BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

LOG FILE ANALYSIS AND ANOMALY DETECTION

DATA COMMUNICATIONS, COMPUTER NETWORKS AND PROTOCOLS – PROJECT

BRNO 2024 **Bc. PETER KOPRDA**

Contents

Chapter 1

Introduction

Log file analysis is a crucial process in the field of data analysis and information technology. It also involves examining log files generated by systems, applications, servers, and networks to extract valuable insights, detect anomalies, and improve security measures. Logs are records of events or actions that occur within a system. Effective log analysis involves data discovery, data preparation, data modeling, interpretation, and communication. Organizations can gain visibility into their digital infrastructure.

This project is organized into several chapters. Chapter 2 provides an introduction to the log file analysis. Chapter 3 presents description and analysis of the log dataset. Chapter 4 include how data were pre-processed, what features to select, and how to represent events. Chapter 5 describes the implemented tool its behavior and input parameters, Chapter 6 include some examples of testing process.

Chapter 2

Log file analysis

Log file analysis [4] is a basic method that offers a significant understanding of cognitive procedures and interaction with computer applications. This process includes the collection, examination, and explanation of information from log files. By employing these techniques, we can understand the behavior of the system, pinpoint problems, and spot irregularities. This chapter is designed to outline the objectives of log file analysis and the techniques that can be employed in the process, particularly in the realm of anomaly detection.

2.1 Purpose of log file analysis

The principal aim of analyzing log files [1] is to ensure effective monitoring and management of the system by reviewing log data. This investigative process helps IT administrators and teams effectively oversee and control their systems. Through the examination of log files, they can recognize patterns, spot irregularities, resolve problems, guarantee system dependability, improve security measures, and boost system efficiency. The application of log file analysis provides an understanding of how these elements affect behavior. Monitoring user behavior is relevant in two scenarios, both practical and scientific.

In a practical context [4], understanding the approach individuals take when interacting with a computer program can be highly beneficial. As computer usage becomes increasingly prevalent in daily life, human-computer interaction is gaining importance in various domains. With more people interacting with computer applications, there is a growing need to study user behavior and preferences. Analyzing user preferences and typical usage sequences can inform interface design modifications to align with user expectations.

In a scientific context [4], the way individuals interact with a computer application is of substantial relevance to social science. Numerous studies in education have explored the effectiveness of computer-assisted learning. Although computers can augment the process of learning, many students face difficulties in comprehending the software utilized. A considerable amount of research has been dedicated to understanding how these elements impact learning results, particularly in the context of discovery learning via computer simulations.

2.2 Steps and methods of log file analysis

The log-based anomaly detection involves four steps (as shown in Figure 2.1): log collection, log parsing, feature extraction and anomaly detection [3].

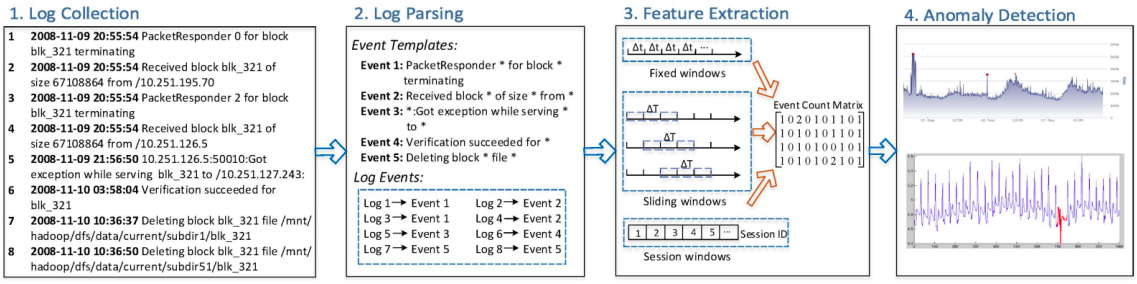

Figure 2.1: Framework of anomaly detection. Retrieved from [3].

2.2.1 Log collection

Large-scale systems generate logs as a standard practice to record system states and runtime information. Each log record typically consists of a timestamp and a message indicating the event or action that occurred. [3]

2.2.2 Log parsing

Log parsing aims to structure unstructured logs by extracting event templates from free-form text. Each log message is parsed into an event template consisting of a constant part and specific parameters. [3]

There are two types of parsing methods [3]:

1. Clustering-based method – Computing distances between logs and then use clustering techniques to group them into clusters. The event templates are then generated from each cluster.

2. Heuristic-based method – Counting the occurrences of each word at each log position. Then, frequent words are selected and combined as event candidates, from which some are chosen to be the log events.

2.2.3 Feature extraction

The next step is to encode logs into numerical feature vectors for machine learning models. This process involves slicing the raw logs into log sequences using various grouping techniques [3]:

1. Fixed window – This approach relies on a timestamp that logs the time of each occurrence. Every fixed window has its own size, indicative of the time span or duration. The size of the window is represented by a constant value – Δt.

2. Sliding window – Consists of two attributes: the size of the window and the size of the step. The step size is less than the window size, which results in the overlapping of various windows.

3. Session window – This method is based on identifiers rather than timestamps. These identifiers serve to distinguish various execution routes in certain log data.

Once we have generated log sequences using windowing, it allows us to construct an event count matrix X. Within each log sequence, we count the frequency of each log event

to create the event count vector. To illustrate, [0, 0, 2, 3, 0, 1, 0] means that event 3 took place twice, and event 4 occurred three times. [3]

2.2.4 Anomaly detection

Once the feature matrix is prepared, it can be inserted into machine learning models for training. These models are then used to generate a model specifically designed for anomaly detection. [3]

There are various methods that can be used for anomaly detection; some of them use supervised learning (machine learning task of creating a model based on training data that has been labeled), some of them unsupervised learning (machine learning task of creating a model based on training data that has not been labeled) [3]. For example:

- *Logistic regression* (supervised anomaly detection) – This technique is extensively utilized for classification. Logistic regression calculates the likelihood `p` for all potential states. When a new instance appears, the logistic function is capable of computing the likelihood `p` for all potential states. For anomaly detection, log sequences create event count vectors, forming instances with labels. Training builds a logistic regression model, applied to testing instances to determine anomalies [3].

- *Decision Tree* (supervised anomaly detection) – This method uses branching to describe the expected scenario for each instance. The decision tree is built from the top downwards, utilizing training data [3].

- *SVM* (supervised anomaly detection) – The approach of the Support Vector Machine (SVM) is to non-linearly project vectors into a feature space of very high dimensionality, with the aim of creating a linear decision boundary (hyperplane) within this feature space [5]. In the context of anomaly detection using SVM, a new instance would be identified as an anomaly if it is positioned above the hyperplane, but it would be classified as normal if it is not [3].

- *Log Clustering* (unsupervised anomaly detection) – In this method, we organize log sequences into clusters based on their similarities. By clustering logs, we can efficiently represent different categories of events within the system. At the stage of initializing the knowledge base, logs are transformed into vectors and grouped into sets of normal and abnormal events. Vectors that represent the usual behavior are identified for each cluster. In the process of online learning, new logs are incorporated, and the clusters are progressively updated. When detecting anomalies, the distances to the representative vectors are calculated, and anomalies are marked if they surpass certain thresholds [2].

- *PCA* (unsupervised anomaly detection) – Principal Component Analysis (PCA) is a statistical method that is commonly used to reduce dimensions. The fundamental concept involves mapping high-dimensional data onto a new coordinate system made up of k principal components. PCA determines the k principal components by identifying components that capture the greatest variance in the high-dimensional data. For the purpose of anomaly detection, every log sequence is transformed into an event count vector. Subsequently, PCA is utilized to discover relationships among the dimensions of these event count vectors [3].

Chapter 3

Log dataset analysis

This chapter dives deep into the examination of a chosen log dataset for this project. The dataset under consideration is derived from the ZooKeeper service[1] – a centralized platform responsible for the preservation of configuration data, naming, offering distributed synchronization, and facilitating group services. The log dataset is constructed by compiling logs from the ZooKeeper service across a network of 32 machines, spanning nearly a 27-day timeframe [6].

3.1 Dataset overview

The chosen dataset possesses a size of 9.95 MB and contains 74 380 lines [6]. Given that it lacks labels, only unsupervised learning techniques can be employed for machine learning methods. This dataset is downloaded from the Github page[2], which provides links to a variety of log datasets. The chosen log dataset consists of unprocessed log entries; however, it is possible to find structured 2000 log entries (for example for ZooKeeper service log[3]). This can give us a deeper overview of the selected data set.

```
2015-07-29 17:41:41,648 - INFO [main:QuorumPeer@913] - tickTime set to 2000
2015-07-29 17:41:41,738 - WARN [WorkerSender[myid=1]:QuorumCnxManager@368]
- Cannot open channel to 3 at election address /10.10.34.13:3888
2015-07-29 23:44:23,842 - ERROR [CommitProcessor:2:NIOServerCnxn@180]
- Unexpected Exception:
```

Listing 3.1: A sample from ZooKeeper log dataset.

A sample from ZooKeeper log dataset is shown in Listing 3.1. This sample contains three different log entries. Each log record contains several elements. The first two elements ("DateTime" and "Log level") can be easily parsed. However, in order to extract the "Node", "Component" and "Id" elements, it is required to interpret the string that is positioned between the log level element and the final hyphen ('-') in the log record. This is demonstrated in Listing 3.1 – `[main:QuorumPeer@913]` in the first log record, `[WorkerSender[myid=1]:QuorumCnxManager@368]` in the second log record, and `[CommitProcessor:2:NIOServerCnxn@180]` in the final log record. Table 3.1 shows how

[1] https://zookeeper.apache.org
[2] https://github.com/logpai/loghub
[3] https://github.com/logpai/loghub/blob/master/Zookeeper/Zookeeper_2k.log_structured.csv

it looks before parsing that element into three elements and Table 3.2 how it looks after parsing that element into three elements.

Table 3.1: Before parsing

Log elements
[main:QuorumPeer@913]
[WorkerSender[myid=1]:QuorumCnxManager@368]
[CommitProcessor:2:NIOServerCnxn@180]

Table 3.2: After parsing

Node	**Component**	**Id**
main	QuorumPeer	913
WorkerSender[myid=1]	QuorumCnxManager	368
CommitProcessor	2:NIOServerCnxn	180

3.2 Elements of log entries

Each log record has 6 different elements:

1. **DateTime** – this indicates the date and time when the log entry was recorded. The datetime follows the pattern `YYYY-mm-dd HH:MM:SS,f`, where:

   - `YYYY` represents the year in four digits,
   - `mm` represents the month in two digits,
   - `dd` represents the day of the month in two digits,
   - `HH` represents the hour in 24-hour format,
   - `MM` represents the minute in two digits,
   - `SS` represents the second in two digits, and
   - `f` represents the miliseconds in three digits.

   This format provides a precise indication of when the log entry was recorded.

2. **Log level** – this indicates the severity of the log message. The log dataset contains three categories of log messages: `INFO`, `WARN`, and `ERROR`. Each category may have a unique role in expressing various aspects of the system's functionality and potential problems.

3. **Node** – represents the specific node or entity within the system where the log message originated. This element in Table 3.1 is located between the first open square bracket ('[') and the first colon symbol (':').

4. **Component** – specifies the particular component or module of the system associated with the log message. This element in Table 3.1 is located between the first colon symbol (':') and the "at" symbol ('@').

5. **Id** – an identifier associated with the log record, which may be useful for tracking or correlating related events. This element in Table 3.1 is located between the "at" symbol and the final close square bracket (']').

6. **Content** – the rest of the log message represents the actual content of the log message, providing details about an event, action, or error (e.g. in Listing 3.1 the "Content" element for the first log record is "tickTime set to 200").

On the Github page[4], there are defined event templates for this dataset. There are 50 different event templates, each of them represents some different group of events. However, the original log file contains log records that do not correspond to these specified log events. As a result, it is necessary to eliminate the events in the log file that are not identifiable by these already established events.

[4] https://github.com/logpai/loghub/blob/master/Zookeeper/Zookeeper_2k.log_templates.csv

Chapter 4

Modeling log events

In this chapter, we set out on the task of modelling log events, a crucial step in utilizing machine learning methods for log analysis and anomaly detection. We start by clarifying the complex process of data pre-processing, feature selection, and event representation, thereby establishing the foundation for building significant models.

4.1 Data pre-processing

The initial step that needs to be taken is the pre-processing of the data. Initially, the data is presented in a log format, hence, it would be most beneficial to convert this data into a two-dimensional data structure, such as a 2D array. This method makes it straightforward to ascertain the values of each log record. For instance, Listing 4.1 provides an example of a raw log record:

```
2015-07-30 18:18:02,000 - INFO [SessionTracker:ZooKeeperServer@325] -
Expiring session 0x14ede63a5a70004, timeout of 10000ms exceeded
```

Listing 4.1: Example of unprocessed log record.

As previously pointed out in Chapter 3, the log records consist of 6 distinct elements. These records can be transformed, for instance, into a CSV format. This format provides a clearer perspective on the data. In Listing 4.2, the log elements are displayed in CSV format, separated by semicolons. The values correspond to the elements `DateTime`, `Log level`, `Node`, `Component`, `Id`, and `Content`, respectively.

```
2015-07-30 18:18:02.000; INFO; SessionTracker; Expiring session
0x14ede63a5a70004, timeout of 10000ms exceeded; 325; ZooKeeperServer
```

Listing 4.2: Example of log record in CSV format.

4.2 Event representation

Once the data is processed into CSV format, it becomes essential to rearrange these columns for the creation of log events. The best way to represent events is to use `Content` element. This element represents some type of action of the specific log record. However, it becomes imperative to cluster these messages, as identifying every single element is not the most efficient strategy, and some form of data reduction is needed. This goal can be accomplished

by utilizing regular expressions that have the ability to match a variety of specific log messages. There are 5 distinct log values for this element in Listing 4.3. A regular expression that corresponds to these values is found on line 7.

```
1 Expiring session 0x14ede63a5a70004, timeout of 10000ms exceeded
2 Expiring session 0x14ede63a5a70003, timeout of 10000ms exceeded
3 Expiring session 0x14ede63a5a70001, timeout of 10000ms exceeded
4 Expiring session 0x14ede63a5a70000, timeout of 10000ms exceeded
5 Expiring session 0x14ede63a5a70002, timeout of 10000ms exceeded
6
7 r'Expiring session .*, timeout of .*ms exceeded': 'E15'
```

Listing 4.3: Example of values for `Content` element. On the line 7 is the regular expression which would match these values and `E27` is a suitable event that corresponds to this regular expression.

This method allows us to generate events and assign them to the given regular expressions. Listing 5 demonstrates a processed log record from Listing 4.1 following the assigning of log events to regular expressions.

```
2015-07-30 18:18:02.000; INFO; SessionTracker; E15; 325; ZooKeeperServer
```

Listing 4.4: Processed log record after applying events.

4.3 Feature extraction

Once log events have been assigned, it becomes possible to derive important attributes from these events. The initial input for this process is the log events themselves, while the expected output is an event count matrix. The formulation of an event count matrix requires the categorization of events into distinct groups, each of which symbolizes a log sequence. Each log sequence contains a timestamp that records the time of occurrence for every log. Three distinct methods can be employed to formulate an event count matrix — utilizing fixed windows, a sliding window, or a session window (previously discussed in Chapter 2). Since our dataset is not large, the most suitable strategy would be to employ the "Fixed window" technique. This method can be implemented by establishing a timestamp with a duration of one day. Subsequently, the number of events for each timestamp will be calculated, resulting in the generation of an event count matrix. Equation 4.1 illustrates how the event count matrix should look. The left side of the matrix is populated with dates that serve as timestamps. It does not display the exact time, given that the selected timestamp spans a duration of one day, thus eliminating the necessity for a specific time. The top row is marked with event labels (where E1 stands for event 1, E2 for event 2, and so forth). As an illustration, on the date `2015-07-29`, the event count matrix recorded 581 instances of event 1, 218 instances of event 2, 3 instances of event 3, and 3 instances of event 4. However, on the date `2015-08-01` represented in this matrix, there were zero instances of events 1 through 4. This could potentially indicate that the distribution of these events is not even through the days. Alternatively, it could be that the initial days' events comprise a greater number of configuration log entries. This event count matrix will be appropriate for the final stage of this project, which involves the actual creation of a model for detecting anomalies.

$$
\begin{array}{c} \\ \textbf{2015-07-29} \\ \textbf{2015-07-30} \\ \textbf{2015-07-31} \\ \textbf{2015-08-01} \end{array}
\begin{array}{c}
\begin{array}{cccc} \textbf{E1} & \textbf{E2} & \textbf{E3} & \textbf{E4} \end{array} \\
\begin{pmatrix} 581 & 218 & 3 & 3 \\ 3 & 464 & 4 & 4 \\ 4 & 259 & 7 & 7 \\ 0 & 0 & 0 & 0 \end{pmatrix}
\end{array}
\tag{4.1}
$$

Chapter 5

Implementation

The implementation part of this project is divided into four main parts: parsing log files, creating log events, creating am event count matrix, and creating a model for anomaly detection.

The program has been developed as a CLI[1] application, utilizing the Python3 programming language. The implementation of the program is stored in the `log-monitor.py` file. The dependencies required for this program are listed in the file `requirements.txt`. The required dependencies can be installed by using the following command:

```
$ pip3 install -r requirements.txt
```

After installation of the required dependencies, we can just run the program. The following command can be used to run the program:

```
$ python3 [-h] log-monitor.py -training <file> -testing <file>
  [-contamination VAL] [-threshold VAL]
```

or just use bash script, attached in zip file:

```
$ log-monitor [-h] -training <file> -testing <file> [-contamination VAL]
  [-threshold VAL]
```

where:

- `[-h]` is an optional argument that displays the help message, providing information about how to use the script and its arguments,
- `-training <file>` specifies the path to the file containing training data,
- `-testing <file>` specifies the path to the file containing testing data,
- `-contamination VAL` is an optional argument that sets the contamination parameter for anomaly detection in the Isolation Forest method,
- `-threshold VAL` is an optional argument that sets the threshold for anomaly detection, anomalies with scores above this threshold are considered outliers.

[1] Command-line interface

5.1 Parsing log files

The original log file has been divided into train log file (`train.log`) and testing log file (`test.log`). The dataset has been divided by ratio 3:1 (75% logs are in train log, the rest is in test log). Therefore, the train log file consists of 56 000 lines, and the test log file comprises 18 380 lines. This I found the best approach because you can have enough data not only for training but also for testing.

In the next step, these data have to be processed in a different format because it is harder to extract information from the raw log file. The data can be transformed into a two-dimensional array, which in Python is referred to as a `Dataframe`. A dataframe can consist of various columns, with each column potentially representing a distinct value. As a result, it is an appropriate choice for this project.

5.2 Creating log events

It is feasible to determine the columns to be utilized for event specification from the constructed dataframe. The most effective method is to employ the "Content" column, as previously discussed in Chapter 4. The function `create_log_events()` in `log-monitor.py` is responsible for substituting discovered regular expression matches with corresponding events. Certain log entries contain values that do not correspond to the pre-defined events[2]. Therefore, the log entries that contain these values, are excluded from the dataframe.

5.3 Creating an event count matrix

The next step is to create an event count matrix. Initially, it is necessary to establish a size for the window timestamp. Each timestamp has a count for every event within that timestamp, resulting in one row for the ultimate event count matrix. Given that we have 50 specified events, for the sake of simplicity and manageability, we must ensure that the timestamp window is not excessively large. Therefore, a suitable size for the timestamp window could be one day. Subsequently, when we do this for every timestamp, we can connect these timestamps together and we will get an event count matrix with 50 columns and 28 rows. The creation of an event count matrix is defined in function `create_event_count_matrix()`.

5.4 Creating a model for an anomaly detection

The last step actually includes creating a model for training and testing for an anomaly detection. We got an event count matrix from the previous step, but for the better results during training process, it is better to normalize these data. For that, we can use `MinMaxScaler`[3] from the module `sklearn.preprocessing`. This estimator adjusts each attribute (column) separately within the interval [0, 1]. This method of normalization is especially beneficial when the input data does not adhere to a Gaussian distribution or when the standard deviation is extremely low, indicating that this approach is appropriate for our data.

[2] https://github.com/logpai/loghub/blob/master/Zookeeper/Zookeeper_2k.log_templates.csv
[3] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html

When it comes to model training, various strategies can be employed. The clustering technique appears to be a good fit for this kind of data, but the data points are widely dispersed. Therefore, the clusters would need to be either quite large, leading to a high risk of errors, or the cluster count must be significantly reduced, which would mean each cluster contains only one type of data. An alternative method that can be utilized is the Isolation Forest, which was indeed selected for the model's training and evaluation.

Chapter 6

Testing

The testing stage was carried out using various values for the `contamination` parameter and the `threshold` parameter. This method allows us to effortlessly obtain the values that generate the most optimal outcomes. Listing 6.1 shows the first 15 lines of the model trained to test in the test data set. For the training of the model, the contamination parameter is configured to 0.5 and the threshold parameter is established at 0.7. The output includes an anomaly score corresponding to each timestamp. The anomaly scores represent the degree of abnormality or deviation from the expected behavior for each timestamp in the dataset. If the anomaly score is 0 or close to 0, than it indicates that the data events are not considered anomalous. Take, for instance, the date 2015-08-01 in Listing 6.1, which has an anomaly score of 0. This implies that there were no anomalies on this particular date.

```
$ python3 log-monitor.py -training=train.log -testing=test.log \
-contamination=0.5 -threshold=0.7
Anomaly score of 2015-07-29 00:00:00: -0.3646350673322062
Anomaly score of 2015-07-30 00:00:00: -0.41862647568083444
Anomaly score of 2015-07-31 00:00:00: -0.3175350132431699
Anomaly score of 2015-08-01 00:00:00: 0.0
Anomaly score of 2015-08-02 00:00:00: 0.0
Anomaly score of 2015-08-03 00:00:00: 0.0
Anomaly score of 2015-08-04 00:00:00: 0.0
Anomaly score of 2015-08-05 00:00:00: 0.0
Anomaly score of 2015-08-06 00:00:00: 0.0
Anomaly score of 2015-08-07 00:00:00: -0.21294385202419108
Anomaly score of 2015-08-08 00:00:00: 0.0
Anomaly score of 2015-08-09 00:00:00: 0.0
Anomaly score of 2015-08-10 00:00:00: -0.3098874452675616
Anomaly score of 2015-08-11 00:00:00: -0.0771104426777034
Anomaly score of 2015-08-12 00:00:00: 0.0
```

Listing 6.1: First example of testing.

Example two (Listing 6.2) demonstrates an alternative method. The selected value for the parameter `contamination` was set to 0.05 and the `threshold` was set to 0.7. This instance reveals that nearly every timestamp possesses a similar value for the anomaly score. Although this isn't necessarily incorrect, it does indicate that certain lines share the same value for the floating point.

```
$ python3 log-monitor.py -training=train.log -testing=test.log \
-contamination=0.05 -threshold=0.7
Anomaly score of 2015-07-29 00:00:00: 0.014658706121825404
Anomaly score of 2015-07-30 00:00:00: -0.021816175951273498
Anomaly score of 2015-07-31 00:00:00: 0.0860513212575742
Anomaly score of 2015-08-01 00:00:00: 0.3935400025964608
Anomaly score of 2015-08-02 00:00:00: 0.3935400025964608
Anomaly score of 2015-08-03 00:00:00: 0.3935400025964608
Anomaly score of 2015-08-04 00:00:00: 0.3935400025964608
Anomaly score of 2015-08-05 00:00:00: 0.3935400025964608
Anomaly score of 2015-08-06 00:00:00: 0.3935400025964608
Anomaly score of 2015-08-07 00:00:00: 0.18664306579976153
Anomaly score of 2015-08-08 00:00:00: 0.3935400025964608
Anomaly score of 2015-08-09 00:00:00: 0.3935400025964608
Anomaly score of 2015-08-10 00:00:00: 0.10674833052529042
Anomaly score of 2015-08-11 00:00:00: 0.32958362857768214
Anomaly score of 2015-08-12 00:00:00: 0.3935400025964608
```

Listing 6.2: Second example of testing.

Chapter 7

Conclusion

In summary, the development and evaluation of an application have been finalized. The training phase effectively conditions the model, which is then applied in the testing phase. The Isolation Forest was selected for the training model. The primary shortcomings of this dataset are its limited records, making the task of training a precise model somewhat complex.

Bibliography

[1] Aivalis, C. J. Log File Analysis. In: Xiang, Z., Fuchs, M., Gretzel, U. and Höpken, W., ed. *Handbook of e-Tourism*. Cham: Springer International Publishing, 2020, p. 1–25. ISBN 978-3-030-05324-6.

[2] Egersdoerfer, C., Zhang, D. and Dai, D. ClusterLog: Clustering Logs for Effective Log-based Anomaly Detection. In: *2022 IEEE/ACM 12th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. 2022, p. 1–10.

[3] He, S., Zhu, J., He, P. and Lyu, M. R. Experience Report: System Log Analysis for Anomaly Detection. In: *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. 2016, p. 207–218.

[4] Hulshof, C. D. Log File Analysis. In: Kempf Leonard, K., ed. *Encyclopedia of Social Measurement*. New York: Elsevier, 2005, p. 577–583. ISBN 978-0-12-369398-3.

[5] Wang, H., Xiong, J., Yao, Z. et al. Research survey on support vector machine. In: *10th EAI International Conference on Mobile Multimedia Communications*. 2017, p. 95–103.

[6] Zhu, J., He, S., He, P. et al. Loghub: A large collection of system log datasets for ai-driven log analytics. In: IEEE. *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. 2023, p. 355–366.