NLP-Based .NET CLR Event Logs Analyzer

Maxim Stavtsev
mastavtsev@edu.hse.ru
Higher School of Economics
Moscow, Russia

Sergey Shershakov
sshershakov@hse.ru
Higher School of Economics
Moscow, Russia

Abstract

In this paper, we present a tool for analyzing .NET CLR event logs based on a novel method inspired by Natural Language Processing (NLP) approach. Our research addresses the growing need for effective monitoring and optimization of software systems through detailed event log analysis. We utilize a BERT-based architecture with enhanced tokenization process tailored to event logs. The tool, developed using Python, its libraries, and an SQLite database, allows both conducting experiments for academic purposes and solving industry-emerging tasks efficiently. 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, proving the potential of NLP methods in enhancing the reliability and stability of software systems.

Demo video: YouTube
GitHub: 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 for understanding system functions and identifying optimization opportunities. Process mining combines process science and data analysis methods to extract value from such logs, which allows one to optimize processes. This project extends the approach proposed by Stepanov and Mitsyuk [6] 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. Unsupervised learning is utilized due to the large, unlabeled datasets typical in software systems. Using transformer-based NLP methods, we tokenize 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 machine learning model training and implementation of the proposed algorithms, and finally, Section 5 offers a summary of the paper.

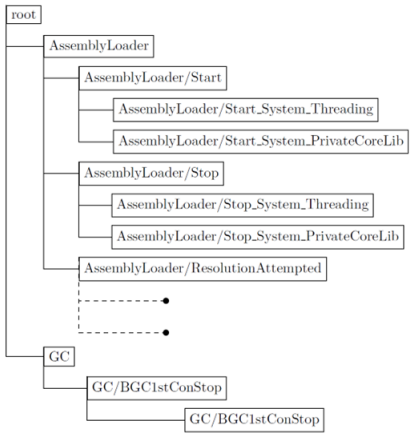

Figure 1. An example of a predefined hierarchy used to raise the abstraction level of low-level event logs, where "root" is the most abstract event and *AssemblyLoader/Start_System_Threading* is the most specific, detailed event.

2 Related Work

In [6] authors propose a method for extracting high-level activities from low-level event logs of program execution. To achieve this, they developed a tool called Procfiler, which collects events occurring during the execution of programs written in the C# programming language in the .NET CLR runtime environment and creates a log from them. They then apply a predefined hierarchy to raise the abstraction level of the events. 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 tool, specifically logs with the lowest level of abstraction, meaning they contain events that are the leaves in 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 [3]. However, this significantly reduces the applicability of such methods in real systems, as it requires a pre-labeled dataset, and more importantly, limits the ability to detect previously unseen anomalies. There are works that use unsupervised learning approaches [2] based on LSTM, as well as the BERT model [5], 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 the automated pattern and anomaly detection for the domain of low-level event .NET CLR logs.

3 Algorithms

There are some major limitations to the rapid and efficient analysis by process mining methods. Among them are the large volume of event logs and the need to perform preliminary analysis of input data to understand the structure of interacting process elements. One of the most important hypotheses in this work is that approaches adapted from the field of NLP can remove such limitations.

3.1 Event Log Encoding

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

Table 1 shows an example fragment of an event log, where each event has three mandatory attributes: it is the realization of some activity from the set of activities $\mathbb{A} - act_i$, a timestamp ts_i corresponding to the event's start time, and a process identifier during which it was executed. The set $\mathbb{A}$ is finite and defined by the set of activities allowed in the .NET runtime environment, their list is presented in the Appendix of the work [6].

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

Trace ID	Activity	Time
31237	*Method/MemoryAllocatedForJitCode*	14:27:57
-1	*Method/LoadVerbose*	14:27:57
31237	*GC/SampledObjectAllocation*	14:27:57
31237	Buffer/Returned	15:37:34
…	…	…

Considered event log contains events related to the .NET application's execution thread (e.g., ID 31237) and system threads (e.g., ID -1). These events can be combined into one trace by merging events according to their IDs and timestamps, resulting in the trace $trace_1$.

$$\begin{aligned}\text{trace\_1} = \langle &\text{Method/MemoryAllocatedForJitCode},\\ &\text{GC/SampledObjectAllocation},\\ &\text{Method/LoadVerbose},\\ &\text{Buffer/Returned}\rangle.\end{aligned}$$

By using this approach, we obtained 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 work is encoded with a unique non-control Unicode character[1], 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.

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

Then the trace $trace_1$ can be represented as the sequence $seq_1 = "acbd"$.

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 training the tokenizer, we obtained the final set of allowed tokens $\mathbb{T}$, where each token is a subsequence of Unicode characters.

Any sequence seq_i can be represented as $tokens_i = (t_j : t_j \in \mathbb{T}, i = 1, \ldots, \varphi(seq))$, where $\varphi(seq)$ determines the number of tokens. The tokenization process, defined as $\mathcal{T} : seq_i \rightarrow tokens_i$, converts seq_1 into the set $tokens_1 = ['ac', 'bd']$. Tokenization also compresses traces, reducing the input sequence length significantly.

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 [7], 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]. For this reason, in this work, we apply a BERT-based model for anomaly detection. We claim that a model trained on unlabelled correct (without anomalies) event logs can detect events that do not match the context of normal behavior.

[1] https://en.wikipedia.org/wiki/List_of_Unicode_characters

After analyzing existing BERT-based model architectures, we selected the SqueezeBERT architecture [4] 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 manage 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

4.1 Patterns Detection

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
```

Tokens in the tokenizer's dictionary reflect frequently occurring interactions, thus considered patterns in this work. For instance, a group of events 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 create sequences. These sequences are then tokenized using the trained tokenizers, resulting in a list of tokens for each trace.

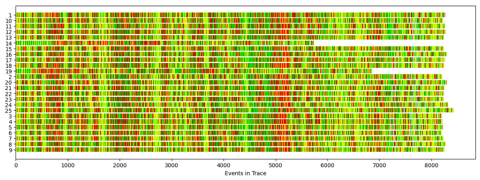

Figure 2. Non tokenized log

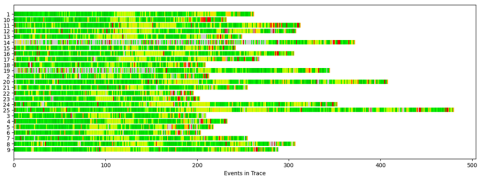

Figure 3. Tokenized log with LoA 10

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

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 reduced the average trace length significantly: from 8000 events in the non-tokenized log (Figure 2) to 200-300 tokens at an abstraction level of 10 (Figure 3).

The method of pattern detection presented in this work is an alternative to the repeated alphabets method presented in [6]. 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 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 use the LAMB [8] optimizer for training, final model has 43.6 million parameters, is trained on the same data used for tokenizers, with a context window size of 512 tokens, padding shorter traces with the [PAD] token. Training was conducted for 300 epochs in the Google Colab environment using an Nvidia Tesla A100 GPU.

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

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
```

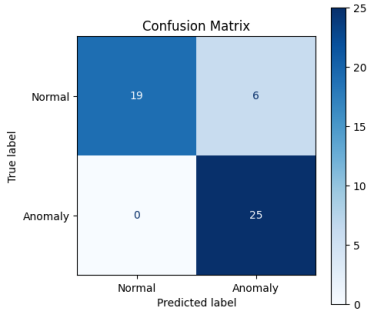

Figure 4. Confusion matrix of model performance

evaluation using two methods — probability-based and loss-based, as well as Brier score evaluation.

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 probability of the observed token is below 0.85, it is considered anomalous. Similarly, if the loss function value is below 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.

We also utilized an SQLite database to store these three evaluations for each trace. If an identical trace appears again, we retrieve the scores from the database instead of rerunning the model. This approach saves time and computational resources.

Model validation was done based on 25 synthetically generated anomalous traces, as well as 25 traces with normal behavior. Anomalous traces were obtained by adding 5 random events at random positions, simulating the expected anomalous behavior in the trace. Validation results are presented in the confusion matrix in Figure 4.

As observed in Figure 4, 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.

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] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[2] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 1285–1298.

[3] Shaohan Huang, Yi Liu, Carol Fung, Rong He, Yining Zhao, Hailong Yang, and Zhongzhi Luan. 2020. Hitanomaly: Hierarchical transformers for anomaly detection in system log. *IEEE transactions on network and service management* 17, 4 (2020), 2064–2076.

[4] Forrest N Iandola, Albert E Shaw, Ravi Krishna, and Kurt W Keutzer. 2020. SqueezeBERT: What can computer vision teach NLP about efficient neural networks? *arXiv preprint arXiv:2006.11316* (2020).

[5] Yukyung Lee, Jina Kim, and Pilsung Kang. 2023. Lanobert: System log anomaly detection based on bert masked language model. *Applied Soft Computing* 146 (2023), 110689.

[6] Evgenii V Stepanov and Alexey A Mitsyuk. 2024. Extracting high-level activities from low-level program execution logs. *Automated Software Engineering* 31, 2 (2024), 41.

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[8] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. 2019. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv preprint arXiv:1904.00962* (2019).