# Preface

This master's thesis concludes our *Masters degree in Technology - 5 year integrated study*. We have found it to be both interesting and challenging, and now look forward to new challenges in working life.

We would like to express our gratitude to our supervisor Anders Christensen, who conceived the idea of a configuration-less log analyser. During the past six months, he has always had his door open for us whenever we were in need of guidance or inspiration. We appreciate the opportunity to work with him on such an interesting subject.

We would also like to thank UNINETT Norid AS and Magni Onsoien who provided us with test material. Additionally we would like to send our thanks to Lasse Karstensen for helpful advices on programming issues, and for his endless patience.

Trondheim, July 22, 2009:

Jenny Marie Ellingsæter

Frode Sandholtbråten

# Abstract

System logs contain messages from a wide range of applications. They are the natural starting point when troubleshooting a system. The usual approach for analysing system logs is to write a number of regular expressions to match specific keywords and events. When the number of expressions grows large, the analysis solution becomes unmaintainable. In addition, the use of regular expressions requires the system administrator to have extensive knowledge of the system at hand.

This thesis presents methods for performing log analysis without regular expressions. This is an area of system administration that has attracted very few researchers. Therefore, little published research is available on the subject.

Much effort has been put into the task of generating patterns from log file. These patterns are an important prerequisites for statistical analysis. Patterns could also be used to identify transactions for use in Markov models.

None of the existing pattern mining algorithm for system logs produce satisfactory results. To solve the task at hand, a new method for mining patterns is developed. Several different approaches were tested. An approach based on inserting log lines into a tree structure turned out to be a very promising. It outputs good quality patterns and its resource use is moderate.

Log analysis without prior knowledge of the system at hand have been proven difficult. This thesis shows that methods where some basic knowledge of systems in general is exploited, are the most promising ones. Other approaches based on Markov models and neural networks are suggested in this thesis, but they have not been tested to full extend and require some more work before being useful.

# Contents

# List of Tables

# List of Figures

# Part I

# Assignment description and literature review

# Chapter 1

# The assignment

System logs contain messages from systems and applications, making it a good place to start when debugging a system. But, as they include messages from a wide range of applications, and sometimes from several hosts at the same time, it is close to impossible to single out the important messages. The amount of data recorded every day is often overwhelming, and most of the messages written to the logs are only routine messages. Consequently, the logs are only consulted when users are complaining about services that are not working or unexpected behaviour occurs.

In an ideal world, the system itself would notify the administrators about services that fails or events that require additional examination. The log system should be capable of deciding, by itself, if a log message or a cluster of log message are of importance for the service or not. Instead of having to read through all log files line by line, the administrators should only be informed about the important messages.

The usual approach to solve this problem is to create regular expressions to filter out important messages. As this approach requires keywords to be efficient, the administrators need a great deal of expertise and experience. Knowledge about the output from applications and systems is critical. As keywords and regular expressions are static, the only way to keep the configuration up to date is to add new regular expressions every time an application is installed or updated. The result is an endless amount of rules that are not maintained since no-one knows exactly what the rules do.

Instead of having to write large amount of rules to filter out unimportant messages, or noise, from the logs, the system should be capable of learning which messages are not part of a normal state. By making use of historical data, a clever log analyser should be able to filter out noise based on previous experience. By doing so, the administrators no longer have to spend time updating regular expressions by reading through the logs line by line.

Benefits of a dynamic solution includes reduction both in time spent administrating the log monitoring and the system resources necessary for parsing

log files. Another huge benefit is the possibility for identifying issues that exists but are yet to be discovered. These unknown problems are usually overlooked because no-one is actively looking for new error messages or deviations from what is defined as a normal state.

The focus of this thesis is investigate methods and algorithms that could be used in such an analyser. Reducing the amount of configuration necessary to an absolute minimum should be a top priority.

## 1.1 Objectives

There are two ways of attacking the massive amounts of log messages produced. The first approach is to identify anomalies in log files by detecting unusual messages and patterns. In this approach, the output from the analyser is a set of messages that have not occurred earlier, or messages that are known to indicate anomalies.

The second approach is to identify known patterns and remove them from the output. The result is all the messages that does not match a known pattern from earlier analysis'.

Both approaches have their strengths and weaknesses. Detection of every anomaly in a huge log file is unrealistic. Nevertheless, given the current state of affairs where system logs are usually not read at all, detection of a single anomaly is better than none at all.

On the other hand, removal of known patterns from log files before presenting them to system administrators might not reduce the amount of messages sufficiently. If the number of messages is still large, there is reason to believe that the logs will still be ignored.

A good pattern recognition technique is necessary in both approaches, but the second approach is more dependent upon it. When detecting anomalies, it is only necessary to be lucky once or twice.. When removing known patterns, it is necessary to reduce the size of the log file to a fraction of its original size.

Due to the reasons mentioned above, it was chosen to focus is on detection of anomalies.

The task, as described in this chapter, can therefore be reduced to two main objectives:

1. Find, evaluate and implement an efficient and accurate method for classifying log messages.
2. Evaluate, implement and test ideas for anomaly detection and presentation that do not depend upon regular expressions.

## 1.2 Regular expressions

The usual approach for filtering out interesting parts of system logs us by using regular expressions. The regular expressions make it possible to match characters, words, patterns of characters or strings. Regular expressions are written in a formal language, but there exists a number of different accents. A number of programming languages and tools have their own way of writing and evaluating regular expressions.

A simple example of a regular expression is *ab*c* which means zero or more instances of *b*. This expression will match *ac*, *abc*, *abbc* and so on, but not *cba*, *acb* or *cab*.

Regular expressions are used in data mining tools to filter out noise or to select interesting elements for display. They are the quick and easy way of finding specific messages or clusters of messages in log files. As long as you know what you are searching for, it is straight forward to write a regular expression that matches it.

A number of applications utilising regular expressions have emerged on the market trying to solve the problem of detecting anomalies. Lire[1] works in such a way. It has predefined regular expressions to match messages from databases, UNIX systems, web servers and similar services. Another examples of products taking the regular expression approach are LogWatch[2] and LogCheck[3].

Applications using regular expressions faces the problem of regular expressions being difficult to maintain. Usually, there is no standardised ways of structuring messages, making the them basically free-form text messages. This means that all applications sending messages may use their own standard. This makes it difficult to utilise regular expressions since they have to be tailored for specific applications.

Another complicating factor is that messages from applications might change between releases. Therefore, an expression that match an important message from version 2.1 of an application might not work with version 2.2. In order to match the new message, another regular expression has to be added to the application responsible for parsing log files. After a while, a significant amount of time is spent adding, updating or deleting regular expressions. Also, as the number of regular expressions explodes, the need for computer resources explodes in a similar way since every regular expression has to be checked against each message.

[1] See `http://www.logreport.org`
[2] See `http://www.logwatch.org`
[3] See `http://logcheck.org/`

## 1.3 The IT2901 project

The task of creating a tool that can use text mining techniques to extract important messages from system logs was given to a group of students in the IT2901 - "Informatics Project II" course at IDI/NTNU[4]. The students choose to focus on a few different algorithms for identifying abnormalities [1]: NIDES, Markov chains and sequence clustering algorithms such as Apriori. They also suggested that artificial neural networks and ID3 could be use to solve the task, even though they did not pursue these algorithms further.

Their report states that they implemented NIDES, Markov chains and ID3, but it does not state in what degree they were successful. Their report says very little about the results and in what degree they were able to identify anomalies in log files.

As the source code was not available at the time of writing, a code review and validation of their results has not been possible. Due to the lack of source code and description of their results in the report, a revisit of some of the algorithms is found necessary to determine if they are applicable in the log analysis domain.

## 1.4 Limiting the scope

The term "system logs" is dependent upon the system and the environment it is deployed in. Operating systems and applications have their own way of documenting what have been done. There exists a wide range of log standards and an even wider range of non-standard ways of creating log files.

In order to limit the scope, it was chosen to focus on the *BSD Syslog Protocol* as specified by *RFC 3164* [2].

The BSD Syslog Protocol was initially implemented in logging applications on UNIX systems. It is a widely used protocol where messages from a single host can either be stored locally or be forwarded to a log host. By consolidating logs, it is easier for system administrators to analyse messages and identify problems.

The reason for choosing Syslog is that it is widely used on UNIX systems and its derivatives, for example FreeBSD and Linux. It is well known in the open source community, and a breakthrough in this area would have widespread impact on log analysis for a large number of users and companies.

One benefit of choosing Syslog is that a part of the Syslog message is standardised. By having a standardised message format, it is easier to handle and automatically analyse messages. This is important in order to limit the amount of configuration needed in order to utilise the log analyser. As the

[4] See `http://www.idi.ntnu.no/`

format of the log is well known, the administrators does not have to specify it themselves.

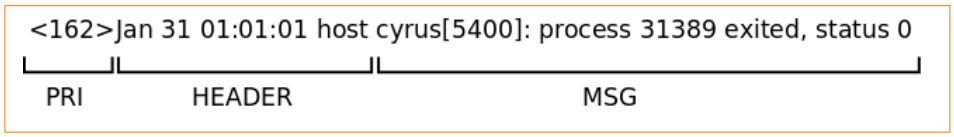



Figure 1.1: Illustration of the Syslog format

Figure 1.1 illustrates an example Syslog message. It consists of three parts. The priority (*PRI*) is a compound code, consisting of numbers. It describes the originating facility (process or daemon) of the message, and the severity of the message. The header (*HEADER*) contains a time stamp indicating when the message was logged, and a reference to the originating host. The third and last part is the message (*MSG*). It consist of a process name, an optional PID-number (process identifier, enclosed in square brackets), and a free-form text message. The contents of the Syslog message is decided by the application producing the message.

The BSD Syslog protocol, including facility and severity codes, is further described in Appendix A.

Even though the focus is on logs from the *Syslog* protocol, the methods used to filter out noise and identify interesting elements should be applicable on a wide range of system logs.

## 1.5 How to read this thesis

This report is divided into three separate parts to increase readability and to provide a natural border between different topics.

**Part one** consists of three chapters. **Chapter 1** is about the task given, a description of the problem at hand and the decisions made in order to limit the scope. **Chapter 2** provides an synopsis of algorithms believed to be useful in the log analysis domain while **Chapter 3** gives a presentation of some of the work done by others on the subject of log analysis.

**Part two** describes our ideas on how to solve the problems at hand. **Chapter 4** describes ideas that we thought of but chose not to pursue. The next chapter, **Chapter 5** describes some approaches to one of the most significant sub-problems at hand: pattern mining. Pattern mining is also the area of focus in **Chapter 6**, where pattern mining based on tree structures are presented. This approach tries to conquer some of the issues with the approaches presented in the previous chapter.

**Chapter 7** illustrates how the pattern mining described in chapters 5 and 6 could be used to perform statistical analysis on log files to detect anomalies. Statistical analysis is also the area of focus in **Chapter 8**, where messages

that appear regularly in log files are looked at. Part 2 ends with **Chapter 9** that describes how Markov models could be used to model message correlation and flow.

**Part three** summaries our work by first presenting our conclusion on the subject in **Chapter 10**. The second chapter of this part, **Chapter 11**, contains some thoughts of the road ahead.

## 1.6 Definitions and abbreviations

It is possible to interpret some words in different ways depending upon the context. This section lists the definitions used in this report.

**Application:** A program or a collection of programs working together to a solve a common task.

**Process:** A running instance of an application. There might be several processes from the same application.

**Log message:** A single message from a process. In most cases, this is a single line written to the log file.

**Logline:** Same as *log message*

**RFC:** Request for Comments - technical specifications and policy documents produced by the Internet Engineering Task Force (IETF). See `http://www.rfc-editor.org/` for more information.

**"Quality" when talking about patterns:** "Quality" is measured by looking at how accurate the patterns are, and how many variables are present in the formed patterns. To be useful, the patterns must be as accurate as possible with none or very few variables. High quality means that this is achieved. Low quality means that the patterns are inaccurate and/or contain too many variables.

# Chapter 2

# Algorithms

Pattern mining and anomaly detection have been performed in other settings for a long time. Pattern recognition in text have attracted a lot of researchers for a long time and a number of algorithms have been described to do it efficient.

This chapter describes some algorithms that might be applicable to the log analysis domain. Bayesian filtering, Apriori and decision three algorithms are all ways of creating patterns based on input. The ability to correctly classify log lines containing a large number of variables is crucial, making the pattern mining an important prerequisite for the rest of the algorithms.

Markov models are used to describe how a world evolves between states, making it usable for detecting anomalies. Artificial neural networks are ways of utilising output to learn how the system should behave when new input is introduced. It tries to mimic how the human brain behaves.

The chapter ends with a description of the statistical component of NIDES. NIDES is a intrusion detection system for computer systems which detects anomalies based on statistical analysis. Implementing NIDES is one way of utilising patterns made earlier.

## 2.1 Bayesian learning

Bayesian learning algorithms [3] use probabilities together with observed data to reach optimal decisions. The Bayesian algorithms are all based on Bayes' theorem. It relates the conditional and marginal probabilities of two events.

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)} \tag{2.1}$$

$P(A)$ and $P(B)$ are prior, or marginal, probabilities. A marginal probability

$P(A)$ is the probability of an event $A$, regardless of whether event $B$ occurs or not. Thus, it is *unconditional*. $P(A \mid B)$ and $P(B \mid A)$ are *conditional probabilities*. $P(A \mid B)$ is the conditional probability of the occurrence of event $A$, given $B$. $P(A \mid B)$ is also called the *posterior probability*, since it depends upon the specified value of $B$.

Following is a simple example. Imagine a log analyser whose objective is to classify log events as either normal or abnormal. It is known that abnormal events make up 1% of all log events. The remaining evens are considered as part of the normal system state. From previous experience with the log analyser it is known that some messages classified as abnormal, turns out to be normal events. Additionally, some abnormal events are not recognised as abnormal, and are classified as normal. To summarise:

| | | |
|---|---|---|
| $P(A)$ | = | probability of log event being normal = 99% |
| $P(A')$ | = | probability of log event being abnormal = 1% |
| $P(B \mid A)$ | = | probability of log event classified as abnormal, when event is normal = 10% |
| $P(B \mid A')$ | = | probability of log event classified as abnormal, when event is abnormal = 98% |
| $P(B)$ | = | probability of log event being classified as abnormal |

To find the probability of an event actually being normal when it is classified as abnormal, $P(A \mid B)$ , Bayes' theorem can be used. First we need to find $P\,(B)$. The *law of total probability* states that:

$$\begin{aligned} P(A) &= \sum_n P(B \mid A_n)P(A_n) \\ &= P(B \mid A)P(A) + P(B \mid A')P(A') \end{aligned} \tag{2.2}$$

By applying Equation 2.2, $P(B)$ is found to be 0.1088. Now that $P(B)$ is found, $P(A \mid B)$ can be calculated:

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)} = \frac{0.10 * 0.99}{0.1088} = 0.527$$

To conclude the example, there is a 52% chance that a log event classified as an abnormality actually is a normal event.

There exists many different Bayesian learning algorithms. In the following the naïve Bayes classifier will be described. It is names *naïve* because it makes the simplifying assumption that two events are independent. Events are independent in the sense that the occurrence of one event neither makes it more or less probable that the other event will occur.

The naïve Bayes classifier can be applied to tasks where each object $x$ consists of a set of attribute values. Figure 2.1 shows how such an object can

be illustrated as a Bayesian network [4]. The target function $f(x)$ can take any value from some finite set $V$. The naïve Bayes classifier is therefore particularly useful in classifying text documents, and it is also used in Bayesian spam filtering.

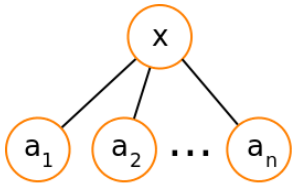



Figure 2.1: Object represented as an Bayesian network

To understand how the classifier works, it is necessary to understand MAP. A *maximum a posteriori* (MAP) hypothesis is the most probable hypothesis $h$ from among a set of candidate hypotheses $H$. Bayes' theorem can be used to calculate the posterior probability of each candidate hypothesis. D is the observed data and can be left out since it is independent of $h$.

$$\begin{aligned} h_{MAP} &\equiv \underset{h \in H}{\operatorname{argmax}} P(h \mid D) \\ &= \underset{h \in H}{\operatorname{argmax}} \frac{P(D \mid h)P(h)}{P(D)} \\ &= \underset{h \in H}{\operatorname{argmax}} P(D \mid h)P(h) \end{aligned} \tag{2.3}$$

The naïve Bayes classifier works as follows. It first receives a set of training examples of the target function $f(x)$. The classifier then receives a new object, consisting of a set of attribute values $< a_1, a_2, \ldots, a_n >$. The classifier's task is to predict the target value, or classification, of this new object. Based on the knowledge it holds so far, the classifier assigns the *most probable target value*, $v_{MAP}$, to the new object, based on the object's attribute values $< a_1, a_2, \ldots, a_n >$.

$$v_{MAP} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j \mid a_1, a_2, \ldots, a_n) \tag{2.4}$$

$$v_{MAP} = \underset{v_j \in V}{\operatorname{argmax}} \frac{P(a_1, a_2, \ldots, a_n \mid v_j)P(v_j)}{P(a_1, a_2, \ldots, a_n)} \tag{2.5}$$

$$= \underset{v_j \in V}{\operatorname{argmax}} P(a_1, a_2, \ldots, a_n \mid v_j)P(v_j) \tag{2.6}$$

The estimation of every $P(a_1, a_2, \ldots, a_n)$ is not feasible without an enormous set of training data. As mentioned above, the naïve Bayes classifier makes independence assumptions. Consequently, the probability of observing the set $a_1, a_2, \ldots, a_n$ is simply the product of the probability of each

individual attribute. The naïve Bayes classifier can therefore be expressed as in Equation 2.7. $V_{NB}$ denotes the target value output by the classifier.

$$v_{NB} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j) \prod_i P(a_i \mid v_j) \tag{2.7}$$

### Advantages and disadvantages

According to [3] the naïve Bayes classifier performs comparable to both neural networks and decision tree learning in some domains. However, the Bayes classifier needs an initial learning step to estimate the various $P(v_j)$ and $P(a_i \mid v_j)$ terms.

As opposed to other learning algorithm, the naïve Bayes classifier performs no search through the space of possible hypothesis, but simply count the frequencies of various attribute value combinations within the training examples.

Training is the challenge with Bayesian algorithms. By only counting frequencies of various attribute value combinations, they require a training set that can teach them what messages are normal and which are anomalies. As there are no way of creating a finite set of messages that could appear in log messages, it is only possible to give the filter a sense of what is normal or wrong. Secondly, the log messages are usually so short that only a few words could be used for determining whether the message is an anomaly.

It is possible to create a training set that could be used to train the Bayesian filter with the most common errors, but it is estimated that the number of false positives will be so high that the method itself is useless without tweaking. Therefore, Bayesian algorithms will not be investigated any further.

## 2.2 Apriori

The Apriori algorithm, developed by Agrawal and Srikant [5], seeks to discover association rules between items in a large database of sales transactions. The entrance of bar-code technology made it possible to store sales data. Information about sales transactions is often known as *basket data*. A typical record consists of the transaction date and the items purchased in the transaction. The Apriori algorithm attempts to find associations between items purchased in the same transaction. As an example, if a transaction contains eggs and bacon, it is also likely to contain tomato beans, {Eggs, Bacon} $\Longrightarrow$ {Tomato beans}. Such association-information is valuable in terms of for example sales campaigns and store layout.

Following is a description of the problem. The set $I = \{i_1, i_2, \ldots, i_m\}$ represents all items in the basket data. $D$ is the set of all transactions. Each transaction $T$ is associated with a unique identifier, $TID$, and consists of

a set of items such that $T \subseteq I$. If the set of items contains $k$ items, it is denoted a $k$-itemset. As an example, the set {Eggs,Bacon} is a 2-itemset.

An association rule implicates that $X \Longrightarrow Y$, where $X \subset I$, $Y \subset I$, and $X \cap Y = \emptyset$. Each association rule is measured in terms of its *support* and confidence. An association rule's support, $s$, represents how many percentage of the transactions in $D$ contain $X \cup Y$. The support measure may be used to eliminate uninteresting rules, and to assure that rules do not occur simply by chance [6]. The confidence of the rule describes how many percentage of all transactions that contains $X$, also contain $Y$. Confidence is a measure of of strong the relation between the elements in the rule is [6]. More formally, this can be written as,

$$\text{Support count} = \sigma(X) = |\{T | X \subseteq T,\ T \in D\}| \tag{2.8}$$

$$Support, s(X \Longrightarrow Y) = \frac{\sigma(X \cup Y)}{N} \tag{2.9}$$

$$Confidence, c(X \Longrightarrow Y) = \frac{\sigma(X \cap Y)}{\sigma(X)} \tag{2.10}$$

where $N$ is the total number of transactions. Apriori attempts to generate all association rules that have support greater than a minimum support, *minsup*, and confidence greater than a minimum, *minconf*.

Every itemset (and item) has a support *count*, which represents the number of transactions which contain the specific itmeset. All itemsets with support greater than or equal to minsup are called *large*. Itemsets with support less than minsup are called *small*.

The Apriori algorithm starts by discovering large itemsets. In the first iteration over the data, the support of individual items are calculated, and small items are discarded. In the following iterations over the data, a seed set of large itemsets found in the previous pass is used for generating new *candidate* itemsets. The support for these candidate itemsets are calculated as they are discovered. When completed, small itemsets are again discarded. This process of discovering large itemsets continues until no new itemsets are found. The intuition behind generating candidate itemsets based on the large itemsets found in the previous pass, is that any subset of a large itemset must itself be large.

```
1  L_1 = {large 1-itemset}
2  for ( k=2; L_{k-1} ≠ ∅; k++ ) do begin
3      C_k = apriori-gen(L_{k-1}); //New candidates
4      forall transactions t ∈ D do begin
5          C_t = subset(C_k, t); //Candidates contained in t
6          forall candidates c ∈ C_t do
7              c.count++;
8      end
9      L_k = {c ∈ C_k} | c.count ≥ minsup}
10 end
11 Answer = ⋃_k L_k;
```

Listing 2.1: Apriori algorithm, fetched from [5]

Listing 2.1 describes the overall Apriori algorithm. In the pseudocode, $L_k$ is a set of large $k$-itemsets. $C_k$ is a set of candidate $k$-itemsets. The algorithm starts by discovering all large 1-itemsets (line 1).

```
1  insert into C_k
2  select p.item_1, p.item_2, ..., p.item_{k-2}, q.item_{k-1}
3  from L_{k-1} p, L_{k-1} q
4  where p.item_1 = q.item_1, ..., p.item_{k-2} = q.item_{k-2},
5        p.item_{k-1} < q.item_{k-1};
6
7  // pruning
8  forall itemsets c ∈ C_k do
9      forall (k-1)-subsets s of c do
10         if (s ∉ L_{k-1} then
11             delete c from C_k;
```

Listing 2.2: apriori-gen function, fetched from [5]

The *apriori-gen* function (line 3) takes as argument the set of previously found large itemset, $L_{k-1}$. Listing 2.2 describes how apriori-gen operates. It first joins $L_{k-1}$ (renamed $p$) with $L_{k-1}$ (renamed $q$), i.e. a self-join. The join combines itemsets from $p$ and $q$ where all but the last item in the set are identical, and where the last item in $p$ is smaller than the last item in $q$. The resulting $k$-itemsets are stored in $C_k$. The following prune step removes all itemsets $c$ in $C_k$ that contains a $(k-1)$-subset not present in $L_{k-1}$.

The candidate itemsets generated by the apriori-gen function are stored in a hash tree. Leaf nodes contain a list of itemsets, while interior nodes contain a hash table. The subset-function (line 5 in Listing 2.1) uses this hash-tree to find all candidates contained in a transaction $t$. For every candidate found, that candidate's count is incremented. Finally, every candidate with a count larger than minsup will be stored in $L_k$ and next used as basis for generating new large itemsets. This procedure continues until no new itemsets can be

found, and the algorithm terminates, returning the union of all large large itemsets found.

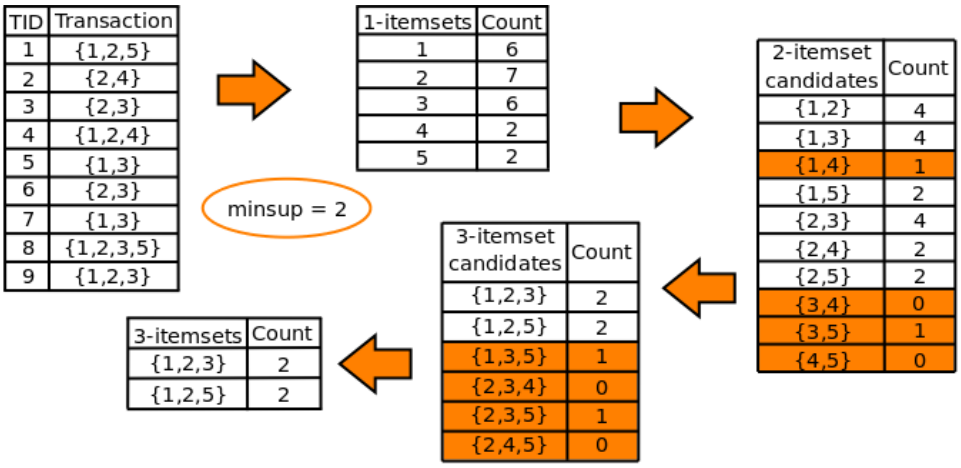



Figure 2.2: Simple Apriori example

Figure 2.2 illustrates a database of sales transactions. Each transaction consists of a *transaction id*, TID, and a set of *items* purchased in that particular transaction. The first step of the algorithm calculates the *support count* (abbreviated Count) for every unique item in the data set. The result is a set of 1-itemsets and their corresponding counts. The *minimum support* (minsup) is set to 2. None of the 1-itemsets have counts less than 2, and thus none of the 1-itemsets are pruned. The second step of the algorithm performs a self-join on the set of 1-itemsets. Since the sets $\{1, 4\}$, $\{3, 4\}$, $\{3, 5\}$, and $\{4, 5\}$ occur less than two times in the sales transaction database, these itemsets are pruned. The algorithm continues by performing a self-join on the remaining 2-itemsets, returning a set of 3-itemset candidates. Only two 3-itemsets satisfy minsup, and a fourth self-join is performed on these two itemsets. The result is two 3-itemsets, $\{1, 2, 3\}$ and $\{1, 2, 5\}$, both with support count 2. A last self-join would prove that no new large itemsets are to be found.

### Advantages and disadvantages

According to Risto Vaarandi [7], Apriori has exponential complexity. The Apriori algorithm tests a large number of frequent word combinations, when in fact only a small number of combinations are actually present in the data set. Vaarandi has tried to overcome some of the problems of Apriori in his work. SLCT and LogHound, described in Sections 3.2 and 3.3, are based on the Apriori algorithm.

## 2.3 Decision tree learning: ID3 and C4.5

The ID3 [8] algorithm and its successor, the C4.5 [8], are decision tree learning algorithms. Decision tree learning is a way of approximating discrete-valued target functions where the learned functions is represented by a decision tree. Another way is to present the threes as sets of if-then rules to improve human readability.

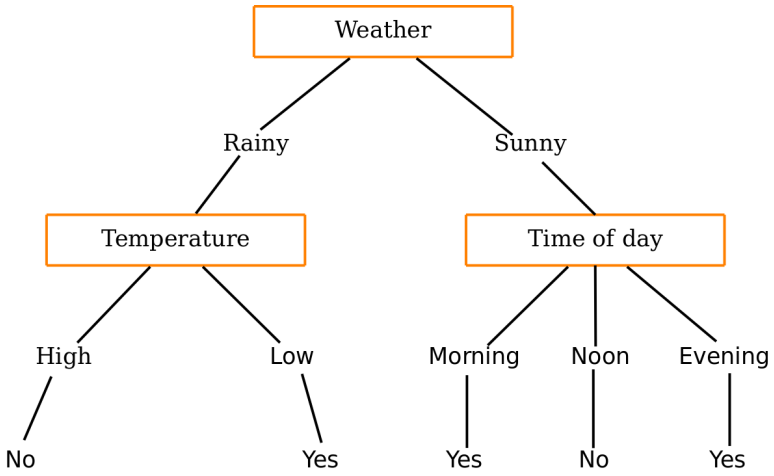



Figure 2.3: A simple example of a decision tree for deciding if one should go fishing.

The decision tree algorithm builds a tree from the root to some leaf node. Each node represents a test of some attribute. Each branch corresponds to one of the possible values for the attribute.

A simple decision tree is illustrated in Figure 2.3. It decides if one should go fishing based on information about the weather, temperature and time of day.

Mitchell [8] says in his book that decision tree learning is best suited for problems with the following characteristics:

- *Instances are represented by attribute-value pairs.* Instances are described by a fixed set of attributes, (e.g., *Temperature*) and their values (e.g., *Hot*).
- The target function has discrete output values.
- Disjunctive descriptions may be required.
- The training data may contain errors.
- The training data may contain missing attribute values.

### 2.3.1 ID3 algorithm

The ID3 algorithm were invented by Ross Quinlan [8]. The ID3 algorithm learns decision trees by constructing them top-down. Each instance at-

tribute is evaluated using a statistical test to determine how well it alone classifies the training examples. This done in order to answer the question *"which attribute should be tested at the root of the tree?"*. At the end, it constructs a greedy search for an acceptable decision tree which the algorithm never backtracks. This avoids having to reconsider earlier choices.

Mitchell [8] specifies in his book a simplified version of the ID3 algorithm. In the following algorithm description, *Input_examples* are the training examples for the decision three. *Target_attribute* is the attribute whose value is to be predicted by the tree. *Attributes* is a list of other attributes that may be tested by the learned decision tree. The algorithm returns a decision tree that correctly classifies the given input examples.

Algorithm ID3(*Input_examples, Target_attribute, Attributes)*:

- Create a *Root* node for the tree.
- If all *Input_examples* are positive, Return the single-node tree *Root*, with label = +
- If all *Input_examples* are negative, Return the single-node tree *Root*, with label = -
- If *Attributes* is empty, Return the single-node tree *Root*, with label = most common value of *Target_attribute* in *Input_examples*.
- Otherwise Begin
    - A ← the attribute from *Attributes* that best[1] classifies *Input_examples*
    - The decision attribute for *Root* ← A
    - For each possible value, $v_i$, of A,
        - Add a new tree branch below *Root*, corresponding to the test A = $v_i$.
        - Let $Input\_examples_{v_i}$ be the subset of *Input_examples* that have values $v_i$ for A
        - If $Input\_examples_{v_i}$ is empty
            - The below this new branch add a leaf node with label = most common value of *Target_attribute* in *Input_examples*
            - Else below this new branch add the subtree ID3($Input\_examples_{v_i}$, *Target_attribute, Attributes* - {A})
- End
- Return *Root*

A central task in ID3 is selecting the attribute to test at each node. A statistical property, the *information gain* is defined. It measures how well a given attribute separates the training examples according to their target classification. The *information gain* is used to select among the candidate attributes at each step while growing the three.

To calculate the *information gain*, one first need to define the entropy. It characterizes the (im)purity of an arbitrary collection of examples. In equation 2.11, $c$ is the different values a target attribute can have and $p_i$ is the proportion of S belonging to class $i$. The logarithm is base 2 because entropy

[1]The best attribute is the one with highest *information gain* as defined in Equation 2.12.

is a measure of the expected encoding length measured in *bits*.

$$Entropy(S) \equiv \sum_{i=1}^{c} -p_i log_2 p_i \tag{2.11}$$

The *Information Gain (Gain(S, A))*, where A is an attribute, is defined as follows:

$$Gain(S, A) \equiv Entropy(S) - \sum_{v \in (Values(A))} \frac{|S_v|}{|S|} Entropy(S_v) \tag{2.12}$$

In Equation 2.12, Values(A) is the set of all possible values for attribute A. $S_v$ is the subset of S for which attribute A has value $v$.

The information gain is used by ID3 to select the best attribute at each step in growing the tree. The value of Gain(S, A) describes how many bits that are saved when encoding the target value of an arbitrary member of S, by knowing the value of attribute A.

### 2.3.2 Expanding ID3: CS4.5

The C4.5 algorithm is an evolution of ID3. It was designed by Quinlan to solve some of the problems identified in ID3 [8]:

- Incorporating continuous-values attributes - ID3 is only capable of handling a discrete set of values.
- Alternative measures for selecting attributes - The information gain measure favours those attributes that have a very large number of possible values.
- Handling training examples with missing attribute values - CS4.5 assigns a probability to each of the possible values of A.
- Handling attributes with differing costs
- Rule post-pruning - attempts to remove branches that do not help and replacing them with leaf nodes.

A further improved version of ID3/C4.5, called See5/C5, is available from `http://www.rulequest.com/` as a commercial solution.

### 2.3.3 Using decision trees in log analysis

Decision tree algorithms could be used to mine message patterns from log files. Mining patterns are a central problem when talking about a configuration-less log analyser since it is necessary to classify messages.

Decision trees are usually used to solve decision problems. It requires training data that exposes both the questions and the answers. Although the

idea is simple, as illustrated in Figure 2.3, a decision tree containing every log message will be fairly complex.

The characteristics described by Mitchell [8] for problems well suited for decision trees does not match the log anomaly problem very well. There are no attribute-value pairs or discrete output values.

The decision trees might be used for learning patterns in log files, but the idea of using decision trees to classify anomalies will not be explored any further.

## 2.4 Markov chains and Markov models

Markov chains are based on state space diagrams and were first introduced by Andrey Markov in 1906. The Markov chain is a stochastic process with the Markov property. The Markov property states that *given the present state, future states are independent of the past states*. This means that the description of the present state should contain all the information that could influence the evolution of the process. The formal definition of a discrete Markov chain:

$$Pr(X_{n+1} = x | X_n = x_n, \ldots, X_1 = x_1) = Pr(X_{n+1} = x | X_n = x_n) \quad (2.13)$$

The continuous-time Markov process:

$$Pr(X_{n+1} = x | X_n = y) = Pr(X_n = x | X_{n-1} = y) \quad (2.14)$$

Markov models are created by identifying all possible states in the world. Then, the transitions must be identified and the chain must be parameterised by specifying how much time is spent in each state or the probability for a transition from a state to another. In the regular Markov model, the only variables are the state transition probabilities since all the states are directly visible.

A simple Markov model is illustrated in Figure 2.4. This model has a total of four states (A, B, C and D). State A is *transient*, meaning that it will not be possible to return to it after going to state D. States that can always be revisited in the future after leaving the state are called *recurrent states*. In Figure 2.4, states B, and C are *recurrent*. States A, B and C forms a Markov chain since they can all reach each other.

When the model is created, it must be calibrated in order to reflect the actual world. This especially applies to the time spent in each state or the probability for a transition from one state to another.

A number of variants of Markov models and chains exists. The most significant variants are *Hidden Markov models (HMM)* and *Markov chain Monte*

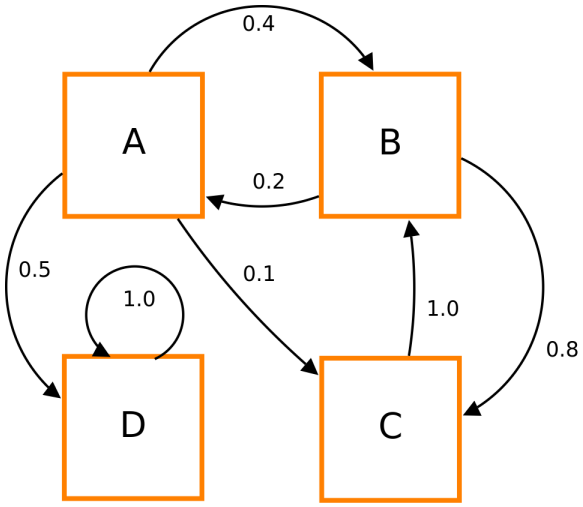



Figure 2.4: Example of a Markov model

*Carlo (MCMC).*

The Hidden Markov model is a statistical model where the system is assumed to be a Markov process where there are hidden parameters in the observable data. The HMM is used in pattern recognition applications such as speech, handwriting and bioinformatics [9].

Markov chain Monte Carlo is a class of algorithms. It contains algorithms for sampling from probability distributions based on constructing a Markov chain that has the desired distribution as its equilibrium distribution. It is widely used in Bayesian statistics, computational physics and computational biology.

Markov models are interesting in this session because it generates states based on historical data to predict future development. The Markov models can be applied to a increase level of complexity in the log files, from the simplest one being transitions between processes to the most complex where patterns within a single process are modelled.

The Markov algorithm is utilised in Chapter 9 to make models of how message flow and correlation appears in system logs.

## 2.5 Artificial neural networks

The artificial neural network is designed to simulate biological learning systems built of interconnected neurons. Each neuron in the artificial neural network takes a number of inputs and outputs a single, real-valued output.

### 2.5.1 Relation to the human brain

In the human brain, neurons collect input signals from a structure called dendrites. An electrical signal is sent trough a long stand, the axon, that splits into thousand of branches. At the end of each branch, a synapse converts the activity from the axon into electric effects that inhibit or exhibits activity from the axon into electrical effects that inhibit or excite activity in the connected neurons. Learning is induced by changing the effectiveness of the synapse's influence on other neurons.

The switching speed of the human neurons is about $10^{-3}$ seconds. In total, there are approximately $10^{11}$ neurons in a human brain. Combined, the neurons is capable of complex operations such as face recognition in about $10^{-1}$ seconds. In order to be able to do such a complex tasks, scientists believe that the brain uses highly parallel processes operating on representations that are distributed over many neurons [10].

The artificial neural networks try to simulate this highly parallel network of interconnected neurons. A simple artificial neuron is illustrated in Figure 2.5. In this figure there are a number of inputs, $X_1$ to $X_n$, a teach/use switch, an input for teaching and a single output.

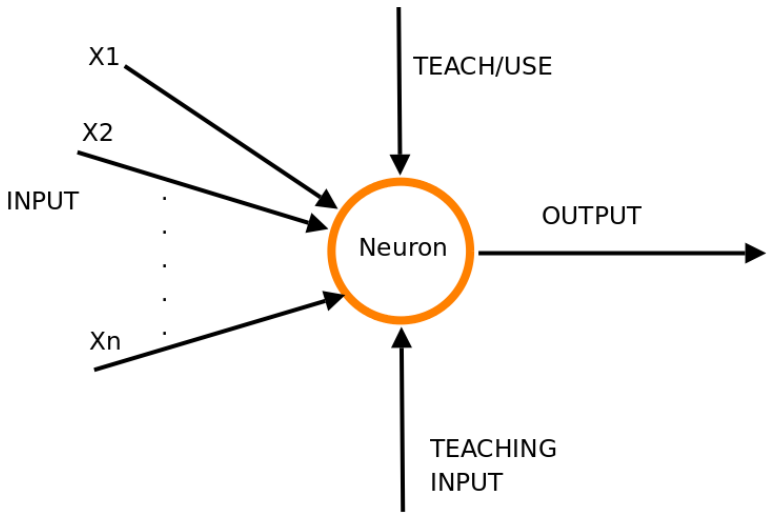



Figure 2.5: A simple neuron

In his book, Mitchell [10] states that artificial neural network learning is well-suited to problems in which the training data corresponds to noisy, complex sensor data such as inputs from cameras and microphones.

A prototype of artificial neural network learning is Pomerleu's ALVINN [10], a system using artificial neural networks to steer an autonomous vehicle driving on public speedways.

### 2.5.2 Network topologies

Simple, artificial neurons (or *units*), work together to create networks. Each node receives input from neighbours or external sources, and computes an output which is sent to other units. The network consists of three distinct types of units:

- *input* units: receive data from outside the neural network.
- *hidden* units: receive data from input units and forwards the output to other units.
- *output* units: receive data from either *input* or *hidden* units and send the data out of the neural network.

A simple example of a neural network is given in Figure 2.6.

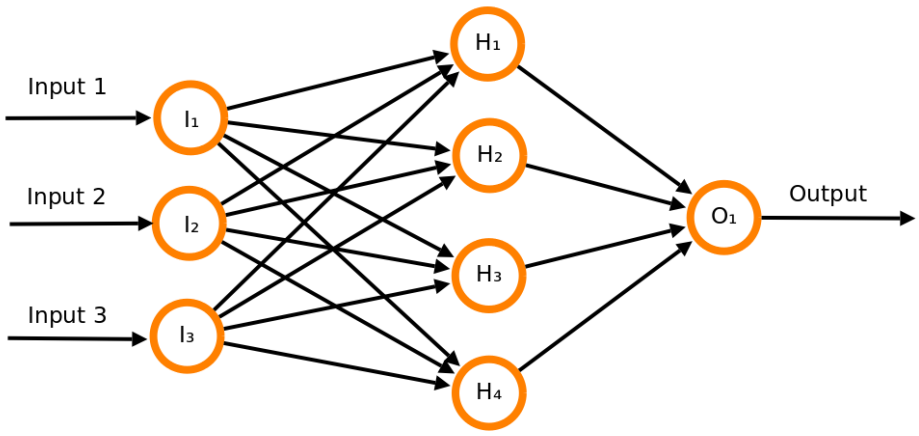



Figure 2.6: A simple neural network with three input nodes (I), four hidden nodes (H) and one output node (O).

According to Kröse and van der Smagt [11], there are two main network topologies in artificial neural networks:

*Feed-forward* networks, where the data flow from input to output units is unidirectional. The data processing can extend multiple (layers of) units, but no feedback connections are present.

*Recurrent* networks that do contain feedback connections. Contrary to feed-forward networks, the dynamical properties of the network are important. In some cases, the activation values of the units undergo a relaxation process such that the network will evolve to a stable state in which these activations do not change any more. In other applications, the change of the activation values of the output neurons are significant, such that the dynamical behaviour constitutes the output of the network [12].

Examples of feed-forward networks are Perceptron and Adaline. Perceptron is explained in a subsequent section. The Jordan network, Elman network and Hopfield network are examples of recurrent artificial neural networks. These recurrent networks are described in detail by Kröse and van der Smagt's [11].

### 2.5.3 Learning

Kröse and van der Smagt [11] classifies the learning situations for artificial neural networks into two distinct classes:

*Supervised* or *Associative learning*, in which the network is trained by providing it with input and matching output patterns. These input-output pairs can be provided by an external teacher, or by the system which contains the network (self-supervised). In this case, the patterns have to be created before the teaching begins.

*Unsupervised learning* or *Self-organisation* in which an (output) unit is trained to respond to clusters of pattern within the input. In this paradigm the system is supposed to discover statistically salient features of the input population. Unlike the supervised learning paradigm, there is no *a priori* set of categories into which the patterns are to be classified; the system must develop its own representation of the input stimuli.

In addition, a third method of teaching has been developed. The *reinforcement learning* is defined by Kaelbling and Moore [13]:

*Reinforcement learning* is the problem faced by an agent that must learn behaviour through trial-and-error interactions with a dynamic environment. It does so by getting rewards if the solution tried is the correct one. The agent attempts to determine both its immediate reward and the next state of the environment. This could be achieved by using *Markov decision processes* (MDPs).

### 2.5.4 Perceptrons

One of the classical feed-forward artificial neural networks was invented by Rosenblatt [14] in 1959. It uses *perceptrons* to simulate neurons. The Perceptron, as illustrated in Figure 2.7, takes a vector of real-valued inputs ($X_1$ to $X_n$) and calculates a linear combination of these inputs. The output is 1 if the result is greater than a predefined threshold, and -1 if not, as illustrated

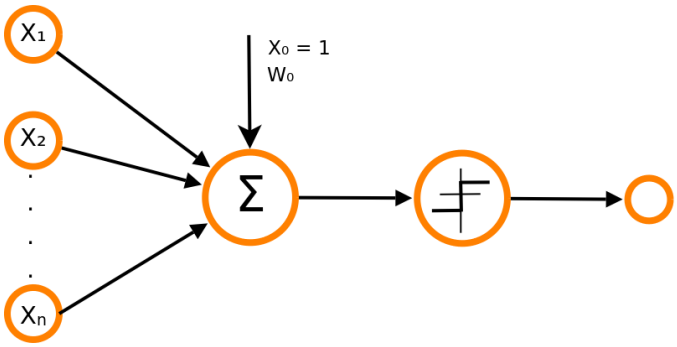



Figure 2.7: A simple perceptron

by the sign function in Figure 2.7 and also shown in Equation 2.15:

$$output(x_1, ...., x_n) = \begin{cases} 1 & \text{if } w_0 + w_1x_1 + w_2x_2 + ... + w_nx_n > 0 \\ -1 & \text{otherwise} \end{cases} \tag{2.15}$$

Here, inputs are named $x_1$ to $x_n$ while the learned weights are $w_0$ to $w_n$. The simple perceptron is capable of boolean AND, OR, NAND and NOR, but not XOR [10].

Equation 2.15 can also be expressed with vectors as shown in Equation 2.16:

$$o(\vec{x}) = sgn(\vec{w} \cdot \vec{x}) \tag{2.16}$$

where the *sign function (sgn)* is defined as follows:

$$sgn(y) = \begin{cases} 1 & \text{if y} > 0 \\ -1 & \text{otherwise} \end{cases}$$

There are two algorithms for learning the weights in Perceptron: The perceptron training rule and the delta rule. The perceptron training rule updates weights based on the error in the thresholded perceptron output, while the delta rule updates weights based on the error in the unthresholded linear combination of inputs [10]. Michell [10] also mentions linear programming as a possible learning algorithm.

The delta rule is the basis for the backpropagation algorithm that learns the weights for a multilayer network. The backpropagation algorithm uses gradient descent to attempt to minimize the squared error between the network output values and the target values for these outputs.

### 2.5.5 Sigmoid unit

Since the perceptron is only capable of expressing linear decision surfaces, a different unit is necessary in order to utilise the backpropagation algorithm. One example of such a unit is the *sigmoid* unit. It is very similar to the perceptron, but is based on a smoothed, differentiable threshold function as shown in Equation 2.17:

$$o = \sigma(\vec{w} \cdot \vec{x}) \tag{2.17}$$

where

$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

Detailed description of the backpropagation algorithm is outside the scope for this introduction to neural networks. The complete algorithm is described in detail by Mitchell [10] and it is also discussed by Kröse and van der Smagt [11].

### 2.5.6 Neural networks in log analysis

Neural networks are good at utilising a large amount of input values and give an output representing all the input values. In theory, it is designed to take lessons from its input values to improve its output.

Neural networks are used in such a way to filter out spam. SpamAssassin, as described in Section 4.1, utilize neural networks to decide whether or not an email message is spam. It does so by accumulating scores from a wide range of tests.

Although the theory behind the single neuron is simple, implementation of neural networks is a complex task requiring a significant amount of knowledge and previous experience with them. To utilise neural networks in the same way as SpamAssassin does, it is necessary to create a wide range of simple tests that can be applied to the log files.

Ideas using neural networks are discussed in Sections 4.2 and 11.3.

## 2.6 The statistical component of NIDES

NIDES [15], or Next-Generation Intrusion Detection Expert System, and its statistical component, is used to detect anomalous events on monitored computers. It does so by monitoring and learning the normal use and flagging behaviour that deviates significantly from what is considered normal. NIDES creates profiles for users, groups, remote hosts and the overall system. It was created by the Stanford Research Institute (SRI) [15].

### 2.6.1 Description

The algorithm only stores statistics such as frequencies, means and covariances instead of keeping a complete historical audit. It does not rely on information about known attacks. Instead, it relies on statistical analysis to spot abnormalities and new types of attacks.

As the NIDES algorithm is used to monitor a complete system, the authors of [15] found that is was useful to classify the different types of individual measures into four classes:

- *Intensity measures* - To track the number of audit records that occur in different time intervals. Can detect bursts of activity or prolonged, abnormal activity.
- *Audit record distribution measure* - Tracks all types of activity and compares it to a long-time profile in order to detect changes in the usage pattern.
- *Categorical measures* - For example, it could include the names of files accessed, terminal ID and names of remote hosts used.
- *Counting measures* - Might include CPU time used and the amount of I/O. Compared to a historical profile to determine if recent usage is abnormal.

#### NIDES statistics

NIDES generates a single test statistical value, $T^2$, to indicate the degree of abnormality in the user's behaviour. Large values indicates that there might be some abnormality while low values indicate that the usage pattern is normal compared to what has been done in the past. The $T^2$ is a result of a number of individual measures, $S_i$. Each $S_i$ is a measure of the degree of abnormality of behaviour with regard to a specific feature. $T^2$ is calculated as shown in Equation 2.18:

$$T^2 = \frac{({S_1}^2 + {S_2}^2 + ... + {S_n}^2)}{n} \tag{2.18}$$

#### Calculating S from Q from intensity measures

The individual $S$ values are calculated based on the class they belong to. The value $S$ is derived from a corresponding statistic called $Q$. A half-life value is used to make sure that the more recent audit records have more influence than older ones. For the intensity measures, $Q$ is the number of audit records that have arrived in the recent past. Here, *recent past* corresponds to the few last minutes if the half-life is set to 1 minute and the last few hours if it is set to 1 hour. To transform $Q$ to $S$, one need knowledge of the historical

distribution of $Q$. NIDES keeps a historical profile of all previous values of $Q$ in order to compare the current value with the historical ones.

When the historical distribution of $Q$ is known, the algorithm to transform $Q$ into $S$ is as follows [15]:

1. Let $P_m$ denote the relative frequency with which Q belongs to the $m^{th}$ interval. There are 32 values for $P_m$ with $0 \leq \text{m} \leq 31$.
2. For the $m^{th}$ interval, let $TPROB_m$ denote the sum of $P_m$ and all other P values that are smaller than or equal to $P_m$ in magnitude.
3. For the $m^{th}$ interval, let $s_m$ be the value such that the probability that a normally distributed variable with mean 0 and variance 1 is large than $s_m$ in absolute value equals $TPROB_m$. The value of $s_m$ satisfies the Equation:

$$P(|N(0,1)| \geq s_m) = TPROB_m \tag{2.19}$$

   $s_m$ is not allowed to be greater than 4.0
4. Suppose that after processing an audit record one find that the Q value is in the $m^th$ interval. Then $S$ is set equal to $s_m$, the $s$ value corresponding to $TPROB_m$.

**Calculating S from Q from all other measures**

Computing $S$ from $Q$ for all other measures is easier. $Q$ compares short-term behaviour against long-term behaviour. This is done by calculating a long-term profile for $Q$ with 32 intervals. Instead of measuring units of audit records, the range will be expressed in terms of the degree of similarity between the short-term profile and the long-term profile. The larger number, the less similarity there is. The Equation for calculating $TPROB_m$ is shown in Equation 2.20:

$$TPROB_m = P_m + P_{m+1} + .... + P_{31} \tag{2.20}$$

**Calculating the frequency distribution for Q**

The historical frequency distribution Equation 2.21 is used for all types of measures. To calculate the distribution, 32 bins are used and set a $Q_{max}$. $Q_{max}$ should be the maximum value that is expect to see for $Q$ and is dependent upon the particular type of measure one is considering. To calculate $P_m$ on the $k_{th}$ day:

$$P_{m,k} = \left(\frac{1}{N_k}\right) \sum_{j=1}^{k} (W_{m,j} 2^{-b(k-j)}) \tag{2.21}$$

In Equation 2.21, $k$ is the number of days since the monitoring began, $b$ is the half-life, $W_{m,j}$ is the number of audit records on the $j^th$ day for which $Q$

was in the $m^th$ interval and $N_k$ is the exponentially weighted total number of audit records that have occurred since the monitoring began. The formula for $N_k$ is shown as Equation 2.22 were $W_j$ is the number of audit records that occurred on the $j^th$ day.

$$N_k = \sum_{j=1}^{k} W_j 2^{-b(k-j)} \tag{2.22}$$

The authors of [15] recommend that the following recursive formulas are used instead of Equations 2.21 and 2.22 to avoid having to keep a large sum:

$$P_{m,k} = \frac{(2^{-b} P_{m,k-1} N_{k-1} + W_{m,k})}{N_k} \tag{2.23}$$

where $N_k$ is:

$$N_k = 2^{-b} N_{k-1} + W_m \tag{2.24}$$

**Computing the Q statistic for the intensity measures**

When initialising auditing for a user, the value of Q has to be set to a predetermined number. This could be zero or the statistical mean value for other users that fit the same utility model.

Formula 2.25 specifies how Q should be updated:

$$Q_{n+1} = 1 + 2^{-rt} Q_n \tag{2.25}$$

In formula 2.25, $t$ is the time since the last audit record and $r$ is the half-life. The half-life makes sure that $Q$ is mostly influenced by the most recent audit records.

**Computing the Q statistics for the audit record distribution measure**

We must calculate a long-term historical relative frequency, $f_m$, for each activity type. The sums of the $f_m$ might be greater than 1.0 even though each one is specified to be in the interval from 0.0 to 1.0. This because a single audit record might be a part of more than one activity type.

Equation 2.26 specifies how $f_m$ on the $k^{th}$ day is calculated:

$$f_{m,k} = \left(\frac{1}{N_k}\right) \sum_{j=1}^{k} (W_{m,j} 2^{-b(k-j)}) \tag{2.26}$$

Here, $N_k$ and $b$ are the same values as in Equation 2.21 while $W_{m,j}$ is the number of audit records on the $j^{th}$ day that indicate that the $m^{th}$ activity type occurred.

The value of $Q$ is defined in Equation 2.27:

$$Q_n = \sum_{m=1}^{M} \left[ \frac{(g_{m,n} - f_m)^2}{V_m} \right] \tag{2.27}$$

$g_{m,n}$ is defined as the relative frequency with which the $m^{th}$ activity type has occurred in the recent past (which ends as the $n^{th}$ audit record). $V_m$ is the approximate variance of the $g_{m,n}$.

$g_{m,n}$ could be calculated in two ways:

$$g_{m,n} = \left( \frac{1}{N_r} \right) \sum_{j=1}^{n} \left[ I(j,m) 2^{-r(n-j)} \right] \tag{2.28}$$

or iterative:

$$g_{m,n} = 2^{-r} g_{m,n-1} + \left[ \frac{I(n,m)}{N_r} \right] \tag{2.29}$$

In Equations 2.28 and 2.29, $j$ is an index denoting audit record sequence, *I(j,m)* is 1.0 if the $j^{th}$ audit record indicates activity of type $m$ has occurred and 0.0 otherwise. $r$ is the half-life and $N_r$ is the sample size for the $Q$ statistics, given by Equation 2.30:

$$N_r = \sum_{j=1}^{n} 2^{-r(n-j)} \tag{2.30}$$

$V_m$ is given by Formula 2.31:

$$V_m = \frac{f_m(1 - f_m)}{N_r} \tag{2.31}$$

$V_m$ is not allowed to be smaller than $0.01/N_r$.

### Computing the Q statistics for categorical measures

$Q$ for the categorical measures is calculated in the same way as for calculation $Q$ for the audit record distribution measure. The only difference is that $Q$ is only updated whenever the audit record contains information relevant to the particular measure instead of being updated for each audit record.

#### Computing the Q statistic for counting measures

The counting measures are translated into categorical measures by dividing counts into 32 geometrically scaled intervals. When a value arrived, it is classified into interval $m$ which triggers the categorical event $m$. As so, the $Q$ statistic is calculated in the same fashion as for any other categorical measure.

### 2.6.2 Using NIDES to detect anomalies in log files

NIDES is used to detect anomalous use patterns on a closely monitored system. Its monitors are installed as a part of the operating system and reports back to a centralised system to calculate the statistics.

In order to make good use of NIDES, it is crucial to classify log messages into a number of classes to which the statistics are applied. This requires that either a pre-defined classification is created beforehand, or that the log messages are categorized on-the-fly. A pattern mining technique could be used to create the patterns on the fly.

The NIDES algorithm creates a pre-defined threshold for what is considered normal behaviour for a user or system. If a measurement exceeds the pre-defined threshold or if a new type of activity is discovered, a human is supposed to act upon the alarm to decide if a violation of the rules have occurred.

Even though the NIDES algorithm might not be directly usable in a setting where one should avoid pre-configuration, its base idea of creating a threshold for normal behaviour is useful. Statistically speaking, a stable system should deliver nearly the same type and number of messages every day, only adjusted for season variations.

## 2.7 Conclusion

The only algorithms with potential for use in configuration-less log analysis are Markov models and neural networks. The Markov models are way of describing how a world evolves from one state to another while the neural network could be used to either classify log messages or to aggregate log messages. These ideas will be elaborated in coming chapters.

Bayesian learning, Apriori, decision tree algorithms and NIDES have been ruled out from further analysis, either because of complexity, unsolvable issues or because they are not applicable to the problem at hand.

The following chapter will present some of the previous work performed by other researchers on the area of pattern mining, anomaly detection and log visualisation.

# Chapter 3

# Related work

Log analysis have been an area of interest for a number of years, but there are only a few papers describing how one can do it without specific knowledge about the content. Risto Vaarandi has performed some research on mining patterns from log files and how to correlate events. His work is described in this chapter. The same is the log visualisation tool created by Takada and Koike, MieLog.

Some commercial applications have emerged on the marked. Descriptions of commercial solutions are kept to a minimum since there is no way of telling how they do their analysis. This is due to their closed source nature and lack of published research papers.

## 3.1 Simple Event Correlator

The Simple Event Correlator (SEC) [16] is an event correlator based on predefined rules created by Risto Vaarandi. It is designed to be easily customisable, and usable for a large range of event correlation tasks. It can be used separately, or in conjunction with other applications. SEC is distributed under the terms of the GNU General Public License.

SEC works as follows. A file stream serves input events into SEC. SEC then applies user-specified shell commands to the input events, and generates output events.

To recognise and handle input events, SEC uses regular expressions. This is justified by the assumption that most system- and network administrators are familiar with the regular expression language. Additionally, it allows SEC to cope with various input event formats. Figure 3.1 illustrates the overall steps in SEC.

As mentioned above, SEC is based on predefined rules. These rules describe which actions to take when a specific event is recognised. The rules used with SEC are stored in text files. This eases the modification, or creation, of rules.

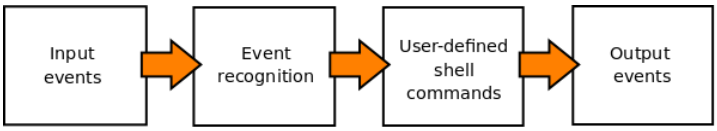



Figure 3.1: The different steps in SEC

A rule definition contains an *event matching condition*. In addition, most rule definitions contain a *list of actions*, and optionally a *boolean expression of contexts*.

Listing 3.2 shows an example rule, fetched from SEC's manual page[1]. The rule is activated once an input event matches the regular expression pattern. Imagine that the NFS server in question is called *Dilbert*. The correlation operation started by this rule calls *notify.sh "Dilbert is not responding"*. It then waits for the line *NFS server Dilbert ok* for exactly one hour. In the meantime it ignores all identical events. When the line *NFS server Dilbert ok* appears, the shell command *notify.sh "Dilbert ok"* is executed. If no such OK-message appears within the hour, the correlation operation simply terminates without performing any action.

```
type=Pair
ptype=RegExp
pattern=NFS server (\S+) not responding
desc=$1 is not responding
action=shellcmd notify.sh "%s"
ptype2=substr
pattern2=NFS server $1 ok
desc2=$1 OK
action2=shellcmd notify.sh "%s"
window=3600
```

Figure 3.2: SEC sample rule, fetched from SEC's man page

The contexts represent all that SEC has learned during the event correlation process. Each of these contexts has a lifetime. It can be either finite or infinite. Contexts can be used for dynamically activating or deactivating rules. Additionally, they can act as event stores, such that interesting events can be associated with a context, and at a later time be handed over for external processing.

When a rule matches an input event, one of two scenarios will take place. SEC either starts a new event correlation operation, or the event will be correlated by an already running event correlation operation. SEC stores operations that can not be immediately completed in its working memory.

[1]The manual page is distributed with the application and was last updated January 2009

This is the case if the operation involves correlation over a time window. Rules, contexts, and data about running child processes are also stored in working memory.

Figure 3.3 illustrates how a SEC ruleset operates when it recognises an input event.

### Advantages and disadvantages

SEC is a useful tool for correlating events, especially because it takes into consideration repeating or bursty events. Unfortunately, SEC uses hard coded regular expressions for recognising events. Consequently, new or unknown events will not be discovered. In addition, the regular expressions demand constant manual observation and updating to keep track of log message changes.

## 3.2 Simple Logfile Clustering Tool

Simple Logfile Clustering Tool (SLCT) [7] is a second tool developed by Risto Vaarandi. Unlike SEC, it does not correlate log events, but uses a clustering algorithm to mine line patterns from event logs.

Vaarandi claims that most words in log file, hereafter called *data set*, occur only a few times, and that a significant fraction of the words only occur once. Vaarandi also found that only a small fraction of the words were frequent, meaning they occur at least once per 1000 or 10000 lines. Additionally, he found strong correlations between frequent words.

Consider the following example log-lines, fetched from [7].

```
Router myrouter1 interface 192.168.13.1 down
Router myrouter2 interface 10.10.10.12 down
Router myrouter3 interface 192.168.22.5 down
```

When events like these occur many times, constant parts of strings will become frequent words. These frequent words occur together many times in the data set. In SLCT this set of frequent words will correspond to the line pattern *Router * interface * down*.

Following is a description of the clustering algorithm used in SLCT. The data set is assumed to consist of log file lines, which again consists of attributes. The attribute values are the words the log message is made up of. The data set's dimension, $n$, corresponds to the maximum number of words per log line. A subset $S$ of the data set where certain attributes have identical values is called a *region*. The set $\{(attribute_1 = value_1), \ldots, (attribute_k = value_k)\}$ $(1 \leq k \leq n)$ is called the set of *fixed attributes* of region $S$. If $k = 1$ then the region is called a *1-region* and consists of only

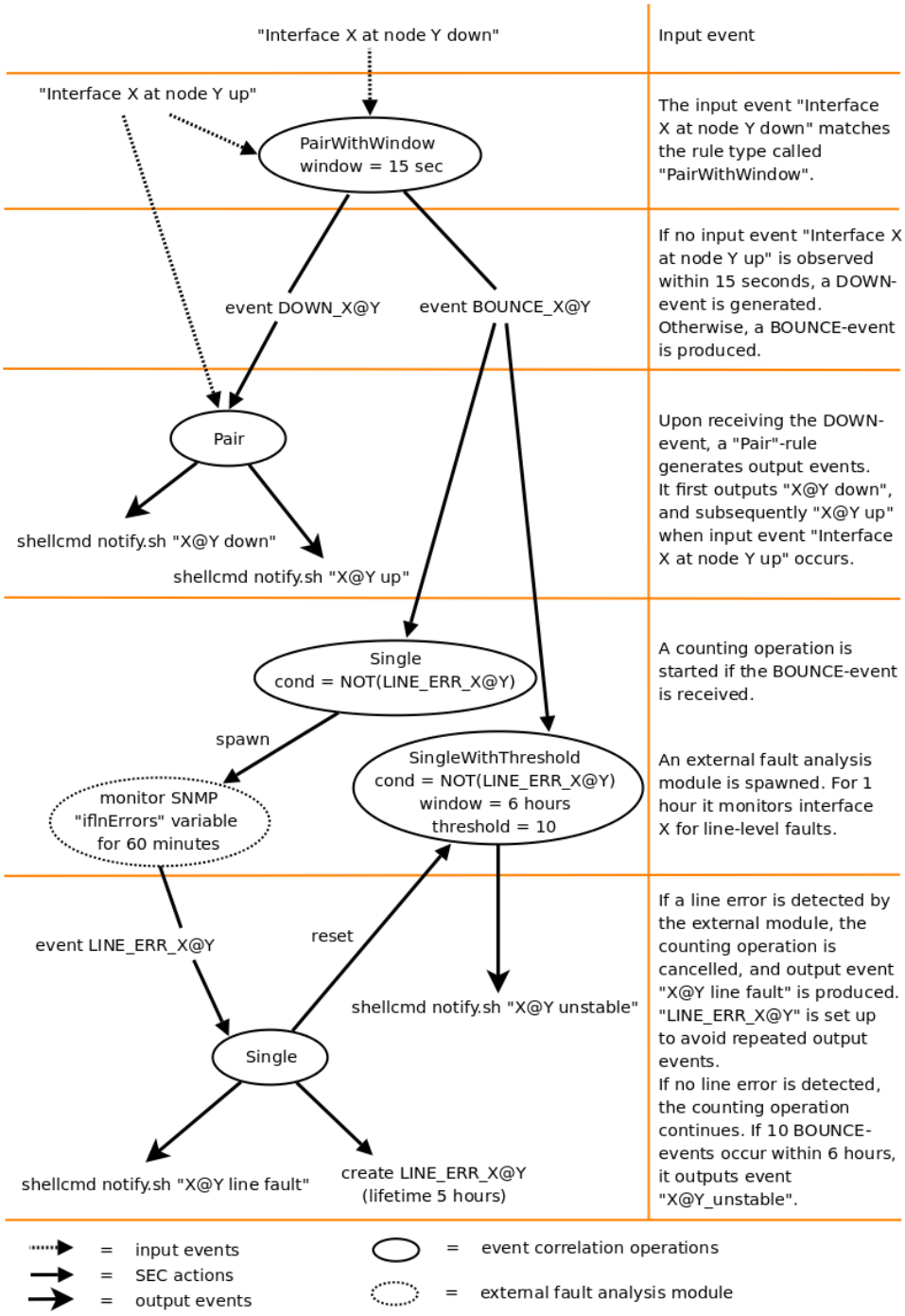



Figure 3.3: A SEC ruleset, adapted from [16]

one fixed attribute. The example log lines described above are examples of a 3-region subset, with attribute-value pairs (1=Router), (3=interface), and (5=down).

Figure 3.4 illustrates the three steps made by the SLCT algorithm. First, it iterates over the data set to identify all dense 1-regions, i.e. it discovers all frequent words in the data set. Whether a word is considered frequent or not depends on how many times the word occurs in the data set. A user-defined *support threshold* separates frequent from infrequent words. This step is identical to the first step of the Apriori algorithm described in Section 2.2.

Second, when all frequent words are found, the algorithm iterates over the data set one more time to build cluster candidates. A candidate table is used to store the cluster candidates as they are discovered. Line by line the algorithm checks whether the line contains one or more frequent words. If that is the case, then a cluster candidate is formed. If the cluster candidate already exists in the cluster table, its support value is incremented. Otherwise the candidate is created and its support value set to 1. In both cases, the line is assigned to the cluster candidate. Imagine that (1,'Router'), (3,'interface'), and (5,'down') have been found to be dense 1-region fixed attributes. Then the example log lines described above will form a 3-region cluster candidate with fixed attributes {(1,'Router'),(3,'interface'),(5,'down')}.

The last step of the algorithm iterates through the cluster candidates in the candidate table and discards all candidates with support value less than the support threshold, $S$. The rest are reported as clusters. Each cluster corresponds to a certain line pattern. As an example, the cluster {(1,'Router'),(3,'interface'),(5,'down')} corresponds to the line pattern *Router * interface * down*, as described above.

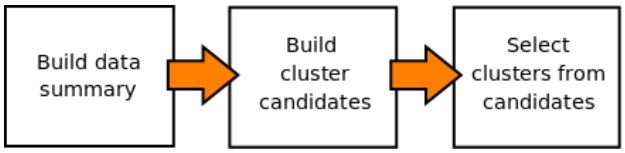



Figure 3.4: The various steps in SLCT

### Advantages and disadvantages

SLCT only mines frequent patterns and ignores infrequent ones. It could be used for anomaly detection in sense of statistics, but are useless for fault-detections where faults occur rarely. This is because it ignores infrequent patterns.

Performance-wise, it stores all data structures in memory. This may be problematic if the data sets grow large, in example when the number of unique words grows large. In terms of memory costs, the most expensive part of the algorithm is the first step when the data summary is built.

To solve the issue with memory usage, Risto Vaarandi suggest pre-processing the data using a word summary vector, to estimate which words need not be counted. This procedure is further described in Section 3.3 as *item summary vector* (ISV) in the Loghound tool. The LogHound tool is a tool that tries to solve some of the problems with SLCT.

## 3.3 LogHound

LogHound [17] is another log mining tool developed by Risto Vaarandi. Its goal is to mine frequent patterns from event logs, like SLCT. LogHound makes use of a breadth-first algorithm and is based on the Apriori [5], FP-growth [18], and Eclat [19] algorithms. It seeks to improve speed and memory-usage issues present in those algorithms.

Loghound works as follows. It takes as input a database $D$, consisting of transactions. Each transaction $T$ consists of a transaction identifier, $tid$, and a set of items, $Y$. An optional iteration over the database to filter out irrelevant items can be performed if the database is large. The algorithm performs a new iteration over the database and detects all items with a frequency larger than $c$. $c$ is a user defined support threshold. Another iteration over the database calculates dependencies between the frequent items, and detects itemsets with frequencies larger than $c$. Using the set of frequent items, each item's dependency prefix is found. A final iteration over the database is performed. For each frequent itemset found, the itemset is stored either in a cache tree or in an out-of-cache file, depending on whether its frequency is smaller or larger than the support threshold. Further, a prefix tree, called a *trie*, is created based on the itemsets stored in the cache tree and the out-of-cache file. The trie's size is reduced using a trie reduction technique. Figure 3.5 illustrates the steps performed by LogHound.

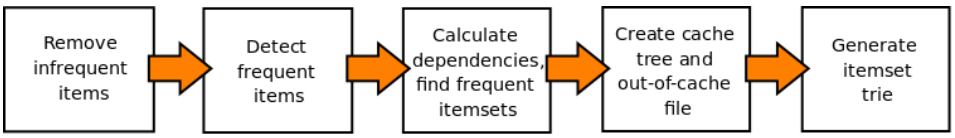



Figure 3.5: The various steps in LogHound

### Detailed description

Following is a more detailed description of the algorithm used in LogHound.

The algorithm first detects frequent items by counting the number of times each unique item occurs in the database. The number of unique items in a database can be very large. Consequently, the entire set of items and their corresponding occurrence-count may be too large to be store in main memory. Vaarandi [17] suggests elimination of infrequent items before counting.

This is done by building an *item summary vector* (ISV), consisting of $m$ counters, all initialised to zero. A fast hashing function is applied to each item in the data set, returning an integer number between 0 and $(m-1)$. Each time an integer $i$ is calculated, the $i$th counter in the ISV is incremented. When the construction of the ISV has completed, the algorithm starts counting the items in the log, ignoring those items in the ISV with a count less than a predefined support threshold.

To further improve memory usage, Vaarandi proposes to store the most frequently used transactions in a *cache tree*. The cache tree is stored in main memory. It contains all sets of frequent items that correspond to $c$ or more transactions, where $c$ is defined by the user.

To create the cache tree, the algorithm has to detect all sets of frequent items which correspond to at least $c$ transactions. To do so, another summary vector is created. The *transaction summary vector* (TSV) is constructed in a similar manner as the ISV. For each transaction the set $X = Y \cap F$ is found. $Y$ is the set of items in the current transaction, while $F$ is the set of all frequent items found in the previous step. $X$ is then hashed to an integer value, and the corresponding counter in the TSV is incremented.

A fourth pass over the data calculates the itemset $X$'s hash value, and looks up the corresponding count in the TSV. If the count is less than the support threshold $c$, $X$ is saved to an out-of-cache file as a separate record. Otherwise $X$ is saved into the cache tree. If $node(X)$ already exists in the cache tree, its counter is increased. Otherwise, the node is created and the counter is set to 1.

To reduce the size of the itemset trie, the algorithm develop only the trie branches that contain unique information. Following are a few definitions. Note that in a set X = $\{x_1,\ldots,x_k\}$, it is assumed that $x_1 < x_k$, as in Apriori [5].

$F = \{f_1, \ldots, f_n\}$, the set of all frequent items

$cover(X) = \{tid | (tid, Y) \in D, X \subseteq Y\}$, cover of itemset $X$

$dep(f_i) = \{f_j | f_i \neq f_j, cover(\{f_i\}) \subseteq cover(\{f_j\})\}$, dependency set of $f_i$

$pr(f_i) = \{f_j | f_j \in dep(f_i), f_j < f_i\}$, dependency prefix of $f_i$

Figure 3.6 illustrates a simple database and its corresponding trie. Each path from the root to a non-root node represents a frequent itemset. Each edge is labeled with a frequent item, and each node contains a counter for that itemset. The support threshold is 2.

The technique used for reducing the trie size is to create a node in the trie only when the itemset contains its dependency prefix.

When constructing the trie, the algorithm first creates the root node, detects frequent items and finds their dependency sets. The algorithm terminates if no frequent items are found. Otherwise, it creates nodes for frequent

| tid | Itemset |
|---|---|
| 1 | abcde |
| 2 | abc |
| 3 | bcd |
| 4 | abc |
| 5 | ab |

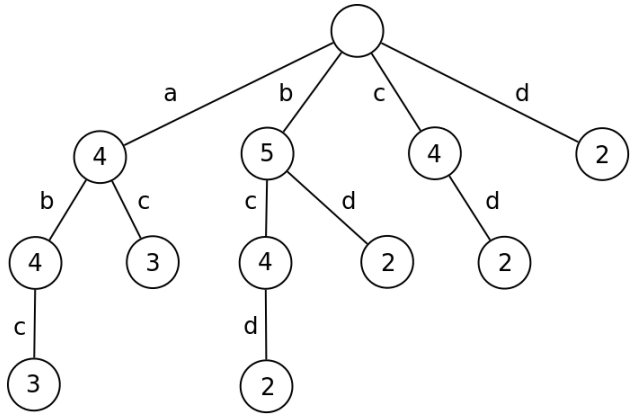



Figure 3.6: Example database and corresponding trie, adapted from [17]

items with empty dependency prefixes, and attach them to the root node. The algorithm proceeds by building the trie layer by layer, according to a procedure called *ProcItemset*, described in [17]. Each non-root node in the resulting trie represents a frequent itmeset, which contains its dependency prefix. All frequent itmesets can be derived from the nodes in the trie.

Figure 3.7 shows the same database as in Figure 3.6. The figure shows the set $F$ of all frequent items in the database, and calculates the dependency sets and dependency prefixes. The complete reduced trie is illustrated to the right.

| tid | Itemset |
|---|---|
| 1 | abcde |
| 2 | abc |
| 3 | bcd |
| 4 | abc |
| 5 | ab |

$F = \{a, b, c, d\}$

**Dependency sets:**

$dep(d) = \{b, c\}$
$dep(c) = dep(a) = \{b\}$
$dep(b) = \emptyset$

**Dependency prefixes:**

$pr(b) = \emptyset$
$pr(c) = pr(a) = \{b\}$
$pr(d) = \{b, c\}$

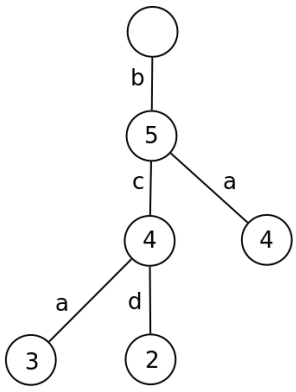



Figure 3.7: Example database and reduced trie, adapted from [17]

## Advantages and disadvantages

The LogHound algorithm is well suited for large databases, as it does not assume that all stored data structures will fit in main memory. LogHound attempts to discover all frequent patterns in a log, and discards all non-frequent items and itemsets. As a result, LogHound is useful when one needs good statistics of normal or frequent behaviour. It can be used to detect

significant variance in the number of specific transactions. Both a significant increase of a specific transaction, or the absence of such transactions may be an indicator of a problem or failure. With respect to infrequent errors, which may be of great interest, LogHound is useless. Unless such error-transactions become bursty, LogHound discards all infrequent items and itemsets, and thus can not discover such transactions.

## 3.4 MieLog

MieLog is an interactive, visual log browser created by Tetsuji Takada and Hideki Koike [20]. According to Takada and Koike [20], "MieLog is just a log browser, not an automated log inspection tool". Consequently, it is still necessary with a human to read through the logs and decide which parts requires additional examination.

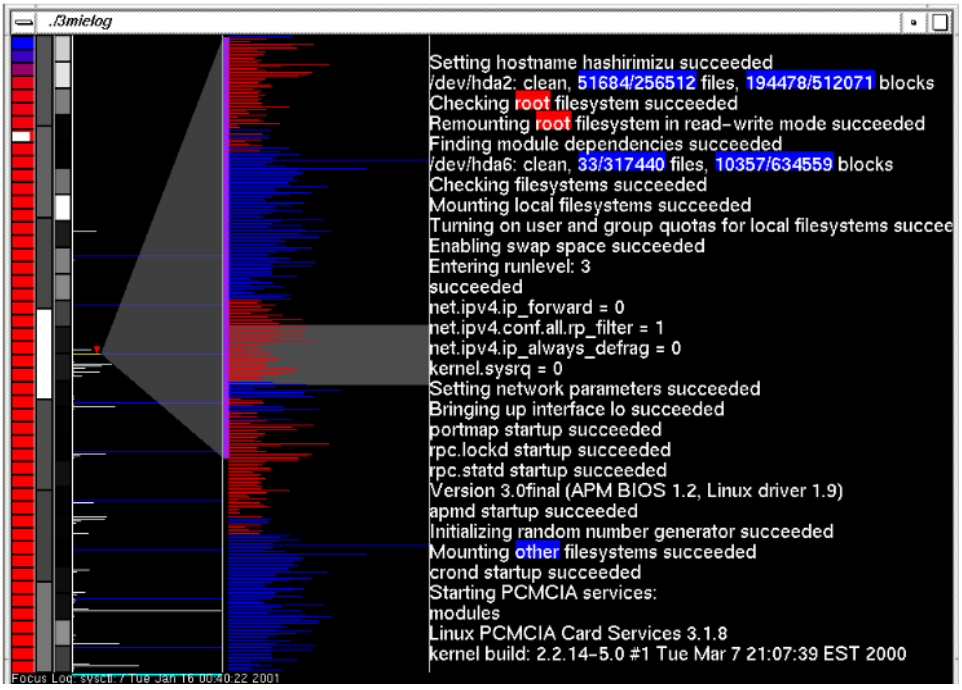



Figure 3.8: Screenshot from MieLog from [21] illustrating how colours are used to highlight unusual messages.

MieLog combines information visualisation and statistical analysis to aid administrators with identifying unusual log messages.

To find the most interesting parts of the system logs, MieLog first converts various log files into a "Generalized Log Format (GLF)" This is done to be able to display multiple log files and formats in a uniform way. After the conversion is performed, MieLog performs statistical analysis to extract frequency information regarding time, tags and messages. In addition, it

also has the ability to use pre-defined keywords or phrases

By combining GLF-formatted log messages, frequency information and pre-defined keywords, MieLog is capable of visualising log files. The visualisation assists an administrator with no prior knowledge or experience about the log files to identify anomalies. Figure 3.8 illustrates how MieLog is using colours to highlight unusual log messages.

## 3.5 Examples of commercial solutions

LogRhythm[2] market itself as a *comprehensive, integrated log management, log analysis and event management solution capable of supporting logs from virtually any log source* [22]. Its architecture and key functionality is illustrated in Figure 3.9.

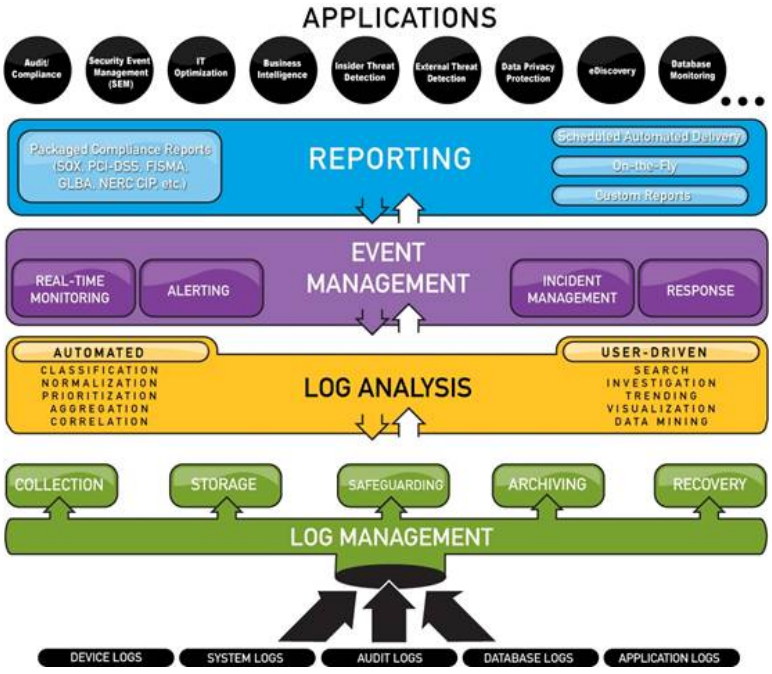



Figure 3.9: LogRhythm architecture as illustrated in [22].

Splunk[3] is another commercial solution targeted at solving log management and analysis problems. It is capable of indexing logs so that it is possible to search, send out automatic alerts and reports, and share logs between multiple administrators. Its main focus is log aggregation and search.

Common for both these commercial solutions is that they do not disclose how they perform their analysis. The administrators feed the product with system logs and get some kind of output from the product. Both products

[2]See `http://www.logrhythm.com` for more information
[3]See `http://www.splunk.com` for more information about the product

use graphs as a way of illustrating the current status, making it easy to spot abnormalities.

The lack of information about how the commercial solutions do their log analysis makes them unsuitable for further analysis in this thesis.

## 3.6 Conclusion

The work previously done has primarily focused on mining patterns from log files. Both the Simple Logfile Clustering Tool (SLCT) and LogHound are using statistics to find patterns and remove variables, but they both have their issues. The biggest issue with SLCT is that it is memory intensive since it stores all data structures in memory.

LogHound tries to solve the memory issue in SLCT by utilizing an *item summary vector*. The drawback is that it have problems finding infrequent errors, which is bad in a setting where it will be used to find anomalies. Due to these issues, these tools will not be an area of focus further on.

The Simple Event Correlator (SEC) is using regular expressions to correlate events in log files. As the regular expressions have to written beforehand, the solution will not be investigated any further.

The commercial solutions mentioned, LogRhythm and Splunk, does not disclose how they perform their anomaly detection and statistical calculations. Given their current state, it is sound to believe that they utilise regular expressions in one way or another to remove noise in logs.

The only idea that will be revisited in later chapters is the colouring idea from MieLog. Instead of being an anomaly detector, MieLog focuses on making it easier for the reader to find areas of interests by utilising statistics and colouring.

In the following chapters, ideas emerged from the background research described in Chapters 2 and 3 will be presented.

# Part II

# Ideas and results

# Chapter 4

# Ideas not pursued

The literature review and walk through of similar solutions generated a lot of ideas, some more volatile than others. To solve the task, a quick evaluation of the ideas generated had to be performed and some ideas had to be abandoned.

This chapter describes some of the ideas that were found unsuitable for further development. Both ideas for how to discover anomalies and presentation of these anomalies are briefly presented.

When reading this chapter, keep in mind that these ideas have been written off early in the process. Some open questions and ways of further develop the ideas might exists.

## 4.1 Use SpamAassassin to detect anomalies

SpamAssassin is commonly known as a spam detection application. It uses a wide range of small and simple tests to compute a value stating the probability for a message being spam.

The current version of SpamAssassin, 3.2.x, is built as a neural network trained with error back propagation. It performs a wide range of tests on a single email to ensure that the number of false positives and negatives are minimized. The list includes tests to see if an email originates from a known open relay, has ROT13 encoded email addresses in itand if it contains known bad words. The complete list of tests is available from the applications web page[1].

SpamAssassin combines its own tests with other open source projects such as Razor[2] and Pyzor[3]. Razor and Pyzor are distributed, collaborative spam detection and filtering network that relies on user contribution and feedback

[1] `http://spamassassin.apache.org/tests_3_2_x.html`
[2] `http://razor.sf.net`
[3] `http://pyzor.sf.net`

to filter known spam.

The use of SpamAssassin is interesting because of its use of a wide range of techniques to classify spam. Neural networks, Bayesian learning and user feedback are already implemented in SpamAssassin. The idea is to take a single log message and send it through SpamAssassin to score it. The score should say how common the log message is compared to the other log messages that SpamAssassin have seen.

One problem that might arise is that a single log line contains too little information for SpamAssassin to give any valuable output. It might be necessary to feed it with clusters of messages, for example the messages from last minute from a specific application.

Another problem is that SpamAssassin uses header checks to detect spam. Headers are either added to email messages when they are sent or when they are handled at email servers. These headers will necessary be the same for all the log messages since they originates from the same server. This makes all header checks useless. The lack of header checks will drastically influence SpamAssassin's ability to detect anomalities as the number of variables is reduced significantly.

## 4.2 Neural networks

Neural networks, described in Section 2.5, are good at utilising a large amount of input values and give an output representing all the input values. In theory, it is designed to take lessons from its input values to improve its output.

SpamAssassin, described in Section 4.1, utilize neural networks to decide whether or not an email message is spam. It does so by accumulating scores from a wide range of tests. The same idea could be applied to anomaly detection.

Although the theory behind the single neuron is simple, implementation of neural networks is a complex task requiring a significant amount of knowledge and previous experience with them. To utilise neural networks in the same way as SpamAssassin does, it is necessary to create a wide range of simple tests that can be applied to the log files.

The following tests could be used as an input to the neural network:

- Message thresholds - whether or not a message pattern occurs within a pre-calculated number as described in Chapter 7
- Missing patterns
- Regularity - Anomalies are detected on the basis of message regularity. Some messages occur on a pre-defined schedule, as described in Chapter 8.
- Past history - if a message has been reported to the administrator on

a number of occasions, it might not be that important to report the same message again.
- Markov models - as described in Chapter 9, could be used to identify anomalies in the message flow from a single application or anomalies in the transition between processes.

The user feedback (see Section 4.3) is essential for how useful the neural network is in its classification of anomalies. By training the neural network with user feedback, it should be capable of delivering highly value information back to the user.

The output from the neural network could be used to decide the current state of the system or how the output should be coloured, as described in the following Sections.

The biggest disadvantage of neural networks is that they are very complex by nature. The implementation requires a significant amount of resources and it is necessary, if used as suggested, to identify a wide range of tests to make the neural network useful. Currently, there is only five tests that could be used as inputs to the neural network. The amount of tests is too low to justify spending time on implementing a fully-fledged neural network.

## 4.3 User feedback

One way of defining the important lines are by asking the user to identify them based on filtered output from previous runs and other, automated tests. The administrator could manually inspect the output from the automatic analyser and identify a number of lines that are actual anomalies that need some kind of attention. When these lines are identified, the analyser should keep a close watch for messages of the same type.

The feedback system could also be used to track changes. If an administrator know that he has corrected an error based on reports from the logs, he could tell the analyser to watch out for the same messages that identified the anomaly in the first place. This would make it possible to check if an error has been resolved without having to manually monitor the log files for changes.

### 4.3.1 Tagging log lines

Log messages can be tagged in several ways: either by scoring them compared to all the other messages presented to the administrator, or by asking the administrator to select those messages that were important.

The score-based system is harder to use since it requires that the administrator makes decisions on how important an message is compared to all the other ones. On the other hand, when a reasonable number of messages are

scored, it would be possible to rank messages based on their score. This information could be used to discard less important messages and to display only high-priority messages, even though there are lot of messages that potentially could be important.

The other method of tagging log messages is a simplified version of the above. By reducing the span of the score to a binary value (important: yes or no), the administrators do not have to relate messages to each other. Instead, he can pick the messages he thinks are important and discard all the others.

### 4.3.2 Storing feedback data

Feedback on the messages should be stored together with the appropriate message to make it possible to trace possible misclassifications. Storing the feedback database on the same machine where the analyser is used is a simple solution and makes the feedback data available for use in later analysis'.

By centralising storage of the feedback data, multiple systems could benefit from the scoring done by a one administrator. The analyser could check the centralised database to see if a message is tagged or not. The information gathered from the centralised database should be used when deciding if it should discard the message as common or display it as an anomaly.

Special considerations must be taken when storing log messages in a centralised database. Within a single organisation, data can flow freely, but if the centralised database is placed on the Internet, the log messages have to be filtered. Filtering log messages for personal content would mean that important information is lost in the transformation.

Data poisoning is another problem with a centralised database. To be useful, content has to be relevant and consistent. A large number of irrelevant records will slow down lookups while inconsistent data will give users false positives.

While the centralised database is an interesting case, the issue of making the data anonymous and verifying it is outside the scope of this thesis. It will not be investigated any further.

### 4.3.3 Assessment

User feedback is a way of introducing human decision making into the automatic log analysis. A human touch should increase the quality of the output so that it is more precise.

The drawback is that without a good, automatic analyser that could provide the user with a few items to give feedback on, the ideas is merely another way of writing configuration.

A centralised database could reduce the workload for each administrator, but it introduces problems with consistency, privacy and security.

Given that a good, automatic analyser does not currently exists, the idea is put on hold until the analyser is created.

## 4.4 Filter repositories

Given that the task of making an analyser that both is configuration-less and provides valuable output is too hard, a fallback is a good alternative.

One solution is to remove most of the configuration for the end user. There is a significant overlap in the processes running on computer systems. If only a few users write some specific patterns to match log messages from a specific application, these patterns could be saved in a repository. People wanting to use these filters could synchronise their local filter sets with a centralised repository on a regular basis. By adding some simple logic, the filter updater could be made smart enough to only download the filters that apply to that specific system, either by looking at the package manager operating on the local system or by looking at the current system logs. This solves the update problem that existing regular expression based analysers have. Lire, LogWatch and LogCheck (see Section 1.2) are all distributed with predefined regular expressions that are only updated when a new software release is installed.

This approach have the same challenges as the user feedback (Section 4.3) when it comes to security and privacy. The patterns have to written in such a way that all variables are identified and removed. One way to do this is to employ a helper application that identifies patterns based on log input. The patterns could be of significant help to the users that write patterns.

Another concern is that malicious users could create patterns that removes information that should appear in the log files. This could for example be messages from exploits, bot nets and such. One way to avoid such problems is to employ some sort of feedback or user review before the patterns are added to the repository that is available for users.

One interesting feature with this idea is that all the filtering is performed by patterns verified by humans. The patterns may be created with the help of generated patterns, but at least one user have verified and submitted the pattern. By making all the patterns available for download, only a few users have to write patterns for a single application. If the patterns are updated regularly, there is less of a chance for the patterns to be outdated.

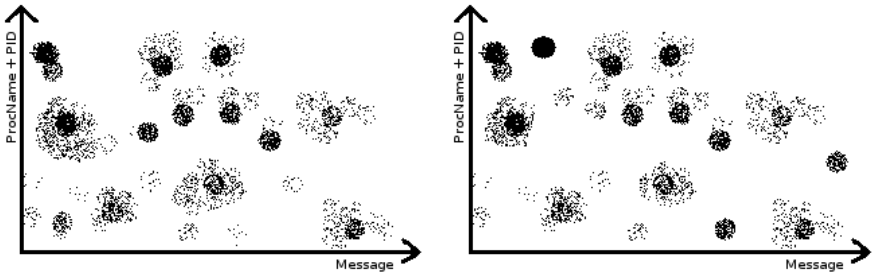
The main reason for not implementing this idea is that is not a solution to the original problem, it merely circumvents it by distributing the job of creating filters to a number of users. Also, it is necessary to create a distributed architecture to make a client-server model in order to get the idea into

production. Given these two reasons, it was chosen not to implement this idea.

## 4.5 Graph plotting

The graph plotting idea can be described as follows:

Given a large enough sample space, it should be possible to plot each log message in an image so that one pixel corresponds to one message. The message should be transformed to a numerical value by the use of a hash algorithm. Given that the idea works, the log messages should form clusters of pixels that deviates from the background. These clusters are usual in the logs and should be considered part of the stable state. Clusters that appears in the image for one period but not in another should be considered an anomaly.




(a) (b)

Figure 4.1: Mock-up plots of log data from two different periods.

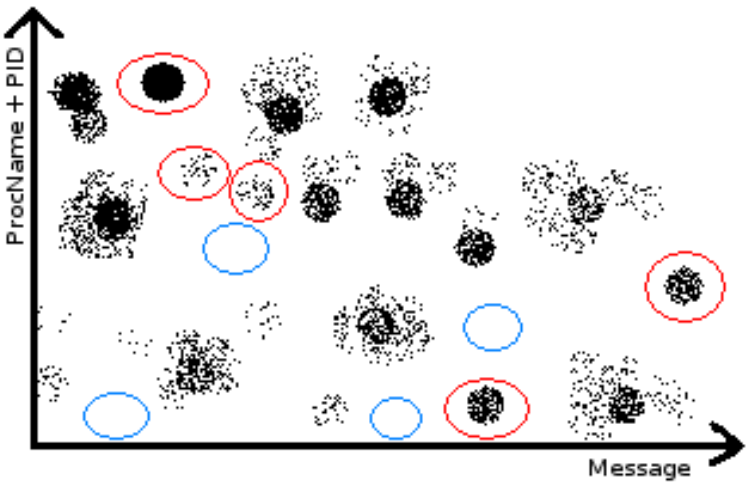



Figure 4.2: A mock-up of the differences between the graphs in Figure 4.1. New clusters are encircled in red, missing clusters are encircled in blue.

A hash algorithm has to be chosen so that the log lines could be transformed

from a string to a numerical value that can be plotted in a graph. The hash algorithm must be designed in such a way that small changes in the input does only makes small changes to the output. Collisions in the sample space is not a concern. Rather, collisions are good since they mean that the log messages are either similar or quite similar.

On the other hand, the hash algorithm has to be able to differentiate between different types of log messages so that a large enough sample space is utilised. If the hash algorithm is poorly designed, only a few, large clusters will appear in the image, making it impossible to spot differences between images.

Figure 4.1 illustrates how two log periods could be plotted in a graph. By comparing them against each other, the differences emphasized in Figure 4.5 appears. Figure 4.5 shows four missing clusters and five new ones. These are message clusters that should be selected for further analysis.

The graph plotting idea can be further developed into a visualisation tool by making the graph interactive. By either holding the mouse pointer over a cluster or by selecting a region, the log messages belonging to that cluster or region should be displayed. It should also be possible to zoom in on separate regions or clusters.

Lack of a good pattern mining technique is the main reason for not developing this idea any further.

## 4.6 Bird's twittering

The bird's twittering idea is not a way of extracting interesting items from log files. It is a way of presenting the current state of the system to the users by the use of sound instead of text output.

Humans are very good at detecting anomalies from background noise. According to Covey, Malmierca and Perez-Gonzales, there are neurons in the brain that are stimulated if sound appears [23]. These neurons did not forward their stimuli if they were fed with sounds that were repeated more than four or five times per second. But, if there were new sounds that occurred in the pattern, they sent a signal to other neurons, signalling an anomaly in the background noise.

The idea is to present the current state of the system by utilizing bird's humming or another, non-intrusive background noise that are very common. When the system is considered to be in a stable state without errors, a series of speakers will output the background noise. If there are errors, uncommon sounds will be inserted in the background noise to signal the pattern.

To work effectively, the amount of unusual sounds must be kept to an absolute minimum. The threshold for signalling an error must be set to a level where only critical errors are reported. If the sound signalling errors are played too often, it is possible that the users perceive it as part of the

background noise.

The idea was left dead due to the lack of a working pattern mining technique and because it is one of the more peculiar solutions. It also requires that there are system operators located in area where the sounds are being played so they can act upon detected anomalies.

## 4.7 Colouring output

MieLog, described in Section 3.4, uses colours to emphasize sections of logs to make it easier to spot anomalies. The use of colours makes it easier for a human to separate important messages from the noise.

Syslog messages are tagged with facility and severity. If the output from the applications is trusted, severity could be one of the factors that decides what colour a message should get.

The statistical part of the analyser, as described in Chapter 7 provides valuable input to the colour coder. Deviations from a fairly established schedule are not easy to spot without some kind of emphasis. By giving the message that deviates from the usual pattern a colour, it would be easy for a human to spot the anomaly while reading through the most important messages of the day.

Another valuable input to the colour coder is the input that the user itself has provided through the user feedback, If the user has already said that the lines are important by giving the analyser feedback, it is critical to emphasis these lines in the visualisation tool.

## 4.8 Conclusion

The reasons for leaving these ideas out in the cold are as diverse as the ideas themselves. The SpamAssassin idea is abandoned because of the lack of enough variables. SpamAssassin is dependent upon a large number of variables to determine if a message is spam or not. By sending just one and one log message from the same host, the number of variables are reduced to an amount considered too low to be efficient.

In this chapter, the idea of using neural networks to score results was presented. This is the same way as SpamAssassin calculates its spam score. Although it is tempting to create such a network, it requires a lot of inputs to be useful. As of now, there is only a handful of inputs available which makes the idea too complex for implementation.

User feedback is a nice way of including users in the decision process, but it requires some automatic processing first. It also requires a framework capable of classifying and storing feedback on-line.

Instead of storing feedback, it is possible to create a repository with filters on-line which users could synchronize. The repository idea is one of the most promising ones because it solves some of the problems with maintenance of regular expressions. Creating a configuration-less or configuration-light log analyser is not only about removing the configuration altogether, it is about making it easier for the users. By having a repository, most of the users would never see the regular expressions.

Another issue illustrated in this chapter is log browsing. Reading through entire logs are a time consuming process where most of the information is redundant. By filtering out the redundant information or finding other ways of presenting the data, one could make the task much easier. Colours, sound and graph plotting are ideas that might be successful, given the right input.

Altogether, all these ideas are left out from further analysis. The ideas concerning log presentation is dependent upon input from a log analyser capable of classifying log messages, which are not available at the time of writing. In some degree, this also applies to the user feedback and filter repository ideas. These two ideas also requires a framework and a server side implementation which is found to be too time consuming to be implemented. Also, the filter repository and user feedback ideas does not solve the problem, they merely circumvent it.

That being said, the ideas presented in this chapter are not found completely useless. If a good pattern mining technique is developed, some of these ideas could be used to further enhance the log analysis and log browsing.

The focus is now shifted to ideas believed to be better for solving the specific problem of configuration-less log analysis.. Mining patterns from log files is next up since it is a prerequisite for a lot of other ideas.

# Chapter 5

# Pattern mining approaches

Before any further analysis of the data in a log file can be performed, it is necessary to learn to recognise loglines. This can be done by categorising, or clustering, of the different loglines present in the log file. It requires the ability to recognise known loglines, but also to discover new ones.

The contents of a logline is entirely dictated by the reporting application. Using regular expressions (see Section 1.2) to match messages from applications requires constant updating, insertion of new expressions to match new messages, and deletion of stale expressions. This requires much maintenance, which is both time consuming and expensive. For every new application added, regular expressions must be tailored to recognise messages reported by that specific application.

To avoid using regular expressions to match various loglines, new ways of recognizing logglines aare required. As became clear in Chapters 2 and 3, none of the existing methods for finding and clustering loglines return sufficiently accurate results. The Apriori algorithm is inefficient in terms of the number of calculations it performs, SLCT only mines frequent patterns, and the same is true for LogHound. A new, more precise way of mining logline patterns is needed.

The following approaches describe various ways of mining logline patterns from log files. The objective is to find patterns that can describe the various loglines appearing in a log file, without previous knowledge of the contents of the loglines' contents.

The approaches are evaluated in terms quality of the patterns they produce and their performance.. "Quality" is measured by evaluating how accurate the patterns are and how many variables are left in the formed patterns. To be useful, the patterns must be as accurate as possible with none or very few variables. High quality means that this is achieved. Low quality means that the patterns are inaccurate and/or contain too many variables.

All the approaches described in the following have a corresponding proof-of-concept implementation. These implementations are documented in Ap-

pendix C.

The output examples presented in this chapter were generalted by analysing the example log in Appendix B.3 using the proof-of-concept-code. The example contains a total of 15 messages.

Now, a few words about loglines, and extraction of the message part.

## 5.1 Extracting the message

Figure 5.1 illustrates two sample loglines, and the corresponding pattern formed by them. The term *logline* is applied to describe an actual logline present in the log file, but also denotes a group of loglines that naturally belong together.

A logline consists of $n$ items (or single words), and the length of the logline corresponds to the number of items it contains. $Keywords$ are the items in a logline that, with the aid of human judgement, form the static part of a logline. $Variables$, such as identifiers, usernames, and email addresses, are the items in a logline that may change each time an instance of the logline-group is observed.

```
dovecot: [ab123 mail.info] IMAP(john) Disconnected Logged out
dovecot: [ac234 mail.info] IMAP(jane) Disconnected Logged out

dovecot ID mail.info IMAP Disconnected Logged out
```

Figure 5.1: Sample pattern

In the example given in Figure 5.1, the identifiers ($ab123$ and $ac234$) and the usernames ($john$ and $jane$) are recognised as variables, and are pruned away. Keywords, like $dovecot$, $mail.info$, and $Disconnected$, constitute the items that stay unchanged within a group of similar loglines.

It is assumed that the format used in the logfile is known, making it possible to extract only the message-part of the logline. When mining for logline patterns, only the message-part of the logline is considered, while the priority and head is ignored. See Appendix A for more info on Syslog.

The message part of the logline is inspected to remove unwanted elements. Email addresses will normally not be part of any logline pattern. They are easily recognised, and to reduce the possibility of email addresses turning up in patterns, they are therefore removed when found.[1] The same is true

[1] One problem may occur when attempting to recognise email addresses. Often observed in system logs are phrases like *sender=john@example.com*. Since the equal sign is a valid character in email addresses, the entire phrase may be recognised as an email address. Since it is chiefly spammers who make use of equal signs in their email addresses, and these addresses usually contain a wide range of letters, numbers and special characters, an attempt to extract the *sender=* part from the email address is made.

for IPv6 addresses.

Further, many processes in a logfile keep the same process number for a long while. This can cause patterns to be formed including this process number. When the process number changes, a new pattern may be created, even if the two patterns represent the same logline group. Since we are interested in including timestamps into the patterns, number words are removed. With the number words, IPv4 addresses are removed as well.

For some reason, some application creators think it is a brilliant idea to separate each word in the message part with a comma, colon, semicolon or similar, instead of the more customary whitespace. This makes the entire line appear like one single word. To be able to remove variables from the message, it must be split. This is simply done by replacing any character that does not normally occur in a word, with a whitespace.

Be aware that such removing of particular characters may lead to unexpected word-division. It may cause otherwise similar loglines to be handled separately, since the word-division may lead to loglines of different lengths (in terms of number of items). Still, this is not likely to happen frequently, as logline items rarely contain special characters like # or $ , unless it is spam. Additionally, items containing such a character is likely to be a variable and not a keyword.

This preprocessing of logline messages is used in all the approaches described below.

In the proof-of-concept code, a variable called *common_items_factor* is used to determine if two lines are similar enough. The default value of this variable is 0.6. This is the same value as the Python module *difflib*[2] uses as a cutoff value in its *get_close_match()* method. Two log lines most score at least the value of the *common_items_factor* when compared to be considered similar.

## 5.2 Item occurrence frequencies

Inspired by the Apriori algorithm, the first attempt to form patterns is based on counting the occurrences of unique items in a log file. When iterating over the logfile the first time, each unique item in the file is stored together with its occurrence count. The result is a collection of 1-itemsets.

A user defined threshold multiplied with the number of loglines in the log file produces a limit value. If a 1-itemset has a count less than this limit, it is pruned away. The result is a collection of frequent 1-itemsets.

During the second iteration over the logfile, the frequent 1-itemsets are used to decide which items in a logline should be included in the logline pattern, and which should not. If the item is not to be found amongst the remaining

[2] `http://docs.python.org/library/difflib.html`

1-itemsets, it is ignored. The remaining items in the logline forms a pattern. If the number of items in the pattern does not exceed a threshold, the line is added in its entirety. This is because very short patterns will match many loglines, the result being that dissimilar loglines are incorrectly clustered together and counted as one. If the pattern does not already exist, it is saved together with a count of 1, otherwise the existing pattern's count is increased by 1.

The result after the second iteration is a collection of n-itemsets, sorted by length. Each line in the logfile is represented in terms of one pattern, and the pattern's count describes how frequent that particular logline is.

A simple clustering algorithm compares all itemsets of the same length, and attempts to create a common pattern from similar loglines. It compares two and two itemsets item for item, and discards loglines producing a common fraction of less than 0.6. 60% is chosen because it is a commonly used factor for determining likeness.

The flow chart in Figure 5.2 illustrates the process performed on an itemset to find similar itemsets it can be merged with. Only itemsets not already merged are considered. For each itemset of equal length as the current itemset, the fraction of similar items between the two is calculated. If the fraction is less than 60% the algorithm discards the itemset being evaluated and fetches the next.

If the common fraction is equal to or greater than 60%, but a better match was already found, the itemset is discarded and the next one fetched.

When an itemset is found that produces a higher common fraction than what has been found hitherto, the list of potential candidates found so far is emptied. The new found itemset is added as a candidate, and the algorithm moves on to the next itemset.

If the itemset evaluated produces a common fraction equal to the best common fraction, the itemset is simply added to the candidate list. When no more itemsets of correct length is found, the current itemset is merged with all the itemsets in the candidate list.

The merge process is very simple. The current itemset form the first draft of the pattern. For each itemset in the candidate list the pattern is slowly adapted trough comparing the pattern, item for item, with the candidate itemset, and replacing mismatches with a $\star$ symbol. In this way the final pattern will represent the common items between the current itemset and all the itemsets in the candidate list.

### Issues and results

Using the frequency of unique items as basis when creating patterns is problematic. Bursty loglines, and loglines that naturally follow each other and report the same identifiers, usernames etc. can cause such variables to be-

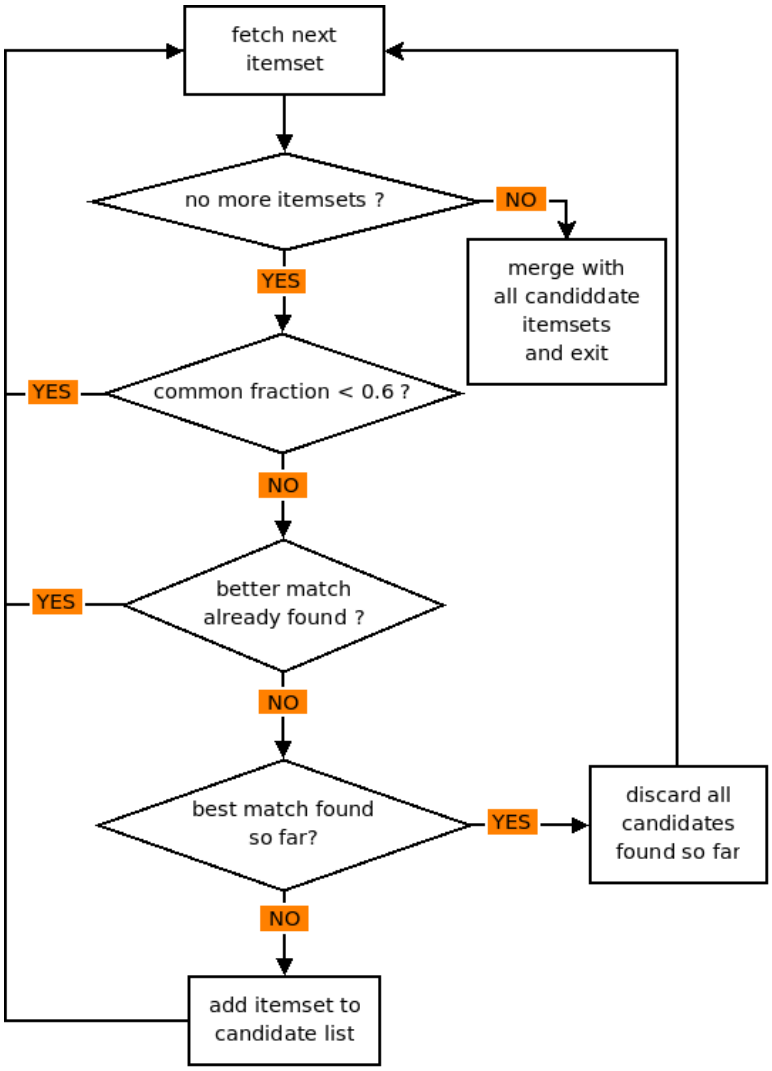



Figure 5.2: Determining likeness between itemsets

come frequent 1-itemsets.

By including the item's position on the logline when counting unique items, many of the 1-itemsets that were incorrectly considered frequent are now correctly pruned. However, it does not eliminate them all. Usernames, identifiers and such, may occur at the same position, even in dissimilar loglines, and in messages from dissimilar processes.

It is reasonable to assume that similar loglines are equally long, in terms of the number of items it contains. Attempting to include the logline length as well, causes more problems than it solves. As an example, process names are usually frequent enough to be considered as frequent 1-itemset. When the logline length is included, the process name may actually be pruned, since processes produce messages of various length. The contents of the message

may be considered more frequent than the process, since similar messages may be reported by other processes as well.

In the prototype created to illustrate this approach, the item's position is included. As a result, an item may be presented as several 1-itemsets if it occurs at more than on line position in the log file.

Creating patterns based on item occurrence frequencies is not a promising approach. Firstly, it is difficult to set the correct threshold at which to prune infrequent 1-itemsets. Secondly, it is hard to assure that unwanted variables are not amongst the frequent 1-itemsets.

```
dovecot ID mail.info IMAP Connection closed 2
dovecot ID mail.info IMAP mbox data INBOX INDEX  1
dovecot ID mail.info POP3 Effective uid gid 2
dovecot ID mail.info POP3 mbox data INBOX INDEX  2
dovecot ID mail.info auth default lookup 3
dovecot ID mail.info auth default passwd bob lookup 1
mimedefang.pl Time 2
mimedefang.pl filter_sender Whitelisted at 2
```

Figure 5.3: Output from the item occurrence frequencies algorithm

Figure 5.3 gives an example of the output from the occurrence frequencies algorithm. The algorithm has identified eight distinct patterns from the 15 lines provided. Because the example log is very small, a threshold of 0.15 was used. The default value is 0.001 which is suited for logfiles of around 200MiB.

The output in Figure 5.3 illustrates the problem of frequent variables. With a threshold of 0.20 and 15 loglines, every variable with a count equal to or higher than $0.20 * 15 = \lfloor 2.25 \rfloor = 2$ is considered frequent.

The patterns in Figure 5.3 are generally of high quality. Only items with counts less than 2 are pruned away, and since there are two or three occurrences of almost every logline type in the example log, every keyword occurs twice. An exception is the logline containing *IMAP, bob, mbox, data, INBOX and INDEX*. This logline is handled nicely because all the variables it contains occur only once in the logfile at those positions. For example, the username *bob* occurs a total of 4 times, but only once at position 4.

The reason why the username *bob* occurs in the sixth pattern is not caused by it being considered a frequent 1-itemset, as described above. It is caused by the algorithm that compares and clusters similar loglines. Figure 5.4 illustrates how this situation occurred. The clustering algorithm starts by comparing the itemset [*dovecot ID mail.info auth default shadow bill lookup*] to all other itemsets of equal length. It first finds an almost identical logline, where only the username differs. As a result, the common fraction between the two itemsets is 0.875. The next logline found also produce the same common fraction, because the username, a variable, is identical to the username in the itemset we are currently looking at. Consequently one of the

*passwd lookup* itemset is merged with the *shadow lookup* itemset. When the last *passwd lookup* itemset is found, it produces a common fraction of 0.75, and is discarded as a match, since better matches are already found.

```
Looking at itemset:
['dovecot', 'ID', 'mail.info', 'auth', 'default', 'shadow', 'bill',
    'lookup']
    Best match found hitherto:
        dovecot ID mail.info auth default shadow bob lookup, with
            0.875.
    Candidate found:
        dovecot ID mail.info auth default passwd bill lookup, with
            0.875
    Non-candidate:
        dovecot ID mail.info auth default passwd bob lookup, with 0.75
```

Figure 5.4: Output illustrating an issue with item occurrence frequencies

## 5.3 Subsets

Another approach, also inspired by the Apriori algorithm, make use of item subsets to mine patterns. For each logline in the log, all possible subsets of the items present in the line is found. If the subset does not already exist, it is stored and its frequency count is set to 1. Otherwise, the subset's count is increased by one.

When finished, all subset counts are normalised. If the combination {process name, keyword} has occurred 100 times, and the set {process name} has occurred 1000 times, the normalised count is 0.1.

The result is a large collection of subsets and their normalised frequencies. Rare combinations such as an identifier and a process name will have a low normalised frequency, whereas two keywords that usually appear together on a logline will have a high frequency.

A second iteration over the log is needed to analyse each logline. The subsets found in the previous step are used to discover which items in a logline does not occur as frequently as the rest of the items on the same line.

### Issues and results

The resulting patterns mined using this approach produce high quality patterns. Unfortunately, calculating all subsets of every logline in the log produce vast numbers of subsets, and the process of finding all the subsets is itself slow.

Imagine the short message "user john logged out". This line produces the following subsets:

| | | |
|---|---|---|
| (out) | (john, out) | (user, john, out) |
| (user) | (user, john) | (john, logged, out) |
| (john) | (logged, out) | (user, logged, out) |
| (logged) | (john, logged) | (user, john, logged) |
| (user, out) | (user, logged) | (user, john, logged, out) |

In total, the line produces 15 subsets. If the line had contained one more word, the number of subsets would increase to 31, and further to 63 for a line with 6 words. Keeping in mind that most loglines are usually longer than 6 words, it is easy to imagine that the number of subsets produced will be very high.

Additionally, the subset calculation must be performed twice to produce the resulting patterns. The approach is too slow to work in practice, unless measures are taken to handle the vast memory usage. See Section 5.6 for details on resource utilisation. This approach will not be investigated further.

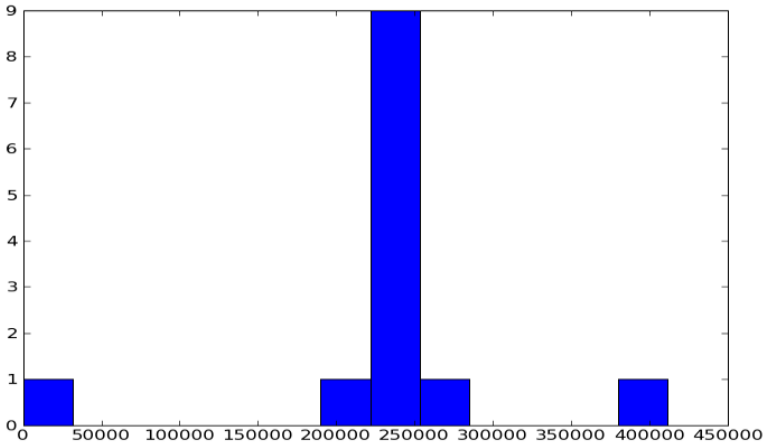
Output from the example log in Appendix B.3 contained 13 008 unique subsets. The output is not shown for obvious reasons. The subsets are generated similarly as the example subsets in Figure 5.3.

## 5.4 Histograms

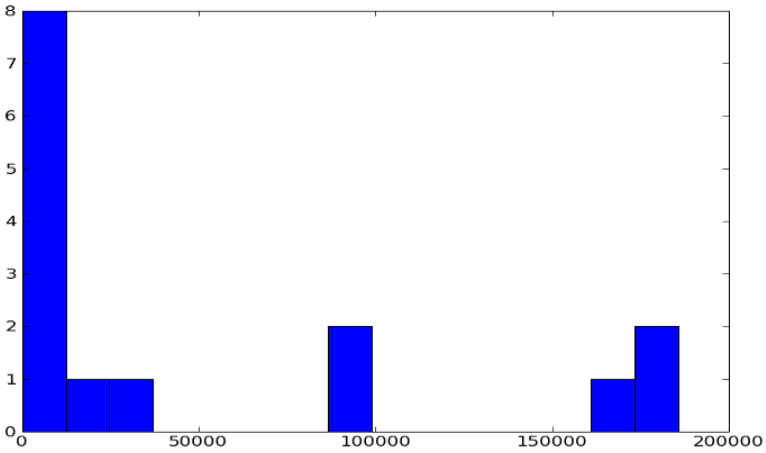
Even if the approach based on item frequencies was not too promising, another angle of attack was attempted. This approach also start by counting all unique 1-itemsets in the logfile. When iterating over the logfile the second time, it builds a histogram based on the logline item's counts. The idea is that the placement and the area covered by the histogram bar can illustrate whether the items covered by the bar should be pruned or not. If the bar to the far left in the histogram covers a small fraction of the total area in the histogram, the items covered by this bar are probably variables we would like to get rid of.

The approach is an attempt to use 1-itemset counts locally within a single logline, to look for obvious outliers. Outliers in this context only include items with significantly lower count than the rest.

Figure 5.5 illustrates the histograms of two different loglines, picked from a log file with roughly 900 thousand lines. The y-axis represents the number of 1-itemsets contained in a bar, while the x-axis represents the 1-itemset's frequency. In Figure 5.5a, the rightmost bar represents a single 1-itemset, and its frequency is in the range between approximately 375 thousand and 410 thousand. Figure 5.5a shows an obvious outlier (its leftmost bar), while Figure 5.5b has none.




(a)




(b)

Figure 5.5: Histogram examples of two different loglines

### Issues and results

The approach works well for loglines where the gap between the outliers and the rest of the items is significant, and where the frequency span for the remaining items is relatively narrow. Unfortunately, as illustrated in the two examples in Figure 5.5a and 5.5b, the frequency span is usually very large.

The largest problem is caused by the fact that process names usually have a very high frequency, while most frequent keywords are usually significantly less frequent. The keywords and what would otherwise be considered as outliers, are merged together in the same histogram bar. As a consequence, it becomes difficult to identify the outliers, since they no longer are considered outliers. The problem arises when the span in frequencies become large, and a group of keywords have relatively low frequencies compared to the rest. The alternative is either to delete keywords that may be requisite to create

```
dovecot ID mail.info IMAP Connection closed 1
dovecot ID mail.info IMAP john 1
dovecot ID mail.info IMAP mbox data INBOX INDEX 1
dovecot ID mail.info POP3 Effective uid gid 1
dovecot ID mail.info POP3 john 1
dovecot ID mail.info POP3 john mbox data INBOX INDEX 1
dovecot ID mail.info POP3 mbox data INBOX INDEX 1
dovecot ID mail.info auth default lookup 4
mimedefang.pl filter_sender Whitelisted at 2
mimedefang.pl n1JN00 Time 1
mimedefang.pl n1JN02 Time 1
```

Figure 5.6: Output from the histogram algorithm

a good pattern, or to risk including variables into the patterns.

None of the alternatives above results in quality patterns. The patterns either become too rough to reflect the varieties of loglines, or they contain a high number of unwanted variables.

Figure 5.5b illustrates a situation where keywords and variables have been grouped together under the leftmost bar. In Figure 5.5a the leftmost bar actually contains a variable, but it is not easy to verify whether it is a variable, or simply a less frequent keyword. Both examples illustrate how the keywords' frequency varies, even within a single logfile. It makes it very difficult to identify variables based on their frequency. It all boils down to the same problems experienced in the first attempt to create patterns based on item occurrence frequencies.

Consequently, it is concluded that any approach based on item occurrence frequencies alone does not create promising results.

Figure 5.9 gives an example of the output from the primary sorting algorithm. The algorithm has identified 8 distinct patterns from the 15 lines provided. The results illustrates some of the problems discussed above. The username *john* appears several times in the generated patterns, because it has a relatively high count in the example log.

Figure 5.7 shows some of the output from the prototype that generated the results in Figure 5.9. The first example illustrates how the pattern *dovecot ID mail.info IMAP john* is created. First look at the frequencies listed. The header items *dovecot*, *ID*, and *mail.info* have a much higher count than the remaining items. The histogram, represented as a list, contains the same number of "bars" as there are items in the itemset. The histogram illustrates the distribution of frequencies within an itemset.

The histogram prototype's objective is to remove the leftmost bars, as long as the area covered by these bars do not exceed 20% of the total area covered by the itemset. The 20% limit is set to allow a deletion for each fifth item. This limit is set for testing purposes only. As will be shown shortly, a hard coded limit has its limitations.

A bar's area is calculated from the product of the frequency value and the number of times the frequency is observed in the itemset. In the first example in Figure 5.7 the frequency 11 is observed 3 times. The rightmost bar in the histogram thus covers an area of $11 * 3 = 33$. Obviously, the area covered by these three values constitute a great part of the total area covered by the itemset, which is 46 $((11*3)+(4+3)+(2*3))$. Consequently, all items with frequency 2 are deleted, since the area covered by these items, $2 * 3 = 6$, amount to less than 20% of the total area. Including the next bar as well would cause the area to exceed 20% of the total area. Thus, items $POP3$ and *john* are kept. This explains why the username *john* appears in the pattern. Similar examples would prove why it appears in other patterns as well.

The explanation for why the *mimedefang.pl,Time* patterns include variables is much simpler. 20% of 3 is less than zero. Thus, none of the items in such a short line can be deleted without deleting more than 20% or the total area.

```
['dovecot','ID', 'mail.info', 'POP3', 'john', 'Effective', 'uid',
    'gid']
for deletion:  [5, 6, 7]
frequencies: [11, 11, 11, 4, 3, 2, 2, 2]
histogram: [0 3 2 0 0 0 0 3]
result: ['dovecot', 'ID', 'mail.info', 'POP3', 'john']
-----------------
['mimedefang.pl', 'n1JN00', 'Time']
for deletion:  []
frequencies: [4, 1, 2]
histogram: [1 1 1]
result: ['mimedefang.pl', 'n1JN00', 'Time']
```

Figure 5.7: Histogram examples for pattern creation

## 5.5 Primary sorting

The primary sorting approach attempts to avoid the problems that occur when making use of 1-itemset occurrence frequencies only. Instead of pruning away 1-itemsets with low frequencies, all but the most frequent 1-itemsets are removed. This decreases the probability of including variables amongst the 1-itemsets.

The primary sorting prototype make use of mechanisms from the item occurrence frequencies approach in Section 5.2. It uses its itemset generator to generate its 1-itemsets, and reimplements the process of calculating likeness between itemsets, as described in Figure 5.2.

After generating the 1-itemsets, the $frequent$ 1-itemsets are used to form primary patters. For each logline, a primary pattern is formed by the items that are present among the frequent 1-itemsets. If the pattern does not already exist, it is saved, otherwise it is ignored. The primary patterns are

very coarse, and their purpose is to act as "buckets", into which similar loglines will be sorted.

```
for logline in bucket:
    if logline already included in a pattern:
        continue
    for other_logline in bucket:
        if other_logline alredy included in a pattern:
            continue

        if similarity(logline, other_logline) < 0.6:
            continue

        if similarity < max_similarity_seen_so_far:
            continue
        if similarity == max_similaryt_seen_so_far:
            append other_logline to list of candidates
            continue
        if similarity > max_similarity_seen_so_far:
            dismiss all candidates found so far
            append other_logline to list of candidates

    pattern = logline

    for candidate in candidates:
        merge with pattern
```

Figure 5.8: Pseudocode describing the primary sorting algorithm

After the primary pattens are formed, another iteration over the log file sorts each logline into the primary patterns it resembles the most. If a logline does not match any primary patterns to a certain degree, it is placed in a bucket with other outliers.

When all loglines are sorted into their primary pattern bucket, they need to be compared with each other, to form the final patterns. Assuming that similar loglines are equally long, the loglines can be further sorted by length, to reduce the number of comparisons needed.

Two loglines are assumed to be similar if more than 60% of their items are identical. The following pseudocode describes how patterns are found within one bucket, where loglines have the line length $n$.

For each logline in the bucket, the algorithm finds the candidates with the closest match. By merging the logline and the candidates together, a pattern is formed, where all variables are discovered and removed. The pattern's frequency is calculated based on how many loglines are involved when forming the pattern. Once a logline has been included in forming a pattern, it is never considered again. As a result, the number of comparisons decrease continuously.

When all loglines within all buckets have been investigated, the result is

a list of patterns and their frequency, where each logline in the log file is represented by one pattern.

### Issues and results

```
dovecot ID mail.info IMAP Connection closed 2
dovecot ID mail.info IMAP bob mbox data /home/bob/mail/ INBOX
    /var/mail/bob INDEX /srv/dovecot/var/indexes/bob 1
dovecot ID mail.info POP3 Effective uid gid 2
dovecot ID mail.info POP3 mbox data INBOX INDEX 2
dovecot ID mail.info auth default lookup 3
dovecot ID mail.info auth default passwd bob lookup 1
mimedefang.pl Time 2
mimedefang.pl filter_sender Whitelisted at 2
```

Figure 5.9: Output from the primary sorting algorithm

Figure 5.9 gives an example of the output from the primary sorting algorithm. The algorithm has identified 12 distinct patterns from the 15 lines provided. Since the sample log file is quite small, a non-standard threshold of 0.3 was used. The threshold, which the user can adjust manually when launching the prototype, is set to 0.05 by default. The resulting patterns are of high quality. Whether the second pattern in the output should be clustered with the pattern [*dovecot ID mail.info POP3 mbox data INBOX INDEX*] is a question about pattern precision. If they are clustered, the keywords $POP3$ and $IMAP$ are lost. Additionally, if another $IMAP$ logline comes along, with different variables from the pattern in the output example, the variables in the pattern would be pruned away. The result would be a pattern resembling [*dovecot ID mail.info POP3 mbox data INBOX INDEX*].

Patterns produced using this approach are detailed enough to be useful, and the number of unwanted variables is generally very low.

Unfortunately, the number of comparisons within a bucket may become very large, even when clustring the loglines according to their lenght. By creating less rough primary patterns, loglines will be distribued to a larger number of buckets, and the overall number of comparisons within each bucket will decrease. Unfortunately, creating more precise primary patterns also increase the chances of including variables in the primary patterns. When variables are included, fewer loglines will match the primary pattern, and more loglines will be put in the outlier bucket. As a consequence, the number of comparisons within the outlier bucket explode.

Nevertheless, some loglines may be very frequent within a log file, and these loglines will be clustered together regardless of the number of primary pattern buckets available. The problem is unavoidable, unless a more efficient way of discovering and merging similar loglines is found.

The conclusion is that the algorithm is too ineffective, and must be optimised

to be applicable in real systems.

## 5.6 Conclusion

Pattern mining based on item frequencies alone is not recommended, due to many error sources. It is indeed possible to cluster loglines, but it is a challenge to do it efficiently while still providing quality patterns.

| **Method** | **CPU time(s)** | **Memory usage(KiB)** |
|---|---|---|
| Histograms | 51,806 | 19 172 |
| Occurrence frequencies | 37,395 | 4 239 |
| Primary sorting | 478,630 | 79 480 |
| Subsets | NA | NA |

Table 5.1: Resource usage for the approaches described in this chapter

Table 5.6 and its corresponding graphs, in Figures 5.10 and 5.11 illustrates the difference between the approaches discussed in this chapter.

The table and graphs are a result of running an analysis on a log file consisting of 125 698 log lines (14 076 129 bytes). It contains log lines from 21 different processes including "cyrus", "postfix" and "cron". A standard desktop machine[3] was used for doing measurements. Values for the *subsets* approach are not available because of its massive resource consumption. Tests with the *subsets* approach were cancelled after the implementation had consumed 12GiB of RAM and nearly 7 hours of CPU time[4]. The actual performance of the different approaches is somewhat dependent upon the content of the log files.. For example, the *primary sorting* algorithm will be very inefficient if there is a lot of similar log messages in the log file.

*Histograms* and *occurrence frequencies* are both fairly efficient while requiring low amounts of memory, but as stated in their respective sections, they do not produce the desired results. *Subsets* produces high quality patterns, but the conclusion remains the same as in Section 5.3: the approach is too slow to be of any use.

Section 5.5 states that *primary sorting* produces detailed and useful patterns while keeping the number of unwanted variables low. But, it comes at a cost. Figures 5.10 and 5.11 shows that *primary sorting* is slowest and consumes most memory. It is several times slower than the *histograms* and *occurrence frequencies* approaches.

The next chapter describes another pattern mining technique: tree structures. It is based on the primary sorting idea, but utilise tree structures to

[3]Intel Pentium 4 with HyperThreading and 1GiB RAM

[4]Due to the massive resource demands it was tested on a different machine than the other approaches

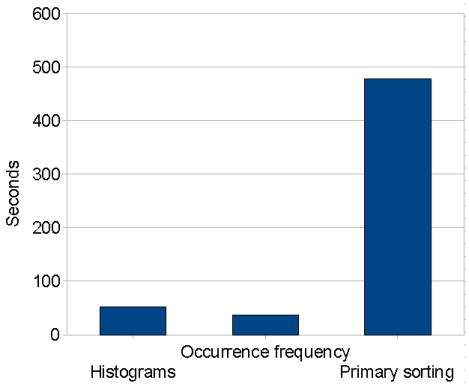



Figure 5.10: CPU usage in seconds

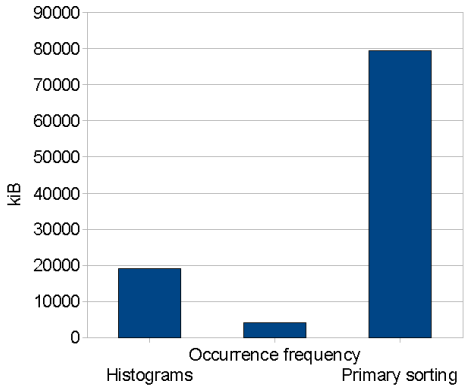



Figure 5.11: Memory usage in KiB

reduce the number of comparisons necessary to create patterns. Presumable, tree structures should reduce CPU and memory usage while still providing pattern of the same, or better, quality.

# Chapter 6

# Pattern mining based on tree structures

Using rough patterns to sort loglines into separate buckets, as described in Section 5.5, proved to be a promising approach, apart from being inefficient. The approach described below seeks to find a more efficient way to cluster similar loglines together, to form a common pattern.

The approach is based on building a tree structure. The goal is to create a tree consisting of all the patterns found in the log file. Every logline is inserted into the tree, either directly, or through merging it into an already existing tree *branch*. A branch represents another logline, or the result of the merge of one or more loglines.

Before describing how the algorithm works, an introduction to the tree *node* is needed. Figure 6.1 illustrates a branch with node $B$'s attributes shown.

Every node in the tree, apart from the root node, represents a logline item. A node has a label that is assigned the name of the item it represents. Each node has a list of references to all its children, and a reference to its sole parent. In Figure 6.1 $B$ has only one child ($C$), and its parent is $A$. Every node apart from the root node has a parent. Additionally, every node has a *tail length*. It describes how many nodes follow the current node in vertical direction. It thus describes the length of the following branch, or branches. Lastly, a node has a dictionary of hidden labels. The hidden labels will be discussed in detail later.

Each tree node has a counter, simply called *count*, that describes how many inserted loglines had this particular item at the given position. When a tree node is first created, its count is set to 1. The *pattern count* variable indicates how many of the inserted loglines actually ended at this node. The pattern count default is 0. In Figure 6.1 the last node in the branch, $D$, has a pattern count of 1. All nodes with a pattern count higher than 0 will from now on be illustrated as a rectangle with a folded corner. A node's count will be attached it to the node edge between the node and its parent,

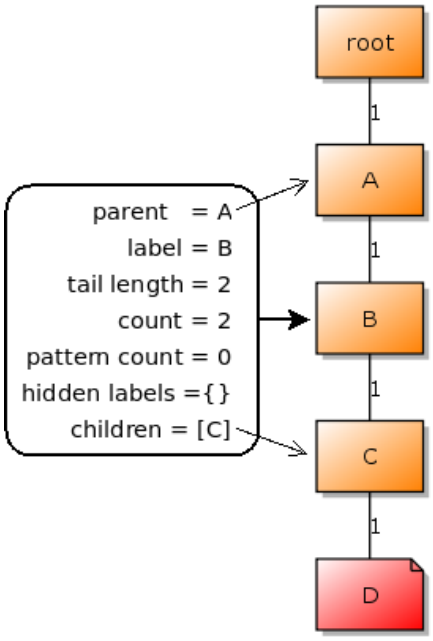



Figure 6.1: Illustration of a tree node and its attributes.

as illustrated.

A node's count is always the sum of its children's counts and its own pattern count. Thus, if the node has no children, its pattern count must be equal to its node count. Equation 6.1 illustrates this rule. $n$ is the number of children under the node.

$$count = pattern\ count + \sum_{i=1}^{n}(child_n\ count) \tag{6.1}$$

After this quick introduction to the tree node, it is time to look at how the tree structure method works. Figure 6.2 illustrates the four stages used by the tree structure approach to produce the final patterns. A *starnode* is simply a node with more than one label. It is called a starnode because a $\star$ has replaced the node's original label. In the following, each stage will be described in detail.

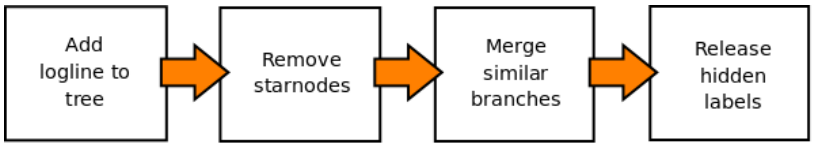



Figure 6.2: The four stages of the tree structure approach

## 6.1 Adding loglines to tree

Before any lines are inserted into the tree, a root node is created. Figure 6.3 illustrates how a logline (to the right) is about to be inserted into a tree with one existing branch, [*A B C D*]. The node labeled *root* indicates the root node into which all loglines are inserted.

The two first logline items, *A* and *B*, have similar names as the two first nodes in the branch. Thus, the logline and the branch have a common stem. The nodes [*A B*] will form the *stemlist*. All tree nodes represented in the stem will immediately have their count increased by one.

The remaining logline items *G* and *H* are denoted the *subline*. The *subline* is a list of the logline items that were not found directly in the branch. The subline must be further investigated before it is decided how the remaining items should be added to the tree. They can either be inserted directly under the last found stem node, or they may be merged into other existing nodes.

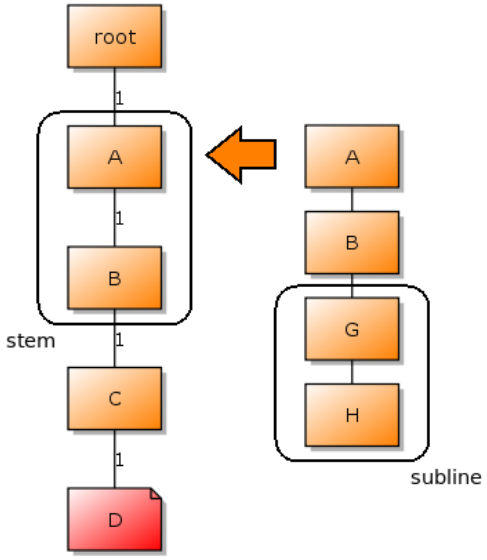



Figure 6.3: Illustration of the terminology used when describing tree structure operations.

The flow chart in Figure 6.4 illustrates how the insertion operates after fetching a logline from the log file. When the first logline is read from the log file, it is inserted in its entirety into the tree, directly under the root node. As long as the subsequent loglines do not share the same stem with any of the existing branches in the tree, all the items in the logline are inserted into the tree, unchanged. This is based on the assumption that the first item in a logline is always the process name, and thus a keyword and not a variable. Recall from Section 5.1 that all words consisting of numbers only are pruned away from the logline. Thus the process name will not contain

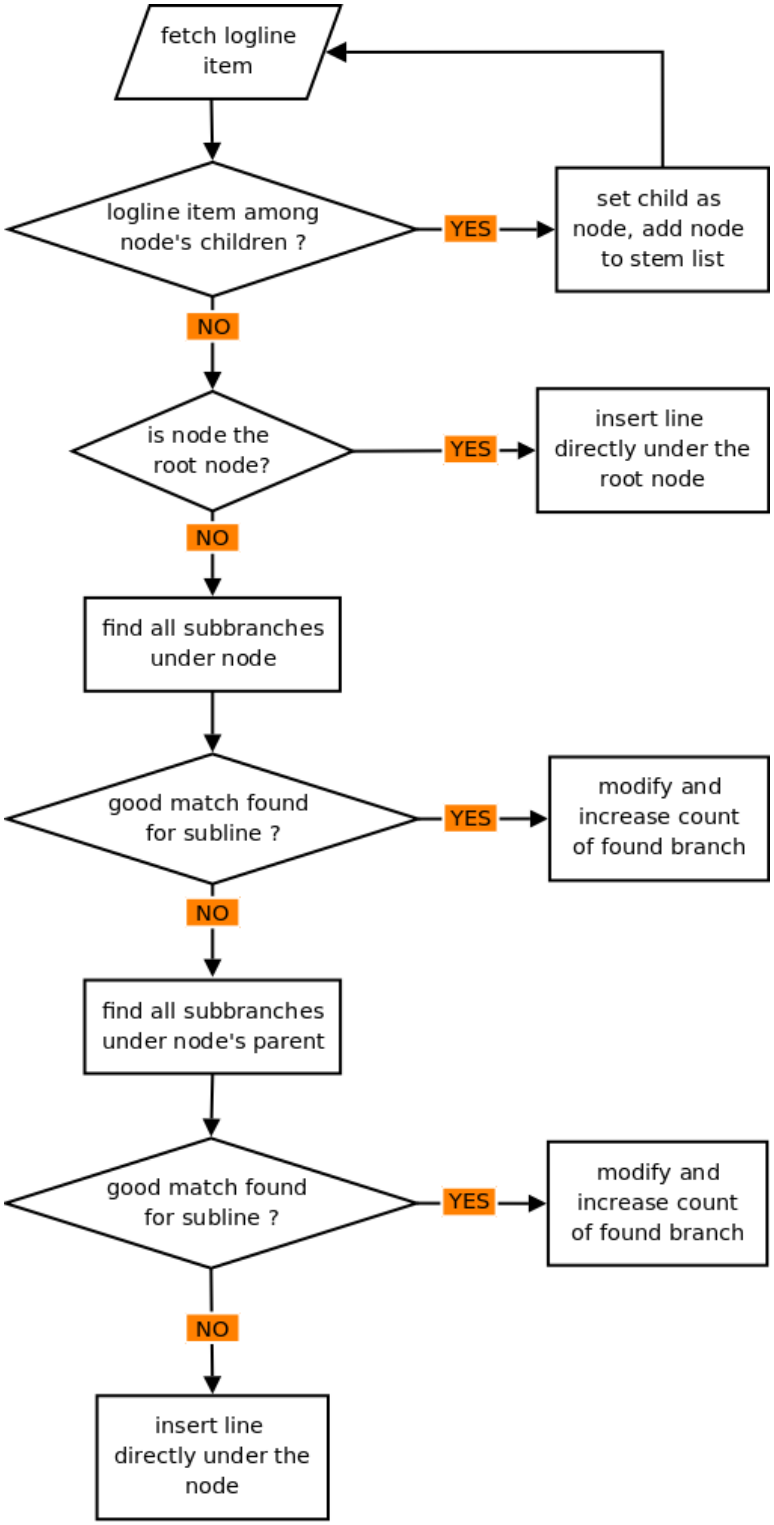



Figure 6.4: Flow chart illustrating how a logline is inserted into the tree

any process identification(pid) number.

As illustrated in Figure 6.4 the algorithm iterates over every item in the logline. For each new logline inserted, the *node* variable is reset to point to the root node. For each item iterated over, the node's children are investigated to see if any of them share the item's name. If it does, this child's count is increased, the child appended to the *stem* list, and finally the *node* variable is set to point to the child and the iteration continues.

If all the logline's items were found in existing tree node's, then the logline is successfully inserted, and the next logline can be fetched from the log file. If, however, an item occurs that does not have a match in the node's children, more investigation is needed. All the node's subbranches with the same length as the subline are investigated to see if any of them are a close match to the subline. The particulars of how this calculation is performed will be explained later. If a close match is found, the subbranch is modified to form a common pattern with the subline.

Figure 6.5 illustrates a situation where a logline $[A\ B\ C\ H]$ is about to be inserted into an existing branch $[A\ B\ C\ D]$. The nodes $[A\ B\ C]$ constitutes the stem, while the last logline item $H$ will either have to be added to the $[A\ B\ C]$ branch, or be merged with the last node $D$.

Figure 6.6 demonstrates what happens when this merge is performed. The $D$ node is renamed to a *starnode*, giving it a $\star$ label. The previous label $D$ and the newly arrived item $H$ are both stored in the starnode's hidden labels. More on hidden labels later. The count of the last node is increased, and the insertion is complete.

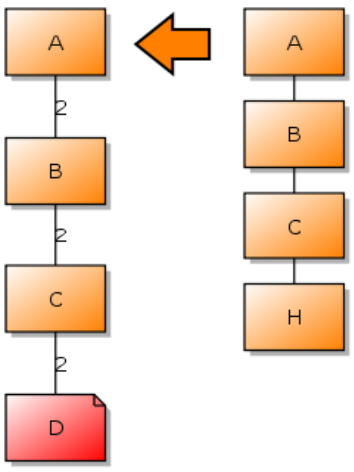



Figure 6.5: Inserting line into existing branch

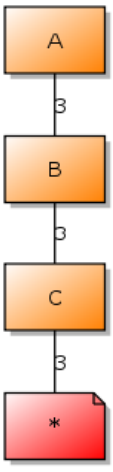



Figure 6.6: Result of inserting line, after renaming existing node

If a good match is not found, the subbranches of the the node's parent are investigated, and the same comparison is performed. If a close match is found, then the subbranch of the parent node is chosen and modified,

otherwise, the subline is inserted directly under the node.

Figure 6.7 illustrates why, when no good subbranches are found under the node, the subbranches of the node's parent is investigated. By looking at the two branches already residing in the tree, it is clear that the leftmost branch is the one most similar to the items being inserted. However, both item $A$ and $K$ are found in the tree, and thus the algorithm will search under node $K$ to find a branch similar to the subline $[B\ C]$. Figure 6.7b illustrates how the tree would look like if the subbranch was inserted under the $K$ node. Obviously this is suboptimal since a very close matching branch already exists under node $A$. Therefore, when no match is found under the current node, we move one step up in the tree and repeat the search. The result is shown in Figure 6.7c.

The rationale for this action is that the last stem node found may actually be a variable, as all variables are added to the tree when a logline of a particular type is inserted the first time. To illustrate with a practical example, imagine that two lines have been inserted into the tree, [*user john logged in*] and [*user jane logged in*]. These two loglines have produced the tree branch pattern [*user* $\star$ *logged in*]. Also, a user paul has been denied access twice, producing the tree pattern [*user paul denied access*]. Then, poor paul is granted access, and produce the logline [*user paul logged in*]. No doubt this last line belongs with the other *logged in*-messages, and not under the *paul* node.

After having describing the overall procedure for how loglines are inserted into the tree structure, it is time to look at how a tree branch is considered a good match to a subline.

As described earlier, all the node's subbranches, of equal length as the subline, are first mapped out. A simple method calculates how many operations are needed to turn the branch into the subline. It is based on a low threshold, throwing away all branches who are less than 20% similar to the subline. The threshold is arbitrarily chosen. Its only purpose is to make sure that obvious mismatches are discarded. The method returns, at most, the two best candidates. If no candidate is found, the search is terminated.

If however, this first screening returns a candidate or two, the results must be further investigated for a conclusion to be reached. Since the screening process simply counts the number of operations required, such as insertions and deletions, to turn the branch into the subline, it does not take into account that the order of the items is significant. The candidates are therefore investigated more thoroughly by comparing the subline and the branch, item for item. The purpose is to decide which of the candidate branches is the best fit for the subline, if indeed any of them are close enough matches.

Exactly how to decide if a branch is a good match or not, is a challenge. Since there is no prior knowledge of the items about to be inserted, direct comparison item for item, and simple mathematics is used to calculate the likeness between the logline and branch.

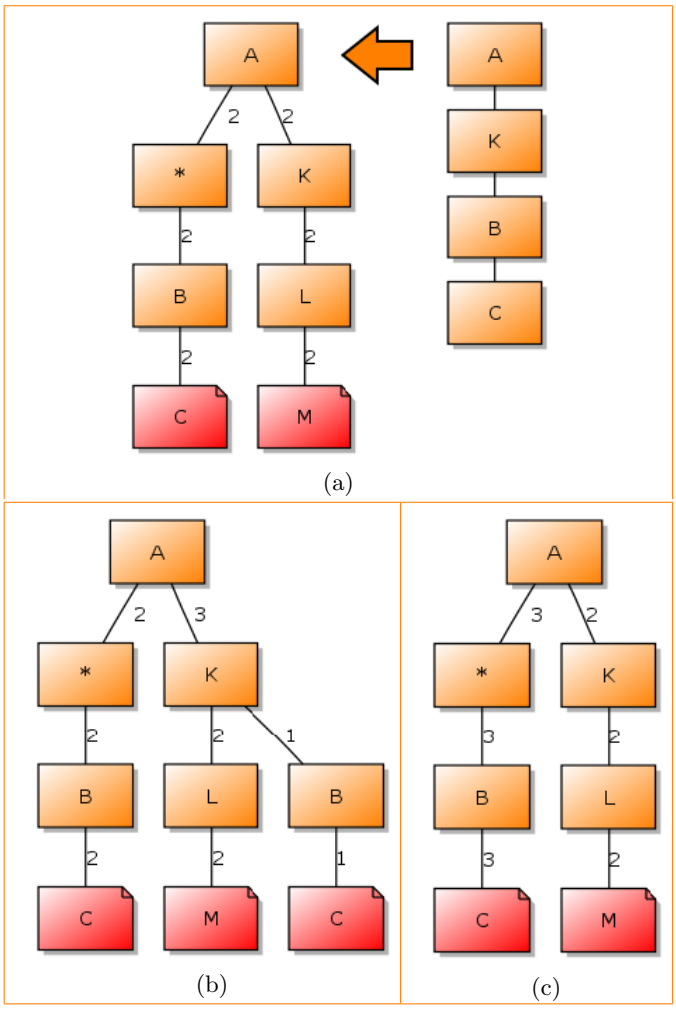



Figure 6.7: Insertion example where parent node should be investigated

If a branch is found to be a good match, some changes may have to be performed on its nodes. A list denoted the *layout* is used as a guideline for which nodes should be modified, and how.

For each comparison being performed, a number is added to the *layout* list. If the item and the corresponding branch node's label are identical, then a 0 is appended to the layout. If they are not identical, a 1 is appended. If however, the node that the item is being compared to has a star($\star$) as its label, then a 2 is appended to the layout. In the layout, a 0 means that the node stays unchanged during insertion, a 1 means that the node should be renamed to a starnode, and a 2 means that the item name should be added to the node's hidden labels.

Beneath is an example of how the layout is generated. The first node does not have a label identical to the first item to be inserted, and thus the layout element at this position must indicate that a rename should take place. The


Figure 6.8: Flow chart illustrating the process of accepting or rejecting a branch.

$B$ node in the branch will stay unchanged, and the corresponding layout element is a 0. The same is the case for node $C$, $K$, and $L$. The fourht node in the branch has been previously renamed, and already holds hidden labels. To indicate that another hidden label should be added to the node, corresponding element in the layout is a 2. The complete layout for the branch [*A B C * K L*] will be [*1 0 0 2 0 0*].

```
Branch:  [ A B C * K L ]
Logline: [ D B C R K B ]
Layout:  [ 1 0 0 2 0 0 ]
```

When all items have been compared, the layout describes how many of the branch's node's matched, and how many mismatches there were. The more matches there are, the better the branch suits the logline. The rationale for treating starnodes as neither a match(0) or a mismatch(1) is because they are neither. We treat them as *neutrals*. When calculating the *common fraction* ( the amount of matches per element in the layout ), we simply take the starnodes out of the equation. Equation 6.2 shows how the common fraction is calculated.

$$common\ fraction = \frac{\sum(matches)}{\sum(matches) + \sum(mismatches)} \tag{6.2}$$

We wish to find the best branch into which we can merge the subline. To do that, we must find the branch with the highest common fraction. However, loglines may have very different lengths, and the amount of keywords per variable may also vary greatly. Finding the correct threshold for when a branch should be discarded and not is therefore difficult, and may have to be tailored to the specific log file.

The flow chart in Figure 6.8 illustrates how the branch is evaluated. The default variable values from the prototype are the result of much trying and failing. The values chosen are based on the results from testing a wide range of different loglines. All variables can be changed in the prototype's configuration file. The variables used in the flow chart are abbreviations. They are described in Table 6.1. The default values are shown in brackets.

The first test is whether the common fraction is less than the $cfl$ limit. If it is, the branch is considered a bad match, and a common fraction of 0 is returned, together with the layout. If the branch passes the first test, it continues to the next. It tests whether the length of the stem constitutes a fraction more than $lsf$ of the entire branch. If in addition the common fraction is greater than a limit $lscf$, the branch is considered a good match, and the common fraction and layout is returned. Otherwise, the testing continues.

If however, the length of the logline to be inserted is equal or shorter than a threshold $vsl$, then the branch is accepted and we return the common fraction and the layout. The first test already assured that the minimum

| | | |
|---|---|---|
| cfl | (0.60) | *common factor limit*, the lowest acceptable fraction of 1's in the layout when ignoring 2s. |
| lsf | (0.66) | *long stem fraction*, fraction at which the stem is considered long, compared to the length of the entire branch. |
| lscf | (0.75) | *long stem common fraction*, acceptance limit when stem is considered long. |
| vsl | (3) | *very short line*, a number for when a line is considered very short. |
| sscf | (0.75) | *short stem common fraction*, acceptance limit when stem is not considered long. |
| vss | (2) | *very short subline*, a number for when a subline is considered very short. |
| mcl | (0.50) | *maximum changes limit*, upper limit for the acceptable number of 1's in the layout. |
| mml | (0.40) | *minimum matches limit*, lower limit for the required number of 0's in the layout. |

Table 6.1: Configuration variables

required matches is fulfilled, and when the logline is short, this usually constitutes a large fraction of the layout.

If the common fraction is greater than a fraction $sscf$, and the length of the subline is equal to or shorter than a threshold $vss$, then again the branch is considered a close match, and the common fraction and layout is returned.

If the fraction of changes required to the branch (i.e. the number of 1's per element in the layout) is larger than a threshold $mcl$, or the required number of matches (i.e. the number of 0's in the layout) is lower than a threshold $mml$, the branch is again rejected. Note that the $mcl$ and $mml$ limits depend on the thresholds set for what is considered a short line and a short subline, respectively. Shorter lines may call for fewer changes to be accepted, while shorter sublines may require fewer matches. Unlike the common fraction, the change fraction and matches fraction are calculated based on the entire layout, and does not ignore starnodes. Thus, these fractions assure that the number of starnodes present in a branch is kept at an acceptable level. Finally, if the branch survived the last two tests, it is accepted, and the common factor and layout is returned.

All candidates returned from the first screening are tested in this manner, and the candidate that returns the highest common factor is selected.

When actually inserting the subline into the tree structure, the branch nodes are iterated over, and for each node the accompanied layout is consulted. To recap, if the corresponding element in the layout is a 0, the node count is simply increased. If it is a 2, then the corresponding logline item is inserted into the node's hidden labels, and the count increased. Finally, a 1 demands that the node should be renamed to a starnode, appending the current label

and the corresponding logline item to the node's hidden labels. The newborn starnode's count is increased. The last node in the branch will always have its pattern count increased, regardless of the corresponding layout number.

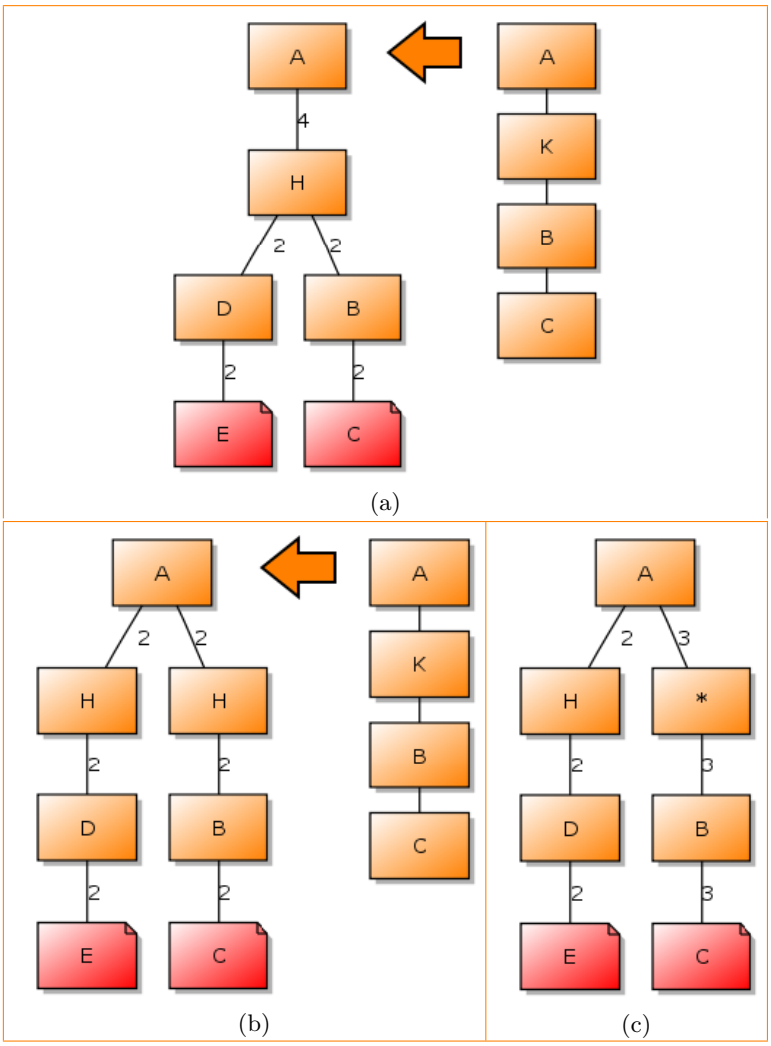



Figure 6.9: Splitting of branches before rename of node

It is worth noting that if a node is about to be renamed to a starnode, and the node has more than one subbranch, the subbranch being modified must be split out. If it is not, the other subbranches may loose precision. Figure 6.9 illustrates such a situation. The topmost node $A$ represents an arbitrary node somewhere in the tree structure. It is not the root node.

The $K$ item in the logline being inserted will cause the $H$ node to be renamed. Before the renaming can take place, the subbranch $[H\ B\ C]$ must be split out. Otherwise, the branch $[A\ H\ D\ E]$ will loose precision when its $H$ node is renamed to a starnode. $H$ may be a keyword, and it should not be renamed before a logline similar to $[A\ H\ D\ E]$ comes along and changes it. Figure 6.9b illustrates how the branch is split out into a separate subbranch

under the $A$ node. In Figure 6.9c the node has been renamed and the logline inserted.

## 6.2 Removal of starnodes

When all loglines have been inserted into the tree structure, the starnodes must be removed. It is a simple procedure. Postorder traversal is used to visit all the nodes in the tree. When a starnode is found, all its children are adopted by the starnode's parent, and its hidden labels are also given to the parent node. If the starnode has a pattern count higher than 0, the parent's pattern count is increased accordingly. Finally, the starnode is deleted.

There is one exception from this procedure. If the starnode's parent is another starnode, the hidden labels are not adopted by the parent. The count of a starnode is the sum of its hidden labels. Thus, if a starnode inherits another starnode's hidden labels, the node count will become incorrect. It may also cause problems for the releasing of hidden labels later on. Figure 6.10 and Figure 6.11 illustrate the process of removing starnodes. The topmost node $R$ is not the root node, but could be any other node situated somewhere in the tree structure. Observe that node $R$ will inherit hidden labels from both its starnode children. The second starnode in the rightmost branch will simply be deleted, and the $B$ in the shortest branch will inherit the pattern count from its starchild.

## 6.3 Merging of branches

The removal of starnodes may cause some branches in the tree to become identical. These branches need to be merged to form one unified pattern. It is also desirable to merge branches with similar stems, to ease the process of releasing hidden labels in the next step.

The algorithm works in a similar manner as for the removal of starnodes, using postorder traversal of the tree. For each node visited, the parent node is investigated to see if the node has a sibling with an identical label. The length of the branches is no longer of interest. Also, the nodes' tail lengths might be incorrect, due to the removal of starnodes, making the branches shorter. If no sibling was found, the algorithm simply moves on to the next node.

Figure 6.12a illustrates how a merge is performed when a sibling is found. The algorithm starts at the lowest, leftmost node $G$. Since $G$ has no siblings, the algorithm moves on to its parent node $B$, finds no siblings and continues up to node $A$. Node $A$ finds a sibling in the middle $A$ node. Before these two nodes can be merged, potential children, grandchildren etc. must be merged first. A recursive method searches down the two branches until either one of the branches ends, or the two branches no longer have nodes in common at

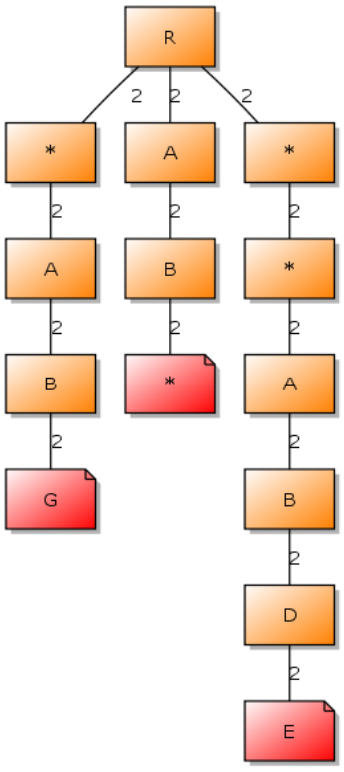



Figure 6.10: Tree before starnodes are removed.

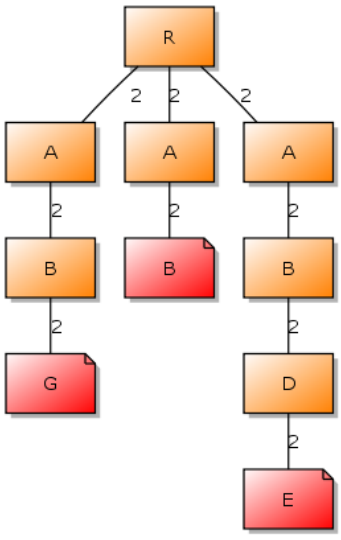



Figure 6.11: Tree after starnodes have been removed.

the same depth. From that point, the algorithm starts merging descendants of the current $A$ node into the nodes in the middle branch, as it moves back up the tree. Before a node can be merged into another, the other node must first adopt the node's children. The node gives away all its hidden labels, count and pattern count to the other node. Finally the node is deleted and the merge is complete. Note that the adoption of children may also call for merging of branches, if the adoptive parent already has a child with identical label.

In the case of the $A$ node, the recursive search only moves one level down the tree. Since the middle $B$ have no children, the leftmost node $B$'s child $G$ can be adopted directly by the middle $B$ node. This operation is shown in Figure 6.12b. When the leftmost $B$ have no more children left, it can be merged into the middle $B$ node. All $B$'s hidden labels, pattern count and node count is added to the middle node, before the current $B$ is deleted. The result is given in Figure 6.12c.

The algorithm moves back up to the leftmost $A$. Since it has no more children, it can be merged directly into the middle $A$. Like before hidden labels, count, and pattern count are moved before the node is deleted. Figure 6.12d shows the result after having merged the two first branches. The algorithm continues in the same manner, until all branches with common stems are

merged. Figures 6.12e, 6.12f, and 6.12g illustrate the final steps of the merge process.

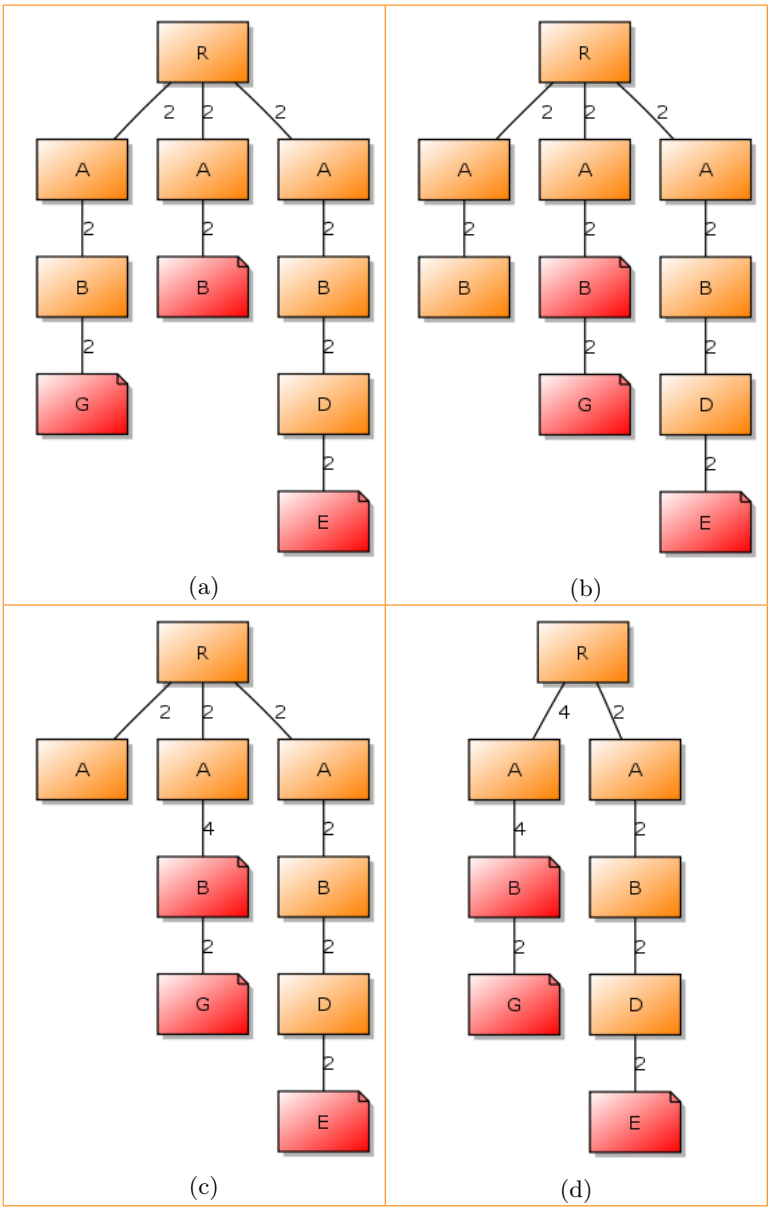



Figure 6.12: Merging of branches, part 1

## 6.4 Releasing hidden labels

Not much have been said about the hidden labels so far. As described earlier, hidden labels are stored in starnodes to keep track of all the items the starnode has been assigned during the insertion of loglines. This way, no information is lost when renaming a node to a starnode.

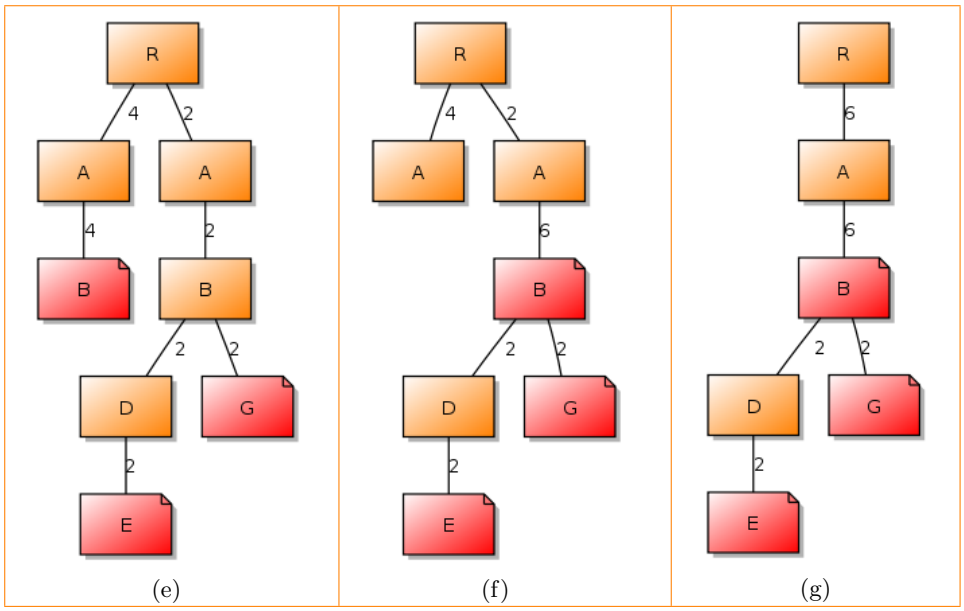



Figure 6.12: Merging of branches, part 2

The hidden labels is a nested dictionary of the form:

{ successor : {label : value, label : value}, successor : {label : value} }

The keys of the inner dictionaries are the actual hidden labels, the names of the items hidden behind the starnode. The value is a counter for how many times that particular label has been added to the hidden label dictionary.

A starnode may have more than one subbranch underneath it in the tree structure. It is therefore necessary to to keep track of which of these subbranches the hidden label belongs to. Recall from Section 6.1 that a branch may have to be split out before a node can be renamed to a starnode. Without keeping track of which subbranch the hidden label belongs to, it is impossible to correctly divide the hidden labels between branches. Leaving an incorrect label in a node may eventually result in illegal patterns. More on that in a moment.

The outer dictionary key, the *successor*, is a text representation of all the non-starnodes in the subbranch the labels belong to. An empty successor(") simply indicates that the label belongs to the last node in a branch. Every starnode with an empty successor must have a pattern count equal to the sum of all hidden label values in the empty successor. Figure 6.13 illustrates some hidden label examples. In Figure 6.13a we see an example of an empty successor. In Figure 6.13b the starnode has two different successors. To produce these hidden labels, four loglines must have been inserted, $[A\ s\ D]$, $[A\ t\ D]$, $[A\ s\ D\ K]$, and $[A\ t\ D\ K]$.

Figure 6.13c and Figure 6.13d show the same two trees after the starnodes

have been removed. Observe in Figure 6.13c that only the hidden labels from the upper right starnode is inherited by the $R$ node. The hidden labels from the lower right starnode are lost. The middle $B$ node has inherited its starnode child's hidden labels and pattern count, making it the last node in the branch. In Figure 6.13d the $A$ node has inherited all hidden labels from its previous starnode child.

The reason for storing the hidden labels in the first place is because keywords sometimes are confused for variables by the insertion process. It is desirable to keep as many keywords as possible in a pattern. Attempting to achieve that goal by demanding a high degree of matches before merging a logline into a tree branch, might result in a higher fraction of variables present in the final patterns. Variables in the final patterns produce less correct results than having more coarse patterns. Consequently, the insertion process should allow less similar loglines and branches to be merged, rather than risk having variables in the patterns. The hidden labels assure that no information is lost along the way.

The releasing of hidden labels is a step added to the tree structure approach in an attempt to dig out some of those incorrectly hidden keywords.

The algorithm works its way down the tree, using preorder traversal. For each successor in the node's hidden labels, each of the node's children are investigated to see if their label is present among the successor's hidden labels. If none of them can be found, the algorithm moves on to the next node.

The rationale for choosing this way of identifying potential keywords is this: If a node with an identical label as the hidden label exist underneath the node, it is reasonable to assume that the hidden label is a keyword.

If a child's label is discovered amongst the hidden labels, the hidden label's successor is investigated. If the successor is empty(''), it signals that the hidden label used to reside in a node that marked the end of a branch. The child's count and pattern count is increased according to the hidden label's value, and the hidden label is deleted from the node.

If the successor is not empty, all subbranches under the node must be investigated to find the branch consisting of the nodes listed in the successor. If the count of the first node in the branch equals the hidden label's value, the entire branch is adopted by the child. Note that merging may be necessary if the child node already has a child with the same name as the hidden label.

When the first node in the branch has a count larger[1] than the value of the hidden label, the branch must be split out.

Unlike the split performed in the insertion process when renaming nodes, this split may result in two identical branches, possibly with different node

[1]The branch count can never be smaller than the value of the hidden label. If that happens, an error has occurred in the code.

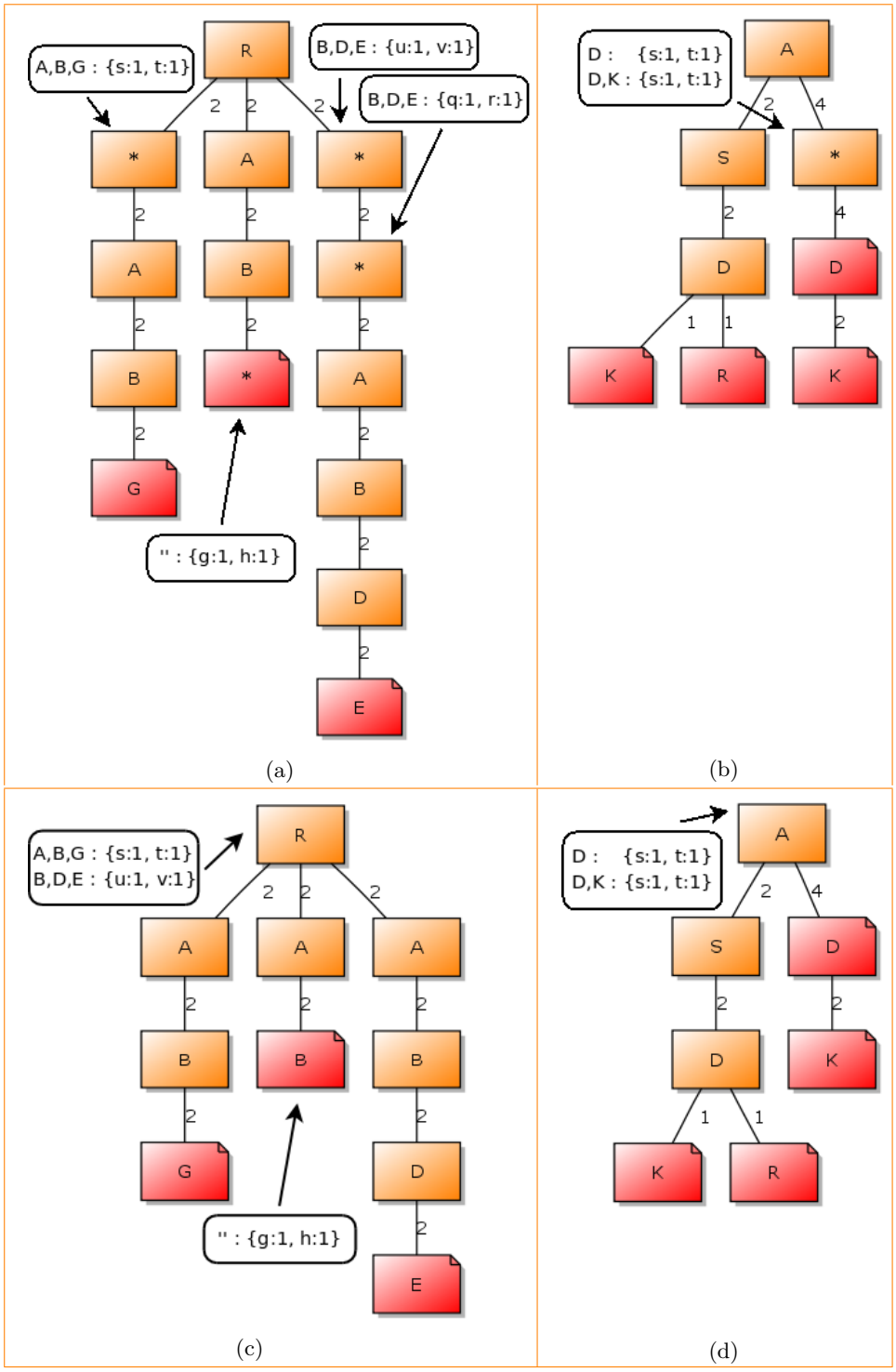



Figure 6.13: Hidden labels before and after removal of starnodes

counts. As a consequence of this splitting, the hidden labels residing in the affected nodes are deleted. The rationale behind this action is that there is no way of knowing to which branch the various hidden labels belong. Keeping incorrect hidden labels may eventually cause incorrect patterns to be formed.

The newly split out branch with node count similar to the hidden label value is afterwards adopted by the child node. Again, merging may be necessary. The child's node count is increased, but not its pattern count.

When a hidden label is deleted, its successor is examined to see if it has only one hidden label left. If that is the case, it is very likely that this remaining hidden label is also a keyword. If the hidden label is not found amongst the node's children, a new tree node is created with label equal to the hidden label and count 0. The algorithm proceeds as usual to release the hidden label.

Figure 6.14 illustrates an example of how the process of releasing hidden labels is performed. The box to the right in the figures show which hidden labels remain in the $A$ node. The colored box appearing to the left indicates which hidden label is being handled at the moment.

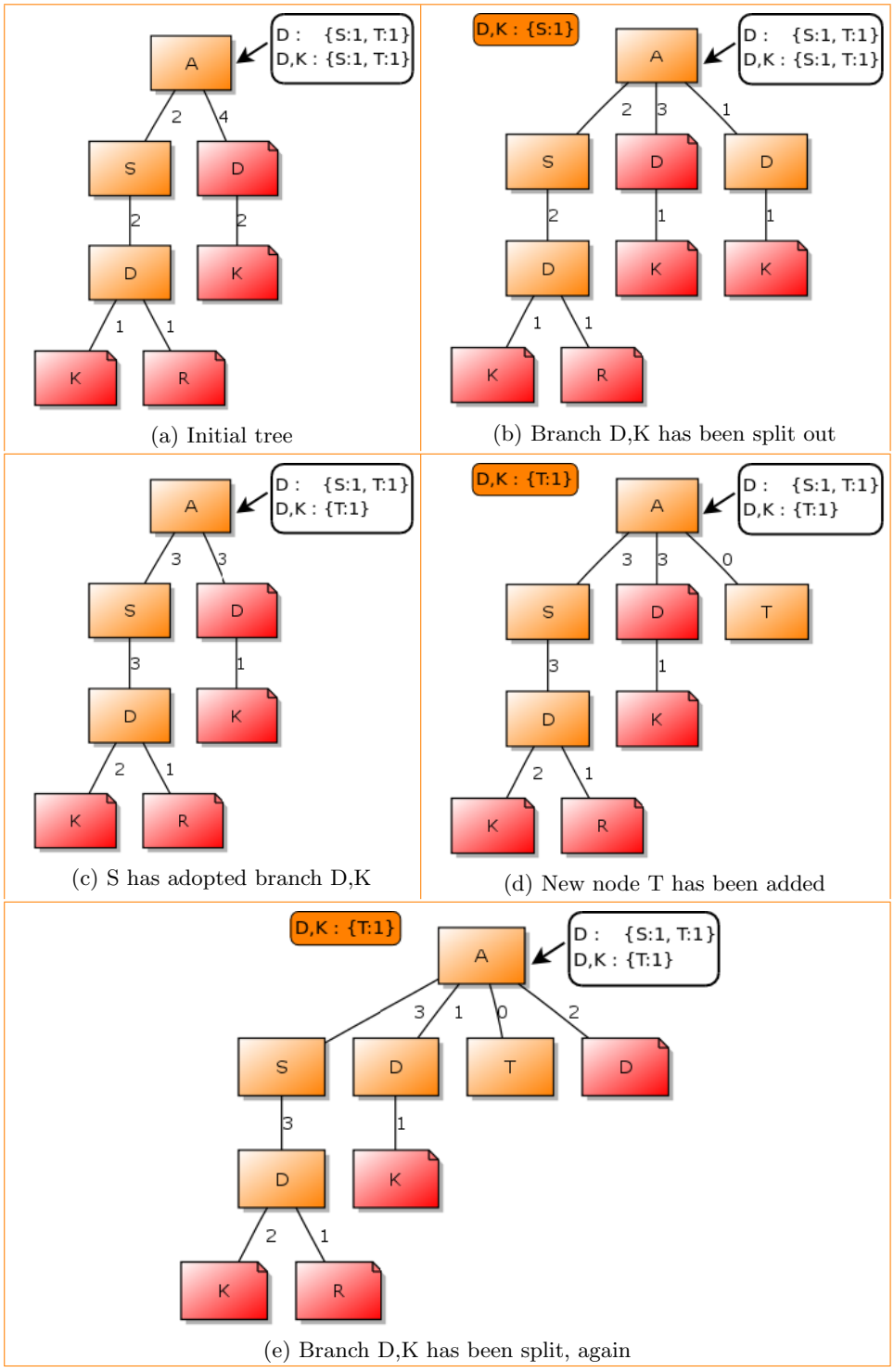
The algorithm starts at node $A$, with the successor *D,K*. It finds a child with label identical to to the first hidden label $S$. Figure 6.14b illustrates how the *D,K* branch is split into two identical branches, with different node counts. In Figure 6.14c the $S$ node has adopted the newly split out branch. After removing hidden label $S$ from successor *D,K*, only one more hidden label remains. This last label, $T$, is released in Figure 6.14d.

The algorithm continues to investigate hidden labels belonging to successor *D,K*, and observes that a child $T$ exists. Once more the *D,K* branch must be split. As shown in Figure 6.14e the branch is now split into a branches *D,K*, and *D*. The node count of node $T$ is increased according to the hidden label value, and branch *D,K* is adopted by node $T$.

Since there are no more hidden labels to investigate under successor *D,K*, the algorithm moves on to the next successor *D*. It starts with hidden label $S$. The $S$ child is discovered and the $D$ node is split into two identical nodes. In Figure 6.14h the $S$ node has adopted the split out $D$, and only one more hidden label remains. Since the $D$ node has equal count as the hidden label value, the node can be adopted directly by node $T$. This completes the example. Observe that the small example tree has become more detailed than the initial tree.

Figure B.5 in Appendix B shows how logline patterns change after hidden labels have been released. The patterns becomes more precise.

This last step of releasing hidden labels is not mandatory. If the resulting patterns from the merging step is sufficiently detailed, the releasing of hidden labels can be dropped.




(a) Initial tree

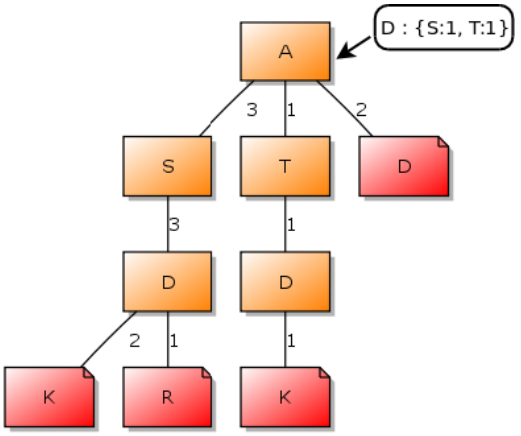
(b) Branch D,K has been split out

(c) S has adopted branch D,K

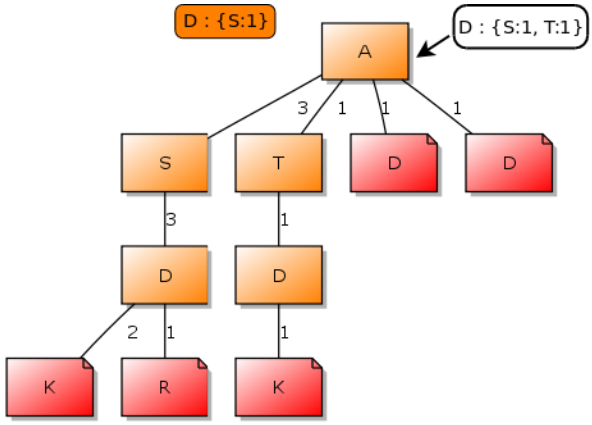
(d) New node T has been added

(e) Branch D,K has been split, again

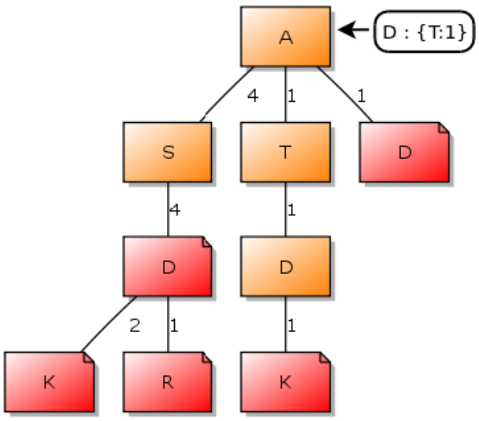
Figure 6.14: Releasing of hidden labels, part 1



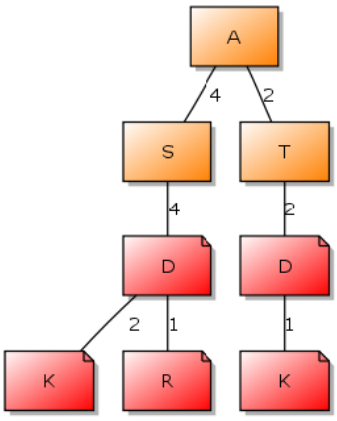

(f) Node T has adopted branch D,K




(g) Node D has been split




(h) Node D has been adopted by S




(i) Node T has adopted D

Figure 6.14: Releasing of hidden labels, part 2

## 6.5 Combining the stages

The tree structure method is significantly more complex than the other pattern mining approaches investigated. It was therefore necessary to explain the four stages of the approach in separate steps. In the following an example illustrating the entire approach is given. It shows how the four stages work in combination.

Figure 6.15a illustrates a logline about to be inserted into an existing subtree. $A$ can be any node in the tree structure, including the root node. The logline items $A$ and $B$ are found directly in the tree, and the tree nodes [$A$ $B$] will form the list if stem nodes. Note that only the leftmost $B$ node is considered, since the rightmost $B$ has tail length different from the length of the remaining items to insert ($K$, $D$ and $E$). Recall that a node's tail length is the length of the branch(es) residing under it, also known as the node's subbranches. During the insertion process, all subbranches under a node must have the same length.

After investigating the branches under node $B$, the insertion process will identify branch [$A$ $B$ $C$ $D$ $E$] as the closest match to the logline. The branch's layout will be [*0 0 1 0 0*], meaning that all nodes will stay unchanged except for node $C$, which will have to be renamed to a starnode.

Since the last node in the stem list, the leftmost $B$, has more than one subbranch, the branch [$C$ $D$ $E$] has to be split out before any changes can be made to it. Otherwise, the branch [$A$ $B$ $C$ $R$ $S$] will be changed too, and consequently loose precision. Figure 6.15b illustrates the splitting process.

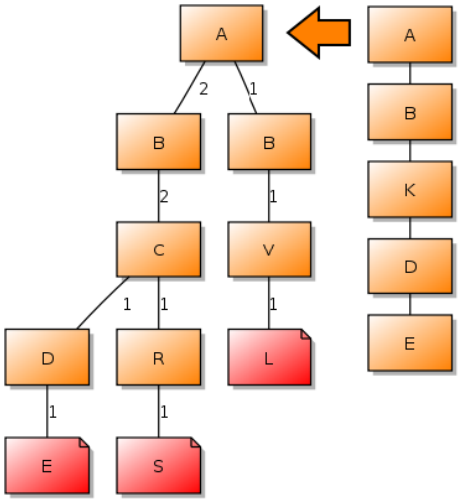
In Figure 6.15c the logline has been inserted. Observe that the count of all the branch's nodes have been increased by one. The newly renamed node has received two hidden labels, $C$ and $K$, both belonging to the successor [$D$,$E$].

When the insertion process has completed, the method starts pruning away all starnodes. The result is shown in Figure 6.15d. The starnode's parent, the leftmost $B$, has inherited all its hidden labels.

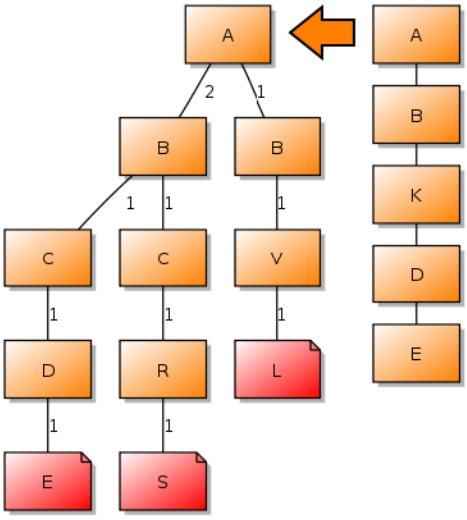
Figure 6.15e illustrates the result after similar branches have merged. The only merged performed in this example is the merging of the two $B$'s.

The process of releasing hidden labels produce the final result shown in Figure 6.15f. When node $B$ is investigated, it is discovered that a node with a similar labels as the hidden label $C$ exists among $B$'s children. Thus, the hidden label can be released.

First, the child's count is increased by 1, which is the value of the hidden label. The hidden label belongs to the successor [$D$,$E$]. The branch [$D$,$E$] is therefore split into two equal branches, and one of the branches is inherited by the child $C$. The original branch [$A$ $B$ $C$ $D$ $E$] has been restored, and the precision of the patterns have been improved. This completes the example.

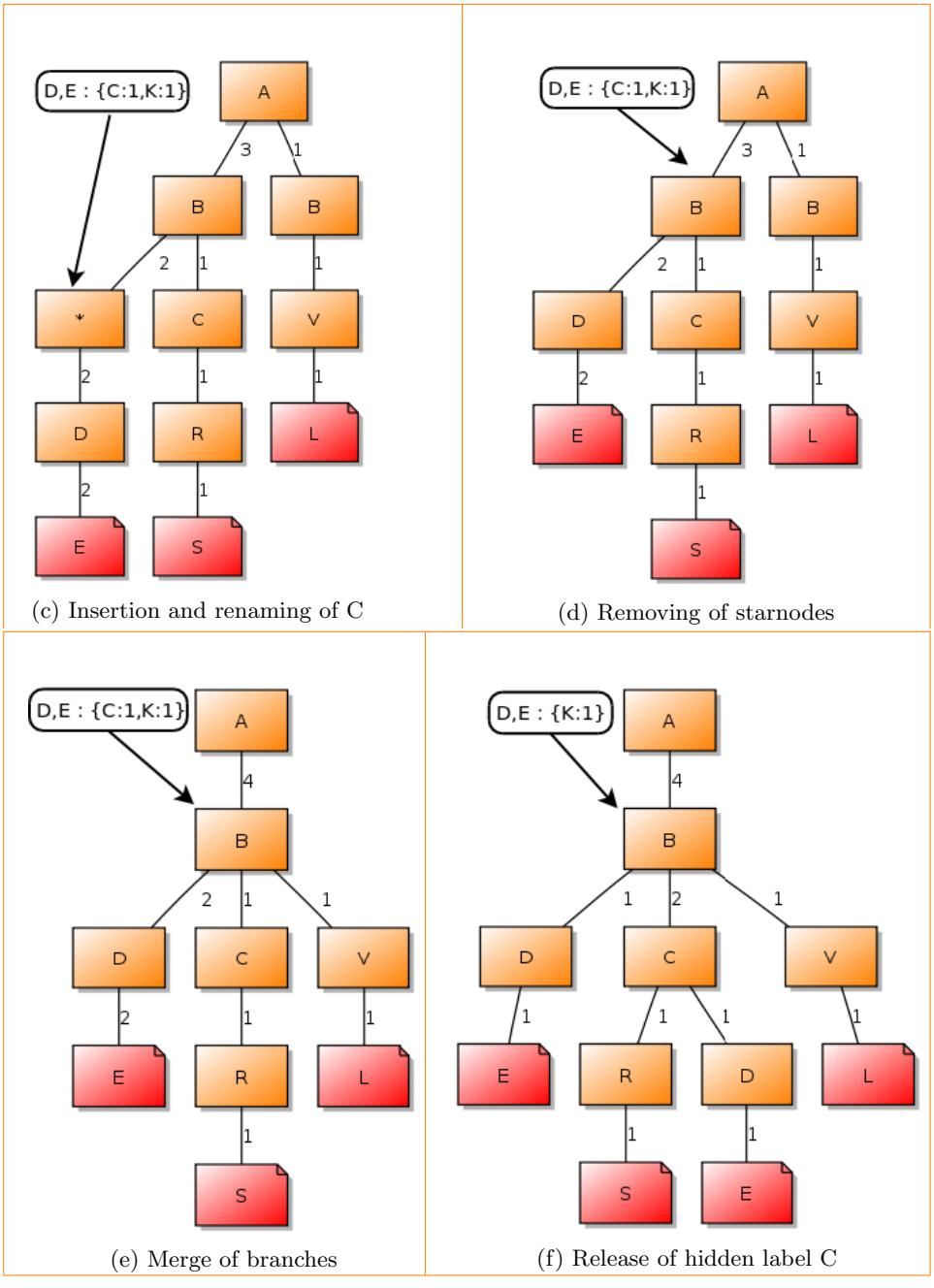



(a) Initial tree




(b) Split branches before insert

Figure 6.15: Example showing all four stages, part 1




(c) Insertion and renaming of C

(d) Removing of starnodes

(e) Merge of branches

(f) Release of hidden label C

Figure 6.15: Example showing all four stages, part 2

## Issues and results

The release of hidden labels may create more precise patterns. However, imagine that we have the following branches in the tree structure:

```
user * logged in
user * logged out
user paul denied access
```

The username *paul* is also present amongst the two starnodes hidden labels. When the starnodes are removed, the *user* receives the hidden labels from the two starnodes. Finally, when releasing hidden labels, the *paul* node will cause every *paul*-hidden label to be released. The result is several patterns containing the user name *paul*. To avoid such situations to occur, a threshold on the hidden label value is introduced. The default threshold value is 20%. (It can be changed using the prototype's configuration file.) If the hidden label value does not constitute at least 20% of the sum of all hidden labels belonging to the successor, the label is not released. The rationale behind this measure is that variables usually result in numerous hidden labels, whereas the number of keywords will usually be limited to a small amount.

It is of course still possible that a username can be released, but in that case it would have to have a value significantly higher than the rest of the usernames, or the number of hidden labels must be very low. In the first case, the username may actually be of interest, since such behaviour may be an indicator of abnormalities.

The approach applied above to qualify a hidden label for release is a simple one. It would be interesting to introduce more analysis into the release algorithm, to find more keywords. It is difficult to discover keywords incorrectly hidden in starnodes during insertion of loglines. No previous knowledge of the inserted items is available, and nothing is known about the future loglines to be inserted. However, when all loglines have been inserted into the tree structure, we can for example compare the number of hidden labels to the node count. If the number of hidden labels is very low compared to the node count, it is likely that the hidden labels are keywords.

Imagine that the following loglines are inserted into the tree:

```
connect from host example1.example.com
disconnect from host exampler1.example.com
connect from host example2.example.com
disconnect from host example2.example.com
connect from host example3.example.com
disconnect from host example3.example.com
```

What happens is that the insertion process recognise the hostname as a keyword, while the keywords *connect* and *disconnec* are considered variables. The resulting patterns will look like this:

```
* from host example1.example.com
* from host example2.example.com
* from host example3.example.com
```

This is an example of a problem that most likely can be solved by further analysis of the hidden labels. The problem can be recognised by a node having a relatively large number of successors, but only a few unique hidden labels. Further analysis of hidden labels is the subject of future work.

Returning to the insertion process in Section 6.1, the main issue is to find a potential branch to merge an incoming logline with. It is a challenge to find a formula that will accommodate loglines of all lengths, and with a various fraction of keywords. The most difficult loglines to handle are those with a very low fraction of keywords, such as reports about spam emails where the email subject is attached. To avoid loglines that are difficult to cluster together from polluting the results, the prototype accepts an exception list. The exception list consists of simple regular expressions, one per line, and will exclude all loglines matching any of the exceptions.

Each resulting pattern has inherited the pattern count of the last node in the branch it represents. To assure that the resulting patterns are correct, a primitive test script has been created. It simply searches for the patterns in the log file, using the unix *grep* command. Once it has found a match it returns. There is no use in finding all instances, since the pattern count can not be verified directly. The reason for this is that a pattern may match loglines also matched by other patterns, potentially causing the reported count to be much higher than the registered pattern count. Instead, the sum of all the patterns is compared to the number of loglines in the log file. When subtracting the loglines matched by the patterns from the previously mentioned exceptions list, the sum of the pattens should be equal to the log file length. During the test phase of the prototype, no incorrect patterns were found.[2]

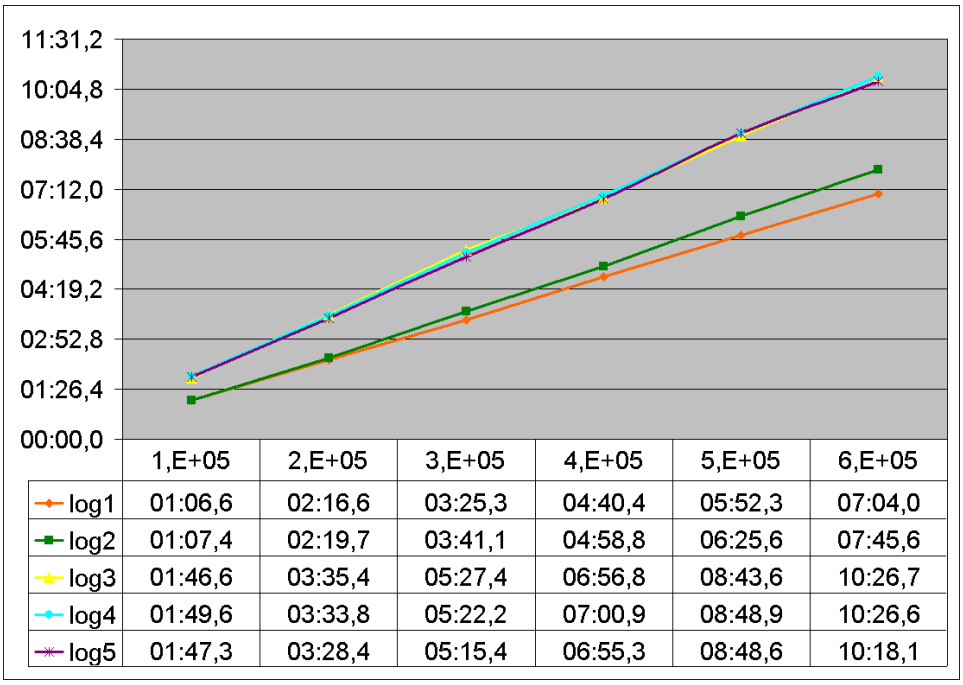
### 6.5.1 Performance

Figure 6.16 shows the execution times for 5 different log files. *log*3, *log*4, and *log*5[3] contain a wide variety of mail-related loglines, of different shapes and lengths. *log*1 and *log*2[4] are DNS *named* logs, and consist chiefly of very similar loglines of nearly the same length.

[2]Occasionally, symbols not matched by the regular expression metacharacter . (dot) occur in the logfile. The grep command is then unable to find a match for the pattern, and the pattern is incorrectly reported as a non-match. Additionally, regular expression metacharacters such as + (plus) can also cause the search to fail. These problems have not been prioritised, and error correction is left for future work.

[3]Provided by the Department of Computer and Information Science at the Norwegian University of Science and Technology.

[4]Provided by UNINETT Norid AS, the .no registry




| | 1,E+05 | 2,E+05 | 3,E+05 | 4,E+05 | 5,E+05 | 6,E+05 |
|---|---|---|---|---|---|---|
| log1 | 01:06,6 | 02:16,6 | 03:25,3 | 04:40,4 | 05:52,3 | 07:04,0 |
| log2 | 01:07,4 | 02:19,7 | 03:41,1 | 04:58,8 | 06:25,6 | 07:45,6 |
| log3 | 01:46,6 | 03:35,4 | 05:27,4 | 06:56,8 | 08:43,6 | 10:26,7 |
| log4 | 01:49,6 | 03:33,8 | 05:22,2 | 07:00,9 | 08:48,9 | 10:26,6 |
| log5 | 01:47,3 | 03:28,4 | 05:15,4 | 06:55,3 | 08:48,6 | 10:18,1 |

Figure 6.16: Tree structure execution times for 5 different log files

Even though the two groups of logs have very different characteristics, the tree structure method's performance is equally good in both cases. The small increase in execution time for log 3, 4, and 5 is most likely due to the fact that the average logline lenght is longer in these logs than in the *named* logs. Additionally, only inserted loglines are counted. $log3$, $log4$, and $log5$ all include loglines that are matched by exceptions in the exception file, and ignored. Thus more loglines in the mail logs must be iterated over than in the *named* logs.

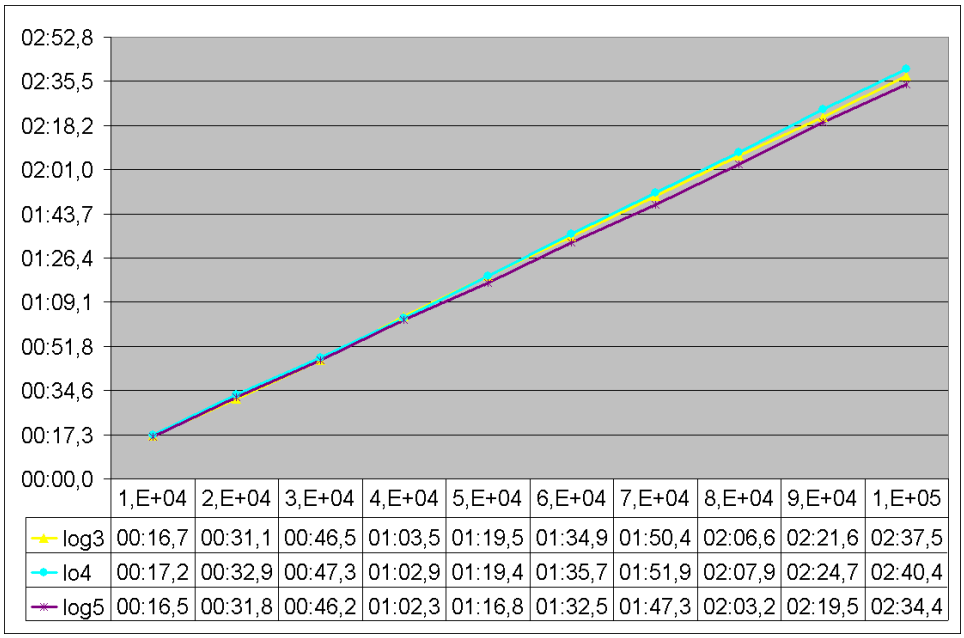
As seen from the chart, the tree structure method has a linear execution time. At 600 000 loglines inserted into the tree structure, the highest execution time is just above 10 minutes.[5]

Figure 6.17 shows a drilldown of the execution times for $log3$, $log4$, and $log5$ between 1000 and 100000 inserted loglines.

The chart in Figure 6.18 illustrates the average time spent on inserting a single logline, measured between 100 and 1 million inserted loglines. Observe that the average time spent on each logline is significantly higher for small values of insertions. The reason for this is that the start up cost of Python constitutes a greater part of the total execution time. As the number of inserted loglines grows, the average time flattens out and stabilises around 12 ns.

The resulting patterns are of high quality. Variables, like usernames and

[5]The tests were run on an Intel Core Duo 1.2GHz laptop with 1.5 GB RAM.




| | 1,E+04 | 2,E+04 | 3,E+04 | 4,E+04 | 5,E+04 | 6,E+04 | 7,E+04 | 8,E+04 | 9,E+04 | 1,E+05 |
|---|---|---|---|---|---|---|---|---|---|---|
| log3 | 00:16,7 | 00:31,1 | 00:46,5 | 01:03,5 | 01:19,5 | 01:34,9 | 01:50,4 | 02:06,6 | 02:21,6 | 02:37,5 |
| lo4 | 00:17,2 | 00:32,9 | 00:47,3 | 01:02,9 | 01:19,4 | 01:35,7 | 01:51,9 | 02:07,9 | 02:24,7 | 02:40,4 |
| log5 | 00:16,5 | 00:31,8 | 00:46,2 | 01:02,3 | 01:16,8 | 01:32,5 | 01:47,3 | 02:03,2 | 02:19,5 | 02:34,4 |

Figure 6.17: Tree structure execution times drilldown.

identifiers, are only present when a logline occurs one single time in the entire log file, or when the logline contains the same variables each time it appears. Thus there is nothing to compare it with, and consequently no way of separating variables from keys.

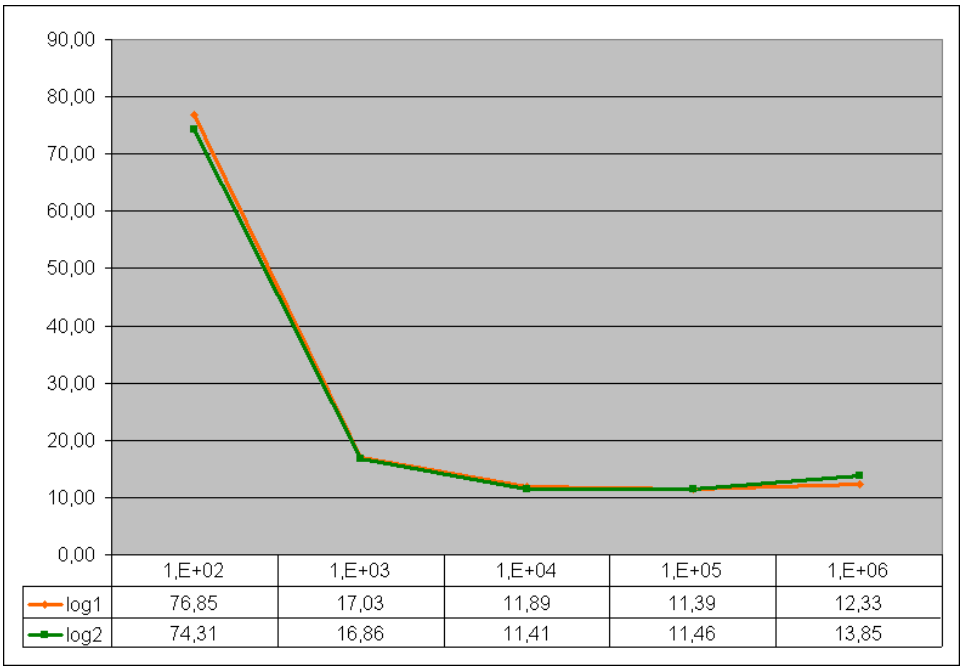
Pattern mining with tree structures were created to conquer some of the issues with the approaches described in Chapter 5.

Recall from Section 5.6 that the most promising approach in terms of pattern quality, *primary sorting*, was also the slowest one.

| **Method** | **CPU time(s)** | **Memory usage(KiB)** |
|---|---|---|
| Histograms | 51,806 | 19 172 |
| Occurrence frequencies | 37,395 | 4 239 |
| Primary sorting | 478,630 | 79 480 |
| Subsets | NA | NA |
| Tree structures | 64,49 | 9 256 |

Table 6.2: Resource usage for the pattern mining approaches

Given the same input on the same hardware as the other approaches already presented, the *tree structures* approach is many times faster than *primary sorting*. Compared to the other approaches, it is still one of the slower ones, but not significantly. When it comes to memory usage, it is one of the best approaches. The results are found in Table 6.5.1




| | 1,E+02 | 1,E+03 | 1,E+04 | 1,E+05 | 1,E+06 |
|---|---|---|---|---|---|
| log1 | 76,85 | 17,03 | 11,89 | 11,39 | 12,33 |
| log2 | 74,31 | 16,86 | 11,41 | 11,46 | 13,85 |

Figure 6.18: Execution timess for inserting a single line.

## 6.6 Conclusion

The *tree structure* approach produces high quality patterns even though its resource use is quite low compared to some of the other approaches. Performance-wise it is significantly faster than the *primary sorting* approach described in Section 5.5 while still producing patterns of the same, or better, quality. Consequently, *tree structures* is the preferred approach to pattern mining.

As mentioned in Section 1.1, a good pattern mining technique is an essential part of doing configuration-less log analysis. It is a prerequisite for doing log line classification since it removes variables that would otherwise clutter the string matching.

By having a working pattern mining technique, it becomes feasible to do some analysis of the log files. By being able to classify log lines, it is possible to discover irregular changes in the number of messages that appears. An example of such analysis is described in Chapter 7.

Other examples of ideas that benefits from patterns are Markov chains for transactions, as described in Chapter 9, and all the log browsing and presentation ideas briefly presented in Chapter 4.

Patterns can also be regarded as a way of creating a summary of a log file. The unique patterns create a coarse summary of the log file content, making it easier to get an overview of the system.

The patterns produced by the *tree structures* approach will be used in the next chapter on statistical analysis.

# Chapter 7

# Statistical analysis

The goal of the statistical analysis approach is to discover abnormalities within a logfile. An abnormality can be that a usually frequent pattern fails to appear, that a pattern has an unusually high or low occurrence frequency, or that a pattern appears for the first time. Such abnormalities can be tokens of an error situation in the systems monitored by the logfile. For example, if there suddenly are very few emails delivered, this can be a token of an overloaded email system. If there is a high increase in the number of failed log in attempts, this can be sign of an attempted break-in. Lastly, if a pattern is seen for the first time, it is likely that an unusual situation has occurred. Imagine that a database server suddenly restarts by itself. The incident-report written to the log will be seen as an unusual pattern, and thus reported as an abnormality.

The pattern mining approach described in Chapter 6 makes it possible to cluster similar loglines, by representing the clusters in form of pattern. Recall that a pattern is a generalisation of similar loglines. The pattern mining approach also presents the frequencies of the different patterns, meaning that it is known how many times a certain type of logline has occured in the log file.

It is possible to make use of the results from the pattern mining approach. On a daily basis, the patterns can be mined from the log and the results stored in a database. By keeping track of the mean value and the standard deviation for each pattern, it is possible to check whether a pattern's frequency deviates too much from the expected value. If it does, the pattern can be used to locate the particular loglines, and these can be written to a report. If a pattern has not been seen before, it too will be reported. In cases where generally frequent patterns fail to appear in the log file, the pattern will be reported as unexpectedly missing.

## 7.1 Periodic changes

A good statistical analyser must be capable of handling natural changes in the input. When it comes to system logs, external factors could significantly influence the amount and type of messages that appears.

In addition to changes introduced by the system administrators when they add, remove or change services, natural changes also occurs. Based on the type of services, there might be significant periodic changes. A service targeted for work environments would naturally be used mostly during the day while services targeted for home users would mostly be used in the evening.

The service targeted for work environments will also be affected by the week cycle. Fewer people work during the weekends. This would lead to fewer messages from those types of services in logs.

In a broader spectrum, season variations will also apply. During holidays such as Christmas and Easter, it should be expected that the service usage pattern is different from a normal day. The hardest deviation to adjust for are statutory holidays, since they may occur on different calendar dates each year.

Some of these periodic changes could be handled. Changes during the day and week, and to some extent during the year, is easy to handle by creating a profile for the actual log system. It is possible to handle Christmas since it is placed on the same calendar date each year. Reports from arbitrary holidays should either be ignored by the system administrators, or the problem could solved by attaching a calendar service to the analyser.

The proof-of-concept code does not take into account periodic changes.

## 7.2 Simple analysis example

Figure 7.1 describes a simplified example of how such a statistical analysis can be performed. Figure 7.1a shows a short list of imaginary patterns and their counts. Figure 7.1b show the same patterns, with the pattern count displayed for each previous observation. In Figure 7.1c some calculations have been made. For each pattern the number of times the pattern has been observed, and its expected count value, $E(X)$, is shown. The last field is the expectancy value of the squared random variable, $E(X^2)$. The random variable $X$ represents the observed pattern's counts, that is, the counts represented in Figure 7.1b.

As illustrated by Equation 7.1, the expected value of $X$ is simply what is commonly known as the *mean*. $x_i$ denotes the i'th observed count, and $N$ is the total number of times the pattern has been observed. $E(X^2)$ is

| pattern | current obs |
|---|---|
| user logged out | 227 |
| unknown user denied access | 9702 |
| connection closed connection timed out | 36 |
| sender notify cannot send message for days | 101 |

(a) Logline patterns with frequencies

| pattern | obs1 | obs2 | obs3 |
|---|---|---|---|
| user logged out | 243 | 232 | 241 |
| unknown user denied access | 352 | 1040 | 782 |
| connection closed connection timed out | 42 | 38 | 45 |
| sender notify cannot send message for days | 121 | 212 | 142 |

(b) Previous frequency observations

| pattern | obs | $E(X)$ | $E(X^2)$ |
|---|---|---|---|
| user logged out | 3 | 239 | 56985 |
| unknown user denied access | 3 | 725 | 605676 |
| connection closed connection timed out | 3 | 42 | 1744 |
| sender notify cannot send message for days | 3 | 158 | 79749 |

(c) Database records

| pattern | $\sigma$ | $3\sigma$ | lower | upper |
|---|---|---|---|---|
| user logged out | 5 | 14 | 225 | 253 |
| unknown user denied access | 284 | 851 | -126 | 1576 |
| connection closed connection timed out | 3 | 9 | 33 | 51 |
| sender notify cannot send message for days | 39 | 117 | 41 | 275 |

(d) Upper and lower limits

Figure 7.1: Example of how a statistical analysis can be performed

calculated similarly, see Equation 7.2

$$E(X) = \mu = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{7.1}$$

$$E(X^2) = \frac{1}{N}\sum_{i=1}^{N} x_i^2 \tag{7.2}$$

To find out how much a pattern's current count deviates from the counts observed in the past, we need to find the random variable $X$'s *standard deviation* (SD). The standard deviation can be easily calculated from the variance of $X$. The variance can be calculated using the formula in Equation 7.3. To achieve the standard deviation we can simply take the square

root of the variance, see Equation 7.4

$$Var(X) = \sigma^2 = E(X^2) - (E(X))^2 \tag{7.3}$$

$$SD = \sqrt{Var(X)} = \sigma \tag{7.4}$$

The standard deviation is a measure for how spread out the observed counts are. If the standard deviation is low, the observed values varies little, while the opposite is true if the standard deviation is high. According to the *empirical rule* 99.7% of the values lie within 3 standard deviations of the mean for normal distributions.

It is not known that the observed data will be normally distributed, but it is assumed that the frequent patterns will eventually cluster about the mean value. Therefore the standard deviation will be used to calculate whether a pattern's count is within the expected range. A pattern's current count $x$ will be reported if it lies beyond the expected value $\pm$ 3 standard deviations ($E(X) \pm \sigma$). The number of standard deviations should be editable to the user.

For each of the patterns in Figure 7.1c the variance is calculated from the values in the two last column, $E(X)$ and $E(X^2)$, according to Equation 7.3. Then Equation 7.4 is applied to produce the standard deviation. Note that the numbers presented in the tables are rounded off upwards. When calculating the standard deviation, more precise numbers are required to achieve correct results.

When comparing the current frequencies found in Figure 7.1a with the upper and lower limits given in Figure 7.1d, it becomes clear that the pattern "unknown user denied access" has a current frequency far beyond what has been observed so far. Consequently this pattern will be reported.

Every pattern not previously observed should be reported as new. It is also desirable to report patterns that in the past has proved to be frequent, but who are not present in the pattern file being analysed. By counting the number of analysed pattern files where the pattern was present, and counting the total number of analysed pattern files, the pattern's occurrence fraction can be calculated. Equation 7.5 shows how this fraction is calculated.

$$\textit{occurrence fraction} = \frac{\textit{number of pattern files where pattern is present}}{\textit{number of pattern files}} \tag{7.5}$$

If the pattern's occurrence fraction is equal to or higher than a given threshold, the pattern is considered frequent.

## 7.3 Implementation

The statistical analysis prototype stores all patterns in an SQLite database [24]. A configuration file is used to specify the path to the database, and the name of the table. If the database does not exist, it is created. So is the table, if it does not already exist. Figure 7.2 shows the create statement for the database. *patterns* is the default name for the database table name.

```
CREATE TABLE patterns (
    logdate DATE,
    pattern TEXT,
    count INTEGER,
    PRIMARY KEY (logdate, pattern)
);
```

Figure 7.2: Database create statement

The prototype begin by first making sure that the logdate does not already exist in the database table. If it does, it is assumed that the patterns have already been inserted, and the program exits.

If the logdate does not exist in the table, the algorithm continues by pruning away old records from the database. How old the records residing in the database are allowed to be, is decided by a variable in the configuration file.

The generated logline patterns are then fetched from file. If the database or table was just created, the patterns are inserted directly into the database table. If no errors occur, the number of patterns inserted into the database is reported.

When the database table already contains data, the generated patterns are compared to the existing patterns in the database. Firstly, the algorithm discovers all *frequent* patterns available in the table, but not in the newly generated patterns. These are reported as missing, whereas every generated pattern not present in the database table are reported as new.

For all patterns present both in the pattern file and the database, further analysis is required. For each of these patterns, the standard deviation is calculated based on all the patterns previously recorded counts, which are stored in the database. A variable in the configuration file specifies how many standard deviations the current pattern count can deviate from the expected value (i.e. the mean value of all the pattern's previous counts ) before it is reported as an irregularity.

The default number of standard deviations is set to 3. Thus, if the current pattern's count is less than the expected value minus three standard deviations, the pattern is reported as underrepresented. If the count is greater than the expected value plus three standard deviations, it is reported as overrepresented.

Finally, the patterns are stored to the database table. If the insertion is successful, the report is generated and stored.

## 7.4 Conclusion

Figure 7.3 shows an example of a report generated by the statistical analysis prototype.

```
12 patterns saved to database.
New:
    58,   dovecot ID mail.info pop3-login Disconnected Inactivity rip
        lip
Overrepresented:
     4762 ( mean:  823.50, sd: 47.02 ) dovecot ID mail.info IMAP
         Disconnected
    20012 ( mean:    8.75, sd:  1.48 ) dovecot ID mail.info auth
        default shadow Password mismatch
Underrepresented:
     1115 ( mean: 1445.00, sd: 41.23 ) dovecot ID mail.info auth
         default new auth connection pid
Missing:  ( mean count)
    63,   dovecot ID mail.info IMAP Connection closed
```

Figure 7.3: Debug output

There is one issue with the statistical analysis and the way it calculates whether a pattern count is within the expected range. If the pattern occurs only once in the database, the pattern will be reported as irregular the next time it is observed, unless it has the exact same value as the pattern already residing in the database. One solution to the problem may be to allow the database a "learning period". Until a pattern has been observed at least $n$ times, it is inserted directly into the database without any analysis performed on it. This measure will assure that the database contains a representative number of data to compare the new values against.

The same problem appears when reporting frequent patterns not present in the pattern file being analysed. Imagine that only one pattern file has been previously analysed, and consequently that only one set of observations is available. If the pattern file currently being analysed do not contain some of these patterns, they will all be reported as frequent. Even if they have occurred only once. The learning period mentioned above should therefore also influence when patterns are considered frequent. database who are not present in the current pattern file being analysed.

This solving of this issue has been left for future work.

# Chapter 8

# Regularity

Another way of measuring the normal state statistically is by looking at when messages arrives. If there are certain messages that arrives at predefined points in time, deviations from these points indicate an anomaly.

An example of messages that should appear regular are those originating from the "cron"[1] daemon. "cron" is a time-based scheduler available for unix operating systems. With "cron", it is possible to schedule tasks to be run at specific points in time; for example, every five minutes.

Deviations from the assumed schedule could indicate an overloaded or not responding server.

The idea is based on domain knowledge of how systems work and which applications are running on them, but regular messages are so common that a way of handling regular messages is necessary. The expected utility of an implementation that keeps an eye on regular messages justifies the assumptions that has to be made.

From here on, the implementation (proof-of-concept) will be referred to as "the module".

## 8.1 Assumptions

The module, called *regularity.py*, takes a few assumptions to make it easier to identify, verify and handle regular messages in log files:

- A training log file will be provided for initially teaching the module which messages are regular.
- A message must appear at least ten times in the training log file to be considered a regular message.

[1] See `http://www.unixgeeks.org/security/newbie/unix/cron-1.html` for an introduction to cron

- The message must not deviate from its schedule in the training log file. It must appear on time throughout the whole file to be added to the regular candidates list.
- A message can not be regular if its content is different each time. Only messages that are the same each time are identified as regular messages. This is fine when handling messages from applications such as "cron", which is an important point of focus for this module.
- Some small variations of the message time stamp is common and do not signal an anomaly. The module accepts messages that are either five seconds early or five seconds late when creating new candidates for regular messages. When monitoring regular messages, the time window is increased to a total of forty seconds to reduce the number of anomalies reported due to time skew or small delays.
- Messages that appear multiple times per minute is considered as noise and are filtered out before they are considered a candidate for regular messages.

## 8.2 Implementation details

The module is created so that it only has to make a single pass of the log file. During the pass, it creates a data structure containing all the messages that appears in the log file. The messages are sorted on the originating host and process so that messages only appear once. The log message is hashed to an MD5 sum since it it easier to work with hashes than text strings.

The module writes two file to disk after the run: a "candidates.txt" file containing all the messages considered as candidates and "msgs.txt" containing a single instance of each log message and its corresponding MD5 hash.

The candidate messages are identified by their host, process and hash. In addition, they have a confidence value and a delta associated with them. The confidence value is used to describe the trust given to the log message. Each message starts with a confidence of *2* when first added. If a message for some reason is reported as an anomaly, the value is decremented. If it reaches zero, the message is considered not to be regular any more and is removed from the list of candidates. If it is reported as an anomaly in a log file, but are regular again in consecutive passes, the confidence value is restored to *2*. The value of *2* is chosen because it provides a trade-off between reduction of the candidate list and the ability for some messages to deviate without being considered as faulty candidates. A higher value would mean that it would take longer time to remove a faulty candidate while a lower value with mean that messages would be removed when a single anomaly occur.

The delta is the interval between consecutive messages from the same process at the same host. It is calculated on the basis of the first three messages that appear in the training log and is saved together with the candidate

in the file. Later messages are compared to the delta value to see if they are on schedule or are deviating too much. If they deviate more than forty seconds (20 seconds before or after the estimated time), an anomaly warning is raised and the message is reported to the user.

If the files "candidates.txt" and "msgs.txt" exist in the current directory together with the log file that should be checked, a single pass is performed to find anomalies. It compares the messages found in the log file with the candidates stored in "candidates.txt". If there are messages that are outside the allowed 40 second window or messages that are missing, they are reported.

After the log file is scanned for anomalies, or if two files are not found, the module goes into learning mode directly and builds a list of messages considered as regular based on the log file given. In learning mode, the scanner does not report anomalies.

## 8.3 Output example

```
Irregularities in the log file( 4):

bitbucket CRON (root) CMD (/usr/libexec/atrun 2>&1 > /dev/null)
    should have occurred at 2009-02-20 09:50:00, however, it occured
    at 2009-02-20 09:55:00

bitbucket CRON (adf) CMD (./sysmetrics/sysmetrics.cron -cron 3 #
    SYSMETRICS) should have occurred at 2009-02-20 13:32:00, however,
    it occured at 2009-02-20 13:37:00

erots /usr/sbin/cron (root) CMD (/etc/cfcron > /dev/null 2>&1 )
    should have occurred at 2009-02-20 03:00:00, however, it occured
    at 2009-02-20 03:15:00

erots /usr/sbin/cron (root) CMD (/usr/sbin/newsyslog)  should have
    occurred at 2009-02-20 03:00:00, however, it occured at 2009-02-20
     04:00:00

Removing faulty candidate 4ba579d19c7c47e49319b940f9ae48ab from CRON
    at bitbucket
```

Figure 8.1: Output from the regularity module

Figure 8.1 displays a report from the module after scanning through a log file. It has identified four irregularities that deviates from the learned pattern in previous log files. In addition, it have removed one candidate that is believed to be faulty since it has been reported as an irregularity on more than one occasion.

The output shows for example that "cron" should have executed */etc/cfcron* at 03:00:00, but that it was delayed until 03:15:00.

## 8.4 Conclusion

The regularity approach works very nice with messages from process schedulers such as the "cron" daemon. Since "cron" sends the same message each time a script or application is executed, it is easy to identify its output in large log files. The module correctly identifies messages that do not appear as expected.

When dynamically creating candidate lists for regular messages, some messages that actually are irregular would be flagged as regular since they appear regular in the log snapshot used for training. As the test module also learn while scanning for irregularities, some messages will be added to the regular messages list even though they actually are not regular.

During testing, messages from DHCP and mail daemons were proven troublesome. The DHCP daemons report every time a client requests an IP address. The requests are somewhat semi-regular since a client will request a renew of the current IP address based on the lease time [25]. But, the daemon reports the same log message if a client requests the same IP address after a reboot or if the user manually requests a renew. The two last scenarios will be reported as an anomaly since they, in most cases, happen at the "wrong" time.

In the case of mail daemons, a number of issues might appear. Some email clients automatically check the server for new messaged every $N$ minutes. By doing so, they generate log messages for each login. These messages appears to be regular but in fact, they are only regular during the current session. If the session contains more than 10 logins, the messages are considered regular as we assume that messages that appears more than 10 times with an equal amount of time between the messages are regular.

Messages that are regular only during an interval are not handled well with our solution. During testing, some messages appeared only between 00:01 and 05:31 with 30 minutes between each message. One approach to solving this problem is to create a list for each candidate message containing all the times where the message should appear during a whole day. This requires that the test log spans at least a whole day (24 hours) to be useful. Solutions to this problem have not been incorporated in this proof-of-concept.

A trade-off between the dynamic nature of the logs and the desire for as little noise as possible has to be negotiated. To negotiate this trade-off, a *confidence* value is used. Due to the dynamic and chaotic nature of system logs, some false positives must be tolerated.

Our implementation of the idea of regular messages will be most valuable if messages from daemons that are known to report on a pre-defined schedule are written to its own log file. This would reduce the number of messages that are mistakenly classified as regular.

# Chapter 9

# Markov models

Markov chains and models, as described in Section 2.4, are ways of describing how the world transition from one state to another. By investigating the past and building a model of the world, it is possible to predict future behaviour. As said in the presentation of the task, Chapter 1), a clever log analyser should be capable of learning which messages are part of a normal state. By looking at historic logs, it is possible to build Markov models that describe how processes interact and the transitions between states in processes.

System logs include enough information to build several different models with increasing levels of complexity. In this chapter, three different ideas will be described: process interaction, severity codes and message flows. A combination of message flows and process interaction is also described as this is the ultimate goal for Markov models when used in log analysis.

From here on, the implementation (proof-of-concept) will be referred to as "the module".

## 9.1 Process interaction

The simplest model to build is the one describing the interaction between processes in the log file. Processes interact constantly, making a pattern that is easily detected when reading through the system logs.

### 9.1.1 The idea

Figure 9.1 illustrates how processes interact during a mail delivery. The process *postfix/smtpd* gets a connection from an outside host and passes the connection info to *postgrey* for host greylisting. When the host is cleared, *postfix/smtpd* accepts the email. In this particular example, a total of eight different processes are involved in receiving an email. In a stable state, the above example demonstrates the norm on how email is received.

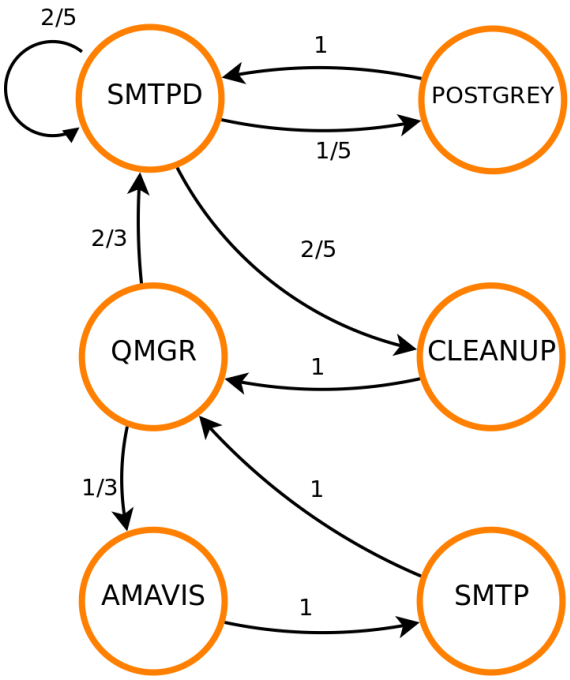



Figure 9.1: Markov model illustrating the information found in Figure A.3

Similar examples are common. Processes doing a specific tasks interact in predefined ways, making patterns in the log files. These patterns should be the same as long as the system is in a stable state, and it should therefore be possible to create a Markov model that correctly describes the interaction between processes. Figure 9.1 is an simple example of a Markov model of the information found in Appendix A.3.

### 9.1.2 Assumptions

- A training log file will be provided to teach the module which processes that are running on the system and the probability for transitions between them.
- All transitions not already known from previous runs are treated as anomalies and reported to the user before being added to the transition database.

### 9.1.3 Implementation details

The module, called *markov.py*, was created in such a way that it only has to look at two messages at the same time. It creates a list of all transitions found in the file with the transition as the key and the number of occurrences as the value. After parsing the log file, the probability is calculated by iterating through all nodes and dividing the transition value to the next process name by the total number of transitions from the current node.

If the module is in learning mode, no output will be provided. The learning mode only learns the probability for a transition from one process to another and creates a dictionary describing these transitions and their probability. This dictionary is used later to compare the old values with the new one.

When not in learning mode, the module creates the same Markov model and compares its transition probability values with the ones calculated from previous log files. Transition probabilities that deviates more than a predefined percentage or transitions that are new are reported back to the user. After they are reported, the dictionary is updated with the new values and saved to disk as "markovstat.txt".

### 9.1.4 Output from the module

```
The transition sshd,in.telnetd is new to us..
The transition in.telnetd,in.telnetd is new to us..
The transition last,sshd differ with more than 40.0% from the previous
     run. Not updating the statistics file with it, this is an
    anomaly!
The transition wtmptail,in.telnetd is new to us..
```

Figure 9.2: Output from the regularity module

Figure 9.2 illustrates output from the module. In this example, three new transitions are found while one known transition probability deviates more than 40% from the historical one.

### 9.1.5 Issues and results

The idea assumes that transitions between processes in log files are fairly stable and follow a common pattern. As it turns out, this is not the case. The chaotic nature of log files, and especially syslog files which might contain messages from a wide range of applications at the same time, makes it impossible to create a useful Markov model.

After the initial implementation, the module was used at some example log files. During testing, the anomaly threshold was adjusted so that the number of anomalies reported was kept to a reasonable amount.

To get useful output, is was discovered that a threshold of at least 40% in both directions was necessary to reduce the number of reported anomalies to modest amount.

The idea of looking at process interaction in log files will probably be useful in log files where the number of processes present is small and the processes are closely interconnected. This is the case where the system administrators have chosen to split log messages into separate files based on the facility code.

In the general case, where a large amount of processes writes their log messages to a common file, the Markov model is susceptible to Markov state explosion. A large amount of states are necessary to describe the world. In addition, since the log files are chaotic by nature, a message from one process could be followed by a message from any other process. This makes it hard to say with a reasonable amount of confidence the possibility for a transition from one process to another.

The state explosion problem was also verified by creation three-way-transitions (from ⇒ including ⇒ to) instead of two-way-transitions (from ⇒ to). As suspected, the number of anomalies detected exploded.

Due to the problems mentioned above, the idea of creating a Markov model describing transitions between processes in log files is useless.

## 9.2 Severity codes

A syslog message contains, as required by RFC3164 [2], a facility and severity code which is combined into a priority value. The facility and severity codes, listed in Appendix A, allows applications to tag the log message with an indication of what kind of service the message originated from and its priority within that particular application.

### 9.2.1 The idea

Since the priority codes are determined by the application sending the message, a Markov model should be restricted to a single application.

Figure 9.3 shows an example of how messages from a process changes priority. In 95 out of 100 messages, an *Info* message follows another *Info* message. In the remaining messages, 4 of the *Info* messages are followed by a *Warning* and 1 leads directly to a message marked as an *Alert*.

In Figure 9.3, *Emergency* messages are in 99 out of a 100 messaged followed by another *Emergency* message. Only 1 out of a 100 messages are followed by an *Info* message, which might indicate that the problem is solved and the system is back to a normal state.

### 9.2.2 Issues and results

The idea behind using severity to identify important log messages requires that the input is consistent and relevant. As Syslog does not have any control over its input, it is up to the creators of applications to decide how they use severity codes for messages.

It turns out that different applications use the codes for different purposes. As there are no consistent way of tagging messages, it is very difficult to

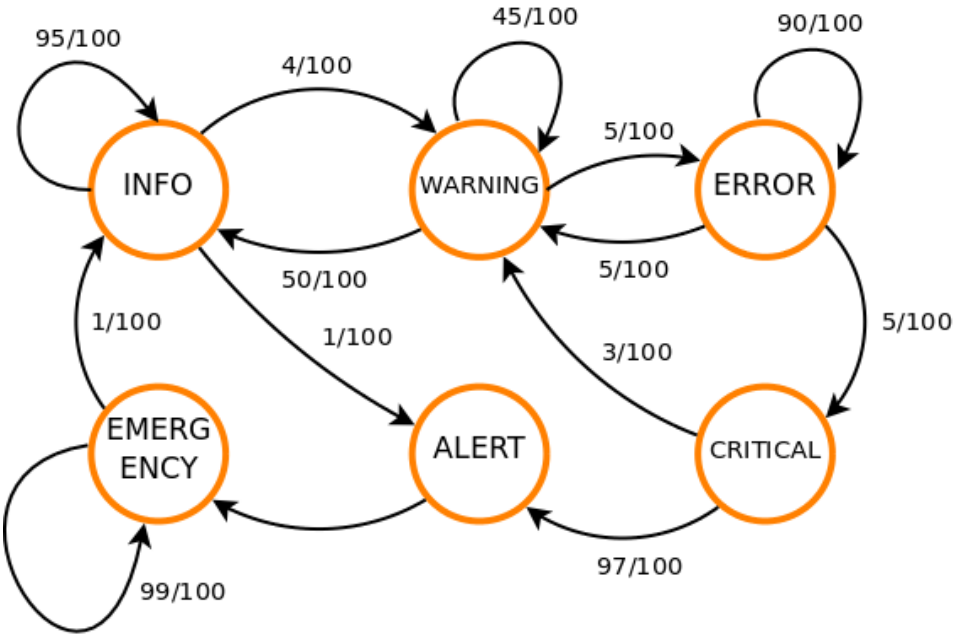



Figure 9.3: Example of how priority codes can be used to create a Markov model stating how messages messages from an application changes priority based on external events

decide if a message is important or not based only on their severity code. If the developers decided to tag messages in a way that does not give the administrators any valuable output, the use of severity as an indicator would provide administrators with a lot of unimportant data.

The following example is from the Apache HTTP daemon, one of the most popular open source applications. The Apache HTTP daemon usually use its own log files, but it is possible to configure it to use Syslog in addition to the daemon specific files.

Apache implements all the severity codes listed in Appendix A.2. According to the Apache manual[1], the severity codes are used as illustrated in Table 9.1.

Now, by looking at Table 9.1, it is clear that the examples for *notice* and *warn* are messages that a system administrator want. It is important to investigate the reasons behind core dumps and a crash.

On the other hand, the manual clearly states that messages with the *error* priority could be "Premature end of script headers". This includes errors such as "404 - File Not Found". A 404 error basically means that the HTTP daemon did not find the file that the user requested. Now, that might be important to the web site developer that might have misspelled a link or

[1]See `http://httpd.apache.org/docs/2.2/mod/core.html#loglevel`

| Level | Description | Example |
|---|---|---|
| *emerg* | Emergencies - system is unusable | "Child cannot open lock file. Exiting" |
| *alert* | Action must be taken immediately | "getpwuid: couldn't determine user name from uid" |
| *crit* | Critical conditions | "socket: Failed to get a socket , exiting child" |
| *error* | Error conditions | "Premature end of script headers" |
| *warn* | Warning conditions | "child process 1234 did not exit, sending another SIGHUP" |
| *notice* | Normal but significant condition | "httpd: caught SIGBUS, attempting to dump core in ..." |
| *info* | Informational | "Server seems busy, (you may need to increase StartServers, or Min/MaxSpareServers)..." |
| *debug* | Debug-level messages | "Opening config file ..." |

Table 9.1: Apache HTTP syslog severity code example

removed the wrong file, but the system administrators should not have to care about such trivial errors. Even so, the Apache HTTP developers have decided to tag the "404 - File Not Found" with a higher priority (*error*) than "httpd: caught SIGBUS, attempting to dump core in ..." (*notice*).

The example above illustrates the difficulty of handling input that you have no control over. Choices made by one group of users might turn out to be a bad design strategy for another.

The Markov model should be capable of handling such design issues since it only looks at the transition probabilities and not the messages itself. As the "404 - File Not Found" message could be quite frequent, transitions to the *error* state should be common. On the other hand, there is a possibility for the developers to tag all their log messages with just one severity code, making it impossible to separate messages.

Another problem with using severity as an indicator is the fact that the specification for Syslog, RFC3164[2], does only specify that the priority code (the facility and severity code combined) should be transmitted on the wire. It does not specify that the code should be saved in the actual log files. It is up to the developers of the syslog daemons to decide how the messages are stored. As it turns out, a number of syslog implementations does not store the priority code together with the log message.

Based on the discovery of the mentioned issues, the Markov model that use priority as states is not implemented nor actually tested. The reason is that the expected output is suspected to be of too little value for system administrators, if the input values exists at all in the log files.

## 9.3 Markov models for message flows

Instead of looking at process interaction or severity codes in the syslog messages, focus is now changed to internal message flow in a single process.

### 9.3.1 The idea

By removing variables from log messages, it is possible to classify messages on-the-fly. This makes it possible to anticipate log messages within a single application and to detect anomalies by actually parsing the log messages. Removing variables dynamically is a complex task, requiring a fair amount of historical data, but the same data could be used for training the Markov model. A fairly complete Markov model based on messages from applications would easily detect a deviation from the usual pattern.

The pattern matching described in Chapters 5 and 6 is the foundation for this idea. Log messages usually consists of some key words and a wide range of variables. By identifying the variables and focusing on the key words, patterns emerge. These patterns could be used to describe how the operation of a process is by looking at how messages appear in the log files.


Figure 9.4: Illustration of the normal flow of messages from postfix/smtpd in the example log file found in Appendix A.3

Figure 9.4 is a very simple example of how messages from *postfix/smptd* appear in the log file example found in Appendix A.3. The nodes are different patterns that occur in the log file. After some initial training, the Markov model should provide a view of the stable state.

The idea is interesting since it focuses on message flow and how programs are working internally instead of focusing on just single log messages. With this approach, transactions will be the area of focus. Deviations from a firmly established transaction pattern are a strong indication of an anomaly.

To be able to identify message flow patterns, some pre-requisites must be met:

- A fully working pattern mining technique must be in place. To be able to detect message flows, the input must be a string where all variables are removed.
- The module need some way of identifying running processes. It is not enough to know the process name, it also need the process identification number (PID) to be able to separate different instances of the

same process. This is important as an infinite number of instances of the same process might be in different states at the same time.
- A significant amount of historical data containing a fairly complete picture of a stable state must be present. The historical data is necessary for creating the initial Markov model.
- The historical data must not contain a large amount of messages from a state where the process was not working as the error state then will be considered part of the stable, working state.

### 9.3.2 Issues and results

Due to time constraints, this idea was never implemented. Even so, it is believed that this method is one of the most promising techniques for identifying anomalies in log files.

A process that behaves normally will follow the same pattern every time it is launched. Although there might be some variations due to changes in the input and the current system states, it should be possible to create a model that describes the message flow based on historical data.

Based on the model of the stable state, it is possible to say something about which messages are regular and which are not. Non-regular messages, new messages and message transitions that are uncommon should be treated as anomalies and reported to the user. Event though this would mean that changes to the system, for example if a new process is introduced to the system, are reported as anomalies in the beginning, the output from the module should be valuable to the administrators.

Creation of the model is highly dependent upon a working pattern mining technique. Without it, a state explosion will occur due to the number of variables (IP and email addresses, host names, file names etc.) in log messages. If a state explosion occurs, both the performance and the quality of the output will be degraded significantly. Without patterns that are close to free from variables, it will be significantly harder to determine the message flow patterns due to the extra transitions that will occur. For example, instead of having one state stating that an email has been received, it could end up with 10 000 different states which only differ on the sender and receiver address in the email.

## 9.4 Combining process interaction and message flow

Markov models for process interaction and message flows only solve part of the problem. System logs usually describes a vaguely organized ballet between a significant amount of processes and possible also hosts. The system log example given in Figure A.3 illustrates how an email is delivered from an external host to a local user on the given host. Processes interact

with each other and send log messages to describe what they are doing.

Constructing such a model is a very complex task. Not only is it required with a very good pattern mining technique to remove all variables from the log messages, it is also necessary with a technique that can identify transactions dynamically. As the task given states that the system should require as little configuration as possible, it is not an option to create a high level description of common transactions by hand.

Currently, only the pattern mining problem is, to some extent, solved, as described in Chapter 6. The identification of transactions requires a lot of research and has not been a point of focus.

## 9.5 Common problems with Markov models

The transition probability is an important property of a Markov chain or model. The analyser created should be capable of learning the transitions by itself, making it necessary to recalculate the transition probabilities frequently to compensate for natural changes.

A complicating factor is the fact that the model will have no idea about how the world is put together. Even though the model might have access to a large amount of historical data, it could never know if it has a complete world or not. The only case where it is possible to create a complete, finished model beforehand is when using *Facility* and *Severity code* as in 9.2). This because these codes are defined in RFC3164 [2].

When working with process interaction (Section 9.1.1) and actual log messages (Section 9.3), there is always a possibility for new messages and processes to show up. Markov specifies that the Markov model should be created by identifying all possible states in the world and then identify transitions and their transition probability (Section 2.4). Even so, the Hidden Markov Model (HMM) algorithm allows for hidden parameters in the observable data as long as it is assumed that the process behind is a Markov process. The dynamical world makes it complicated to reduce the number of false positives.

One way of handling cases where a new state shows up is to mark is an anomaly and report back to the user if one shows up. When recalculating the new probabilities on basis of the old, historical data and the new log file, all new states could be added in the same process, creating a more complete picture of the world. The drawback with such a solution is that there will be a significant amount of messages reported to the users in the beginning. As the model gets a more complete view of the world, it should stabilise until major changes are introduced to the system. When such changes occur, the problem will reappear.

## 9.6 Conclusion

The Markov idea consists of three different ways of constructing the Markov models: Process interaction, severity codes and log message patterns to predict message flow. The ideas differ in complexity, from the most simple ones concerning process interaction and severity codes, to the fairly complex message flow model.

Due to the nature of system logs, the process interaction idea failed to deliver the anticipated results. Its main problem is that it only looks at process names without taking into consideration that several processes might be active at the same time, and some of them might be different instances of the same application. Since a multitude of processes are active at the same time, and at least some of them are govern by external events, the output will be in random order. The implementation of this idea proved that at least a 40% variation in both ways must be allowed to get the number of reported anomalies down to a reasonable level. A threshold of 80% is too large to provide any meaningful results.

The next idea use the severity codes that all log messages to Syslog are required to have. The idea is quite simple and elegant: given that the application developers use the severity codes carefully, it is possible to select the severity codes that are rare, or transitions from one state to another that are not very common.

Although it is not very difficult to create models based on the severity code, the idea was terminated before being implemented due to two reasons. The first being is that application developers does not use the codes in a consistent way. As the Apache example illustrates, knowledge about the specific application is usually required for interpreting severity codes. The second reason is that the Syslog only requires that the severity code is transmitted over the wire. As it turns out, the severity codes are usually never written to disk.

The last idea is the most promising one given the technology today. By looking at the actual messages from an application, it is possible to generate a model that illustrates normal program flow, or transactions. The idea incorporates the idea of transaction processing instead of looking at separate values. Although the complexity is much higher than for the two other ideas, the value of the output should justify the added complexity. That being said, the idea was never materialised into code due to the lack of a working pattern mining technique.

The ultimate solution with Markov models is the combination of process interaction with message flows. Together, these two models will give a good description of transactions rather than individual messages. As processes interact all the time, a Markov model should not only look at message flows within a single process, but also between processes and possible between different hosts. Though, given the techniques described in preceding chapters,

there are currently no way of creating this model without manually writing a high level description of the transactions in a configuration file.

# Part III

# Conclusion and future work

# Chapter 10

# Conclusion

The main objective for this thesis was to test out methods for doing log analysis without relying on regular expressions. In the end, these methods should be used to create an configuration-less log analyser capable of handling a wide range of log files.

The task of creating such an analyser was commenced by another project in 2005 by a group attending the course "IT2901 - Informatics Project II". Their project was poorly documented and their final results are not stated in their report. It was therefore necessary to revisit some of the approaches they proposed and possibly extend them.

The task of mining patterns from system logs was, and still is, one of the biggest challenges in configuration-less log analysis. Pattern mining is a prerequisite for a lot of analysis methods and algorithms that could be used to analyse system logs.

Algorithms such as Bayesian filtering, Apriori and decision learning trees were found unsuitable. After a study of Risto Vaarandi's tools, the *Simple Logfile Clustering Tool* and *LogHound*, it became clear that a new method for mining patterns was required.

Inspired by the Apriori algorithm, item occurrence frequencies and item subsets were implemented and tested. In both attempts, the results were unsatisfactory. Histograms and primary sorting proved somewhat promising, but not sufficient.

The most promising pattern mining technique use tree structures. By building a tree with all the words in a log message as nodes, it became possible to identify variables and remove them from the tree. After some modification processes, the tree contains patterns that are free of variables given that there existed a sufficient amount of messages to begin with.

By having a proposed solution to the pattern mining issue makes it possible to do statistical analysis on log files. By gathering historic data on previously generated patterns, service failures or abnormal use of the system can be

detected.

Only one approach using Markov models was implemented and tested, and found unusable. The same result was found for another idea using the *Facility* and *Severity* code from the Syslog specification. The last idea presented, where models are created to describe how transactions occurs, are the most promising of all the Markov model ideas. But, it is a fairly complex idea and requires much research on how to identify transactions run-time without any prior knowledge of the system at hand.

Configuration-less log analysis have proven to be difficult. The lack of a clear and concise specification is prominent, making it very hard to do automatic analysis. Pattern mining suffers particularly from the lack of rules regarding the format of the log message.

Regular expressions are guaranteed to provide results since they specify exactly what the output should be. The drawbacks are the need of specific knowledge about the system at hand and the potential for an explosion in the number of rules necessary to get the wanted output. One idea suggested in this report specifies a method for distributing the job of creating regular expressions to a community of users. The online repository and the analyser using it is not a true configuration-less log analyser, but it should make it easier for administrators to perform their analysis.

One major drawback of regular expression is that messages that are forgotten, or are unknown at the time when the regular expression was written, will be ignored during the analysis. Here, the configuration-less approach has a major benefit. It does not care about the semantics of the message. Instead, it identifies the message as one that is uncommon and/or does not conform to the historical behaviour of a service.

Regular expressions and pattern mining shares a common goal: to be able to identify and classify text strings based on keywords. Pattern mining generates these patterns automatically, while regular expressions are initially written by hand. Combining the two approaches is possible by using the generated patterns as a foundation and improving them manually. Such a solution is easier to maintain since the regular expressions are to a great extent generated automatically based on log files from similar systems..

Configuration-less log analysis have too many unresolved challenges to be fully useful today. A combination of regular expressions and configuration-less log analysis might be the best solution for log analysis. Combining the power of regular expressions with fuzzy search will identify both known and previously unknown issues.

# Chapter 11

# Future work

This chapter offers our take on what should be done for configuration-less analysis to become successful.

## 11.1 Standardising log messages

The specification of Syslog, RFC3164 [2], was written a long time after the actual development of the protocol. The RFC cites that "*This document describes the observed behavior of the syslog protocol*". This affects how efficient and clean the implementations are. Most of the log messages are nothing more than free-form text messages with some specified fields as illustrated in Appendix A. Free-form text messages are hard to handle, making it difficult to evaluate and take proper action based on its content. This is one of the greater challenges with respect to pattern mining.

A more standardised log format would benefit the users. A stricter specification when it comes to codes could make it easier to detect anomalies. This especially applies to the *Severity* and *Facility* codes in Syslog messages. Section 9.2 illustrates how difficult it is to trust the codes reported by the applications.

A new log message standard is necessary to be able to handle messages efficiently, without knowledge of the specific application in question. The objective is to be able to make high-level assumptions about the log lines and its content. Such a standard must be applicable to a wide range of devices, operating systems and applications, such that one analyser could be used on logs from all units.

Currently, there exists an ongoing process of rewriting the Syslog standard. The new standard, RFC5424 [26] and its amendments, separates the Syslog messages from its transport protocol, making it possible to use any number of transport protocols for transmission of Syslog messages. RFC5426 [26] defines the new Syslog standard, RFC5425 [27] defines TLS transport map-

ping, and RFC5426 [28] defines UDP transmission. RFC5427 [29] concerns textual conventions for Syslog management. Besides separating transport protocols and Syslog message specification, the new RFC brings some new, mandatory fields to Syslog messages. The free-form MSG field, which has been proven difficult to handle, is still present in the new standard.

As the drafts are currently on the track for becoming an published RFC, this report have not taken the changes into account. There are currently no implementation of this draft, making it difficult to evaluate the changes from the former standard. The changes in RFC5424 should be evaluated with focus on automatic analysis.

Lastly it must be mentioned that regardless of the improvements introduced by a new Syslog standard, the quality of log messages will never improve until application creators prioritise writing structured, and well-formed messages.

## 11.2 Pattern mining

The pattern mining approaches described in this thesis have not been evaluated on a broad spectrum of log files. It may still contain some sharp edges where some log files might trigger unpredicted behaviour or errors in the code. Performance tweaking have neither been an area of focus during the development of the methods. Both issues should be carried out before the techniques are put into production.

Given that a working pattern mining technique is present, a number of options appears. This thesis have already described some of the areas were patterns could be used: statistical analysis and Markov modelling of process transactions. Other ideas have also been briefly mentioned.

There are certainly other areas where patterns and pattern mining could be used. One handy, side effect feature of the resulting patterns is that the patterns mined from the logs provides a brief summary of the log file content. This content could be utilise in a log browser to make it easier to browse through logs manually.

## 11.3 Neural networks

Section 4.2 described one way of utilising neural networks: combining a number of tests and use neural network to score the results.

There might be other ways of utilising neural networks in log analysis. One idea could be to build a network capable of classifying messages based on previous examples and training input. Another idea might be to train the neural network on what is considered an anomaly and make it detect new anomalies in later log files.

Neural networks are complex pieces of software. As the authors have little or no previous experience with artificial intelligence, and neural networks in particular, use of neural networks have not been an area of great focus. The of neural networks in system log analysis might be a separate master's thesis for someone interested in artificial intelligence.

## 11.4 Markov models

The ultimate goal for the Markov models is to describe the whole log file in general, high level terms. By combining the message flow model and the process interaction model, a new model capable of describing how processes collaborate appears. As said in Section 9.4, the creation of such a model is highly complex. In order to describe the process interaction properly, it is necessary to create some sort of transaction description dynamically. Currently, there are no way of doing this. One solution might be to create transaction descriptions for some of the most common transactions by hand and then use these descriptions to create the model and generate the transition probabilities.

# Bibliography


[1] Audun Hoel, Johan Gudheim Hansen, Andres Burø, and Øyvin Halfdan Thuv. Sluttrapport IT2901 Gruppe 12, 2005. Report submitted as part of the IT2901 course at IDI/NTNU.

[2] C. Lonvick. *The BSD Syslog Protocol*, 2001. RFC 3164.

[3] Tom M. Mitchell. *Machine Learning*, chapter 6, pages 154–200 (The McGraw Hill Companies Inc.), 1997. ISBN 0-07-042807-7.

[4] Mehran Sahami, Susan Dumais, David Heckerman, and Eric Horvitz. A Bayesian Approach to Filtering Junk E-mail. *AAAI Workshop on Learning for Text Categorization*, July 198.

[5] Rakesh Agrawal and Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499 (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA), 1994.

[6] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to data mining*, chapter 6 (Pearson Addison Wesley), 2006.

[7] Risto Vaarandi. A data clustering algorithm for mining patterns from event logs. *IP Operations and Management, 2003. (IPOM 2003). 3rd IEEE Workshop on*, pages 119–126, October 2003.

[8] Tom M. Mitchell. *Machine Learning*, pages 52–80 (The McGraw Hill Companies Inc.), 1997. ISBN 0-07-042807-7.

[9] Christopher M. Bishop. *Pattern Recognition and Machine Learning*, pages 610–635 (Springer Science+Business Media, LLC). ISBN 978-0387-31073-2.

[10] Tom M. Mitchell. *Machine Learning*, pages 81–127 (The McGraw Hill Companies Inc.), 1997. ISBN 0-07-042807-7.

[11] Ben Kröse and Patric van der Smagt. An Introduction to Neural Networks. 1996.

[12] Barak A. Pearlmutter. Dynamic recurrent neural networks. Technical report, School of Computer Science, Carnegie Mellon University, 1990.

[13] Leslie Pack Kaelbling and Andrew W. Moore. Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research 4*, pages 237–285, 1996.

[14] F. Rosenblatt. *Principles of Neurodynamics*, pages 23, 26 (Spartan Books), 1959.

[15] Harold S. Javitz and Alfonso Valdes. *The NIDES Statistical Component Description and Justification*, 1994.

[16] Risto Vaarandi. SEC - a lightweight event correlation tool. *IP Operations and Management, 2002 IEEE Workshop on*, pages 111–115, 2002.

[17] A Breadth-First Algorithm for Mining Frequent Patterns from Event Logs. In *Intelligence in Communication Systems*, volume Volume 3283/2004 of *Lecture Notes in Computer Science*, pages 293–308 (Springer Berlin / Heidelberg), 2004.

[18] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 1–12 (ACM, New York, NY, USA), 2000.

[19] M.J. Zaki. Scalable algorithms for association mining. *Knowledge and Data Engineering, IEEE Transactions on*, 12(3):372–390, May/Jun 2000.

[20] Tetsuji Takada and Hideki Koike. Mielog: A highly interactive visual log browser using information visualization and statistical analysis. In *In Proc. USENIX Conf. on System Administration*, pages 133–144. 2002.

[21] Tetsuji Takada and Hideki Koike. MieLog: A Highly Interactive Visual Log Browser Using Information Visualization and Statistical Analysis. LISA 2002 presentation, November 2002.

[22] LogRhythm. LogRhythm Integrated Solution - Log Management, Log Analysis and Event Management, 2008. URL `http://logrhythm.com/Products/LogRhythmIntegratedSolution.aspx`. Accessed on February 23rd.

[23] David Pérez-Gonzáles, Manuel S. Malmierca, and Ellen Covey. Novelty detector neurons in the mammalian auditory midbrain. *European Journal Of Neuroscience*, 22:2879–2885, 2005.

[24] SQLite. URL `http://www.sqlite.org`.

[25] R. Droms. RFC 2131 - Dynamic Host Configuration Protocol. 1997.

[26] R. Gerhards. *The Syslog Protocol*, 2009. Draft for RFC 5424, obsoletes RFC 3154 when done.

[27] Ed. F. Miao, Ed. Y. Ma, and Ed. J. Salowey. *Transport Layer Security (TLS) Transport Mapping for Syslog*, 2009. Draft for RFC 5425.

[28] A. Okmianski. *Transmission of Syslog Messages over UDP*, 2009. Draft for RFC 5426.

[29] G. Keeni. *Textual Conventions for Syslog Management*, 2009. Draft for RFC 5427.

# Appendix A

# The BSD Syslog Protocol

The BSD Syslog Protocol was initially implemented in logging applications on UNIX systems. The protocol is capable of both saving messages locally and to forward them to a central logging host over a network [2]. There also exists Syslog implementations for the Windows operating system[1].

A Syslog message consists of three parts: priority *PRI*), a header (*HEADER*) and the message from the application (*MSG*). If a device sends a message that does not conform to the standard, a intermediate host (the relay) has to add the missing parts before forwarding the message to the log host. Details on how this is done is given in RFC3164 [2].

The total length of the Syslog message is restricted to a minimum of 0 bytes and a maximum of 1024 bytes.

A syslog message conforming to the specification should follow the format *<PRI>HEADER MSG*. Figure A.1 shows a sample logline, and illustrates how the logline is split into the three parts in accordance with the format specification.

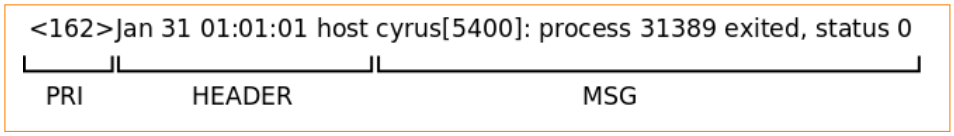



Figure A.1: Illustration of the Syslog format

The priority part, *PRI*, is always enclosed by leading and trailing angle brackets, <>. It contains a priority value which is a combination of a *Facility code* and a *Severity code*. Section A describes *Facility* and *Severity* codes in more details. The maximum length of the priority part is five characters.

The header contains a time stamp and an indication of the hostname or IP

[1]Examples are the "Kiwi Syslog Server" available from `http://www.kiwisyslog.com` and WinSyslog from `http://www.winsyslog.com`.

address of the device sending the message. The required format is "Mmm dd hh:mm:ss" where "Mmm" is the month as abbreviated in the English language, "dd" is the day of the month. The hostname may either be the hostname or the IP address of the device sending the message. It must not contain the fully-qualified domain name.

The MSG part contains the actual message. There is no restrictions on the content besides that it must contain visible characters. It consists of a *TAG* field and a *CONTENT* field. The tag field contains the name of the program or process that generated the message while the content field contains the details of the message. The content field is generally a free-form message, as illustrated in the example above. Figure A.2 illustrates the tag and content fields of the Syslog message part.

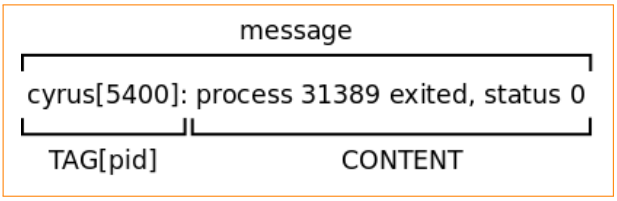



Figure A.2: Illustration of the Syslog MSG part

Generally, if the device sending the message has any concept of processes, it has been considered good practice to say something about the process that generated the message. A common way of doing this is by adding additional information in the beginning of the content field. The format "TAG[*pid*]", where *pid* is the process identifier, is common but not required.

The content of the Syslog message is decided by the application sending it, as long as it conforms to the standard. If a Syslog server receives a message that is not conforming to the standard, it should append the missing parts to the message before further action is taken.

## A.1 Facility codes

Syslog related codes as specified in RFC3164 [2].

| Number | Facility |
|---|---|
| 0 | kernel messages |
| 1 | user-level messages |
| 2 | mail system |
| 3 | system daemons |
| 4 | security / authorization messages |
| 5 | messages generated internally by syslogd |
| 6 | line printer subsystem |
| 7 | network news subsystem |
| 8 | UUCP subsystem |

| Number | Facility |
|---|---|
| 9 | clock daemon |
| 10 | security / authorization messages |
| 11 | FTP daemon |
| 12 | NTP subsystem |
| 13 | log audit |
| 14 | log alert |
| 15 | clock daemon |
| 16 | local use 0 (local0) |
| 17 | local use 1 (local1) |
| 18 | local use 2 (local2) |
| 19 | local use 3 (local3) |
| 20 | local use 4 (local4) |
| 21 | local use 5 (local5) |
| 22 | local use 6 (local6) |
| 23 | local use 7 (local7) |

Table A.1: Syslog facility values

## A.2 Severity codes

| Code | Priority |
|---|---|
| 0 | Emergency: system is unusable |
| 1 | Alert: action must be taken immediately |
| 2 | Critical: critical conditions |
| 3 | Error: error conditions |
| 4 | Warning: warning conditions |
| 5 | Notice: normal but significant condition |
| 6 | Informational: informational messages |
| 7 | Debug: debug-level messages |

Table A.2: Syslog message severity codes

## A.3 Log file examples

```
Apr 28 10:27:43 paragon postfix/smtpd[23806]: connect from pil.idi.ntnu.no[129.241.107.93]

Apr 28 10:27:46 paragon postgrey[2621]: action=pass, reason=client whitelist, client_name=pil.idi.ntnu.no,
    client_address=129.241.107.93, sender=tdt4285-gr1-bounces@idi.ntnu.no, recipient=frode@sandholtbraaten.com

Apr 28 10:27:46 paragon postfix/smtpd[23806]: 9731CC51702: client=pil.idi.ntnu.no[129.241.107.93]

Apr 28 10:27:46 paragon postfix/cleanup[23809]: 9731CC51702: message-id=<DD432DF2-97F8-4E05-80C2-A50A1E93E9D5@gmail.com>

Apr 28 10:27:46 paragon postfix/qmgr[20761]: 9731CC51702: from=<tdt4285-gr1-bounces@idi.ntnu.no>, size=7836, nrcpt=1 (queue
     active)

Apr 28 10:27:46 paragon postfix/smtpd[23806]: disconnect from pil.idi.ntnu.no[129.241.107.93]

Apr 28 10:27:47 paragon postfix/smtpd[23814]: connect from localhost[127.0.0.1]

Apr 28 10:27:47 paragon postfix/smtpd[23814]: 6F4E7C61464: client=localhost[127.0.0.1]

Apr 28 10:27:47 paragon postfix/cleanup[23809]: 6F4E7C61464: message-id=<DD432DF2-97F8-4E05-80C2-A50A1E93E9D5@gmail.com>

Apr 28 10:27:47 paragon postfix/qmgr[20761]: 6F4E7C61464: from=<tdt4285-gr1-bounces@idi.ntnu.no>, size=8556, nrcpt=1 (queue
     active)

Apr 28 10:27:47 paragon amavis[26245]: (26245-11) Passed CLEAN, [129.241.107.93] [84.48.52.88]
    <tdt4285-gr1-bounces@idi.ntnu.no> -> <frode.sandholtbraaten.com@mail.frode.biz>, Message-ID:
    <DD432DF2-97F8-4E05-80C2-A50A1E93E9D5@gmail.com>, mail_id: VHn39sfS7Wlm, Hits: 0, size: 7835, queued_as: 6F4E7C61464,
    838 ms

Apr 28 10:27:47 paragon postfix/smtp[23810]: 9731CC51702: to=<frode.sandholtbraaten.com@mail.frode.biz>,
    orig_to=<frode@sandholtbraaten.com>, relay=127.0.0.1[127.0.0.1]:10024, delay=3.8, delays=2.9/0.02/0.01/0.84, dsn=2.0.0,
     status=sent (250 2.0.0 Ok, id=26245-11, from MTA([127.0.0.1]:10025): 250 2.0.0 Ok: queued as 6F4E7C61464)

Apr 28 10:27:47 paragon postfix/qmgr[20761]: 9731CC51702: removed

Apr 28 10:27:47 paragon postfix/smtpd[23814]: disconnect from localhost[127.0.0.1]
```

Figure A.3: An example illustrating process interaction in system logs

```
Feb 17 03:10:12 <mail.info> eik dovecot: [ID 107833 mail.info] auth(default): client in: AUTH 1 PLAIN service=IMAP secured
    lip=176.16.145.15 rip=176.16.15.82 resp=<hidden>

Feb 17 03:10:12 <mail.info> eik dovecot: [ID 107833 mail.info] auth(default): client out: OK 1 user=jane

Feb 17 03:10:12 <mail.info> eik dovecot: [ID 107833 mail.info] auth(default): master out: USER 249094 jane
    system_user=jane      uid=41024 gid=13731      home=/home/b/6/jane

Feb 17 03:10:12 <mail.info> eik dovecot: [ID 107833 mail.info] imap-login: Login: user=<jane>, method=PLAIN,
    rip=176.16.15.82, lip=176.16.108.15, TLS

Feb 17 03:10:12 <mail.info> eik dovecot: [ID 107833 mail.info] IMAP(jane): Disconnected: Logged out

Feb 17 03:10:12 <mail.info> eik dovecot: [ID 107833 mail.info] auth(default): new auth connection: pid=919

Feb 17 02:55:18 <mail.info> pisa clamd[65913]: SelfCheck: Database status OK.

Feb 17 02:55:22 <mail.info> pisa mimedefang-multiplexor[16385]: Killing idle slave 2 (pid 62642): Slave has processed 500
    requests

Feb 17 03:10:17 <mail.info> eik dovecot: [ID 107833 mail.info] auth(default): client in: AUTH 1 PLAIN service=IMAP secured
    lip=176.16.108.15 rip=176.241.22.165 resp=<hidden>

Feb 17 03:10:17 <mail.info> eik dovecot: [ID 107833 mail.info] auth(default): client out: OK 1 user=john

Feb 17 03:10:17 <mail.info> eik dovecot: [ID 107833 mail.info] auth(default): master out: USER 249095 john
    system_user=john      uid=22024 gid=13731      home=/home/b/11/john

Feb 17 03:10:17 <mail.info> eik dovecot: [ID 107833 mail.info] imap-login: Login: user=<john>, method=PLAIN,
    rip=176.241.22.165, lip=176.16.108.15, TLS

Feb 17 03:10:17 <mail.info> eik dovecot: [ID 107833 mail.info] IMAP(john): Disconnected: Logged out

Feb 17 03:10:18 <mail.info> eik dovecot: [ID 107833 mail.info] auth(default): client in: AUTH 1 PLAIN service=IMAP secured
    lip=176.16.108.15 rip=176.241.22.165 resp=<hidden>
```

Figure A.4: Example loglines from Syslog

# Appendix B

# Tree Structure approach results and examples

This appendix supplies some of the results from the tree structure approach in Chapter 6.

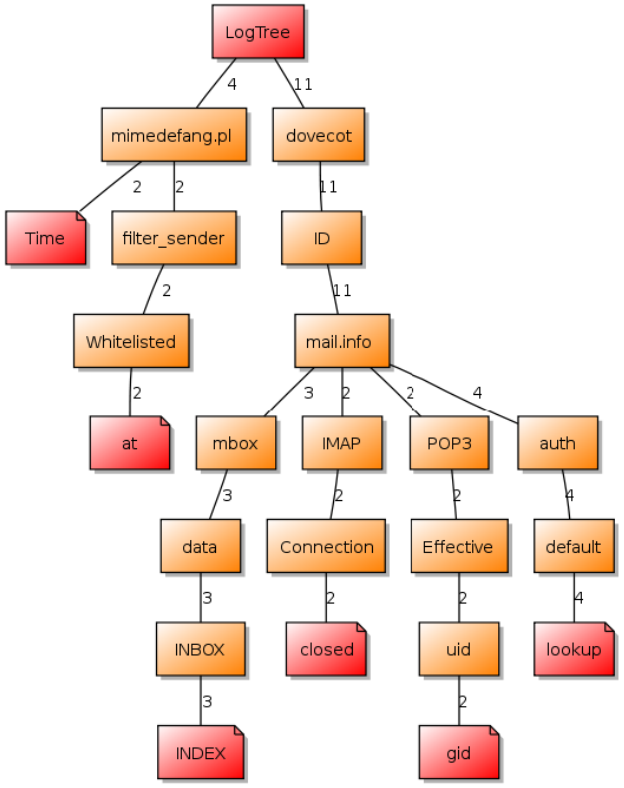



Figure B.1: Graphical representation of the illustrative loglines from Figure B.3, after branches have been merged.

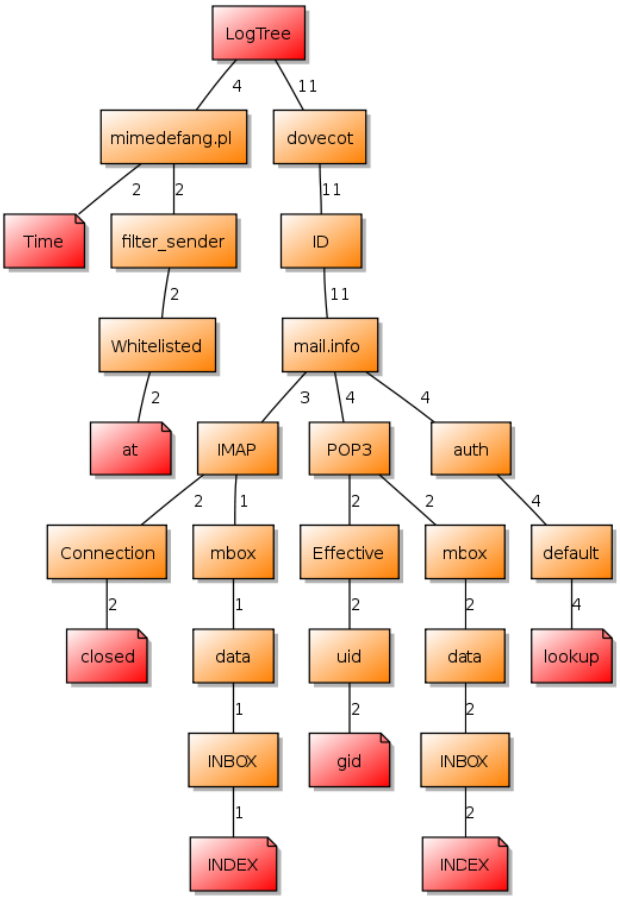



Figure B.2: Graphical representation of the illustrative loglines from Figure B.3, after hidden labels have been released. (Ergo the final result.) Compare with the tree in Figure B.1. Notice how the branch [*mbox data INBOX INDEX*] have been split into two branches. This is the result of releasing hidden labels $IMAP$ and $POP3$ from node $mail.info$. Note that the sum of the two branches still adds up to three. The loglines in Figure B.3 can be inspected to verify that there are three loglines containing [*mbox data INBOX INDEX*], and that two of them includes $POP3$, while the last contains $IMAP$.

```
Feb 19 23:44:50 <mail.info> cat mimedefang.pl[163]: n1JN00: Time=1

Feb 20 00:06:15 <mail.info> fox dovecot: [ID 105 mail.info] POP3(john): mbox:
    data=/home/john/mail/:INBOX=/var/mail/john:INDEX=/srv/dovecot/var/indexes/john

Feb 20 00:03:07 <mail.info> fox dovecot: [ID 106 mail.info] IMAP(jane): Connection closed

Feb 20 00:04:20 <mail.info> fox dovecot: [ID 107 mail.info] POP3(paul): Effective uid=1404, gid=13730

Feb 19 23:44:50 <mail.info> cat mimedefang.pl[16385]: n1JN02: Time=1

Feb 20 00:04:57 <mail.info> fox dovecot: [ID 108 mail.info] POP3(john): Effective uid=15956, gid=13730

Feb 20 00:17:57 <mail.info> eik dovecot: [ID 113 mail.info] auth(default): shadow(bill,124.231.107.91): lookup

Feb 20 00:17:57 <mail.info> eik dovecot: [ID 114 mail.info] auth(default): passwd(bill,124.231.107.91): lookup

Feb 20 00:05:10 <mail.info> fox dovecot: [ID 109 mail.info] POP3(ben): mbox:
    data=/home/ben/mail/:INBOX=/var/mail/ben:INDEX=/srv/dovecot/var/indexes/ben

Feb 20 00:06:32 <mail.info> fox dovecot: [ID 110 mail.info] IMAP(bob): mbox:
    data=/home/bob/mail/:INBOX=/var/mail/bob:INDEX=/srv/dovecot/var/indexes/bob

Feb 19 23:51:25 <mail.info> cat mimedefang.pl[164]: filter_sender: Whitelisted, <bob@example.com> at [124.222.143.106]
    [124.232.143.106]

Feb 19 23:51:26 <mail.info> cat mimedefang.pl[165]: filter_sender: Whitelisted, <bill@example.com> at [124.232.143.196]
    [124.232.143.196]

Feb 20 00:02:26 <mail.info> fox dovecot: [ID 111 mail.info] IMAP(john): Connection closed

Feb 20 00:17:54 <mail.info> eik dovecot: [ID 115 mail.info] auth(default): shadow(bob,212.05.28.184): lookup

Feb 20 00:17:54 <mail.info> eik dovecot: [ID 116 mail.info] auth(default): passwd(bob,212.05.28.184): lookup
```

Figure B.3: Example of loglines with various lengths and keyword fractions

```
dovecot ID mail.info auth default client in AUTH PLAIN service IMAP secured lip rip resp hidden 3
dovecot ID mail.info auth default client out OK user 2
dovecot ID mail.info auth default master out USER system_user uid gid home 2
dovecot ID mail.info auth default new auth connection pid 1
dovecot ID mail.info imap-login Login user method PLAIN rip lip TLS 2
dovecot ID mail.info IMAP Disconnected Logged out 2
clamd SelfCheck Database status OK. 1
mimedefang-multiplexor Killing idle slave pid Slave has processed requests 1
```

Figure B.4: Results from mining logline patterns from the example log in Figure A.4.

```
Before releasing of hidden labels:

dovecot ID mail.info auth default client in AUTH PLAIN service secured lip rip resp hidden 10637
dovecot ID mail.info auth default client in AUTH PLAIN service lip rip resp hidden 1290
dovecot ID mail.info auth default client in AUTH service IMAP secured lip rip 288
dovecot ID mail.info auth default client in AUTH service IMAP lip rip 135

After release of hidden labels:

dovecot ID mail.info auth default client in AUTH PLAIN service IMAP lip rip 121
dovecot ID mail.info auth default client in AUTH PLAIN service IMAP lip rip resp hidden 357
dovecot ID mail.info auth default client in AUTH PLAIN service IMAP secured lip rip 243
dovecot ID mail.info auth default client in AUTH PLAIN service IMAP secured lip rip resp hidden 8079
dovecot ID mail.info auth default client in AUTH PLAIN service POP3 secured lip rip resp hidden 2558
dovecot ID mail.info auth default client in AUTH PLAIN service POP3 lip rip resp hidden 933
dovecot ID mail.info auth default client in AUTH LOGIN service IMAP lip rip 14
dovecot ID mail.info auth default client in AUTH LOGIN service IMAP secured lip rip 45
```

Figure B.5: Results from releasing hidden labels. This example is an extract from the resulting patterns produced by a system log from the Department of Computer and Information Science(IDI) at the Norwegian University of Science and Technlogy.

# Appendix C

# Descriptions of files attached

This appendix provides an overview of the files attached to this report. The code is proof-of-concepts used to test ideas from the report. It was decided to only include a short description of each file instead of adding the whole source code to the report. All code is written as valid Python 2.5 code.

Note that all the below methods are proof-of-concepts only. Log files containing blank lines, or lines that do not conform to the Syslog format, may cause the prototypes to fail. Best practice techniques, such as writing to a temporary file while generating the results, have not been prioritised.

## C.1 List of files and folders

**Message sequences** :

- regexp.py
- regularity.py
- markov.py

**Pattern mining** :

- Item occurrence frequencies :
  - occurrence_freq.py
  - itemset_gen.py
  - simple_clustring.py
  - msg_extraction.py
- Subsets :
  - subsets.py
  - msg_extraction.py
- Primary sorting :
  - primary_sorting.py
  - itemset_gen.py
  - msg_extraction.py

- Histograms :
  - histograms.py
  - msg_extraction.py
- Tree structure :
  - TreeStructure.py
  - TreeStructure.conf
  - exceptions
  - TreeNode2.py
  - unittests_TreeNode.py
  - tree_graph.py
  - notugly.xsl
  - verify_results.py
  - msg_extraction.py

**Statistical analysis** :

- statistical_analysis.py
- statistical.config

## C.2 Message sequences

The prototypes in this section implement the ideas described in Chapter 8 on regularity and Chapter 9 on Markov models.

### C.2.1 regexp.py

*regexp.py* contains regular expressions used to split log message strings into parts defined by the Syslog specification [2]. *markov.py* and *regularity.py* use *regexp.py*.

### C.2.2 regularity.py

*regularity.py* is the proof-of-concept for the idea presented in Chapter 8. It depends on *regexp.py* and takes a single log file as input:

```
$ python regularity.py logfile
```

It creates two files:

- *candidates.txt* containing the candidates described by host, process, message hash, delta, last seen and confidence value.
- *msgs.txt* containing the actual messages (hash plus text string) for the candidates.

If the above mentioned files exists, they are used to validate the results from the *logfile* and to identify anomalies.

### C.2.3 markov.py

*markov.py* is the proof-of-concept for the idea presented in Section 9.1.1. It takes a single log file as input:

```
$ python markov.py logfile
```

It creates one file after it has analysed the log file:

- *markovstat.txt* containing all state transitions and their probability.

If *markovstat.txt* already exists, it is used to validate the results from the *logfile* and to identify anomalies.

## C.3 Pattern mining

Every pattern mining prototype described in the following make use of the *msg_extraction.py* file. *msg_extraction.py* contains regular expressions for fetching the message part of Syslog-formatted loglines. It also contains logic for discovering and removing ip addresses, email addresses and other easily recognisable variables. For simplicity the *msg_extraction.py* file has been added to all folders.

All prototypes require two input arguments. They need the path to the logfile being subject for analysis, and the path to a directory to which the results can be written. All prototypes can be run using:

```
$ python filename.py -l logfile -p resultsdirectory
```

In addition all prototypes have a standard help function that will list all available optional arguments:

```
$ python filename.py --help
```

The prototypes described in the following three sections implement the ideas discussed in Sections 5.2, 5.3, and 5.4. All these approaches is somehow based on an item's occurrence frequency in a log file.

### C.3.1 Item occurrence frequencies

The item occurrence prototype consists of three files. The *occurrence_freq.py* holds the main method, and calls the two other files, *itemset_gen.py* and *simple_clustring.py*.

*occurrence_freq.py* outputs two files. The first is a set of all n-itemsets found in the log, and the second is the final patterns. The *simple_clustring.py* uses the data in the n-itemsets file to generate the final results.

### C.3.2 Subsets

The subsets approach consists of one single file, *subsets.py*. It outputs a file with all the generated subsets found in the logfile, but does not proceed to analyse the results, since the method proves to be too slow for practical use.

### C.3.3 Histograms

The histogram prototype consists of one file, *histograms.py*. It requires the same two arguments as the above mentioned prototypes, and it outputs a file with generated 1-itemsets, and a file with the final results.

### C.3.4 Primary sorting

The primary sorting idea is described in Section 5.5. It uses wide, or coarse, n-itemsets to group similar loglines.

The *primary_sorting.py* uses *itemset_gen.py* to generate its 1-itemsets. It takes the usual input arguments as described above, and outputs a single file containing the final results.

### C.3.5 Tree structure

The tree structure approach is described in Chapter 6. Loglines being inserted into the tree are either inserted as new branches, or are merged into an already existing branch.

The tree structure prototype consists of totally nine files. The *msg_extraction.py* is the same file used for all the above prototypes. *TreeStructure.py* holds the main method. It takes two mandatory arguments, like the above. The *TreeStructure.py* file reads a configuration file called *TreeStructure.config*, by default. Additionally, a list of exceptions can be provided. Example exceptions are stored in the *exceptions* file, and the configuration keeps track of which exception file is in use.

*TreeStructure.py* make use the methods in *TreeNode2.py*, while the *unittests_TreeNode2.py* file provides unit tests for the methods in *TreeNode2.py*.

If the number of tree nodes created does not exceed a certain threshold,[1] prototype outputs a graphical illustration of the tree generated by the prototype. If the results from all intermediate stages in the algorithm are requested, a corresponding graphical illustration will follow.

The XML style sheet file *notugly.xsl* is created by Vidar Hokstad.[2] It simply smartens up the output graphs.

[1] Drawing above 1000 nodes may take some time.
[2] Available at `http://www.hokstad.com`

Finally the *verify_results.py* file provides primitive methods to assure that the resulting patterns are correct. *verify_results.py* requires three input arguments. First the usual path to the logfile, then the path to the resulting patterns, and finally the path to the exceptions file:

```
$ python verify_results.py logfile patterns exceptions
```

The resulting report is simply printed to standard output.

## C.4 Statistical analysis

The *statistical_analysis.py* prototype analyses the outputted pattern files, and reports back about any anomalies found. It could be that an otherwise frequent pattern is missing, that a pattern has a count much higher, or lower, than expected, or that a new pattern is found.

Every pattern is stored in a database. The prototype use a configuration file with information about which database to use. If the database does not exist, it is created.

*statistical_analysis.py* takes three mandatory arguments. The first is the path to the patterns to analyse, the second is the path to the directory where the results should be stored. Lastly, the prototype needs a date to identify the patterns in the database (preferably the date of the pattern's originating log file).

```
$ python statistical_analysis.py -p patterns -r resultsdirectory
    -t 2009-06-23
```

The prototype outputs a single report file, named according to the date argument, *2009-06-23_report*.