Proprietary and Confidential Information

Revision History

|  |  |
| --- | --- |
| **Date** | **Change Description** |
| Mar-23-2020 | Initial draft |
|  |  |

Table of Contents

[1 Introduction 1](#_Toc38469410)

[2 The Learning System 1](#_Toc38469411)

[2.1 Overview 1](#_Toc38469412)

[2.2 Pre-Processing 2](#_Toc38469413)

[2.2.1 About the Timestamp and Labels in Train Dataset 2](#_Toc38469414)

[2.2.2 Rectify Various Logs 2](#_Toc38469415)

[2.2.3 Format the Tables 3](#_Toc38469416)

[2.2.4 Convert Multi-Line Log to One-Line Format 4](#_Toc38469417)

[2.2.5 Extract the Label Vector from Train Dataset 4](#_Toc38469418)

[2.3 Clustering 5](#_Toc38469419)

[2.3.1 Template 5](#_Toc38469420)

[2.3.2 Algorithm of Drain 6](#_Toc38469421)

[2.3.3 Template ID 8](#_Toc38469422)

[2.3.4 The Application Layer of Drain 8](#_Toc38469423)

[2.4 Feature Extraction 9](#_Toc38469424)

[2.4.1 Event Count Matrix 9](#_Toc38469425)

[2.4.2 Windowing 9](#_Toc38469426)

[2.4.3 Template ID vs. Template Index 11](#_Toc38469427)

[2.4.4 Event Count Matrix Constructing 11](#_Toc38469428)

[2.4.5 Tf-Idf 12](#_Toc38469429)

[2.5 Training 13](#_Toc38469430)

[2.5.1 The Models 13](#_Toc38469431)

[2.5.2 Preservation of Trained Models 13](#_Toc38469432)

[2.6 Prediction 14](#_Toc38469433)

[2.6.1 Validation Metrics 14](#_Toc38469434)

[2.6.2 For Distribution 14](#_Toc38469435)

[2.7 Post-Processing 14](#_Toc38469436)

[3 Incremental Learning 15](#_Toc38469437)

[3.1 Overview 15](#_Toc38469438)

[3.2 Incremental Clustering 15](#_Toc38469440)

[3.2.1 Template Library 16](#_Toc38469441)

[3.2.2 Convert Drain to be Incremental 16](#_Toc38469442)

[3.2.3 Mark Templates Status 17](#_Toc38469443)

[3.2.4 Benefit to The Old School 18](#_Toc38469444)

[3.3 Incremental Feature Extraction 18](#_Toc38469445)

[3.3.1 Define a Size of Feature Set 18](#_Toc38469446)

[3.3.2 Initialization of Feature Set 18](#_Toc38469447)

[3.3.3 Update of Feature Set 19](#_Toc38469448)

[3.4 Incremental Tf-Idf 20](#_Toc38469449)

[3.4.1 The Tf 20](#_Toc38469450)

[3.4.2 The Idf 20](#_Toc38469451)

[3.5 Incremental Training 20](#_Toc38469456)

[3.5.1 The Models 20](#_Toc38469457)

[3.5.2 The Intermediate Trained Model 20](#_Toc38469458)

[4 The Old School System 21](#_Toc38469459)

[4.1 Overview 21](#_Toc38469460)

[4.2 Pre-Processing and Clustering 21](#_Toc38469461)

[4.3 The Knowledgebase 21](#_Toc38469462)

[4.3.1 The Knowledge Item without Parameters 22](#_Toc38469463)

[4.3.2 The Knowledge Item with Parameters 22](#_Toc38469464)

[4.4 Extract the Parameters from Log 22](#_Toc38469465)

[4.4.1 The Log Format in Structured Logs File 22](#_Toc38469466)

[4.4.2 Parse the Parameters 22](#_Toc38469467)

[4.5 Retrieve the Knowledgebase 22](#_Toc38469468)

[4.6 Post-Processing 22](#_Toc38469469)

[5 Real-Time Prediction 23](#_Toc38469470)

[5.1 Overview 23](#_Toc38469471)

[5.2 Sampler 23](#_Toc38469472)

[5.2.1 Helper Thread 23](#_Toc38469473)

[5.2.2 Fetch Data from Buffer 23](#_Toc38469474)

[5.3 Pre-Processing 23](#_Toc38469475)

[5.4 Clustering 24](#_Toc38469476)

[5.5 Feature Extraction in Learning System 24](#_Toc38469477)

[5.6 Prediction and The Old School System 24](#_Toc38469478)

[5.7 Post-Processing 24](#_Toc38469479)

[6 Appendix 24](#_Toc38469480)

[6.1 Labeling Assistant 24](#_Toc38469481)

[6.2 Consideration about the Log on Cable Modem 24](#_Toc38469482)

[6.3 Known Issues and Further Improvements 25](#_Toc38469483)

[6.3.1 Variable Template Classification 25](#_Toc38469484)

[6.3.2 Need Boost the Partial Fit Precision 25](#_Toc38469485)

[6.3.3 Case of Sparse Feature Vector 25](#_Toc38469486)

[6.3.4 If Very Few Logs are Contained in an Instance/Window 25](#_Toc38469487)

# 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, 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 in the training dataset before. It is a big challenge because of the characteristics of logs as dataset.

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

# The Learning System

## Overview

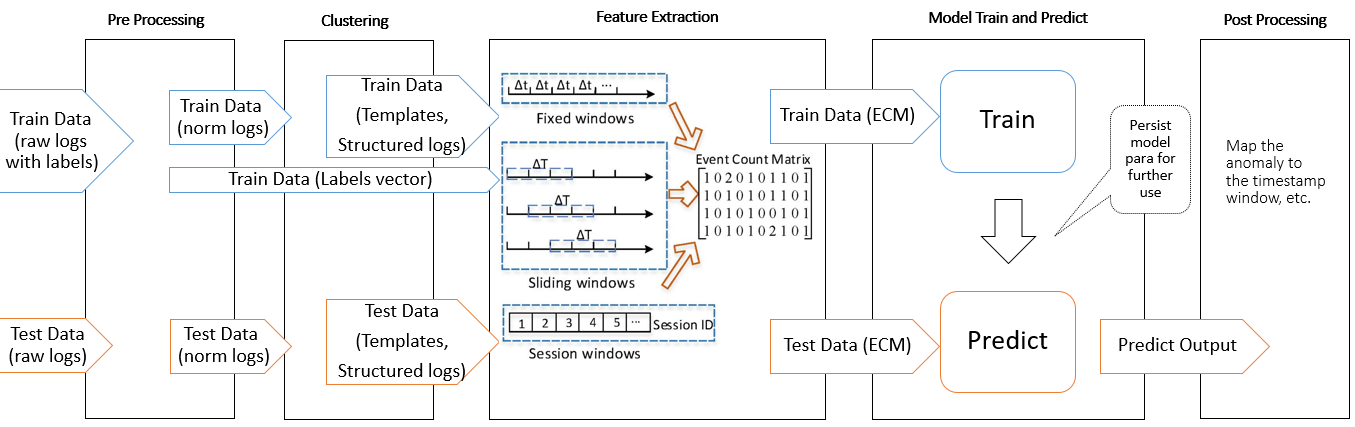


Figure 1: Learning System Blocks

The system 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 do deal with some known anomalies. We might also need merge multiple log files into one big train dataset in advance.

## 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 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 we want to convert more logs, we just need provide more regular expression of the candidates to list below.

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

There are various tables that 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 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 are in logs/ directory.

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

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

Algorithm 6 (revised)

 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:         if token1 == '<\*>':

15:             numOfPara += 1

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

17:             *# Paper: continue*

18:         if token1 == token2:

19:             simTokens += 1

20:         *# Do not accept seq2 if some special tokens are different*

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

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

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

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

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

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

27:                 sTokenNoMatch = 1

28:                 break

29:         if sTokenNoMatch:

30:             break

...

Algorithm 6 is the core to calculate the token similarity.

***Revised 1***: Line 17, we count the <\*> in the template when calculate similarity. ***Revised 2***: Line 23 ~ 30, to do something to prevent Drain from over-parsing some tokens. The corresponding regular expressions are defined in Drain application specific code, see section 2.3.4.

Algorithm 7 (revised)

 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 = 0.8

22:     newCluster.initst = newCluster.st

...

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

***Revised 1***: line 11, we will convert the digits in the Drain-app layer, so disable it here. ***Revised 2***: line 17/21, the adaptive threshold is not good so replace it with a static value. This is a heuristic value after many tests.

### 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 core Drain. 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 6 Revised 2.
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 are used to 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 don’t 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 which 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 be related to the context in the former time window. By using Sliding window, we can decrease the probability of this wrongly log partitioning.

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

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

12:         **BREAK**

13:     **END**

14: **END**

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 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 don’t care the order of features in the matrix. Some model like Random Forest randomly selects subset of features but doesn’t 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 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 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 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**

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

10:         **TRY**

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

12:         **EXCEPT**

13:             **CONTINUE**

13:         **END**

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

15:     **END**

16:     labels 🡨 append(label)

17: **END**

Algorithm 10 constructs the ECM. The *expanded\_indexes\_list* is the one we get in algorithm 9. The *label\_vector* is the one we get in section 2.2.5 algorithm 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 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 don’t 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 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 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 don’t fit into main memory. This also can be thought as an online or incremental learning method.

However before we can use the incremental training, we need to resolve the issue of feature set changing with the incremental input training dataset in advance.



## 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.

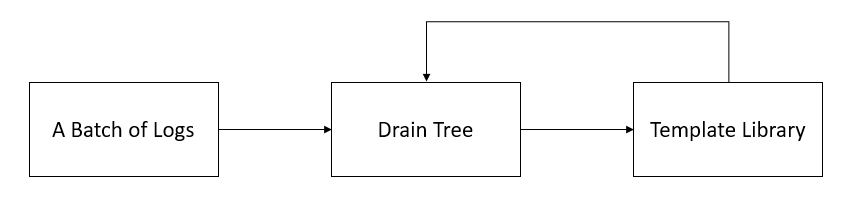


Figure 2: Incremental Drain

Algorithm 13 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 13 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 (*newCluster*, *existingCluster*) is a structure that has fields like template, logIDs that converged to the same template, 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 14 below. The second one will be explained in section 3.2.3.

Algorithm 14 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 doesn’t 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 15 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 13. So here in algorithm 15, 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 16 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 17 to update the STIDLE. Note that the new eventId is added to a randomized empty position in STIDLE.

Algorithm 17 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 18 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 19 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.

# The Old School System

## Overview

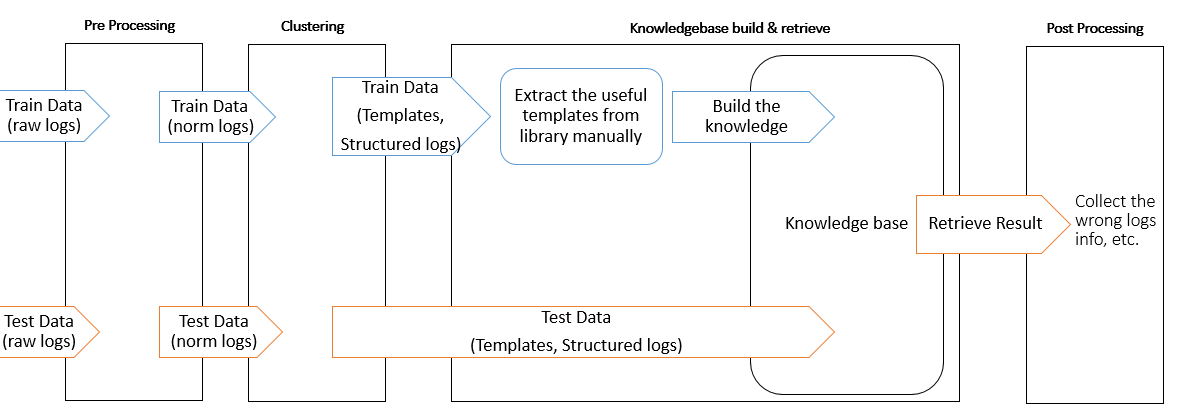


Figure 3: Old School System (OSS)

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 decrease 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 20 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/.

# Real-Time Prediction

## Overview

It is possible to make the prediction (not training) of learning system or the old school system 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 old school, it just needs at least one log to work. To share the pre-processing and clustering blocks, define a frontend block to sample the logs every 10 seconds.

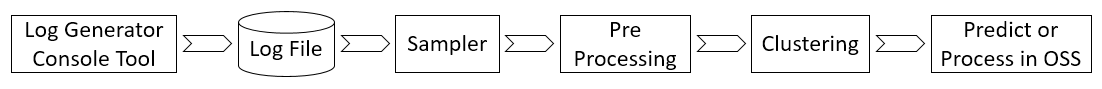


Figure 4: Real-time Prediction

## Sampler

Define a logs FIFO buffer inside Sampler firstly. Create a helper thread to continually read the new logs from the file (on the disk) into the buffer. Read a window (e.g. 10 sec) of logs from the buffer to feed the Pre-Processing block.

### Helper Thread

The Log Generator / Console Tool continually appends new logs to the file on the disk. The helper thread monitors the log file and read the new logs into the log buffer. To make the real-time meaningful, we drop the new logs when the buffer is full.

To monitor the log file changes, we can poll or use event handler (Select which one is TBD). Once file changes, load the new logs to buffer. Use the latest timestamp for the next round comparing.

### Fetch Data from Buffer

The window might contain very few logs, say one or two. This is a real issue for both training and predicting, see 6.3.4. It might help add a 2nd criterion like log set number. When sampling the logs, do not split the multi-line log or table within the window.

## Pre-Processing

Pre-processing block needn’t 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. Consider this optimization later when it is necessary.

## Feature Extraction in Learning System

Each time what we get from the sampler is an instance, which includes 10 seconds window logs. So we needn’t do windowing on these logs again and instead allocate them into one instance directly. The windowing algorithm in 2.4.2 can be bypassed for real-time prediction. There is only one tuple of *(start\_index, end\_index)*. The *start\_end\_index\_list* is *[(start\_index, end\_index)]*.

## Prediction and The Old School System

These blocks are not affected.

## 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 do the labeling manually but we still can label some known error logs automatically. See the /tools/labelassist.py. Add more regular expressions for more known logs.

## Consideration about the Log on Cable Modem

Logs are occasionally messed up by multi threads printings. We will see many logs are split when hundreds of thousands lines are logged. To recover the messed up logs, it’s 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 make any contributions 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.

### Need Boost the Partial Fit Precision

As partial fit doesn’t 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.

Use line number to calculate window instead of timestamp? How about the correlation between logs?

The last instance/window usually contains fewer logs.