) from each line in to another data file, which is then processed by the tools. Unlike CAPRI all the other tools mine patterns based on actual data values. Therefore, we cannot do an exact comparison. We use SLCT and LogHound for the same data types A, B, and C and provide a comparison in terms of functionality, coverage, and abstractness of the patterns. Some examples of the discovered patterns for the different data types for SLCT and LogHound are shown in Table 3. More detailed test results including sample data sets of each type are available online with CAPRI.

Table 3. Examples of extracted clusters or line patterns for data types A, B and C using SLCT and LogHound

SLCT	A	771 744 * STC10799
	B	* : 2010-12-01-10.34.15.693000-300 * LEVEL: Event INSTANCE: DB2 NODE : 000 PID : 3740 TID : 4148 PROC : DB2STOP.EXE
	C	* * 192.168.0.224 TCP_MISS/200 * GET * - IRECT/125.23.216.203 * * 192.168.0.68 TCP_MISS/200 507 POST http://rcv-srv37.inplay.tubemogul.co...eiver/services - DIRECT/174.129.41.128 application/xml
LogHound	A	* * * * * * * * : * * CORRELATION-ID=db2jcc_appli * * (SUSPENDED) (FOR) (PRT1) (AT) (RBA) * LRSN * (PRIOR)
	B	* * * * * * : (DATA) * : * 34 (DATA) (#2) (:) * PD_SQLER_TYPE_FMP_HANDLE, * (bytes)
	C	* * * * * * * - * * * * * GET * (-) * * * TCP_MISS/200 * GET * (-)

Table 4: Study of the comparative performances of CAPRI, SLCT and LogHound for different types and sizes of data

Data type	File Size (KB)	Lines of data	Test case	No. of line patterns			Support threshold
				CAPRI	SLCT	LogHound	
A	106	1679	1	7	15	44	30
			2	13	34	116	20
			3	7	14	40	40
	59 MB	938262	4	28	8356	52636	30
B	176	2826	1	88	24	275	30
			2	101	80	444	20
			3	76	22	273	40
C	16	113	1	9	1	9	30
			2	11	2	10	20
			3	7	1	8	40

Table 4 shows the comparison between the three tools in terms of the number of line patterns retrieved by each for the same data set as in Table 2. An additional highlighted row is shown (row A.4 in Table 4) as test case 4 for data type A that uses a larger data file. Since the other tools take only the minimum term support threshold, it is listed on the rightmost column. The same value is used as the *minLineSup* for CAPRI.

5.4.1 Observations and Inferences

CAPRI discovers a concise number of type-casted line patterns in contrast to the other tools, which use the actual data values. CAPRI also finds the term and value patterns. Abstract patterns that are composed of only "*" as shown in Table 4 discovered by SLCT and LogHound, do not present any useful information. Some patterns like 771 and 744 as extracted by SLCT for the data set in Fig. 1 are confusing. CAPRI retrieves more useful patterns with none composed of only "*". Fewer patterns are discovered

by SLCT for single line messages in data types B and C as shown in Table 4 because it finds only the frequent clusters. LogHound finds more patterns, and therefore, has better coverage than SLCT for all the different data types but it has many abstract and redundant super patterns as shown in Table 3. For multi-line messages in large files CAPRI does a much better job than the other tools as demonstrated in test case 4 in Table 3. It also mines rules, identifies rare and interesting patterns and optionally generates an output file with lines labeled with pattern ids. CAPRI is based on java and is, therefore, platform independent. Both the other tools use C-code and are designed to run on UNIX.

5.5 Usability of the Tool for Big Data

CAPRI discovers line patterns with 100% coverage for each data line and labels each data line with a pattern ID. As ongoing work, we are implementing CAPRI using Amazon's Elastic Map Reduce (EMR) [1] framework to maximize its efficiency in processing big data files. EMR uses the Hadoop framework with cloud resources for storage and computation. For the decision support applications, we define rules to process each type of line pattern for further log analysis. For example, p5 in Fig. 8 denotes start of a new day and p3 indicates continuation of a message. The rules such as p1-p3⇒TL=T0 enable automated reformatting to generate single line messages from the multi-line messages using the matching line numbers in line patterns p1 and p3 and correction of line orders. The term and value patterns such as "i2.i2.i2" and "DSNJ????" are used to identify time and error codes, constant and variable terms to facilitate message correlation analysis.

6. CONCLUSION

We present the CAPRI tool which demonstrates a type-casting technique, a bitmap multiplication algorithm to compute the support for discovering closed frequent line patterns in a single pass over big data, and allows incremental mining for large data files. CAPRI discovers both frequent and rare line patterns in a type-casted format from both single and multi-line semi-structured log data. We developed CAPRI based on observations of the human cognitive approach towards line patterns and the typical data analytics requirements with respect to log data analysis. Existing approaches return line patterns based on the actual data values which result in very long lists of message patterns or return only the frequent patterns. The type-casting technique provides a much shorter list of line patterns for the same support threshold. CAPRI discovers frequent term and value patterns as well, which gives important semantic information about the data. CAPRI achieves 100% recall by labeling and categorizing each line of data with the closest matching frequent line pattern ID for effective post-processing. It generates rules, which show the contextual relationship between pairs of line patterns with corresponding support and confidence values. Users can choose to filter out rules having low support and confidence, and thereby, generate a more effective rule set. The rule set can be used to identify errors in line order, to reformat multi-line messages or explore further correlations in the data.

For future work, we like to extend the usability of CAPRI to create a semantic vocabulary of type-casted frequent term patterns to enable human-like recognition capabilities. We like to define rules to specify interesting patterns and to constrain the extraction of value patterns for known terms such as date and time. We also plan to carry out experiments on Amazon EMR with big datasets.

7. ACKNOWLEDGMENTS

We acknowledge the contributions of MITACS (Mathematics of Information Technology and Complex Systems) Elevate and CA Technologies for supporting the research.

8. REFERENCES

[1] Amazon Elastic Map Reduce. Available online at: http://aws.amazon.com/elasticmapreduce/.

[2] CAPRI: type-CAsted Pattern and Rule mIner. 2013. At: http://research.cs.queensu.ca/home/farhana/capri.html.

[3] Fu, Q., Lou, J.G., Wang, Y., and Li, J., 2009. Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis. *In proc. of the IEEE ICDM*, pp. 149 – 158, Miami, FL.

[4] Hellerstein, J., Ma, S., and Perng, C., 2002. Discovering Actionable Patterns in Event Data. *IBM System Journal*, vol. 41(3).

[5] Jiang, Z., Hassan, A., Hamann, G, and Flora, P., 2008. An Automated Approach for Abstracting Execution Logs to Execution Events. *Journal of Software Maintenance and Evolution: Research and Practice,* vol. 20, pp. 249–267.

[6] Makanju, A., Brooks, S., Zincir-Heywood, A., Milios, E., 2008. LogView: Visualizing Event Log Clusters. *In proc. of annual conference on Privacy, Security and Trust,* Fredericton, NB, pp. 99 – 108, IEEE.

[7] Makanju, A., Zincir-Heywood, A. N., Milios, E., 2009. Clustering Event Logs by Iterative Partitioning. *In proc. of the ACM SIG KDD*, Paris, France, pp. 1255-1263, ACM.

[8] Stearley, J., 2004. Towards Informatic Analysis of Syslogs. In *proc. of the IEEE Intl. Conf. on Cluster Computing (CLUSER)*, San Diego, California, USA, pp. 309-318.

[9] Subbulakshmi, T., Mathew, G., Shalinie, S., 2010. Real Time Classification and Clustering of IDS Alerts using Machine Learning Algorithm. In *proc. of Intl. Journal of Artificial Intelligence and Applications (IJAIA)*, vol. 1(1).

[10] Vaarandi, R., 2008. Mining Event Logs with SLCT and LogHound. *IEEE/IFIP Network Operations and Management Symposium: Pervasive Management for Ubiquitous Networks and Services, NOMS*, Salvador, Bahia, Brazil, pp. 1071-1074, IEEE.

[11] Vaarandi, R., Podins, K., 2010. Network IDS Alert Classification with Frequent Itemset Mining and Data Clustering. *In intl. conf. on Network and Service Mgmt. (CNSM)*, Niagara Falls, Canada, pp. 451-456, IEEE.

[12] Xu, W., Huang, L., Fox, A., Patterson, D., and Jordan, M., 2009. Detecting Large-Scale System Problems by Mining Console Logs. *In proc. of SOSP*, Montana, US, ACM.

[13] Zhang, S., and Wu, X., 2011. Fundamentals of Association Rules in Data Mining and Knowledge Discovery, *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 1(2), John Wiley & Sons.

[14] Zulkernine, F., Martin, P., Soltani, S., Powley, W., Mankovski, S., and Addleman, M., 2012. Towards a Training Oriented Adaptive Decision Support System, in *proc. of IEEE ICDE workshop on Data-Driven Decision Guidance and Support System (DGSS)*, Washington DC. US.