Reprinted from *Proceedings of the 2003 IEEE Workshop on IP Operations and Management*.
(ISBN: 0-7803-8199-8)

A Data Clustering Algorithm for Mining Patterns From Event Logs

Risto Vaarandi
Department of Computer Engineering
Tallinn Technical University
Tallinn, Estonia
risto.vaarandi@eyp.ee

***Abstract*— Today, event logs contain vast amounts of data that can easily overwhelm a human. Therefore, mining patterns from event logs is an important system management task. This paper presents a novel clustering algorithm for log file data sets which helps one to detect frequent patterns from log files, to build log file profiles, and to identify anomalous log file lines.**

Keywords—system monitoring, data mining, data clustering

I. INTRODUCTION

Event logging and log files are playing an increasingly important role in system and network management. Over the past two decades, the BSD *syslog* protocol [1] has become a widely accepted standard that is supported on many operating systems and is implemented in a wide range of system devices. Well-written system applications either use the *syslog* protocol or produce log files in custom format, while many devices like routers, switches, laser printers, etc. are able to log their events to remote host using the *syslog* protocol. Normally, events are logged as single-line textual messages. Since log files are an excellent source for determining the health status of the system, many sites have built a centralized logging and log file monitoring infrastructure. Because of the importance of log files as the source of system health information, a number of tools have been developed for monitoring log files, e.g., Swatch [2], Logsurfer [3], and SEC [4].

Log file monitoring techniques can be categorized into fault detection and anomaly detection. In the case of *fault detection*, the domain expert creates a database of fault message patterns. If a line is appended to a log file that matches a pattern, the log file monitor takes a certain action. This commonly used approach has one serious flaw - only those faults that are already known to the domain expert can be detected. If a previously unknown fault condition occurs, the log file monitor simply ignores the corresponding message in the log file, since there is no match for it in the pattern database. Also, it is often difficult to find a person with sufficient knowledge about the system. In the case of *anomaly detection*, a system profile is created which reflects normal system activity. If messages are logged that do not fit the profile, an alarm is raised. With this approach, previously unknown fault conditions are detected, but on the other hand, creating the system profile by hand is time-consuming and error-prone.

In order to solve the knowledge acquisition problems, various methods have been employed, with data mining methods being one of the most popular choices [5, 6, 7, 8, 9]. In most research papers, the focus has been on mining frequent patterns from event logs. This helps one to find patterns that characterize the normal behavior of the system, and facilitates the creation of the system profile. However, as pointed out in [8], the mining of infrequent patterns is equally important, since this might reveal anomalous events that represent unexpected behavior of the system, e.g., previously unknown fault conditions. Recent research papers have mainly proposed the mining of temporal patterns from event logs with various association rule algorithms [5, 6, 7, 8, 9]. These algorithms assume that the event log has been normalized, i.e., all events in the event log have a common format. Typically, each event is assumed to have at least the following attributes: *timestamp* of event occurrence, the *event type*, and the name of the *node* which issued the event (though the node name is often encoded in the event type). Association rule algorithms have been often used for detecting temporal associations between event types [5, 6, 7, 8, 9], e.g., if events of type A and B occur within 5 seconds, they will be followed by an event of type C within 60 seconds (each detected temporal association has a certain frequency and confidence).

Although association rule algorithms are powerful, they often can't be directly applied to log files, because log file lines do not have a common format. Furthermore, log file lines seldom have all the attributes that are needed by the association rule algorithms. For example, the widely used *syslog* protocol does not impose strict requirements on the log message format [1]. A typical *syslog* message has just the timestamp, hostname, and program name attributes that are followed by a free-form message string, but only the message string part is mandatory [1]. A detailed discussion of the shortcomings of the *syslog* protocol can be found in [1, 10].

One important attribute that log file lines often lack is the event type. Fortunately, it is possible to derive event types from log file lines, since very often the events of the same type correspond to a certain line pattern. For example, the lines

Router myrouter1 interface 192.168.13.1 down

Router myrouter2 interface 10.10.10.12 down

Router myrouter5 interface 192.168.22.5 down

This work is supported by the Union Bank of Estonia.

represent the event type "interface down", and correspond to the line pattern `Router * interface * down`. Line patterns could be identified by manually reviewing log files, but this is feasible for small log files only.

One appealing choice for solving this problem is the employment of data clustering algorithms. Clustering algorithms [11, 12] aim at dividing the set of objects into groups (clusters), where objects in each cluster are similar to each other (and as dissimilar as possible to objects from other clusters). Objects that do not fit well to any of the clusters detected by the algorithm are considered to form a special cluster of *outliers*. When log file lines are viewed as objects, clustering algorithms are a natural choice, because line patterns form natural clusters - lines that match a certain pattern are all similar to each other, and generally dissimilar to lines that match other patterns. After the clusters (event types) have been identified, association rule algorithms can be applied for detecting temporal associations between event types.

However, note that log file data clustering is not merely a preprocessing step. A clustering algorithm could identify many line patterns that reflect normal system activity and that can be immediately included in the system profile, since the user does not wish to analyze them further with the association rule algorithms. Furthermore, the cluster of outliers that is formed by the clustering algorithm contains infrequent lines that could represent previously unknown fault conditions, or other unexpected behavior of the system that deserves closer investigation.

Although data clustering algorithms provide the user a valuable insight into event logs, they have received little attention in the context of system and network management. In this paper, we discuss existing data clustering algorithms, and propose a new clustering algorithm for mining line patterns from log files. We also present an experimental clustering tool called SLCT (Simple Logfile Clustering Tool). The rest of this paper is organized as follows: section 2 discusses related work on data clustering, section 3 presents a new clustering algorithm for log file data sets, section 4 describes SLCT, and section 5 concludes the paper.

II. Related Work

Clustering methods have been researched extensively over the past decades, and many algorithms have been developed [11, 12]. The clustering problem is often defined as follows: given a set of points with n attributes in the data space R^n, find a partition of points into clusters so that points within each cluster are close (similar) to each other. In order to determine, how close (similar) two points x and y are to each other, a distance function $d(x, y)$ is employed. Many algorithms use a certain variant of L_p norm (p = 1, 2, ...) for the distance function:

$$d_p(x, y) = \sqrt[p]{\sum_{i=1}^{n} |x_i - y_i|^p}$$

Today, there are two major challenges for traditional clustering methods that were originally designed for clustering numerical data in low-dimensional spaces (where usually n is well below 10).

Firstly, quite many data sets consist of points with *categorical* attributes, where the domain of an attribute is a finite and unordered set of values [13, 14]. As an example, consider a categorical data set with attributes *car-manufacturer*, *model*, *type*, and *color*, and data points ('Honda', 'Civic', 'hatchback', 'green') and ('Ford', 'Focus', 'sedan', 'red'). Also, it is quite common for categorical data that different points can have different number of attributes. Therefore, it is not obvious how to measure the distance between data points. Though several popular distance functions for categorical data exist (such as the Jaccard coefficient [12, 13]), the choice of the right function is often not an easy task. Note that log file lines can be viewed as points from a categorical data set, since each line can be divided into words, with the n-th word serving as a value for the n-th attribute. For example, the log file line *Connection from 192.168.1.1* could be represented by the data point ('Connection', 'from', '192.168.1.1'). We will use this representation of log file data in the remainder of this paper.

Secondly, quite many data sets today are high-dimensional, where data points can easily have tens of attributes. Unfortunately, traditional clustering methods have been found not to work well when they are applied to high-dimensional data. As the number of dimensions n increases, it is often the case that for every pair of points there exist dimensions where these points are far apart from each other, which makes the detection of any clusters almost impossible (according to some sources, this problem starts to be severe when $n \geq 15$) [12, 15, 16]. Furthermore, traditional clustering methods are often unable to detect natural clusters that exist in subspaces of the original high-dimensional space [15, 16]. For instance, data points (1333, 1, 1, 99, 25, 2033, 1044), (12, 1, 1, 724, 667, 36, 2307), and (501, 1, 1, 1822, 1749, 808, 9838) are not seen as a cluster by many traditional methods, since in the original data space they are not very close to each other. On the other hand, they form a very dense cluster in the second and third dimension of the space.

The dimensionality problems described above are also relevant to the clustering of log file data, since log file data is typically high-dimensional (i.e., there are usually more than just 3-4 words on every line), and most of the line patterns correspond to clusters in subspaces. For example, the lines

log: connection from 192.168.1.1

log: RSA key generation complete

log: Password authentication for john accepted.

form a natural cluster in the first dimension of the data space, and correspond to the line pattern `log: *`.

During past few years, several algorithms have been developed for clustering high-dimensional data, like CLIQUE, MAFIA, CACTUS, and PROCLUS. The CLIQUE [15] and MAFIA [17] algorithms closely remind the Apriori algorithm for mining frequent itemsets [18]: they start with identifying all clusters in 1-dimensional subspaces, and after they have identified clusters $C_1,...,C_m$ in *(k-1)*-dimensional subspaces,

they form cluster candidates for *k*-dimensional subspaces from $C_1,...,C_m$, and then check which of those candidates are actual clusters. Those algorithms are effective in discovering clusters in subspaces, because they do not attempt to measure distance between individual points, which is often meaningless in a high-dimensional data space. Instead, their approach is *density based*, where a clustering algorithm tries to identify *dense regions* in the data space, and forms clusters from those regions. Unfortunately, the CLIQUE and MAFIA algorithms suffer from the fact that Apriori-like candidate generation and testing involves exponential complexity and high runtime overhead [19, 20] – in order to produce a frequent *m*-itemset, the algorithm must first produce $2^m - 2$ subsets of that *m*-itemset. The CACTUS algorithm [14] first makes a pass over the data and builds a data summary, then generates cluster candidates during the second pass using the data summary, and finally determines the set of actual clusters. Although CACTUS makes only two passes over the data and is therefore fast, it is susceptible to the phenomenon of chaining (long strings of points are assigned to the same cluster) [11], which is undesirable if one wants to discover patterns from log files. The PROCLUS algorithm [16] uses the K-medoid method for detecting K clusters in subspaces of the original space. However, in the case of log file data the number of clusters can rarely be predicted accurately, and therefore it is not obvious what is the right value for K.

Though several clustering algorithms exist for high-dimensional data spaces, they are not very suitable for clustering log file lines, largely because they don't take into account the nature of log file data. In the next section, we will first discuss the properties of log file data, and then we will present a fast clustering algorithm that relies on these properties.

III. CLUSTERING LOG FILE DATA

A. The Nature of Log File Data

The nature of the data to be clustered plays a key role when choosing the right algorithm for clustering. Most of the clustering algorithms have been designed for generic data sets such as market basket data, where no specific assumptions about the nature of data are made. However, when we inspect the content of typical log files at the word level, there are two important properties that distinguish log file data from a generic data set. During our experiments that revealed these properties, we used six logfile data sets from various domains: HP OpenView event log file, mail server log file (the server was running *sendmail*, *ipopd*, and *imapd* daemons), Squid cache server log file, Internet banking server log file, file and print server log file, and Win2000 domain controller log file. Although it is impossible to verify that the properties we have discovered characterize every logfile ever created on earth, we still believe that they are common to a wide range of logfile data sets.

Firstly, majority of the words occur only a few times in the data set. Table 1 presents the results of an experiment for estimating the occurrence times of words in log file data. The results show that a majority of words were very infrequent, and a significant fraction of words appeared just once in the data set (one might argue that most of the words occurring once are timestamps, but when timestamps were removed from data sets, we observed no significant differences in the experiment results). Also, only a small fraction of words were relatively frequent, i.e., they occurred at least once per every 10,000 or 1,000 lines. Similar phenomena have been observed for World Wide Web data, where during an experiment nearly 50% of the words were found to occur once only [21].

TABLE I. OCCURRENCE TIMES OF WORDS IN LOG FILE DATA

Data set	Mail server log file (Linux)	Cache server log file (Linux)	HP OpenView event log file (Solaris)	Internet banking server log file (Solaris)	File and print server log file (Win2000)	Domain controller log file (Win2000)
Data set size	1025.3 MB, 7,657,148 lines	1088.9 MB, 8,189,780 lines	696.9 MB, 1,835,679 lines	2872.4 MB, 14,733,696 lines	2451.6 MB, 7,935,958 lines	1043.9 MB, 4,891,883 lines
Total number of different words	1,700,840	1,887,780	1,739,185	2,008,418	11,893,846	4,016,009
The number of words occurring once	848,033	1,023,029	1,330,257	434,337	11,635,297	3,948,414
The number of words occurring 5 times or less	1,443,159	1,456,489	1,509,973	1,082,687	11,709,484	3,951,492
The number of words occurring 10 times or less	1,472,296	1,568,165	1,582,970	1,419,138	11,716,395	3,953,698
The number of words occurring 20 times or less	1,493,160	1,695,336	1,668,060	1,669,784	11,722,699	3,956,850
The number of words occurring at least once per 10,000 lines	3,091	3,122	6,933	4,263	2,452	2,617
The number of words occurring at least once per 1,000 lines	627	476	1,242	304	817	293

Secondly, we discovered that there were many strong correlations between words that occurred frequently. As we found, this effect is caused by the fact that a message is generally formatted according to a certain format string before it is logged, e.g.,

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

When events of the same type are logged many times, constant parts of the format string will become frequent words which occur together many times in the data set. In the next subsection we will present a clustering algorithm that relies on the special properties of log file data.

B. The clustering algorithm

Our aim was to design an algorithm which would be fast and make only a few passes over the data, and which would detect clusters that are present in subspaces of the original data space. The algorithm relies on the special properties of log file data discussed in the previous subsection, and uses the density based approach for clustering.

The data space is assumed to contain data points with categorical attributes, where each point represents a line from a log file data set. The attributes of each data point are the words from the corresponding log file line. The data space has n dimensions, where n is the maximum number of words per line in the data set. A *region* S is a subset of the data space, where certain attributes $i_1,...,i_k$ ($1 \le k \le n$) of all points that belong to S have identical values $v_1,...,v_k$: $\forall x \in S$, $x_{i1} = v_1$, ..., $x_{ik} = v_k$. We call the set $\{(i_1,v_1),...,(i_k,v_k)\}$ the set of *fixed attributes* of region S. If $k=1$ (i.e., there is just one fixed attribute), the region is called *1-region*. A *dense region* is a region that contains at least N points, where N is the *support threshold value* given by the user.

The algorithm consists of three steps like the CACTUS algorithm [14] – it first makes a pass over the data and builds a data summary, and then makes another pass to build cluster candidates, using the summary information collected before. As a final step, clusters are selected from the set of candidates.

During the first step of the algorithm (data summarization), the algorithm identifies all dense 1-regions. Note that this task is equivalent to the mining of *frequent words* from the data set (the word position in the line is taken into account during the mining). A word is considered frequent if it occurs at least N times in the data set, where N is the user-specified support threshold value.

After dense 1-regions (frequent words) have been identified, the algorithm builds all cluster candidates during one pass. The cluster candidates are kept in the candidate table, which is initially empty. The data set is processed line by line, and when a line is found to belong to one or more dense 1-regions (i.e., one or more frequent words have been discovered on the line), a cluster candidate is formed. If the cluster candidate is not present in the candidate table, it will be inserted into the table with the support value 1, otherwise its support value will be incremented. In both cases, the line is assigned to the cluster candidate. The cluster candidate is formed in the following way: if the line belongs to m dense 1-regions that have fixed attributes $(i_1,v_1),...,(i_m,v_m)$, then the cluster candidate is a region with the set of fixed attributes $\{(i_1,v_1),...,(i_m,v_m)\}$. For example, if the line is *Connection from 192.168.1.1*, and there exist a dense 1-region with the fixed attribute (1, 'Connection') and another dense 1-region with the fixed attribute (2, 'from'), then a region with the set of fixed attributes {(1, 'Connection'), (2, 'from')} becomes the cluster candidate.

During the final step of the algorithm, the candidate table is inspected, and all regions with support values equal or greater than the support threshold value (i.e., regions that are guaranteed to be dense) are reported by the algorithm as clusters. Because of the definition of a region, each cluster corresponds to a certain line pattern, e.g., the cluster with the set of fixed attributes {(1, 'Password'), (2, 'authentication'), (3, 'for'), (5, 'accepted')} corresponds to the line pattern `Password authentication for * accepted`. Thus, the algorithm can report clusters in a concise way by just printing out line patterns, without reporting individual lines that belong to each cluster. The CLIQUE algorithm reports clusters in a similar manner [15].

The first step of the algorithm reminds very closely the popular Apriori algorithm for mining frequent itemsets [18], since frequent words can be viewed as frequent 1-itemsets. Then, however, our algorithm takes a rather different approach, generating all cluster candidates at once. There are several reasons for that. Firstly, Apriori algorithm is expensive in terms of runtime [19, 20], since the candidate generation and testing involves exponential complexity. Secondly, since one of the properties of log file data is that there are many strong correlations between frequent words, it makes little sense to test a potentially huge number of frequent word combinations that are generated by Apriori, while only a relatively small number of combinations are present in the data set. It is much more reasonable to identify the existing combinations during a single pass over the data, and verify after the pass which of them correspond to clusters.

It should be noted that since Apriori uses level-wise candidate generation, it is able to detect patterns that our algorithm does not report. E.g., if words *A*, *B*, *C*, and *D* are frequent, and the only combinations of them in the data set are *A B C* and *A B D*, then our algorithm will not inspect the pattern *A B*, although it could have the required support. On the other hand, by restricting the search our algorithm avoids reporting all subsets of a frequent itemset that can easily overwhelm the user, but rather aims at detecting maximal frequent itemsets only (several pattern-mining algorithms like Max-Miner [19] use the similar approach).

In order to compare the runtimes of our algorithm and Apriori-based algorithm, we implemented both algorithms in Perl and tested them against three small log file data sets. Table 2 presents the results of our tests that were conducted on 1,5GHz Pentium4 workstation with 256MB of memory and Redhat 8.0 Linux as operating system (the sizes of log files A, B, and C were 180KB, 1814KB, and 4005KB, respectively).

The results obtained show that our clustering algorithm is superior to the Apriori-based clustering scheme in terms of runtime cost. The results also indicate that Apriori-based clustering schemes are appropriate only for small log file data sets and high support thresholds.

TABLE II. THE RUNTIME COMPARISON OF OUR ALGORITHM AND APRIORI-BASED ALGORITHM

	Support threshold 50%	Support threshold 25%	Support threshold 10%	Support threshold 5%	Support threshold 1%
Our algorithm for A	1 second	1 second	1 second	2 seconds	2 seconds
Apriori for A	2 seconds	16 seconds	96 seconds	145 seconds	5650 seconds
Our algorithm for B	5 seconds	5 seconds	5 seconds	6 seconds	6 seconds
Apriori for B	9 seconds	28 seconds	115 seconds	206 seconds	2770 seconds
Our algorithm for C	10 seconds	10 seconds	12 seconds	12 seconds	13 seconds
Apriori for C	182 seconds	182 seconds	18950 seconds	29062 seconds	427791 seconds

Although our algorithm makes just two passes over the data and is therefore fast, it could consume a lot of memory when applied to a larger data set. In the next subsection we will discuss the memory cost issues in more detail.

C. The Memory Cost of The Algorithm

In terms of memory cost, the most expensive part of the algorithm is the first step when the data summary is built. During the data summarization, the algorithm seeks for frequent words in the data set, by splitting each line into words. For each word, the algorithm checks whether the word is present in the word table (or vocabulary), and if it isn't, it will be inserted into the vocabulary with its occurrence counter set to 1. If the word is present in the vocabulary, its occurrence counter will be incremented.

If the vocabulary is built for a large data set, it is likely to consume a lot of memory. When vocabularies were built for data sets from Table 1, we discovered that they consumed hundreds of megabytes of memory, with the largest vocabulary occupying 653 MB (the tests were made on Sun Fire V480 server with 4 GB of memory, and each vocabulary was implemented as a move-to-front hash table which is an efficient data structure for accumulating words [21]). As the size of the data set grows to tens or hundreds of gigabytes, the situation is very likely to deteriorate further, and the vocabulary could not fit into the main memory anymore.

On the other hand, one of the properties of log file data is that a majority of the words are very infrequent. Therefore, storing those very infrequent words to memory is a waste of space. Unfortunately, it is impossible to predict during the vocabulary construction which words will finally be infrequent.

In order to cope with this problem, we use the following technique - we first estimate which words *need not* to be stored in memory, and then create the vocabulary without irrelevant words in it. Before the data pass is made for building the vocabulary, the algorithm makes an extra pass over the data and builds a word summary vector. The word summary vector is made up of m counters (numbered from 0 to m-1) with each counter initialized to zero. During the pass over the data, a fast string hashing function is applied to each word. The function returns integer values from 0 to m-1, and each time the value i is calculated for a word, the i-th counter in the vector will be incremented. Since efficient string hashing functions are uniform [22], i.e., the probability of an arbitrary string hashing to a given value i is $1/m$, then each counter in the vector will correspond roughly to W / m words, where W is the number of different words in the data set. If words $w_1,...,w_k$ are all words that hash to the value i, and the words $w_1,...,w_k$ occur $t_1,...,t_k$ times, respectively, then the value of the i-th counter in the vector equals to the sum $t_1+...+t_k$.

After the summary vector has been constructed, the algorithm starts building the vocabulary, but only those words will be inserted into the vocabulary for which their counter values are equal or greater than the support threshold value given by the user. Words that do not fulfill this criterion can't be frequent, because their occurrence times are guaranteed to be below the support threshold.

Given that a majority of the words are very infrequent, this simple technique is quite powerful. If the vector is large enough, a majority of the counters in the vector will have very infrequent words associated with them, and therefore many counter values will never cross the support threshold.

In order to measure the effectiveness of the word summary vector technique, we made a number of experiments with data sets from Table 1. We used the support thresholds of 1%, 0.1%, and 0.01% together with the vectors of 5,000, 20,000, and 100,000 counters, respectively (each counter consumed 4 bytes of memory). The experiments suggest that the employment of the word summary vector dramatically reduces vocabulary sizes, and large amounts of memory will be saved. During the experiments, vocabulary sizes decreased 9.93-99.36 times, and 32.74 times as an average. On the other hand, the memory requirements for storing the vectors were relatively small - the largest vector we used during the experiments occupied less than 400 KB of memory.

If the user has specified a very low support threshold, there could be a large number of cluster candidates with very small support values, and the candidate table could consume a significant amount of memory. In order to avoid this, the summary vector technique can also be applied to cluster candidates – before the candidate table is built, the algorithm makes an extra pass over the data and builds a summary vector for candidates, which is later used to reduce the number of candidates inserted into the candidate table.

IV. SIMPLE LOGFILE CLUSTERING TOOL

In order to implement the log file clustering algorithm described in the previous section, an experimental tool called SLCT (Simple Logfile Clustering Tool) has been developed. SLCT has been written in C and has been primarily used on Redhat 8.0 Linux and Solaris8, but it should compile and work on most modern UNIX platforms.

SLCT uses move-to-front hash tables for implementing the vocabulary and the candidate table. Experiments with large vocabularies have demonstrated that move-to-front hash table is an efficient data structure with very low data access times, even when the hash table is full and many words are connected to each hash table slot [21]. Since the speed of the hashing function has a critical importance for the efficiency of the hash table, SLCT employs the fast and efficient Shift-Add-Xor string hashing algorithm [22]. This algorithm is not only used for hash table operations, but also for building summary vectors.

SLCT is given a list of log files and a support threshold as input, and after it has detected a clustering on input data, it reports clusters in a concise way by printing out line patterns that correspond to clusters, e.g.,

*Dec 18 * myhost.mydomain * connect from*

*Dec 18 * myhost.mydomain * log: Connection from * port*

*Dec 18 * myhost.mydomain * log:*

The user can specify a command line flag that forces SLCT to inspect each cluster candidate more closely, before it starts the search for clusters in the candidate table. For each candidate *C*, SLCT checks whether there are other candidates in the table that represent more specific line patterns. In the above example, the second pattern is more specific than the third, since all lines that match the second pattern also match the third. If candidates $C_1,...,C_k$ representing more specific patterns are found for the candidate *C*, the support values of the candidates $C_1,...,C_k$ are added to the support value of *C*, and all lines that belong to candidates $C_1,...,C_k$ are also considered to belong to the candidate *C*. In that way, a line can belong to more than one cluster simultaneously, and more general line patterns are always reported, even when their original support values were below the threshold. Although traditional clustering algorithms require that every point must be part of one cluster only, there are several algorithms like CLIQUE which do not strictly follow this requirement, in order to achieve clustering results that are more comprehensible to the end user [15].

By default, SLCT does not report the lines that do not belong to any of the detected clusters. As SLCT processes the data set, each detected outlier line could be stored to memory, but this is way too expensive in terms of memory cost. If the end user has specified a certain command line flag, SLCT makes another pass over the data after clusters have been detected, and writes all outlier lines to a file. Also, variable parts of cluster descriptions are refined during the pass, by inspecting them for constant heads and tails. Fig. 1 depicts the sample output from SLCT.

If the log file is larger, running SLCT on the data set just once might not be sufficient, because interesting cluster candidates might have very different support values. If the support threshold value is too large, many interesting clusters will not be detected. If the value is too small, interesting cluster candidates could be split unnecessarily into many subcandidates that represent rather specific line patterns and have quite small support values (in the worst case, there will be no cluster candidates that cross the support threshold).

```
$ slct-0.01/slct -o outliers -r -s 8% myhost.mydomain-log

Dec 18 * myhost.mydomain sshd[*]: connect from 172.26.242.178
Support: 262

Dec 18 * myhost.mydomain sshd[*]: log: Connection from
172.26.242.178 port *
Support: 262

Dec 18 * myhost.mydomain sshd[*]: fatal: Did not receive ident
string.
Support: 289

Dec 18 * myhost.mydomain * log: * * * *
Support: 176

Dec 18 * myhost.mydomain sshd[*]: connect from 1*
Support: 308

Dec 18 * myhost.mydomain sshd[*]: log: Connection from 1* port *
Support: 308

Dec 18 * myhost.mydomain sshd[*]: log: * authentication for *
accepted.
Support: 171

Dec 18 * myhost.mydomain sshd[*]: log: Closing connection to 1*
Support: 201

$ wc -l outliers
    168 outliers
```

Figure 1. Sample output from SLCT.

```
[mail.alert] sendmail[***]: NOQUEUE: SYSERR(***): Arguments too
long

[mail.crit] sendmail[***]: NOQUEUE: SYSERR(***): can not
chdir(/var/spool/mqueue/): Permission denied

[mail.crit] sendmail[***]: ***: SYSERR(root): collect: I/O error
on connection from ***, from=<***>

[mail.crit] sendmail[***]: ***: SYSERR(root): putbody: write
error: Input/output error

[mail.warning] sendmail[***]: STARTTLS=server, error: accept
failed=-1, SSL_error=5, timedout=0

[mail.warning] sendmail[***]: STARTTLS: read error=read R BLOCK
(-1)

[mail.warning] sendmail[***]: STARTTLS: read error=syscall error
(-1)

[mail.warning] sendmail[***]: STARTTLS: write error=generic SSL
error (-1)

[syslog.notice] syslog-ng[***]: Error accepting AF_UNIX
connection, opened connections: 300, max: 300

[syslog.notice] syslog-ng[***]: Error connecting to remote host
(***), reattempting in 60 seconds

[auth.crit] su[***]: pam_krb5: authenticate error: Decrypt
integrity check failed (-1765328353)

[auth.crit] login[***]: pam_krb5: authenticate error:
Input/output error (5)

[auth.notice] login[***]: FAILED LOGIN 1 FROM (null) FOR root,
Authentication failure

[authpriv.info] sshd[***]: Failed password for root from *** port
*** ssh2

[mail.alert] imapd[***]: Unable to load certificate from ***,
host=*** [***]

[mail.alert] imapd[***]: Fatal disk error user=*** host=*** [***]
mbx=***: Disk quota exceeded
```

Figure 2. Sample anomalous log file lines.

In order to solve this problem, an iterative approach suggested in [8] could be applied. SLCT is first invoked with a relatively high threshold (e.g., 5% or 10%), in order to discover very frequent patterns. If there are many outliers after the first run, SLCT will be applied to the file of outliers, and this process will be repeated until the cluster of outliers is small and contains very infrequent lines (which are possibly anomalous and deserve closer investigation). Also, if one wishes to analyze one particular cluster in the data set more closely, SLCT allows the user to specify a regular expression filter that will pass relevant lines only.

Fig. 2 depicts sample anomalous log file lines that we discovered when the iterative clustering approach was applied to one of our test data sets (the mail server log file from Table 1). Altogether, four iterations were used, and the cluster of outliers contained 318,166, 98,811, 22,807, and 5,390 lines after the first, second, third, and fourth (the final) step, respectively. At each step, the support threshold value was set to 5%. The final cluster of outliers was then reviewed manually, and the lines representing previously unknown fault conditions were selected. The lines in Fig. 2 represent various system faults, such as internal errors of the *sendmail*, *imapd*, and *syslogd* daemon, but also unsuccessful attempts to gain system administrator privileges (for the reasons of privacy and security, real IP numbers, host names, user names, and other such data have been replaced with wildcards in Fig. 2). The lines that are not present in Fig. 2 represented various rare normal system activities or minor faults, such as scheduled tasks of the *crond* and *named* daemon, nightly restarts of the *syslogd* daemon during log rotations, unsuccessful login attempts for regular users, etc.

We have made many experiments with SLCT, and it has proven to be a useful tool for building log file profiles and detecting interesting patterns from log files. Table 3 presents the results of some our experiments for measuring the runtime and memory consumption of SLCT. The experiments were conducted on 1,5GHz Pentium4 workstation with 256MB of memory and Redhat 8.0 Linux as operating system. For all data clustering tasks, a word summary vector of size 5,000 counters was used. Since SLCT was also instructed to identify outlier points, four passes over the data were made altogether during the experiments. The results show that our algorithm has modest memory requirements, and finds many clusters from large log files in a relatively short amount of time.

V. Future Work And Availability Information

For a future work, we plan to investigate various association rule algorithms, in order to create a set of tools for building log file profiles. We will be focusing on algorithms for detecting temporal patterns, but also on algorithms for detecting associations between event attributes within a single event cluster.

SLCT is distributed under the terms of GNU GPL, and is available at http://kodu.neti.ee/~risto/slct/.

Acknowledgments

The author wishes to express his gratitude to Mr. Tõnu Liik, Mr. Ain Rasva, Dr. Paul Leis, Mr. Ants Leitmäe, and Mr. Kaido Raiend from the Union Bank of Estonia for their kind support. Also, the author thanks Mr. Bennett Todd and Mr. Jon Stearley for providing feedback about SLCT.

TABLE III. Runtime And Memory Consumption of SLCT

Data set	Support threshold	The number of detected clusters	The number of outlier points	Memory consumption	Runtime
Mail server log file	10%	17	318,166	1252 KB	7 min 17 sec
Mail server log file	5%	20	318,166	1372 KB	7 min 17 sec
Mail server log file	0.5%	181	41,365	4260 KB	7 min 54 sec
Cache server log file	10%	16	0	1352 KB	10 min 35 sec
Cache server log file	5%	32	0	1668 KB	10 min 35 sec
Cache server log file	0.5%	133	8	3512 KB	10 min 56 sec
HP OpenView event log file	10%	86	6,626	2640 KB	13 min 53 sec
HP OpenView event log file	5%	97	6,523	3308 KB	13 min 59 sec
HP OpenView event log file	0.5%	352	10,559	8256 KB	13 min 51 sec
Internet banking server log file	10%	56	2,769	1888 KB	35 min 25 sec
Internet banking server log file	5%	126	2,335	3508 KB	38 min 26 sec
Internet banking server log file	0.5%	663	214	25072 KB	45 min 8 sec
File and print server log file	10%	18	145	4540 KB	27 min 30 sec
File and print server log file	5%	35	145	6164 KB	28 min 42 sec
File and print server log file	0.5%	211	133	19096 KB	31 min 10 sec
Domain controller log file	10%	22	1,256	3596 KB	11 min 11 sec
Domain controller log file	5%	34	1,256	5112 KB	11 min 38 sec
Domain controller log file	0.5%	65	3,388	9132 KB	11 min 56 sec

REFERENCES

[1] C. Lonvick, "The BSD syslog Protocol", *RFC3164*, 2001.

[2] Stephen E. Hansen and E. Todd Atkins, "Automated System Monitoring and Notification With Swatch", *Proceedings of the USENIX 7th System Administration Conference*, 1993.

[3] Wolfgang Ley and Uwe Ellerman, logsurfer(1) manual page, unpublished (see http://www.cert.dfn.de/eng/logsurf/), 1995.

[4] Risto Vaarandi, "SEC - a Lightweight Event Correlation Tool", *Proceedings of the 2nd IEEE Workshop on IP Operations and Management*, 2002.

[5] H. Mannila, H. Toivonen, and A. I. Verkamo, "Discovery of frequent episodes in event sequences", *Data Mining and Knowledge Discovery*, Vol. 1(3), 1997.

[6] M. Klemettinen, H. Mannila, and H. Toivonen, "Rule Discovery in Telecommunication Alarm Data", *Journal of Network and Systems Management*, Vol. 7(4), 1999.

[7] Qingguo Zheng, Ke Xu, Weifeng Lv, and Shilong Ma, "Intelligent Search of Correlated Alarms from Database Containing Noise Data", *Proceedings of the 8th IEEE/IFIP Network Operations and Management Symposium*, 2002.

[8] L. Burns, J. L. Hellerstein, S. Ma, C. S. Perng, D. A. Rabenhorst, and D. Taylor, "A Systematic Approach to Discovering Correlation Rules For Event Management", *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management*, 2001.

[9] Sheng Ma and Joseph L. Hellerstein, "Mining Partially Periodic Event Patterns with Unknown Periods", *Proceedings of the 16th International Conference on Data Engineering*, 2000.

[10] Matt Bing and Carl Erickson, "Extending UNIX System Logging with SHARP", *Proceedings of the USENIX 14th System Administration Conference*, 2000.

[11] David Hand, Heikki Mannila, and Padhraic Smyth, *Principles of Data Mining*, The MIT Press, 2001.

[12] Pavel Berkhin, "Survey of Clustering Data Mining Techniques", unpublished (see http://citeseer.nj.nec.com/berkhin02survey.html), 2002.

[13] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim, "ROCK: A Robust Clustering Algorithm for Categorical Attributes", *Information Systems*, Vol. 25(5), 2000.

[14] Venkatesh Ganti, Johannes Gehrke, and Raghu Ramakrishnan, "CACTUS – Clustering Categorical Data Using Summaries", *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1999.

[15] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan, "Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1998.

[16] Charu C. Aggarwal, Cecilia Procopiuc, Joel L. Wolf, Philip S. Yu, and Jong Soo Park, "Fast Algorithms for Projected Clustering", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1999.

[17] Sanjay Goil, Harsha Nagesh, and Alok Choudhary, "MAFIA: Efficient and Scalable Subspace Clustering for Very Large Data Sets", *Technical Report No. CPDC-TR-9906-010*, Northwestern University, 1999.

[18] Rakesh Agrawal and Ramakrishnan Srikant, "Fast Algorithms for Mining Association Rules", *Proceedings of the 20th International Conference on Very Large Data Bases*, 1994.

[19] Roberto J. Bayardo Jr., "Efficiently Mining Long Patterns from Databases", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1998.

[20] Jiawei Han, Jian Pei, and Yiwen Yin, "Mining Frequent Patterns without Candidate Generation", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2000.

[21] Justin Zobel, Steffen Heinz, and Hugh E. Williams, "In-memory Hash Tables for Accumulating Text Vocabularies", *Information Processing Letters*, Vol. 80(6), 2001.

[22] M. V. Ramakrishna and Justin Zobel, "Performance in Practice of String Hashing Functions", *Proceedings of the 5th International Conference on Database Systems for Advanced Applications*, 1997.