Journal of The Korea Society of Computer and Information
Vol. 23 No. 10, pp. 11-21, October 2018
www.ksci.re.kr
https://doi.org/10.9708/jksci.2018.23.10.011

Accurate and Efficient Log Template Discovery Technique

Byungchul Tak*

Abstract

In this paper we propose a novel log template discovery algorithm which achieves high quality of discovered log templates through iterative log filtering technique. Log templates are the static string pattern of logs that are used to produce actual logs by inserting variable values during runtime. Identifying individual logs into their template category correctly enables us to conduct automated analysis using state-of-the-art machine learning techniques. Our technique looks at the group of logs column-wise and filters the logs that have the value of the highest proportion. We repeat this process per each column until we are left with highly homogeneous set of logs that most likely belong to the same log template category. Then, we determine which column is the static part and which is the variable part by vertically comparing all the logs in the group. This process repeats until we have discovered all the templates from given logs. Also, during this process we discover the custom patterns such as ID formats that are unique to the application. This information helps us quickly identify such strings in the logs as variable parts thereby further increasing the accuracy of the discovered log templates. Existing solutions suffer from log templates being too general or too specific because of the inability to detect custom patterns. Through extensive evaluations we have learned that our proposed method achieves 2 to 20 times better accuracy.

▸Keyword: Log template, Log analysis, Log parsing

I. Introduction

오늘날의 클라우드 어플리케이션들은 규모와 복잡도가 점점 증가하고 있으며 이에 따라 어플리케이션들을 관리하고 운용하는 일은 더욱 어려워지고 있다. 특히, 어플리케이션 동작에 문제가 발생하여 성능이 느려지거나 멈춘 경우, 문제의 원인을 신속하게 파악하여 이를 고친 후 재구동하는 일은 수익성과 직결되므로 매우 중요한 기술이다. 이러한 문제 원인 분석 (RCA: Root Cause Analysis) 기술은 대부분 어플리케이션과 운영체제에서 생성한 로그데이터에 의존한다 [1, 2, 3].

문제의 원인 분석 및 고장 복구에 사용하는 로그 정보는 실질적으로 유일하게 주어지는 정보이다. 상용 시스템들은 접근이 철저하게 제한되며 문제를 해결하려는 엔지니어에게는 대신 로그 데이터만이 주어진다. 하지만, 끊임없이 생성되는 로그 데이터의 양은 방대하므로 기존의 수동적인 로그 판독이나 키워드 검색으로는 원하는 결과를 얻기 불가능한 수준이다. 이러한 대량의 로그 데이터에 숨겨진 중요한 정보들을 찾아내기 위하여 다양한 시도들이 있어 왔다. 특히, 다양한 데이터 마이닝 기법과 인공 지능 기법, 그리고 최근에는 딥러닝(Deep Learning)을 적용하여 로그를 분석하는 연구 기법들이 제안되었다 [4, 5, 6, 7, 8, 23, 24, 25].

많은 양의 로그 데이터를 효과적으로 분석하여 유용한 정보를 찾아내기 위해서는 자동화된 방법을 사용하여야 한다. 로그 분석 자동화의 첫 단계는 각각의 로그 라인들을 로그 템플릿이라 불리는 범주 혹은 카테고리 별로 분류하는 작업이다. 생성되는 로그들은 일반적으로 유한개의 고정된 텍스트를 바탕으로

• First Author: Byungchul Tak, Corresponding Author: Byungchul Tak
*Byungchul Tak (bctak@knu.ac.kr), School of Computer Science and Engineering, Kyungpook National University
• Received: 2018. 09. 28, Revised: 2018. 10. 20, Accepted: 2018. 10. 22.
• This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education(NRF-2017R1D1A3B03035777).

하여 변수 부분을 숫자나 스트링을 삽입하여 만들어 진다. 예를 들어 아래와 같은 간단한 형식의 로그 한 줄을 가정해 보자.

```
2017-12-25 17:50:16,134 INFO StateChange: BlockManager: ask 192.168.121.9 to delete [blk_1073742940_2118]
```

이 로그에서 날짜, 시간, IP주소 그리고 blk로 시작하는 ID는 그때마다 변하는 부분이며 “INFO”, “StateChange:”, “BlockManager:”, “ask”, “to delete”와 같은 부분은 고정된 문자열임을 유추할 수 있다. 이 중 고정된 부분만을 모아 ID를 부여하고 하나의 패턴으로 인식할 수 있으며 이러한 패턴들의 집합이 로그 템플릿(log template)을 이룬다. 예제의 로그 템플릿은 와일드카드 문자를 사용하여 다음과 같이 간단한 정규식(regular expression) 형태로 표현된다.

```
*-*-* *:*:*,* INFO StateChange: BlockManager: ask * to delete [*]
```

로그들을 이와 같은 로그 템플릿의 ID들로 변환하면 반복 패턴의 인식, 패턴의 변화 감지, 시스템 Anomaly 감지, 머신 러닝을 위한 feature vector 생성 등 다양한 기법의 입력으로 활용가능하게 된다.

주어진 어플리케이션에 대한 로그 템플릿을 발견하는 문제는 로그 분석의 출발점이 되며 분석 결과의 질에 큰 영향을 주므로 매우 중요한 작업이다. 하지만 대부분의 연구에서는 소스코드 내의 로그 출력문으로부터 수동 혹은 반자동 방식으로 고정 텍스트 형태를 추출하거나 주어진 로그를 직접 검토하여 템플릿을 추출하는 수작업에 의존하였다. 하지만 어플리케이션의 수도 급격히 늘어나고 있으며 하나의 어플리케이션도 빠르게 버전이 바뀌므로 이러한 수동적인 방식으로는 로그 분석에 필요한 로그 템플릿들을 확보하기는 어렵다.

로그 템플릿을 찾는 문제는 결국 로그에서 정적 부분 (static part)과 동적/변수 부분 (variable part)을 인식하는 문제인데 이를 위해 먼저 하나의 로그 템플릿에서 생성된 로그들만을 모아야 한다. 그런 후 라인들의 수직적 비교를 통해 값이 항상 일정한 위치의 단어를 고정 부분으로 인식한다. 하지만 최초에는 로그 템플릿을 모르는 상태이므로 같은 종류의 로그들을 모으는 작업은 휴리스틱(Heuristic)에 의존한다. 즉, 로그 템플릿 발견 문제의 핵심은 얼마나 정확하게 동일한 로그 템플릿에서 생성된 로그들만을 모으는가이다. 기존의 많은 연구에서는 이를 위해 다양한 기법을 사용하였다. 데이터 마이닝의 frequent set mining 기법 적용 [9, 10], 로그의 길이 별로 일차 분류 후 로그 내에서 다시 특정 위치에 나타나는 값을 기준으로 이차분류 [11, 12], 자주 등장하는 특정 단어들의 쌍의 개수를 최대화하는 최적화 문제로 변형[13]하는 등의 다양한 방법이 시도되었다. 하지만, 정확도가 높지 않거나 수행시간이 지나치게 긴 문제점들이 있다. 또한, 제안된 알고리즘들은 여러 개의 하이퍼파라미터를 설정하도록 요구하는데 사용자가 이러한 다수의 하이퍼파라미터들의 의미를 이해하고 설정해야 하는 단점도 있다.

본 연구에서는 이러한 문제점들을 해결하기 위하여 새로운 방식인 ‘Iterative Filtering’ 기법을 제안한다. 제안하는 기법은 로그내의 각 토큰 별로 값들의 분포를 조사하여 정적 부분이 될 가능성이 가장 높은 위치의 값을 기준으로 로그를 필터링하는 과정을 반복하여 최종적으로는 한 종류의 로그 템플릿에 속하는 로그들만을 효과적으로 모으는 방식이다. 부가적으로 기존의 연구에서는 다루지 않았던 문제인 어플리케이션 고유의 ID 형식을 자동 인식할 수 기능도 가진다. 어플리케이션 고유의 ID 형식을 파악하면 동적 부분을 인식하는 능력이 향상되어 전체 로그 템플릿 발견의 정확도를 높일 수 있다. 이러한 기능들을 구현하여 실험한 결과 기존의 연구들에 비해 정확도는 2배에서 최대 20배 이상 향상되었으며 시간 비용도 크지 않아 실용성이 높은 것으로 나타났다.

이 논문은 다음과 같은 순서를 따른다. 제2장에서는 관련 연구 소개와 로그 템플릿 연구의 설명에 필요한 기본지식 및 용어를 소개하고 로그 템플릿 발견 문제의 어려운 점을 설명한다. 제3장에서는 로그 템플릿 문제를 정량적으로 정의한다. 제4장에서는 제안하는 알고리즘의 원리와 세부사항을 제시하며 제5장에서 기존의 기법과 비교하여 알고리즘의 성능을 평가한다. 마지막으로 제6장에서 결론을 내린다.

II. Preliminaries

이 장에서는 먼저 관련 연구를 살펴 본 후 본 연구에서 사용하는 용어들에 대한 정의를 내리고 로그 템플릿 문제의 이해를 위해 필요한 개념들을 소개한다. 그리고 로그 템플릿 발견 문제의 어려운 점을 기술한다.

1. Related works

로그 템플릿을 구하는 문제는 2000년대 초반부터 꾸준하게 연구되어 왔다. 초창기 사용된 방법은 클러스터링 기법을 사용하여 로그들을 우선 클러스터들로 구분한 후 각 클러스터별로 로그들에서 정적인 부분과 동적인 부분을 비교하여 알아내는 것이었다. 하지만 이러한 방법은 클러스터링의 결과에 따라 로그 템플릿의 질이 크게 좌우되어 클러스터링을 정확하게 하는 것이 중요하였다. SLCT기법은 2단계로 구성되는데 첫 단계에서는 frequent word를 먼저 찾아내고 두 번째 단계에서는 frequent word의 조합으로 이루어진 word group을 사용하여 threshold개 이상으로 등장하는 word group을 클러스터의 성립 조건으로 활용하였다 [9, 10]. IPLoM은 word 분석 대신 3개의 Heuristic을 순차적으로 적용하여 빠르게 로그들을 파티션(partition)하는 방식을 제안하였다. 첫 번째로는 로그들의 길이에 따라 로그들을 분류하였으며 그 다음 파티션 내에서 로그의 같은 위치에 있는 단어들을 살펴보고 distinct한 값이 적은 위치를 기준으로 다시 로그들을 분류하는 등의 방법을 사용하였다 [11, 12]. 하지만 본 연구의 평가에서는 heuristic들이

```
2017-12-23 16:26:28,217 INFO org.apache.hadoop.hdfs.server.datanode.DataNode: Got finalize command for block pool BP-5116165-192.168.91.226
2017-12-23 16:27:27,586 INFO org.apache.hadoop.ipc.Server: Starting Socket Reader #1 for port 50020
2017-12-23 16:28:35,399 DEBUG org.apache.hadoop.mapred.ShuffleHandler: Added token for job_1514014057225_0001
2017-12-23 16:28:39,906 INFO Extract jar:file:/opt/hadoop-2.7.4/hadoop/yarn/hadoop-yarn-common-2.7.4.jar!/webapps/node to /tmp/Jetty_19tj0x/webapp
2017-12-23 16:28:40,724 WARN org.mortbay.log: Started HttpServer2$SelectChannelConnectorWithSafeStartup@0.0.0.0:8042
2017-12-23 16:29:40,724 DEBUG org.apache.hadoop.yarn.webapp.WebApps: Web app node started at 8042
```

Fig. 1. Sample Logs from Hadoop Application

```
* * INFO org.apache.hadoop.hdfs.server.datanode.DataNode: Got * command for block pool *
* * INFO org.apache.hadoop.ipc.Server: Starting Socket Reader * for port *
* * DEBUG org.apache.hadoop.mapred.ShuffleHandler: Added token for *
* * INFO Extract jar:file:* to *
* * WARN org.mortbay.log: Started *@*
* * DEBUG org.apache.hadoop.yarn.webapp.WebApps: Web app node started at *
```

Fig. 2. Log Template set of Hadoop Logs

항상 잘 맞지는 않아 로그 템플릿의 질이 좋지 못한 것을 확인하였다. LogSig는 SLCT와 유사하게 단어 분석을 통하여 로그 템플릿을 발견하는 알고리즘을 제시하였다. 모든 중요단어들의 쌍을 나열한 후 공통적인 단어 쌍들이 각 클러스터에서 가장 많도록 하는 문제를 최적화(optimization) 문제로 formulation하여 클러스터를 구성하였다 [13]. 하지만 연산을 많이 필요로 해서 수행 시간이 너무 길어지는 단점을 가진다. LKE는 로그들을 문자열 유사도(string similarity)를 기준으로 클러스터를 생성하는 방식을 사용한다 [15]. 이 기법도 문자열 유사도의 계산에 많은 시간을 소요하므로 실제 사용하기에는 너무 오래 걸리는 단점을 가진다. LogTree는 로그의 형식을 트리형태로 인식하여야 한다는 점을 주장하여 트리 비교 알고리즘을 적용한 후 클러스터링을 수행한다 [20]. 하지만, 로그를 트리로 구성하기 위하여 사용자가 모든 로그들의 형식을 트리형태로 파싱할 수 있도록 문법을 제공할 것을 요구하므로 현실성이 떨어진다. SHISO는 실시간으로 입력되는 로그 스트림으로부터 로그 템플릿을 발견하는 온라인 방식의 알고리즘이다 [21]. 이와 같은 온라인 계열로는 비교적 최근 발표된 Drain기법이 있다 [14]. Drain은 로그를 토큰화한 후 처음 토큰부터 보면서 suffix tree를 구성해 나가는 방식을 사용한다. One-pass로 로그들을 처리하면서 트리를 생성하므로 수행시간 면에서는 매우 빠른 성능을 보인다. 하지만 어플리케이션에서만 사용하는 특정형태의 ID와 같은 토큰들을 자동으로 동적 부분으로 인식하지 못하여 사용자가 그러한 패턴을 정규식(Regular expression)의 형태로 입력할 것을 가정한다.

지금까지 다양한 로그 템플릿 발견 알고리즘들이 연구되었지만 많은 알고리즘들이 여러 개의 threshold값을 사용자가 튜닝해서 사용하도록 요구하고 있어 현실적으로는 사용에 어려움이 있다. 그리고 본 연구에서의 평가에 의하면 정확도가 대부분 만족스럽지 못한 수준이다. 본 연구에서는 이러한 약점들을 보완하는 알고리즘을 제시한다.

2. Definitions

2.1 Log Line

한 줄의 로그를 의미한다. 일반적으로 한 로그 라인은 date, time, 로그 레벨, 프로세스명, PID, 자유형태의 로그 텍스트 필드들을 가진다. 포함되는 필드의 개수와 종류는 어플리케이션마다 상이하다. Fig. 1은 Hadoop 로그의 예이다. 날짜, 시간, 로그레벨 (INFO, DEBUG, WARN, ERROR, FATAL 등), Java 클래스명, 그리고 비정형 데이터인 로그 텍스트로 구성되어 있다. Fig. 1에는 한 줄에 하나의 로그를 보이지만 하나의 로그가 여러 줄로 나누어진 경우도 존재한다.

2.2 Log Template

로그의 기본이 되는 형식을 의미한다. 로그는 어플리케이션 내부에서 print문으로 출력이 되어 콘솔에 보이거나 로그 파일에 저장이 되는데 이때 출력 되는 모든 로그들은 유한개의 형식으로부터 생성된다. 이는 어플리케이션 내부의 print문이 정적부분(static part)과 동적부분 인 변수들로 구성된 문자열을 출력하기 때문이다. Fig. 1의 두 번째 로그의 경우 포트 번호 50020은 상황에 따라 바뀌는 변수임을 유추할 수 있다. 로그 템플릿은 정적, 동적 부분을 구분하여 Fig. 2와 같이 정규식(Regular expression)으로 표현된다. 동적 부분은 간단하게 wildcard, '*' 기호로 표시한다. 로그 템플릿 파악은 어플리케이션의 소스코드에서 로그 print문을 찾아 인자로 전달되는 스트링을 살펴보면 파악할 수 있다. 하지만 출력 문자열이 복잡한 단계를 거치며 완성되기도 하고 언어에 따라 표현 형식도 다양하므로 자동으로 로그 출력문에서 텍스트를 추출하는 것은 매우 많은 노력을 필요로 한다.

2.3 Log Column

한 줄의 로그를 토큰화하여 n개의 토큰으로 변형하였을 때 0부터 n-1까지의 각 토큰의 위치를 칼럼이라고 지칭한다. 로그의 집합에서 특정 칼럼에 오는 distinct한 값의 개수는 얼마이고 무엇인지를 논의할 때 사용한다.

2.4 Value Cardinality

이 논문에서는 한 로그 칼럼에서 보이는 값들에 대해 각각 몇 개의 인스턴스가 있는지를 의미한다. 예를 들어 Fig. 1에서는 모든 로그에서 세 번째 단어를 살펴보면 INFO, DEBUG,

WARN의 3가지 값들이 존재한다. 이때 INFO는 세 번 나타나므로 Value Cardinality는 3이 된다. DEBUG와 WARN은 각각 Value Cardinality가 2와 1이 된다. 네 번째 토큰은 하나를 제외하고는 모두 자바객체명인데 로그라인들이 각각 다른 값을 가지므로 이 칼럼에서는 모든 값들에 대하여 Value Cardinality가 전부 1이 된다.

2.4 Known Pattern & Custom Pattern

로그에는 다양한 스트링 패턴들이 존재한다. 이 중 한 종류는 Table 1과 같이 일반적으로 널리 쓰이는 Known패턴들이다. 로그 텍스트 내의 Known패턴들은 동적 부분이므로 로그 템플릿 생성 시 '*'로 치환할 수 있다. 하지만 Known패턴들 사이에 포함관계가 있어 단순 greedy 방식의 치환은 오류를 발생한다. 예를 들어, IP주소와 포트는 file path의 일부분으로 사용되기도 하므로 IP주소와 포트를 먼저 '*'로 치환하면 file path 인식이 문제가 된다. 그리고 Known패턴들의 앞과 뒤에 위치하는 문자에도 주의를 기울여야한다. 예를 들어 정수의 경우 정수만 기계적으로 치환할 경우 UUID에서 내부의 숫자만 전부 '*'로 변환되고 UUID 자체는 인식하지 못하게 된다.

Table 1. Known Patterns

Pattern Name	Example
Time	14:23:05
IP address and port number	192.168.15.3:8080
UUID	00abc59f-7bc3-4f98-b699-11bc7aac967f
File path	/tmp/hadoop/dfs/in_use.lock
URL	http://dimos:50070/imagetransfer?getedit=1&startTxId=40998
Integer	123, 38920, -6
Floating Point	89.234, -1.09
Hexadecimal	0xFFFD1FA3, 0xbc34
Email	john@research.com
Date	2018 Jun 03, 2006/03/12

Custom패턴들은 어플리케이션에서 자체적으로 정의하여 사용하는 고유 형태의 ID의 형식을 의미한다. Hadoop의 경우 block ID로 **blk_1073742391_1567,** 컨테이너 ID로 **container_1514014057225_0001_01_000017**와 같이 정의해서 사용하며 이러한 ID들은 10여 가지가 넘는다. Custom패턴을 인식하지 못할 경우 실제로는 동적 부분을 정적 부분으로 잘못 판단하여 개별 ID별로 별개의 로그 템플릿이 생성되는 오류가 발생하기도 한다. 이처럼 Custom패턴의 인식 가능 여부는 로그 템플릿 발견에 큰 영향을 끼치므로 중요한 문제이지만 현재까지는 자동으로 추출하는 기법은 제안되지 않고 있다. Drain [14]에서는 사람이 Custom패턴들을 정규식의 형태로 입력해 주는 것을 가정한다. 본 연구에서는 Custom패턴들을 자동으로 발견하는 기능을 가지고 있어 어플리케이션에 대한 지식에 의존하지 않고도 정확도가 높은 로그 템플릿을 발견할 수 있다.

3. Challenges

로그 템플릿을 구하는 일반적인 방법은 동일한 템플릿에 속할 가능성이 높은 로그들을 모은 후 이 로그들의 칼럼을 차례로 비교하며 정적인 부분과 동적인 부분을 판별하는 것이다. 첫 번째 단계인 로그 파티션(partition) 단계는 로그 템플릿을 이미 알고 있어야 정확하게 로그를 모을 수 있다는 모순이 있다. 기존의 연구에서는 이를 위해 다양한 Heuristic들을 사용하였다. IPLoM [11] 에서는 로그 라인의 토큰 개수를 기준으로 일차적으로 파티션 한 후 distinct한 값의 개수가 가장 적은 칼럼을 기준으로 파티션한다. 마지막으로는 칼럼들 간의 값의 포함관계를 조사하여 추가적으로 파티션을 한다. SLCT [9,10] 는 frequent word들의 그룹을 사용하여 클러스터링 한 후 로그들을 비교한다. LKE [15]는 string edit distance를 기준으로 클러스터링 하여 로그를 모은다. 하지만, 이러한 방법들은 두 가지 문제점을 가진다. 첫째로, 클러스터에 다른 로그 템플릿에 속하는 로그들이 일부분 섞여 있을 경우 로그들을 비교하여 로그 템플릿을 생성하는 방법을 적용하기가 어려워진다. 실제로 정적 단어인 부분이 다른 로그들이 섞여 있음으로 인해서 동적인 부분처럼 보일수도 있어서 이를 판별하는데 어려움이 발생한다. 두 번째로는 사용 빈도가 낮은 비교적 희귀한 로그 템플릿으로부터 생성된 로그는 그 로그의 개수가 적어서 단어의 위치를 비교하여 정적 또는 동적 부분임을 판단하기 어려운 경우가 생긴다. 희귀한 로그 템플릿들의 개수가 상당수 되므로 다수의 로그 템플릿들을 찾지 못하거나 동적 부분을 정적인 부분으로 판별하여 잘못된 로그 템플릿을 생성하기도 한다.

III. Problem Definition

본 연구의 정성적 목표는 주어진 로그들의 기본 바탕이 되는 로그 템플릿들을 가장 정확하게 찾아내는 것이다. 정확함의 정도를 측정하기 위해서는 이상적으로는 대상이 되는 어플리케이션의 실제 정답이 되는 로그 템플릿들이 필요하지만 이는 아직 구하기 이전이므로 존재하지 않는다. 따라서 올바른 로그 템플릿들이 가져야 하는 성질들을 정의하여 정량적으로 이를 계산하여 정답에 얼마나 근접하는지를 유추하는 방식을 사용한다. 이를 위해 다음의 두 가지 측정치를 정의한다.

첫째, 로그 템플릿은 인식된 정적 부분을 많이 포함할수록 정확도가 오르는 것으로 볼 수 있다. 이를 위해서 식(1)과 같이 정의되는 정적 부분의 길이 전체 합을 사용한다.

$$S = \sum_{i=1}^{m} static(t_i) \qquad (1)$$

여기서 m은 전체 로그 템플릿의 개수이며 static()은 정적부분의 캐릭터 개수를 반환한다. 이 식에 의하면 정적 부분을 조

금이라도 더 인식해 낸 로그 템플릿이 더 높은 값을 가지게 된다. 이 성질은 로그 템플릿에서 와일드카드(*)의 개수가 최소화되도록 유도하기 위함이다. 정적 부분이 적고 '*'가 많은 로그 템플릿은 많은 수의 로그들이 매칭(matching) 되지만 실제로 서로 다른 로그 템플릿에 속하는 로그들까지도 하나의 로그 템플릿으로 묶일 수 있는 위험이 있다. 반대로, 정적 부분이 지나치게 많을 경우는 동적 변수 값의 하나를 정적 토큰으로 인식하여 매칭 되는 로그의 수가 적어지게 되며 실제로는 하나의 로그 템플릿이 되어야 하지만 값의 개수만큼 여러 개로 분리될 수도 있다. 따라서 중간의 균형점이 존재하며 이를 찾기 위해서 지나치게 많은 토큰을 정적 부분으로 인식한 경우는 패널티를 주는 식이 필요하다. 그런 경우는 해당 로그 템플릿에 매칭되는 로그의 개수가 적게 되는 점에 착안하여 로그 템플릿들에 매칭되는 로그 개수의 평균값 U를 도입한다.

$$U = \frac{1}{m}\sum_{i=1}^{m}\left|match(t_i)\right| \qquad (2)$$

식(2)에서 match()함수는 매칭되는 로그들을 반환하는 함수이며 절대값 기호를 사용하여 로그들의 개수를 의미한다. 로그 템플릿에서 동적 부분을 정적인 부분으로 잘못 인식하여 로그 템플릿의 개수가 증가하면 U의 값은 낮아지므로 정적 부분 인식 오류에 대한 패널티로 작용한다. 최종적으로는 S와 U를 사용하여 식(3)의 값이 최대화 되는 로그 템플릿의 집합이 더 실제 로그 템플릿에 근접한 것으로 정의한다.

$$Score = S \cdot U \qquad (3)$$
$$= \sum_{i=1}^{m} static(t_i) \cdot \frac{1}{m}\sum_{i=1}^{m}\left|match(t_i)\right|$$

로그 템플릿 발견 문제는 이 Score 값이 높아지는 알고리즘을 찾는 문제로 정의한다. 제안하는 알고리즘은 Score값이 높아지는 방향으로 정적 부분과 동적 부분을 인식하는 결정을 내리는 방식으로 디자인된다.

IV. Log Template Discovery Algorithm

1. Basic Principle

로그의 템플릿을 생성하는 작업은 우선 동일한 로그 템플릿에서 생성된 로그들을 모은 후 로그들의 동일한 칼럼 위치에 있는 단어(혹은 토큰)들을 수직적으로 살펴보며 그 위치에 하나의 값만 항상 발생하는지 아닌지에 따라 정적인지 동적인지를 판별 할 수 있게 된다. 하지만, 이 작업을 위해서는 동일한 로그 템플릿에서 생성된 로그들만을 먼저 모아야 한다. 다른 템플릿에 속하는 로그가 하나라도 섞일 경우 정적 토큰이라도 그 칼럼의 Value Cardinality가 1 이상이 되어 정적 토큰임을 판단하기 어려워진다.

```
INFO FSImageFormat: Image file /tmp/fsimage_0014 of 1043 bytes saved in 0 seconds.
INFO FSImageFormat: Image file /tmp/fsimage_0093 of 1023 bytes saved in 1 seconds.
INFO Scheduling blk_2944 file BP-1512025890710 for deletion
INFO FSImageFormat: Image file /tmp/fsimage_0015 of 2023 bytes saved in 2 seconds.
DEBUG Configuration found resource core-site.xml at file:/usr/core-site.xml
INFO FSImageFormat: Image file /tmp/fsimage_0159 of 1157 bytes saved in 3 seconds.
INFO Scheduling blk_2950 file BP-1512025890710 for deletion
INFO FSImageFormat: Image file /tmp/fsimage_3148 of 1784 bytes saved in 0 seconds.
INFO FSImageFormat: Image file /tmp/fsimage_3199 of 1661 bytes saved in 1 seconds.
INFO Scheduling blk_2945 file BP-1512025890710 for deletion
INFO FSImageFormat: Image file /tmp/fsimage_2893 of 1663 bytes saved in 2 seconds.
DEBUG Configuration found resource yarn-site.xml at file:/usr/yarn-site.xml
```

Fig. 3. Log Sample

이 문제를 해결하기 위해서 본 연구에서는 로그의 각 토큰 위치에서의 Value Cardinality를 사용한 Iterative Filtering 알고리즘을 제안한다. 아이디어의 핵심은 로그의 각 토큰 칼럼 위치에서의 distinct 값의 개수를 파악하여 가장 큰 Value Cardinality의 (즉, 가장 큰 비중을 가지는) 값을 가지는 로그들만을 선택해서 로그의 집합의 크기를 줄여 나가는 것을 반복하는 것이다. 예시를 통하여 원리를 설명하기 위하여 Fig. 3의 열두 줄의 로그를 사용한다.

Fig. 3의 로그들은 Fig. 3에서 보이는 세 개의 로그 템플릿으로부터 생성된 로그들이다. 알고리즘의 최종 목표는 Fig. 3와 동일한 로그 템플릿들을 찾아내는 것이다. 이 예는 간단하므로 육안으로 로그들이 어떤 로그 템플릿으로부터 생성되었는지 쉽게 확인할 수 있다. Fig. 3의 첫 번째 로그 템플릿으로부터는 일곱 개의 로그가 생성되었으며, 두 번째로부터는 세 개의 로그, 세 번째로부터는 두 개의 로그가 생성되었다.

```
(1) INFO FSImageFormat: Image file * of * bytes saved in * seconds.
(2) INFO Scheduling * file * for deletion
(3) DEBUG Configuration found resource * at file:*
```

Fig. 4. Log Templates

첫 단계에서는 각 로그 라인들을 구분문자(delimiter)를 사용하여 단어로 분리하는 토큰화 (tokenization) 작업을 수행한다. 예시에서는 설명의 간소화를 위하여 스페이스 문자 하나만을 구분문자로 사용한다. 실제 로그에서는 다른 다양한 구분문자들이 사용되므로 이들을 함께 적용하여야 한다.

Fig. 5는 토큰화하여 세로로 칼럼위치가 정렬된 상태의 로그이다. 제안하는 알고리즘은 처음 각 칼럼별로 Value의 분포를 구한다. 예시에서는 총 12개의 칼럼이 존재한다. 첫 번째 칼럼의 경우 값의 종류는 'INFO', 'DEBUG'와 같이 두 개이므로 Value Cardinality는 2가 된다. 칼럼 3과 7의 테이블에서 보이는 바와 같이 Value Cardinality가 각각 5와 10이 된다. 칼럼 12의 경우는 한 종류의 값만 존재한다. (로그의 길이가 12개의 토큰 미만인 것들이 있으므로 null값이 추가로 있는 것으로 간주하여도 무방하다.) Fig. 3은 4개의 칼럼에 대한 테이블 만을 선택적으로 보여주지만 실제로는 모든 12개의 칼럼에 대해 이러한 테이블을 작성한다. 이 테이블 중에서 가장 높은 Value

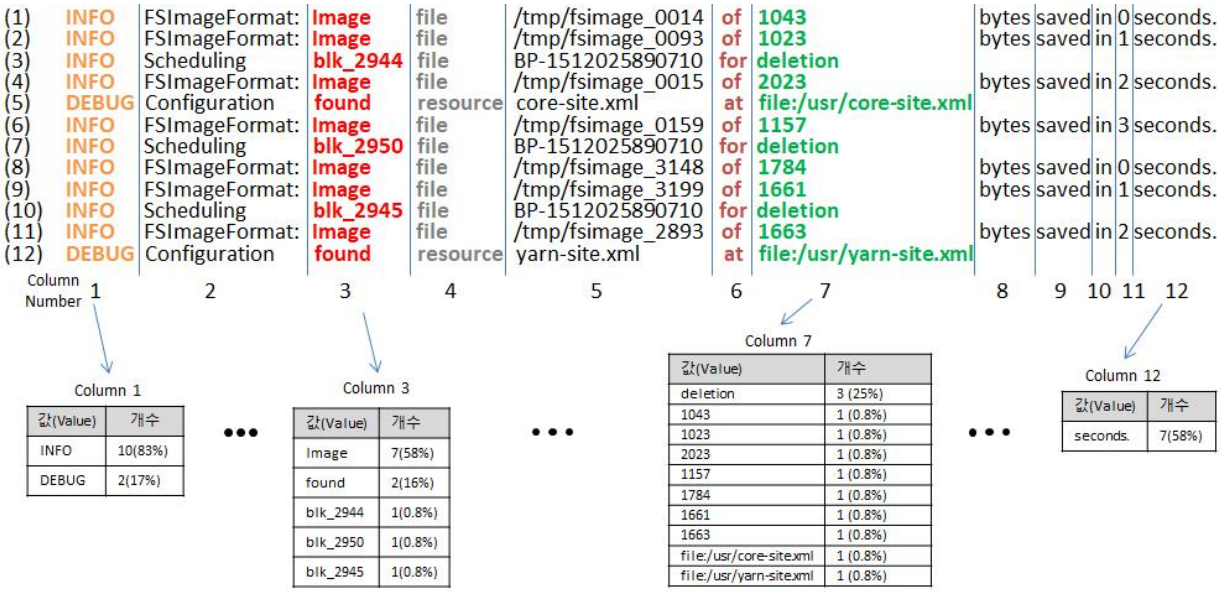


Fig. 5. The First Step of Iterative Log Filtering Using Column Value Cardinality

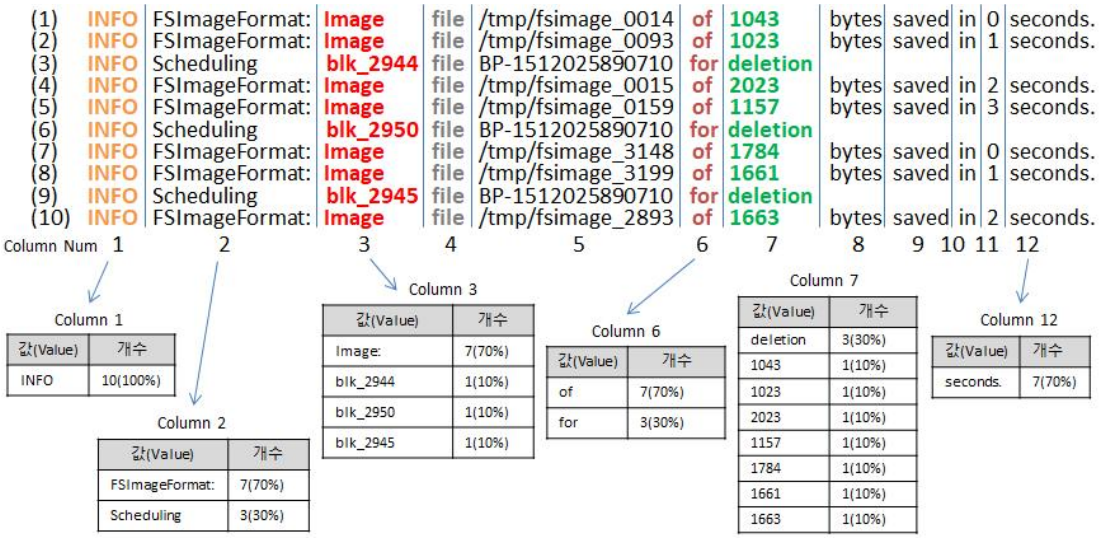


Fig. 6. The Second Step of Iterative Log Filtering Using Column Value Cardinality

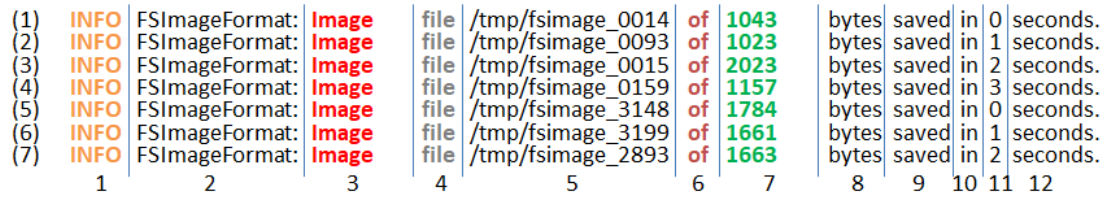


Fig. 7. The Third Step of Iterative Log Filtering Using Column Value Cardinality

Cardinality를 가지는 값이 있는 칼럼에서 그 값을 선택한다. Fig. 5에서는 칼럼1의 'INFO'가 cardinality 10, 즉 83%이므로 이 값을 선택한다. 다음, 전체 로그 12개에서 첫 번째 칼럼이 'INFO'값을 가지는 로그만 남기고 나머지는 제거(필터링)한다.

Fig. 6은 전 단계에서 남겨진 로그들에 대하여 동일하게 토큰화 작업을 수행하고 각 칼럼에서 Value 분포를 구한 상태를 나타낸다. 칼럼1은 전 단계에서 'INFO'인 부분만을 골라내었으므로 여기서는 100% 'INFO'로 된다. 전 단계에서 로그를 필터링하여 100% 하나의 값을 가지는 칼럼들이 여럿 존재함을 볼 수 있다. 따라서 100% 한 값을 가지는 칼럼들은 이 단계에서는 무시된다. Fig. 6의 테이블들에 의하면 칼럼 2,3,6은 모두 제일 큰 값이 cardinality 7, 즉 70%의 값을 가진다. 이들 중 임의의 칼럼을 선택하여 필터링을 수행한다.

필터링의 결과로 Fig. 7에 보이는 바와 같이 7개의 로그만

남게 된다. 실제로 이 로그들은 Fig. 4의 로그 템플릿 (1)에 속하는 로그임을 눈으로 확인할 수 있다. 또한 칼럼 5, 7, 11을 제외한 모든 칼럼들이 100%의 단일 값을 가지는 것도 확인 할 수 있다. 칼럼 5, 7, 11은 Value Cardinality가 모두 1, 즉 모두 unique한 값들만을 존재하므로, 로그 템플릿에서 동적 부분이라 판단할 수 있다. 이 칼럼들을 '*'로 치환 후 모든 토큰들을 concatenate하면 다음과 같이 된다.

INFO FSImageFormat: Image file * of * bytes saved in * seconds

즉, Fig. 4의 로그 템플릿 (1)과 동일한 스트링을 얻게 되므로 하나의 로그 템플릿을 구하는 작업을 완료하게 된다.

이와 같이 하나의 정규식 형태의 로그 템플릿을 구하면 이에 match되는 로그들을 찾아 전부 제거한다. 그리고 남은 로그들을 가지고 다시 Iterative Filtering 방법을 Fig. 5, 6, 7과 같이 적용하여 또 다른 로그 템플릿을 찾아낸다. 알고리즘의 종료 조건은 모든 로그들이 소진되는 시점이다.

2. Formal Algorithm Description

2.1 Iterative Filtering Parameter Determination

이 단락에서는 각 iteration의 단계에서 결정하여야 할 칼럼 번호 α와 그 칼럼 위치에서 가장 높은 빈도를 보이는 값 β를 찾는 방법을 정의한다. 주어진 전체 로그라인의 수는 n이며 이들은 총 m개의 로그 템플릿에서 생성되었다고 가정한다. 하나의 로그를 l이라고 할 경우 i번째 로그 li는 ki개의 토큰으로 구성된다. 연산자'·'는 concatenation을 의미한다.

$$l_i = w_{i,1} \cdot w_{i,2} \cdot \ldots \cdot w_{i,k_i} \qquad (4)$$

로그 필터링을 위해서는 토큰의 칼럼번호와 값을 결정하여야 한다. 이를 위해 N(j,x)함수를 정의한다. N(j,x)함수는 인자로 칼럼번호 j와 문자열 x를 필요로 하며 $\{x \mid x \in \bigcup_{i=1}^{n} w_{i,j}\}$와 같다. 집합 $Q = \{s \mid 1 \le s \le \max_{1 \le i \le n}(k_i)\}$(즉, 전체 로그라인들 중 가장 큰 토큰의 수 보다 작거나 같은 수)라 할 때, 칼럼번호 j는 $j \in Q$이다. 이 함수 N(j,x)은 임의의 칼럼 j에서의 주어진 x의 인스턴스의 개수 (즉, x값이 몇 번 나타나는가)를 구한다. 이를 사용하여, 칼럼 j를 고정하였을 경우 그 칼럼에서 가장 Value Cardinality가 높은 토큰 A(j)는 식(5)와 같이 표현된다.

$$A(j) = \arg max_x N(j,x) \qquad (5)$$

그리고 그 Value Cardinality B(j)는 식(6)과 같이 표현된다.

$$B(j) = \max N(j,x) \qquad (6)$$

따라서 로그 필터링을 위해 결정하여야 할 토큰의 칼럼 번호 β는 식(7)과 같다.

$$\begin{aligned} \beta &= \arg max_{j \in S} B(j) \\ &= \arg max_{j \in S}(\max N(j,x)) \end{aligned} \qquad (7)$$

그리고 그 칼럼에서의 토큰 값을 α라고 할 경우 그 값은 식(8)을 사용하여 구할 수 있다.

$$\begin{aligned} \alpha &= A(\beta) \\ &= \arg max_x N(\beta,x) \\ &= \arg max_x N(\arg max_{j \in S}(\max N(j,x)),x) \end{aligned} \qquad (8)$$

Input: log set L
Output: list of log m templates t_1, ... t_m

```
1: L = Load_log_data(logfile_name)
2: while size(L)>0 do
3:     R=Random_select(L)
4:     R_T=Tokenize(R)
5:     filter_vect=[0,0, ... , 0] of size max_token_count(R_T)
6:     filter_word=[null,...,null] of size max_token_count(R_T)
7:     while filter_vect has 0 do
8:         S_T=do_log_filtering(R_T,filter_vect,filter_word)
9:         α,β=determine_filter_param(S_T)
10:        if custom_pattern_detected then
11:            substitute_pattern(L)
12:            goto line 2
13:        end if
14:        filter_vect[β] = 1
15:        if values(β) is uniformly distributed then
16:            filter_word[β] = '*'
17:        else
18:            filter_word[β] = α
19:        end if
20:    end while
21:    /* Generate log template in regular expression */
22:    for token in filter_word do
23:        token=escape_regex_characters(token)
24:        log_template=log_template+token
25:    end for
26:    L=match_and_remove(L,log_template)
27:    log_template_list.append(log_template)
28:end while
```

Algorithm 1. Pseudocode of Iterative Filtering Algorithm

이와 같이 구한 α, β를 사용하여 n개의 로그라인들 중에서 β 칼럼에 α 값을 가진 것들만 선택하여 축소된 n'개의 로그라인들을 구성한다. α값은 로그 템플릿에서 정적 토큰으로 인식되어 기록된다. 이 스텝은 정적 부분의 길이를 늘이게 되므로 제3장의 Score식에서 S의 값을 증가시켜 Score의 값도 높이는 역할을 한다.

2.2 Algorithm Pseudocode

Algorithm 1은 제안하는 기법의 로직을 Pseudo-code로 나타낸다. 전체적인 구조는 두 개의 while 루프로 구성된다. 외곽의 while 루프는 한 번에 하나의 로그 템플릿을 구하는 작업을

하며 내부의 while 루프는 한 로그 템플릿을 구하기 위해 필요한 여러 번의 필터링 작업에 해당한다.

먼저 로그 데이터를 파일로부터 읽어서 메모리에 올린다. Line 3에서는 무작위로 로그의 일부분만을 선택하여 R을 만든다. 이렇게 하는 이유는 데이터의 크기를 줄여도 결과의 무결성에는 영향이 없으며 성능향상도 얻을 수 있기 때문이다. 랜덤 샘플링 크기는 설정 가능하며 보통은 200~300 로그라인 정도로 한다. 이후, 샘플링 한 로그를 토큰화하여 RT를 만든다. 필터링에 필요한 데이터 구조는 이 토큰화한 로그에서 가장 많은 토큰을 가지는 로그의 토큰 수를 기준으로 생성한다 (Line5-6).

내부 while 루프는 먼저 현재의 필터링 데이터 구조를 사용하여 RT에서 로그들을 추려내어 ST를 만든다. 이 로그들로부터 2.1에서 설명하는 정의에 따라 α와 β값을 구한다(Line 9). 이 과정에서 로그에 존재하는 Custom패턴을 발견하는 경우가 발생하는데 이때는 전체 로그에서 이 패턴에 match되는 문자열을 L에서 모두 찾아서 특별기호로 치환한 후 현재의 외곽 while 루프를 재시작 한다. Custom패턴 발견 기법은 제4장 3절에서 설명한다.

Line 14에서는 α와 β를 이용하여 필터링 데이터 구조를 업데이트 한다. 즉, β칼럼은 유효한 칼럼임을 표시하기 위하여 filter_vect[β]를 1로 세팅하며 그 위치에서의 필터링 값은 Line 15에서 wildcard('*') 혹은 α로 설정한다. 칼럼 β의 값들이 많은 **unique**값들만 있으면 이 칼럼은 동적 부분이라고 간주하여 '*'를 사용하며 아닐 경우는 α를 그대로 입력한다. 현재는 칼럼 내에 모든 값들이 전부 **unique**하면 '*'로 변환하도록 하지만 값들이 완전하게 **uniform**하지 않더라도 동적 부분일 수도 있다. 따라서 **unique**한 값의 개수가 특정 **threshold** 이상일 경우 '*'로 변환하도록 하는 방법도 사용할 수 있다.

Line 22에서는 **filter_word**를 하나의 스트링으로 만들어 로그 템플릿을 정규식의 형태로 생성해 낸다. 이 때, 스트링 내에서 정규식의 특수 기호로 해석될 수 있는 문자들은 **escape**한다. 이 정규식 형태의 로그 템플릿을 사용하여 **L**에서 **match**되는 모든 로그를 제거한다. 새로 구한 로그 템플릿은 리스트에 저장한다. 외곽 **while** 루프가 종료되면 **log_template_list**에는 **m**개의 로그 템플릿이 저장되어 있다.

3. Custom Pattern Detection

Iterative 필터링 과정의 마지막 단계에 근접할수록 각 칼럼들의 **Value Cardinality**는 양극화된다. 즉, 정적 토큰이 위치한 칼럼의 경우는 **distinct**값이 1이 되어 **Value Cardinality**는 **100%**가 되지만 동적 토큰이 위치한 칼럼의 경우는 **Value Cardinality**가 1이 되며 값들의 분포도 **Uniform**분포에 근접한다. 제안 알고리즘에서는 모든 값들의 **Value Cardinality**가 **1**이 되는 경우 특정 패턴을 따르는 변수가 그 칼럼에 있을 가능성이 높다고 판단하여 패턴의 존재유무를 테스트한다. 현재의 테스트는 우선 **character**의 개수가 모든 값에서 동일한지를 검사한다. 만약 그러하면 **character**의 위치별로 항상 알파벳이거나 숫자거나 그 이외의 특수 문자인지를 검사한다. 모든 값들이 동일하게 각 위치의 **character**가 알파벳, 숫자, 특수문자이면 패턴이 있는 것으로 간주하여 정규식을 도출한다. 이 정규식은 **Known**패턴으로 편입되며 이후의 로그 템플릿 생성 과정에서 이 패턴을 만족하는 칼럼은 동적 (**variable**) 토큰으로 인식하여 '*'로 치환된다. 또한, 이미 발견된 로그 템플릿에서도 이 패턴을 따르는 토큰은 '*'로 치환된다.

V. Evaluation

1. Experimental Data

제안하는 로그 템플릿 발견 알고리즘의 성능 평가를 위하여 Table 2와 같이 현재 널리 쓰이고 있는 세 종류의 어플리케이션들의 로그를 사용하였다.

Table 2. Log Size of Application Logs

Application	Version	Log Line Count
Hadoop	2.7.4	75703
OpenStack	Mikata	110249
Cassandra	3.11.0	51318

Hadoop [16]은 map-reduce 패러다임을 구현한 빅 데이터 처리를 위한 프레임 워크이다. 데이터 파일을 n개의 데이터 셋으로 분할하고 HDFS를 사용하여 분해 후 계산결과를 각 노드에서 다시 수집하는 방식으로 동작한다. Hadoop의 job 스케줄링과 리소스 관리는 Yarn [22]을 통해 수행된다. 따라서 전제 Hadoop의 로그는 HDFS, Yarn, Mapreduce, 그리고 그 외의 모듈들로 구성되는데 이중 Yarn 로그가 88%를 이룬다.

OpenStack [17]은 IaaS (Infrastructure-as-a-Service) 방식의 클라우드 서비스를 위한 VM관리, 생성, UI 등을 제공하는 플랫폼 소프트웨어이며 실험을 위해서 DevStack을 사용하여 설치하였다. OpenStack로그는 총 24개의 컴포넌트에서 생성된 로그들을 포함한다. 이 중 네트워크 가상화를 담당하는 open_vswitch 컴포넌트와 네트워크 관리를 위한 Neutron 컴포넌트의 로그들이 63%를 차지한다.

Cassandra [18]는 와이드 칼럼을 지원하는 특징을 가지는 NoSQL 데이터베이스이며 decentralized 구조를 사용하여 높은 수준의 scalability를 가지고 있는 소프트웨어이다.

성능 비교의 대상은 기존의 기법들인 SLCT, IPLoM, LogSig, Drain을 사용하였다. IPLoM, LogSig, Drain은 [19]에서 사용한 소스코드 (https://github.com/logpai/logparser)를 사용하였다. IPLoM과 LogSig는 원저자들의 코드가 공개되지 않아서 [19]의 저자들이 성능평가를 위하여 직접 구현하였다. LKE 소스 코드도 여기에 포함되어 있지만 로그데이터의 처

리에 3일 이상 걸려 평가에서 제외하였다. SLCT는 C로 구현된 원저자의 코드를 사용하였다.

2. Accuracy

정확도의 측정을 위해 Hadoop, OpenStack, Cassandra 로그들의 실제 로그 템플릿들을 수작업으로 먼저 구하였으며 각각 367개, 52개, 132개의 로그 템플릿들로 구성된 것으로 파악되었다. 실제 정답이 되는 로그 템플릿을 GT (Ground Truth)로 표시한다. 정확도의 일반적인 척도는 precision과 recall을 사용하는 F-measure이지만 이 실험에서는 GT의 로그 템플릿들이 각각 얼마나 다른 알고리즘들의 결과에 존재하는지를 측정하는 대신 그 로그 템플릿들이 매칭하는 개수들의 수열을 비교하였다. 로그 템플릿 자체를 스트링 비교하지 않는 이유는 동일한 로그 템플릿이라도 정규식 표현의 차이가 있을 수 있으며 실제로는 동적 토큰이지만 그 값이 하나만 로그에 보여서 정적 토큰처럼 처리되기도 하여 로그 템플릿들은 정확하게 일치하지 않을 수 있기 때문이다.

Table 3. Log Template Count Comparison

Technique	Hadoop	OpenStack	Cassandra
Ground Truth	368	53	133
Ours	344	53	178
Drain	352	44	246
IPLoM	222	147	275
SLCT	1658	323	2022
LogSig	368	53	133

Table 3은 알고리즘 별로 발견한 로그 템플릿의 개수를 비교하여 보여준다. GT와 비교하였을 때 제안하는 알고리즘(Ours)이 가장 근접하게 실제 정답과 비슷한 개수의 로그 템플릿을 발견한 것을 볼 수 있다. LogSig의 경우는 파라미터로 찾고자 하는 로그 템플릿의 개수를 지정하여야 하므로 실험에서는 GT과 동일한 개수를 지정하였다. 하지만 실제로는 주어진 로그의 템플릿 개수를 알지 못하므로 LogSig 알고리즘이 로그 템플릿의 개수를 정확하게 찾은 것은 아니다.

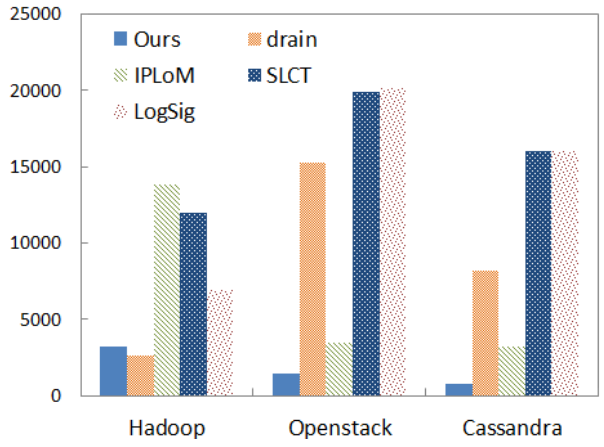


Fig. 8. Log Template Accuracy Comparison

Fig. 8은 제안하는 알고리즘(Ours로 표시)과 다른 기존의 알고리즘과의 정확도 차이를 보여준다. 이 그래프는 각 알고리즘별로 발견한 로그 템플릿에 매칭되는 로그들의 개수를 정렬한 벡터를 생성한 후 order 2를 가지는 Minkowski distance를 계산하여 비교하였다. Minkowski distance D는 두 벡터 X와 Y에 대하여 식(9)로 정의되며 order p가 2인 경우는 Euclidean 거리를 의미한다.

$$D(X,Y)=\left(\sum_{i=1}^{n}|x_i-y_i|^p\right)^{1/p} \qquad (9)$$

D(X,Y) 값이 적을수록 GT와 비슷함을 의미한다. Ours는 모든 세 종류의 로그에서 GT에 매우 근접한 값을 가지는 것으로 나타났다. Drain의 경우는 Hadoop로그에서는 Ours를 조금 앞서지만 다른 로그들에서는 좋지 못한 성능을 보여준다. 이 결과에서 Ours의 성능이 다른 알고리즘에 비해 대부분의 경우 정확도 면에서 크게 앞서는 것을 확인할 수 있다.

Table 4. Algorithm Running Time Comparison

Technique	Hadoop	Open-stack	Cassandra	Speed (line/sec)
Ours	1m25s	20s	2m17s	980
Drain	9.8s	5.6s	3.6s	12488
IPLoM	1m22s	2m2s	54s	920
SLCT	3.5s	6.2s	3.6s	17839
LogSig	148m17s	121m43s	380m48s	6.1

3. Time Cost

Table 4는 로그 템플릿 발견 알고리즘들의 수행시간을 비교하여 보여준다. Drain과 SLCT는 수행시간 측면에서 초단위의 매우 짧은 수행시간을 보인다. 그 다음으로 Ours와 IPLoM은 분단위의 수행시간을 보인다. LogSig와 LKE는 수행속도가 지나치게 느려 실용성에 문제가 있다고 판단된다. 처리속도 측면에서는 제안하는 방법은 약 980 line/sec의 속도록 로그를 처리한 것으로 나타났다. 이 속도는 Hadoop, OpenStack, Cassandra의 전체 로그수를 전체 수행시간으로 나눈 값이다. Drain과 SLCT는 비슷하게 매우 빠른 속도를 보여주며 제안기법은 두 번째 그룹에 속하는 것을 볼 수 있다.

수행시간 면에서는 Ours가 가장 빠르지는 않지만 위에서 보인 정확도를 고려하면 초단위의 성능을 가지는 Drain과 IPLoM과 비교해 뒤떨어지는 것은 아니라 판단된다. 정확도를 높이기 위해 들어가는 추가적은 연산으로 인한 소요시간이므로 분단위의 수행시간 소요는 적절한 것으로 판단된다.

IV. Conclusions

본 연구에서는 로그 템플릿을 효과적으로 발견하기 위한

Iterative Filtering 방식의 새로운 알고리즘을 제안하였다. 이 기법은 기존의 Heuristic을 사용한 텍스트 클러스터링 방식이나 Suffix트리 구성 방식과는 다르게 로그의 각 토큰 칼럼들의 값의 분포를 살펴 가장 비율이 높은 값들을 골라 반복적으로 필터링을 함으로써 동일한 로그 템플릿에 속할 가능성이 가장 높은 로그들을 분리해 내는 방법이다. 이렇게 분리된 로그 그룹들을 비교하여 동적인 부분과 정적인 부분을 판별해 로그 템플릿을 생성한다. 또한 이 스텝들을 수행하는 과정에서 어플리케이션 고유의 ID포맷을 추가로 발견하여 추후 로그에서 이 부분은 바로 동적 부분임을 인식하는 것이 가능하게 되었다. 고유 ID포맷을 올바르게 인식하지 못할 경우 개별 ID값들을 정적으로 인식하여 로그 템플릿의 개수가 과도하게 많아지는 문제가 있는데 본 연구에서는 이 문제를 해결하였다.

제안하는 로그 템플릿 발견 기법을 구현하여 발견된 로그 템플릿의 정확도와 수행시간을 기존의 네 가지 기법과 비교하였다. 실험 결과 제안된 기법은 정확도에서 2배에서 최대 20배 이상을 나타낼 수 있도록 구현되었고 시간 성능 면에서는 주어진 로그 데이터양을 처리하는데 다른 기법들의 수행속도들의 중간 값을 보이는 것으로 나타났다. 그리고 기존의 기법들은 로그를 수집한 어플리케이션에 따라 정확도의 편차가 큰 것으로 나타나서 실제 다양한 로그에 적용하였을 때 그 결과를 신뢰하기 어려운 문제점이 있지만 제안하는 기법은 안정적으로 편차가 낮게 가장 우수한 정확도를 보여주었다. 시간 성능은 제일 우수하지는 않지만 정확도가 낮은 경우 시간 성능은 의미가 적으므로 제안하는 기법의 단점이 되지는 않는다고 판단한다.

추후 연구 방향으로는 Custom패턴의 정확도를 높이기 위한 정교한 알고리즘의 고안, Iterative Filtering의 속도를 향상하기 위한 로그 전처리 기법의 도입 등이 있다. 장기적으로는 제안하는 알고리즘을 사용하여 대량의 로그에서 반복되는 패턴의 유무를 자동으로 파악하는 확장성(scalability)이 높은 알고리즘을 개발하는 것이다.

REFERENCES

[1] K. Kc and X. Gu. Elt, “Efficient log-based troubleshooting system for computing infrastructures,” In IEEE International Symposium on Reliable Distributed Systems (SRDS), Oct. 2011.

[2] B. C. Tak, S. Tao, L. Yang, C. Zhu, and Y. Ruan, “LOGAN: Problem Diagnosis in the Cloud Using Log-based Reference Models,” In IEEE International Conference on Cloud Engineering (IC2E), pp. 62-67, Apr. 2016

[3] X. Fu, R. Ren, S. A. McKee, J. Zhan, and N. Sun, “Digging deeper into cluster system logs for failure prediction and root cause diagnosis,” In 2014 IEEE International Conference on Cluster Computing (CLUSTER), pages 103-112. IEEE, Sep. 2014.

[4] Min Du, Feifei Li, Guineng Zheng, Vivek Srikumar, “DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning,” In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM, Oct. 2017.

[5] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, “Detecting large-scale system problems by mining console logs,” In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP ’09, pages 117-132, New York, NY, USA, Oct. 2009. ACM.

[6] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li, “Mining invariants from console logs for system problem detection,” In Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIX ATC’10, pages 24-24, Berkeley, CA, USA, Jun. 2010. USENIX Association.

[7] C. Lim, N. Singh, and S. Yajnik, “A log mining approach to failure analysis of enterprise telephony systems,” In IEEE International Conference on Dependable Systems and Networks, Jun. 2008.

[8] K. Nagaraj, C. Killian, and J. Neville, “Structured comparative analysis of systems logs to diagnose performance problems,” In Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI’12, pages 26-26, Berkeley, CA, USA, Apr. 2012.

[9] Risto Vaarandi, “A Data Clustering Algorithm for Mining Patterns from Event Logs,” In Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM). Piscataway, NJ, USA, 119-126, Oct. 2003.

[10] R Vaarandi, “Mining event logs with SLCT and LogHound”. In 2008 IEEE Network Operations and Management Symposium (NOMS). Apr. 2008.

[11] A. Makanju, A. Zincir-Heywood, and E. Milios, “A lightweight algorithm for message type extraction in system application logs,” IEEE Transactions on Knowledge and Data Engineering, Nov. 2012.

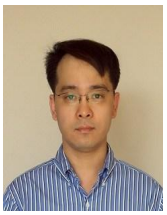
[12] A. A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, “Clustering event logs using iterative partitioning,” in KDD ’09: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining. New York, NY, USA: ACM, Jun. 2009, pp. 1255-1264.

[13] L. Tang, T. Li, and C. shing Perng, “Logsig: Generating system events from raw textual logs,” in Proceedings of ACM Conference on Information and Knowledge Management (CIKM), Oct. 2011.

[14] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R. Lyu,

"Drain: An Online Log Parsing Approach with Fixed Depth Tree," In Proceedings of the IEEE International Conference on Web Services (ICWS 2017). Piscataway, NJ, USA, 33-40, Jun. 2017.
[15] Q. Fu, J. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in ICDM'09: Proc. of International Conference on Data Mining, Dec. 2009.
[16] Apache Hadoop, https://hadoop.apache.org/
[17] OpenStack, https://www.openstack.org/
[18] Apache Cassandra, http://cassandra.apache.org/
[19] P. He, J. Zhu, S. He, J. Li and M. R. Lyu, "An Evaluation Study on Log Parsing and Its Use in Log Mining", in Proceedings of DSN'16: Proc. of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Jun. 2016.
[20] L. Tang and T. Li, "Logtree: A framework for generating system events from raw textual logs," In Proceedings of the 2010 IEEE International Conference on Data Mining, ICDM '10. Washington, DC, USA: IEEE Computer Society, Dec. 2010.
[21] M. Mizutani, "Incremental mining of system log format," in SCC'13: Proc. of the 10th International Conference on Services Computing, Jun. 2013.
[22] Apache Hadoop YARN, https://hortonworks.com/apache/yarn/
[23] Fei Wu, Pranay Anchuri, Zhenhui Li, "Structural Event Detection from Log Messages," Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, Aug. 2017.
[24] Animesh Nandi, Atri Mandal, Shubham Atreja, Gargi B. Dasgupta, Subhrajit Bhattacharya, "Anomaly Detection Using Program Control Flow Graph Mining from Execution Logs," Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, Aug. 2016.
[25] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, Tao Xie, "Log 2 : A Cost-Aware Logging Mechanism for Performance Diagnosis," In Proceedings of the 2015 USENIX Conference on USENIX Annual Technical Conference, USENIX ATC'15, Jun. 2010. USENIX Association.

Authors



Byungchul Tak received his B.S. degree from Yonsei University in 2000, M.S. from KAIST in 2003 and Ph.D. degrees in Computer Science and Engineering from the Pennsylvania State University at University Park in 2012. Dr. Tak joined the faculty of the Department of Computer Science at Kyungpook National University, Dague, Korea, in 2017. He is currently an Assistant Professor in the Department of Computer Science. He is interested in cloud computing, distributed systems, operating system and big data analytics.