A Clustering Algorithm for Logfile Data Sets

Risto Vaarandi

Department of Computer Engineering
Tallinn Technical University
Raja 15, Tallinn, Estonia
risto.vaarandi@eyp.ee

Abstract. Today, vast amounts of system status and health information are stored in logfiles. Therefore, mining patterns from logfiles is an important system management task. This paper presents a novel clustering algorithm for logfile data sets which helps one to detect frequent patterns from logfiles, to build logfile models, and to identify anomalous logfile lines.

1 Introduction

System monitoring - surveillance of the system for possible faults and malfunctions - is an important part of system management. When appropriate monitoring techniques are applied to system components, most of the system faults and malfunctions are detected early, before they will escalate to more serious problems.

One of the most important monitoring techniques is the logfile monitoring. Today, almost every application, service, operating system, network device, or other system component is able to produce a comprehensible logfile where all relevant events concerning the system component are logged. The notion of relevance is domain-dependent, but as a general rule, all fault conditions are considered relevant and logged. Therefore, logfiles are an excellent source for determining the health status of the system.

Because of the importance of logfiles as the source of system health information, a number of tools have been developed for monitoring logfiles, e.g., Swatch [1], Logsurfer [2], SEC [3], etc. Since normally events are logged as single-line textual messages, a typical logfile monitoring tool is a script or program that inspects every line added to the logfile, by comparing the line with patterns from the database of *fault message patterns*. If a pattern (or several patterns) match the line, the logfile monitor executes a certain action (e.g., sends an SMS-message to the mobile phone of the system administrator). Most often, the system administrators rely on their past experience and create the pattern databases by hand, by writing down patterns for all fault message types that are known to them. This commonly used approach has one serious flaw - only those faults that are already known to the system administrator can be detected. If a previously unknown fault condition occurs, the logfile monitor simply ignores the corresponding message in the logfile, since there is no match for it in the pattern database.

In order to solve this problem, the following model-based approach can be employed. First, the system administrator creates the database of fault message

patterns as usual. Then the system administrator tries to identify all logfile lines that do not represent fault conditions but rather reflect normal system activity, e.g., messages about successful completion of transactions. Once such lines have been identified (if there are any), the system administrator creates the database of *normal message patterns* that match those lines. Those two pattern databases constitute the model of the logfile. If a message is logged that does not fit the model, i.e., it does not represent any known fault or normal system activity, the message can be regarded as *anomalous* and directed to further processing.

However, the model-based approach works well only if the system administrator has created a good model for the logfile. For example, if the database of normal message patterns is not precise or is incomplete, many false alarms could be produced. Furthermore, if the logfile is larger and contains a wide variety of messages, the task of creating the model for it by hand can be extremely tedious, time-consuming, and error-prone. Thus, it is essential to have methods and tools for automating the model creation. Unfortunately, relatively little work has been done in this particular area, and currently no such open source tools are available.

One appealing choice for solving this problem is the employment of data clustering algorithms. Clustering algorithms aim at dividing the set of objects into groups, where objects in each group (or cluster) are similar to each other (and as dissimilar as possible to objects from other groups). When logfile 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. If such natural clusters could be detected with a software tool, it would greatly alleviate the problem of logfile model generation.

It should be noted that not all logfiles need to be analyzed with such a tool. If the file is small, or if only a few different messages are logged into the logfile, the model for it can be created manually with a little effort. Therefore, this paper focuses on the logfiles that are larger, and contain a large number of different messages.

In this paper, the author proposes a new clustering algorithm for mining patterns from logfiles, and presents 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 logfile data sets, section 4 describes SLCT, and section 5 concludes the paper.

2 Related work on data clustering

Clustering methods have been researched extensively over the past decades, and many algorithms have been developed [4]. The clustering problem is often defined as follows: given a set of points with n attributes in the data space $\Re^n$, 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 [5, 6]. 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 [5]), the choice of the right function is often not an easy task. Note that logfile 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 logfile 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 logfile data in the rest 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 $n \geq 15$) [4, 7, 8]. Furthermore, traditional clustering methods are often unable to detect natural clusters that exist in subspaces of the original high-dimensional space [7, 8]. 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 logfile data, since logfile 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 [7] and MAFIA [9] algorithms closely remind the Apriori algorithm for mining frequent itemsets [10]: 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 [11, 12, 13]. The CACTUS algorithm [6] 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 tends to generate clusters with stretched shapes, which is undesirable if one wants to discover patterns from logfiles. The PROCLUS algorithm [8] uses the K-medoid method [4] for detecting K clusters in subspaces of the original space. However, in the case of logfile 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 logfile lines, largely because they don't take into account the nature of logfile data. In the next section, we will first discuss the properties of logfile data, and then we will present a fast clustering algorithm that relies on these properties.

3 Clustering logfile data

3.1 The nature of logfile 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 logfiles at the word level, there are two important properties that distinguish logfile data from a generic data set.

Firstly, most 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 logfile data.

Table 1. Occurrence times of words in logfile data

Data set	*Data set size*	*Total # of different words*	*# of words occur-ring once*	*# of words occur-ring 2 times or less*	*# of words occur-ring 3 times or less*	*# of words occur-ring 5 times or less*	*# of words occur-ring 10 times or less*	*# of words occur-ring 20 times or less*
Mail-server logfile (Linux)	1025.3MB 7,657,148 lines	1,700,840	848,033 (49.9%)	1,350,581 (79.4%)	1,404,748 (82.6%)	1,443,159 (84.8%)	1,472,296 (86.6%)	1,493,160 (87.8%)
Cache server logfile (Linux)	1088.9MB 8,189,780 lines	1,887,780	1,023,029 (54.2%)	1,250,697 (66.3%)	1,359,535 (72.0%)	1,456,489 (77.2%)	1,568,165 (83.1%)	1,695,336 (89.8%)
Authentic ation server logfile (Win 2000)	1043.9MB 4,891,883 lines	4,016,009	3,948,414 (98.3%)	3,949,773 (98.4%)	3,950,439 (98.4%)	3,951,492 (98.4%)	3,953,698 (98.5%)	3,956,850 (98.5%)

The results show that a majority of words are very infrequent, and a significant fraction of words appear just once in the data set. The similar phenomenon has been observed for World Wide Web data, where during an experiment nearly 50% of the words were found to occur once only [14].

The second important property of logfile data is that there are many strong correlations between words that occur frequently. This is not surprising, since before logging, a message is generally formatted according to a certain format string, where parts of the format string are constants, e.g.,

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

When messages of the same type are logged many times, there will also be many lines in the data set which contain all of the constant parts of the format string together. In the next subsection we will present a clustering algorithm that relies on the special properties of logfile data.

3.2 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 logfile data, and uses the density based approach for clustering. Points that do not belong to any of the detected clusters are considered to form a special cluster of *outliers*.

The data space is assumed to contain data points with categorical attributes, where each point represents a line from a logfile data set. The attributes of each data point are the words from the corresponding logfile 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 $i_1,...,i_k$ ($1 \leq k \leq n$) 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, and is somewhat similar to the CACTUS algorithm [6] – 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 a 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 [7].

The first step of the algorithm reminds very closely the popular Apriori algorithm for mining frequent itemsets [10], 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 [11, 12, 13], since the candidate generation and testing involves exponential complexity. Secondly, since one of the properties of logfile data is that there are many strong correlations between frequent words, it makes very 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.

Note that the presence of many strong correlations between frequent words also means that if candidates are generated at once, their number is not likely to be very large, and is often not much larger than the number of frequent words itself (unless the user has given a rather low support threshold value). Therefore, the candidates are very likely to fit into the main memory. However, the memory cost of the algorithm is still an important issue, and in the next subsection we will discuss this matter in more detail.

3.3 The issue of memory cost

Although our algorithm makes just two passes over the data and is therefore fast, there is still one problem which might hinder its use – under certain circumstances the algorithm could consume a lot of memory.

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. Table 2 presents an experiment for measuring the in-memory vocabulary sizes for three data sets (each vocabulary was implemented as a move-to-front hash table which is an efficient data structure for accumulating words [14]).

The results show that even for medium-size 1GB data sets, in-memory vocabularies could occupy hundreds of megabytes of memory. As the size of the data set grows, the situation deteriorates further, and the vocabulary could not fit into the main memory anymore.

Table 2. In-memory vocabulary sizes

Data set	*Size*	*Total # of different words*	*The size of the vocabulary*
Mailserver logfile (Linux)	1025.3MB, 7,657,148 lines	1,700,840	98MB
Cache server logfile (Linux)	1088.9MB, 8,189,780 lines	1,887,780	153MB
Authentication server logfile (Win2000)	1043.9MB, 4,891,883 lines	4,016,009	214MB

On the other hand, one of the properties of logfile 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 [15], 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 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 most of the counter values will never cross the support threshold (unless a very low threshold value has been specified). Table 3 presents an experiment for measuring the effectiveness of the word summary vector technique for three data sets (each counter in the vector 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 experiment, vocabulary sizes decreased 25-100 times). On the other hand, the memory requirements for storing the vector itself are relatively small. For example, the largest vector we used during the experiments occupied less than 400KB of memory.

Table 3. The effectiveness of the summary vector technique

Data set	*Support threshold*	*Vector size*	*Total # of different words*	*# of words in the vocabulary*	*Reduction factor*
Mailserver logfile (Linux)	1% (76,571)	5,000	1,700,840	40,935	41.54
Cache server logfile (Linux)	1% (81,897)	5,000	1,887,780	18,998	99.36
Authentication server logfile (Win2000)	1% (48,918)	5,000	4,016,009	118,208	33.97
Mailserver logfile (Linux)	0.1% (7,657)	20,000	1,700,840	61,244	27.77
Cache server logfile (Linux)	0.1% (8,189)	20,000	1,887,780	50,246	37.57
Authentication server logfile (Win2000)	0.1% (4,891)	20,000	4,016,009	73,376	54.73
Mailserver logfile (Linux)	0.01% (765)	100,000	1,700,840	66,849	25.44
Cache server logfile (Linux)	0.01% (818)	100,000	1,887,780	69,210	27.27
Authentication server logfile (Win2000)	0.01% (489)	100,000	4,016,009	128,922	31.15

If the user has specified a rather low support threshold value, there could be a large number of cluster candidates with very low 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.

4 Simple Logfile Clustering Tool

In order to implement the logfile 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, though 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 [14]. The speed of the hashing function has a critical importance for the efficiency of the hash table, because if a slower hashing function is used, the data access times can increase by an unacceptable margin. In order to avoid this, SLCT uses the fast and efficient Shift-Add-Xor string hashing algorithm [15]. This algorithm is not only used for hash table operations, but also for building summary vectors.

SLCT is given a list of logfiles 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
Support: 570

Dec 18 * myhost.mydomain * log: Connection from * port
Support: 570

Dec 18 * myhost.mydomain * log:
Support: 679
```

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 follow this requirement, in order to achieve

clustering results that are more comprehensible to the end user [7].

By default, SLCT does not report the points that do not belong to any of the detected clusters. Though outlier points are considered to form a special cluster, there is no concise description for this cluster, since it does not correspond to any line pattern. As SLCT processes the data set, each detected outlier point could be stored to memory, but this could be way too expensive in terms of memory cost - especially when a relatively high support threshold value has been specified by the end user, which tends to create few clusters and many outliers. Therefore, SLCT does not discover outliers by default. 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 points to a file. Note that when there are many outlier points, one can apply SLCT to the file of outliers (and possibly repeat this process iteratively for every new outlier file).

We have made many experiments with SLCT, and it has proved to be a useful tool for building logfile models and detecting interesting patterns from logfiles. Table 4 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 5000 was used. Since SLCT was also instructed to identify outlier points, four passes over the data were made altogether.

Table 4. Runtime and memory consumption of SLCT

Data set	*Support threshold*	*# of detected clusters*	*# of outlier points*	*Memory consumption*	*Runtime*
Mailserver logfile (Linux)	10% (765,714)	17	318,166	1252 KB	7min 17sec
Mailserver logfile (Linux)	5% (382,857)	20	318,166	1372 KB	7min 17sec
Mailserver logfile (Linux)	1% (76,571)	89	42,346	2780 KB	7min 38sec
Mailserver logfile (Linux)	0.5% (38,285)	181	41,365	4260 KB	7min 54sec
Cache server logfile (Linux)	10% (818,978)	16	0	1352 KB	10min 35sec
Cache server logfile (Linux)	5% (409,489)	32	0	1668 KB	10min 35sec
Cache server logfile (Linux)	1% (81,897)	90	11	2580 KB	10min 56sec
Cache server logfile (Linux)	0.5% (40,948)	133	8	3512 KB	10min 56sec
Authentication server logfile (Win2000)	10% (489,188)	22	1,256	3596 KB	11min 11sec
Authentication server logfile (Win2000)	5% (244,594)	34	1,256	5112 KB	11min 38sec
Authentication server logfile (Win2000)	1% (48,918)	46	1,256	7348 KB	11min 58sec
Authentication server logfile (Win2000)	0.5% (24,459)	65	3,388	9132 KB	11min 54sec

The results show that our algorithm has modest memory requirements, and finds many clusters from large logfiles in a relatively short amount of time. We also made some tests with CLIQUE algorithm, in order to measure the difference of the two algorithms in terms of runtime. Even for medium support threshold values (such as 10-20%), our algorithm was 5-10 times faster. As the support threshold value was lowered, the difference increased even further.

5 Future work and availability information

For a future work, we plan to investigate various association rule algorithms, in order to create an algorithm for detecting patterns that span over multiple logfile lines and fit into a certain time window. SLCT is distributed under the terms of GNU GPL, and is available at http://kodu.neti.ee/~risto/slct/.

References

1. Stephen E. Hansen and E. Todd Atkins. *Automated System Monitoring and Notification With Swatch*. Proceedings of the USENIX 7th System Administration Conference, pp. 145-155, 1993.
2. Wolfgang Ley and Uwe Ellerman. *logsurfer(1) and logsurfer.conf(4) manual pages*. http://www.cert.dfn.de/eng/logsurf/, 1995.
3. Risto Vaarandi. *Platform Independent Tool for Local Event Correlation*. Acta Cybernetica 15(4), pp. 705-723, 2002.
4. Pavel Berkhin. *Survey of Clustering Data Mining Techniques*. http://citeseer.nj.nec.com/berkhin02survey.html, 2002.
5. Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. *ROCK: A Robust Clustering Algorithm for Categorical Attributes*. Information Systems 25(5), pp. 345-366, 2000.
6. 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, pp. 73-83, 1999.
7. 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, pp. 94-105, 1998.
8. 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, pp. 61-72, 1999.
9. 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.
10. Rakesh Agrawal and Ramakrishnan Srikant. *Fast Algorithms for Mining Association Rules*. Proceedings of the 20th International Conference on Very Large Data Bases, pp. 487-499, 1994.
11. Roberto J. Bayardo Jr., Rakesh Agrawal, and Dimitrios Gunopulos. *Constraint-Based Rule Mining in Large, Dense Databases*. Proceedings of the 15th International Conference on Data Engineering, pp. 188-197, 1999.
12. Roberto J. Bayardo Jr. *Efficiently Mining Long Patterns from Databases*. Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 85-93, 1998.
13. Jiawei Han, Jian Pei, and Yiwen Yin. *Mining Frequent Patterns without Candidate Generation*. Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 1-12, 2000.
14. Justin Zobel, Steffen Heinz, and Hugh E. Williams. *In-memory Hash Tables for Accumulating Text Vocabularies*. Information Processing Letters, 80(6), pp. 271-277, 2001.
15. 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, pp. 215-224, 1997.