# A novel pattern-based edit distance for automatic log parsing: Implementation and reproducibility notes

Maxime Raynal[1,2], Marc-Olivier Buob[1], and Georges Quénot[2]

[1] Nokia Bell Labs, France
[2] Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, F-38000 Grenoble France

**Abstract.** This paper presents a detailed and reproducible description of the algorithms and experiments published in our ICPR paper (*A novel pattern-based edit distance for automatic log parsing*). It discusses the implementation, our methodology, our experimental setup, the considered performance metrics and the influence of the main parameters of the compared algorithms.

**Keywords:** Edit distance · log clustering · dynamic programming · reproducible research.

## 1 Introduction

Unstructured data is ubiquitous, and its lack of structure makes it difficult to analyze. As a sequel, it often ends up being unused [19]. In practice, processing unstructured data forces to develop dedicated parsers to convert it to a more convenient and structured format. This problem arises in network management, especially when analyzing system *logs*[3] or system command outputs. Unfortunately, developing parsers is often tedious, time consuming and error prone.

To automate log parsing, it is required to better understand the structure of the file that must be processed. In the literature, grouping the lines of a log having the same underlying structure and semantics is often referred to as the *log clustering* (and sometimes, *log parsing*) problem. To solve this problem, we propose in [18] a novel pattern-based distance and a clustering algorithm built on top of it. As a result, our clustering algorithm partitions input log lines so that each group of lines conforms to a same underlying structure (*template*). It worth noting that the templates are not known *a priori* and are inferred during the log clustering step.

This companion paper details how to reproduce our experiments. Section 2 explains some of our implementation choices. Section 3 describes how to install, setup and use our module through a minimal example. Section 4 recalls the main steps of our algorithm and details how to tune each hyper-parameter. Section 5 presents how we compared though experiments our proposal against two state of the art solutions and discusses the influence of each hyper parameter. Section 6 concludes the paper.

[3] Logs are text files, where each line usually corresponds to a timestamped message.

## 2 Implementation considerations

The `pattern clustering` code architecture involves C++ and Python 3 code. The core algorithm is implemented in C++, while the Python wrapping eases its usage. Implementing the core algorithm in C++ improves the performance of the pattern clustering by a factor of 100 compared to a pure Python implementation, and hence allows to process larger log files. The Python wrapping is realized thanks to `libpython` and the `Boost.python` libraries [1]. As the module includes a C++ core, once compiled, the `pattern_clustering` module only works for the python version corresponding to the `libpython` and `Boost.python` libraries we linked to.

We could have restricted to `libpython`, but we decided to also rely on the Boost library for two reasons. First, `Boost.python` allows to keep the C++ core independent from the implementation details imposed by `libpython`. Second, wrapping C++ objects usable from the python interpreter imposes to build the appropriate `libpython` objects. This task is significantly eased by using the `Boost.python` library.

In our case, the `pattern_distance` and `pattern_clustering` primitives take in parameter pattern automata and vectors [18]. All these variables are represented C++-side by using vectors. As vectors are not handy to craft automata, we require an intermediate automaton-like class. We could have used `Boost.graph`, but for sake of simplicity, we kept the graph aspects in the Python part of module.

To do so, we decided to use the `pybgl` Python module [10] for two reasons. First, `pybgl` provides an automaton class that can easily extended to implement pattern automata. Second, it provides all the primitives required to build an automaton from an arbitrary regular expression. Once the pattern automata are built, the Python/C++ binding allows to transparently convert them to C++ vectors (as well as Python lists), and conversely, to transform C++ results (vectors) to Python lists.

## 3 Installation steps

The installation steps are described in the wiki of the `pattern_clustering` repository [9]. The installation of the `libboost-dev` and `libpython3-dev` libraries is explained. Unfortunately, a PIP-based installation is not yet available as it would require to compile a version for every target operating system and Python version. Doing so is not straightforward, even using projects like ManyLinux [8], and that is why we decided to provide only a source-based installation.

Once installed, the end-user can run the minimal example provided in Figure 1. To obtain more user-friendly results, we refer the user to the Jupyter notebooks provided in the `pattern_clustering` repository.

```python
from pattern_clustering import pattern_clustering

FILENAME = "/var/log/Xorg.0.log" # Or any arbitrary log file
with open(FILENAME) as f:
    LINES = [line.strip() for line in f.readlines()]

print(pattern_clustering(LINES))
```

Fig. 1: Minimal example using the `pattern_clustering` function.

## 4 Pattern clustering usage

This section presents the parameters of our module and more advanced usages than the one provided in Figure 1. It also discusses how to tune each hyper-parameter if the default settings are not satisfactory.

As explained in [18], the `pattern_clustering` primitive takes the following parameters:

- `lines`: an iterable object (e.g., a `list`) of strings corresponding to each input log line. As the pattern clustering algorithm is greedy, one could pass an iterator allowing to process an input log file in a streaming fashion.
- `map_name_dfa`: a dictionary mapping pattern names with the corresponding deterministic finite automaton (DFA). In [18], this corresponds to the pattern collection denoted by $\mathcal{P}$. We detail this parameter in Section 4.1.
- `densities`: the density of each pattern is a value between 0 and 1 reflecting how strict is a pattern. In [18], this corresponds to results returned by the density function $\rho$. The vector of densities offers the opportunity to use alternative density functions.
- `threshold`: this value, between 0 and 1, indicates how close must be the elements involved in a cluster from the cluster's representative. In [18], this corresponds to $D$. Small values tend to increase the number of output clusters.
- `use_async`: a Boolean indicating whether the pattern clustering computations must be parallelized.
- `make_mg`: the strategy used to build pattern automata from string according to $\mathcal{P}$. The end-user must keep the default value to conform to our reproduce the pattern automata simplifications and experiments presented in [18].

The value returned by the `pattern_clustering` primitive is detailed in Section 4.2. Finally, Section 4.3 presents two ways to perform the clustering.

### 4.1 Pattern collection

In our implementation, each pattern is identified by a string. Some patterns are predefined in our module and one can get the whole list of supported patterns by running the following snippet:

```
from pattern_clustering import get_pattern_names
print(get_pattern_names())
```

Fig. 2: Snippet illustrating how to list the predefined patterns.

```
from pattern_clustering import *
from pybgl.regexp import compile_dfa

MAP_NAME_DFA = make_map_name_dfa()
MAP_NAME_DFA["letters"] = compile_dfa("[a-zA-Z]+")
print(pattern_clustering(LINES, map_name_dfa=MAP_NAME_DFA))
```

Fig. 3: Snippet illustrating how to tune the pattern collection.

By default, the pattern collection involves most of them. One may tune this collection by discarding some keys, modifying some automata, or inject custom patterns. The example below shows how to inject a custom pattern named `letters` in the default collection. In the details, the `compile_dfa` processes the input regular expression using the Shunting Yard algorithm [12]. By using the Thompson transformation [20], it progressively builds a non-deterministic finite automaton (NFA). Finally, the NFA is transformed to its corresponding minimal DFA using the Moore algorithm [17].

### 4.2 Returned value

Once the clustering is computed, each input line is remapped with the appropriate cluster. The `pattern_clustering` returns a list where each element represents a cluster. The $i^{th}$ element of this list gathers the line number of the lines belonging to the $i^{th}$ cluster.

### 4.3 Dropping duplicated pattern automata

Our module allows to drop duplicated pattern automata. This feature is relevant if the end-user considers that every line conforming to the same pattern automaton must always fall in the same cluster. Dropping duplicated pattern automata limits the number of elements to cluster and thus accelerates the processing.

However, we did not use this feature in our experiments. Indeed, we observed that it could affect the quality of the clustering, as in some situations, two lines conforming to the same pattern automaton should fall in distinct clusters.

## 5 Experimental setup

In [18], we compare the pattern clustering against two state-of-the-art algorithms, namely Drain [14] and LogMine [13]. Our experimental setup is quite similar to the one described in [21]. This section details the main differences.

### 5.1 Drain and LogMine integration

The standard implementations of Drain [3] and LogMine [7] do not output the cluster assigned to each input log line. This information is required to compute the accuracy (see Section 5.5). That is why we have forked these standard implementations and adapted their outputs [2,6]. Our modifications are minor, and thus do not affect the results and only induce negligible time overhead.

### 5.2 Loghub dataset

We perform our experiments on the Loghub dataset. It involves 16 log files described in detail in [15] (size, number of messages, labeling, etc.). The Loghub repository [5] provides a small excerpt of each log file, whereas the Zenodo repository [11] contains the complete logs.

### 5.3 Ground truth

The Loghub repository [5] provides for each log file the corresponding ground truth. A ground truth maps templates (i.e., a string involving some wildcards denoted by `<*>`) with the corresponding lines of log.

It's worth noting that each ground truth has been manually obtained. During our experiments, we have observed they contain several inconsistencies. In particular, we have found some clusters that have no reason to be split. For example, the original *Android* ground truth distinguishes the three following templates:

- `animateCollapsePanels:flags=<*>,`
  `force=false, delayed=false, mExpandedVisible=false`
- `animateCollapsePanels:flags=<*>,`
  `force=false, delayed=false, mExpandedVisible=true`
- `animateCollapsePanels:flags=<*>,`
  `force=true, delayed=true, mExpandedVisible=true`

... while it would be more natural to merge them in a single template:

- `animateCollapsePanels:flags=<*>,`
  `force=<*>, delayed=<*>, mExpandedVisible=<*>`

We have checked each ground truth and fixed all the inconsistencies we have found. The original and the fixed versions of the ground truths are made available in [4]. One may easily compare them using a `diff`-like utility. All our experiments are performed using the fixed ground truths.

### 5.4 Experimental parameters

The three considered clustering algorithms mainly require two parameters, namely the pattern collection and the clustering threshold.

For each dataset, our experiments consider two pattern collections:

- *Minimal collection.* Our initial motivation is to design a generic log clustering tool, and thus this collection only includes universal patterns (i.e., patterns like dates, times, network addresses, numerical values).
- *Specific collections.* In [21], the authors tailor dataset-dependant to see how good each algorithm with a high prior knowledge could be. As a sequel, the resulting collection is highly dependent on the input dataset and requires significant end-user intervention.

To get a full benchmark, our experiments compare the results obtained for each dataset with the specific and the minimal collections.

As done in [21], the threshold is calibrated by running the experiments with several values. We keep the best results obtained w.r.t the tested thresholds.

To make our experiments easily reproducible, all the simulation parameters are made available in our repository. We also provide notebooks allowing to run our experiment pipeline.

### 5.5 Accuracy

The accuracy of each clustering algorithm is evaluated by computing two performance metrics (namely, the parsing accuracy and the adjusted Rand index). Both require a ground truth (see Section 5.3).

The *parsing accuracy* has been introduced in [21]. More formally, given two partitions $\mathcal{C}_1, \mathcal{C}_2$ of a set $E$, the pattern accuracy $PA$ is defined by:

$$PA(\mathcal{C}_1, \mathcal{C}_2) = \frac{1}{|E|} \sum_{C \in \mathcal{C}_1 \cap \mathcal{C}_2} |C|$$

By definition, this metric only rewards clusters that *exactly* matches those listed in the ground truth. As a sequel, if a cluster is slightly different in the results and in the ground truth, this is not rewarded by $PA$; and the bigger the cluster, the bigger the penalty. This means that algorithms returning a clustering with small errors may have a very low parsing accuracy. Conversely, slight updates modifying a large cluster in the ground truth drastically change the parsing accuracy.

The adjusted Rand index [16] is designed to be less sensitive to small variations and hence alleviates all the limitations inherent to the parsing accuracy. Intuitively, it is obtained by counting the number of correct and incorrect *pairwise assignments* and is readjusted depending on the number of clusters and their respective size.

## 6 Conclusion

This companion paper shows how to reproduce the experiments presented in [18] and highlights some of its technical contributions. First, it details the code optimizations (core algorithm written in C++, parallelization) made to run our algorithm on larger logs. Second, it shows the effort made to package the code, so that it is easy to install and use for the end-user. Third, it provides all the technical material needed to reproduce our experiments, and hence allows researchers to compare their proposal against LogMine, Drain and the pattern clustering algorithm. Fourth, it has been the opportunity to enhance the ground truths provided by the Loghub dataset. For all these reasons, we hope that our module will be reused in the future works dealing with log clustering and automatic parsing.

## References

1. Boost C++ library, https://www.boost.org/
2. Drain 3 forked repository, https://github.com/raynalm/Drain3
3. Drain 3 original repository, https://github.com/IBM/Drain3
4. Ground truth templates, https://github.com/nokia/pattern-clustering/tree/main/logs
5. Loghub: A large collection of system log datasets for AI-powered log analytics, https://github.com/logpai/loghub
6. Logmine forked repository, https://github.com/raynalm/logmine
7. Logmine original repository, https://github.com/trungdq88/logmine/
8. ManyLinux GitHub repository, https://github.com/pypa/manylinux
9. Pattern clustering GitHub repository, https://github.com/nokia/pattern-clustering
10. PyBGL GitHub repository, https://github.com/nokia/pybgl
11. Zenodo repository containing the full Loghub logs, https://zenodo.org/record/3227177
12. Dijkstra, E.W.: Algol 60 translation: An algol 60 translator for the x1 and making a translator for algol 60. Stichting Mathematisch Centrum. Rekenafdeling (MR 34/61) (1961)
13. Hamooni, H., Debnath, B., Xu, J., Zhang, H., Jiang, G., Mueen, A.: Logmine: Fast pattern recognition for log analytics. In: Proceedings of the 25th ACM International on Conference on Information and Knowledge Management. pp. 1573–1582 (2016)
14. He, P., Zhu, J., Zheng, Z., Lyu, M.R.: Drain: An online log parsing approach with fixed depth tree. In: 2017 IEEE International Conference on Web Services (ICWS). pp. 33–40. IEEE (2017)
15. He, S., Zhu, J., He, P., Lyu, M.R.: Loghub: A large collection of system log datasets towards automated log analytics. arXiv preprint arXiv:2008.06448 (2020)
16. Hubert, L., Arabie, P.: Comparing partitions. Journal of classification **2**(1), 193–218 (1985)
17. Moore, E.F., et al.: Gedanken-experiments on sequential machines. Automata studies **34**, 129–153 (1956)
18. Raynal, M., Buob, M.O., Quénot, G.: A novel pattern-based edit distance for automatic log parsing. In: ICPR 2022 (2022)
19. Terrizzano, I.G., Schwarz, P.M., Roth, M., Colino, J.E.: Data wrangling: The challenging yourney from the wild to the lake. In: CIDR (2015)
20. Thompson, K.: Programming techniques: Regular expression search algorithm. Communications of the ACM **11**(6), 419–422 (1968)
21. Zhu, J., He, S., Liu, J., He, P., Xie, Q., Zheng, Z., Lyu, M.R.: Tools and benchmarks for automated log parsing. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). pp. 121–130. IEEE (2019)