An Algorithm for Message Type Discovery in Unstructured Log Data

Daniel Tovarňák

Masaryk University
CSIRT-MU

ICSOFT 2019, July 27

Motivation

Log Analysis via Complex Event Processing (CEP)

Data stream processing: real-time data processing paradigm

- commonly used to deal with high-velocity data

CEP: detection of complex patterns in streams of data elements

- visions for use in real-time log analysis, especially security monitoring
- as opposed to full-text indexing and column-based indexing of log data

Event objects: actual representation of the elements in the stream

- expected to be properly structured and described via an explicit data schema
- much like in RDBMS

Unstructured log entries ≠ event objects

- semi-structured log entries ≠ event objects

Logging and Log Data – 5Vs of Big Data

Traditional manifestation – log files with arbitrary text messages

Value: widely-used source of monitoring information

- debugging, troubleshooting, fault detection, security, forensics, compliance

Veracity: poor-quality, unstructured nature, complicated analysis

- `2017-07-23T19:35:45Z [0] ERR!: Jack said he will take care of this!`
- this stems from the way logs are generated – messages in natural language

Variability: pervasive devel. practice spanning SW on all IT layers

- data source and data format heterogeneity

Velocity + Volume: can exceed 100,000 entries/sec, 1 MB/s per node

- HP company – 1×10^{12} entries/day generated, 3×10^{9} entries/day processed

Bridging the Gap by Normalization

Data integration perspective: bridge the gap by normalization

- known pattern to improve interoperability
- missing structure is added via transformation and enrichment
- overall heterogeneity is eliminated thanks to a single canonical form

Normalization: unification of data on any of its 4 layers

- data structures
- data types
- data representation
- transport

Our Goal:

Improve **the way log data can be represented and accessed** by normalizing them into streams of event objects.

Research Goal (Simplified)

```
Dec 03 2016 10:03:44 [147.251.11.100] --- INFO: User bob logged in
2016-12-03T10:03:45Z 147.251.20.110 sshd[1551]: session closed for user alice
Dec 03 2016 10:03:46 [147.251.10.125] --- WARN: User alice failed to log in
3.12.2016 10:03:47 147.251.19.160 [Super.java]: {service=Billing, status=0x2A}
```

↓ **NORMALIZATION** ↓

```
   UserLogin() {ts=...424, host="147.251.11.100", success=True, user="bob"}
SessionClosed() {ts=...425, host="147.251.20.110", user="alice", app="sshd"}
    UserLogin() {ts=...426, host="147.251.10.125", success=False, user="alice"}
 ServiceCrash() {ts=...427, host="147.251.19.160", service="Billing", code=42}
```

⇓ UserLogin ⇓ ⇓ SessionClosed ⇓ ⇓ ServiceCrash ⇓

```
CREATE MAP SCHEMA UserLogin(host string, success boolean, user string);

SELECT host, user, count(*) AS attempts
FROM UserLogin.win:time(30 sec)
WHERE attempts > 1000, success=false
GROUP BY host, user
```

Reactive Normalization

Log Abstraction (Separation)

Log Messages	⇒ Message Types ⇒	Regular Expressions
`User Jack logged in`		
`User John logged out`		
`Service sshd started`	`User * logged * : [$1, $2]`	`User (\w+) logged (\w+)`
`User Bob logged in`	`Service * started : [$1]`	`Service (\w+) started`
`Service httpd started`		
`User Ruth logged out`		

```
LOG.info("User {} logged {}", user, action);
↓
Dec 03 2016 10:03:44 -- INFO: User bob logged in
⇝
User (?<user>\w+) logged (?<action>\w+)
```

Log abstraction is a two-tier procedure:

- **message type discovery**
- pattern-matching via regular expressions

Message Type Discovery

Manual discovery: tiresome process, which leads to errors

- automated approaches are necessary

Static code analysis: perfectly possible

- we were able to discover approx. **4500 message types** in Hadoop source code
- source code is not always available (e.g. for network devices)

Data mining: use already generated log messages (historical data)

- 9 existing approaches were studied, e.g. *SLCT, IPLoM, logSig, N-V, ...*

Existing approaches have accuracy and usability issues

Shortcomings of Existing Approaches

Generation of Overlapping Message Types

- `User root logged *`
- `User * logged in`
- `User * logged *`

No Support for Multiple Token Delimiters

- only a single delimiter for tokenization, e.g. 'space'
- limited accuracy

Complicated Parameterization

- each dataset is different and the algortihms sometimes need to be fine-tuned
- some approaches use up to 5 unbounded parameters

No Support for Multi-Word Variables

- `User foo bar logged in`
- `User root logged in`
- `User {1:2} logged {1:1}`

Extended Nagappan-Vouk Algorithm

```
Service sshd started  | [4,2,4]
Service httpd started | [4,2,4]
Service sshd started  | [4,2,4]
Service httpd started | [4,2,4]
---------------------------------
Service   *   started
```

	1	2	3
Service	4	0	0
httpd	0	2	0
sshd	0	2	0
started	0	0	4

Method of *n*-th percentile: frequency table + percentile threshold

- log messages are tokenized via a set of delimiters
- `[4,2,4]` in example is a log message *score*
- word is a variable, if it has a frequency lower than *n*-th percentile of *score*

Post-processing to improve accuracy and usability

1. eliminate overlapping message types by merging
2. identify multi-word variable positions

Discovered Pattern-Set Example

```
Start processing (xor) Jen=user          Service sshd:22 started
User John logged out                     Start processing (xor) Daniel=user
User Bob logged in                       User Ruth logged out
Start processing (xor) Thomas=user       Start processing (xor) Tom Sawyer=user
Service httpd:8080 started               Start processing (nor) Root=user
```

⇓ `percentile=60, delimiters=' :=\(\)'` ⇓

```
regexes: # regex tokens
  INT:          [integer, "[0-9]+"]
  BOOL:         [boolean, "\btrue\b|\bfalse\b"]
  WORD:         [string,  "[0-9a-zA-Z]+"]
  ARBITRARY:    [string,  "[^ \n\r]+"]
  MWRD_1_2:     [string,  "[^ \n\r]+([\\s][^ \n\r]+){0,1}"]

patterns: # patterns describing the message types
  grp0:
    mt1: 'User %{WORD:var1} logged %{WORD:var2}'
    mt2: 'Start processing (%{WORD:var1}) %{MWRD_1_2:var2}=%{WORD:var3}'
    mt3: 'Service %{WORD:var1}:%{INT:var2} started'
```

Evaluation

Discovered message types partition the log messages into groups

F-measure: common accuracy metric in IR, higher is better

- $F = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}$ – how “close“ our grouping is to the ground truth

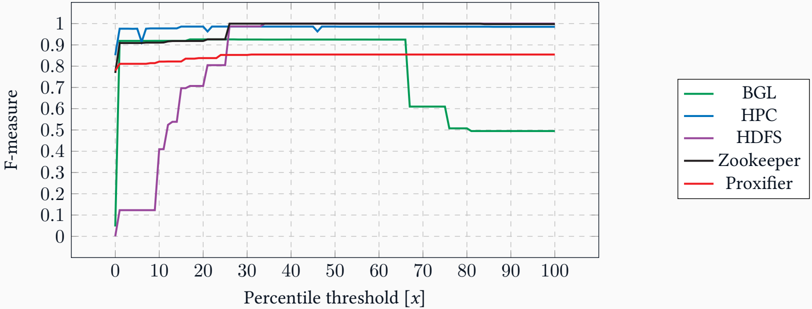
Ground truth: 5 real-life data-sets, MTs manually discovered

- P. He, et al. *An Evaluation Study on Log Parsing and Its Use in Log Mining*
- best average F-measure (*IPLoM*) – ***0.892***

	BGL	HPC	HDFS	Zookeeer	Proxifier	*AVG*
SLCT	0.61	0.81	0.86	0.92	0.89	*0.818*
IPLoM	0.99	0.64	0.99	0.94	0.90	***0.892***
LKE	0.67	0.17	0.57	0.78	0.81	*0.600*
LogSig	0.26	0.77	0.91	0.96	0.84	*0.748*

Results & Findings

	BGL	HPC	HDFS	Zook.	Proxif.	*AVG*
$n = 50, d = space$	0.8556	0.8778	1.0000	0.7882	0.8162	*0.86756*
$n = 50, d = default$	0.9251	0.9861	1.0000	0.9999	0.8547	***0.95316***
$n = 15, d = default$	0.9191	0.9861	0.6965	0.9182	0.8220	*0.86838*
$n = 85, d = default$	0.4949	0.9856	1.0000	0.9979	0.8547	*0.86662*
$n = 50, d = best^*$	0.9985	0.9861	1.0000	0.9999	1.0000	***0.99690***



Summary

Summary & Future Work

The designed algorithm has a very high accuracy on real-world data

Logging code is constantly evolving

How to switch to online (streaming) operation mode?

How to switch to fully-automated mode?

How to version the discovered pattern-sets?

Acknowledgement

EUROPEAN UNION
European Structural and Investment Funds
Operational Programme Research,
Development and Education

MINISTRY OF EDUCATION,
YOUTH AND SPORTS