Записки научных
семинаров ПОМИ
Том 546, 2025 г.

M. Stavtsev, S. Shershakov

NLP-BASED .NET CLR EVENT LOGS ANALYZER

Abstract. In this paper, we present a tool for analyzing .NET CLR event logs based on a novel method inspired by Natural Language Processing approach. Our research addresses the growing demand for effective monitoring and optimization of software systems through detailed event log analysis. We utilize a BERT-based architecture with an enhanced tokenization process customized to event logs. The tool, developed using Python, its libraries, and an SQLite database, allows both conducting experiments for academic purposes and efficiently solving industry-emerging tasks. Our experiments demonstrate the efficacy of our approach in compressing event sequences, detecting recurring patterns, and identifying anomalies. The trained model shows promising results, with a high accuracy rate in anomaly detection, which demonstrates the potential of NLP methods to improve the reliability and stability of software systems.
Demo video: `https://youtu.be/JLCS4F-AlYc`
GitHub: `https://github.com/ironSensei/NLP-CLR-LogAnalyzer`

§1. Introduction

Most organizations use various software systems, necessitating effective monitoring and resource allocation. Event logs, as primary artifacts of software operations, are crucial to understanding system functions and identifying optimization opportunities. Process mining combines process science and data analysis methods to extract value from such logs, which allows the optimization of processes. This project extends the approach proposed by Stepanov and Mitsyuk [8] by enhancing low-level .NET event log analysis in two ways: 1) pattern detection, to understand system interactions, and 2) anomaly detection, to identify and prevent abnormal behaviors.

We apply neural network models for automated and scalable analysis. Self-supervised learning is utilized due to large, unlabeled datasets typical in software systems. Using transformer-based NLP methods, we tokenize

---

Key words and phrases: .NET CLR, BERT, Patterns, Anomalies.
This work is supported by the Faculty of Computer Science at the HSE University.

event traces to identify patterns and anomalies. This work demonstrates the effective application of NLP techniques to .NET CLR event log analysis.

This article is organized as follows: Section 2 provides an overview of existing solutions for event log analysis, Section 3 describes the algorithms used in the project, Section 4 presents the proposed method for event log analysis, including training a machine learning model and implementing the proposed algorithms, and finally Section 5 offers a summary of the article.

§2. Related Work

In [8], the authors propose a method for extracting high-level activities from low-level event logs of program execution. To achieve this, they developed two tools: Procfiler and Ficus. Procfiler logs events that occur during the execution of programs written in the C# programming language within the .NET CLR runtime environment. Ficus acts as a parser for the event log collected by Procfiler; it removes all system information from events, leaving only activity names, and applies a predefined hierarchy to adjust the level of event abstraction. An example of such a hierarchy is shown in Figure 1. The root is an artificially added, most abstract event. For instance, *AssemblyLoader/Start_ System_ Threading* represents a very specific and detailed event. This event can be abstracted to a higher level by naming it *AssemblyLoader/Start*.

In our project, we utilized the results of the Procfiler and Ficus tools, specifically logs with the lowest level of abstraction, meaning they contain events that are the leaves of the hierarchical tree shown in Figure 1.

For the task of anomaly detection in event logs, supervised learning methods have been applied, treating this task as a binary classification problem [4, 7]. However, this significantly reduces the applicability of such methods in real systems, as it requires a prelabeled dataset, and more importantly, limits the ability to detect previously unseen anomalies. There are studies that use self-supervised learning approaches [2] based on LSTM, as well as the BERT model [6, 3], which is based on the transformer architecture and employs preliminary tokenization of the event log.

To the best of our knowledge, no previous study has investigated the applicability of NLP methods for automated pattern and anomaly detection for the domain of low-level event .NET CLR logs.

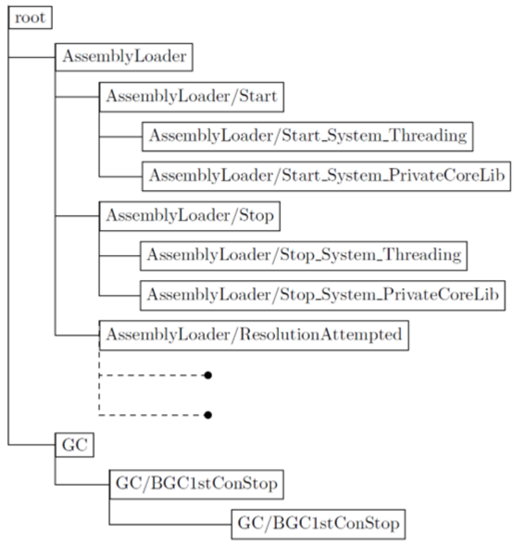

Figure 1. An example of a predefined hierarchy used to raise the abstraction level of low-level event logs, where "root" is the most abstract event and *AssemblyLoader/Start_ System_ Threading* is the most specific, detailed event.

§3. Algorithms

There are several major limitations to the rapid and efficient analysis using process mining methods. Among them are the large volume of event logs and the need for a preliminary analysis of the input data to understand the structure of interacting process elements. One of the key hypotheses in this work is that approaches adapted from the field of NLP can help overcome these limitations.

3.1. Event Log Encoding. Most NLP algorithms are designed for sequential data. For the task of event log analysis, we need to represent logs as sequences for further application of the algorithm.

Table 1 shows an example fragment of an event log, where each event has three mandatory attributes: it represents the execution of some activity from the set of activities $\mathbb{A}$ – denoted as act_i, a timestamp ts_i corresponding to the event's start time, and a process identifier indicating the process during which it was executed. The set $\mathbb{A}$ is finite and defined by the set of activities allowed in the .NET runtime environment [8].

Trace ID	Activity	Time
31237	*Method/MemoryAllocatedForJitCode*	14:27:01
31237	*Method/LoadVerbose*	14:27:50
31237	*GC/SampledObjectAllocation*	14:27:15
31237	Buffer/Returned	15:28:40
...	...	...

Table 1. Example fragment of a low-level event log from the CLR environment.

To construct a single trace from the events listed in Table 1, all events sharing the same ID should be grouped and sorted by execution time. This process produces the trace:

$$trace_1 = \langle \text{Method/MemoryAllocatedForJitCode}, \\ \text{GC/SampledObjectAllocation}, \text{Method/LoadVerbose}, \\ \text{Buffer/Returned} \rangle.$$

By using this approach, we obtain the final set of event log traces.

In order to represent traces as textual sequences, each activity from the set of all allowed activities $\mathbb{A}$ in this study is encoded with a unique non-control Unicode character, let $\mathbb{U}$ be the subset of these characters. Thus, we formed a bijection between the set $\mathbb{A}$ and the set $\mathbb{U}$, $f : \mathbb{A} \longleftrightarrow \mathbb{U}$.

For example, consider a bijective function $f_0 \subset f$, defined by the set of pairs.

$$\begin{aligned} f_0 = \{ & (\text{Method/MemoryAllocatedForJitCode}, \text{“a”}), \\ & (\text{Method/LoadVerbose}, \text{“b”}), \\ & (\text{GC/SampledObjectAllocation}, \text{“c”}), \\ & (\text{Buffer/Returned}, \text{“d”}) \} \end{aligned}$$

Then the trace $trace_1$ can be represented as the sequence $seq_1 = \text{“acbd”}$.

We reduced the task of representing a trace to a sequence of Unicode characters, solvable using NLP tokenization algorithms such as BPE, WordPiece, and Unigram. We chose the BPE algorithm because it preserves a dictionary of tokens, merging the most common pairs. After completing the tokenizer training, we obtained the final set of permitted tokens, denoted as $\mathbb{T}$, where each token represents a subsequence of Unicode characters.

Any sequence seq_k can be represented as

$$tokens_i = (t_j : t_j \in \mathbb{T}, i = 1, \ldots, \varphi(seq_k))$$

where $\varphi(seq_k)$ determines the number of tokens. The tokenization process, defined as $\mathcal{T} : seq_i \to tokens_i$, converts seq_1 into the set $tokens_1 = ['ac', 'bd']$. Tokenization also compresses traces, significantly reducing the input sequence length for the neural network.

Each token is encoded with a unique number corresponding to a value in the embedding table (numeric vectors), which in turn are trainable parameters of the neural network, the configuration of which we will describe in Section 3.2. Using numeric vectors, we can encode traces and feed them to the neural network input.

3.2. Neural Network Configuration. A key algorithm in deep learning, especially in NLP, is the transformer architecture [9], which consists of an encoder and a decoder. The encoder processes the input sequence with the attention mechanism and feed-forward layers, producing vectors that the decoder further transforms into a probability vector. BERT [1] is a transformer model that is based only on the encoder and applies attention to tokens based on their context within a sequence.

One of the ways to train BERT is the Masked Language Modeling (MLM) approach, which involves masking a certain percentage of randomly selected tokens with a special token [MASK]. During training, the model aims to minimize the loss function's error by predicting the token hidden behind the [MASK]. We claim that a model trained on unlabeled correct (without anomalies) event logs can detect events that do not match the context of normal behavior. For this reason, in this work, we apply a BERT-based model for anomaly detection.

After analyzing existing BERT-based model architectures, we selected the SqueezeBERT architecture [5] for this work. The authors of the work presenting this architecture show that the majority of the model's parameters and the bulk of the time during its application are concentrated in the feed-forward layers. They propose a convolution-based approach borrowed from the field of computer vision to optimize the neural network. As a result, they managed to reduce the number of parameters to approximately

40 million, compared to 100 million in original BERT, which requires less computational resources, without significant loss of quality.

§4. Method

Algorithm 1 Algorithm for Extracting Traces

```
function ProcessXESTraces(input_filepath)
    log ← ReadXES(input_filepath)
    event_log ← FilterLogForNeededColumns(log)
    needed_indexes ← GetNeededIndexes(event_log)
    outliers ← IdentifyOutliers(event_log, needed_indexes)
    event_log ← FilterLogByIndexes(event_log, needed_indexes)
    traces_log ← CreateTracesLog(event_log)
    final_trace_log ← IntegrateOutliersIntoTraces(traces_log, outliers)
    list_of_traces ← ConvertTracesToList(final_trace_log)
    return list_of_traces
end function
```

Algorithm 2 Algorithm for Tokenizing Traces

```
function ProcessTracesToSequences(traces, LoA)
    accepted_events ← LoadAcceptedEvents()
    event_codes ← MapEventsToCodes(accepted_events)
    sequences ← []
    for each trace in traces do
        sequence ← ConvertTraceToSequence(trace, event_codes)
        sequences.append(sequence)
    end for
    processed_traces ← TokenizeSequences(sequences, LoA)
    return processed_traces
end function
```

4.1. Patterns Detection. Tokens in the tokenizer's dictionary reflect frequently occurring interactions, thus considered patterns in this work. For instance, a group of events encoded by symbols a, b, c appearing as the token *bac* in the event trace is a pattern. We trained 13 tokenizers with dictionary sizes from 512 to 20,000 tokens, some with a maximum token length limit, to analyze these patterns at different abstraction levels – higher levels encode larger numbers of events into single tokens.

The pattern detection process involves two algorithms. Algorithm 1 describes obtaining a list of traces from raw CLR low-level event logs by extracting necessary columns and combining events by timestamps. Algorithm 2 extracts tokens from event traces at a specified abstraction level (LoA), forming a list of acceptable events and applying a mapping to

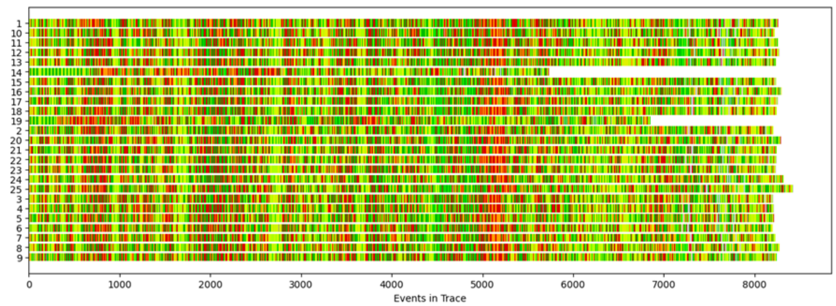

Figure 2. Non tokenized log.

create sequences. These sequences are then tokenized using the trained tokenizers, resulting in a list of tokens for each trace.

Consider an example of the results of the pattern search algorithms on event logs from 25 runs of the same C# program.

We visualized the obtained traces, showing the trace ID on the horizontal axis and the number of events (tokens) on the vertical axis, with each color representing a different token/event. Tokenization not only significantly reduced the average trace length from 8000 events in the non-tokenized log (Figure 2) to 200-300 tokens at an abstraction level of 10 (Figure 3), but also produced a list of frequently co-occurring events during program execution, which is the set of tokens of the encoded trace of program execution. This list can be further analyzed by domain experts to gain deeper insights into the execution process and potentially identify performance bottlenecks.

The method of pattern detection presented in this work is an alternative to the repeated alphabets method presented in [8]. Thus, we can say that our method is more versatile as it is used not only for pattern search, but also when using the SqueezeBERT neural network.

4.2. Anomalies Detection. Anomaly detection is based on the SqueezeBERT neural network architecture, and there are two main approaches in the NLP field for using machine learning models. The first approach involves fine-tuning an already trained model for specific tasks, which is usually optimal. However, this approach is not feasible in our case as the BERT-based model has not been previously applied to .NET CLR event

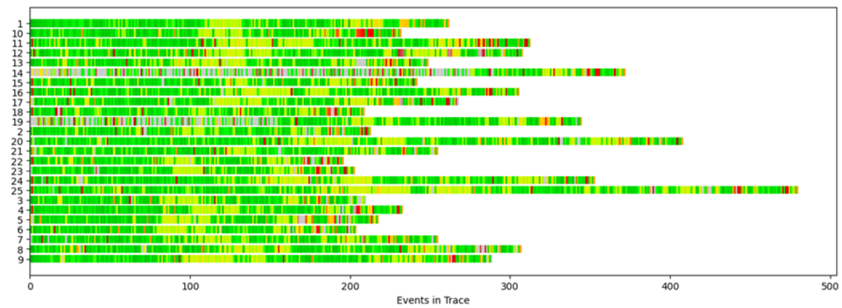

Figure 3. Tokenized log with LoA 10.

logs. Therefore, we train the model from scratch using a tokenizer with a maximum abstraction level of 13, a dictionary size of 20,000 tokens, and a maximum token length of 300 characters.

We used the LAMB optimizer [10] for training. The final model consists of 43.6 million parameters and was trained on the same dataset used for tokenizers. The context window size was set to 512 tokens, with shorter traces padded using the [PAD] token. Training was conducted for 300 epochs in the Google Colab environment on an Nvidia Tesla A100 GPU. The statistics of the learning rate, gradient norm, and loss are presented in Figure 4.

Algorithm 3 Algorithm for Anomaly Detection

```
function EvaluateTraces(traces)
    for each trace in traces do
        tokens ← ProcessTracesToSequences(trace, 13)
        (probs, loss) ← EvaluateByTokens(tokens)
        brier ← EvaluateTraceBrier(tokens)
        count_abnormal ← CountNonEmpty([probs, loss, brier])
        if count_abnormal ⩾ 2 then
            Print "Trace" + trace + "is abnormal."
        else
            Print "Trace" + trace + "is normal."
        end if
    end for
end function
```

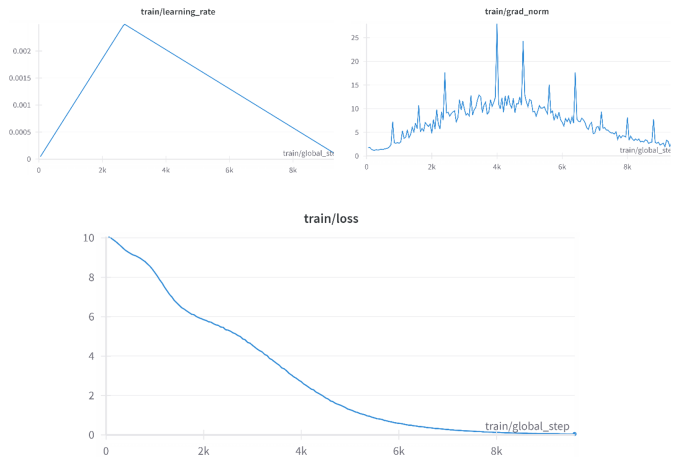

Figure 4. Dynamics of learning rate, gradient norm, and loss function during final model training.

The anomaly detection algorithm is described in Algorithm 3. It takes a list of traces as input, which are subsequently tokenized based on Algorithm 2. Then, it performs an evaluation using two methods: probability-based and loss-based, as well as Brier score evaluation.

The probability and loss evaluation is performed by masking each token in a trace with the [MASK] token, applying the SqueezeBERT model, and comparing the model's output with the observed value. If the predicted probability of the observed token is less than 0.85, it is considered anomalous. Similarly, if the loss function value exceeds 0.05, the token is considered anomalous.

The Brier score helps detect anomalies at group token levels, increasing the accuracy of detecting incorrect behavior. By masking 20% of randomly selected tokens and calculating the Brier score for them, we determine if the entire trace is anomalous if the score exceeds 0.5.

If two out of the three scores indicate anomalies in a trace, then the whole trace is considered anomalous.

We also used a SQLite database to store these three evaluations for each trace. If an identical trace appears again, we retrieve the scores from the

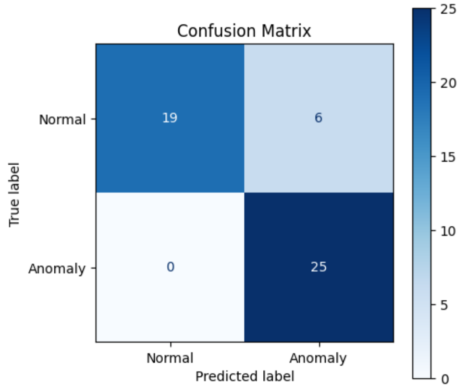

Figure 5. Confusion matrix of model performance.

database instead of rerunning the model. This approach saves time and computational resources.

Model validation was performed on the basis of 25 synthetically generated anomalous traces, as well as 25 traces with normal behavior. Anomalous traces were obtained by adding five random events at random positions, simulating the expected anomalous behavior in the trace. However, it is important to note that all 50 traces belong to the same domain as those in the training dataset, e.g., they originate from the same software execution. The validation results are presented in the confusion matrix in Figure 5.

As observed in Figure 5, the developed model shows satisfactory results; however, it makes errors in 6 cases. This error is less problematic because falsely labeling a normal trace as anomalous is preferable to missing an actual anomaly. Increasing the volume of training data could improve the model's quality, so further training on a larger dataset is recommended.

Meanwhile, testing the proposed approach on traces from another software domain yields worse results, indicating that our model is unable to generalize to other .NET CLR logs. Thus, it needs to be trained specifically for a single software system.

§5. Conclusion

In this study, existing NLP approaches applied to the analysis of event logs were examined. We developed a tool in Python, which is designed for analyzing patterns and anomalies in logs of .NET CLR applications. Our tool supports multiple levels of abstraction for pattern detection and utilizes an SQLite database to store and reference previously analyzed traces, optimizing performance. A model based on the BERT architecture, specifically SqueezeBERT, was trained from scratch. Validation results demonstrate that the model performs well in anomaly detection, and it is expected that increasing the volume of data can improve its quality. Thus, we have shown that NLP approaches can be effectively applied to the analysis of event logs in the .NET CLR runtime environment.

References

1. J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, *BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding*, ArXiv preprint arXiv:1810.04805 (2018).
2. M. Du, F. Li, G. Zheng, and V. Srikumar, *DeepLog: Anomaly Detection and Diagnosis from System Logs Through Deep Learning*, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017, pp. 1285–1298.
3. H. Guo, S. Yuan, and X. Wu, *LogBERT: Log Anomaly Detection via BERT*, in: 2021 International Joint Conference on Neural Networks (IJCNN), IEEE, 2021, pp. 1–8.
4. S. Huang, Y. Liu, C. Fung, R. He, Y. Zhao, H. Yang, and Z. Luan, *HiTANomaly: Hierarchical Transformers for Anomaly Detection in System Log.* — IEEE Transactions on Network and Service Management **17**(4) (2020), 2064–2076.
5. F. N. Iandola, A. E. Shaw, R. Krishna, and K. W. Keutzer, *SqueezeBERT: What Can Computer Vision Teach NLP About Efficient Neural Networks?*, ArXiv preprint arXiv:2006.11316 (2020).
6. Y. Lee, J. Kim, and P. Kang, *LanoBERT: System Log Anomaly Detection Based on BERT Masked Language Model.* — Applied Soft Computing **146** (2023), 110689.
7. S. Nedelkoski, J. Bogatinovski, A. Acker, J. Cardoso, and O. Kao, *Self-Attentive Classification-Based Anomaly Detection in Unstructured Logs*, in: 2020 IEEE International Conference on Data Mining (ICDM), IEEE, 2020, pp. 1196–1201.
8. E. V. Stepanov and A. A. Mitsyuk, *Extracting High-Level Activities from Low-Level Program Execution Logs.* — Automated Software Engineering **31**(2) (2024), 41.
9. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, EĹukasz Kaiser, and I. Polosukhin, *Attention Is All You Need*, in: Advances in Neural Information Processing Systems **30**, 2017.

10. Y. You, J. Li, S. Reddi, J. Hseu, S. Kumar, S. Bhojanapalli, X. Song, J. Demmel, K. Keutzer, and C.-J. Hsieh, *Large Batch Optimization for Deep Learning: Training BERT in 76 Minutes*, ArXiv preprint arXiv:1904.00962 (2019).

HSE University, Moscow, Russia
E-mail: mastavtsev@edu.hse.ru
E-mail: sshershakov@hse.ru

Поступило 28 февраля 2025 г.