Semi-Automatic Discovery of Extraction Patterns for Log Analysis

David Carasso
Splunk Inc.
San Francisco, CA
(415) 848-8400
david@splunk.com

Abstract

This paper presents an interactive bootstrapping process used in Splunk that automatically learns to extract fields from log events. End users simply select one or more example values of a field and a learning process discovers additional instances, along with the patterns to extract them. The user is able to correct the instances and save the extraction patterns. Immediately afterward, while searching log events the newly-taught fields will be extracted from the event's raw text.

Categories and Subject Descriptors

I.I.2.6 Artificial Intelligence – Learning : Knowledge Acquisition,

II.I.5.5 Artificial Intelligence – Pattern Recognition : Interactive Systems

General Terms

Algorithms, Experimentation

Keywords

Log files, data center, information retrieval (IR), information extraction (IE).

1. Introduction

Today, the modern datacenter can consist of thousands of servers, each running dozens of applications, which output hundreds of log files, documenting what they are doing and when. It is not atypical for gigabytes of log files to be generated per day. For the system administrator or analyst trying to understand what his system is doing, this is a daunting task.

Consider the failure of one server running out of memory. This might crash a custom database application, with its own log files, resulting in another server's application failing, with its own log files. Eventually, these failures may cause other servers to fail. Trying to forensically diagnose what went wrong involves discovering which machines failed first, which applications on those machines failed, and why.

There are several reasons why this is a particularly hard problem. First, gigabytes of new data may be generated daily. Second, despite expectations, log files are not simple structured records. The truth is that any software programmer who knows how to call printf() is a log producer, and rarely are logs consistent within an application, and certainly not across applications and versions. Sometimes, a logged error event will say "out of memory," but just as likely is an event to indicate out of memory with a “-1” in a very specific area (e.g., the 3rd number after a “]”).

Splunk tries to solve many of the system administrator's problems with a powerful distributed search engine that knows how to handle log files better than traditional document retrieval systems. But critical to the value of searching log files is the ability to understand those events.

In this paper, we describe a semi-automatic method of teaching the system how to extract knowledge at search time from log files. Once the system knows how to extract a field from a log event's raw text, more powerful operations can be performed. For example, rather than just searching for "-1" and returning many irrelevant results, we can then search for "memory_code=-1", where an event's memory_code value is extracted automatically. We can cluster events by memory_code values and notify users when unexpected values are seen. The result is fast, flexible, interactive log analysis.

2. Extraction Patterns

Information extraction (IE) on log files is typically done by manually-created regular expressions. A typical example is Splunk's manual method of specifying well-known codes used in Cisco log files:

```
[cisco-codes]
REGEX = : %(\w+)-(?:\w+-)?[0-7])-(\w+):
FORMAT = product::$1 code::$2 severity::$3
```

This specifies that, from the raw text of a Cisco log event, we should set the "product" field to the first alphanumeric after a ": %"; "code", to the value after that; and "severity" to the value after that. This manual method is time-consuming and error prone, and as a result, we need an automated method to generate extraction patterns.

3. Discovering Extraction Patterns

Several algorithms have been developed to automatically learn to extract values, but most of these algorithms require special training resources, such as marked-up data files or large lists of extraction examples. For the sheer variety of log formats and versions, providing large amounts of training data is decidedly impractical. Further, our goal is to discover good extraction patterns, and not simply the one-time discovery of extracted values.

As the basis for discovering extraction patterns, we looked to Riloff [1] and Soderland [2] on learning dictionaries in the context of natural language processing. Riloff's mutual-bootstrapping technique uses unannotated data and a handful of seed words, from which it bootstraps initial extraction patterns, which are the basis for learning additional words, which in turn refines the extraction patterns.

3.1 Extraction Discovery

Our method differs in several important ways. First, Riloff's goal was to discover good terms, while our goal is to discover good extraction patterns for future use. As a result, we are not directly concerned with scoring extracted terms. Second, our algorithm is used within an interactive environment and therefore allows for user feedback on the terms discovered, thereby affecting the extraction patterns generated. Third, because of the interactive environment, we are less concerned with over-generating incorrect terms, and, as a result, we generate extraction patterns from all known values with each iteration, rather than only from the initial seed values. Finally, our usage does not use linguistic extractions of the form "hijacking of <np>", but uses regular expressions more in tune with the log file world: structure varies less than natural language, surrounding fields can have an unlimited number of values, and punctuation is critical.

Extraction Discovery Algorithm:

```
events = { input log events }
good_values = { input seed field value terms }
bad_values = {}
patterns = {}
for i = 1 to 5:
  # generate patterns for each instance of
  # each good_value in events, keeping track of
  # which values each pattern extracted
  patterns = genPatterns(events, good_values)
  # if too many patterns, just keep the best
  # scoring patterns, where a pattern's score is
  # proportional to the percent of its unique
  # extractions that are in good_values
  prune(patterns)
  # run patterns over events for more extractions
  extractions = extractValues(events, patterns)
  # get feedback from user as to which
  # extractions are erroneous
  bad_values = user feedback on extractions
  # add to good_values any accepted extraction
  good_values += extractions - bad_values
  # remove any pattern that generated a bad_value
  removePatterns(patterns, bad_values)
  if no change in patterns: break
save(patterns)
```

3.2 Pattern Generation

Given the raw log samples in Listing 1, to extract "crond", one could imagine a large variety of regular expressions that could be generated, requiring rigid conformity to the number and type of characters before and after the process_name field value in question, as well as for the field value itself. For example, one could imagine an expression such as:

> [A-Z][a-z][a-z] [0-9][0-9] [0-9][0-9]:[0-9][0-9]:[0-9][0-9] [a-z]{7} ([a-z]{5})\([a-z]{3}_[a-z]{4})....

Through empirical trial-and-error, it became clear that it is highly desirable for users to easily understand the patterns generated, as they may need to manually modify them in rare situations. Therefore, simpler patterns, even at the expense of some over-generalization, are desirable. Specifically:

- ●Rarely is the number of characters critical when character types (e.g., alphabetic, numeric, whitespace, etc.) consecutively repeat. Additionally, alphabetic case is rarely critical. Thus, "chrond", rather than being represented as "[a-z]{6}", could be represented as "\w+" (one or more alphanumeric values).
- ●Rarely is it important to specify what comes after the extracted field value. We limit this constraint to just the single punctuation character after the extracted value. Thus we only specify that the process_name must end with a "(" or "[", rather than fully specifying what comes after.
- ●In log files most punctuation is so structurally meaningful that it is enough to count the number of occurrence of the punctuation just before the value, rather than fully specifying the complete expression that occurs before the field value.

Thus, rather than fully specifying the prefix "Mar 10 16:49:29 mcdavid " to get to "chrond", we skip past the first two colons to get in the ballpark, and then skip past the next word (e.g., "mcdavid"). The result is a concise and understandable pattern.

Pattern Generation Algorithm

```
genPatterns(events, values):
  patterns = {}
  for each event in events:
    for each value in values:
      if value in event:
        patterns += genPattern(event, value)
  return patterns

genPattern(event, value):
  start = event.position(value)
  end = start + value.length()
  prefix = genPrefixRegex(event[0:start])
  value_regex = genRegex(value)
  suffix = event[end]
  punct = "\t()[]{}*+^$!-\\?!@#%+=:<>,? "
  if suffix not in value and suffix not in punct:
    suffix = ""
  return prefix+'('+value_regex+')'+suffix

genPrefixRegex(prefix):
  puncts = "\t()[]{}*+^$!-\\?!@#%+=:<>,?"
  last_pos = pos of right most punct in prefix
  last_punct = prefix[last_pos]
  count = count of last_punct in prefix
  # regex skips count of the last punct
  regex = "(?:.*?"+last_punct+"){"+count+"}"
  # add on any values after the last punct
  regex += genRegex(prefix[last_punct+1:])
```

```
# given "mcdavid <613>" return "\w+ <\d+>"
genRegex(value):
  regex = ""
  for each ch in value:
    if ch is alphabetic or numeric:
       handle case where previous ch
       was the same type and just append "+"
    # for a-z, use \w; for 0-9, use \d,
    # otherwise use append literal character
    regex += regex_type(ch)
  return regex
```

4. Example Usage

Given a set of syslog events (sample in Listing 1), an actual user learning to extract fields might operate as follows:

1. User searches his server and narrows in on a specific syslog file. He discovers in the search results that the system does not know about the "process_name" field. Given an event such as "Mar 10 16:50:02 mcdavid crond(pam_unix)[9639]: session closed for user root", the process_name would be "crond".
2. User selects the "crond" text and clicks on "Learn Field".
3. The system takes the search results and the seeding term, "crond", and discovers an initial set of extraction patterns, showing the patterns, the values they extracted, and the search results. (Output 1 shows these results.)
4. The user notices that the search results have an "ntpd" process, but it is not one of the extracted process names. The ntpd syslog events have a slightly different format, and the pattern generated did not extract it. The pattern learned in Output 1 works on the first two examples of Listing 1, but not the third, which has a different format.
5. The user tells the system that it should have also extracted "ntpd" and the system now relearns the patterns given the seeds "crond" and "ntpd". The system then correctly handles the user's field. (Output 2 shows these results.)
6. If the user is satisfied, he tells the system to save the extraction patterns and use them on syslog events. At his next seach of syslog events, process_name will be extracted at search time and can be used in the search itself. For example, the user can now search for "opened for user root process_name=crond".
7. At any point the user can now upload his extraction patterns to SplunkBase.com, Splunk's community knowledge-sharing site, effectively sharing them with the tens of thousands of users. Over time, all file types and versions can be effectively covered by the community.

Listing 1. Raw Log Event Samples

```
Mar 10 16:49:29 mcdavid su(pam_unix)[9596]: session opened for user root by (uid=500)
Mar 10 16:50:01 mcdavid crond(pam_unix)[9638]: session opened for user root by (uid=0)
Mar 10 16:56:32 mcdavid ntpd[2544]: synchronized to 138.23.180.126, stratum 2
```

Output 1. Initial Extraction Patterns

Input:
"crond" seed term and raw log events from Listing 1.

Output Patterns:
```
# skip to the second colon, then past a number, a word,
# and finally extract the word before an open parenthesizes:
(?:.*?:){2}\d+ \w+ (\w+)\(
```
Output Extractions:
crond, packet, sshd, su

Output 2. Improved Extraction Patterns

Input:
"crond, ntpd" seed terms and raw log events from Listing 1.

Output Patterns:
```
# skip to the second colon, then past a number, a word,
# and finally extract the word before an open parenthesizes
# or an open square bracket:
(?:.*?:){2}\d+ \w+ (\w+)\(
(?:.*?:){2}\d+ \w+ (\w+)\[
```
Output Extractions:
crond, packet, sshd, su, snmpd, osirisd, splunkd, auditd

5. Conclusion

In this paper, we have described an interactive extraction learning algorithm that discovers useful regular expression patterns. With simple point-and-click actions, the search user is empowered to name any unknown field, and teach the system how to extract its values, freeing users from manually creating regular expressions.

6. References

[1]E. Riloff and R. Jones. 1999. Learning Dictionaries for Information Extraction by Multi-Level Bootstrapping. In Proceedings of the AAAI-99. http://www.cs.utah.edu/~riloff/publications.html

[2]Soderland, S.; Fisher, D.; Aseltine, J.; and Lehnert, W. Issues in Inductive Learning of Domain-Specific Text Extraction Rules. In Proceedings of the Workshop on New Approaches to Learning for Natural Language Processing at the Fourteenth International Joint Conference on Artificial Intelligence, 1995. http://citeseer.ist.psu.edu/soderland95issues.html