A novel ML-Driven Simulation Log Debugger

Narasimha Rao Chinni
SoC Dev, DVG-1
Samsung Semiconductor India Research
Bangalore, India
rao.chinni@samsung.com

Sunil Shrirangrao Kashide
SoC Dev, DVG-1
Samsung Semiconductor India Research
Bangalore, India
sunil.sk@samsung.com

Garima Srivastava
SoC Dev, DVG-1
Samsung Semiconductor India Research
Bangalore, India
s.garima@samsung.com

ABSTRACT

In SOC design verification, simulation log files serve as a critical component for understanding system behavior, finding errors, and ensuring design correctness. These logs capture a lot of information including tool messages, signal activities, assertion results, test bench warnings and error messages. The verification engineers rely on them during iterative development. However, the sheer volume and unstructured nature of these files often make manual analysis time-consuming and error-prone. Additionally, the structure, semantics, and verbosity of the simulation log files frequently differ when the same test is run across several labels of a design. It is difficult to compare log files directly.

This paper presents an interactive dashboard called *LogMiner* that solves above mentioned problems by providing an ordered way of analyzing the simulation log file and smart techniques for comparing the log file with a reference. It categorizes the messages based on the type like UVM verbosity, tool messages, VIP messages and customized messages. It segregates the messages required for SoC Debug like booting sequence, core functional messages, also a separate Register window where all register programming is captured. A dedicated user feedback window helps in collecting feedback from user and the same is fed to AI-ML engine for future enhancements.

***LogMiner* collects selected lines matching from all the available logs based on the *Sentence Transformer and Cosine Similarity.* This paper suggests using a fine-tuned ML model. It learns from many context-rich sentences so that it can convert an entire sentence—or even a whole paragraph—into a fixed-size dense vector (an embedding) that captures the meaning. It does compare current simulation log with reference model and provides the matching lines along with missing lines. This helps in identifying root cause quickly.**

I. INTRODUCTION

In complex hardware and software systems, simulation plays a vital role in validating functionality before deployment. These simulations often generate large log files containing details of the simulation that help the design verification engineers to analyze the functionality of the design. However, these logs are typically unstructured and voluminous, making manual inspection and analysis both time-consuming and error-prone.

The log file contains syntax warnings, runtime warnings, fatal errors, tool messages, UVM messages and other user written test bench messages like $display and $monitor statements. It also contains time-stamped messages helping in the debug process. It is a tedious process to traverse through the log file manually and understand the status of the flow. To address this challenge, we propose an **interactive dashboard tool** called **LogMiner**, that is developed using python plotly dash module, to display the contents in an ordered way. The dashboard also enables faster search options to make the analysis faster. LogMiner also leverages Artificial Intelligence and Machine Learning (AI/ML) techniques to assist in the efficient scrutiny of current log file. By automating the extraction of meaningful information, it significantly reduces debugging time and improves overall system observability.

Existing tools including Cogita [4], Splunk [5], ELK [6] are more comprehensive in features and more generic in nature, whereas LogMiner is specifically tailored to fulfill our particular requirements. LogMiner allows regulating preprocessing and deducing the logs to reference and analyzing its sequential flow which helps to detect corner cases and makes it stand different from existing tools.

This paper discusses the design and features of the dashboard, highlights key AI/ML strategy employed, and presents use cases demonstrating its effectiveness in analyzing the log file. The reference generation and comparison process are explained with an example. The log files contain user written messages which might vary from engineer to engineer, so instead of direct comparison of text, the Sentence Transformers architecture is used for a better content-wise comparison.

II. PROBLEM DEFINITION

The simulation log files generated during testing processes are often large, unstructured and difficult to interpret manually. It is difficult and time-consuming to analyze these log files to identify errors, detect anomalies, or extract relevant information. The lack of standardized formats and the presence of verbose, low-level messages further complicate the task, making it tedious for engineers and analysts to draw actionable insights efficiently. In addition to that, the simulation log file of a same test on two different label or version of a design often vary which means same sentence can be written in different phrases. The order of the important messages can be changed based on the new features getting added later. This makes the manual comparison and analysis of the errors and tool messages tedious. This paper addresses these problems by providing an interactive dashboard to analyze errors, warnings and other testbench messages. The dashboard also facilitates the user to generate a reference from the log file and to compare other log files with the reference. The goal is to reduce manual effort, improve accuracy, and facilitate faster decision-making during simulation-based workflows.

III. LITERATURE SURVEY

Vaswani et al., [1] showed a model built entirely on attention mechanisms could beat RNNs by capturing whole-sentence relationships with multi-head self-attention. This breakthrough not only made training much quicker but also laid the groundwork for the massive pre-trained models that have reshaped NLP research and the applications that rely on it.

TheFuzz [2] is a python package which can used for line comparison and it is based on Levenshtein Algorithm. It can determine if two sentences are identical based on how many adjustments are needed to match them, but it is unable to comprehend contextual significance and semantic meaning.

Reimers and Gurevych [3] proposed SBERT (Sentence-BERT), a Siamese version of BERT that generates fixed-size sentence embeddings for efficient semantic comparison tests. SBERT's fine-tuning on NLI and STS datasets allows for quick cosine-similarity-based phrase comparison while preserving high semantic accuracy. Their findings demonstrated significant speedups over cross-encoders, making SBERT suitable for large-scale retrieval and clustering applications. This work created the groundwork for modern sentence embedding models, which are now widely used in NLP.

IV. IMPLEMENTATION– LOGMINER

To address the pain points mentioned in problem definition, we propose a tool called, **LogMiner** that is based on AI-ML Algorithms. This tool has got two features, Reference generator and Log Comparator, which are based on ML technique **Sentence Transformers [7], Cosine Similarity [3]**. The Reference generator can create the reference file of required messages. The Comparator is used to compare the current simulation log messages against golden

reference that model has been trained on. Comparator is based on the ML technique Sentence-Transformers, a fine-tuned ML model utilizing available packages on **Hugging-Face [8]** and Cosine Similarity. Thus LogMiner aids in the root cause analysis and reduces the Turn Around Time in debug. It also segregates different kinds of messages in different tabs, which gives user specific messages quickly by avoiding grep/search techniques. messages. **This is proprietary tool of Samsung semiconductor and does not contribute to Accellera in any means.**

SYSTEM DIAGRAM

Sentence-Transformers is an extended version of **BERT** architecture. BERT is a Google-developed language model which uses the Transformer architecture to understand the meaning and context of words in text. *Sentence Transformer's* main purpose is to handle tasks that demand a deep, contextual understanding of the words in a sentence. It is commonly used for tasks like data deduplication, matching user inputs, and comparing text with minor differences by providing a similarity score (Cosine Similarity). Cosine Similarity is used to find the matching lines based on the threshold. Following Figure-1 describe the high level diagram of proposed solution.

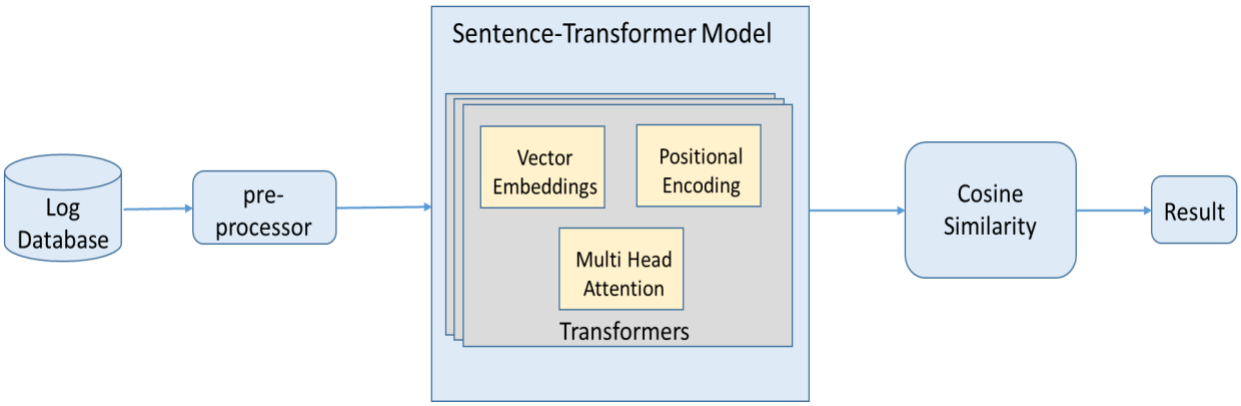

Figure-1: High level system diagram

The log files are **pre-processed** by removing unwanted messages, tool warnings, timestamps etc. analogous to reference. It also finds abnormal behavior like hung, infinite loops, time outs. The pre-processed log is passed through **Sentence-Transformer model** which create vector embeddings that convert text tokens into dense semantic representations for model input, positional encoding adds information about token order, enabling Transformers to understand sentence structure. Multi-head attention lets the model focus on different contextual relationships simultaneously, capturing richer dependencies across the sentence.

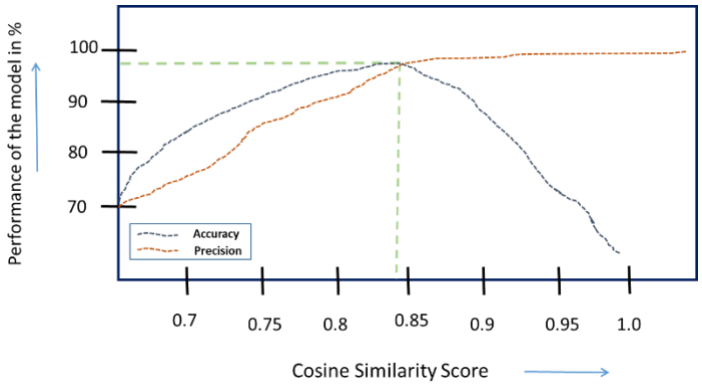

Figure 2: Threshold Value Calculation Cosine Similarity

The model gives an overall score for a particular sequence of sentences which can be compared with reference sequence using **Cosine Similarity** with an ideal threshold value. Ideal threshold value can be selected based on the precision and accuracy of the model in detecting the similar and dissimilar sentence pairs. This Cosine Similarity will help us to decide if two sequence of sentences are semantically similar which is referred as Result.

We employ the Sentence Transformers base model "all-MiniLM-L6-v2" as the foundational embedding model and fine-tune it on our SoC-specific log lines to better capture domain-dependent semantics. The fine-tuning process uses a contrastive loss function, which trains the model to pull semantically similar log messages closer in the embedding space while pushing dissimilar ones farther apart. This approach enables the model to learn subtle contextual patterns unique to our hardware logs, resulting in more accurate similarity detection and downstream analytics.

Figure-2 shows the plot of precision and accuracy of model with Cosine Similarity score. Precision is the percentage of the pairs predicted as "similar" that were actually correct. Accuracy is the percentage of actual "similar" pairs that the model has successfully found. We found Accuracy and Precision intersection at peak when Threshold value is 0.846 which means by selecting the optimum Threshold value, the model achieved the best balance, correctly identifying 98% of the similar pairs without letting in too many false positives.

LogMiner DASHBOARD

The Figure-3 shows the layout of the dashboard. The organized layout and segregation of the log file contents provide a better way for the users to navigate through the contents and extract required information. The buttons and the search keys are highly customizable as per the requirement of the verification team or the design specification. LogMiner has 2 important sections. In section-2 on the right consists horizontal tab where the contents of the log file are segregated accordingly into UVM messages, display messages, warnings, errors, tool alerts and VIP messages. The first tab displays the summary and the UVM_MESSAGES tab displays the main tests sequence messages. ERRORS tab displays the simulation failure messages. There is GLS tab that display GLS parameters, like setup-hold time violation etc.

Figure 3: LogMiner – Interactive web dashboard

Section 3 on the left contains the logo and the **special analysis** buttons. The Register analyzer prints the data of the registers programming that has been performed. All the boot messages and CORE simulation messages are segregated

as well. **LogMining** button has special functionality where a user can compare the loaded log against the golden reference model.

V. RESULTS

In this section we discuss the testing strategy and the results. To begin with, we consider a few lines of a log file and show a comparison between passing and failing logs. Next, we present a plot for order detection of a sequence of log lines. Finally, we describe the regression suite and testing methods, and compare the debugging time of the traditional approach with our solution.

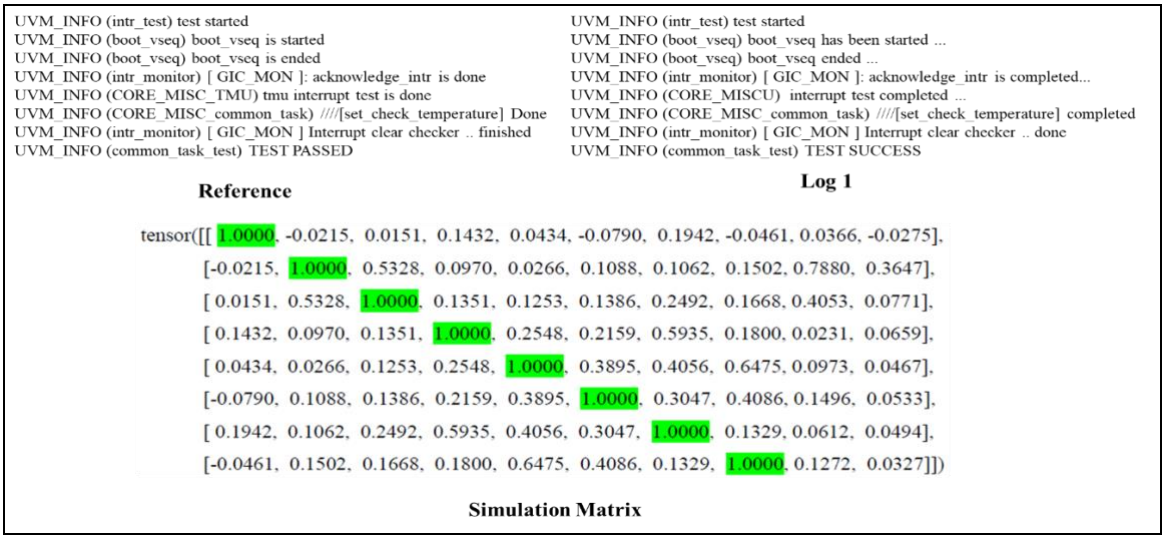

Figure 4: Exact Matching Log

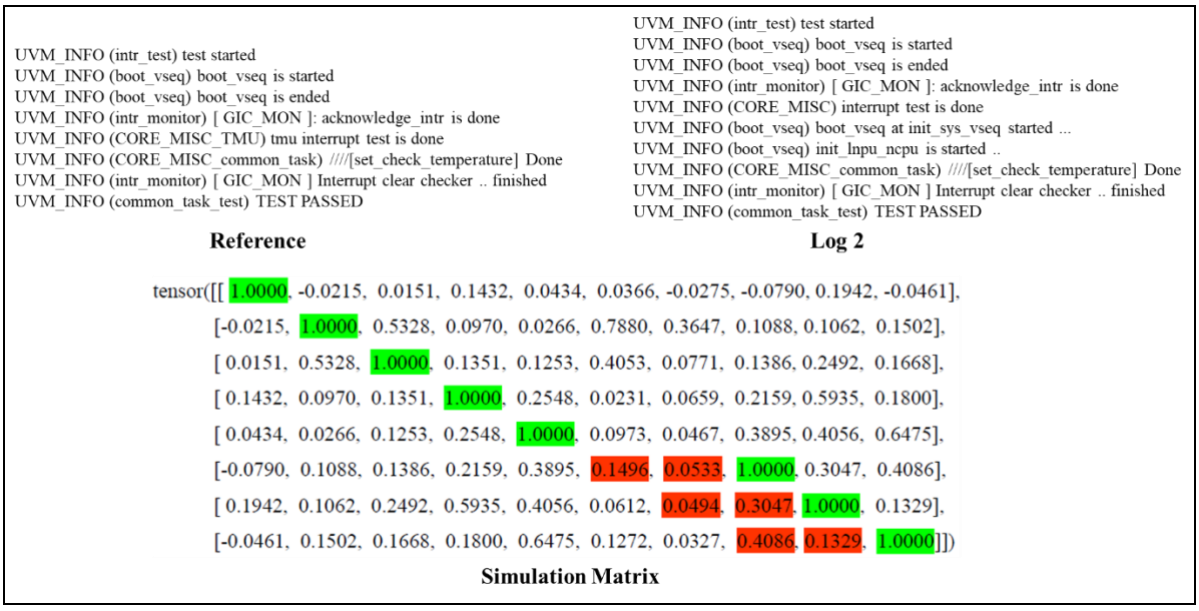

Figure 5: Mismatching Log

Figure-4 and Figure-5 illustrate a comparative analysis between the deduced passing logs and failing logs using a reference-based tensor (simulation matrix). This tensor encodes the pairwise similarity scores between log sentences,

enabling the model to determine both semantic equivalence and adherence to the expected sequence. By interpreting this matrix, the algorithm identifies any deviations in content or ordering; such mismatches are flagged as failure indicators, allowing the system to reliably predict whether a log sequence should be classified as pass or fail."

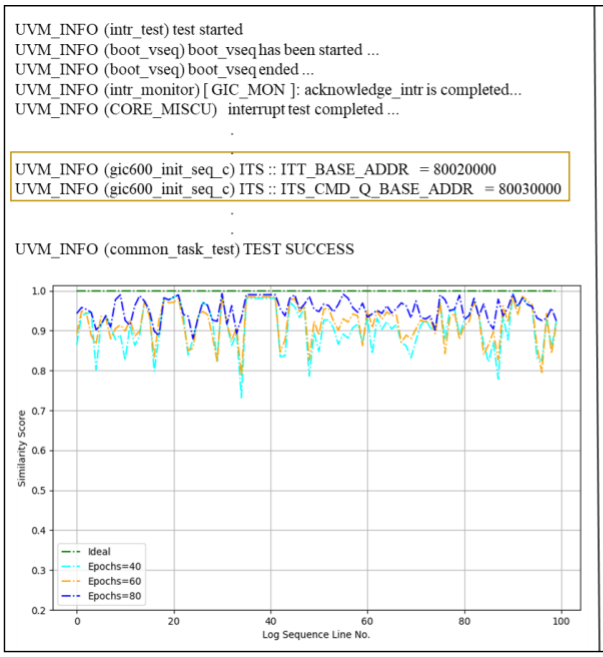

Figure 6: Correct Log Order Detection

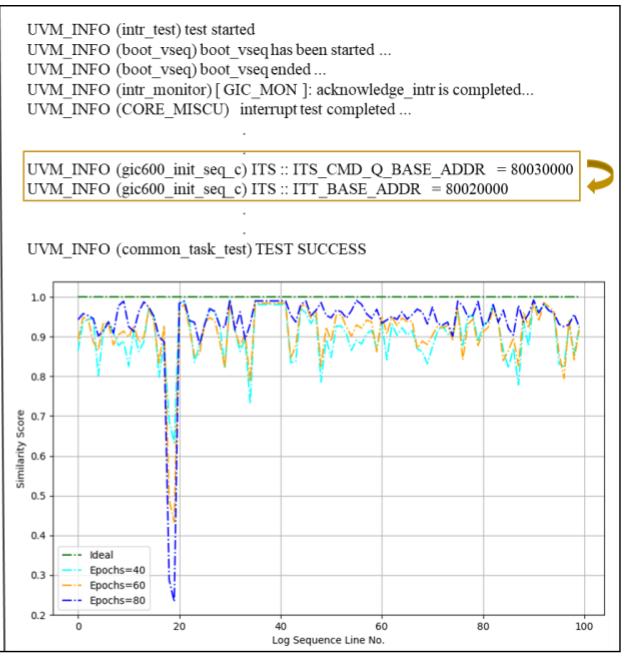

Figure 7: Wrong Log Order Detection

Figure-6 and Figure-7 show the order-detection plots for a log containing 100 lines when the model is evaluated without using the reference log sequence for comparison. As the number of epochs increases, the model's order-detection score approaches 1 (ideal value) for a correctly ordered sequence and approaches 0 for an incorrectly ordered sequence. Figure-6 displays the plot for a correctly ordered sequence. Figure-7 shows the plot for a sequence in which two lines are shuffled, the similarity score for those shuffled lines drops to near 0, indicating that the sequence order does not match the training data.

Now we describe the preliminary results of our solution produced using a regression suite. A regression suite of 5000 test cases is considered for testing. To measure the performance of the LogMiner, **four types of tests** are considered in a SoC consisting of multiple blocks (pcie, usb, power management, GNSS etc.) with booting time of 8ms and test specific run time of 2ms. First test is a simple Register configuration test that configure the required registers of a block. In this test, chip booting is not required, but chip reset and initialization is done. Second one is a Core test in which block / feature specific operation is simulated. Third test does a complex block operation like back to back traffic, multi-tasking etc. The last test does multi booting by the application of Hard reset and power up-down cycle. In terms of number of lines in a simulation log, on an average, SFR test consists of 10K lines, while simple core consists of 100K lines, complex core test can have around 500K lines, multi-boot core can have 1M lines. These lines are exclusive of messages from simulation tool. Also the Log is pre-processed and only required UVM messages are filtered out based on UVM Message ID.

The dashboard **load time** depends on the size of simulation in terms of number of lines and size occupied on disk as well. Pre-processor will help avoid unexpected log sizes due to simulation hung, UVM_TIMEOUT, infinite

loops/wait statements etc. For a simulation of 10K lines, Dashboard gets opened in a fraction of second. While the longest and gracefully ended simluation that consists of 100K lines, load time is around 5 seconds. The LogMiner TAT mentioned below sums up loading, pre-processing and comparison time.

Below Table-1 shows the **TAT** for finding the actual difference in failing and passing log for above mentioned 4 types of tests. We have calculated the TAT of traditional debug method with in a team of 20 engineers with various level of experience ranging from 2 to 15 years, over last 2 projects. Regression suite of 5000 tests are categorized into 4 as explained above. Debug times from all users are collected and averaged as per the traditional method. In LogMiner method, TAT is calculated by a python script.

Sl No.	Test	Traditional Method		LogMiner	
		Log size (lines)	TAT	Log size (lines)	TAT
1.	A basic SFR test	10K	5 min	248	1 min
2.	Simple CORE test	100K	30 min	2.7k	5 min
3.	Complex CORE test	500K	1.5 hrs	13.5k	8 min
4.	Complex CORE Multi Boot test	1M	3 hrs	24.3k	12 min

Table 1: TAT comparison

VI. APPLICATIONS

Our solution can be customizable to SoC, IP, Subsystem level projects and other software testing logs as well. The main applications are

- Helps in Root cause analysis.
- Works as **Waveless** debug provided log message are fair enough.
- Smart comparator: Used to perform quick comparison of failed logs with a reference log created instead of traditional text editors which is difficult to compare with whole lot of information.

VII. FUTURE SCOPE

With the promising features mentioned above, ***LogMiner*** finds its application in many simulation platforms. It can be extended to post-silicon and emulation where ever Log analysis plays a role as primary debug. Modern systems generate logs from many services, containers, or edge devices where LogMiner can be used.

- Enhance to identify root cause deep in the design by providing structured messages those can be linked to find root cause.
- Extending the model to analyze different kind of files
- Enhance AI/ML techniques to suggest better reference models

The current scope of this tool is to identify the first deviation point of simulation. This does not point out the root cause. As a future enhancement, we are working on enhancements to identify root cause in which the first source of failure is identified.

VIII. CONCLUSIONS

LogMiner eases the comparison of log files with reference model and reduces the Turn Around Time for a complete comparison which can be a main deciding difference for the failure. It is based on Sentence Transformers ML technique which is popular in analyzing vast data in SoC regressions. Cosine Similarity is used to measure similarity between log and reference. It segregates different kinds of messages in different tabs, which gives user specific messages quickly rather than using "grep" techniques to get the required messages. This solution is useful at various level of verification from IP to SoC.

IX. REFERENCES

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin, "Attention Is All You Need", 31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA

[2] L. Smith, "Fuzzy String Matching Procedure," Open Bioinformatics Journal, vol. 13, pp. 50-62, 2020.

[3] Reimers and Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks", Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing, pages 3982–3992, Hong Kong, China, November 3–7, 2019. c 2019 Association for Computational Linguistics

[4] Cogita. "https://thevtool.com/vtool-eda"

[5] Splunk. "https://www.splunk.com/"

[6] Elasticsearch, Logstash, and Kibana. "https://www.elastic.co/elastic-stack"

[7] Maria Teresa Colangelo, Marco Meleti, Stefano Guizzardi, Elena Calciolari and Carlo Galli, "A Comparative Analysis of Sentence Transformer Models for Automated Journal Recommendation Using PubMed Metadata"

[8] "Sentence Transformers." Hugging Face, https://huggingface.co/sentence-transformer