Proprietary and Confidential Information

Revision History

|  |  |
| --- | --- |
| **Date** | **Change Description** |
| Mar-23-2020 | Initial draft |
| May-09-2020 | Add unsupervised learning system, aka DeepLog |
| May-29-2020 | Update real-time prediction scheme |
| July-09-2020 | Update DeepLog Exec Path Detection |
| Aug-25-2020 | Update sequence label algorithm and boardfarm adaptation |
| Aug-27-2020 | Update sequence session scheme |
| Sep-09-2020 | Add recovering scheme of messed cm logs |

Table of Contents

[1 Introduction 1](#_Toc50902449)

[2 Supervised Learning System 2](#_Toc50902450)

[2.1 Overview 2](#_Toc50902451)

[2.2 Pre-Processing 2](#_Toc50902452)

[2.2.1 About the Timestamp and Labels in Train Dataset 2](#_Toc50902453)

[2.2.2 Rectify Various Logs 3](#_Toc50902454)

[2.2.3 Format the Tables 4](#_Toc50902455)

[2.2.4 Convert Multi-Line Log to One-Line Format 4](#_Toc50902456)

[2.2.5 Extract the Label Vector from Train Dataset 5](#_Toc50902457)

[2.3 Clustering 6](#_Toc50902458)

[2.3.1 Template 6](#_Toc50902459)

[2.3.2 Algorithm of Drain 6](#_Toc50902460)

[2.3.3 Template ID 9](#_Toc50902461)

[2.3.4 The Application Layer of Drain 9](#_Toc50902462)

[2.4 Feature Extraction 10](#_Toc50902463)

[2.4.1 Event Count Matrix 10](#_Toc50902464)

[2.4.2 Windowing 10](#_Toc50902465)

[2.4.3 Template ID vs. Template Index 12](#_Toc50902466)

[2.4.4 Event Count Matrix Constructing 12](#_Toc50902467)

[2.4.5 Tf-Idf 13](#_Toc50902468)

[2.5 Training 14](#_Toc50902469)

[2.5.1 The Models 14](#_Toc50902470)

[2.5.2 Preservation of Trained Models 14](#_Toc50902471)

[2.6 Prediction 15](#_Toc50902472)

[2.6.1 Validation Metrics 15](#_Toc50902473)

[2.6.2 For Distribution 15](#_Toc50902474)

[2.7 Post-Processing 15](#_Toc50902475)

[3 Incremental Learning 17](#_Toc50902476)

[3.1 Overview 17](#_Toc50902477)

[3.2 Incremental Clustering 17](#_Toc50902479)

[3.2.1 Template Library 17](#_Toc50902480)

[3.2.2 Convert Drain to be Incremental 17](#_Toc50902481)

[3.2.3 Mark Templates Status 19](#_Toc50902482)

[3.2.4 Benefit to the Old School 20](#_Toc50902483)

[3.3 Incremental Feature Extraction 20](#_Toc50902484)

[3.3.1 Define a Size of Feature Set 20](#_Toc50902485)

[3.3.2 Initialization of Feature Set 20](#_Toc50902486)

[3.3.3 Update of Feature Set 20](#_Toc50902487)

[3.4 Incremental Tf-Idf 21](#_Toc50902488)

[3.4.1 The Tf 21](#_Toc50902489)

[3.4.2 The Idf 22](#_Toc50902490)

[3.5 Incremental Training 22](#_Toc50902495)

[3.5.1 The Models 22](#_Toc50902496)

[3.5.2 The Intermediate Trained Model 22](#_Toc50902497)

[4 Old School System 23](#_Toc50902498)

[4.1 Overview 23](#_Toc50902499)

[4.2 Pre-Processing and Clustering 23](#_Toc50902500)

[4.3 The Knowledgebase 23](#_Toc50902501)

[4.3.1 The Knowledge Item without Parameters 24](#_Toc50902502)

[4.3.2 The Knowledge Item with Parameters 24](#_Toc50902503)

[4.4 Extract the Parameters from Log 24](#_Toc50902504)

[4.4.1 The Log Format in Structured Logs File 24](#_Toc50902505)

[4.4.2 Parse the Parameters 24](#_Toc50902506)

[4.5 Retrieve the Knowledgebase 24](#_Toc50902507)

[4.6 Post-Processing 24](#_Toc50902508)

[5 Unsupervised Learning System 25](#_Toc50902509)

[5.1 Overview 25](#_Toc50902510)

[5.2 Execution Path Anomaly Detection 25](#_Toc50902511)

[5.2.1 Preceding Blocks 26](#_Toc50902512)

[5.2.2 Log Key (aka Word in NLP Parlance) Representation 26](#_Toc50902513)

[5.2.3 Log Sequence Representation 27](#_Toc50902514)

[5.2.4 Data Loading 28](#_Toc50902515)

[5.2.5 The Model 29](#_Toc50902516)

[5.2.6 Training and Evaluation 30](#_Toc50902517)

[5.2.7 Prediction 32](#_Toc50902518)

[5.2.8 Post-Processing 32](#_Toc50902519)

[5.2.9 Online Update of Exec Model 33](#_Toc50902520)

[5.3 Parameter Value Anomaly Detection 33](#_Toc50902521)

[5.4 Workflow Construction 33](#_Toc50902522)

[6 Real-Time Prediction 34](#_Toc50902523)

[6.1 Overview 34](#_Toc50902524)

[6.2 Sampler 34](#_Toc50902525)

[6.2.1 Helper Thread 34](#_Toc50902526)

[6.2.2 Fetch Data from Buffer 34](#_Toc50902527)

[6.3 Pre-Processing 35](#_Toc50902528)

[6.4 Clustering 35](#_Toc50902529)

[6.5 Test Data Vectoring in DeepLog 35](#_Toc50902530)

[6.6 Prediction and the Old School System 35](#_Toc50902531)

[6.7 Post-Processing 35](#_Toc50902532)

[7 Appendix 36](#_Toc50902533)

[7.1 Labeling Assistant 36](#_Toc50902534)

[7.2 Consideration about the Log on Cable Modem 36](#_Toc50902535)

[7.3 Known Issues and Further Improvements 36](#_Toc50902536)

[7.3.1 Variable Template Classification 36](#_Toc50902537)

[7.3.2 Need Boost the Partial Fit Precision 36](#_Toc50902538)

[7.3.3 Case of Sparse Feature Vector 36](#_Toc50902539)

[7.3.4 If Very Few Logs are Contained in an Instance/Window 36](#_Toc50902540)

[7.4 Integration with Boardfarm Test System 37](#_Toc50902541)

[7.4.1 Adapting Logs from Boardfarm to Preprocessor 37](#_Toc50902542)

[7.5 Recover messed up logs (for CM prediction only) 38](#_Toc50902543)

[7.5.1 The Cause 38](#_Toc50902544)

[7.5.2 Assumptions 38](#_Toc50902545)

[7.5.3 Algorithm 38](#_Toc50902546)

# Introduction

We have different methods to analyze the logs but need to select which one to be used basing on the characteristics of the target logs and our requirements. Key word searching is much effective for the standardized log like syslog (RFC 5424), which has specific fields like Timestamp, Device-Id, Severity-Level, etc. For the logs that have no standard formats, we can use a technique called Clustering to extract templates in advance and then match or classify each log according to its template. Computer is good at searching, sorting and classifying these kind of things.

However, the mentioned methods above can only parse the standalone log without considering the context where the single log resides. We also need have the knowledge of EACH anomaly log beforehand. Actually one log might be good in one context but might not in another. We human beings with the domain knowledge can easily determine the anomaly according to its context, and even predict or deduce the unknown or new things per the context. It is difficult for the computer to do similar things. With the help of machine learning and some techniques borrowed from text mining and natural language processing (NLP), we hope we can find the anomalies according to the context even if they are new to us, that is to say, we might never label them (for the supervised learning case) in the training dataset before. It is a big challenge because of the characteristics of logs as dataset.

We will design a system that analyzes logs in both machine learning way and old school way to overcome their respective shortcomings.

# Supervised Learning System

## Overview

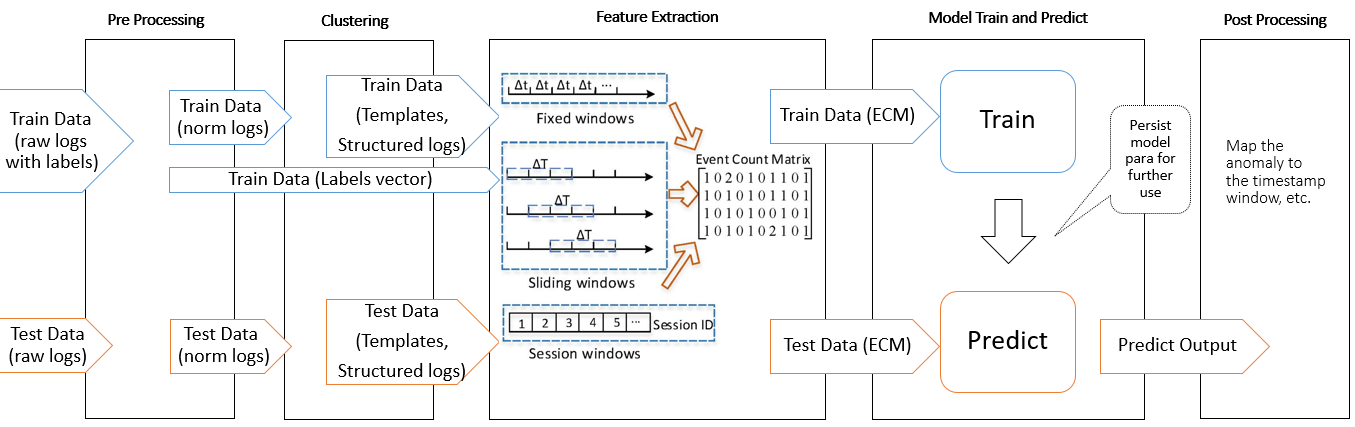


Figure : Supervised Learning System Blocks

This system is for supervised learning that includes five sub-blocks, aka. **Pre Processing**, **Clustering**, **Feature Extraction**, **Train**/**Predict** and **Post Processing**. Before the **Pre Processing** block, for the train dataset we still need label the raw logs manually or use the label assistant (see Appendix 5.1) to deal with some known anomalies. We might also need merge multiple log files into one big train dataset in advance. (The Feature Extraction diagram comes from paper “*Experience Report: System Log Analysis for Anomaly Detection” ISSRE 2016.21)*

## Pre-Processing

Pre-Processing is an application-specific block, say, we should implement this block depending on where the logs come from. For Cable Modem / DOCSIS system, this block relatively does more things than others do like eRouter and STB because of lacking standardization. The purpose of this block is to normalize the logs including rectifying various logs and converting multiline log to one-line format, and extract the label vector for the train dataset.

### About the Timestamp and Labels in Train Dataset

The timestamp format is defined as [YMD-h:m:s.ms], e.g. [20190719-09:31:25.865]. They are added by serial console tool like secureCRT. We will use them later in Feature Extraction to build the data windows, and in Post Processing to trace back the original fault logs.

For labeling, an ‘abn: ’ is added behind the timestamp to indicate this is an abnormal log, e.g. [20190719-09:31:25.865] abn: xx…

### Rectify Various Logs

Definitions for log/line

primary line - no space proceeded

nested line  - one or more spaces proceeded

empty line   - LF or CRLF only in one line

Purge the logs

01) Remove timestamps, console prompts, tables, empty lines

02) Format DS/US channel status tables

03) Remove some tables which are useless

04) Format initial ranging block to one line log

05) Indent some specific lines in multi-line log

06) Remove empty lines

07) Convert a nested line as primary if two more empty lines proceeded

08) Convert some specific lines as primary

09) Remove specific whole multi-line log

10) Split some tokens

...

Except for the DS/US channel status tables formatting, almost all others can define the specific regular expression in a separated list or dictionary, and then we can extend the functionality easily in the future. An example (item 08 in the purge list above) of rectifying the log.

Algorithm 2-1 in pseudo code

1: *# Convert some specific nested lines as primary*

2: **FOR** pattern **IN** sNestedLinePatterns

3:     match 🡨 pattern.match(current line)

4:     **IF** match

5:         convert current line to primary

6:         **BREAK**

7:     **END**

8: **END**

Above code snippet converts some nested lines to primary, in other words, we want some sub-lines of a log as a standalone log. If want to convert more logs pattern, we just need provide more regular expressions to the list.

Regular expression list

"""

Patterns for specific lines which I want to convert them as primary

"""

sNestedLinePattern0 = re.compile(**r**' +DOWNSTREAM STATUS')

sNestedLinePattern1 = re.compile(**r**' +CM Upstream channel info')

sNestedLinePattern2 = re.compile(**r**' +Receive Channel Config\:')

sNestedLinePatterns = [

    sNestedLinePattern0,

    sNestedLinePattern1,

    sNestedLinePattern2

]

### Format the Tables

Various tables should be removed or formatted. Especially DS/US channel status tables give a lot of info and so we reserve them.

Table of DS/US status

# Active Downstream Channel Diagnostics:

#   rx id  dcid    freq, hz  qam  fec   snr, dB   power, dBmV  modulation

#                            plc  prfA

#   -----  ----  ----------  ---  ---  ---------  -----------  ----------

#       0\*    1   300000000   y    y          35            3       Qam64

#       1     2   308000000   y    y          34            4      Qam256

#      32    66   698000000   y    y          35            1    OFDM PLC

The algorithm of formatting a table. Line 2 & 8 consider the case that the table is messed up by printings from other threads.

Algorithm 2-2 in pseudo code

 1: **IF** match table title **&&** in the table

 2:     **IF** *not* nested line **&&** *not* empty line

 3:         This line is messed, delete

 4:     **ELIF** empty line **&&** dsTableEntryProcessed **&&** (*NOT* lastLineMessed)

 5:         reset some variables of processing status

 6:     **ELIF** *NOT* empty line

 7:         dsTableEntryProcessed 🡨 *True*

 8:         **IF** tableMessed

 9:             re-construct the last element

10:         **END**

11:         format the whole new log

12:     **END**

13: **END**

### Convert Multi-Line Log to One-Line Format

After the purge, we can convert all multi-line log to one-line format without difficulties. We only preserve the primary timestamp (the 1st line) for a complete multiline log. The results of train and test are saved to train\_norm.txt & test\_norm.txt in logs/ directory.

Algorithm 2-3 in pseudo code

 1: **FOR** line **IN** file

 2:     save the timestamp for current line

 3:     remove the timestamp from current line

 4:     **IF** nested line

 5:         Concatenate current line to lastLine

 6:     **ELSE** it is primary line

 7:         it means concatenating ends

 8:         combine the timestamp and last line content

 9:         write last line to norm file

10:         update last line parameters

11:             aka.

12:             lastLine 🡨 current line

13:             lastLineTS 🡨 currentLineTS

14:     **END**

15: **END**

16: update the final line of the file and write to norm file

### Extract the Label Vector from Train Dataset

We extract the label vector from train dataset at the end of Pre-Processing and save it as train\_norm.txt\_labels.csv in results/train/. To simplify the code structure, we extract the labels from test\_norm.txt too and save test\_norm.txt\_labels.csv to results/test/. If there are labels in test dataset, then we can verify the accuracy by the way. The label vector file includes two columns, the 1st is LineId which is 1 based line number, and the 2nd is Label where ‘-’ represents normal and ‘a’ represents abnormal.

After extract labels, we remove them from norm file as The Clustering module uses norm file as input however doesn’t need labels.

Algorithm 2-4 in pseudo code

 1: **FOR** each log **IN** norm file

 2:    **IF** match the label pattern 'abn: '

 3:        write 'a' to the vector

 4:        remove the 'abn: ' from current log

 5:    **ELSE**

 6:        write '-' to the vector

7:    **END**

8: **END**

 9: write label vector and lineId to a file

10: overwrite the old norm file with contents that labels are removed

## Clustering

Clustering is a technique to classify texts in a manner of self-organizing. All the logs will be grouped into their corresponding templates.

### Template

There are four logs below. We can manually extract the templates by replacing variables with asteroids. Line 3 and Line 4 share the same template.

We take the template as the feature of the log.

Raw logs

1: [20190719-08:58:34.233] RNG-RSP UsChanId= 149 Adj: freq= 278 Stat= Continue

2: [20190719-08:58:35.227] RNG-RSP UsChanId= 149 Adj: power= -1 Stat= Continue

3: [20190719-08:58:36.220] RNG-RSP UsChanId= 149 Stat= Continue

4: [20190719-08:58:37.718] RNG-RSP UsChanId= 149 Stat= Success

Templates

1: [20190719-08:58:34.233] RNG-RSP UsChanId= <\*> Adj: freq= <\*> Stat= <\*>

2: [20190719-08:58:35.227] RNG-RSP UsChanId= <\*> Adj: power= <\*> Stat= <\*>

3: [20190719-08:58:36.220] RNG-RSP UsChanId= <\*> Stat= <\*>

4: [20190719-08:58:37.718] RNG-RSP UsChanId= <\*> Stat= <\*>

### Algorithm of Drain

The original Drain (fixed **D**epth t**r**ee b**a**sed onl**i**ne log parsi**n**g method) algorithm comes from paper below. It does clustering on the logs to generate the templates and some statistics like how many logs of a certain template in a file.

To understand the algorithm and some terminologies like token layer/similarity/layers/cache, etc., see the paper. We only explain our modifications that accommodate our design.

The original Drain Algorithm and its implementation

 [Arxiv'18] Pinjia He, Jieming Zhu, Hongyu Zhang, Pengcheng Xu,

            Zibin Zheng, and Michael R. Lyu.

            A Directed Acyclic Graph Approach to Online Log Parsing, 2018.

https://github.com/logpai/logparser.git

When applied the Drain on the Cable Modem / DOCSIS logs, we found some issues and revised the algorithms as followings.

Algorithm 2-5 (rev0.1, disable the cache mechanism)

 1: def treeSearch(self, rn, seq):

 2:     """

 3:     Browses the tree in order to find a matching cluster to a log

 4:     It does not generate new node

 5:     Attributes

 6:     ----------

 7:     rn     : Root node

 8:     seq    : Log sequence to test

 9:     return : The matching log cluster

10:     """

11:     retLogCluster = None

12:     seqLen = len(seq)

13:     if seqLen in rn.childD:

14:         *# Check if there is a key with the same length, namely*

15:         *# the cache mechanism.*

16:         *#*

17:         *# Comment it out because cache mechanism may lead to*

18:         *# wrong classification of logs if two or more templates*

19:         *# are similar, in other words, the log may be accepted*

20:         *# by a template w/ matching similarity which is not the*

21:         *# highest.*

22:         *#*

23:         *# Paper: retLogCluster = self.keyTreeSearch(seq)*

...

Algorithm 2-5 is the entry point to search the tree to see if any existing cluster, aka, template can match the new log.

***Rev0.1***: Line 23 above is the implementation per the paper, aka, the cache mechanism. It will lead to wrong classification in some cases.

Algorithm 2-6 (rev0.2)

 1: *# Calculate the similarity. The seq1 is template*

 2: def SeqDist(self, seq1, seq2):

 3:     """

 4:     Calculate the simlilarity between the template and raw log

 5:     Attributes

 6:     ----------

 7:     seq1   : the template

 8:     seq2   : the raw log

 9:     return : retVal that represents the similarity

10:              updateTokenNum, the num of numOfPara (<\*>) in current temp

11:     """

12:     …

13:     for token1, token2 in zip(seq1, seq2):

14: *# 1). Do not change the first token to variable*

15: *…*

16: *# 2). Do not accept successive <\*> <\*> variable*

17: *…*

18:         if token1 == '<\*>':

19:             numOfPara += 1

20:             *# Comment out line below to count <\*> in simTokens*

21:             continue

22:         if token1 == token2:

23:             simTokens += 1

24:         *# 3). Do not accept seq2 if some special tokens are different*

25:         *# between the template seq1 and current log seq2*

26:         *# This can prevent Drain from over-pasering some tokens*

27:         for pn in self.para.rex\_s\_token:

28:             if (pn.fullmatch(token1) and pn.fullmatch(token2) and …

29:                 (pn.fullmatch(token1) and pn.fullmatch(token2)==None) …

30:                 (pn.fullmatch(token2) and pn.fullmatch(token1)==None):

31:                 sTokenNoMatch = 1

32:                 break

33:         if sTokenNoMatch:

34:             break

...

Algorithm 2-6 is the core to calculate the token similarity.

***Rev0.1:*** 1) We count the <\*> in the template when calculate similarity. 2) Do something to prevent Drain from over-parsing some tokens. The corresponding regular expressions have defines in Drain application specific code, see section 2.3.4.

***Rev0.2:*** Do not count the <\*> in similarity. 1) Do not change the first token to var. 2) Do not accept successive two or more variables like … <\*> <\*> …

Algorithm 2-7 (rev0.2)

 1: def addCluster(self, messageL, logIDList, clusterL, ...):

 2:     *# The initial value of st is 0.5 times the percentage*

 3:     *# of non-digit tokens in the log message*

 4:     numOfPara = 0

 5:     for token in messageL:

 6:         *# In the pre-process of Drain domain, I replaced*

 7:         *# all possible digital var with <\*> already*

 8:         *# Do not follow the original method in the paper*

 9:         *# section 4.1.2*

10:         *#*

11:         *# Paper: if self.hasNumbers(token):*

12:         if token == '<\*>':

13:             numOfPara += 1

14:     *# The "st" is similarity threshold used by the similarity*

15:     *# layer, see paper formula (3)*

16:     *#*

17:     *# Paper: newCluster.st = 0.5 \* (1- numOfPara / float(len(logmessageL)))*

18:     *#*

19:     *# Initial st is the lower bound. Make it bigger*

20:     *#* *to avoid over-parsing*

21:     newCluster.st = 0.6 \* (1- numOfPara / float(len(logmessageL)))

22:     newCluster.initst = newCluster.st

...

Algorithm 2-7 is to add a new cluster to the tree.

***Rev0.1***: 1) We will convert the digits in the Drain-app layer, so disable it here. 2) The adaptive threshold is not good so replace it with a static value. This is a heuristic value after many tests.

***Rev0.2***: Because of the change of Rev0.2 in Algorithm 2-6, we recover the adaptive similarity threshold.

### Template ID

The Template ID (aka event ID, feature ID, TID) is the hash value of the template. It represents the template uniquely. We will use the Template ID in Feature Extraction and in the Old School.

### The Application Layer of Drain

This is application-specific layer that do some pre-processing before run the Drain core. It does several things:

1. Switch between train dataset and test dataset according to a configure file.
2. Maintain a regular expression dictionary to replace some variables with <\*>.
3. Maintain a regular expression list to avoid over-parsing of tokens. See Algorithm 2-6 Rev0.2, change 3).
4. Direct Drain to save the results to specific directories.
   1. Template lib file at /results/persist/template\_lib.csv
   2. Train/Test Template files at /results/train & /results/test
   3. Structured log files at /results/train & /results/test

## Feature Extraction

In machine learning, a data sample has multiple features, which distinguish itself from others. We can take a log as one data sample, and the different words in it make up the features. This kind of feature representation has small granularity (aka. on words) that might be useful for natural language processing. For anomaly detection, we do not care a single word but just the whole sentence good or bad. Secondly, we want to consider one log in a context instead of standalone.

### Event Count Matrix

In the area of log analysis, there is a scheme that utilizes log templates as features. Because the feature granularity now is a log, we compose one data sample that includes more than one log, aka logs in a time window. The initial value for a feature in a time window is the count of the occurrences of the log’s template. Then we get the Event Count Matrix (ECM) as below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Instance | Template0 | Template1 | Template2 | Label |
| 1 | 11 | 3 | 20 | 0 |
| 2 | 0 | 6 | 20 | 1 |
| 3 | 0 | 1 | 0 | 0 |
| 4 | 9 | 1 | 0 | 1 |
| 5 | 1 | 10 | 0 | 0 |

One instance represents a time window, in which at least one log is included. The instance 1 above includes 34 logs that are allocated to different 3 templates. The instance 3 has only 1 log, which matches template 1. The last column is label vector. We take the instance as anomaly as long as at least one log in the instance is labeled as abnormal.

Following sections will describe how to construct the ECM and further processing of it before feeding it to train/predict models.

### Windowing

We use windowing to partition the logs into instances or samples. As showed in Feature Extract block, usually there are three kind of windowing schemes, aka Fixed / Sliding / Session. The Session window is a little bit special, as it requires each log associates a session id, and all logs with same session id construct a session window. Not all system logs have session id, so we use a timing window like Fixed or Sliding. As for the selection of Fixed window or Sliding window, the latter is recommend by some research. This is because Fixed window (window size is not big enough) might lead to the uneven distribution of anomalies. E.g., some anomalies in current window might relate to the context in the former time window. By using Sliding window, we can decrease the probability of this wrong log partitioning.

Algorithm 2-8 in pseudo code

1: start\_time 🡨 timestamp\_vector[0]

2: start\_index 🡨 0

3: end\_index 🡨 -1

4: *# Get the first start, end index, end time*

5: **FOR** cur\_time **IN** timestamp\_vector

6:     *# Window end (end\_time) selects the min if not equal*

7:     **IF** cur\_time <= start\_time + ['window\_size']

8:         end\_index 🡨 end\_index+1

9:     **ELSE**

10:         **BREAK**

11:     **END**

12: **END**

13: start\_end\_pair 🡨 (start\_index, end\_index)

14: start\_end\_index\_list 🡨 append(start\_end\_pair)

15: *# Move the start and end index until next sliding window*

16: **WHILE** end\_index < log\_size - 1

17:     prev\_win\_start 🡨 start\_index

18:     **FOR** cur\_time **IN** timestamp\_vector[prev\_win\_start:end]

19:         *# Window start (start\_time) selects the max if not equal*

20:         **IF** cur\_time < start\_time + ['window\_step\_size']

21:             start\_index 🡨 start\_index+1

22:         **ELSE**

23:             start\_time 🡨 cur\_time

24:             **BREAK**

25:         **END**

26:     **END**

27:     end\_index 🡨 start\_index - 1

28:     curr\_win\_start 🡨 start\_index

29:     **FOR** cur\_time **IN** timestamp\_vector[curr\_win\_start:end]

30:         *# Window end (end\_time) selects the min if not equal*

31:         **IF** cur\_time <= start\_time + ['window\_size']

32:             end\_index 🡨 end\_index+1

33:         **ELSE**

34:             **BREAK**

35:         **END**

36:     **END**

37:     start\_end\_pair 🡨 (start\_index, end\_index)

38:     start\_end\_index\_list 🡨 append(start\_end\_pair)

39: **END**

Algorithm 2-8 calculates Sliding windows per the window size and step size on the timestamp vector from train or test dataset. The format is a pair (*start\_index*, *end\_index*) for each window, and all the pairs are stored in a list. Note the *start\_index* is 0 based.

The *start\_end\_index\_list* has the format like this: *[(0, 10), (5, 15), (8, 22), … ]*

### Template ID vs. Template Index

The TID, aka template ID (identification) is the hash value of the template, see section 2.3.3, while TIdx, aka Index or idx means the order number (0 based) of the TID in the TID vector. This is one-to-one mapping between TID and Index. *We will use TIdx as the column index of ECM.* Thus, one column of ECM represents one template uniquely.

The TID vector comes from the template file or library, which is generated in the process of Clustering. The original order of TIDs in the TID vector from Drain algorithm has some special pattern, e.g. it follows the order of logs that appear in the serial console. Thus, accordingly the order of TID in ECM has the same pattern. It is not an issue, as most of machine learning models do not care the order of features in the matrix. Some model like Random Forest randomly selects subset of features but does not require randomizing the features in matrix in advance. However, it is not a bad idea to randomize it before we construct the ECM.

### Event Count Matrix Constructing

To construct the ECM, we need the window pair list, the TID of each log and the TID vector.

Algorithm 2-9 in pseudo code

1: *# Aggregate all the log indexes in each time window*

2: expanded\_indexes\_list 🡨 []

3: **FOR** dummy **IN** [0: inst\_number-1]

4:     index\_list 🡨 []

5:     expanded\_indexes\_list 🡨 append(index\_list)

6: **END**

7: **FOR** i **IN** [0: inst\_number-1]

8:     start\_index 🡨 start\_end\_index\_list[i][0]

9:     end\_index 🡨 start\_end\_index\_list[i][1]

10:     **FOR** l **IN** [start\_index, end\_index]

11:         expanded\_indexes\_list[i] 🡨 append(l)

12:     **END**

13: **END**

Algorithm 2-9 aggregates all the log indexes in each time window. The *start\_end\_index\_list* is the windows we get in section 2.4.2. The *expanded\_indexes\_list* has the format [[0, 1, …, 10], [5, 6, …, 15], [8, 9, …, 22], …]

Algorithm 2-10 in pseudo code

 1: labels 🡨 []

 2: event\_count\_matrix[inst\_number x feature\_number] 🡨 ZEROs

 3: **FOR** j **IN** [0: inst\_number-1]

 4:     label 🡨 0

 5:     **FOR** k **IN** expanded\_indexes\_list[j]

 6:         **IF** label\_vector[k]

 7:             label 🡨 1

 8:         **END**

9:         event\_id 🡨 event\_mapping\_data[k]

10:         **TRY**

11:             event\_index 🡨 event\_id\_shuffled.index(event\_id)

12:         **EXCEPT**

13:             **CONTINUE**

14:         **END**

15:         event\_count\_matrix[j, event\_index] 🡨 self + 1

16:     **END**

17:     labels 🡨 append(label)

18: **END**

Algorithm 2-10 constructs the ECM. The *expanded\_indexes\_list* is the one we get in algorithm 2-9. The *label\_vector* is the one we get in section 2.2.5 algorithm 2-4. It contains labels for each log/line. The *event\_mapping\_data* is the one we get in structured file in the result of Clustering in section 2.3.4. It contains event ID (aka. TID, template ID) for each log. The *event\_id\_shuffled* comes from the randomized TID vector in section 2.4.3.

### Tf-Idf

For the details of tf-idf, see <https://en.wikipedia.org/wiki/Tf-idf>. We apply the tf-idf technique on the ECM. From the viewpoint of tf-idf, a feature is a *term* and an instance or a row in ECM is called a *document*. All the instances/rows in the ECM make up a *corpus*.

The **tf** is term frequency, which means the term count in a document. In our case, it is the count of one feature in the instance. So the original ECM we get from 2.4.4 happens to be the tf.

The **idf** is inverse document frequency and the formula to calculate is as following.

*t: term*

*d: document*

*D: corpus*

*N: number of documents in corpus*

The denominator means the number of documents where the term t appears.

The tf-idf needs tf \* idf. We can use numpy to implement it. The df (line 3) and idf (line 4) are all vectors which are 1 x num\_instance dimension. Line 3 is the denominator of the idf formula. Line 5 is Hadamard product of matrixes.

Algorithm 2-11 in pseudo code

 1: tf\_matrix 🡨 ECM

 2: num\_instance 🡨 ECM rows num

 3: df\_vector 🡨 numpy.sum(ECM>0, axis=0)

 4: idf\_vector 🡨 numpy.log(num\_instance / (df\_vector + 1e-8))

 5: tf\_idf\_matrix 🡨 tf\_matrix \* numpy.tile(idf\_vector, (num\_instance, 1))

 6: new\_ECM 🡨 tf\_idf\_matrix

By default, we calculate the *idf\_vector* on train dataset and apply it to test dataset. We also have the option to calculate the *idf\_vector* on test dataset. The good one is TBD.

## Training

### The Models

We use scikit-learn library to train the models including Decision Tree, Logistic Regression, SVM and Random Forest. We probably need spend some time on the parameters tuning to get a good train/validation result.

### Preservation of Trained Models

After we train the models, scikit-learn provides us two options to preserve our result for future predict. Pickle and joblib can be used to dump the object memory, see link below.

<https://scikit-learn.org/stable/modules/model_persistence.html>

For distribution of training models, pickle/joblib methods are not good as it depends on scikit-learn and its dependencies’ versions. It means you must install the exact same versions on the distribution platform to do any predict, otherwise the result is not reliable.

We persist the models for deployment by using sklearn-onnx converter. This method is less dependent on scikit-learn and its dependencies. Onnx supports most of scikit-learn models.

<http://onnx.ai/sklearn-onnx/>

We save the trained models at results/persist/, e.g. model\_name.onnx.

## Prediction

### Validation Metrics

We use some metrics to validate the trained model as below.

Suppose set {0} is Negative, set {1} is Positive.

TP: True Positive, the result is within set {1}

FP: False Positive, wrongly classify elements of set {0} to set {1}

TN: True Negative, the result is within set {0}

FN: False Negative, wrongly classify elements of set {1} to set {0}

TP+FP: The reported size of set {1}

TP+FN: The actual size of set {1}

TN+FP: The actual size of set {0}

Low Precision means a lot of elements of set {0} are wrongly classified to set {1}, while low Recall means a lot of elements of set {1} are not identified.

### For Distribution

As said in section 2.5.2, we do not use pickle/joblib methods but instead onnx. We use onnx runtime api to load it back.

<https://microsoft.github.io/onnxruntime/python/api_summary.html>

## Post-Processing

The predict result from section 2.6.2 is a vector whose dimension is instance\_num x 1. The element 0 means good sample and 1 means the anomaly. We can trace back to the original raw logs file to find the time windows that contain the anomalies.

Note again that one instance or sample or time window contains more than one logs. We just know that some logs in the window might be wrong but cannot probe which single log it is because of the characteristic of the representation of features.

Algorithm 2-12 below uses the sliding window list (section 2.4.2), structured norm logs file (section 2.3.4) and predict result (section 2.6.2) to locate the anomaly time window. The analysis result is saved to /results/test/anomaly\_timestamp.csv.

Algorithm 2-12 in pseudo code

 1: anomaly\_window\_list 🡨 []

 2: **FOR** i **IN** [0: instance\_num-1]

 3:     **IF** test\_y\_pred[i]

 4:         start\_index 🡨 start\_end\_index\_list[i][0]

 5:         end\_index 🡨 start\_end\_index\_list[i][1]

 6:         anomaly\_window\_list 🡨 append(tuple((start\_index, end\_index)))

 7:     **END**

 8: **END**

 9: norm\_ts\_list 🡨 timestamp vector in structured\_file

10: anomaly\_timestamp\_list 🡨 []

11: **FOR** i **IN** [0: len(anomaly\_window\_list)-1]

12:     x 🡨 anomaly\_window\_list[i][0]

13:     y 🡨 anomaly\_window\_list[i][1]

14:     anomaly\_timestamp\_list 🡨 append(tuple((norm\_ts\_list[x], norm\_ts\_list[y])))

15: **END**

# Incremental Learning

## Overview

The logs as dataset have some special characteristics. E.g. 1) it’s impossible to collect all the logs at one time; 2) with the host system evolving, some logs might be deprecated and some new logs emerges. In other words, the feature set might keep changing. With the learning system in section 2, we do can combine multiple training log files into big one after carefully design the windowing algorithm in section 2.4.2. This is based on the assumption that the time gap between two training files is bigger than the window & step sizes. This requirement usually can be met. However, this method is not flexible to process the dynamic feature set.

Alternatively, the Scikit-learn provides *particial\_fit* method in some models for out-of-core approach: learning from data that do not fit into main memory. This also can be thought as an online or incremental learning method.

Before we start the incremental training, we need to resolve the issue of feature set changing when more and more training datasets flow into the system.



## Incremental Clustering

Having an incremental version of clustering is the first thing to convert the system to adapt to the incremental learning. The Drain algorithm in section 2.3.2 is not incremental and runs from scratch each time, and thus has two main issues: 1) some template might not be generated in the final format, as there might not have enough variations of the same template. 2) Cannot get all the templates unless using the complete dataset.

### Template Library

To let clustering incremental, we need store the template and then update it in the next time we run clustering on a new dataset. The library is stored in results/persist/template\_lib.csv. The lib is nearly as same as the one in results/train/train\_norm.txt\_templates.csv except the first column, which will be explained later in section 3.2.3.

### Convert Drain to be Incremental

Firstly, we need restructure the Drain to make the cluster creation and updating to be modularized. Then we have algorithm below to do the incremental job.

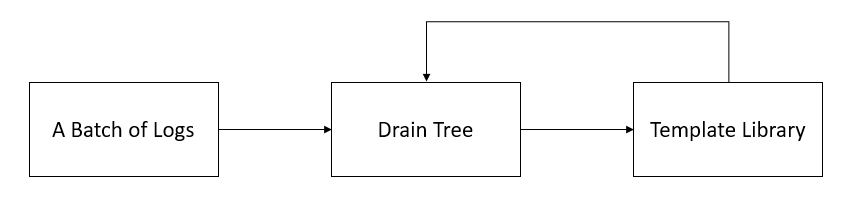


Figure : Incremental Drain

Algorithm 3-1 in pseudo code

 1: Load 🡨 the templates from template library

 2: *# Recover the tree from templates in library*

 3: **FOR** each template **IN** library

4: newCluster 🡨 new

 5:     logCluL 🡨 addCluster()

6: **END**

 7: Load 🡨 the raw log data

 8: *# Update the template library*

 9: **FOR** each log **IN** raw

10:     Search the log in the tree

11:     **IF** *NOT* match one node **IN** tree

12:         newCluster 🡨 new

13:         newCluster 🡨 log

14:         logCluL 🡨 addCluster()

15:     **ELSE**

16:         existingCluster 🡨 log

17:         update template in cluster conditionally

18:     **END**

19: **END**

Algorithm 3-1 is the top-level representation. Line 4, Line 9, Line 13 and Line 16 are the revised versions of Drain in section 2.3.2. Too many details are ignored here because of irrelevance. The main idea is to build the tree with template library firstly, and then do the clustering on the raw logs. The cluster (e.g. *newCluster*, *existingCluster*) is a structure that has fields like template, logIDs that converged to the same template, and occurrences. The cluster is actually a leaf on the tree. The list of clusters, say, *logCluL* is what we get after running Drain and it represents the whole updated templates.

At the end of Drain, we parse the list of clusters: *logCluL* to get the template library, the eventID, etc. This is nearly same as the original Drain but with two exceptions because of incrementing. The first exception is to merge possible duplicated templates with Algorithm 3-2 below. The second one will be explained in section 3.2.3.

Algorithm 3-2 in pseudo code

 1: tmp\_eventL 🡨 []

 2: **FOR** logClust **IN** logClustL

 3:     *# The row[0/1/2/3]: [tmp\_id\_old, tmp\_id, tmp\_str, occurrence]*

 4:     tmp\_unique 🡨 True

 5:     **FOR** row **IN** tmp\_eventL

 6:         **IF** tmp\_id == row[1]

 7:             print("Warning: template is duplicated, merging.")

 8:             tmp\_unique 🡨 False

 9:             **IF** row[0] != row[1]

10:                 **IF** tmp\_id == tmp\_id\_old

11:                     row[0] 🡨 row[1]

12:                 **END**

13:             **END**

14:             row[3] 🡨 row[3] + occurrence

15:             **BREAK**

16:         **END**

17:     **END**

18:     *# Drop current template if it is duplicate*

19:     **IF** tmp\_unique

20:         tmp\_eventL 🡨 append([tmp\_id\_old, tmp\_id, tmp\_str, occurrence])

21:     **END**

22: **END**

Duplication of template is inherent of Drain. Usually it does not happen however, we might see it in incremental Drain. So check & merge the duplicates here.

### Mark Templates Status

In the Incremental Feature Extracting, we need know if a template in library is new or not. A good method is adding an extra column ‘eventID\_old’, which represents the old templates in library before running the Drain.

Algorithm 3-3 in pseudo code

 1: **FOR** logClust **IN** logClustL

 2:     tmp\_str 🡨 logClust.logTemplate

 3:     occurrence 🡨 len(logClust.outcell.logIDL)

 4:     tmp\_id 🡨 hash(tmp\_str)

 5:     tmp\_id\_old 🡨 logClust.template\_id\_old

 6:     ...

 7: **END**

We save the old template ID to each cluster when we rebuild the tree with template library in algorithm 3-1. So here in algorithm 3-3, we can retrieve it and couple it with the new ID of new hashing. In template library, we save old TID in the 1st column and TID in the 2nd column.

### Benefit to the Old School

Even the input dataset has only one log, say, one line; the correct template still can be generated correctly as long as the template library is well trained. Without incremental clustering, we usually cannot get the correct template always when the input dataset has few logs.

## Incremental Feature Extraction

### Define a Size of Feature Set

The incremental training requires the feature set size is fixed, aka the column number of ECM. The value depends on the system where the logs come from. For the Cable Modem / DOCSIS system, we define a value of 2000. That means we suppose the number of all templates from this system is no more than 2000.

### Initialization of Feature Set

In Section 2.4.3, we randomize the feature sequence in ECM. To avoid being inconsistent with former design, we continue to randomize them in the process of update. We use a list to store all the TIDs in the template library, called STIDLE, which is defined as below:

*STIDLE: Shuffled Template Id List Expanded*

Algorithm 3-4 in pseudo code

 1: event\_id\_templates\_ext 🡨 extract the eventId list from template library

 2: event\_id\_templates\_ext 🡨 Pad ZEROs

 3: event\_id\_shuffled 🡨 shuffle (event\_id\_templates\_ext), aka randomization

 4: Save event\_id\_shuffled to disk, aka it is STIDLE.

### Update of Feature Set

Actually, we talk about the update of STIDLE here. It depends on the update of template library at section 3.2. We only update the STIDLE for the training dataset.

If there are new templates that are added to the template library in clustering, we will mark them with old envenId as ZERO in the library. Then we can use algorithm 3-5 to update the STIDLE. Note that the new eventId is added to a randomized empty position in STIDLE.

Algorithm 3-5 in pseudo code

 1: *# Case 1):*

 2: event\_id\_old\_zero 🡨 Find the ZERO values in EventIdOld

 3: idx\_zero\_STIDLE 🡨 Aggregate all idx of ZERO in STIDLE to a new list

 4: idx\_zero\_STIDLE\_shuffled 🡨 shuffle(idx\_zero\_STIDLE)

 5: *# Insert the new EventId to the STIDLE*

 6: **FOR** idx, tid **IN** [0: len(event\_id\_old\_zero)-1]

 7:     **TRY**:

 8:         event\_id\_shuffled.index(tid)

9:     **EXCEPT**:

10:         event\_id\_shuffled[idx\_zero\_STIDLE\_shuffled[idx]] 🡨 tid

11:     **END**

12: **END**

If an existing template is updated in clustering, and then the old and new eventId will not match each. This is the 2nd case we need update in the STIDLE.

Algorithm 3-6 in pseudo code

 1: *# Case 2):*

 2: **FOR** tidOld, tidNew **IN** template library

 3:     **IF** tidOld != '0' **AND** tidOld != tidNew

 4:         idxOld 🡨 event\_id\_shuffled.index(tidOld)

 5:         event\_id\_shuffled[idxOld] 🡨 tidNew

 6:     **END**

 7: **END**

The Case 3): If two or more templates are merged to a new one, we set the old templates to zero and put the new one at the first old one place.

The Case 4): If one or more templates are merged to an existing one, we set the old one value to zero.

The Case 3) & 4) are not supported in the current clustering implementation (Drain), so do not update the STIDLE accordingly here.

## Incremental Tf-Idf

### The Tf

The ECM is Tf, so it is already incremental version after section 3.3 processing.

### The Idf

According to the idf formula in section 2.4.5, we need accumulate the number of instance and df (document frequency) vector across batches of instances (or say across Epochs). So these two values should be saved to a file after each training.

Algorithm 3-7 in pseudo code

 1: df\_vec 🡨 from ECM

 2: df\_vec\_accm, num\_instance\_accm 🡨 from saved file

 3: df\_vec\_accm 🡨 df\_vec\_accm + df\_vec

 4: num\_instance\_accm 🡨 num\_instance\_accm + num\_instance

 5: idf\_vec 🡨 log (num\_instance\_accm/df\_vec\_accm)

 6: file 🡨 idf\_vec

 7: file 🡨 df\_vec\_accm, num\_instance\_accm



## Incremental Training

### The Models

For the incremental training, we use partial\_fit method in some of the models, .e.g. MultinomialNB, SGDClassifier with Perceptron/SVM/LR, etc. Same as section 2.5, parameters tuning is the main effort to spend on.

### The Intermediate Trained Model

In section 2.5.2 we use onnx api to save the trained model for prediction. However, onnx object cannot be reversed back for training again. Here we use the joblib method to dump the whole training object to file (\*.object in results/persist) and reload it back when we train the model using the next batch of data. At the same time, onnx object is saved (\*.onnx in result/persist) for prediction and model distribution.

# Old School System

## Overview

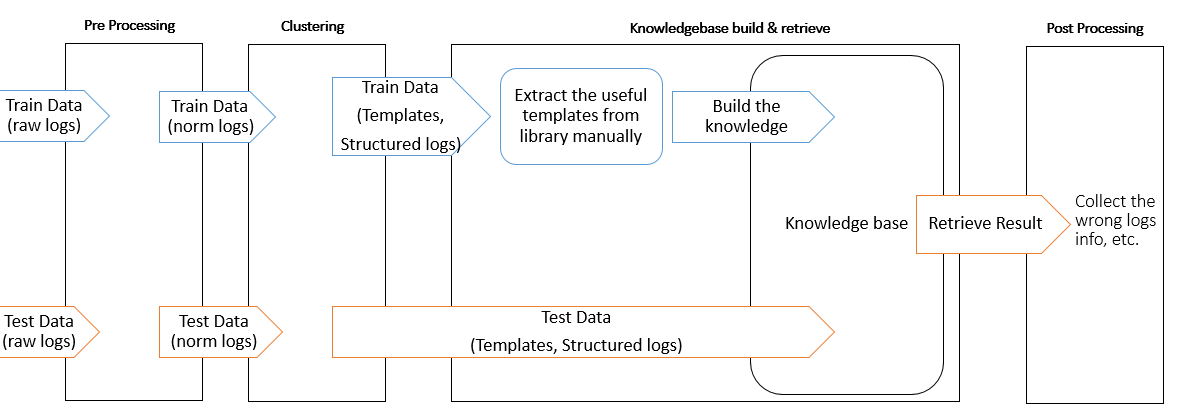


Figure : Old School System (OSS)

The old school system (OSS) shares the same blocks of pre-processing and clustering as the learning system. We manually review the template library and extract the useful templates to build the knowledge base. Usually templates of a system might have several hundreds or thousands; however, we only need review the templates that show the system errors. This greatly decreases our efforts.

## Pre-Processing and Clustering

Share same design with sections 2.2, 2.3 and 3.2

## The Knowledgebase

Table below is a snippet of template library file. The EventId is the unique representation of the template as it is the hash value of the corresponding template string. We store the templates we are interested in knowledgebase. Each item in the knowledgebase is retrieved by the eventId.

|  |  |  |
| --- | --- | --- |
| EventId | EventTemplate | Occurrences |
| 6b0ae484 | Ofdm0: profile0 LOCK in <\*> ms! | 8 |
| c481c3c2 | Telling application we lost lock on QAM channel <\*> | 64 |
| b4a1c2e6 | NumDsChans = <\*>, NumUsChans = <\*> | 39 |
| 594a8f8e | Restoring OFDM DS MAC settings | 2 |
| a82bff10 | Initializing Quarantine D31 DS MAC structures to <\*> | 2 |

### The Knowledge Item without Parameters

If the item has no parameters, it is usually a string contains something like “ERROR”, “Failure”, etc. If one log match this item, we can immediately tell the analyzer something wrong.

### The Knowledge Item with Parameters

Each <\*> is a parameter in the item, which we use to compare with some threshold or binary values. That is to say, if some log matches the template and then we can use parameter values to decide if the log is good or not.

## Extract the Parameters from Log

### The Log Format in Structured Logs File

Below is an example of one log in the structure logs file. Column 1 is eventId, column 2 is raw log and column 3 is the corresponding template.

|  |  |  |
| --- | --- | --- |
| b4a1c2e6 | NumDsChans = 32 NumUsChans = 10 | NumDsChans = <\*>, NumUsChans = <\*> |

### Parse the Parameters

Algorithm 4-1 in pseudo code

 1: idx\_list 🡨 Traverse all <\*> in logEventTemplateL

 2: **FOR** idx **IN** idx\_list

 3:     param\_list 🡨 append(logContentL[idx])

 4: **END**

logEventTemplateL is the template while logContenL is the raw log, both of which are extracted from the structured file.

## Retrieve the Knowledgebase

With the eventId and extracted parameters, we can retrieve the knowledgebase to see if the current log is good or not.

## Post-Processing

Save error log timestamp, description and suggestion to summary file in analysis\_summary.csv under /results/test/.

# Unsupervised Learning System

## Overview

Paper below discussed the anomaly detection with unsupervised deep learning. The DeepLog system includes three modules: Log Key Anomaly Detection model, Parameter Value Anomaly Detection model, and Workflows model. The author claimed that it resolved several issues that other ML based models were confronting by 1) per log entry level instead of session or instance entry level, 2) an extra model for parameters, 3) workflow model for diagnosing after anomalies detected. Additionally, comparing to the design of section 2, we need no longer label the training dataset.

Referenced paper

 [CCS'17]  Min Du, Feifei Li, Guineng Zheng, Vivek Srikumar.

DeepLog: Anomaly Detection and Diagnosis from System Logs through

Deep Learning, 2017

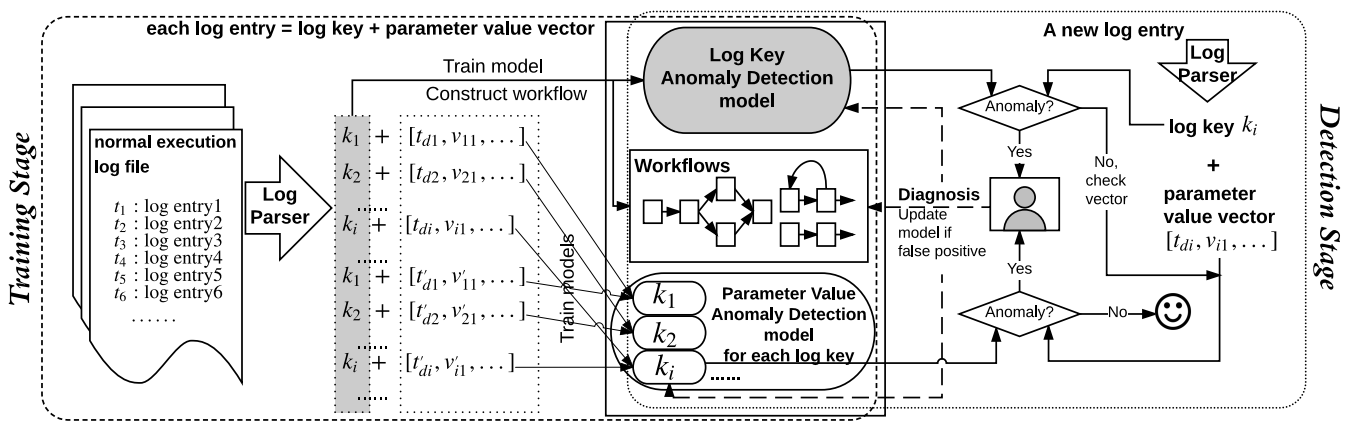


Figure : DeepLog Architecture

We implement DeepLog based on our former works of pre-processing (section 2.2) and clustering (sections 2.3 and 3.2) because DeepLog regards a log template (aka log key in paper) as a natural language word associated with a key ID. The deep learning framework is PyTorch by default but it should be easy porting to Karas / TensorFlow.

## Execution Path Anomaly Detection

This model is a multi-class classifier over recent context. The class set represents the complete library of templates (log keys). The model input is a history of recent log templates, and the output is a probability distribution over the log keys from, representing the probability that the next log key in the sequence is a key. For real log producing system, the size of might vary from hundreds to thousands.

DeepLog uses LSTM as the model.

### Preceding Blocks

We directly use the clustering results: train\_/test\_norm.txt\_structured.csv and train\_/test\_norm.txt\_templates.csv. For the training dataset, we suppose all the logs are normal, and use the training file without labels (aka ‘abn: ‘) for preprocessing. For the target label vector for the multi-class classification training, we will derive it in section 5.2.3. For the verification, let us use the dataset with labels.

Add top-level scripts in the entrance folder to do the manipulation of data per description above. The preprocess\_cm.py need some changes to accommodate the requirement. 1) Calculate the line mapping between raw and norm files, see section 5.2.8. 2) Add the session labels at the starting line in each log file. 3) Add the session label for some special logs. 4) Extract the session labels and save to a file for later use, e.g. sequence slicing, see section 5.2.3.

### Log Key (aka Word in NLP Parlance) Representation

Usually we create a high dimensional embedding vector for each word by a separate training process in NLP with neural networks. The DeepLog paper indicates using one-hot vector. It probably works too by mapping each log key to an integer. Theoretically the word embedding method is the best but it needs us train a complete template library. The integer log key id method is the simplest and space/time saving but needs verify it has similar accuracy as the one-hot vector. We will use integer representation in the implementation firstly and will verify the one-hot vector once prediction done. The word-embedding vector acts as a long-term plan of improvement after the whole DeepLog system done.

The template ID (aka event ID, log key ID) is 32 bit hash value of its template string in section 2.3.3. The hash values have big dynamic range w/o normalization. Considering we will try both integer and one-hot vector representations, let us reuse the idea of section 3.3. We define a word (aka template, event, log key) vocabulary with the size, e.g. 2000. Then load it with the entire template IDs in library and shuffle them. If a new template gets into the template library or an existing one is updated, the vocabulary (a list called vocab) needs get update accordingly.

Algorithm 5-1 in pseudo code

See Algorithm 3-4, 3-5 and 3-6

We load the eventID of each log in structured file into a list called event\_mapping\_data, and then we can map each eventID to a one-based integer in the process of slicing log sequence.

For the one-hot vector, we can deduce it from the same vocabulary.

### Log Sequence Representation

Basic algorithm for slicingWe used timestamps for windowing (window size / step size) in the supervised learning system. For DeepLog model, we use log indexes. E.g. we set the window size (aka sequence length) as 10 (WINDOW\_SIZE = 10), which means we use 10 recent logs to predict the next one.

Based on the logs (we regard them as being in a long sequence) in the structure file of clustering, we slice them into sequences with a predefined window size and step size (default as 1 per the DeepLog paper).

Algorithm 5-2 in pseudo code to slice the sequence

 1: corpus\_sequence 🡨 eventID from structured log file

 2: corpus\_sequence 🡨 replace eventID with int by retrieving vocab

 3: **WHILE** (i + WINDOW\_SIZE) < len(corpus\_sequence)

 4:  log\_sequences 🡨 append(corpus\_sequence[i: i + WINDOW\_SIZE])

 5:  target\_events 🡨 append(corpus\_sequence[i + WINDOW\_SIZE])

6:  validate\_labels 🡨 append(check\_curr\_sequence\_label())

7:  i++

8: **END**

log\_sequences:

[15, 3,  9,  5, 13, 21,  3,  9, 15,  8]

[ 3, 9,  5, 13, 21,  3,  9, 15,  8, 21]

[ 9, 5, 13, 21,  3,  9, 15,  8, 21,  9]

…

The target event of the first/second sequence is 21 and 9 respectively, which will act as the expected output of the model in the Loss / Cost function. The validate labels are only used for validation dataset then we can calculate the metrics of the model. If any of the word in the current sequence (including the target word) has abnormal label, label the completely current sequence as abnormal.

We also need know the log line number of each predict line, and then we can trace which log has error in predict.

Algorithm 5-2 is only the basic algorithm, and we have to consider the session scheme. We suppose the log producer system always outputs a single session logs although it might always have multi-threads. If the system reboots or has some commands to run and then has different behavior, e.g. the goto\_ds commands in Cable Modem, we take the logs as different sessions. The sequence slicing or say windowing should not across the session boundary for train and validation. However, we do not consider the sessions for prediction, and so we expect the model will detect the sequence across session boundary as anomaly.

Therefore, we need session labels to tell train and validation to slice the sequence within each session correctly. We do the session labeling at two places. Firstly, we add the session labels at the starting line in each log file when concatenate them together as one monolith, because we think the logs in different files must not be in one session. Secondly, we mark some special logs as session indicators when preprocessing the raw logs in logparser. After tagging the session labels, we will extract them as a list, e.g. each element represents the session size, and save it to Results/train/train\_norm.txt\_session.pkl for slicing. All these done in preprocessor of logprser. At last, we consider the session size based on the basic algorithm 5-2 for slicing.

### Data Loading

Before feeding the model, we need furthermore organize all the data in some specific data structures to accommodate the multi-process data loading, mini-batch data processing, etc., under the learning framework.

Three kinds of data here, that is, sequence, sequence id, target, and validation label. At first step, all the data are in the python dictionary as below.

Aggregate all the input data:

 1: step1\_dict = {"SeqIdx" : 1-D array, zero-based, sequence index in the logs

 2:               "EventSeq": 2-D array, 1st dim is sequences, 2nd is a window of events

 3:               "Target" : 1-D array, [0, num\_classes-1], class index

 4:               "Label" : 1-D array, 0/1, target label for each sequence

 5:              }

The size of all arrays’ first dimension is the number of sequences in the logs.

This data structure cannot go into the model yet and need further change to an iterate-able (aka train per mini-batch, multi-process loading, etc) structure with the help of framework. First, we let the framework retrieve the data in step1\_dict sample-by-sample (if mini-batch is 1), aka sequence-by-sequence. Subclass the framework Dataset and overwrite some methods. The *index* below represents a complete sample dict (aka SeqIdx + EventSeq + Target + Label). The returned *dataloader* is the final iterator that train/evaluate will use with mini-batch/multi-processing enabled.

Algorithm 5-3: Enable the framework to iterate the data per sample, mini-batch

 1: ...

 2: keys = list(step1\_dict.keys())

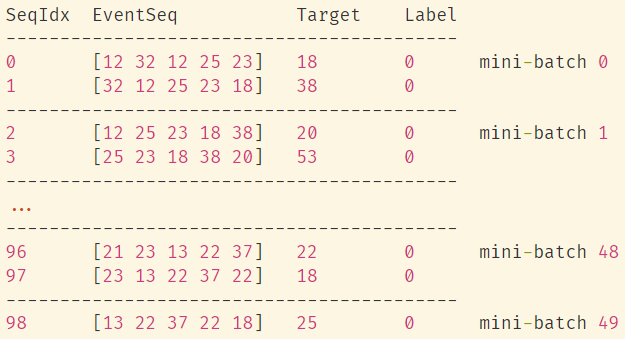
 3: return {k: step1\_dict[k][index] for k in keys}

4: ...

5: dataloader = framework.DataLoader(step1\_dict, batch\_size= …)

After the framework processing, each iteration on the *dataloader* returns a mini-batch size samples (aka a dict). The value parts in the dict are tensors. Each mini-batch is a dict. In the example below the batch size is 2. The first dimensions of all value parts must be same (aka equals batch size). The EventSeq of each mini-batch is a 2-D tensor with size *(batch\_size x seq\_len)*. Note the last batch is a different animal whose 1st dim might be < batch\_size.

dataloader structure



### The Model

The Figure 5 (captured from DeepLog paper) shows the LSTM structure, which set it as two layers. The deep learning framework provides the basic model implementations like RNN and LSTM. We need subclass the base model and overwrite the \_\_init()\_\_ and forward() functions. For the prediction layer we can select nn.Linear.

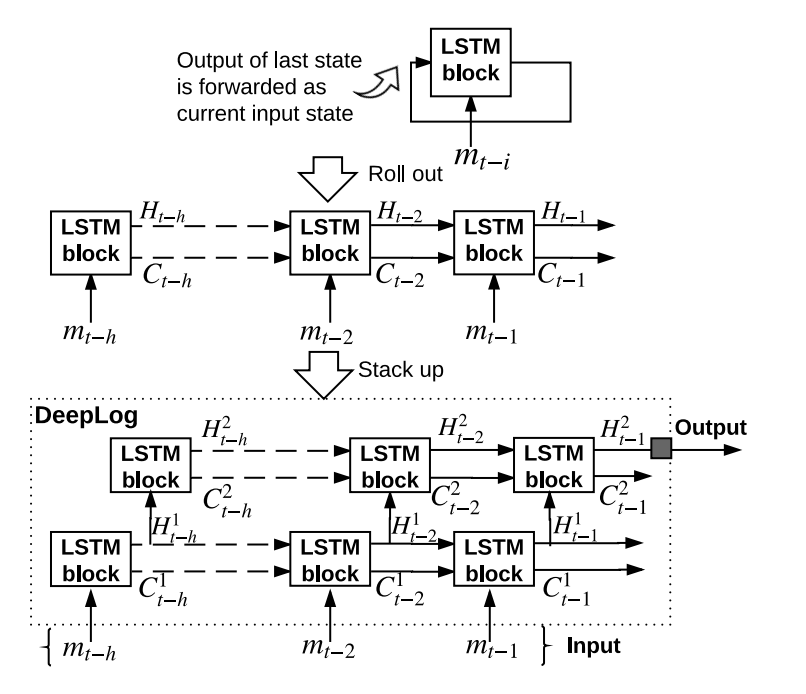


Figure : Stacked LSTM

The LSTM model (nn.LSTM) input is a 3-D tensor with the size *(batch\_size x seq\_len x input\_size)*. In the current implementation, the LSTM cell input (aka in Figure 5) is event index (an integer) so the cell input dimension is one. For the case of one-hot vector or word-embedding cases, it will be different.

The LSTM model (nn.LSTM) output is a 3-D tensor with the size *(batch\_size x seq\_len x num\_directions\*hidden\_size)*. The LSTM cell output (aka in Figure 5, hidden state) dimension is hidden\_size. Besides the output tensor, nn.LSTM also returns hidden state (h\_t) and cell state (c\_t) at time seq\_len, and the dimensions are *(num\_layers \* num\_directions, batch, hidden\_size)*. Note that the nn.LSTM output [:, -1, :] is the last word (in one sequence) hidden state, which has the same values as the 2nd layer h\_t.

By the way, the LSTM model by default defines the input/output tensor shape as *(seq\_len x batch x input)* but it is more natural for the shape *(batch x seq\_len x input)*. For this reason, provide the batch\_first parameter when instantiating the LSTM class. This does not affect the (h\_t, c\_t) part of input/output of LSTM model.

The prediction layer (nn.Linear) input connects to LSTM hidden state, and we only need the last hidden state output from nn.LSTM. So feed nn.LSTM output [:, -1, :] to nn.Linear.

The prediction layer (nn.Linear) output is a 2-D tensor with the *size (batch\_size x num\_classes)*. The num\_classes is the total number of classifications. It is a question as how to select the number. Considering the future of online update of model, it looks good to give it the max size TEMPLATE\_LIB\_SIZE. In this case, many dummy classes have no event index at all.

### Training and Evaluation

Training process follows the steps in framework.

Algorithm 5-4 in pseudo code: Training

 1: model 🡨 DeepLogExec()

 2: criterion 🡨 CrossEntropyLoss() Loss func combining LogSofmax and NLLLoss

 3: optimizer 🡨 Adam()

 4: batch\_cnt 🡨 len(train\_data\_loader)

 5: **FOR** epoch **IN** range(NUM\_EPOCHS)

 6:     epoch\_loss 🡨 0

 7:     **FOR** batch\_in **IN** train\_data\_loader

 8:         *# Forward pass*

 9:         seq 🡨 (batch\_in['EventSeq'] convert to 3-D tensor)

10:         output 🡨 model(seq)

11:         loss 🡨 criterion(output, batch\_in['Target'])

12:

13:         *# Backward pass and optimize*

14:         optimizer.zero\_grad()

15:         loss.backward()

16:         epoch\_loss 🡨 epoch\_loss + loss.item()

17:         optimizer.step()

18:     **END**

19:     epoch\_loss 🡨 epoch\_loss / batch\_cnt

20: **END**

The batch\_in[‘EventSeq’] in dataloader is 2-D tensor *(batch\_size x seq\_len)*. Before feed it to model, convert it to 3-D *tensor (batch\_size x seq\_len x input\_size)* per 5.2.5.

The model output is 2-D tensor *(batch\_size x num\_classes)*, which will be the first parameter of loss function. The second parameter is class index of target event, which is 1-D tensor with size batch\_size. The values of it must be within [0, num\_classes-1].

Algorithm 5-5 in pseudo code: Evaluation

 1: anomaly\_pred 🡨 []

 2: TP, TN, FP, FN 🡨 0

 3: model 🡨 enable evaluation

 4: tensor 🡨 disable gradient calculation

 5: **FOR** batch\_in **IN** data\_loader

 6:     seq 🡨 (batch\_in['EventSeq'] convert to 3-D tensor)

 7:     output 🡨 model(seq)

 8:     pred\_sort 🡨 sort the output per probability high to low

 9:     seq\_target 🡨 batch\_in['Target']

10:     seq\_label 🡨 batch\_in['Label']

11:     **FOR** i **IN** range(realtime batch size)

12:         top\_idx 🡨 pred\_sort[i].index(seq\_target[i])

13:         **IF** seq\_label[i] == 1

14:             **IF** top\_idx >= TOPK

15:                 TP++

16:                 anomaly\_pred 🡨 append(1)

17:             **ELSE**

18:                 FN++

19:                 anomaly\_pred 🡨 append(0)

20:             **END**

21:         **ELSE**

22:             **IF** top\_idx >= TOPK

23:                 anomaly\_pred 🡨 append(1)

24:                 FP++

25:             **ELSE**

26:                 anomaly\_pred 🡨 append(0)

27:                 TN++

28:             **END**

29:         **END**

30:     **END**

31: **END**

After enable evaluation and disable gradient calculation, model() will output the probability of each class in the output layer. The line 8 above sorts the probability of each class from high to low and saves the corresponding class indexes to pred\_sort[] for a whole batch. The line 12 retrieves the index in the pred\_sort[] for each sequence target, aka top\_idx, which represents the prediction result. Define TOPK as a threshold (e.g. 10) for the classification. The anomaly\_pred[] saves the final results for each sequence.

With TP, FP, TN and FN, we can do some metrics to evaluate the model. See 2.6.1.

### Prediction

Compare to evaluation, there are no metrics calculation. Besides anomaly\_pred[] for each sequence, we also calculate the corresponding anomaly line number in the norm file.

Algorithm 5-6 in pseudo code: prediction

 1: j 🡨 0

 2: anomaly\_pred 🡨 []

 3: anomaly\_line 🡨 []

 4: model 🡨 enable evaluation

 5: tensor 🡨 disable gradient calculation

 6: **FOR** batch\_in **IN** data\_loader

 7:     seq 🡨 (batch\_in['EventSeq'] convert to 3-D tensor)

 8:     output 🡨 model(seq)

 9:     pred\_sort 🡨 sort the output per probability high to low

10:     seq\_target 🡨 batch\_in['Target']

11:     seq\_label 🡨 batch\_in['Label']

12:     **FOR** i **IN** range(realtime batch size)

13:         top\_idx 🡨 pred\_sort[i].index(seq\_target[i])

14:         **IF** top\_idx >= TOPK

15:             anomaly\_pred 🡨 append(1)

16:             anomaly\_line 🡨 append(i+1+WINDOW\_SIZE+j\*BATCH\_SIZE)

17:         **ELSE**

18:             anomaly\_pred 🡨 append(0)

19:         **END**

20:     **END**

21:     j++

22: **END**

### Post-Processing

The anomaly\_line[] in 5.2.7 stores the anomaly line number in the norm log file. However, from the raw log file to the norm file, we have some lines/logs removed and some lines combined as one line. We need a mapping between the two log files to let us go back to the raw file.

We actually do this by changing the pre-processing block in section 2.2. Firstly, recording the line mapping between raw file and the new file. Secondly, use this map to calculate the final mapping between raw and norm file in Algorithm 2-3 for time and memory efficiency.

The final prediction results are in /results/test/anomaly\_result.txt.

### Online Update of Exec Model

The DeepLog paper section 3.3 uses incremental model update to resolve the false positive detection. The general online or incremental learning has catastrophic forgetting issue although there are various strategies to combat the accuracy downgrade. It is not cost-efficient by scarifying the accuracy because online update of model is not necessary to be must there in our use case.

We update the model offline with the accumulated train dataset.

## Parameter Value Anomaly Detection

DeepLog trains a separate LSTM model for each distinct log key (template) that has parameters.

## Workflow Construction

TBD.

# Real-Time Prediction

## Overview

It is possible to make the prediction of supervised learning system or the old school system (OSS) to be real-time, say, do one prediction or do old school job every 10~30 seconds on the upcoming logs. Usually as long as there is at least one instance (or sample), the prediction will work. For Cable Modem / DOCSIS logs, one instance includes 10 seconds logs by default. For the OSS, it just needs at least one log to work.

The DeepLog is log level prediction like OSS and so it is more suitable to do real-time prediction than our supervised implementation, which is at instance level. We will try to design the real-time system based on DeepLog and OSS.

We put a Sampler block before pre-processing to fetch the logs in real-time. As to how to fetch, one way is through telnet connection to the log producing system like CM. The real-time log prediction (log consuming system) runs on PC or RG.

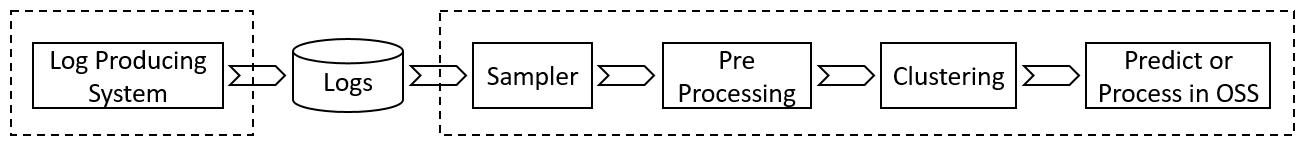


Figure : Real-time Prediction

## Sampler

Define a logs FIFO buffer inside Sampler firstly. Create a helper thread to continually read the new logs from the Log Producing System into the buffer. The main thread read one complete log each time and send it to pre-processing block.

### Helper Thread

The helper thread continuously read each line to the buffer. Define a buffer size and drop the logs when buffer is full.

### Fetch Data from Buffer

When sampling the logs, do not split the multi-line log or table. In other words, feed a complete multi-line log to pre-processing block.

## Pre-Processing

Pre-processing block need not change as it processes the logs line by line. However, for the integrity of some log, e.g. a table or a multi-lines log, the Sampler should account for providing a complete log.

## Clustering

The incremental clustering algorithm accepts as few as one log. The algorithm in section 3.2 can be used directly without any changes. However, each time we run it, the template library will be loaded into memory to rebuild the tree even though there is only one input log. Although it is not time consuming (e.g. no more than one thousand templates), it might be better have the library being in memory always since the first logs sample comes. This requires a big change of current code structure. Do this optimization after the whole real-time prediction system works.

## Test Data Vectoring in DeepLog

This should be different from the offline prediction version of DeepLog.

TBD for details.

## Prediction and the Old School System

Should be same as offline prediction version of DeepLog and OSS.

## Post-Processing

We save the timestamp tuples of anomaly (Prediction) or timestamp of error lines / descriptions/suggestions (OSS) in files. This is as same as 2.7 except replacing overwrite with append. How to represent the result is trivial and dependent on the requirement.

# Appendix

## Labeling Assistant

Although we usually label the train dataset manually but we still can label some known error logs automatically. See the /tools/labelassist.py. To add more regular expressions for more known anomaly logs.

## Consideration about the Log on Cable Modem

Logs can be accidently messed up by multi thread printings. The situation is severely worse when hundreds of thousands lines are logged. To recover the messed up logs, it is time consuming, especially for the training dataset.

## Known Issues and Further Improvements

### Variable Template Classification

For the anomaly detection, the template that has parameters might not contribute for the leaning, as the error attributes are lost without the real values of parameters. Possibly, we can use the knowledgebase to classify the variable templates into good and bad ones. This needs further study.

See section 5, which gives us a solution in the DeepLog paper of ccs’17.

### Need Boost the Partial Fit Precision

As partial fit does not randomize the instances of a batch and does only one epoch training, the fit precision will be slightly lower than the non-partial one when there are not enough batches of train dataset. Need further study.

### Case of Sparse Feature Vector

E.g., only one log is anomaly within one instance/window. Other features are no helpful or related to this anomaly. Need further study.

### If Very Few Logs are Contained in an Instance/Window

Dynamic window size? Step size will be dynamic accordingly.

Should we use line number to calculate window instead of timestamp? How about the correlation between logs? The last instance/window usually contains fewer logs.

## Integration with Boardfarm Test System

### Adapting Logs from Boardfarm to Preprocessor

The logs from Boradfarm system have ‘^M’, which comes from the extra Carriage return (0xD) char. It introduces most of the messes. We firstly remove them by command below:

*sed -e "s/\r//g" cm\_console.log > test\_boardfarm.txt*

Next, we need do three things before feed the boardfarm logs to preprocessor in logparser.

*# 1. Remove the abnormal timestamp with the main timestamp*

*# 2. Add the main timestamp if both main and abnormal timestamps don’t exist*

*# 3. Remove the prompt 'CM> ', which might be anywhere in the line*

Algorithm 7-1 in pseudo code (rev 0.2)

 1: curline\_ts 🡨 '[19700101-00:00:00.000] '

 2: **FOR** line **IN** rawfile

 3:     *# Save the main timestamp*

 4:     match\_ts 🡨 pattern\_ts.match(line)

 5:     **IF** match\_ts

 6:         *# Match the main timestamp*

 7:         curline\_ts 🡨 match\_ts.group(0)

 8:         newline 🡨 pattern\_ts.sub('', line, count=1)

 9:     **ELSE**

10:         match\_abn\_ts 🡨 pattern\_abn\_ts.match(line)

11:         **IF** match\_abn\_ts

12:             *# Match the abnormal timestamp from boardfarm system*

13:             newline 🡨 pattern\_abn\_ts.sub('', line, count=1)

14:         **ELSE**

15:             *# Both main and abnormal timestamp do not exist*

16:             newline 🡨 line

17:         **END**

18:     **END**

19:     *# Remove console prompt*

20:     **IF** pattern\_pt.search(newline)

21:         newline 🡨 pattern\_pt.sub('', newline)

22:     **END**

23:     *# Write current line to a new file with the timestamp*

24:     newline 🡨 curline\_ts + newline

25:     newfile.write(newline)

26: **END**

## Recover messed up logs (for CM prediction only)

### The Cause

The Event Log thread has a lower priority (25) comparing to other normal priority (23) threads, so the normal priority log sometimes interrupts the one from this Event Log thread. The logs from the Event Log thread have a fixed pattern format that starts with “Logging event: ”.

The messed logs usually have two parts with several lines of logs in between.

Typical case 1:

Line 1: [20200827-01:09:28.477] LBcmCmUsChan:: UsTimeRefOk: …

…

Line x: [20200827-01:09:28.493] ogging event: DBC-REQ denied -

Typical case 2:

Line 1: [20200827-01:09:28.477] L CmSnmpAgent:: Re …

…

Line x: [20200827-01:09:28.493] ogging event: DBC-REQ denied –

Typical case 3:

Line 1: [20200827-01:09:28.477] Log

…

Line x: [20200827-01:09:28.493] ging event: DBC-REQ denied -

There are several other cases but are difficult to handle. We consider the most typical cases above.

### Assumptions

The original log o2 breaks log o1 into two pieces, that is, log m1 and m2. In the typical case, the o1 is “Logging event: DBC-REQ denied - …” and o2 is “BcmCmUsChan:: UsTimeRefOk: …”. The corresponding m1 is at Line 1 and m2 is at Line x. There are other good logs between Line 1 and Line x.

1. The template library contains both log o1 and log o2.

2. The logs in between are all good and known logs to library too.

### Algorithm

In the clustering Drain, we add an extra column called EventIdOld to the structured log file. Then, we add extra phase to recover the messed logs before predicting.

We load the templates from the structured file and scan them to look for the new template by comparing the old event id to the new one. If we find one, then remove the head character and calculate the hash value (template id) of the remaining string. If the new calculated id already exists in the template library, then we get the correct log o2 and stop slicing the log m1, else slicing off one heading character again from the substring. If no log o2 exists at the end, we assume m1 is a new template. We save the head string slicing off from m1 for future use.

If we successfully separate o2 from m1, we suppose the next new template will be the second part of o1. That is to say, o1 = head\_string\_of\_m1 + m2.

Algorithm 7-2 in pseudo code (rev 0.1)

1: temp\_lib 🡨 load event id from library

2: log\_rows 🡨 load rows from test structured file

 3: m1\_found 🡨 False

4: o1\_head 🡨 ‘’

 5: **FOR** idx, row **IN** enumerate(log\_rows)

6: head\_l 🡨 bool(head\_char(row[‘EventTemplate’]) == ‘L’)

 7:    **IF** (row[‘EventIdOld’] != ‘0’) **OR** (!m1\_found **AND** !head\_l)

8:     write\_to\_new\_file 🡨 row[‘EventTemplate’]

9:      **CONTINUE**

10:     **END**

11:     **IF** m1\_found

12:      **IF** row[‘EventIdOld’] == ‘0’

13:      o1 🡨 o1\_head + row[‘EventTemplate’]

14:      write\_to\_new\_file 🡨 o1

15:      m1\_found 🡨 False

16:      **END**

17:      **CONTINUE**

18:     **END**

19:     chars\_row 🡨char\_list(row[‘EventTemplate’])

20: **FOR** i **IN** range(chars\_row)

21: o1\_head 🡨 chars\_row[0: i]

22: o2 🡨 cut off o1\_head from chars\_row

23: event\_id\_o2 🡨 hash(string(o2))

24: **IF** event\_id\_o2 **IN** temp\_lib

25: m1\_found 🡨 True

26: write\_to\_new\_file 🡨 o2

27:  **IF** event\_id\_o2 == ‘SPECIAL1’

28: O1\_head 🡨 remove two trailing spaces

29:  **END**

30:  **BREAK**

31: **END**

32: **END**

33: **IF** !m1\_found

34: m1\_found 🡨 True

35: **END**

36: **END**

Line 28 is for case 2 and Line 34 for case 3, wherein we do not write anything for case 3.

For case 3, we remove the Line 1 in the new file for case 3, so we need update the raw / norm line number-mapping vector as below:

Algorithm 7-3 in pseudo code

1: load skipped\_line\_list (we get from Algorithm 7-2 for case 3)

2: reverse the elements in skipped\_line\_list

 3: load raw\_norm\_line\_mapping\_list

 4: **FOR** i reverse(skipped\_line\_list)

5: raw\_norm\_line\_mapping\_list.pop(i)

6: **END**

At last, we run a simplified version of Drain2 on the recovered norm log (actually an intermediate version of templates for each log).