Chapter 1

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.

This chapter is a part of the report delivered. In that report, this chapter is named Appendix C. The main difference between the to versions is that the one included in the complete report contains references to chapters where the background for the implementation is discussed.

1.1 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

■ Histograms :

- histograms.py
- msg_extraction.py

■ Primary sorting :

- primary_sorting.py
- itemset_gen.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

1.2 Message sequences

The ideas implemented here are discussed in the Markov chapter.

1.2.1 regexp.py

regexp.py contains regular expressions used to split log message strings into parts defined by the Syslog specification. *markov.py* and *regularity.py* use *regexp.py*.

1.2.2 regularity.py

regularity.py is the proof-of-concept for the idea presented in the Markov analysis chapter. 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.

1.2.3 markov.py

markov.py is the proof-of-concept for the idea presented in Section **??**. 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.

1.3 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
```

1.3.1 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.

1.3.2 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.

1.3.3 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.

1.3.4 Primary sorting

Primary sorting 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.

1.3.5 Tree structure

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.

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
```

[1] Drawing above 1000 nodes may take some time.

[2] Available at `http://www.hokstad.com`

The resulting report is simply printed to standard output.

1.4 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*.