Logparser Documentation

Release 0.1

LogPAI

Jul 06, 2019
Get Started

1 Demo 3
2 Dependency 5
3 Install docker 7
4 Build docker images 9
5 Overview 11
6 SLCT 13
7 IPlOM 15
8 LKE 17
9 LogSig 19
10 Spell 21
11 Drain 23
12 Papers 25
13 Benchmarks 29
14 Contributors 31
Logparser provides a toolkit and benchmarks for automated log parsing, which is a crucial step towards structured log analytics. By applying logparser, users can automatically learn event templates from unstructured logs and convert raw log messages into a sequence of structured events.
The logparser toolkit is implemented with Python and requires a number of dependency requirements installed. Users are encouraged to set up the local environment for logparser with Anaconda. However, for ease of reproducing our benchmark results, we have built docker images for the running environments. Docker is a popular container technology used in production. If you have docker installed, you can easily pull and run docker containers as follows:

```bash
$ mkdir logparser
$ docker run --name logparser_py2 -it -v logparser:/logparser logpai/logparser:py2
```

Note that if you are going to try MoLFI, which requires Python 3, please run the following container:

```bash
$ mkdir logparser
$ docker run --name logparser_py3 -it -v logparser:/logparser logpai/logparser:py3
```

After starting the docker containers, you can run the demos of logparser on the HDFS sample log:

```bash
$ git clone https://github.com/logpai/logparser.git /logparser/
$ cd /logparser/demo/
$ python Drain_demo.py
```

The logparser demo/benchmark scripts will produce both event templates and structured logs in the result directory:

- HDFS_2k.log_templates.csv
- HDFS_2k.log_structured.csv
The logparser toolkit has the following requirements by default. We recommend users to use Anaconda, which is a popular Python data science platform with many common packages pre-installed.

- python 2.7
- scipy
- numpy
- scikit-learn
- pandas

Some tools have additional dependency requirements:

- SLCT: gcc 4.8.5
- LogCluster: perl 5.22
- MoLFI: python 3.6, deap 1.2.2
- POP: pyspark
CHAPTER 3

Install docker

This is a note showing the steps of installing docker on Ubuntu platforms. If you need more detailed information, please check docker documentation at: https://docs.docker.com/install/linux/docker-ce/ubuntu

**Note:** Uninstall old docker versions if any:

```
$ sudo apt-get remove docker docker-engine docker.io
```

- **Ubuntu 14.04**

  Install `linux-image-extra-*` to allow Docker to use the aufs storage drivers.

  ```
  $ sudo apt-get update
  $ sudo apt-get install \
  linux-image-extra-$\(uname -r\) \
  linux-image-extra-virtual
  ```

  Download docker package file `docker-ce_17.03.2~ce-0~ubuntu-trusty_amd64.deb`.

  ```
  $ sudo dpkg -i ~/docker/docker-ce_17.03.2~ce-0~ubuntu-trusty_amd64.deb
  ```

- **Ubuntu 16.04**

  Download docker package file `docker-ce_17.03.2~ce-0~ubuntu-xenial_amd64.deb`.

  ```
  $ sudo dpkg -i ~/docker/docker-ce_17.03.2~ce-0~ubuntu-xenial_amd64.deb
  ```

Verify that Docker CE is installed correctly by running the hello-world image:

```
$ sudo docker run hello-world
```

Add user to the docker group to run docker commands without sudo:
$ sudo groupadd docker
$ sudo usermod -aG docker $USER
Build docker images

Build logpai/logparser:py2

```
$ sudo docker run --name logparser_py2 -it ubuntu:16.04 bash
$ apt-get update
$ apt-get install -y wget bzip2
$ apt-get install -y gcc perl git
$ rm -rf /var/lib/apt/lists/*

$ cd /
$ mkdir anaconda
$ cd anaconda
$ wget https://repo.anaconda.com/archive/Anaconda2-5.2.0-Linux-x86_64.sh
$ bash Anaconda2-5.2.0-Linux-x86_64.sh
$ source ~/.bashrc
$ cd ..
$ rm -r anaconda
$ exit

$ docker commit logparser_py2 logpai/logparser:py2
$ docker login
$ docker push logpai/logparser:py2
```

Build logpai/logparser:py3

```
$ sudo docker run --name logparser_py3 -it ubuntu:16.04 bash
$ apt-get update
$ apt-get install -y wget bzip2 git
$ rm -rf /var/lib/apt/lists/*

$ cd /
$ mkdir anaconda
$ cd anaconda
```

(continues on next page)
$ wget https://repo.anaconda.com/archive/Anaconda3-5.1.0-Linux-x86_64.sh
$ bash Anaconda3-5.1.0-Linux-x86_64.sh
$ source ~/.bashrc
$ cd ..
$ rm -r anaconda

$ docker commit logparser_py3 logpai/logparser:py3
$ docker login
$ docker push logpai/logparser:py3
Logparser aims to provide a set of open-source tools that are ready for use in production. By applying logparser, users can automatically learn event templates from unstructured logs and convert raw log messages into a sequence of structured events. The following figure illustrates an overview of log parsing.
SLCT is a simple logfile clustering tool designed to find clusters in logfile(s), so that each cluster corresponds to a certain line pattern that occurs frequently enough. With the help of SLCT, one can quickly build a model of logfile(s), and also identify rare lines that do not fit the model (and are possibly anomalous).

**Step 1:** Word vocabulary construction. SLCT makes a pass over the words in all the logs and count the occurrence of them. In this step, the position of the word is also considered. For example, “send” as the 1st word of a log and “send” as the 2nd word of a log are considered different. Word occurs more than support threshold, say N, is defined as frequent word.

**Step 2:** Cluster candidates construction. In this step, SLCT makes the second pass over all the logs, while at this time it focuses on frequent words. All the frequent words in a log will be extracted be the log template of itself. The number of logs that match a certain log template is counted, and each log template represents a cluster candidate.

**Step 3:** Log template extraction. SLCT goes through all cluster candidates and log templates whose corresponding cluster contains more than N logs are selected as the output templates. The logs of clusters which are not selected are placed into outlier class.

**Step 4:** Cluster combination. This step is optional. SLCT could make a pass through all selected clusters and combine two clusters if one of them is the subcluster of the other. For example, cluster “PacketResponder 1 for block * terminating” is the subcluster of “PacketResponder * for block * terminating”. Therefore, these two clusters will be combined in this step.

To provide a common interface for log parsing, we write a Python wrapper around the original SLCT source code in C (released under GPL license). This also eases our benchmarking experiments. Same with the original release, our implementation has only been tested successfully on Linux (compiled with GCC). We tried running the tool on Windows using cygwin with GCC installed, but failed with a crash. You are advised to use the SLCT tool on Linux. But it is still possible to work around the issue if some efforts are made.

Read more information about SLCT from the following paper:

IPLoM (Iterative Partitioning Log Mining) is one of the state-of-the-art algorithms for log parsing. It leverages the unique characteristics of log messages for iterative log partitioning, which thus enables efficient message type extraction. Since the original open-source implementation is not available anymore, we re-implement the algorithm using Python with a nice interface provided. We describe the process of IPLoM as follows.

**Step 1**: Partition by event size. Logs are partitioned into different clusters according to its length. In real world logs, it is possible that logs belong to one template are in variable length. In this case, the result of IPLoM should be postprocessed manually.

**Step 2**: Partition by token position. At this point, each cluster contains logs with the same length. Assuming there are \( m \) logs whose length are \( n \) in a cluster, this cluster can be regarded as an \( m \)-by-\( n \) matrix. This step based on the assumption that the column with least number of unique words (split word position) is the one contains constants. Thus, the split word position is used to partition each cluster, i.e. each generated cluster has the same word in the split word position.

**Step 3**: Partition by search for mapping. In this step, two columns of the logs are selected for further partitioning based on the mapping relation between them. To determine the two columns, the number of unique words in each column is counted (i.e. word count) and the two columns with the most frequently appearing word count are selected. There are four mapping relations: 1-1, 1-M, M-1, M-M. In the case of 1-1 relations, logs contains the same 1-1 relations in the two selected columns are partitioned to the same cluster. For 1-M and M-1 relations, we should firstly decide whether the M side column contains constants or variables. If the M side contains constants, the M side column is used partition logs in 1-M/M-1 relations. Otherwise, the 1 side column is used. Finally, logs in M-M relations are partitioned to one cluster.

**Step 4**: Log template extraction. IPLoM processes through all the clusters generated in previous steps and generates one log template for each of them. For each column in a cluster, the number of unique words is counted. If there is only one unique word in a column, the word is regarded as constant. Otherwise, the words in the column are variables and will be replaced by a wildcard in the output.

Read more information about IPLoM from the following papers:

- Adetokunbo Makanju, A. Nur Zincir-Heywood, Evangelos E. Milios. A Lightweight Algorithm for Message Type Extraction in System Application Logs, *IEEE Transactions on Knowledge and Data Engineering (TKDE)*,
LKE (Log Key Extraction) is one of the representative algorithms for log parsing. It first leverages empirical rules for preprocessing and then uses weighted edit distance for hierarchical clustering of log messages. After further group splitting with fine tuning, log keys are generated from the resulting clusters.

**Step 1**: Log clustering. Weighted edit distance is designed to evaluate the similarity between two logs, $WED = \sum_{i=1}^{n} \frac{1}{1+e^{(x_i-v)}}$. $n$ is the number of edit operations to make two logs the same, $x_i$ is the column index of the word which is edited by the $i$-th operation, $v$ is a parameter to control weight. LKE links two logs if the $WED$ between them is less than a threshold $\sigma$. After going through all pairs of logs, each connected component is regarded as a cluster. Threshold $\sigma$ is automatically calculated by utilizing K-means clustering to separate all $WED$ between all pair of logs into 2 groups, and the largest distance from the group containing smaller $WED$ is selected as the value of $\sigma$.

**Step 2**: Cluster splitting. In this step, some clusters are further partitioned. LKE firstly finds out the longest common sequence (LCS) of all the logs in the same cluster. The rests of the logs are dynamic parts separated by common words, such as “/10.251.210:55700” or “blk_904791815409399662”. The number of unique words in each dynamic part column, which is denoted as $|DP|$, is counted. For example, $|DP| = 2$ for the dynamic part column between “src:” and “dest:” in log 2 and log 3. If the smallest $|DP|$ is less than threshold $\phi$, LKE will use this dynamic part column to partition the cluster.

**Step 3**: Log template extraction. This step is similar to the step 4 of IPLoM. The only difference is that LKE removes all variables when they generate log templates, instead of representing them by wildcards.

Read more information about LKE from the following paper:

LogSig is a message signature based algorithm to generate system events from textual log messages. By searching the most representative message signatures, logSig categorizes log messages into a set of event types. logSig can handle various types of log data, and is able to incorporate human’s domain knowledge to achieve a high performance. We implemented LogSig using Python with a nice interface for benchmarking purpose.

**Step 1**: Word pair generation. In this step, each log is converted to a set of word pairs. For example, “Verification succeeded for blk_904791815409399662” is converted to the following word pairs: (Verification, succeeded), (Verification, for), (Verification, blk_904791815409399662), (succeeded, for), (succeeded, blk_904791815409399662), (for, blk_904791815409399662). Each word pair preserves the order information of the original log.

**Step 2**: Clustering. LogSig requires users to determine the number of clusters, say k, which leads to k randomly partitioned clusters of logs at the beginning of clustering. In each iteration of clustering, LogSig goes through all the logs and move them to other clusters if needed. For each log, potential value, which is based on word pairs generated in step 1, is calculated to decide to which cluster the log should be moved. LogSig keeps clustering until no log is decided to move in one iteration.

**Step 3**: Log template extraction. At this point, there are k clusters of logs. For each cluster, words in more than half of the logs are selected as candidate words of the template. To figure out the order of candidate words, LogSig goes through all the logs in the cluster and count how many times each permutation appears. The most frequent one is the log template of the cluster.

Read more information about LogSig from the following papers:

Spell is an online streaming method to parse logs, which utilizes a longest common subsequence based approach. The key observation is that, if we view the output by a log printing statement (which is a log entry) as a sequence, in most log printing statements, the constant that represents a message type often takes a majority part of the sequence and the parameter values take only a small portion. If two log entries are produced by the same log printing statement stat, but only differ by having different parameter values, the LCS of the two sequences is very likely to be the constant in the code stat, implying a message type.

Initially, the LCSMap list is empty. When a new log entry $e_i$ arrives, it is firstly parsed into a token sequence $s_i$ using a set of delimiters. After that, we compare $s_i$ with the LCSseq’s from all LCSObjects in the current LCSMap, to see if $s_i$ “matches” one of the existing LCSseq’s (hence, line id $i$ is added to the lineldds of the corresponding LCSObject), or we need to create a new LCSObject for LCSMap.

Spell’s workflow is as follows:

Read more information about Drain from the following paper:

Drain is one of the representative algorithms for log parsing. It can parse logs in a streaming and timely manner. To accelerate the parsing process, Drain uses a fixed depth parse tree (See the figure below), which encodes specially designed rules for parsing.
Drain first preprocess logs according to user-defined domain knowledge, i.e., regex. Second, Drain starts from the root node of the parse tree with the preprocessed log message. The 1-st layer nodes in the parse tree represent log groups whose log messages are of different log message lengths. Third, Drain traverses from a 1-st layer node to a leaf node. Drain selects the next internal node by the tokens in the beginning positions of the log message. Then Drain calculates similarity between log message and log event of each log group to decide whether to put the log message into existing log group. Finally, Drain updates the Parser Tree by scanning the tokens in the same position of the log message and the log event.

Read more information about Drain from the following paper:

A list of papers about log parsing


All the log parsers have been evaluated on loghub log samples. We report parsing accuracy as the percentage of accurately parsed log messages. Note that accuracy values above 0.9 are marked in bold, and the best accuracy results achieved are marked with *.
<table>
<thead>
<tr>
<th>Tools</th>
<th>HDFS</th>
<th>Hadoop</th>
<th>Spark</th>
<th>Zookeeper</th>
<th>Open-</th>
<th>BGL</th>
<th>HPC</th>
<th>Thunder-</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLCT</td>
<td>0.545</td>
<td>0.423</td>
<td>0.685</td>
<td>0.726</td>
<td>0.867</td>
<td>0.573</td>
<td>0.839</td>
<td>0.882</td>
</tr>
<tr>
<td>AEL</td>
<td>0.998</td>
<td>0.538</td>
<td>0.905</td>
<td>0.921</td>
<td>0.758</td>
<td>0.957</td>
<td>0.903</td>
<td>0.941</td>
</tr>
<tr>
<td>IPLoM</td>
<td>1*</td>
<td>0.954</td>
<td>0.920</td>
<td>0.962</td>
<td>0.871</td>
<td>0.939</td>
<td>0.824</td>
<td>0.663</td>
</tr>
<tr>
<td>LKE</td>
<td>1*</td>
<td>0.670</td>
<td>0.634</td>
<td>0.438</td>
<td>0.787</td>
<td>0.128</td>
<td>0.574</td>
<td>0.813</td>
</tr>
<tr>
<td>LFA</td>
<td>0.885</td>
<td>0.900</td>
<td>0.994</td>
<td>0.839</td>
<td>0.200</td>
<td>0.854</td>
<td>0.817</td>
<td>0.649</td>
</tr>
<tr>
<td>LogSig</td>
<td>0.850</td>
<td>0.633</td>
<td>0.544</td>
<td>0.738</td>
<td>0.866</td>
<td>0.227</td>
<td>0.354</td>
<td>0.694</td>
</tr>
<tr>
<td>SHISO</td>
<td>0.998</td>
<td>0.867</td>
<td>0.906</td>
<td>0.660</td>
<td>0.722</td>
<td>0.711</td>
<td>0.325</td>
<td>0.576</td>
</tr>
<tr>
<td>LogCluster</td>
<td>0.546</td>
<td>0.563</td>
<td>0.799</td>
<td>0.732</td>
<td>0.696</td>
<td>0.835</td>
<td>0.788</td>
<td>0.599</td>
</tr>
<tr>
<td>LenMa</td>
<td>0.998</td>
<td>0.885</td>
<td>0.884</td>
<td>0.841</td>
<td>0.743</td>
<td>0.690</td>
<td>0.830</td>
<td>0.943</td>
</tr>
<tr>
<td>LogMine</td>
<td>0.851</td>
<td>0.870</td>
<td>0.576</td>
<td>0.688</td>
<td>0.743</td>
<td>0.723</td>
<td>0.784</td>
<td>0.919</td>
</tr>
<tr>
<td>Spell</td>
<td>1*</td>
<td>0.778</td>
<td>0.905</td>
<td>0.964</td>
<td>0.764</td>
<td>0.787</td>
<td>0.654</td>
<td>0.844</td>
</tr>
<tr>
<td>Drain</td>
<td>0.998</td>
<td>0.948</td>
<td>0.920</td>
<td>0.967</td>
<td>0.733</td>
<td>0.963</td>
<td>0.887</td>
<td>0.955</td>
</tr>
<tr>
<td>MoLFI</td>
<td>0.998</td>
<td>0.957</td>
<td>0.418</td>
<td>0.839</td>
<td>0.213</td>
<td>0.960</td>
<td>0.824</td>
<td>0.646</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tools</th>
<th>Windows</th>
<th>Linux</th>
<th>Mac</th>
<th>Android</th>
<th>HealthApp</th>
<th>Apache</th>
<th>OpenSSH</th>
<th>Proxifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLCT</td>
<td>0.697</td>
<td>0.297</td>
<td>0.558</td>
<td>0.882</td>
<td>0.331</td>
<td>0.731</td>
<td>0.521</td>
<td>0.518</td>
</tr>
<tr>
<td>AEL</td>
<td>0.690</td>
<td>0.673</td>
<td>0.764</td>
<td>0.682</td>
<td>0.568</td>
<td>1*</td>
<td>0.538</td>
<td>0.518</td>
</tr>
<tr>
<td>IPLoM</td>
<td>0.567</td>
<td>0.672</td>
<td>0.673</td>
<td>0.712</td>
<td>0.822</td>
<td>1*</td>
<td>0.802</td>
<td>0.515</td>
</tr>
<tr>
<td>LKE</td>
<td>0.990</td>
<td>0.519</td>
<td>0.369</td>
<td>0.909</td>
<td>0.592</td>
<td>1*</td>
<td>0.426</td>
<td>0.495</td>
</tr>
<tr>
<td>LFA</td>
<td>0.588</td>
<td>0.279</td>
<td>0.599</td>
<td>0.616</td>
<td>0.549</td>
<td>1*</td>
<td>0.501</td>
<td>0.026</td>
</tr>
<tr>
<td>LogSig</td>
<td>0.689</td>
<td>0.169</td>
<td>0.478</td>
<td>0.548</td>
<td>0.235</td>
<td>0.582</td>
<td>0.373</td>
<td>0.967</td>
</tr>
<tr>
<td>SHISO</td>
<td>0.701</td>
<td>0.672</td>
<td>0.595</td>
<td>0.585</td>
<td>0.397</td>
<td>1*</td>
<td>0.619</td>
<td>0.517</td>
</tr>
<tr>
<td>LogCluster</td>
<td>0.713</td>
<td>0.629</td>
<td>0.604</td>
<td>0.798</td>
<td>0.531</td>
<td>0.709</td>
<td>0.426</td>
<td>0.951</td>
</tr>
<tr>
<td>LenMa</td>
<td>0.566</td>
<td>0.701</td>
<td>0.698</td>
<td>0.880</td>
<td>0.174</td>
<td>1*</td>
<td>0.925</td>
<td>0.508</td>
</tr>
<tr>
<td>LogMine</td>
<td>0.993</td>
<td>0.612</td>
<td>0.872</td>
<td>0.504</td>
<td>0.684</td>
<td>1*</td>
<td>0.431</td>
<td>0.517</td>
</tr>
<tr>
<td>Spell</td>
<td>0.998</td>
<td>0.605</td>
<td>0.757</td>
<td>0.919</td>
<td>0.639</td>
<td>1*</td>
<td>0.554</td>
<td>0.527</td>
</tr>
<tr>
<td>Drain</td>
<td>0.997</td>
<td>0.690</td>
<td>0.787</td>
<td>0.911</td>
<td>0.780</td>
<td>1*</td>
<td>0.788</td>
<td>0.527</td>
</tr>
<tr>
<td>MoLFI</td>
<td>0.406</td>
<td>0.284</td>
<td>0.636</td>
<td>0.788</td>
<td>0.440</td>
<td>1*</td>
<td>0.50</td>
<td>0.013</td>
</tr>
</tbody>
</table>
CHAPTER 14

Contributors

- Jieming Zhu, The Chinese University of Hong Kong, currently at Huawei 2012 Labs.
- Pinjia He, The Chinese University of Hong Kong
- Shilin He, The Chinese University of Hong Kong
- Jinyang Liu, Sun Yat-Sen University