

MASTER'S THESIS 2026

Hierarchical Self-Supervised Learning for Semantic Embeddings of Crash Dumps

Danny Tang, Ludvig Lindholm

Elektroteknik
Datateknik

ISSN 1650-2884

LU-CS-EX: 2026-28

DEPARTMENT OF COMPUTER SCIENCE

LTH | LUND UNIVERSITY



EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2026-28

**Hierarchical Self-Supervised Learning for
Semantic Embeddings of Crash Dumps**

Hierarkisk självövervakad inläring för
semantisk vektorisering av kraschdumpar

Danny Tang, Ludvig Lindholm

Hierarchical Self-Supervised Learning for Semantic Embeddings of Crash Dumps

Danny Tang
da2628ta-s@student.lu.se

Ludvig Lindholm
lu2166li-s@student.lu.se

June 16, 2026

Master's thesis work carried out at Schneider Electric Buildings.

Supervisor: Görel Hedin, Gorel.Hedin@cs.lth.se;
Emil Eriksson, emil.eriksson@se.com;
Jörgen Malmberg, jorgen.malmberg@se.com

Examiner: Niklas Fors, niklas.fors@cs.lth.se

Abstract

Analyzing software crash dumps is important for diagnosing system failures, but manually grouping them by root cause is time-consuming and error-prone. At Schneider Electric, crash dumps are currently identified using file content hashing, which leaves most crashes unclassified, as any change yields a distinct crash ID.

This thesis proposes a machine-learning-based embedding system that maps an entire crash dump into a single semantic embedding. Ideally, this embedding system automatically groups all crash dumps originating from the same underlying defect by assigning them similar embeddings.

Since only a small fraction of the crash dumps are labeled with underlying defects, we opted for primarily using self-supervised training to maximize the amount of available data to train on. Instead of directly training the model to connect a crash dump to a defect, we train it to position crash dumps within a learned embedding space.

To handle large crash dumps, a hierarchical encoder-based embedding model is introduced. Crash dumps are processed in smaller chunks, which are then combined into a single vector using learned, adaptive weighting. A novel attention-based loss function is introduced to improve training using the learned adaptive weights, together with a multi-phase training curriculum to reduce overfitting of the attention weights.

Our purely self-supervised model significantly outperforms a baseline pre-trained ModernBERT model, nearly halving the False Positive Rate at a 95% retrieval threshold, which shows that the model is feasible to use as a retrieval tool for clustering. Due to extreme ground truth data scarcity, our supervised model is trained on the entire testing dataset and results are thus not directly comparable. However, the supervised model outperforms the self-supervised model on manual retrieval testing in real-world use cases.

Acknowledgements

We would like to thank our supervisors from Schneider Electric for their guidance. We would also like to thank the Gryphons team for providing us with an Nvidia RTX 5090 so we could train our models locally. Finally, we express our gratitude to Görel Hedin, our supervisor from the department, for valuable advice and insights when writing this thesis.

Contents

1	Introduction	9
1.1	Proposed Solution	10
1.2	Challenges	10
1.3	Contribution	10
1.4	Usage of LLMs	11
1.5	Project Disposition	11
2	Background	13
2.1	Crash Dumps	13
2.1.1	Problem	13
2.2	Artificial Neural Networks	14
2.2.1	Training ANNs	15
2.3	Computing Embeddings	16
2.3.1	Embedding	16
2.3.2	Tokenization	17
2.3.3	Word2Vec	18
2.3.4	Encoder models	18
2.3.5	ModernBERT	20
2.4	Training Methods Used	21
2.4.1	Objective and Loss Functions	21
2.4.2	Hard Example Mining	22
2.4.3	Avoiding Representation Collapse	23
2.4.4	Multiple Instance Learning	23
2.4.5	Attention Pooling	24
3	Related Work	25
3.1	Log Analysis and Anomaly Detection	25
3.2	Processing Large Documents	26
3.3	Contrastive Learning in NLP	26
3.4	Summary	26

4	Solution	29
4.1	Models	31
4.2	Analyzing Labeled Crash Dumps	32
4.3	Complete Dataset	34
4.3.1	File filter	34
4.3.2	File Distribution	35
4.4	Preprocessing	37
4.4.1	Masking	37
4.4.2	Drain3	37
4.4.3	Custom Tokenization	37
4.5	Pretraining ModernBERT	39
4.6	Training Pipeline	40
4.6.1	Training	40
4.6.2	Validation	40
4.6.3	Memory	41
4.7	Model Architecture	41
4.7.1	Hyperparameters	44
4.8	Loss Function	44
4.8.1	Margin Loss Function	45
4.8.2	Asymmetric Attention-Conditioned Margin	46
4.8.3	VICReg	49
4.8.4	Hard Example Mining	49
4.8.5	SupCon	50
4.9	Detailed description of the models	51
4.9.1	<i>Baseline</i> Model	51
4.9.2	<i>Sym</i> Model	51
4.9.3	<i>SymAsym</i> Model	51
4.9.4	<i>SA-HEM-SA-SupCon</i> Model	52
4.10	Iterative Architectural Evolution	52
5	Evaluation Framework	55
5.1	Test Defect Pairs	55
5.1.1	Data Contamination	55
5.1.2	Pair-Wise Comparison	56
5.2	Retrieval task	56
5.3	Evaluation challenges	56
6	Quantitative Evaluation	59
6.1	Quantitative Evaluation Metrics	59
6.1.1	Evaluation Metrics	59
6.1.2	Dynamic Threshold Metrics	60
6.2	Quantitative Results and Discussion	61
6.3	<i>SA-HEM-SA-SupCon</i> Quantitative Results	64

7	Distribution Evaluation	67
7.1	Distribution Evaluation Metrics	67
7.1.1	Mean/Median Gap	67
7.1.2	Wasserstein Distance	68
7.1.3	Overlap Coefficient (OVL)	68
7.2	Distribution Results and Discussion	68
7.2.1	Similarity Distributions	69
7.3	<i>SA-HEM-SA-SupCon</i> Distribution Results	71
8	Manual Evaluation	73
8.1	Manual Evaluation Methods	73
8.1.1	Visual Inspection via K-Nearest Neighbors:	73
8.1.2	OpenSearch	73
8.1.3	Web Interface	74
8.2	Manual Inspection Results	74
8.2.1	Manual Inspection of <i>SA-HEM-SA-SupCon</i> model	74
9	Conclusion	77
9.1	Summary of Contributions	77
9.2	Key Findings and Technical Insights	78
9.3	Limitations	78
9.4	Practical Implications for Schneider Electric	79
9.5	Future Work	79
9.5.1	Model Architecture and Training Dynamics	80
9.5.2	Data Generation and Supervised Fine-Tuning	80
9.5.3	Generalization to Other Datasets	81
A	Experiments	87
A.1	Background	87
A.1.1	Cross Entropy Loss	87
A.2	Bag Cohesiveness Classification	87
A.2.1	Task Formulation	88
A.2.2	Implementation Challenges	89
A.2.3	Problems	89
A.2.4	Lessons Learned	90
B	Hyperparameters	91
B.1	<i>SymAsym</i>	91
B.2	<i>SA-HEM-SA-SupCon</i>	92
C	Tables of Figures	97
D	Website	101

Chapter 1

Introduction

All software projects carry a risk of failure, at which point an error log, such as a stacktrace, is often created. These error logs usually contain the root cause of the failure. Analyzing these error logs is not always straightforward, and additional contextual information may be required to identify the underlying cause. Therefore, it is common for additional relevant information to be collected at the time of the crash such as memory usage, storage usage, and system logs.

Schneider Electric Buildings in Lund develops and supports globally deployed products. When encountering software failures, products generate crash dumps, which consist of a stacktrace and additional relevant information such as memory usage, storage usage and system logs. Crash dumps are critical for diagnosing and preventing future software failures. However, using them efficiently requires grouping crash dumps originating from the same root cause. Currently, this clustering process is highly dependent on manual effort from engineering teams, making it very time-consuming and labor-intensive.

Schneider Electric currently uses a clustering algorithm. However, that algorithm is very rudimentary and simply hashes the stacktrace after some small preprocessing, thus grouping only effectively identical crashes. If the crash dump does not contain a stacktrace the system defaults to generating a unique ID.

To address this inefficiency, the primary goal of this thesis is to help Schneider Electric implement an automated retrieval and clustering system. Given a query crash dump, the system should return a list of historical crash dumps that could potentially be related. To achieve this, our solution approach involves computing representations of the text so that similar crash dumps are positioned close together in a vector space (an embedding space), while unrelated crash dumps are pushed further away. The combination of improved retrieval and clustering will likely be very useful in the manual clustering workflow.

1.1 Proposed Solution

Our proposed solution is to use Machine Learning (ML) to create vectorized representations of each crash dump, where the vector has semantic meaning such that similar crash dumps receive high similarity scores, while dissimilar ones receive low scores.

To achieve this, we train a ModernBERT model [23] from scratch using the crash dump data as our corpus. ModernBERT is an improvement on the original BERT [6] model architecture. A key advantage of ModernBERT is its support for an 8,192-token sequence length (compared to BERT’s 512). We theorize that this extended capacity will significantly improve our results on large files by reducing data fragmentation and minimizing the risk of lost context.

We then fine-tune the model using a self-supervised contrastive learning approach. Instead of relying on explicit labels or bag-level classifications, the model is trained with a triplet-loss-inspired objective function. Our hypothesis is that this will encourage the model to pull embeddings of related crash dumps closer together while pushing unrelated ones apart, directly learning the underlying structure of the data.

1.2 Challenges

Most existing machine learning research on crash dumps is focused on anomaly detection, for example identifying whether a specific system state is normal or something malicious [7]. Anomaly detection is a binary classification task, rather than using machine learning for the retrieval of similar crash dumps. Crash dumps are noisy, which makes it difficult to group them. Pretrained embedding models are trained on natural language or code, not on error logs. This thesis will investigate the extent to which a specialized embedding model that has been fine-tuned strictly on crash dumps improves clustering performance.

An additional difficulty in using an embedding model on error logs is that they are often very large and may require context from the entire file. Using an entire file for insights is uncommon as embedding models are typically used to retrieve smaller text chunks from larger documents.

Since we lack a comprehensive labeled dataset of historical crash dumps manually grouped by their underlying defects, we cannot use traditional supervised learning to train the model. Instead, we will investigate how self-supervised and contrastive learning techniques can be adapted to train embedding models on large crash dumps. A key focus has been designing an objective function that captures the underlying semantic meaning and prevents representation collapse. Representation collapse occurs when a model simply maps all crash dumps to the exact same point in the embedding space to minimize loss.

1.3 Contribution

The work presented in this thesis was a fully collaborative effort, with both authors sharing an equal workload across the project’s lifespan. Both authors participated in the overarching design, implementation, and evaluation of the entire machine learning pipeline. The proposed solution was a collaborative effort, and the codebase was largely developed through pair

programming to ensure continuous discussion, problem-solving and mutual understanding. Some later components were more specialized, with Danny doing a larger part of the website, while Ludvig did most of the work on the Transformer experiments and the SupCon and HEM implementations.

The writing of the thesis was divided. Rather than splitting the report by entirely separate chapters, the authors distributed the workload by subsections. For example, the theoretical part in the Background chapter and the methodology in the Solution chapter were drafted by assigning specific subsections to each author. Following the initial drafting, both authors continuously reviewed, edited, and refined each other's written contributions.

1.4 Usage of LLMs

Large Language Models (LLMs) were utilized during the writing of this thesis to assist with grammar and spelling corrections. Additionally, they were used to generate the underlying TikZ code for several figures. We also utilized LLMs when writing code to decrease implementation time and allow for more iterations.

1.5 Project Disposition

The remainder of this thesis is structured as follows:

- **Chapter 2: Background** provides the theoretical foundation and industrial context for the project. It introduces Schneider Electric's crash dump architecture followed by the technical concepts used in the solution, including Artificial Neural Networks and transformer architectures (BERT and ModernBERT).
- **Chapter 4: Solution** details the methodology and implementation of the proposed machine learning pipeline. It covers the data preprocessing and masking strategies required to clean the logs, the design of the objective function, and the complete training procedure and model architecture.
- **Chapter 5: Evaluation Approach** outlines the challenges of evaluating unsupervised clustering and explains the methodology for pairwise comparison using test defect pairs.
- **Chapter 6: Quantitative Evaluation** describes the threshold-based retrieval metrics used and presents the quantitative performance of our proposed models.
- **Chapter 7: Distribution Evaluation** details the distribution metrics, such as Wasserstein distance, and provides an analysis of how well the models separate related and unrelated crash dumps.
- **Chapter 8: Manual Evaluation** covers the qualitative assessment of the models using visual inspection, a custom web interface, and expert feedback.

- **Chapter 9: Conclusion** summarizes the findings of the thesis, discusses the practical implications for automating issue clustering at Schneider Electric, and suggests potential avenues for future work and optimization.

Chapter 2

Background

This chapter presents the theoretical foundation for the thesis. It begins by outlining the industrial context at Schneider Electric, including an overview of crash dump data.

Following this, the chapter introduces the technical basis for the proposed solution. It covers fundamental concepts of Artificial Neural Networks (ANNs) and Natural Language Processing (NLP), with a specific focus on transformer architectures such as BERT and ModernBERT. This section also explores more specific components, such as the loss functions utilized in this research, regularization methods, and various machine learning evaluation metrics.

2.1 Crash Dumps

Schneider Electric has an internal Crash Dump Analyzer (CDA). When a device in testing or from a participating customer crashes, a crash dump is created and uploaded to the CDA. Each crash has metadata such as a crash ID, upload time, and in a few cases, associated defects. Additionally, each crash has its own set of crash files. The files consist of system information, memory information, operating system details, and crash logs. These files are the primary way to differentiate between crashes.

Crash dumps commonly, but not always, contain a stacktrace and system information, including storage utilization, memory utilization, package versions, and setup logs. Crash dumps from different projects within Schneider Electric may contain varying sets of files. Some of the logs are rather large (≈ 1 MB), posing additional difficulties when used for training. Crash dumps are often repetitive and verbose, which requires special handling.

2.1.1 Problem

Crash dumps created by Schneider Electric's products are currently identified using file content hashing (or by a random unique ID), which leaves most crashes unclassified, as any tex-

tual change in a file yields a distinct crash ID.

As most crash dumps are assigned a unique ID, the clustering work must be done almost entirely manually, which is extremely time-consuming and error-prone.

The current operational workflow relies on engineering teams monitoring incoming crashes, manually noting crashes that appear to occur multiple times. When a crash ID crosses a frequency threshold, it is marked for further investigation. For crashes using a random unique ID, this frequency threshold will never be passed. Some crashes using deterministic hash-based IDs may also never surpass this threshold due to minor variances in the hashed files.

During investigation, the team manually inspects the crash dump files, prioritizing the highly informative files such as the stacktrace. From this, they attempt to find the likely root cause so that they can manually match it with existing, known Jira tickets. If they do not know of a likely ticket, additional searches are performed using exact text matches in OpenSearch (Section 8.1.2) or by sampling cases from the same timeframe.

If a manual search finds no existing ticket that is deemed related enough, a new Jira ticket is created. As this is fully manual, there is a large risk of creating duplicated tickets. Duplicating tickets could, for example, lead to the team missing code regressions or spending unnecessary time on re-implementations.

Currently, as noted earlier, many crashes are also fully ignored because it is infeasible to manually label hundreds of crashes per day, as each takes a considerable amount of time.

2.2 Artificial Neural Networks

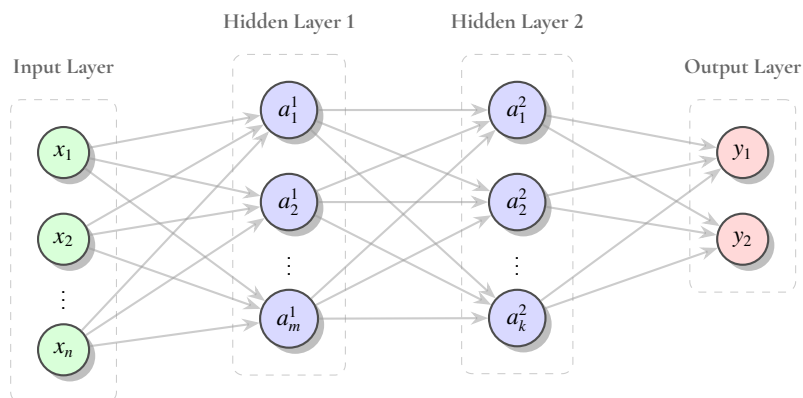


Figure 2.1: Architecture of a standard Artificial Neural Network.

Since our solution is built on a neural network encoder, we will first review the basics of Artificial Neural Networks (ANNs). ANNs are computational models inspired by biological neural networks and are commonly used for problems that are difficult to manually model precisely due to their complexity.

A standard ANN is composed of multiple interconnected layers, each consisting of multiple neurons, as shown in Figure 2.1.

This architecture is typically divided into three distinct types of layers: the input layer, one or more hidden layers, and an output layer.

In Figure 2.1, $\mathbf{x} = (x_1, \dots, x_n)$ is a vector of initial input values fed into the network. As the data is processed through the respective hidden layer neurons and passed to the next (a_i^l , where l is a layer index and i is an intra-layer index), non-linear calculations are performed. Finally, the output layer produces the network's prediction (y_1, y_2) , commonly denoted as $\hat{\mathbf{y}}$.

Detail of a Single Hidden Neuron

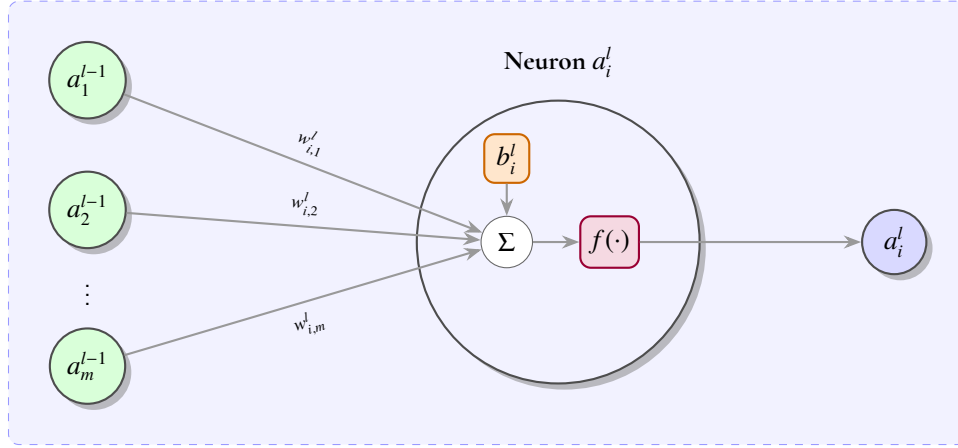


Figure 2.2: Computational detail of a generic hidden neuron a_i^l , illustrating how the activation function is applied internally to the weighted sum of inputs from the preceding layer. a_i^l signifies the result of the operation.

At a lower level, as detailed in Figure 2.2, each individual neuron computes a weighted sum of its inputs. The connections between neurons carry learned weights, denoted as $w_{i,j}^l$ (representing the weight from the j -th neuron in the previous layer to the i -th neuron in the current layer), and each neuron typically incorporates a bias term, b_i^l . At the start of training, both weights and biases are generally assigned random values. To enable the network to learn, non-linear mappings, the resulting scalar sum is passed through a non-linear activation function $f(\cdot)$, which propagates the output to the subsequent layer.

For example, calculating the value for a_i^l can be done using the following equation:

$$a_i^l = f\left(\sum_{j=1}^m (w_{i,j}^l \cdot a_j^{l-1}) + b_i^l\right)$$

Here, m denotes the number of neurons in the preceding layer, $w_{i,j}^l$ represents the weight between the j -th neuron of the previous layer and the i -th neuron of the current layer, and b_i^l is the bias for the node whose value we are currently calculating (a_i^l).

2.2.1 Training ANNs

Training a neural network is the process of iteratively tuning the weights and biases with the goal of minimizing a loss function. The loss function is a mathematical representation of the model's error, measuring the difference between its prediction and the optimal result.

To minimize this loss, the network relies on an algorithm called backpropagation. During backpropagation, the network computes the gradient of the loss function with respect to

every single weight and bias in the model. By using an optimization algorithm to update the parameters in the opposite direction of this gradient, the network iteratively reduces its error and learns the task.

In supervised learning, where the model is provided with labeled data, common loss functions include Mean Squared Error (MSE), which directly measures the difference between the ground truth label and the model's prediction.

Conversely, in unsupervised learning, ground truth labels are not provided. The goal is instead to force the model to learn the underlying structure or distribution of data. Unsupervised learning is an umbrella term which covers all learning methods where ground truth labels do not exist, such as in self-supervised learning where labels are created from the data itself.

In the context of this thesis, we utilize self-supervised learning due to our lack of a large labeled dataset where crash dumps are mapped to defects. Thus, pure supervised learning is not feasible and we must instead learn the semantic structure of logs by creating labels from our dataset itself.

To ensure a model performs reliably on new information rather than simply memorizing training examples (overfitting), the dataset is commonly split into three parts:

- **Training set:** Used to train the model and update its weights.
- **Validation set:** Used to tune hyperparameters and see if the model overfits during training.
- **Testing set:** Saved until the end to provide an unbiased evaluation of unseen data.

While the network automatically learns weights and biases during training, hyperparameters are settings which are often configured manually before training and not modified during backpropagation. Hyperparameters control settings regarding how the model learns and the structure of the model. Examples include learning rate (the step size used when optimizing weights), batch size (the number of training samples processed before weights are updated), and the number of layers in the model.

2.3 Computing Embeddings

This section explains what embeddings are, how models that generate them function, and how they have evolved over time. It will explain tokenization, attention and models such as Word2Vec, BERT, and ModernBERT which utilize these fundamental Natural Language Processing (NLP) techniques.

2.3.1 Embedding

This thesis aims to create a single semantically accurate embedding of each crash dump. In standard NLP the goal is often to calculate an embedding per token (subword) or per chunk of text. More generally, an embedding is a way to represent an object as a vector of numbers, where the location of those vectors in an n -dimensional latent space carries semantic meaning.

This attribute of embeddings being locations in a latent space is fundamental for this thesis as our model would ideally position similar crash dumps close together in the latent space and dissimilar ones far apart.

2.3.2 Tokenization

In this thesis we will train a custom tokenizer because the vocabulary of Schneider Electric's crash dumps is dissimilar to natural language. Tokenization is the fundamental preprocessing step in Natural Language Processing (NLP) where raw text is segmented into smaller, discrete units called tokens. These units, which can represent words, characters, or subwords, serve as the basic building blocks for the model. Once segmented, these tokens are mapped to unique integer values within a predefined vocabulary. This effectively converts human-readable strings into a numerical format that machine learning models can process.

Our implementation uses Byte Pair Encoding (BPE), which is a widely used tokenization strategy and is used in many BERT-based architectures.

BPE efficiently creates a vocabulary for any input corpus by starting with a baseline vocabulary of single characters. It then iteratively scans the data, identifies two consecutive symbols that appear together most frequently, and concatenates them to form a new token. This process continues until a predefined vocabulary size is reached. As a result, BPE allows models to represent frequent words as a single token, while breaking down rare, complex, or unseen words into smaller subwords. This method effectively eliminates the risk of unknown words for reasonable vocabulary sizes.

Tokens

A token is the smallest possible unit known by the model. Depending on the strategy employed, a token may represent an entire word, a subword unit (such as a prefix or suffix), or an individual character. By utilizing subword units, modern models can represent any text while keeping the total number of unique identifiers manageable. In natural language common words are assigned a token. For crash dumps, which do not consist of normal words, using a standard vocabulary is suboptimal as each segment will require many tokens to represent it.

Vocabulary

The vocabulary is the finite set of unique tokens that a machine learning model is capable of recognizing. The size of the vocabulary is an important hyperparameter that balances the efficiency of the model with how well the model represents the language. A vocabulary that is too small forces the model to fragment words into smaller subwords or rely on out-of-vocabulary tokens (UNK) for unrecognized words. On the other hand, a large vocabulary increases the size of the model's embedding layer, leading to higher memory consumption.

Normalization

Normalization is the initial preprocessing step applied to raw text before it is divided into tokens. Its primary purpose is to clean and standardize the input data, which helps reduce

the overall size of the vocabulary. Common normalization includes standardizing letter case, removing extra whitespace, and standardizing Unicode characters.

A smaller vocabulary generally decreases the required model size and complexity, which can help reduce the training time. An example of normalization in the context of crash dumps is to convert paths from Linux and Windows to look the same, or to remove uninformative timestamps.

Tokenization metrics

The effectiveness of a tokenization strategy is evaluated through metrics such as out-of-vocabulary rate, coverage, and fertility.

- **Out-of-Vocabulary (OOV) Rate:** The frequency at which a tokenizer encounters words entirely absent from the predefined vocabulary. A high *OOV* rate degrades model performance by forcing the use of generic unknown tokens (UNK). *OOV* should generally be minimal.
- **Coverage:** The percentage of a given text that the tokenizer can successfully represent without resorting to unknown tokens. High word-level coverage indicates an efficient vocabulary and it is calculated as $1 - \text{OOV}$. Coverage should be very close to 1.0 (100%).
- **Fertility:** The average number of tokens produced per word. A fertility rate of 1.0 means 1 token per word [19].

2.3.3 Word2Vec

Word2Vec [13] introduced the concept of representing words as fixed-length vectors. In these models, words that appear in similar contexts are placed closely together because they share semantic features. A defining characteristic of these models is their ability to perform vector operations to capture different semantic relationships: for example,

$$\vec{v}(\text{king}) - \vec{v}(\text{man}) + \vec{v}(\text{woman}) \approx \vec{v}(\text{queen})$$

results in a vector that is closest to the representation of the word "queen" [13].

2.3.4 Encoder models

Attention

Attention is the mechanism that enables the BERT model to process every token in a sequence simultaneously, weighing the relevance of all other tokens to adjust the embedding of each word based on its specific context.

Figure 2.3 illustrates this concept using a simplified sequence which could occur in a crash dump. It demonstrates how a single token uses context from surrounding tokens based on their relevance.

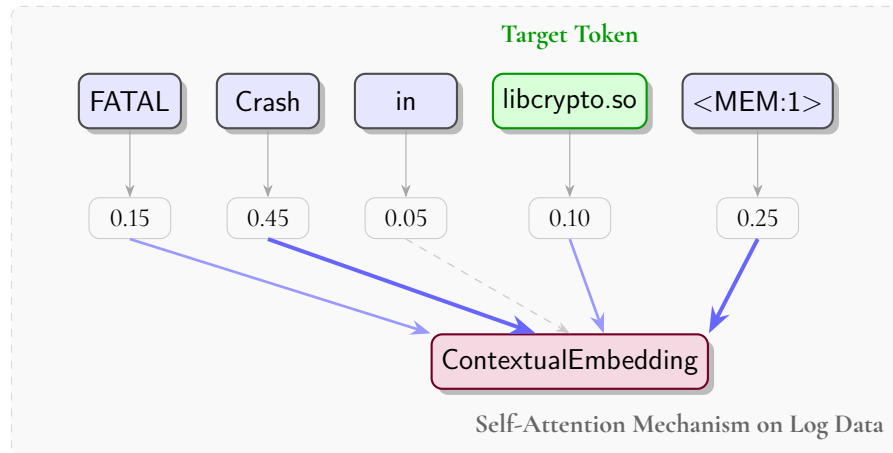


Figure 2.3: A conceptual illustration of the self-attention mechanism applied to a crash dump sequence. To generate the contextualized embedding for the target module token `libcrypto.so`, the model assigns varying attention weights to all surrounding tokens. High relevance tokens, such as the failure event type (`Crash`) and the masked memory address (`<MEM:1>`), receive higher weights, linking the module directly to the specific failure context.

The primary goal of this bidirectional self-attention is to allow each token to attend to the entire sequence, capturing dependencies regardless of their distance [22]. An architecture built primarily upon these attention mechanisms is called a Transformer.

Bidirectional Encoder

The drawback of early text embeddings such as Word2Vec is that their vector representations are static. Static embeddings mean that a word receives the same embedding regardless of its surrounding context. For example, the word "bank" would have an identical vector representation whether it refers to a "river bank" or a "financial bank".

To overcome this limitation, the field of natural language processing shifted toward contextual embeddings, meaning representations that dynamically adjusted based on the entire sentence. Addressing this exact need, the Bidirectional Encoder Representations from Transformers (BERT) [6] architecture was created in 2018. It aimed to create a good base model that could capture the surrounding context around words and be fine-tuned to solve many different tasks.

Unlike contemporary language models (LMs), BERT was pretrained bidirectionally by conditioning on both earlier and later tokens in all layers. BERT is an encoder-only model, which means that it in a single pass, generates a contextualized representation of a given text chunk. It generates an embedding for each token in the given text chunk and additionally prepends a `[CLS]` token embedding, which is the embedding for the entire chunk. To allow bidirectional training BERT introduced a new objective function called Masked Language Modeling (MLM).

MLM works by randomly hiding (masking) specific tokens in an input sequence and requiring the model to predict them based on the surrounding unmasked text using bidirectional self-attention. With the attention mechanism that captures the bidirectional context,

the model builds a vector representation of the hidden token. A linear layer then converts this vector into raw numerical scores for every word in the vocabulary, which a softmax function then normalizes into a probability distribution to predict the missing word. The loss is then calculated so that the predicted word matches the actual masked word. Simplified, MLM is the textbook task of attempting to predict what word has been removed in a sentence.

As a result of BERT being bidirectionally pretrained so that each token can attend to both past and future tokens and thus use them as context, BERT learned very effective language representations. It could thus be fine-tuned by only adding one new layer. This led to it obtaining new state-of-the-art (SOTA) results on eleven different tasks, such as improving the General Language Understanding Evaluation (GLUE) SOTA results to 80.5% which represented a 7.7 percentage point absolute improvement [6].

2.3.5 ModernBERT

Encoding large files such as those from crash dumps requires splitting each file into chunks. When chunks are small, the risk of breaking up semantically important passages increases and aggregation becomes more difficult. ModernBERT, released in 2024, is a significant advancement over the original BERT architecture. It extends the supported context window to 8,192 tokens and was pretrained on a corpus of 2 trillion tokens [23].

Like its predecessor (BERT), ModernBERT is an encoder-only model, but it incorporates numerous architectural optimizations designed to improve memory utilization and accelerate inference on GPUs.

The ModernBERT authors made several modifications to streamline the network structure. ModernBERT removes bias terms across the normalization, attention, and feedforward (MLP) layers. It also replaces traditional absolute positional embeddings with Rotary Positional Embeddings (RoPE) and implements updated normalization techniques.

One of the most important efficiency improvements is the introduction of alternating global and local attention layers. Standard global attention, where every token attends to every other token in the sequence, is highly computationally expensive and scales quadratically ($O(n^2)$) with context size. To mitigate this, ModernBERT restricts global attention to every third layer. The intermediate layers utilize local attention, meaning each token only attends to others within a restricted sliding window [23]. This maintains contextual awareness while drastically reducing computational overhead compared to the original BERT, which relied exclusively on global attention.

Managing Large Context

While ModernBERT's alternating attention layers reduce the computational and memory complexity, embedding large chunks, such as those from crash dumps, is still very expensive. Because all chunks from all files must be encoded simultaneously, storing the intermediate activations required to calculate gradients during backpropagation creates a massive memory bottleneck on the GPU.

To address this, both ModernBERT and our implementation utilize Flash Attention 2 [4]. It mitigates the memory bottleneck by optimizing read and write operations on the GPU [5] as well as optimizing the work partitioning [4].

2.4 Training Methods Used

2.4.1 Objective and Loss Functions

The objective function is the overall function that the model attempts to minimize during training. An objective function is often composed of multiple loss functions and regularization terms. Loss functions convert the difference between the optimal outcome and the model's prediction into a numerical value that can be used to give the model feedback and guide its learning via backpropagation.

There are different loss functions for different tasks. This section will briefly describe the loss functions utilized in this project. Regularization terms are mathematical constraints which are added to the objective function to prevent the model from learning shortcuts or suffering from representation collapse.

In this thesis, we utilize VICReg as a regularization term and triplet margin loss as a loss function for our basic model.

Contrastive Loss

Contrastive loss is a distance-based loss function. Unlike a traditional classification loss function that compares the model's output on a single example with the optimal result, contrastive loss operates on pairs of samples. The objective is to learn an embedding space where similar samples are mapped close to one another, while dissimilar samples are pushed apart, which aligns well with our crash dump positioning goal. The broader family of loss functions which are used in this way is called metric learning.

This thesis will not use a standard contrastive loss formulation but instead modify it so that it works with no ground truth labels.

Triplet Margin Loss

Triplet margin loss is an extension of contrastive loss where three samples are evaluated at the same time rather than a single pair. A triplet consists of an anchor sample A , a positive sample P , and a negative sample N . The goal of the triplet loss is to ensure that the anchor is closer to the positive sample than it is to the negative sample by at least a predefined margin γ .

Figure 2.4 shows how triplet loss functions in both the informative and uninformative case. An informative case occurs when the model encounters a hard negative (N_{hard}). This gives a nonzero loss as $d(A, N_{hard}) < d(A, P) + \gamma$. However, if we instead have N_{easy} we would get a zero loss as $d(A, N_{easy}) > d(A, P) + \gamma$ which teaches the model nothing. In metric learning, the function $d(A, N)$ returns the distance between A and N . In this thesis we use cosine distance as our distance function.

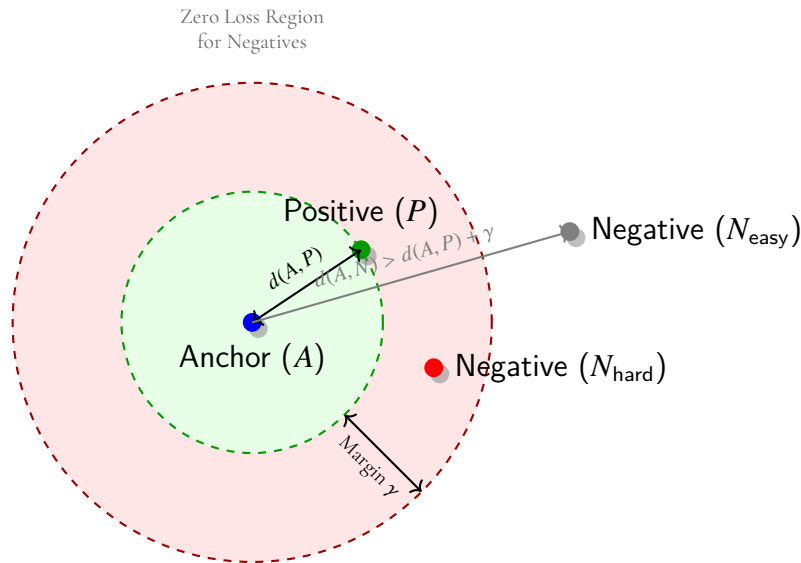


Figure 2.4: A geometric visualization of Triplet Margin Loss. The model is penalized (red zone) if the Negative sample falls within the margin γ of the Positive sample’s distance from the Anchor. Negatives outside this zone yield a loss of zero.

This approach was used in facial recognition tasks in the FaceNet model [20]. In FaceNet, an anchor was an image of a specific person, and the positive sample was another picture of that same person, and the negative sample was a picture of a different person. In this thesis, we modify the standard triplet margin loss so that it works without labeled crash dumps.

SupCon

Supervised Contrastive Learning (SupCon) is a fully supervised method that improves on earlier contrastive losses such as triplet and margin loss. Its goal is to maximize the information gain per labeled sample, which is very helpful for this thesis as we have access to very little labeled data.

In summary, SupCon works by pulling together embeddings of the same class instances while repelling embeddings of other classes. SupCon was shown to be more robust to corruptions in images and capable of handling smaller datasets better [11]. These losses are calculated within the same batch, which likely results in information-dense loss gradients per training step.

In our setting, each distinct defect can be seen as a separate class. Crash dumps that share a defect are therefore positive instances of that class, and SupCon pulls their embeddings together while pushing away crash dumps from other defects.

2.4.2 Hard Example Mining

A problem with triplet margin loss is that most triplets are likely to be uninformative as the samples are likely to be randomly positioned in the latent space, and thus far enough from the anchor that zero loss is given. This risks greatly increasing the convergence time as computing them has no impact on the model’s training. This problem is mentioned in

FaceNet where they propose two Hard Example Mining solutions: either generating triplets offline at some interval or selecting difficult triplets from a batch [20].

In general, Hard Example Mining (HEM) is a strategy in machine learning in which the training dataset is sampled to increase information density. This is useful in the triplet case mentioned earlier but also in the related case when the model performs well overall but has poor performance on a subset of the dataset. A large reason for poor performance on subsets is that a vast majority of naively created triplets will be trivial.

There are multiple methods of HEM: for example, Online Hard Example Mining (OHEM) [21]. In simplified terms, OHEM works by only performing backpropagation on the hardest (commonly measured by loss) examples in a batch or iteration. This is meant to focus the gradient updates on the most informative samples.

There are also static HEM schemes that often work by creating an index of the dataset using a checkpoint version of the model. That index is then used to retrieve hard examples to train on. Issues with using a static index are that the embeddings in it will drift from the embeddings generated by the model, potentially yielding less useful examples over time. This is however easier to implement and removes the computational cost of running inference to update the index at set intervals and is thus very efficient as retrieval operations using embeddings are highly efficient. In this thesis we will use static HEM in our final supervised model.

2.4.3 Avoiding Representation Collapse

When training embedding models, especially in self-supervised or weakly supervised settings, a common source of failure is representation collapse. This occurs when the model learns to output a nearly identical vector for all inputs to minimize the distance-based loss functions, effectively failing to learn any meaningful semantic information.

VICREG

VICREG (Variance-Invariance-Covariance Regularization) addresses this issue by applying three regularization terms to the embeddings that are compared [2].

- **Variance:** Variance is a term that maintains a minimum level of variance across a batch, making it impossible for the model to collapse into a single point.
- **Invariance:** A term that ensures that inputs that should be similar (like log chunks from the same crash) are mapped close together in embedding space.
- **Covariance:** Prevents different variables within a single embedding vector from copying each other, forcing dimensions to be independent. As a result, the model is pushed to learn a wider variety of distinct features.

2.4.4 Multiple Instance Learning

Multiple Instance Learning (MIL) is a variation of supervised learning where data is arranged in sets (bags). In traditional supervised learning, each instance is labeled. However, in multi-

ple instance learning, bags of instances are assigned labels. MIL models are trained to output a single result relating to the entire bag [10].

Under standard MIL assumptions, a bag is labeled positive if it contains at least one positive instance, and negative if all instances are negative. MIL has been utilized in medical imaging, for example when detecting cancer in whole-slide tissue images. These images are commonly extremely high-resolution which makes processing and training models very computationally costly. Another reason for the use of MIL in the medical field is that most labeled training data only contains whole-slide labels where the entire slide is either labeled "cancerous" or "benign", instead of having more fine-grained pixel-level annotations.

As a mitigation tactic, MIL splits the image into multiple parts (instances) and the set of instances in an image is called a bag. Each of these instances can then be processed in parallel and their results are aggregated.

The solution proposed in this thesis is inspired by this medical application of MIL. Just as whole-slide tissue images typically only have a single slide-level label, our crash dumps (which contain multiple distinct log files) are also evaluated under a single bag-level label. Additionally, even if fine-grained supervised training were possible, individual log files often lack sufficient context to be accurately and consistently positioned within a latent space on their own.

2.4.5 Attention Pooling

To classify a bag in a MIL architecture, information from individual instances must be collected into one single representation. Typical pooling mechanisms, such as mean pooling or max pooling, are often used. They treat instances as equally important or rely on the most extreme value [10].

Attention pooling addresses this by computing a weighted average of instance embeddings, where weights are learned by neural networks. This mechanism allows the model to focus on the most informative instances, while ignoring irrelevant ones. In this thesis, we use two levels of trainable attention pooling layers. The first layer learns which specific chunks are most informative within a single file, and the second learns which files are most important to the overall crash dump.

Chapter 3

Related Work

This chapter explores related work. We will start by exploring existing log analysis and log anomaly detection models. We will then briefly discuss more general work regarding processing large documents. Lastly, we will discuss Supervised Contrastive learning and its limitations for our use-case.

3.1 Log Analysis and Anomaly Detection

The domain of automated log analysis has been researched extensively, with a primary focus on anomaly detection. The goal of anomaly detection is to identify deviations from normal behavior to flag potential malicious activities, system breaches, or failures in real-time.

Several anomaly detection systems, such as LogBERT [7] and LogFormer [8], create and utilize embeddings within their architectures. Because these models generate intermediate embeddings, they are highly relevant to our work, even though they were not explicitly purpose-built for retrieval and clustering tasks.

LogBERT utilizes a BERT-based encoding model to assign embeddings to log sequences, optimizing the latent space so that anomalies are mapped far away from the tightly grouped normal logs. Similarly, LogFormer uses a Sentence-BERT [16] encoder to create log sequence embeddings, which are then fed into a classifier to categorize the sequence as normal or anomalous.

A critical limitation of these approaches for our use case, however, is that they work at the log sequence level, relying on a constant, limited number of lines. Consequently, they are not directly transferable to our domain, which requires embedding entire crash dumps containing a dynamic and vastly larger number of lines split over multiple files. An additional limitation is that these models do not an architecture which is suited for using multiple log files per crash to position them accurately in a latent space.

While the fundamental encoding logic is shared, our use case requires an architecture capable of handling significantly larger contexts and aggregating information across multiple

files.

3.2 Processing Large Documents

While recent advancements in NLP have introduced sparse-attention models, such as ModernBERT [23] and Longformer [3], to mitigate the quadratic complexity of standard attention, these models are still primarily designed for continuous documents and can still not handle millions of tokens which can be needed for crash dumps. Additionally, they do not natively support hierarchical structures such as crash dumps, where a crash dump consists of a dynamic set of files, whose individual embeddings should be combined to create an aggregated representation for the entire crash dump.

To improve scalability, research has been conducted on hierarchical encoding architectures. An example of a hierarchical structure can be seen in Attention-based Deep Multiple Instance Learning [10], where the authors introduce attention pooling as a method of aggregating information from patches of an image to then generate a single prediction for the entire image, instead of per patch. An obvious limitation of this method for our use-case is that it focuses strictly on images. Instead, this thesis aims to explore whether a similar framework can be used on groups of documents where each file can be seen as a patch.

3.3 Contrastive Learning in NLP

One of the most important research areas in ML is in finding loss functions which maximize information gain per sample. One such framework is Supervised Contrastive Learning (SupCon) [11] which is especially powerful for learning accurate embeddings as it efficiently utilizes labels to pull together samples of the same class while pushing apart others. Before SupCon a common method was triplet loss which was popularized by FaceNet [20]. However, an issue with triplet loss was that it only compares single samples, not all in a batch. Using all samples in a batch, as SupCon does, improves the amount of information that can be learned from each batch while adding negligible overhead.

A fundamental problem with using SupCon for our use-case is the need for labeled data which is extremely scarce for our use-case. This requirement for vast labeled data makes it infeasible to train using SupCon only.

Instead, we implement a pipeline where we start training self-supervised using our custom triplet loss and training pipelines, but then finetune our model using SupCon and our small labeled dataset to improve the global structure of the latent space while keeping learned knowledge from earlier stages in the training and the collective information from all 300,000 crash dumps.

3.4 Summary

While anomaly detection models handle log syntax, hierarchical models manage length, and contrastive learning efficiently learns embeddings, to the best of our knowledge, there is

currently no existing framework that combines these to cluster crash dumps which consist of multiple log files in a self-supervised manner.

Chapter 4

Solution

This chapter presents the proposed solution for clustering crash dumps with extremely scarce ground truth data. The complete pipeline can be seen in Figure 4.2. The process begins with data preprocessing to clean and standardize noise by masking information such as dates and memory addresses. Following this, we explain how we trained a ModernBERT model from scratch and utilized a MIL-inspired, triplet-loss-based, adaptive attention-conditioned margin in our objective function, see Section 4.8.1. The objective function was one of the most challenging parts of this project as we lacked enough labels to safely use only a supervised loss function. Instead, we researched alternatives that are self-supervised. Finally, we describe a training pipeline consisting of five phases where the last phase is supervised using SupCon on the small labeled dataset.

Throughout the solution and evaluation sections, there will be a guidance figure like Figure 4.1 in each section to help the reader orient themselves.



Figure 4.1: Guidance figure displaying in which step we are at in the pipeline

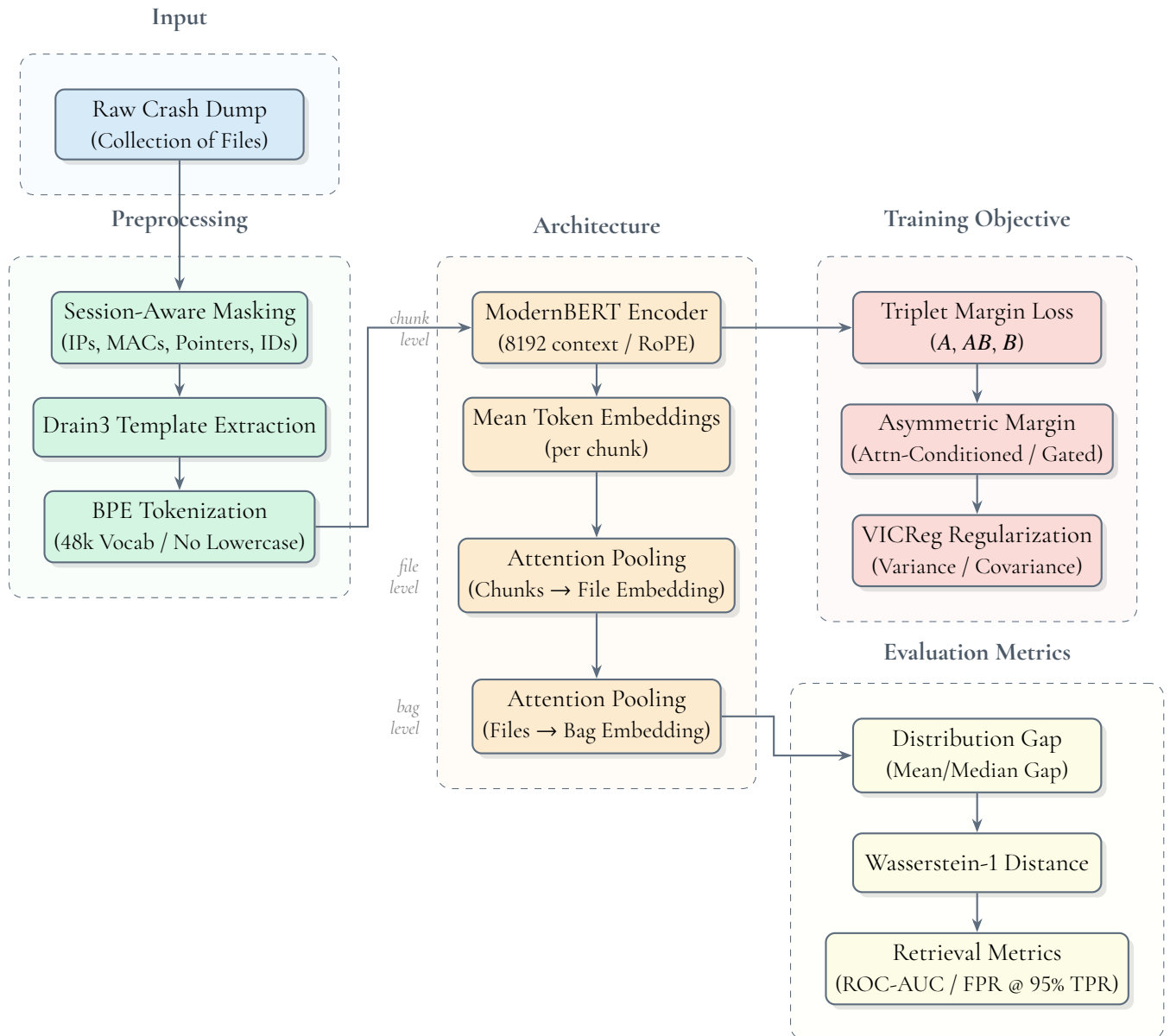


Figure 4.2: Refined pipeline overview. The modular structure clearly separates the preprocessing pipeline from the core MIL-based architecture and its dual training/evaluation objectives.

4.1 Models

This section gives a brief overview of our models, including how their training pipelines differ. A detailed description of the models is available in Section 4.9.

- *Baseline*: Utilizes the pretrained ModernBERT-base model with no custom training. Generates a bag embedding by hierarchically encoding and mean pooling. This is a naive baseline which all our proposed models are compared with.
- *Custom ModernBERT*: A custom ModernBERT-base model trained on our crash dump dataset. Training is self-supervised and described in Section 4.5.
- *Sym*: *Custom ModernBERT* based self-supervised model trained using standard triplet loss with a static margin. The model is trained to position crash dumps in latent space in a consistent manner.
- *SymAsym*: *Custom ModernBERT* based self-supervised model with more complex training curriculum. First epochs trained with standard triplet loss and then the rest using our asymmetric triplet loss.
- *SA-HEM-SA-SupCon*: A partly supervised model based on the *SymAsym* model. Uses a retrieval index created with *SymAsym* to retrieve difficult datapoints to train on. Runs the *SymAsym* curriculum again with the difficult datapoints. Then trains supervised using SupCon on the limited dataset of labeled defects.

Additionally, all models except *Baseline* and *Custom ModernBERT* use VICReg to avoid representation collapse.

4.2 Analyzing Labeled Crash Dumps



We started with manually analyzing the available labeled data. The dataset consists of over 300000 crash dumps, each of which has one or more associated Jira ticket (defect) as a label. It is common that a single defect has multiple related crash dumps. Additionally, each crash dump has a crash ID that serves as the label for the current clustering method. The ID is commonly created by taking a simple hash of the stacktrace file if it exists, else the ID is fully random.

The labeled dataset only contains around 700 unique crash IDs and was thus deemed unsuitable for the main training step as it would be extremely likely to overfit and not generalize well. Therefore, we opted to use the labeled dataset for testing and validation to ensure that the model achieves the target objective.

Because individual crash dumps can be associated with multiple defects, we used transitive closure to group related crash dumps into connected components, or "islands". This ensured consistent labeling and prevented the evaluation metrics from unfairly penalizing the model for finding valid indirect relationships. For example, if crash dump *A* was linked to the same defect as crash dump *B*, and crash dump *B* shared a different defect with crash dump *C*, all three were grouped into a single island and treated as related. When testing, identical crash IDs were not compared as they were likely to be extremely similar and thus trivial.

As part of our data analysis, we created a graph visualization to view the relations between the defects and crash dumps. The visualization can be seen in Figure 4.3 and it makes it clear that there are relatively few crash dumps that are connected via a common defect. This can also be seen in Table 4.1. Because contrastive learning relies on comparing positive pairs, each usable island must contain at least two crash dumps which further decreases the amount of usable labeled datapoints.

Number of islands	Defects in island
536	1
77	2
32	3
6	4
6	5
5	6
3	7
2	8
2	9
1	16
1	21
1	52

Table 4.1: Island size distribution in the labeled dataset. Islands are defined as being a group of crash dump and defect pairs such that they create a chain. Each island is a group in Figure 4.3.

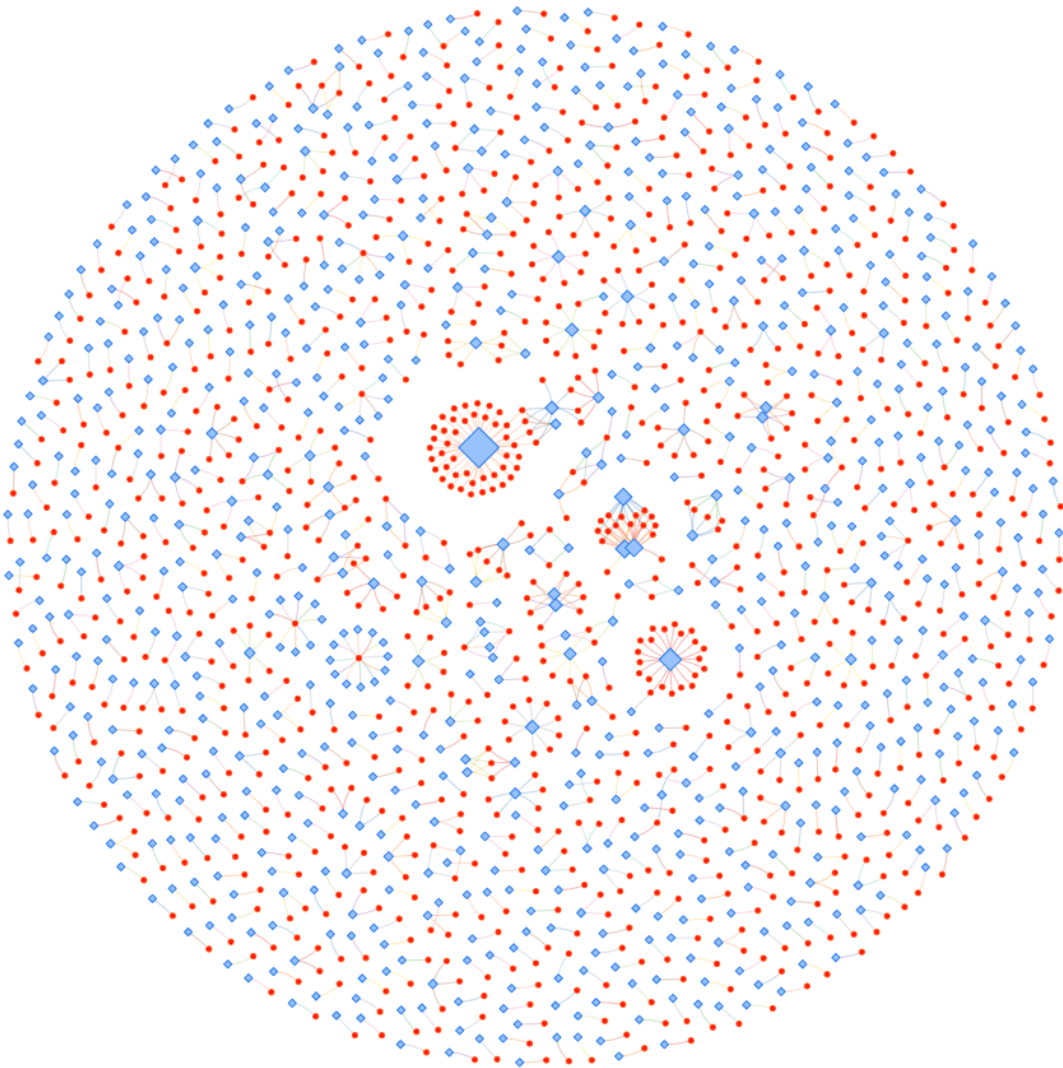


Figure 4.3: Relationship between labeled crash dumps. Blue squares are defects (labels), red circles are crash dumps. If a defect is related to a crash dump they have an edge.

4.3 Complete Dataset

In an effort to understand the structure of the crash dumps in the complete dataset including crash dumps without labels, we analyzed the frequencies of files across crash dumps. There were 106 distinct file types in total but some of them only occurred a couple of times. Only 35 files occurred more than 3000 times over our entire dataset of roughly 300,000 crash dumps. To simplify analysis and decrease training noise, we opted to remove all files that occurred fewer than 3,000 times in our dataset, resulting in the files seen in Table 4.2. The limit of 3,000 was chosen as it sits within a sharp drop-off in file frequencies which naturally separates common files from specific and manually added files. Examples of these files and their contents can be seen in Table 4.4.

Filename	Count	Filename	Count
system_info.txt	265,910	stacktrace-full.txt	118,116
dmesg.txt	217,840	trace.log	102,228
mmc_info.txt	217,840	stacktrace-stripped.txt	101,869
uptime.txt	217,840	stacktrace-extended.txt	72,425
platform_variables.txt	217,838	mem_info.txt	71,067
system_log.txt	217,838	log-gdb.txt	70,883
df.txt	217,646	package_name_version.txt	66,590
system_log_errors.txt	217,644	dump_nsp_servers.log	54,486
supercap_status.txt	217,155	system.log	47,760
resetcpu_log.txt	217,030	analysis-report.txt	47,237
packages_installed.txt	213,090	log-cdb.txt	47,237
journal.txt	143,654	orion_proxy_server.log	46,033
journal-err.txt	143,651	oopslog.txt	6,634
settings.txt	143,651	CoreLog.txt	5,947
journal-verify.txt	137,016	version_history.txt	5,945
match.txt	137,016	httpaccess.log	4,608
match_wo_stamp.txt	137,016	lsblk.txt	4,556
stacktrace.txt	122,615		
log-setup-debug.txt	118,520		

Table 4.2: All files with more than 3,000 occurrences in the crash dump dataset.

4.3.1 File filter

Some of the files seen in Table 4.2 were excluded from training as they appeared, upon inspection, to be mostly noise and thus risky to include. These excluded files, detailed in Table 4.3, typically contain static platform variables such as serial numbers, manufacturing IDs or user/system settings. This step is necessary to prevent the model from learning specific shortcuts using device identifiers or similar. An example of shortcut learning occurred when the model learned that it could achieve a high score by only looking at the OS or device identifiers.

4.3.2 File Distribution

To avoid shortcut learning using the distribution of files in a crash dump, we created a co-occurrence matrix that contains how often pairs of files occur in the same natural crash dump. We visualized the matrix as a heatmap and the results are visible in Figure C.1. From this, we can see that there are two large subsets of the dataset.

- **True Crashes:** These dumps are characterized by having stacktraces or core dumps. They represent fatal, unhandled exceptions where an application or system halted abruptly, leaving a snapshot of the call stack at the exact moment of failure.
- **Errors (Logs):** These dumps are characterized by having system logs, for example Linux journal files (`journal.log`). Rather than an abrupt halt, these represent continuously recorded warnings, states, and error messages.

This was mostly used in our initial pure MIL solution described in Appendix A.2, but helped us get a better understanding of shortcut learning in practice.

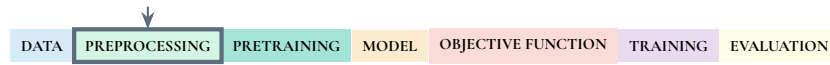
Filename	Reason
<code>time.txt</code>	Only contains a timestamp
<code>manual.txt</code>	Only contains a timestamp
<code>version_history.txt</code>	Workstation version, and timestamps
<code>defect.txt</code>	Contains ground-truth labels if existing (leak risk)
<code>journal-verify.txt</code>	Verifies journals exist, not informative
<code>match.txt</code>	Regex extract from Journal, duplicated information
<code>match_wo_stamp.txt</code>	Regex extract from Journal, duplicated information
<code>system_log_errors.txt</code>	Duplicate subset of <code>system_log.txt</code>
<code>orion_proxy_server.log</code>	Project specific, leak risk.

Table 4.3: List of manually ignored files used in preprocessing.

File Name	Content Snippet / Example Data
system_log.txt	- Reboot - vector : <MEM:1> - <MEM:2> (4 kB)
stacktrace-extended.txt	Backtrace stopped: previous frame identical to this frame (corrupt stack?)
lsblk.txt	mmcblk0boot0 16M; mmcblk0boot1 16M
dmesg.txt	Booting Linux on physical CPU 0 - Linux version 3.5.0-lsp-3.3.1-<DEVICE_ID:4>
resetcpu_log.txt	LED <MEM:29> <MEM:48> <MEM:18> ... set startup mode ... store data
system_info.txt	hardware=armv7l; OS=Linux 3.5.0-lsp-3.3.1-<DEVICE_ID:4>
uptime.txt	up 12 days, 9:08, load average: 12.01, 4.15, 2.49
mem_info.txt	MemTotal: 497044 kB - MemFree: 352716 kB
df.txt	rootfs 242.7M total - 228K used - 242.5M free
dump_nsp_servers.log	about to postcatch - FULL-PATH=/var/sbo/log/crash/<DEVICE_ID:32> - postcatch done
platform_variables.txt	system.network= (empty); dns0=<IP:4>
supercap_status.txt	Charge capacitance: 11.326167 - Charge voltage: 2.702051
trace.log	SSL channel closed - tlsio_openssl.c:797; close called when already CLOSED
log-setup-debug.txt	Probing artifactory... Found release <IP:15> of product ASP
mmc_info.txt	eMMC CID: fe014e4d4d433038474115b1e49ea100; d04f01320f59...
log-gdb.txt	core.1523 truncated - expected ≥ 772MB, found 10MB
stacktrace-stripped.txt	__GI_raise - __GI_abort - ContractViolationAction_Abort::OnViolation
packages_installed.txt	as-devs - 1.0-r6; <DEVICE_ID:65> - git10; avahi-daemon - 0.8-r0
package_name_version.txt	<DEVICE_ID:22> - <IP:3>-1
stacktrace.txt	panic: invalid memory address or nil pointer dereference
analysis-report.txt	Exception Analysis - *****
log-cdb.txt	Windows Debugger 6.2.8229.0 - User Mini Dump File
system.log	RecoverableException: Communication Error: Offline (<MEM:72>)
httpaccess.log	POST /json/POST 200 - Chrome/<IP:11> - multiple users (to-chowiczp, bakierskik)
stacktrace-full.txt	Thread 134 (LWP 795): #0 ?? () #1 ?? ()
journal.txt	vmalloc : <MEM:5> - <MEM:6> (744 MB)
journal-err.txt	pmem: Graceful shutdown intentionally disabled
settings.txt	system.network.domain= (empty); dns0=<IP:6>
oopslog.txt	TX Checksum insertion supported

Table 4.4: Summary of diagnostic and log file contents after preprocessing and masking has been applied.

4.4 Preprocessing



Raw crash dumps contain high-cardinality data such as unique IP addresses, MAC addresses, and timestamps, which do not contribute to semantic understanding of the defect. To address this, a preprocessing pipeline was implemented to normalize, filter, and structure the log data before tokenization.

4.4.1 Masking

Entities such as MAC addresses, IP addresses, device/project IDs and memory pointers are identified and replaced with indexed placeholder tokens such as (`<IP:1>`, `<MEM:1>`). This masking process is session-aware within the content of a single crash dump. If a specific memory address or IP appears across multiple files within the same dump, it is mapped to the same token ID. This preserves relational context between different log files in the same dump. This also holds true for timestamps, which are indexed in chronological order, even across different files, to preserve as much information as possible while decreasing the risk of shortcut learning.

To further decrease the risk of shortcut learning, entities such as project names, version IDs, and device IDs were masked. The motivation behind removing this metadata is to steer the model towards learning semantic meaning, which is important for finding the underlying reason for a crash. If the model is allowed to use information such as IP addresses, it may detect that some set of IP addresses corresponds to the same issues. This would hurt generalization. Since IP addresses are not always static and the same program may be deployed with new IP addresses, this is a misleading data point. Additionally, data such as memory addresses have very high cardinality but are irrelevant from a crash-analysis standpoint as memory addresses are ephemeral by nature.

4.4.2 Drain3

Log files can often contain repetitive messages and log entries generated by loops or recurring errors. The pipeline utilizes a library called Drain3 [9]. Drain3 is a log parser that extracts repetitive log messages.

4.4.3 Custom Tokenization

We opted to use Byte Pair Encoding (BPE) as our tokenization algorithm together with a regex pre-tokenizer. The pre-tokenizer makes sure that the tokenizer does not merge tokens in, for example, a file path with surrounding tokens. The file paths or other matched patterns can, however, be decomposed into subtokens by BPE.

We used the Unicode Normalization Form D (NFD) to decompose characters, but omitted any lowercasing steps to preserve the case-sensitivity of our dataset.

Table 4.5: Regex pre-tokenizer patterns applied before BPE training, in priority order (first match wins). Patterns 1–3 are tailored to the preprocessing pipeline. Patterns 4–7 are general.

Priority	Pattern / Description
1	<[A-Z_]+:\d+> Preprocessing placeholders (e.g. <MEM:1>, <IP:3>, <DEVICE_ID:12>)
2	[0-9a-fA-F]{8}(-[0-9a-fA-F]{4}){3}-[0-9a-fA-F]{12} UUIDs (e.g. 550e8400-e29b-41d4-a716-446655440000)
3	(?:[A-Za-z]:)?(?:[/\]^\\s]+)+ File paths (e.g. /var/log/syslog, C:\Windows\System32)
4	\b\d+\b Integers (e.g. 404, 1024)
5	\w+ Unicode words. Letters, digits, underscore (e.g. kernel, BSOD)
6	[^\w\s] Punctuation and symbols (e.g. :, =, ()
7	\s Whitespace characters

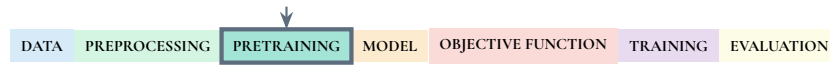
The specific pre-tokenization rules we used can be seen in Table 4.5. To perform the pre-tokenization and the tokenization, we used the HuggingFace `tokenizers` library [14].

To tune our configuration values for tokenization, we primarily monitored fertility, vocabulary utilization, and the rate of unknown tokens. We evaluated two different vocabulary sizes: 32,000 and 48,000. As shown in Table 4.6, the fertility rate for the larger vocabulary size (≈ 5) is significantly lower than for the smaller one (≈ 11). Because fertility captures how fragmented a base unit is (the number of tokens a word consists of), a lower value is generally preferable. However, a value too close to 1 could signify an overfitted tokenizer or an excessively large vocabulary. After reviewing these results, we decided to proceed with the 48,000-token vocabulary model.

Table 4.6: Comparison of tokenizer performance on 5,000 evaluation texts.

Metric	32k Vocabulary	48k Vocabulary
Coverage	1.0	1.0
Fertility	11.0626	5.0605
Vocab Utilization	0.838	0.8263
Vocab Size	32,000	48,000

4.5 Pretraining ModernBERT



Because ModernBERT is pretrained primarily on natural language and code, crash dumps fall entirely outside its training domain, making the base model unsuitable for encoding crash dumps. To solve this issue, we opted to retrain it from randomly initialized weights on our complete dataset of crash dumps. Another benefit of fully retraining is the ability to change the vocabulary and the tokenization scheme to better fit our domain. This custom pretrained model will be called *Custom ModernBERT* and uses the smaller *ModernBERT-base*.

Pretraining Methodology

As we decided to use a custom tokenization scheme, we were forced to either fully pretrain the model from scratch or to initialize our embeddings from the already-trained ModernBERT for overlapping tokens in our new vocabulary. To decide between these two options, we analyzed how many of the tokens from our 48k vocabulary exist in ModernBERT’s vocabulary. The results from this analysis showed that the overlap was only 20%. Because this overlap is very low, we decided to avoid the complexity of initializing from the existing embeddings as the convergence rate would likely be negligibly impacted.

Pretraining was done on our completed dataset of unlabeled crash dumps using MLM (Masked Language Modeling). As described in Section 2.3.4, in standard MLM a portion of the input tokens are hidden, and the model is tasked with predicting the original token based solely on the surrounding context.

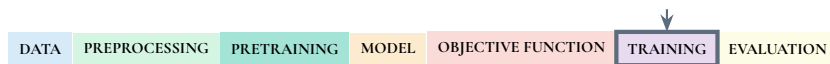
Following standard practices for training long-context models like ModernBERT, our pretraining process was split into three progressively longer phases for computational efficiency. In the original ModernBERT paper the training phase was split into two parts [23], beginning with a 1024-token context window and extending to 8192 tokens in the final phase. This practice is done for computational efficiency as the memory and compute requirements for the transformer scale quadratically with the sequence length. By doing the vast majority of pretraining on shorter sequences the model can learn fundamental vocabulary, syntax and local semantics at a lower computational cost. During the first phase, we trained using a maximum sequence length of 2048 tokens. This was subsequently doubled to 4096 in the second phase, and finally reached ModernBERT’s full capacity of 8192 tokens in the third phase.

In total, we trained the model on 200,000 samples evenly split between these phases. In this context a sample refers to a single continuous sequence of text extracted from one specific crash dump file. To maintain logical context boundaries, we populated each sample with text only from a single file, intentionally avoiding mixing content from multiple unrelated files into one sequence. If a file exceeded the current phase’s sequence length limit we simply truncated it.

For each phase, we used a learning rate scheduler with a linear warmup for 10% of the phase’s samples and then switched to cosine decay. The linear warmup was intended to avoid shocking the encoder when switching context windows, which could unlearn earlier information.

A masking probability of 40% was utilized, which is higher than the 30% probability used by ModernBERT [23]. The motivation for choosing a masking probability of 40% is that log data is more structured than natural language and thus likely easier to predict. Therefore, we increased the masking probability to make the task more difficult, leading to slower convergence toward a better optimum. Out of the 40% masked tokens, 80% were replaced with [MASK], 10% were replaced with a random token and 10% kept the original token.

4.6 Training Pipeline



This section describes all steps of our training pipeline. In summary, we start with pretraining ModernBERT with custom tokenization (Section 4.5). We then use that model as the encoder for our attention pooling and projection layer architecture that will learn to use multiple parts of the crash dump when creating a representation.

4.6.1 Training

In summary, our training loop works by first chunking each file in a crash dump (bag). We then used pretrained ModernBERT to embed them. For each file, the embeddings were pooled together using attention pooling, creating one embedding per file in the bag. We then had a separate attention pooling layer that combined all the file embeddings into a single bag embedding.

These bag embeddings served as the basis for our geometric triplet loss (Section 4.8.1). For each training step, we constructed two pure bags A and B , each containing files from a single crash dump, along with a mixed bag AB that contained files drawn from both. The triplet loss encourages AB to lie geometrically between A and B in embedding space, with an adaptive margin conditioned on the attention conditioning of the mixed files.

4.6.2 Validation

When training a machine learning model, some validation metric is almost always used to verify that the model is not overfitting on its input. When training using MIL, this was simply the same loss term as in training, but used on a separate, randomly selected sample of the entire dataset.

When training with our triplet loss method, using the same loss for validation does not make much sense, as the loss is built from many different parts, such as VICReg and triplet loss terms. We thus opted to only validate the triplet loss on a separate dataset as that is the task that most closely conforms to our goal. The regularization losses would likely not be meaningful for validation.

4.6.3 Memory

Checkpointing

Since our encoder (ModernBERT) processes large chunks of log files, we encountered severe memory limitations during the training phase. Specifically, executing the backward pass creates a massive computational graph that stores all calculations in each neuron. This quickly overloaded the GPU's 32 GB of VRAM. To solve this problem, we introduced gradient checkpointing. Instead of storing all the calculations, checkpointing only saves a few key data points. When the model needs the discarded information to learn from the data, it simply recalculates it on demand. This creates a trade-off: It requires more computation time, but reduces memory usage, allowing us to train the model without exhausting the VRAM.

Fragmentation

In our initial training loop, there was an issue in which the training would crash after some time, often around 6 to 10 hours. After extensive debugging, we noticed that this was the result of memory fragmentation caused by iterative allocation of small vectors in a for-loop. After vectorizing our code so that the operations were done in-place on a single matrix, these issues were solved.

4.7 Model Architecture



The foundation of our models is our shared *Custom ModernBERT* model, which we further train using additional layers. The complete pipeline can be seen in Figure 4.2, and this section further expands on the architecture part.

As the goal of the model is to map an entire crash dump to a single embedding, and each crash dump contains multiple files which are often massive, we create a hierarchical encoder structure. We limit each file to only having 500 chunks, corresponding to $500 \cdot 8192 = 4096000$ tokens. This constraint comes from the VRAM requirement during backpropagation and is almost always enough to ingest the entire crash dump.

For each chunked file in the crash dump, we then iteratively encode chunks in small batches into vector representations. We utilize small batches due to memory limitations. Additionally, we also utilize checkpointing to preserve VRAM.

When each batch is processed the output from the ModernBERT model is $B \times 8192 \times 768$, where B is the batch size. We then take the mean (instead of CLS described in Background 2.3.4) of each chunk's output so that we get $B \times 768$. Our reasoning for mean pooling instead of purely using the CLS token was that mean pooling adds very little overhead and should maximize the usage of the encoder's weights. This batching method is then used until a crash dump is fully encoded into $K \cdot \bar{N}$ vectors (each vector is 768 dimensional) ($c_{k,n}$ as seen in Figure 4.4), where K is the number of files and $\bar{N}$ is the average number of chunks per file.

A positional encoding is then added to each chunk embedding to encode its position within the source file. Each embedding is then normalized with LayerNorm. For simplicity

we used a standard sinusoidal position encoding. Adding positional encoding makes it so that the model can easily learn where in a file the chunk occurred. With this information the model can learn to attend based on relative positions [22].

At this point we are entering the "Chunk Attn Pool" step in Figure 4.4, and have a set of K groups each containing $\bar{N}$ chunk embeddings of size 768. As our goal is to convert each crash dump into a single embedding we will now continue aggregating these embeddings. The aggregation is done using attention pooling as described in Background 2.4.5. To do this we organize our chunks so that they are put in buckets depending on their file origins. All chunk embeddings from the same file are then passed into the attention pooling block which computes attention weights based on interactions between chunk embeddings for each embedding. The attention weights are normalized to sum to 1 across all chunks in a file. The weight for a chunk measures how important the attention pooling block deems it to be. These weights are then used to calculate a weighted average embedding for the file which gives us an f_K seen in Figure 4.4. When this has been done for all files our crash dump is of dimensions: $K \times 768$.

We then perform LayerNorm again on each of these vectors and use another attention pooling layer to create a single vector representation with 768 dimensions. This representation is not created purely with attention pooling but we also add a residual connection which mean-pools. This residual is not weighted, so both signals are always present which promotes stable training and prevents the model from becoming overly reliant on file attention weights. The residual pooling was added as a stabilization method to decrease the attention jitter, where file attention weights would change rapidly. The residual connection also preserves a global signal and preserves information from more unimportant files.

We now have a single 768-dimensional embedding for the crash dump (bag). This representation is then passed through a nonlinear projection head ($768 \rightarrow 1024 \rightarrow 128$). This downsampling is done as we noticed that it improved generalization due to reducing the risk of overfitting to noise.

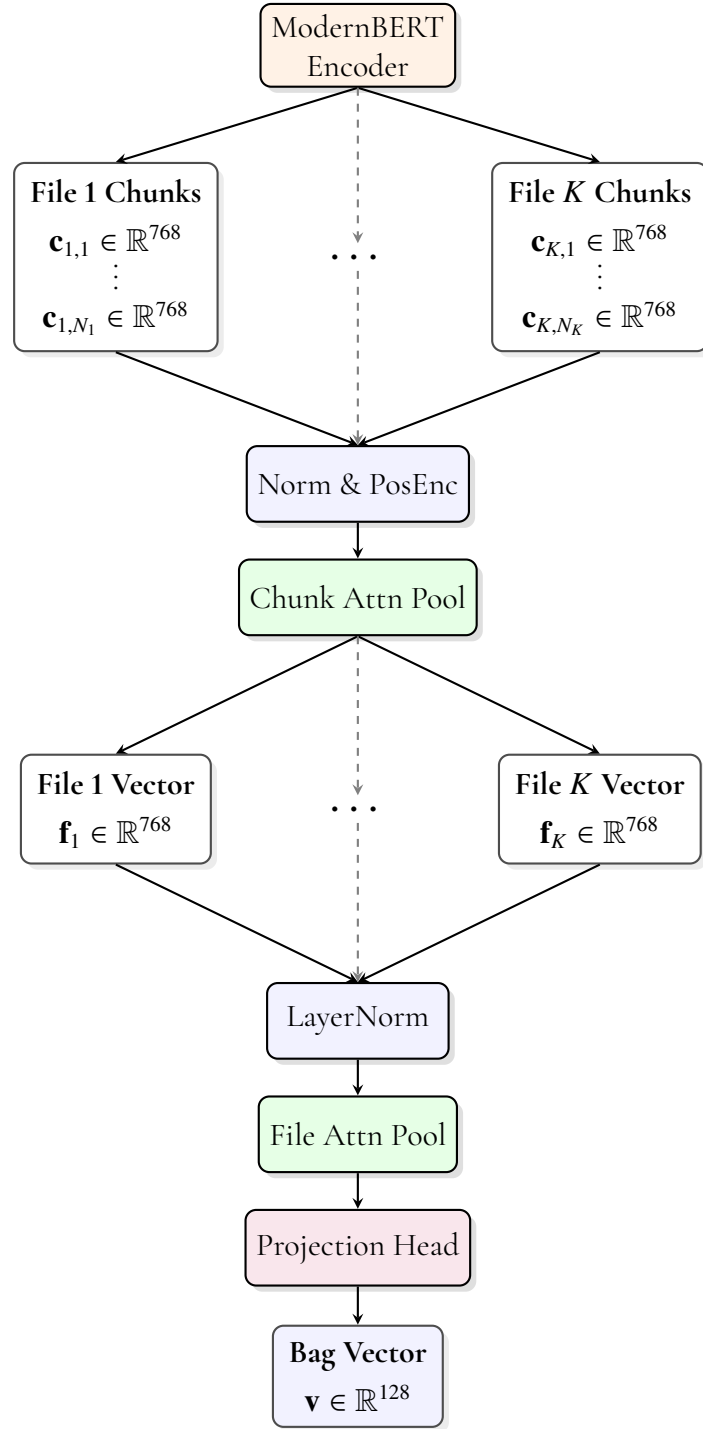


Figure 4.4: Shared architecture for all our models (Except the model described in Appendix A.2). The ModernBERT encoder outputs a sequence of chunk embeddings ($\mathbf{c}_{k,j}$) for each of the K files. In the notation $\mathbf{c}_{k,j}$, k is the index of the file and j is the chunk index in file k . Each file's sequence of chunks passes through normalization, positional encoding, and chunk-level attention pooling to produce K distinct file vectors ($\mathbf{f}_1 \dots \mathbf{f}_K$). These are normalized and reduced by the file-level attention pool into a single bag vector ($\mathbf{v}$), representing the entire crash dump, finally mapped to a 128-dimensional representation.

4.7.1 Hyperparameters

The most important hyperparameters while training our *SymAsym* model can be seen in Table 4.7 and a full list can be seen in Appendix B.1.

Table 4.7: Key hyperparameters used for training the *SymAsym* model. A complete list of configuration parameters is available in the Appendix B.1.

Category	Parameter	Value
Architecture	Base Model	<i>Custom ModernBERT</i>
	Max Sequence Length	8192
	Embedding Dimension	768
	Projection Dimension	128
Optimization	Batch Size	64
	Encoder Learning Rate	5×10^{-7}
	Attention Learning Rate	4×10^{-4}
	Projection Learning Rate	4×10^{-4}
Objective	Triplet Margin (γ_0)	0.05
	Variance Weight	0.05
	Covariance Weight	0.01
	Asymmetric Margin	True

4.8 Loss Function



As mentioned in the Background (Section 2.4.1), the loss function is one of the most important parts of training a machine learning model. The loss function is what teaches the model what to do and what not to do. Initially, we attempted to train by performing binary classification on whether a bag (Section 2.4.4) of files was cohesive or not. This led to shortcut learning and was discontinued. This behavior is documented in Appendix A.

Instead, inspired by how MIL represents each crash dump as a bag of files, we created a margin loss function that requires the model to predict the embedding for a mix of files from crash dump *A* and crash dump *B* (which we call bag *AB*). The model is penalized if the embedding of bag *AB* is not geometrically centered between the embeddings of *A* and *B*. When training this model, we noticed that it converged very quickly to relatively good results (Section 6). This rapid convergence suggested that the optimization problem was too easy.

To make the problem more difficult so as to allow more learning, we introduce an asymmetric attention-conditioned margin loss. This margin loss is similar to the standard triplet loss function except that we decrease the zero loss space and its center depending on how important files from the different bags are.

4.8.1 Margin Loss Function

We created a custom loss function with which we attempt to maximize the value of the information we have about the source of a mixed bag. As seen in Figure 4.5, this was implemented by enforcing that a pure bag A is closer to AB than A is to B and vice versa, where AB is a mix of A and B . This only uses information that we currently have and creates rather informative gradients. This loss was implemented as follows:

Let $A_i, B_i \in \mathbb{R}^D$ be embeddings of two pure crash-dump bags, and let $AB_i \in \mathbb{R}^D$ denote the embedding of a mixed bag constructed from both A_i and B_i . We use cosine distance

$$d(x, y) = 1 - \frac{x^\top y}{\|x\| \|y\|}.$$

For each mixed sample, we compute

$$d_{AB,A}^{(i)} = d(AB_i, A_i), \quad d_{AB,B}^{(i)} = d(AB_i, B_i), \quad d_{A,B}^{(i)} = d(A_i, B_i).$$

The objective encourages the mixed bag AB_i to lie between its two sources by enforcing that it is closer to both A_i and B_i than A_i and B_i are to each other. With margin $\gamma > 0$, the loss for sample i is

$$L^{(i)} = \max(0, d_{AB,A}^{(i)} - d_{A,B}^{(i)} + \gamma) + \max(0, d_{AB,B}^{(i)} - d_{A,B}^{(i)} + \gamma).$$

The final loss averages over all K mixed samples:

$$\mathcal{L} = \frac{1}{K} \sum_{i=1}^K L^{(i)}.$$

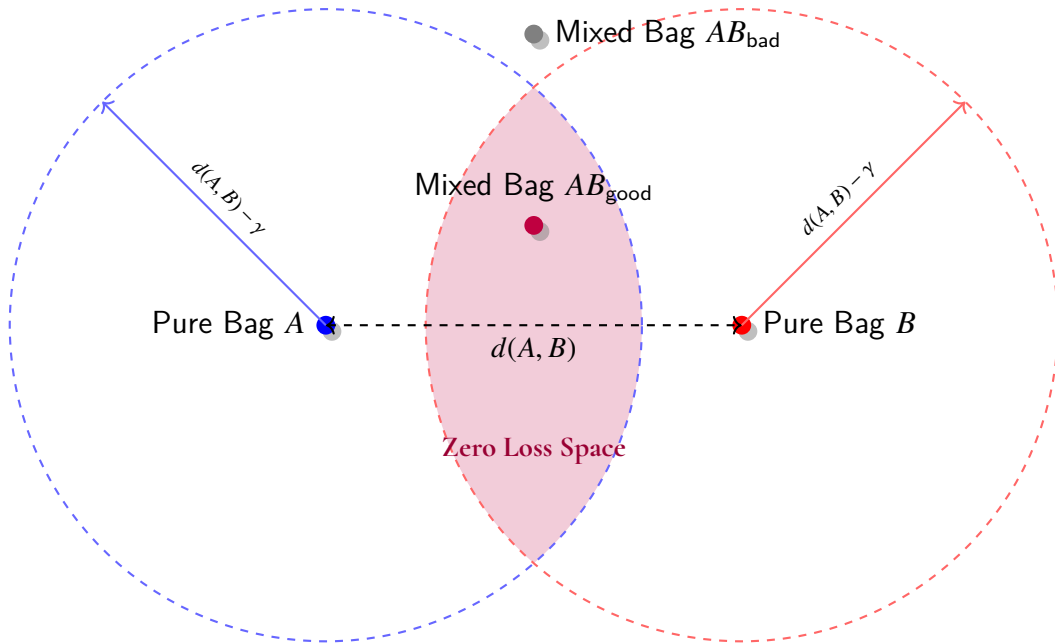


Figure 4.5: Geometric visualization of the custom margin loss. For the loss to evaluate to zero, the mixed bag AB must lie geometrically between pure bags A and B , falling specifically within the intersection of their target radii $(d(A, B) - \gamma)$.

4.8.2 Asymmetric Attention-Conditioned Margin

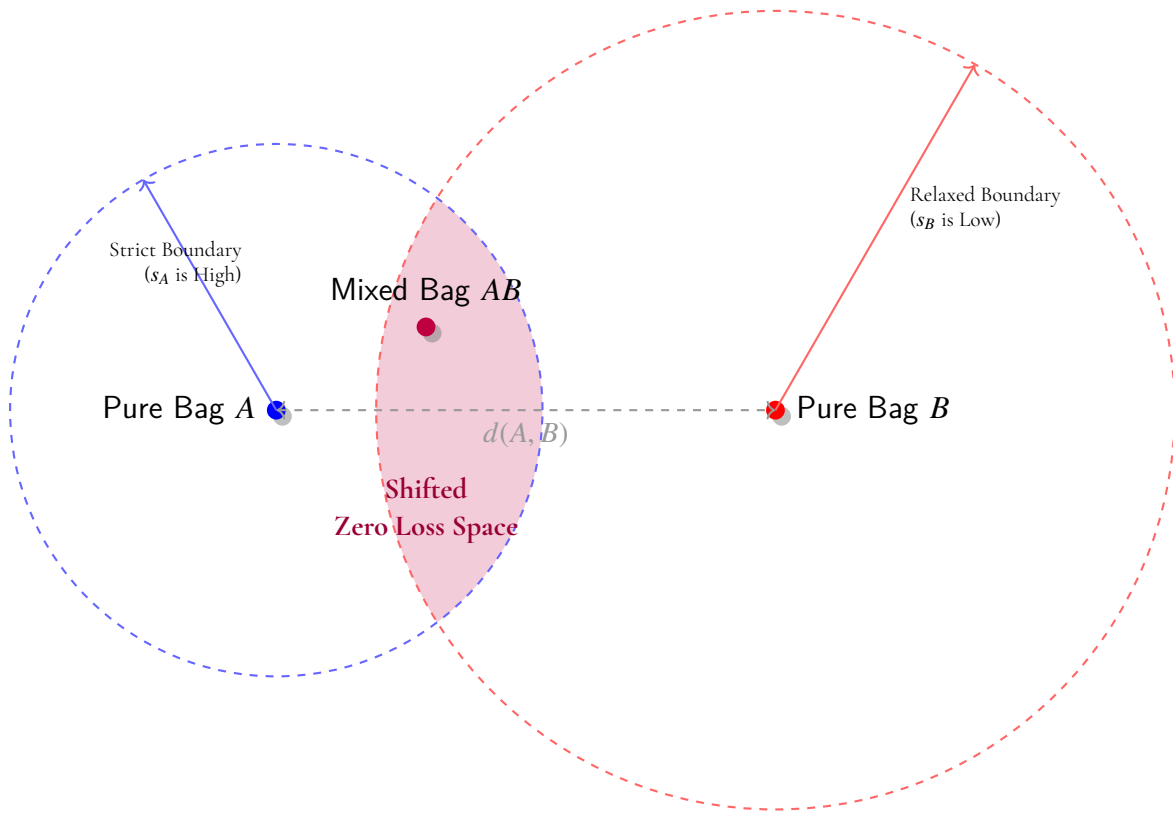


Figure 4.6: Geometric visualization of the asymmetric attention-conditioned margin. When the mixed bag AB is constructed using highly attended files from bag A (s_A is high) and only background files from bag B (s_B is low), the valid boundary radii become unequal. The zero-loss intersection shifts significantly toward A , enforcing that the mixed embedding strongly retains the semantic identity of its most critical source files.

The triplet loss function treats A and B as anchors and enforces that the mixed bag AB lies geometrically between them in embedding space. As seen in the margin loss function (Section 4.8.1), the base constraint requires:

$$d(AB, A) < d(A, B) + \gamma_0 \quad \text{and} \quad d(AB, B) < d(A, B) + \gamma_0 \quad (4.1)$$

where $d(\cdot, \cdot)$ denotes cosine distance and γ_0 is a fixed margin. Not all mixes are equally meaningful since files have different importance. Consider a mixed bag, AB , composed of high-attention files from bag A and low-attention files from bag B . In this scenario, the objective is to ensure that the resulting representation of AB is positioned closer to A than to B in the embedding space. Replacing important files changes the bag's identity, so this mix should produce a large shift in embedding space. On the other hand, replacing files that the model barely attends to should produce a minor shift. A fixed margin cannot capture this distinction.

To address this, we have an asymmetric attention margin that scales the triplet margin based on the attention of the mixed files, meaning how important those files were in their respective parent bags. To help get an intuitive understanding of this loss function, Figure 4.6 shows the case when AB should be closer to A than to B as A 's file should have a greater impact on its positioning based on our learned attention pooling.

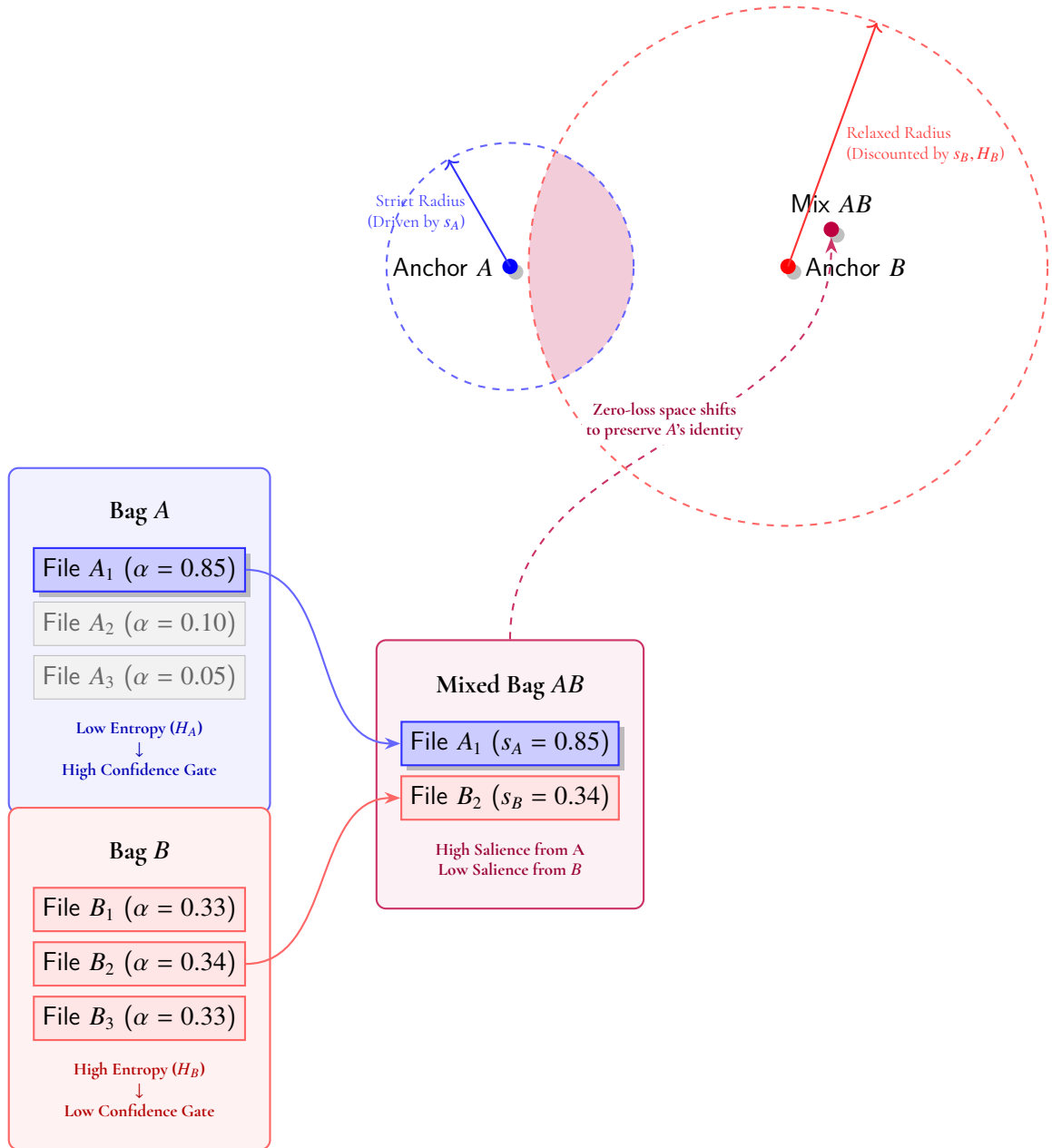


Figure 4.7: Intuitive visualization of the attention-conditioned margin. Bag A has a peaked attention distribution (low entropy), while Bag B has a uniform distribution (high entropy). When Mixed Bag AB incorporates salient files from A and background files from B , the resulting geometric target space shifts toward A , enforcing a stricter margin to preserve the semantic identity of the most salient parent.

Attention Conditioning

For each mixed bag AB , we know which files came from which parent. We look up the file-level attention weight that each mixed file received in its original parent bag and sum them by parent. We call this the attention conditioning s .

$$s_A^{(k)} = \sum_{f \in AB_k \text{ from } A} \alpha_A(f), \quad s_B^{(k)} = \sum_{f \in AB_k \text{ from } B} \alpha_B(f) \quad (4.2)$$

k represents the index of the specific mixed bag AB_k , and f denotes an individual file belonging to that mixed bag. The variable $\alpha_A(f)$ corresponds to the specific attention weight that file f was assigned when evaluated in its original parent bag A . A high value of s_A means the mix took files that parent A considered important. However, a low value of attention conditioning suggests only background files were taken.

Confidence Gating

We gate the attention conditioning using the entropy of the parent attention distribution because attention weights are only meaningful when the distribution is confident. A confident distribution implies that the model concentrates its weights on a select few files rather than spreading them evenly across the entire bag.

For a pure bag with attention weights α_i , the entropy measure that was used was standard Shannon entropy which is defined as $H = -\sum_i \alpha_i \log \alpha_i$. Shannon entropy is common in classification and clustering tasks in ML. A low entropy value means a cluster contains mostly one specific type of label, indicating high purity and predictability. Conversely, a high entropy value signifies a highly mixed cluster with many different labels, representing a greater uncertainty.

We convert this into a confidence score using a sigmoid gate. A sigmoid function is specifically chosen here because it provides a differentiable transition between states, avoiding sudden gradient jumps and training instability that a hard step function threshold would introduce. The confidence score is defined as follows:

$$c = \sigma(k \cdot (H_0 - H)) \quad (4.3)$$

H_0 is an entropy level reference, and k scales the gate. When the attention distribution is close to uniform, entropy is high, which makes the confidence low and suppresses the adaptive margin. When attention is peaked (low entropy), confidence becomes high and the attention conditioning is trusted. If the confidence falls below a minimum threshold, the gate is turned off, and the model falls back to the base margin.

Effective Margin

The total effective margin is a combination of the base margin with an attention-conditioned adjustment:

$$\gamma_{\text{eff}}^{(k)} = \gamma_0 + \min\left(\lambda(t) \cdot c \cdot 1[c \geq c_{\min}] \cdot \max(s_A^{(k)} + s_B^{(k)} - \tau, 0) \cdot s, \gamma_{\max}\right) \quad (4.4)$$

τ is an attention-conditioning floor that filters out low-attention-conditioned mixes. s is a scale factor, $\gamma_{\max}$ caps the maximum margin, and $\lambda(t)$ is a warm start schedule.

Warm-Start Schedule

To avoid creating a feedback loop that could potentially cause our file attention weights to converge to meaningless values, we phased in the attention-conditioned margin gradually using a schedule with three phases.

1. **Disabled** (epochs 0 to W): The model trains with only the base triplet margin ($\lambda = 0$, $\beta = 0$). This allows the encoder and attention layers to form initial representations without interference.
2. **Ramp** (epochs W to $W + R$): The schedule parameter $\lambda(t)$ increases from 0 to $\lambda_{\max}$ following a cosine curve, and a blending coefficient β ramps from 0 to $\beta_{\max}$.
3. **Max** (epochs beyond $W + R$): Both λ and β remain at their maximum values.

The full triplet loss is calculated using the following attention-conditioned formulation:

$$\mathcal{L}_{\text{triplet}} = (1 - \beta) \cdot \mathcal{L}_{\text{base}} + \beta \cdot \mathcal{L}_{\text{att-cond}} \quad (4.5)$$

For phase 1, where the attention conditioning is disabled, $\beta = 0$, meaning that only the static symmetric margin is used. In phase 2, we gradually increase β until it reaches its maximum value, which is then maintained for the rest of the training.

4.8.3 VICReg

To prevent representation collapse in both our supervised and self-supervised models, we integrated VICReg into our training objective. Since our dataset consists of repetitive log files, the risk of our model relying on shortcuts or collapsing its embedding space is significant.

The VICReg terms were applied to the file-embedding output after the attention pooling layer for each training batch. We did not utilize the invariance term of VICReg. In standard VICReg, the invariance term relies on generating two augmented views of the same sample and forcing their embeddings to be identical. However, we did not apply any intra-file text augmentation to crash dumps such as randomly dropping lines or masking text chunks. We did, however, use the variance and covariance terms, which proved to help against shortcut learning. The resulting VICReg terms were then added to the loss terms to create the respective objective functions.

4.8.4 Hard Example Mining

When analyzing what files the *SymAsym* model attended to the most, we noticed that it preferred files that held identifying information such as the OS or the commit hash. We theorize that the reason behind that is that a random pair from the dataset is likely to be from completely different projects for which the easiest positioning is attempting to assign a unique location in the latent space.

We implemented a Hard Example Mining (HEM) method to decrease the effectiveness of this shortcut method. The goal is to select more difficult pairs to use when continuing the training of *SymAsym*. This method eventually led to the creation of the *SA-HEM-SA-SupCon* model.

The reasoning behind this is that simply assigning a unique spot in space based on device or code version identifiers decreases in efficiency when comparing pairs that are from the same project and may even be from the same device and code version.

Our HEM implementation is rudimentary and simply uses a static index. The index was built using the *SymAsym* model and the training dataset is created by selecting random pairs from the [200, 1000] most similar (by cosine similarity) candidates to the anchor. The mined neighbor to A was used as the paired crash dump B in the triplet loss function using (A, AB, B) . To avoid the model unlearning how to differentiate between very different crash dumps we also randomly sample 50% of the crash dumps in a batch from the entire corpus. The [200, 1000] range was selected to limit the number of positives while still containing informative negatives. Many unexpected positive pairs could lead to the model learning to push them apart in the latent space. This range is, however, not optimal, as there are some defects that have over 200 crashes from which we could incorrectly use true positives as negatives.

4.8.5 SupCon

While a majority of our training pipeline relies on self-supervised learning due to the lack of labels, a small subset of our dataset is manually labeled with Jira defect tickets. To utilize this ground truth data, we introduced a Supervised Contrastive (SupCon, Section 2.4.1) learning phase for our model.

SupCon maximizes the utility of the limited labels. For any given anchor crash dump in a batch, SupCon contrasts it against all other crash dumps in that same batch. It pulls together the embeddings of all crash dumps that share the same defect while pushing apart embeddings that do not share a defect.

The reasoning behind doing a supervised phase was an attempt to minimize and correct for the faults created after using HEM. When using HEM, we noticed that the model correctly separated many of the defects. We had two primary random pair distributions, one with high similarities and one with low, but we theorized that this was only an effect of HEM losing some global knowledge. Because HEM disproportionately trains the model on difficult cases while removing easily separable random pairs, the model likely begins to forget the global structure while optimizing the local.

To solve this, we thus applied SupCon as the final phase, hoping to correct the global knowledge while keeping the valid information learned from the HEM phase.

As we only have a very small labeled dataset where we use pairs of defects, we made the problematic choice of not withholding part of the dataset for validation. This choice was made because the scale of our dataset scales roughly quadratically with the number of pairs, so removing even 10% would have a large impact on dataset size.

An additional reason for not creating a validation dataset for the SupCon stage is the difficulty of creating a representative dataset without losing too many pairs. We would likely be required to remove full islands to lessen the risk of data leakage between train and validation datasets. This is, however, still risky, as there are cases where two defects are very similar and should likely be merged, which would still effectively be data leakage. Creating a truly indicative train/validation split was deemed too time-consuming.

4.9 Detailed description of the models

This section describes the various models in greater detail.

4.9.1 *Baseline Model*

To establish a baseline, we implemented a simple model for comparison. We established this baseline by utilizing the pretrained ModernBERT-base model in a naive manner. Specifically, we combined all files within a crash dump into a single large string. We then divided this string into segments of 8,192 tokens, padding them where necessary to maintain a consistent sequence length.

No overlap was used during this segmentation process. However, we did utilize attention masks to ensure any padded subsequences were appropriately ignored. These segments were then fed into the pretrained ModernBERT model to generate individual chunk embeddings. Finally we applied a naive, two-layer mean pooling to create a single representation for the entire crash dump. First all chunk embeddings within a specific file were mean-pooled to create a file-level embedding, then all file level embeddings were mean-pooled together to generate the final crash dump representation.

4.9.2 *Sym Model*

The *Sym* model introduces our proposed Hierarchical Multiple Instance Learning. Instead of just standard ModernBERT, this solution utilizes our domain-specific *Custom ModernBERT* as the encoder. It replaces the mean pooling of the *Baseline* with hierarchical attention pooling. Specifically it uses two distinct attention pooling layers. The first aggregates chunk-level embeddings into one file level representation by learning which segments contain the most important information. The second layer aggregates those file-level representations into a single bag-level embedding for the entire crash dump, allowing the network to weigh which files are most informative to the overall crash dump. Finally, the bag-level representation is passed through a projection head to compress the embedding space (from 768 to 128). The *Sym* model was also trained using a fixed-margin symmetric triplet loss, alongside VICReg regularization to prevent representation collapse.

4.9.3 *SymAsym Model*

The *SymAsym* model introduced a new training loss function and is an improved version of the *Sym* model. It shares the exact same hierarchical MIL architecture, attention pooling layers, and projection head as the *Sym* model. The only difference is its loss function. The *SymAsym* model was trained using both the standard triplet loss and the asymmetric attention-conditioned margin that scales based on the learned attention weights (Section 4.8.2).

4.9.4 *SA-HEM-SA-SupCon* Model

The *SA-HEM-SA-SupCon* model is our proposed supervised solution which extends the capabilities of the *SymAsym* model. This model takes the fully trained weights of the *SymAsym* model (at epoch 170) and applies a two-phase fine-tuning process.

The first phase is fine-tuned on Hard Example Mining (Section 4.8.4). Using a retrieval index created from the *SymAsym* embeddings, we identified and isolated "hard" data points. The model was then trained exclusively on these difficult examples using the same asymmetric loss function. Finally, a second fine-tuning phase was executed using Supervised Contrastive Learning. This phase used the limited dataset of human-labeled defects. The goal of this phase was to guide the model to pull known related defects closer.

Additionally, this model uses a dropout method in both the HEM and SupCon stages with a 10% chance of dropping a file entirely. This dropout contained some restrictions. For example, we required that the combined *AB* bag contained at least one informative file from both *A* and *B*. We simply define an informative file as being one of the following: stacktrace, trace log, or journal file, ensuring essential context is preserved during dropout. The reasoning behind dropping files was to improve the difficulty of the task with the aim of learning more general knowledge and allow further training while slightly decreasing the risk of overfitting.

4.10 Iterative Architectural Evolution

Developing the final architecture was a highly iterative process. Before creating the models presented in this thesis, we conducted many experiments to guide our approach and address unexpected behaviors. This section briefly discusses how our architecture evolved to its current state, highlighting the key challenges we encountered and the corresponding solutions we implemented.

Our initial approach was to formulate the clustering problem as a binary classification task. As in standard MIL the objective was to teach the model to classify whether a bag was purely from a single crash dump or contained files from multiple crash dumps. This model initially appeared promising, achieving good accuracy but upon further investigation, we discovered that it had major shortcut learning problem, where it, for example, learned to spot OS differences within a bag which was an easy tell that it was a mix. Attempts to decrease shortcut learning were done by hardening different shortcut paths but the model would simply discover a new method. This method was thus discontinued and is described in greater detail in Appendix A.2.

After the failure of the initial approach we researched training methods and loss functions which decrease the risk for shortcut learning. We also wanted loss functions which more directly force the model to learn relations in the latent space. In this research we found metric learning methods which directly inspired the *Sym* model. This method did not exhibit obvious overfitting or shortcut learning. However, this standard margin loss converged quickly to disappointing results, which led us to research methods for increasing the task difficulty. Here we got the key insight that the difficulty from *AB* combinations inherently depends on how important parts from *A* and *B* are. As we had access to the model's assessment on the importance of files (from the attention pooling layer) we decided to experiment with

whether that could be used to increase task difficulty. This led us to create the loss function used in *SymAsym*.

A key technical insight arose from analyzing the behavior of the metric objective function and how the average file attention values changed over time. During this analysis we discovered that the model learned a new shortcut method where it would prioritize files such as `system_info`, which contains information that can be used to create a unique identifier for a device at a specific code version. The model had likely learned that if the same device with the same code crashes twice, they are likely related.

To counteract this, we gradually phased in the Asymmetric Attention-conditioned margin loss after initially using only the standard margin loss for some epochs. Even a small number of warmup epochs appears to have modified the file pooling weights enough to avoid total collapse to background files. This gradual blending approach did, however, not fully solve the attention issue, as files such as `stacktrace` were mostly ignored. To solve this, we started experimenting with running some supervised SupCon training, but this alone did not have any significant effect as the training set was likely too small and the model likely managed to only modify the projection layer to fit the data.

When manually inspecting the results using a custom internal web interface (which is detailed later in Section 8.1.3), we noted that the *SymAsym* model correctly retrieved crash dumps from the same projects and often the same code versions, but struggled with detecting small but important semantic differences in the crash dumps. We then theorized that a reason for this is that, as our dataset is large and contains crash dumps from many different projects and code versions, simply clustering each project and version into different clusters and learning this categorization made minimizing the overall loss rather simple. The model effectively learned a shortcut method to minimize the overall loss while not improving on difficult comparisons, which were rarely seen in the training set.

As a solution to this, we started looking into various methods for increasing the difficulty of the dataset in an effort to force the model to learn the crash dump’s semantic meaning. Early ideas were to pivot to using a focal-loss-inspired [12] loss function that increases the loss gradient for more difficult cases, effectively behaving similarly to Online Hard Example Mining [21]. This was, however, deemed to be problematic for our use case because it does not solve the issue of the vast majority of our pairs being very different and would likely require a huge number of epochs to train. Instead, we researched other Hard Example Mining methods and decided to implement retrieval-based HEM.

An index containing embeddings generated with *SymAsym* was created and used to retrieve hard training examples. To avoid the model forgetting global structure, 50% of the samples in a batch were randomly selected from the entire dataset. After training this model performed poorly on our quantitative metrics and even performed worse than the naive *Baseline*. However, it did appear to have more realistic average file attention values.

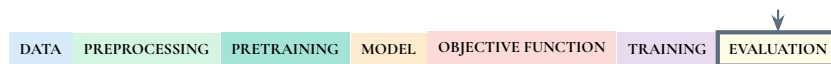
We theorized that this was largely a result of the new task having a much higher difficulty and successfully unlearning the previous shortcut method that decreased the model’s ability for global positioning while potentially having improved in local positioning.

In an attempt to improve the global performance while retaining the file attention weights, we decided to do a supervised training phase using our small labeled dataset and SupCon. During the training of *SA-HEM-SA-SupCon*, file dropout was introduced in an effort to increase difficulty and augment the data to increase the number of samples required before potential overfitting. In an effort to maximize the available training data, we opted not to

split the data into train/test and instead trained on all the data. As a result of this, the quantitative metrics are not comparable with the other models, but during manual inspection, the model seemingly yields much better results.

Chapter 5

Evaluation Framework



Evaluating unsupervised clustering models presents unique challenges for evaluation, primarily due to our severe scarcity of explicit ground truth labels to compare against. This chapter outlines the overarching evaluation framework designed to address this limitation. It details the methodology for constructing test defect pairs from a limited subset of manually labeled data and formulates the core problem as a pairwise similarity and retrieval task.

5.1 Test Defect Pairs

To evaluate the retrieval and clustering performance of all models, we utilized the small human-labeled dataset consisting of historical crash dumps mapped to their underlying defect.

5.1.1 Data Contamination

It is important to note that labeled crash dumps were not excluded from the training sets of either the self-supervised or supervised models. In the supervised case, this is highly likely to inflate the performance metrics due to data leakage. However, for the self-supervised models, the risk of cheating is minimal as the model is never exposed to the defect labels during training.

The decision not to exclude test defect pair crash dumps from the self-supervised training dataset was based on the homogeneity of the data set. Given the extremely large portion of unlabeled crash dumps, it is probable that nearly identical dumps exist within the unlabeled crash dumps. Therefore, excluding only the labeled subset would likely only provide a false sense of security regarding the risks of data leakage. Truly decoupling the dataset and testing

set would likely require using some more primitive embedding model to remove all similar crash dumps from the dataset.

5.1.2 Pair-Wise Comparison

Evaluation was formulated as a pair-wise similarity task. First, we grouped the labeled crash dumps into connected components or islands based on their labels. For example, if crash dump *A* was linked to the same defect as crash dump *B*, and crash dump *B* shared a different defect with crash dump *C*, all three would be grouped into a single island. Next, any two crash dumps within the same island were used to generate positive pairs, while any two crash dumps from different islands were used to generate negative pairs. It is important to note that because the labeled data set is created manually (Section 2.1.1), it likely contains duplicate unlinked tickets. Therefore, some of these generated 'negative pairs' may in reality be undocumented positive pairs. This likely flaw in the ground-truth data means that our model's False Positive Rate may be slightly overestimated, as the model could be correctly identifying links that human engineers previously missed.

It is important to note that we did not include any unlabeled crash dumps when generating negative pairs. Because these dumps lack defect labels, in practice they could stem from the same underlying defect. Including them would risk introducing false negative labels into the ground truth, which would unfairly penalize the model by registering valid semantic matches as false positives.

For every generated pair, we extract the model's embeddings and compute their cosine similarity. By comparing the distribution of similarity scores for the positive pairs against the random negative pairs, we could evaluate the model's performance. Specifically, we evaluated the models using metrics such as ROC-AUC, PR-AUC, and the False Positive Rate at a 95% True Positive Rate (FPR @ 95% TPR), as well as distributional distance metrics like the Overlap Coefficient (OVL), all of which are detailed in the following evaluation chapters.

5.2 Retrieval task

Given a selected crash dump (the query), the objective is to retrieve the most semantically relevant historical crash dumps from a large database. To evaluate how well the model performs this retrieval task, we rely on the pairwise similarity framework established in the previous section. In a successful retrieval system, a query crash dump must yield high similarity scores with historical crash dumps originating from the same defect (positive pairs), while maintaining low similarity scores with unrelated dumps (negative pairs).

5.3 Evaluation challenges

When evaluating the final supervised phase of our pipeline (*SA-HEM-SA-SupCon*) we faced a strict constraint: the fact that we did not have enough labeled data to perform a standard train-test split.

In normal circumstances, a dataset is divided into training, validation, and testing sets to ensure the model generalizes to unseen data. However, applying this standard split to

our specific problem was practically impossible due to the extreme scarcity of ground truth labels. Contrastive learning relies on pairwise relationships (pulling positive pairs together). Therefore, removing even a small percentage of the data for validation results in a large loss of positive pairs.

As a result, we opted to use the entirety of the available labeled defect dataset for the final *SA-HEM-SA-SupCon* fine-tuning phase. While this was necessary to preserve the greatest number of training pairs, it meant that there was no isolated validation set available to objectively measure the final model’s performance.

Chapter 6

Quantitative Evaluation

To objectively measure the retrieval capabilities of the proposed models, mathematical metrics are required. This chapter introduces quantitative evaluation metrics which are useful for retrieval tasks, such as, FPR @ 95% TPR, ROC-AUC, and, ROC curves which are useful for selecting suitable thresholds for retrieval. Following the definitions, the chapter presents and discusses the quantitative results, directly comparing the performance of our proposed models against the baseline.

6.1 Quantitative Evaluation Metrics

To evaluate how well our models perform the retrieval task, we must measure their ability to correctly associate related crash dumps and separate unrelated ones. This section describes the metrics used to assess the models' performance.

6.1.1 Evaluation Metrics

Because we evaluate every pair of crash dumps using a continuous cosine similarity score (ranging from -1 to 1), the system requires a mathematical decision boundary to categorize these scores into the binary outcomes listed below. This boundary is known as the similarity threshold. The threshold acts as the cutoff point for the retrieval task, meaning if the cosine similarity between two crash dumps is greater than or equal to the threshold, the model predicts that they are related (positive prediction). If the cosine similarity is strictly less than the threshold, the model predicts that they are unrelated.

Once the model makes this prediction based on the threshold, the result is compared against the human-labeled ground truth. For instance, if the threshold is set at 0.80 and a pair achieves a similarity of 0.85 , the model predicts they are related. If the ground truth confirms that they share the same root defect, this outcome is recorded as a true positive.

To calculate our evaluation metrics, we count the total number of each outcome across the evaluation dataset:

- **True Positives (TP):** The number of crash dump pairs correctly predicted to belong to the same defect.
- **True Negatives (TN):** The number of crash dump pairs correctly predicted to belong to different defects.
- **False Positives (FP):** The number of crash dump pairs falsely predicted to belong to the same defect.
- **False Negatives (FN):** The number of crash dump pairs falsely predicted to belong to different defects.

Using these absolute counts, we calculate the following evaluation metrics:

- **True Positive Rate (TPR):** $\frac{TP}{TP+FN}$. This represents the number of true positives divided by the total number of pairs that actually belong to the same defect.
- **False Positive Rate (FPR):** $\frac{FP}{FP+TN}$. This represents the number of false positives divided by the total number of pairs that actually belong to different defects.
- **True Negative Rate (TNR):** $\frac{TN}{TN+FP}$. This represents the number of true negatives divided by the total number of pairs that actually belong to different defects.
- **False Negative Rate (FNR):** $\frac{FN}{FN+TP}$. This represents the number of false negatives divided by the total number of pairs that actually belong to the same defect.
- **Precision:** $\frac{TP}{TP+FP}$. This represents the number of true positives divided by the total number of pairs that the model predicted to belong to the same defect.
- **Recall:** $\frac{TP}{TP+FN}$. This is mathematically identical to the True Positive Rate, representing the fraction of actual same-defect pairs that the model successfully identified.

6.1.2 Dynamic Threshold Metrics

Because the choice of a specific similarity threshold directly dictates how many pairs are classified as positive or negative, evaluating the model at just a single fixed threshold is insufficient. Therefore, we utilize dynamic threshold tests to evaluate the model's performance comprehensively across all possible decision boundaries.

ROC-AUC

To capture the overall retrieval capability of the model, we use the Receiver Operating Characteristic (ROC) curve, which evaluates the model dynamically across all possible classification thresholds. The ROC curve is generated by sweeping the decision threshold from its most strict value down to the most lenient. At every step, the model's TPR is calculated and

plotted on the Y-axis against its FPR on the x axis. The area under the curve (AUC) summarizes this plot into a single metric ranging from 0 to 1. Intuitively, the ROC-AUC represents the probability that the model will correctly assign a higher similarity score to a randomly chosen positive pair than to a randomly chosen negative pair. A value of 1 indicates perfect classification, while 0.5 represents random guessing.

PR-AUC

Precision-Recall Area Under the Curve (PR-AUC) measures the area under the Precision-Recall curve. The Precision-Recall curve is created by plotting Precision on the y-axis against Recall on the x-axis over all similarity thresholds.

While ROC-AUC provides a general overview of model performance, it can overestimate results on imbalanced datasets where negative pairs greatly outnumber positive pairs. As the calculation of False Positive Rate includes True Negatives in the denominator, a large number of negative samples can minimize the FPR even when the number of false positives is high. We include this metric as our small labeled dataset is highly imbalanced in terms of how many positive pairs one sample has in relation to the number of possible negative pairs.

Equal Error Rate (EER)

To evaluate the optimal threshold for classifying whether a pair is positive or negative, we also calculate the Equal Error Rate (EER). The EER is the point at which the False Positive Rate (FPR) and the False Negative Rate (FNR) are exactly equal. In our project, this metric can be used as it indicates how well the model performs when neither favoring Precision nor Recall. A lower EER indicates a more accurate model that successfully minimizes both the merging of unrelated crash dumps and the separation of related ones.

FPR @ 95% TPR

Potentially the most important metric for our use case is the False Positive Rate at a 95% True Positive Rate (FPR @ 95% TPR). This measures what the false positive rate is when the decision threshold is explicitly set so that exactly 95% of the true positive pairs are successfully retrieved.

This metric correlates very strongly with the model's practical usefulness for retrieval, as it represents the number of false alarms that an engineer would have to sift through when finding related pairs. See Figure 6.1c for a visual representation of how this metric can be set depending on the required TPR.

6.2 Quantitative Results and Discussion

To measure the effectiveness of our proposed self-supervised learning approach, we compared our trained models (the *Sym* model defined in Section 4.9.2 and the *SymAsym* model defined in Section 4.9.3) against the naive *Baseline* described in Section 4.9.1. Our final supervised model *SA-HEM-SA-SupCon* is intentionally left out due to data leakage between training and validation.

Table 6.1: Comparison of model performance against theoretical optimal values. The vertical bar separates metric metadata from experimental results. **Bold** values indicate the best performance achieved. The *Baseline* represents the naive pooling of pretrained ModernBERT embeddings. The *Sym* model utilizes a Triplet Loss (A, AB, B) architecture with a 0.05 margin and VICReg. The *SymAsym* model uses our Asymmetric Attention-Conditioned Margin.

Metric	Range	Optimal	Base	Sym	SymAsym
ROC-AUC $\uparrow$	[0, 1]	1.0	0.8750	0.9163	0.9209
PR-AUC $\uparrow$	[0, 1]	1.0	0.8665	0.9081	0.9089
Equal Error Rate $\downarrow$	[0, 1]	0.0	0.1894	0.1725	0.1523
FPR @ 95% TPR $\downarrow$	[0, 1]	0.0	0.4726	0.3052	0.2457

As seen in Table 6.1, the *SymAsym* and *Sym* models both perform better than the *Baseline* model on all metrics. The overall retrieval quality of the models is captured by the ROC-AUC and PR-AUC scores. The *Baseline* achieves a ROC-AUC of **0.8750**, but the introduction of the hierarchical MIL architecture and base triplet loss in the *Sym* model provides a large improvement to **0.9163**. The *SymAsym* model further improves the performance to **0.9209**, indicating good performance at ranking true positive pairs above negative pairs across all possible thresholds.

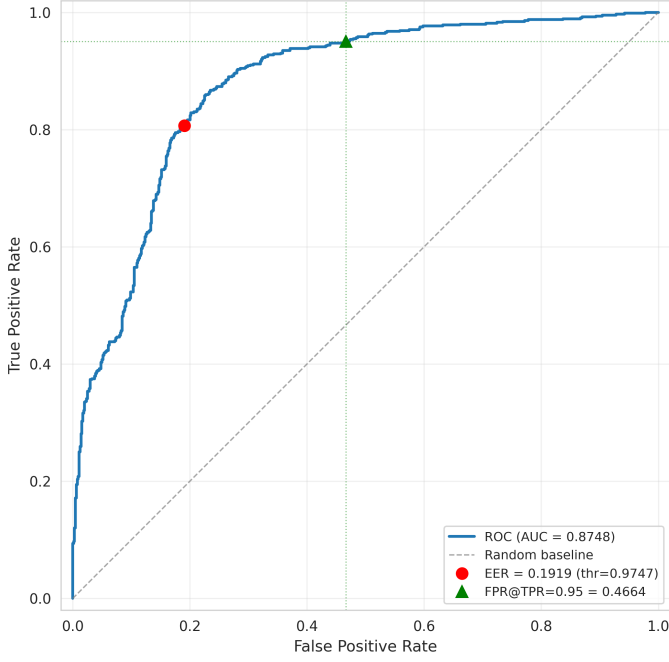
Importantly, because our evaluation dataset is highly imbalanced, containing far more negative pairs than positive ones, the PR-AUC serves as a critical indicator of true performance. The *SymAsym* model achieves a PR-AUC of **0.9089** (compared to the *Baseline*’s **0.8665**). This confirms that the model maintains high precision even as recall increases, proving that the ROC gains are true improvements and not caused by a large number of true negatives.

To understand the models’ performance at a balanced decision boundary, we evaluate the Equal Error Rate (EER). The *SymAsym* model successfully reduces the EER to **0.1523**, a significant improvement over the *Baseline*’s **0.1894**. This means that when the model is forced to balance precision and recall equally, it makes significantly fewer combined false positive and false negative errors.

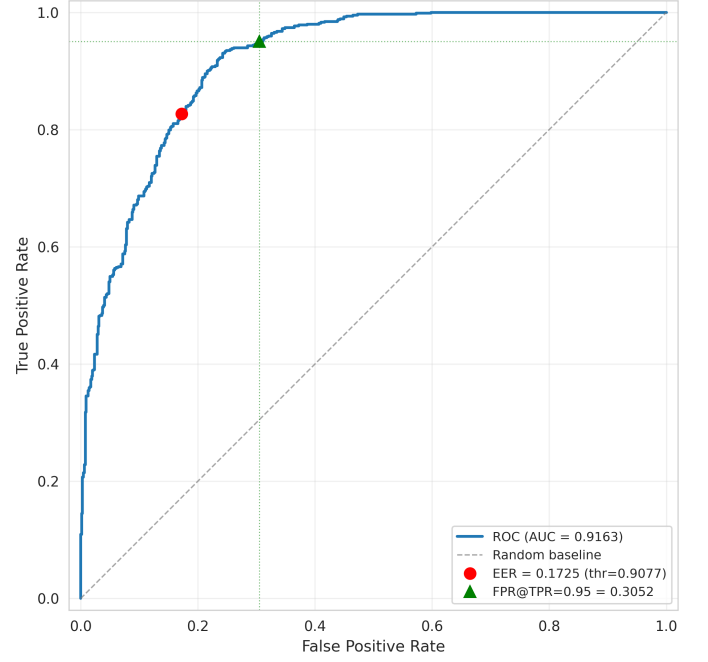
Furthermore, *SymAsym* reduced the False Positive Rate (at 95% TPR) to **0.2457** from the *Baseline*’s **0.4726**. This result means that when the threshold is set so that 95% of the positive crash dump pairs are correctly identified, then only around 25% of the negative crash dump pairs are incorrectly identified as positive.

ROC Curve Analysis

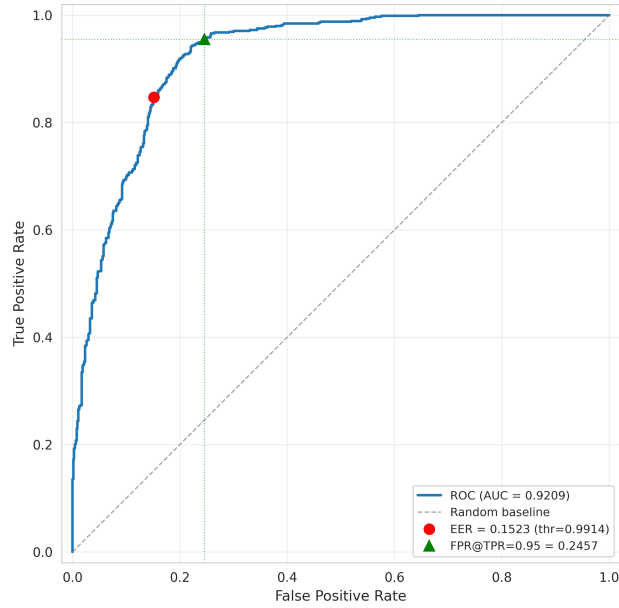
The ROC curve plots the True Positive Rate (TPR) against the False Positive Rate (FPR) for various thresholds. The TPR is the ratio between the correctly classified positive pairs and all positive pairs. The FPR is the ratio between the false positives (the negative pairs that were incorrectly classified as positive) and all negative pairs. The blue line illustrates the model’s performance across all possible cosine thresholds. Key metrics are marked on the curve: the Equal Error Rate (EER) is denoted by a red dot, and the False Positive Rate at 95% TPR is marked by a green triangle.



(a) Baseline ROC Curve
AUC: 0.875



(b) Sym ROC Curve
AUC: 0.9163



(c) SymAsym ROC Curve
AUC: 0.9209

Figure 6.1: Comparison of ROC Curves: (a) Naive ModernBERT-base pooling. (b) Hierarchical MIL architecture with standard triplet loss. (c) Proposed *SymAsym* model with asymmetric attention-conditioned margin. Each model's performance on the True Positive Rate (TPR) vs. False Positive Rate (FPR) across all possible thresholds is visualized.

Evaluating the ROC curves across the three models shows the progressive impact of our architectural and objective function improvements. As shown in Figure 6.1a, the *Baseline* model achieves an ROC-AUC of 0.875. The introduction of file- and bag-level attention pooling combined with the base triplet loss in the *Sym* model yielded clear improvements, increasing the ROC-AUC to 0.9163 as shown in figure 6.1b. Finally, *SymAsym* achieved a maximum ROC-AUC of 0.9209 (Figure 6.1c), showing that the asymmetric, attention-conditioned margin successfully optimizes the model’s overall retrieval capabilities across all possible thresholds.

6.3 *SA-HEM-SA-SupCon* Quantitative Results

The *SA-HEM-SA-SupCon* model includes everything proposed in our pipeline, incorporating both Hard Example Mining (HEM) and Supervised Contrastive Learning. However, since the SupCon phase required the entirety of our small labeled dataset for training (as discussed in Section 4.6.2), we do not have a distinct, unseen validation dataset to perform an unbiased quantitative evaluation. Therefore, the metrics for the *SA-HEM-SA-SupCon* model cannot be directly or fairly compared with the other models (*Sym* and *SymAsym*). Because the model has already seen the evaluation data during its training phase, there is a high risk of data leakage and overfitting.

As shown in Figure 6.2, the model achieves a near-perfect ROC-AUC of 0.975 (Figure 6.2). However, for the reasons stated above, we present the results in Table 6.2 strictly as a theoretical upper bound of the model’s capacity.

Table 6.2: Theoretical Upper Bound. Performance of the *SA-HEM-SA-SupCon* model. **Note:** These quantitative results are highly inflated because the final SupCon stage was trained using the exact same labeled defect dataset used for this evaluation. We therefore treat these metrics strictly as a theoretical upper bound, and prioritize qualitative results from real-world use when drawing conclusions.

Metric	Range	Optimal	<i>SA-HEM-SA-SupCon</i>
ROC-AUC $\uparrow$	[0, 1]	1.0	0.9745
PR-AUC $\uparrow$	[0, 1]	1.0	0.9727
Equal Error Rate $\downarrow$	[0, 1]	0.0	0.0634
FPR @ 95% TPR $\downarrow$	[0, 1]	0.0	0.0768

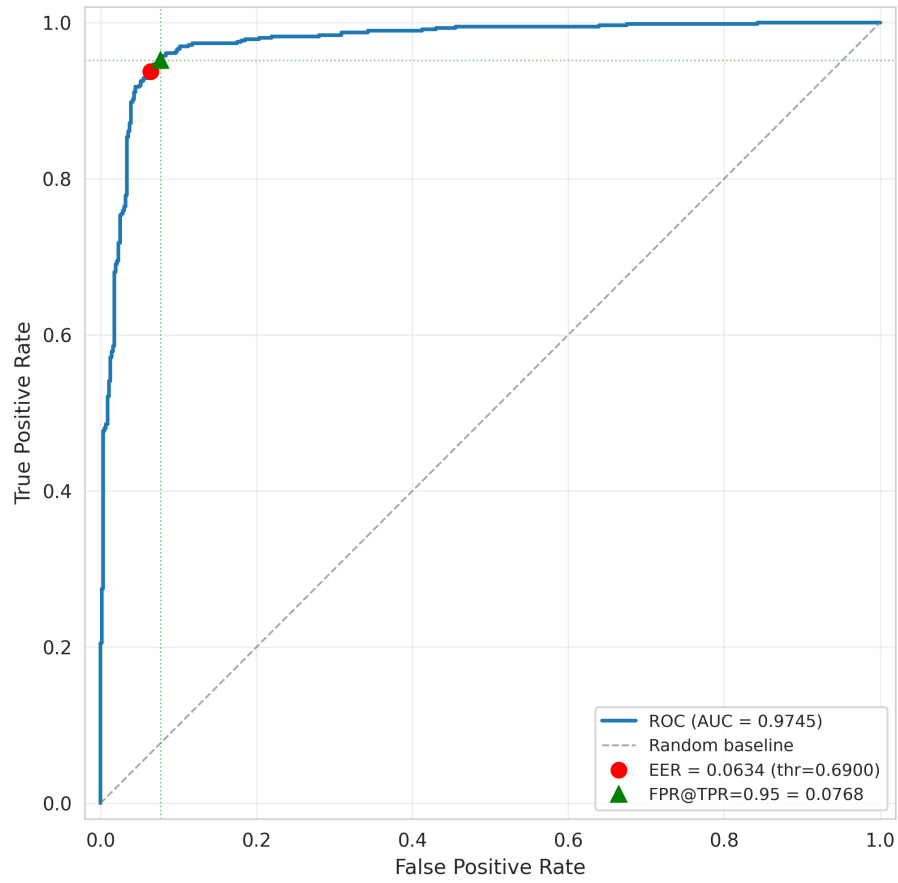


Figure 6.2: SA-HEM-SA-SupCon Model ROC Curve: The SA-HEM-SA-SupCon model achieves a ROC-AUC of 0.9745. Note that this metric is likely inflated due to the lack of an isolated evaluation set.

Chapter 7

Distribution Evaluation

This chapter details the distribution-based metrics used to capture how well the models separate related and unrelated crash dumps, and discusses the distribution results for all models.

7.1 Distribution Evaluation Metrics

While quantitative metrics like ROC-AUC evaluate the model's performance at different thresholds and the model's retrieval capabilities, another way to evaluate the model's performance is by analyzing the probability distributions of the similarity scores.

In this context, a distribution refers to the frequency curve of the cosine similarity scores generated by the model. To visualize and measure how well the model separates unrelated and related data, we had two distributions.

The positive pair distribution represents the similarity scores of all crash dump pairs that shared the same defect. In a well-trained model this distribution should cluster towards 1.0, which is the highest similarity score.

The negative pair distribution represents the similarity scores across all crash dump pairs that did not share the same defect. Ideally this distribution should cluster lower on the similarity scale.

Geometrically, these two distributions can be thought of as two peaks or mountains of data points plotted along an x-axis representing cosine similarity from -1 to 1 . If the model has successfully learned the semantic meaning of crash dumps, these two distributions should pull apart, creating a gap between them.

7.1.1 Mean/Median Gap

Two important distribution metrics were simply the difference between the mean cosine scores of the two distributions and the difference between the median cosine scores of the

two distributions. This roughly shows how different two distributions are and can be used as a simple metric to verify that the model is truly learning our task.

7.1.2 Wasserstein Distance

We also utilized the Wasserstein distance between the distributions, as it reflects the amount of mass that must be moved to make the distributions match. This is a better indicator for model performance than mean or median gap when the distributions are not Gaussian.

More clearly, the Wasserstein distance calculates the minimum cost required to transform one distribution into the shape of the other. The cost is defined as the amount of probability mass moved multiplied by the distance it must travel [1]. The formula is the following:

$$W_1(\mathbb{P}_r, \mathbb{P}_g) = \inf_{\gamma \in \Pi(\mathbb{P}_r, \mathbb{P}_g)} \mathbb{E}_{(x,y) \sim \gamma} [\|x - y\|]$$

- γ represents a specific "transport plan" detailing how to move probability mass from point x to point y .
- $\Pi(\mathbb{P}_r, \mathbb{P}_g)$ is the set of all possible transport plans.
- $\|x - y\|$ is the actual distance the mass has to travel.
- **inf** (infimum) means we are selecting the single most efficient, lowest-cost transport plan out of all the possibilities.

A high-value Wasserstein cost means that the two distributions are very different, requiring massive effort to make them match. A low value means that the distributions are already very similar.

7.1.3 Overlap Coefficient (OVL)

While Wasserstein measures the difference between the distributions, it does not directly quantify how much they overlap. To evaluate this we also utilized an Overlap Coefficient (OVL). The OVL measures the shared area under the two probability density functions for positive and negative pairs. In the context of our thesis, OVL helps us measure how much the distributions have separated so that retrieval methods will be practical. If we have high overlap, the results from a retrieval task will return both negative and positive pairs among the top results.

7.2 Distribution Results and Discussion

This section evaluates how well our proposed models separate positive and negative pairs within the latent space. To measure this, we compare the similarity distributions of the *Sym* and *SymAsym* models against the *Baseline*.

As seen in Table 7.1, the *SymAsym* and *Sym* models both perform better than the *Baseline* model on all metrics. The *Sym* model does, however, have a greater mean and median gap than the *SymAsym* model. This is likely the result of the *SymAsym* model having a few more

Table 7.1: Distribution Metrics. Comparison of model embedding distributions with theoretical optimal values. The vertical bar separates metric metadata from experimental results. **Bold** values indicate the best performance achieved.

Metric	Range	Optimal	Base	Sym	SymAsym
Mean Gap $\uparrow$	[0, 2]	2.0	0.0429	0.2488	0.1914
Median Gap $\uparrow$	[0, 2]	2.0	0.0462	0.2772	0.2351
Wasserstein-1 $\uparrow$	[0, 2]	2.0	0.0429	0.2488	0.1914
Overlap Coefficient $\downarrow$	[0, 1]	0.0	0.3433	0.3025	0.2781

confident false positives than the *Sym* model has. The *Sym* model’s negative pairs are all in a cohesive segment which can be seen in Figure 7.1b.

Despite this, the *SymAsym* model demonstrated superior performance on one of the most critical metrics: the Overlap Coefficient. It reduced the overlap coefficient gap to **0.2781**. A lower overlap coefficient indicates that the distributions of related and unrelated crash dumps are distinct.

Table 7.1 also shows that both self-supervised models have improved mean and median gap metrics for the similarity distributions. These values are however still far below the optimal value of 2. This is expected, as 2 would only be the result if all positive pairs had exactly 1 as similarity and all negative pairs -1 . In essence the same logic also applies to Wasserstein which measures how much area under the curve would need to be moved for the distributions to be identical.

7.2.1 Similarity Distributions

The similarity distribution plot visualizes the overlap of cosine similarities for both positive and negative pairs. The green distribution represents crash dump pairs sharing the same underlying defect, while the red distribution represents negative pairs (unrelated crash dumps). The X-axis denotes the computed cosine-similarity score (ranging between -1 and 1) between pairs, and the y-axis represents the number of pairs falling into that specific similarity range.

In an ideal embedding space, these two distributions would be completely disjoint. The green (positive) distribution would be clustered near 1 , while the red (negative) distribution would be clustered near -1 . However, such a result in a real-world model would be an extremely strong indicator of overfitting. In our case there might also be some negative pairs which are mislabeled and should truly be positive pairs.

Note that the X-axis limits are not fixed from -1 to 1 across all plots. Different models project their embeddings differently, meaning their actual cosine similarity scores often fall within a much smaller, compressed range.

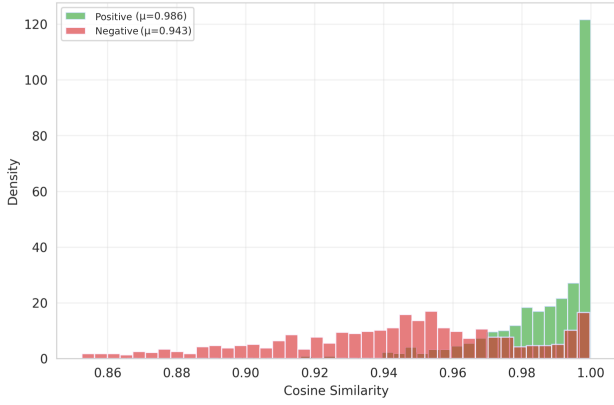
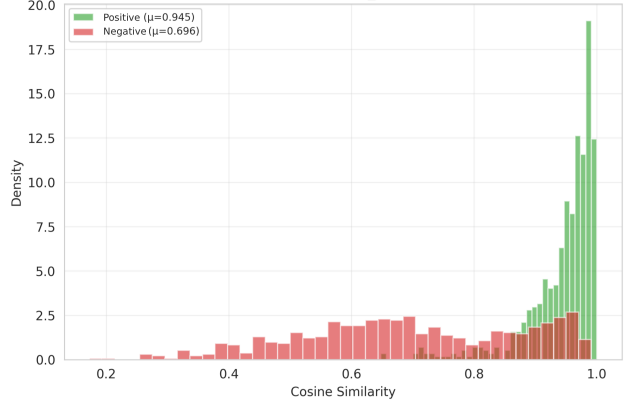
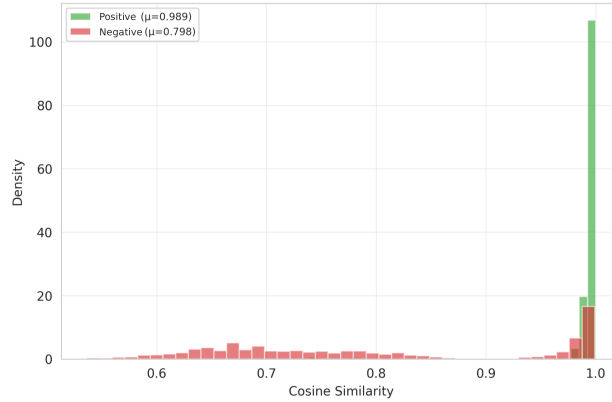
(a) *Baseline* distributions(b) *Sym* distributions(c) *SymAsym* distributions

Figure 7.1: Comparison of Similarity Distributions: Probability density of cosine similarities for positive pairs (same-defect) versus negative pairs (unrelated) across three models. (a) *Baseline* utilizing non-specialized embeddings. (b) *Sym* trained with hierarchical MIL and a standard triplet margin. (c) The proposed *SymAsym* model utilizing the asymmetric attention-conditioned margin.

Analyzing the similarity distributions provides deeper insight into how effectively each model separates positive and negative pairs. In the *Baseline* model (Figure 7.1a), the distributions for positive and negative pairs are distinct enough to serve as a rudimentary retrieval tool. However, a substantial area of overlap remains, indicating that the non-specialized embedding space struggles to enforce strict semantic boundaries between related and unrelated crash dumps.

The *Sym* model demonstrates a visibly clearer separation of these distributions, as seen in Figure 7.1b. This improved distinction over the *Baseline* confirms that integrating a domain-specific ModernBERT encoder with a standard triplet margin loss effectively trains the network to pull related crash dumps closer together while actively pushing unrelated ones apart.

Finally, as illustrated in the similarity distributions for the *SymAsym* model (Figure 7.1c), the proposed architecture exhibits the most refined embedding space. The asymmetric loss function successfully compresses the vast majority of negative pairs down to a cosine similarity of approximately 0.7, creating a distinct gap between the positive and negative distributions. However, a notable cluster of negative pairs still remains at higher similarity scores, resulting in some overlap.

7.3 SA-HEM-SA-SupCon Distribution Results

As discussed in previous chapters, the *SA-HEM-SA-SupCon* model was evaluated on the same labeled dataset used during its supervised training phase. This data leakage prevents a fair comparison with the purely self-supervised models, making any analysis of these distributions statistically invalid as a measure of true generalization.

As shown in Figure 7.2 and Table 7.2, the model achieves near-perfect separation on this memorized dataset. The supervised fine-tuning successfully drives the overlap coefficient down to 0.1089, creating a massive, distinct gap between the positive (related) and negative (unrelated) crash dump pairs.

Table 7.2: Theoretical Upper Bound: Distribution Metrics. Performance of the *SA-HEM-SA-SupCon* model. **Note:** As with the retrieval metrics, these results are highly inflated due to data leakage from the training phase. They serve strictly as a theoretical upper bound.

Metric	Range	Optimal	<i>SA-HEM-SA-SupCon</i>
Mean Gap $\uparrow$	[0, 2]	2.0	0.8701
Median Gap $\uparrow$	[0, 2]	2.0	0.9551
Wasserstein-1 $\uparrow$	[0, 2]	2.0	0.8701
Overlap Coefficient $\downarrow$	[0, 1]	0.0	0.1089

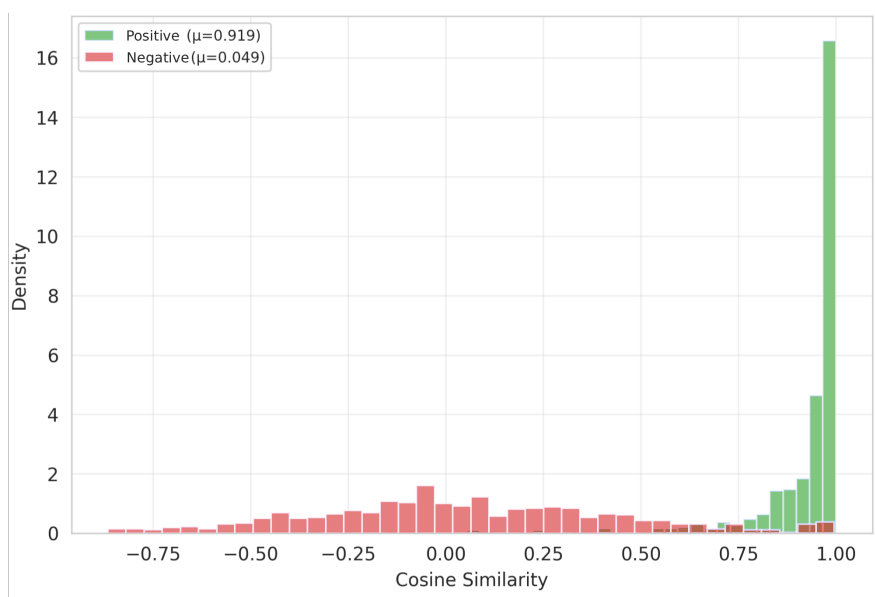


Figure 7.2: SA-HEM-SA-SupCon Model Similarity Distributions: Showcasing the geometric separation between positive (related) and negative (unrelated) pairs after supervised fine-tuning.

Chapter 8

Manual Evaluation

While quantitative and distributional metrics provide a mathematical assessment of model performance, real-world validation should also be used to confirm model performance. This chapter details the manual evaluation methods utilized to validate the model's real-world performance as a retrieval engine for crash dumps. This chapter also introduces the interactive tools developed for this purpose.

8.1 Manual Evaluation Methods

In addition to the mathematical quantitative and distribution metrics mentioned earlier, we also performed manual inspections and expert feedback to verify the model's performance. Note that evaluating the model manually and with expert feedback comes with some bias.

8.1.1 Visual Inspection via K-Nearest Neighbors:

To test the model on existing data, we utilized Atlas [18], an open-source embedding visualization library. It utilizes a density map clustering algorithm [17] to reduce our high-dimensional bag embeddings into an interactive 2D space. This allowed for manual inspection of the local neighborhoods surrounding specific crash dumps. By looking at the k-nearest neighbors around known defects, we could visually verify whether the model successfully grouped semantically related crash dumps together.

8.1.2 OpenSearch

Schneider Electric currently uses OpenSearch to allow manual text search in crash dumps. OpenSearch is a search and analytics engine which provides many useful features for indexing and searching data from various sources. It offers powerful text-search capabilities as well as

support for custom preprocessing pipelines and ML models. OpenSearch also includes a built-in vector database [15], which we utilized to store the model’s embeddings for manual inspection.

8.1.3 Web Interface

To further validate the model and gather feedback from domain experts, we developed an internal web application. Through this interface, it is possible to collect binary feedback per retrieved crash dump regarding whether this result is truly related to the query crash dump.

The interface allowed engineers working directly with the crash dumps to search through the entire database of crash dumps and their model-generated embeddings to retrieve the k-nearest neighbors for any given query crash dump. The website also improved the workflow of analyzing the models as it allows users to view files side by side and inspect the similarity scores. The website also displays the attention values for each file in the attention pooling layer (Background 2.4.5), which helps analyze the model by revealing which files it deems most important.

The website uses OpenSearch to store the preprocessed crash dumps, including each file’s preprocessed text. Both file-level and bag-level embeddings are stored in the OpenSearch index. To compare crash dumps, we use OpenSearch’s built-in vector-based k-NN retrieval algorithm.

Some screenshots of the website are available in Appendix D.

8.2 Manual Inspection Results

In addition to using the quantitative results and the graphs, we also manually inspected groupings created by the model using a custom web server we built.

From this, we noticed some cases in which the model performs well and some cases where it does not. The most problematic cases are when many files are identical or almost identical but some important lines differ, which should make them distinct crashes. In this case the model rather often misclassifies them as being related.

When manually inspecting the results using our web application, we noticed some potential problems. The *SymAsym* model file attentions do not behave as expected. It prioritizes files like analysis report and memory logs over stacktrace and trace log files, which is counter-intuitive and likely an issue with the model and indicates a new shortcut learning method. The *Sym* model struggles with the same problem but to a lesser extent than the *SymAsym* model. We theorize that this is related to the asymmetric attention conditioned loss function described in the Section 4.8.2.

The *SA-HEM-SA-SupCon* suffers from this issue to some extent but appears to weigh file attentions more realistically except in some cases where it has likely slightly overfitted on mapping defects to device IDs.

8.2.1 Manual Inspection of *SA-HEM-SA-SupCon* model

Despite not being able to trust the quantitative metrics described in Chapter 6 at face value, manual inspection indicated that the supervised model’s practical performance was strong.

When deployed via our web application (Section 8.1.3) and queried with unseen crash dumps, the *SA-HEM-SA-SupCon* model outperformed all previous iterations, demonstrating highly cohesive semantic clustering. However, it is important to acknowledge that the current manual evaluation is largely subjective. While the results from the manual inspections are highly promising, we cannot make an entirely objective, quantitative conclusion as to whether this model is better than previous self-supervised ones.

Chapter 9

Conclusion

This thesis investigated the viability of automating the clustering of software crash dumps at Schneider Electric using machine learning. Traditional clustering methods, which rely heavily on lexical matching or hashing, often miss related issues that textually appear different but are semantically similar. To address this we proposed a model that uses a ModernBERT architecture combined with a MIL framework. This section analyzes the results and discusses whether this approach was effective in capturing the true semantic relationships between complex crash dumps.

9.1 Summary of Contributions

The primary goal of this thesis was to address the time-consuming, manual process of grouping crash dumps by their underlying root causes at Schneider Electric. The existing system relied on rudimentary hashing, which fails to capture the semantic meaning of the logs from crash dumps.

To solve this, we developed a self-supervised retrieval pipeline capable of grouping semantically similar crash dumps. Our *SymAsym* model is trained purely on unlabeled crash dumps, requiring no manual labeling of crash dumps to root causes.

This work confirms our hypothesis that fine-tuning a domain-specific ModernBERT model using self-supervised learning within a Hierarchical Multiple Instance Learning (MIL) framework effectively captures the underlying semantic structures of crash dumps. By implementing our asymmetric attention-conditioned margin loss, the model learned to group related crash dumps while pushing unrelated ones apart.

9.2 Key Findings and Technical Insights

We developed several models, most notably our self-supervised *SymAsym* model. Compared against our naive *Baseline* model, which simply pools embeddings from a pretrained, non-specialized ModernBERT model, the *SymAsym* model yielded very promising quantitative results by achieving an FPR @ 95% TPR of 0.2457, which is vastly better than the *Baseline* model's result of 0.4726. Practically, this means that when retrieving 95% of the relevant crash dumps for a given query, our model returns half as many irrelevant results as the *Baseline*.

Additionally, the *SymAsym* model achieves much better ROC-AUC and PR-AUC results when compared to the *Baseline*. In addition to these retrieval metrics, distribution metrics also show improvements, for example, the overlap coefficient has decreased significantly, which means that the separation between the similarity distributions for randomly sampled negative pairs versus positive pairs is now more distinct.

A major technical insight is that the risk of shortcut learning is extremely large when doing self-supervised learning on large and heterogeneous datasets. Our decision to use a loss function based on negative pairs of crash dumps likely increased the risk of shortcut learning further. Our proposed solution for this is using Hard Example Mining (HEM) to effectively make the dataset more homogeneous and thus harder and more informative. When we used HEM after *SymAsym*, the quantitative results became worse than the *Baseline* model. This might be due to the model forgetting how to globally position samples. We thus added a Supervised Contrastive (SupCon) learning stage using our small ground truth dataset. To maximize the dataset size, we did not withhold samples for validation. Results for the supervised model are thus not directly comparable to the self-supervised models, but during manual inspection, the results for the supervised model appear to be better than the best self-supervised *SymAsym* model.

9.3 Limitations

Although our proposed pipeline showed strong retrieval and clustering capabilities, it is affected by several limitations, mainly due to scarcity of labeled data and the complexity of unsupervised learning.

One prominent challenge that limited the performance of our models was the fact that they struggle with cases where only one line differs. In many practical scenarios, two crash dumps may look similar and contain similar system logs, memory states, and so on. However, the models often fail to recognize small but important semantic differences within the stack-trace or trace logs as detecting the large differences is often enough for creating a correct result.

The most significant limitation of this thesis is the scarcity and quality of the human-labeled ground truth data. In addition to the lack of labeled pairs, the existing labels rely on manually created Jira tickets which are susceptible to human error. It is highly probable that the dataset contains unlinked duplicate tickets, meaning our ground truth has missing transitive connections.

Working around limited data with contrastive learning is difficult, as these models heavily rely on diverse positive pairs to create meaningful semantic embeddings. Because training the final *SA-HEM-SA-SupCon* model required the entirety of our labeled defect pairs, we were

unable to create a completely unseen validation set. As a result, we were unable to provide an objective quantitative measure of the final model's performance and capabilities. Instead, we had to evaluate the model using subjective manual inspection. Furthermore, contrastive training would be highly impacted by any incorrectly duplicated tickets as it would push those apart. This is not something we could test well due to training on the entire labeled dataset.

Furthermore, this thesis shows the difficulty of designing a fully self-supervised loss function that can successfully capture deep semantic meaning without relying on shortcut learning. In the *SymAsym* model, the asymmetric attention-conditioned margin loss was built to force geometric separation based on file importance. However, self-supervised objective functions often struggle to differentiate between true semantic meaning and structural predictability. For instance, the model tended to focus its attention on highly repetitive files rather than placing appropriate weight on essential files such as stacktraces.

9.4 Practical Implications for Schneider Electric

If integrated into production, the proposed model offers clear business value for Schneider Electric. Transitioning from the current hashing system to our embedding-based approach allows engineering teams to significantly reduce the manual labor required to group and diagnose software defects.

This utility is most clearly illustrated by our model's retrieval metrics. The *SymAsym* model achieved a False Positive Rate of 0.2457 at a 95% True Positive Rate (TPR). This metric demonstrates that the model successfully retrieves 95% of all historically related crashes while automatically filtering out $\approx 75\%$ of the unrelated crash dumps. If integrated directly into Schneider Electric's Crash Dump Analyzer (CDA), it could greatly improve the workflow by finding likely matches.

The current workflow as described in Section 2.1.1 is problematic mainly due to the time required to map a crash dump to a defect. Time constraints force engineering teams to prioritize labeling crash dumps which cross a frequency threshold as determined by crash ID. The proposed workflow makes it feasible to label a significantly larger number of crash dumps by drastically decreasing the search space by providing the engineer with likely matches.

When using a high cosine similarity threshold, it could also be used as an automatic clustering method as it would allow smaller unimportant changes without creating a fully distinct ID as with the current rudimentary solution.

To prove that this high-dimensional embedding space can be used in a daily engineering workflow, we developed and deployed an interactive web interface (Section 8.1.3). The successful deployment of this interface demonstrates that the current machine learning architecture can be packaged into a time-saving tool.

9.5 Future Work

Although this thesis establishes an effective framework for automating the clustering of crash dumps at Schneider Electric, several areas remain open for future work and improvements.

The following section describes what future work we think would be most interesting and helpful to the field.

9.5.1 Model Architecture and Training Dynamics

Due to having limited computational resources, we could not feasibly attempt training models using encoders with different vocabulary sizes. It would, however, be extremely interesting to find what fertility rate gives the best results or is the most efficient for embedding crash dumps and other data. When selecting our fertility rate, we could not find trustworthy research regarding optimal fertility rates and further research regarding this could be useful for training many models for many different use cases.

Analyzing the extent to which data augmentation impacts performance of models using pair-based loss functions as well as how it impacts generalization and overfitting would also be very interesting. In this case, data augmentation could for example, be done by implementing chunk-level and file-level dropout and training multiple models with different settings and comparing their results. Our self-supervised model *SA-HEM-SA-SupCon* utilized file-level dropout as we theorized that it would decrease the risk of overfitting. Future work specifically comparing different data augmentation methods for unsupervised retrieval systems could likely be highly impactful. Other examples of data augmentation could be different preprocessing, masking or *AB* (Section 4.8) mixing rules.

Research could also be conducted to test if our model would benefit from using transformer-based pooling instead of the attention-based pooling we used. We did some initial testing on this, but due to time constraints and implementation complexity of integrating it with our attention-conditioned loss we decided against using it.

Our HEM method would also benefit from more extensive research on how to optimally select the start and end indices to sample hard examples from. Another interesting and promising option would be to use similarity thresholds instead of index thresholds, as this could decrease the risk of selecting and potentially dispersing true positives.

9.5.2 Data Generation and Supervised Fine-Tuning

A major limitation of our method was the severe lack of human-labeled data. While a larger annotated dataset would improve the model's clustering capabilities, manually labeling crash dumps is extremely time-consuming.

Using only an LLM for clustering or retrieval tasks is infeasible as each new crash dump would require comparing it with other crash dumps on initial indexing due to the lack of an embedding. Future work could however evaluate if Large Language Models (LLMs) could be used to create a labeled dataset.

This could be achieved by prompting an LLM with two crash dumps and instructing it to perform a binary classification on whether they are related, or to generate a continuous similarity score. This could then be used when training the model using Supervised Contrastive learning or similar. Done naively, this labeling task is likely infeasible due to the number of pairs growing quadratically against the number of samples. A potential solution for this is to use Hard Example Mining using an intermediate unsupervised model similar to this thesis. It is also likely that the LLM used must be very large to perform well on huge crash dump comparisons.

9.5.3 Generalization to Other Datasets

Future work could evaluate our self-supervised and supervised training pipelines on publicly available labeled datasets with known ground truth labels. The evaluation must not be only on crash dumps. For example, the pipeline might perform well on tasks such as calculating similarities between books, codebases, or other large collections of text. With modifications to the base encoder and mixing logic, the pipeline could potentially be adapted to compare large images or videos efficiently.

References

- [1] Martin Arjovsky, Soumith Chintala, and Léon Bottou. “Wasserstein Generative Adversarial Networks”. In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, June 2017, pp. 214–223. URL: <https://proceedings.mlr.press/v70/arjovsky17a.html>.
- [2] Adrien Bardes, Jean Ponce, and Yann LeCun. “VICReg: Variance-Invariance-Covariance Regularization for Self-Supervised Learning”. In: *International Conference on Learning Representations*. 2022. URL: <https://openreview.net/forum?id=xm6YD62D1Ub>.
- [3] Iz Beltagy, Matthew E. Peters, and Arman Cohan. *Longformer: The Long-Document Transformer*. 2020. arXiv: 2004.05150 [cs.CL]. URL: <https://arxiv.org/abs/2004.05150>.
- [4] Tri Dao. “FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning”. In: *The Twelfth International Conference on Learning Representations*. 2024. URL: <https://openreview.net/forum?id=mZn2Xyh9Ec>.
- [5] Tri Dao et al. “FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness”. In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo et al. Vol. 35. Curran Associates, Inc., 2022, pp. 16344–16359. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/67d57c32e20fd0a7a302cb81d36e40d5-Paper-Conference.pdf.
- [6] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Ed. by Jill Burstein, Christy Doran, and Thamar Solorio. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: <https://aclanthology.org/N19-1423/>.
- [7] Haixuan Guo, Shuhan Yuan, and Xintao Wu. “LogBERT: Log Anomaly Detection via BERT”. In: *2021 International Joint Conference on Neural Networks (IJCNN)*. 2021, pp. 1–8. DOI: 10.1109/IJCNN52387.2021.9534113.

- [8] Hongcheng Guo et al. “LogFormer: A Pre-train and Tuning Pipeline for Log Anomaly Detection”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 38. 1. 2024, pp. 135–143. DOI: 10.1609/aaai.v38i1.27764.
- [9] Pinjia He et al. “Drain: An Online Log Parsing Approach with Fixed Depth Tree”. In: *2017 IEEE International Conference on Web Services (ICWS)*. Los Alamitos, CA, USA: IEEE Computer Society, June 2017, pp. 33–40. DOI: 10.1109/ICWS.2017.13. URL: <https://doi.ieeecomputersociety.org/10.1109/ICWS.2017.13>.
- [10] Maximilian Ilse, Jakub Tomczak, and Max Welling. “Attention-based Deep Multiple Instance Learning”. In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, Oct. 2018, pp. 2127–2136. URL: <https://proceedings.mlr.press/v80/ilse18a.html>.
- [11] Prannay Khosla et al. “Supervised Contrastive Learning”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 18661–18673. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/d89a66c7c80a29b1bdbab0f2a1a94af8-Paper.pdf.
- [12] Tsung-Yi Lin et al. “Focal Loss for Dense Object Detection”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42.2 (2020), pp. 318–327. DOI: 10.1109/TPAMI.2018.2858826.
- [13] Tomas Mikolov et al. “Distributed Representations of Words and Phrases and their Compositionality”. In: *Advances in Neural Information Processing Systems*. Ed. by C.J. Burges et al. Vol. 26. Curran Associates, Inc., 2013, pp. 3111–3119. URL: https://proceedings.neurips.cc/paper_files/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf.
- [14] Anthony Moi and Nicolas Patry. *HuggingFace’s Tokenizers*. Version 0.13.4. Apr. 2023. URL: <https://github.com/huggingface/tokenizers> (visited on 02/04/2026).
- [15] OpenSearch Project. *OpenSearch*. Opensearch About. URL: <https://opensearch.org/about/> (visited on 01/30/2026).
- [16] Nils Reimers and Iryna Gurevych. “Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks”. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Ed. by Kentaro Inui et al. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 3982–3992. DOI: 10.18653/v1/D19-1410. URL: <https://aclanthology.org/D19-1410/>.
- [17] Donghao Ren, Fred Hohman, and Dominik Moritz. *A Scalable Approach to Clustering Embedding Projections*. 2025. arXiv: 2504.07285 [cs.HC]. URL: <https://arxiv.org/abs/2504.07285>.
- [18] Donghao Ren et al. “Embedding Atlas: Low-Friction, Interactive Embedding Visualization”. In: *2025 IEEE Visualization and Visual Analytics (VIS)*. 2025, pp. 191–195. DOI: 10.1109/VIS60296.2025.00044.

-
- [19] Phillip Rust et al. “How Good is Your Tokenizer? On the Monolingual Performance of Multilingual Language Models”. In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Ed. by Chengqing Zong et al. Online: Association for Computational Linguistics, Aug. 2021, pp. 3118–3135. DOI: 10.18653/v1/2021.acl-long.243. URL: <https://aclanthology.org/2021.acl-long.243/>.
- [20] Florian Schroff, Dmitry Kalenichenko, and James Philbin. “FaceNet: A unified embedding for face recognition and clustering”. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 815–823. DOI: 10.1109/CVPR.2015.7298682.
- [21] Abhinav Shrivastava, Abhinav Gupta, and Ross Girshick. “Training Region-Based Object Detectors with Online Hard Example Mining”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016, pp. 761–769. DOI: 10.1109/CVPR.2016.89.
- [22] Ashish Vaswani et al. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017, pp. 5998–6008. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.
- [23] Benjamin Warner et al. “Smarter, Better, Faster, Longer: A Modern Bidirectional Encoder for Fast, Memory Efficient, and Long Context Finetuning and Inference”. In: *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Wanxiang Che et al. Vienna, Austria: Association for Computational Linguistics, July 2025, pp. 2526–2547. ISBN: 979-8-89176-251-0. DOI: 10.18653/v1/2025.acl-long.127. URL: <https://aclanthology.org/2025.acl-long.127/>.

Appendix A

Experiments

This chapter contains early experiments and the results and insights from them.

A.1 Background

A.1.1 Cross Entropy Loss

Cross entropy loss is commonly used when training classification models where the last layer should output logits indicating which class or classes are the correct ones. To improve convergence and learning for these models, cross entropy penalizes confidently incorrect outputs strongly. For a multiclass classification problem with C classes, the loss L is defined as:

$$L = - \sum_{i=1}^C y_i \log(\hat{y}_i)$$

Where:

- y_i is the ground truth label.
- $\hat{y}_i$ is the predicted probability for class i (usually the output of a Softmax function).

A.2 Bag Cohesiveness Classification

Before implementing the final geometric triplet loss framework (Section 4.8.1), we formulated the problem as a binary classification task. This problem formulation had large problems with shortcut learning and representation collapse. We managed to mitigate the representation collapse problems but could never solve the shortcut learning for this problem.

These early experiments did, however, give us many insights about shortcut mitigation methods and express the importance of formulating a fitting objective function.

A.2.1 Task Formulation

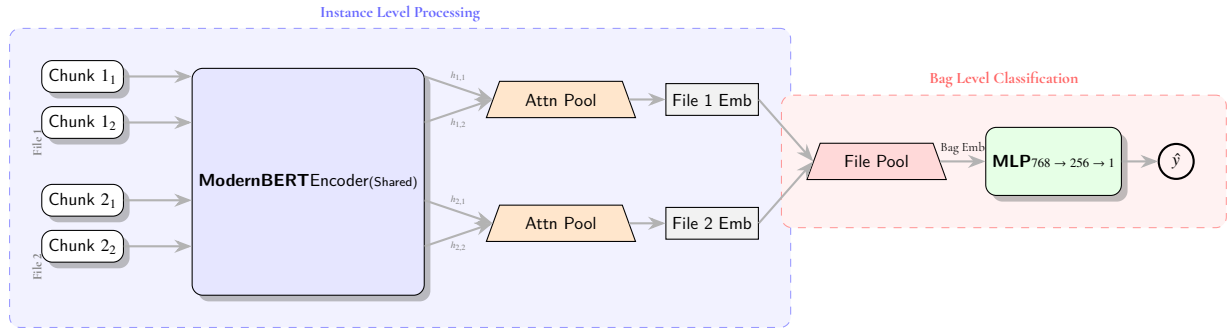


Figure A.1: Architecture of the MIL model, showcasing the whole architecture of the model.

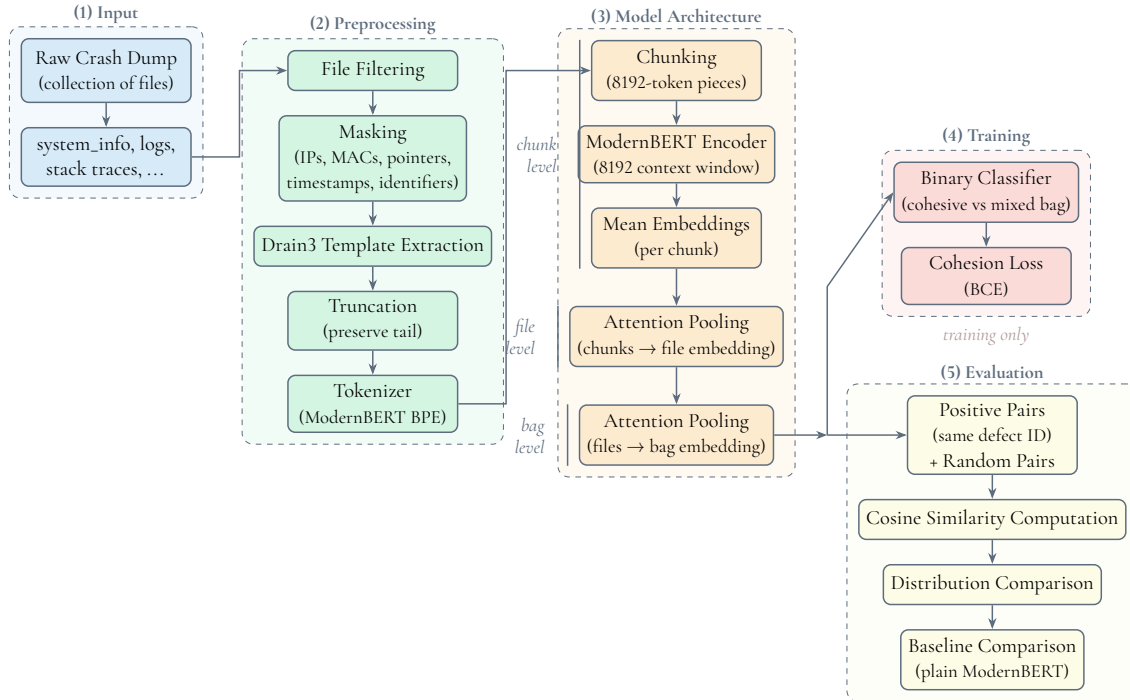


Figure A.2: Crash dump preprocessing, ModernBERT embedding model, training path, and evaluation pipeline.

The initial hypothesis was that semantic representations suitable for clustering and retrieval could be learned by training the model to classify between cohesive and non-cohesive crash dumps. Within a Multiple Instance Learning (MIL) framework, each crash dump was treated as a bag of files.

A cohesive bag was defined as being a crash dump with no additional files, while a non-cohesive crash dump contained files from two distinct crash dumps. The files in a non-cohesive crash dump were randomly drawn from both source crash dumps with some restrictions. The model architecture for the pure MIL method can be seen in Figure A.1.

The training pipeline can be seen in Figure A.2. The model was trained with batches containing an equal amount of cohesive and non-cohesive bags. Early experiments of this method randomly sampled how many files would be drawn from each initial bag, but this was changed to taking an equal amount from both.

As in our *SymAsym* model, files were segmented into chunks and encoded using the ModernBERT model, at this point, we still used the pretrained ModernBERT model. The chunk-level embeddings were then attention-pooled into a file embedding, and file embeddings were then pooled into a single bag embedding. This hierarchical structure is identical to that used in the *SymAsym* model.

The bag-level embedding was then passed to a small classification head ($768 \rightarrow 1$), and the model was optimized using binary cross-entropy loss.

A.2.2 Implementation Challenges

The implementation of this approach required us to solve some implementation challenges to enable training with only 32 GB of VRAM. As backpropagation during training requires large computational graphs, and we encode chunks iteratively, we were required to use gradient checkpointing in the chunking step and limit the number of chunks per forward pass. This memory optimization is still used in the *SymAsym* model.

Initially we also had some issues with using float16, which resulted in unstable training and NaN activations, potentially as a result of using it with Flash Attention. Instead, we switched to bfloat16, which resolved the issue.

Another issue was that the classification head produced very large gradient norms that destabilized training. This was mitigated by introducing layer normalization and increasing batch size. To increase batch size, we had to further vectorize the forward pass to avoid memory fragmentation that caused crashes in long training runs.

A.2.3 Problems

Even when the model successfully achieved high classification accuracy, it failed to create an embedding space suitable for clustering and retrieval. When analyzing the reasons behind this, we discovered that the likely cause was shortcut learning. Instead of learning the complicated root causes behind the failures, the model used package versions, operating system information and project names to create a grouping that could be used to solve the objective.

Some of the most prominent shortcut behaviors were:

- **Operating System:** When creating non-cohesive bags that were mixes of bags from two different operating systems (Linux and Windows), the model quickly learned to read the OS from system info. It also learned to identify OS from differences in how paths are written and the structure of files.
- **Temporal Signals:** File attention analysis revealed that the model heavily favoured the uptime file. The likely reason is that the model learned that how long the server that created the uptime file had been running strongly correlated with the masked timestamps in the rest of the files and more importantly the lengths of the rest of the files. When the model discovered inconsistencies between the uptime and the file lengths in a bag, it classified it as non-cohesive.

An additional issue was that we sometimes mixed very similar bags. When combining two similar bags, the model would potentially correctly classify them as cohesive, which incurs a large loss. As our small dataset is manually labeled, there is a very large risk that some positive pairs are missed, making it dangerous to punish the model for pairing similar pairs. To mitigate this problem, we introduced a Levenshtein similarity safeguard that prevents mixing of crash dumps that are very similar. The Levenshtein filter did, however, not fully solve this problem as it is very rudimentary and likely fails in many cases.

Dataloader

We tested some different dataloader setups, one was where we created fully random mixes of crash dumps. Another where we only mixed crash dumps if they were from the same OS. Additionally, we used the Levenshtein similarity metric to avoid mixing crash dumps with a similarity ratio above 0.8.

A.2.4 Lessons Learned

Although the classification loss decreased steadily and accuracy improved, the distance between embeddings of semantically related crash dumps and unrelated crash dumps decreased over training, collapsing to a single mass. The model effectively learned to solve the classification task while producing embeddings that were not useful for similarity-based retrieval.

This outcome shows a key limitation of binary classification objectives for representation learning. They do not have an explicit requirement to preserve meaningful geometric relationships between instances. These results directly motivated our pivot to a metric-learning method based on geometric constraints, leading to our asymmetric attention-conditioned triplet margin loss described in Section 4.8.1.

Appendix B

Hyperparameters

This Appendix contains all relevant hyperparameters for *SymAsym* and *SA-HEM-SA-SupCon*.

B.1 *SymAsym*

Table B.1: Configuration parameters for dataloading, batching, and crash dump bag generation.

Parameter	Value
num_bags	5000
files_per_bag	8
max_num_chunks	500
max_chunks_in_batch	1500
file_similarity_threshold_pct	0.8
min_cooccurrence_rate	0.05
regenerate_bags_between_epochs	True
triplet_batch_size	16
triplet_num_mixes	12
triplet_max_files_from_a	4
triplet_max_files_from_b	4
use_cross_batch_triplets	True

Table B.2: Configuration parameters for the triplet loss, attention conditioning, and VICReg regularization.

Parameter	Value
triplet_margin	0.05
triplet_enforce_diff	True
use_asymmetric_triplet	True
att_cond_warmup_epochs	2
att_cond_ramp_epochs	1
att_cond_alpha_ramp_epochs	3
att_cond_lambda_max	1.5
att_cond_alpha_max	0.5
att_cond_gamma_max	0.6
att_cond_entropy_k	5.0
att_cond_H0	1.5
att_cond_tau	0.05
att_cond_scale	1.5
att_cond_conf_min	0.2
embedding_variance_weight	0.05
embedding_covariance_weight	0.01
att_reg_entropy_weight	0.01
att_reg_consistency_weight	0.01

Table B.3: Configuration parameters related to hardware utilization, mixed precision, and optimization limits.

Parameter	Value
use_flash_attention	True
use_fp16	True
use_gradient_checkpointing	True
chunks_per_forward	16
max_epochs	500
max_grad_norm	10.0
weight_decay	0.01
scheduler_warmup_pct	0.005
early_stopping_patience	5000
early_stopping_min_delta	0.001
seed	42

B.2 *SA-HEM-SA-SupCon*

This model reused the *SymAsym* as its base and then trained further with the following hyperparameters split by stage (HEM or SupCon).

Table B.4: Configuration parameters for dataloading, batching, and crash dump bag generation for *SA-HEM-SA-SupCon* (HEM vs. SupCon).

Parameter	HEM	SupCon
train_split	0.9	0.9
run_validation	True	True
num_bags	5000	5000
files_per_bag	8	8
max_num_chunks	500	500
max_chunks_in_batch	1500	1500
batch_size	64	64
file_similarity_threshold_pct	0.8	0.8
min_cooccurrence_rate	0.05	0.05
regenerate_bags_between_epochs	True	True
use_os_filter	False	False
max_files_per_dump	null	null
file_drop_rate	0.0	0.0
p_drop_other	0.1	0.1
min_files_after_drop	4	4
always_keep_files	trace.log	trace.log
stacktrace_group_files	stacktrace*.txt	stacktrace*.txt
stacktrace_keep_strategy	priority	priority
journal_group_files	journal*.txt	journal*.txt
journal_keep_strategy	priority	priority

Table B.5: Model and architecture hyperparameters for *SA-HEM-SA-SupCon* (HEM vs. SupCon).

Parameter	HEM	SupCon
max_length	8192	8192
embedding_dim	768	768
projection_dim	128	128
projection_hidden_dim	1024	1024
normalize_embeddings	True	True
attention_dropout	0.0	0.0
classifier_hidden_dim	256	256
classifier_pooling	mean	mean
use_file_transformer	True	True
file_transformer_heads	4	4
file_transformer_layers	1	1
file_transformer_ff_dim	2048	2048
file_transformer_dropout	0.1	0.1
file_transformer_gate_init	-5.0	-5.0
imposter_probability	0.0	0.0
imposter_mean	4.0	4.0
imposter_std	0.0	0.0
freeze_encoder	False	False

Table B.6: Hardware utilization, mixed precision, checkpointing, and training limits for *SA-HEM-SA-SupCon* (HEM vs. SupCon).

Parameter	HEM	SupCon
use_flash_attention	True	True
use_fp16	True	True
use_gradient_checkpointing	True	True
chunks_per_forward	16	16
max_epochs	500	500
max_grad_norm	10.0	10.0
weight_decay	0.01	0.01
scheduler_warmup_pct	0.005	0.005
early_stopping_patience	5000	5000
early_stopping_min_delta	0.001	0.001
seed	42	42
save_every_n_epochs	1	1
save_every_n_batches	1000	1000
log_every_n_steps	10	10

Table B.7: Stage-dependent hyperparameters that differ between HEM and SupCon for *SA-HEM-SA-SupCon*.

Parameter	HEM	SupCon
learning_rate_encoder	$5 \cdot 10^{-7}$	$2 \cdot 10^{-7}$
learning_rate_attention	$4 \cdot 10^{-4}$	$2 \cdot 10^{-4}$
learning_rate_classifier	$3 \cdot 10^{-4}$	$1 \cdot 10^{-4}$
learning_rate_projection	$4 \cdot 10^{-4}$	$2 \cdot 10^{-4}$
embedding_variance_weight	0.05	0.005
embedding_covariance_weight	0.01	0.001
use_cross_batch_triplets	True	False
use_contrastive_finetuning	False	True

Table B.8: Optimization and Triplet-related hyperparameters for *SA-HEM-SA-SupCon*.

Parameter	HEM	SupCon
learning_rate_encoder	$5 \cdot 10^{-7}$	$2 \cdot 10^{-7}$
learning_rate_attention	$4 \cdot 10^{-4}$	$2 \cdot 10^{-4}$
learning_rate_classifier	$3 \cdot 10^{-4}$	$1 \cdot 10^{-4}$
learning_rate_projection	$4 \cdot 10^{-4}$	$2 \cdot 10^{-4}$
learning_rate_file_transformer	$5 \cdot 10^{-5}$	$5 \cdot 10^{-5}$
use_cross_batch_triplets	True	False
triplet_batch_size	16	16
triplet_num_mixes	12	12
triplet_margin	0.05	0.05
triplet_enforce_diff	True	True
triplet_max_files_from_a	4	4
triplet_max_files_from_b	4	4
triplet_similarity_threshold	0.9	0.9
use_hard_neg_mining	True	True
hard_neg_rank_min	200	200
hard_neg_rank_max	500	500
hard_neg_fraction	0.8	0.8

Appendix C

Tables of Figures

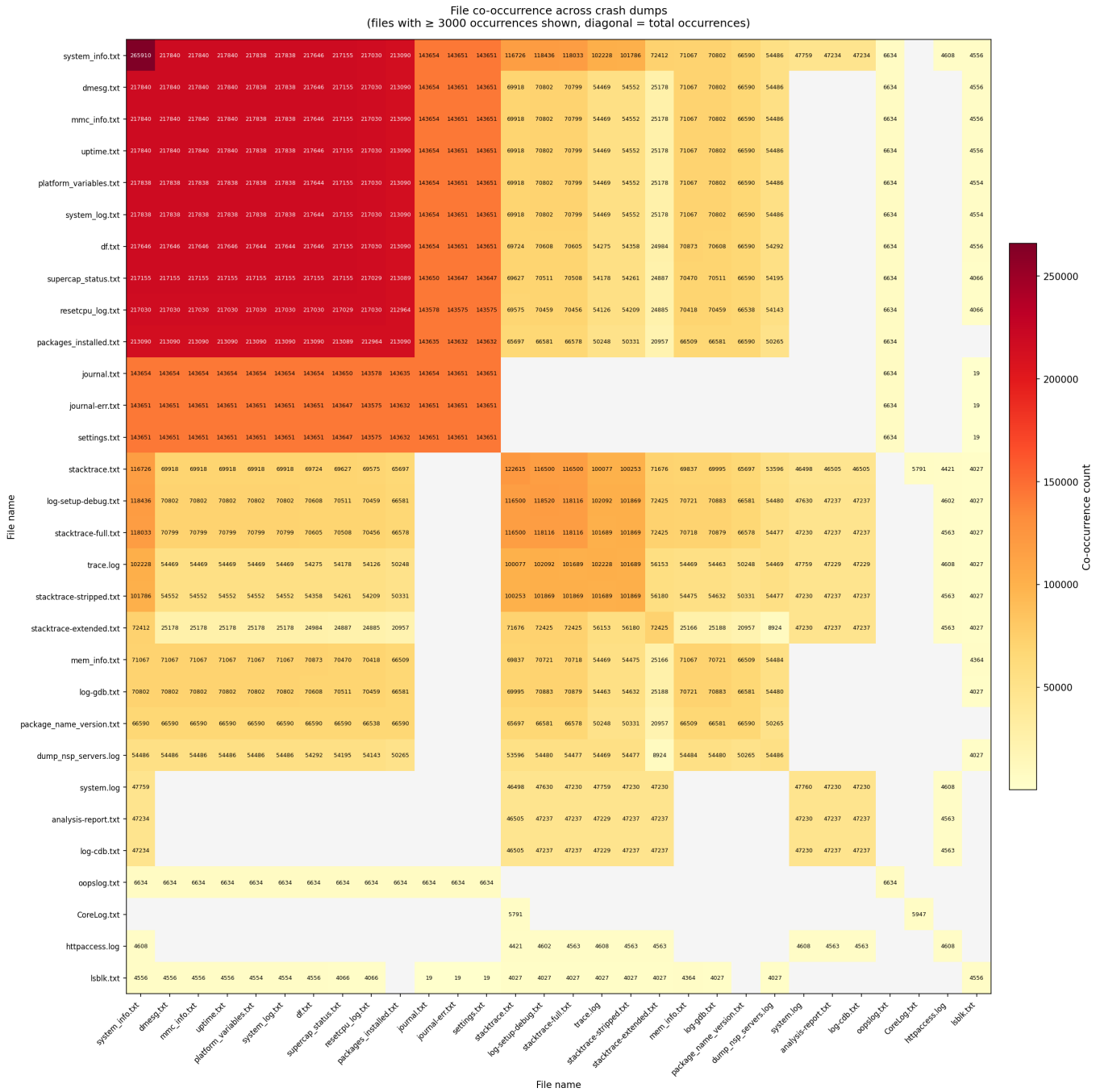


Figure C.1: Co-occurrence heatmap of files that appear in at least 3,000 crash dumps. The matrix includes files all above the cutoff as long as they were not manually excluded (30 distinct file types). The heatmap visualizes how frequently each pair of files appears together within the same crash dump. Darker cells indicate more frequent co-occurrence, while lighter cells indicate rarer combinations.

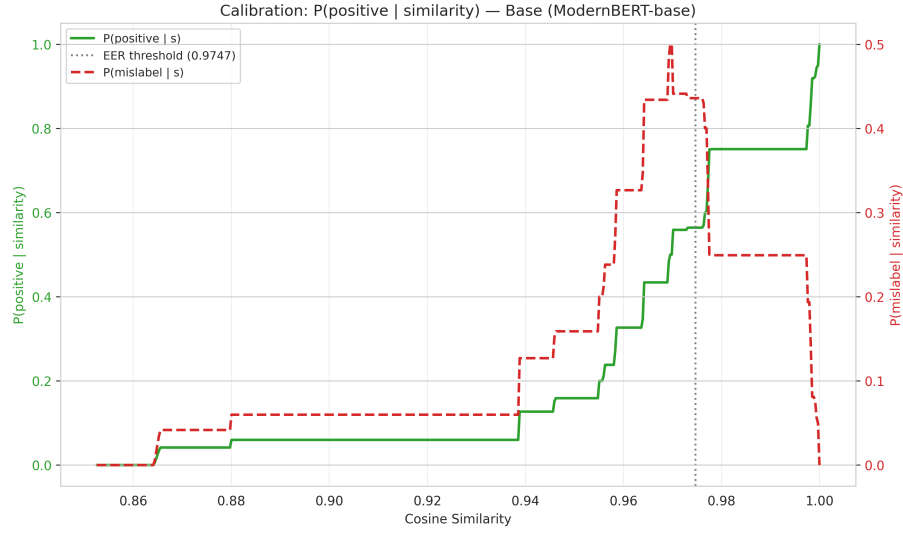


Figure C.2: Base Model Threshold Analysis: The posterior probability for the naive mean pooling method. The broad transition region where they are almost inverses of each other suggests a high degree of overlap between random and related crash dumps, making a reliable operational threshold difficult to define.

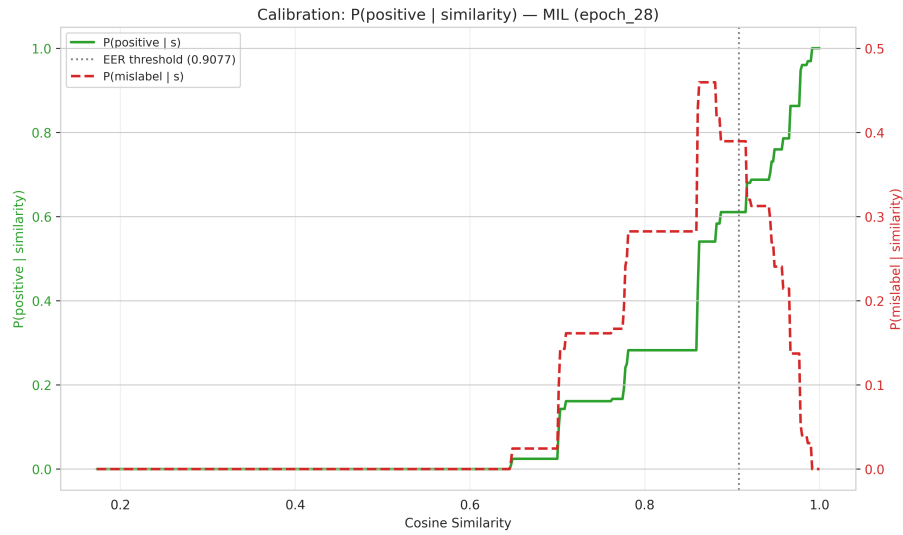


Figure C.3: Sym Model Threshold Analysis (Epoch 28): The transition from negative to related pairs becomes significantly sharper than the *Baseline*. This indicates that the hierarchical MIL architecture has begun to isolate semantic features. The operational threshold is lower than that of the *SymAsym* model, suggesting that the embedding space is not yet as compressed as the final state.

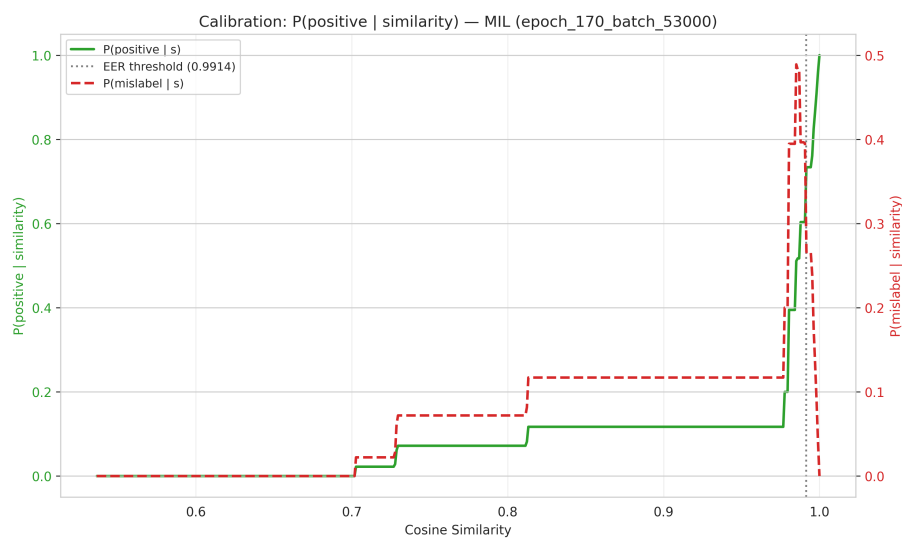


Figure C.4: SymAsym Proposed Model Threshold Analysis: This plot illustrates the empirical probability of a pair being semantically related given its cosine similarity score. The steep curve indicates a highly concentrated embedding space, typical of Transformer-based architectures. The vertical dotted line explicitly identifies the operational Equal Error Rate (EER) threshold at a cosine similarity of 0.9914, providing a concrete cutoff for retrieval applications.

Appendix D

Website

This Appendix provides some screenshots of the custom website created to make using the model for evaluating crash dumps easy.

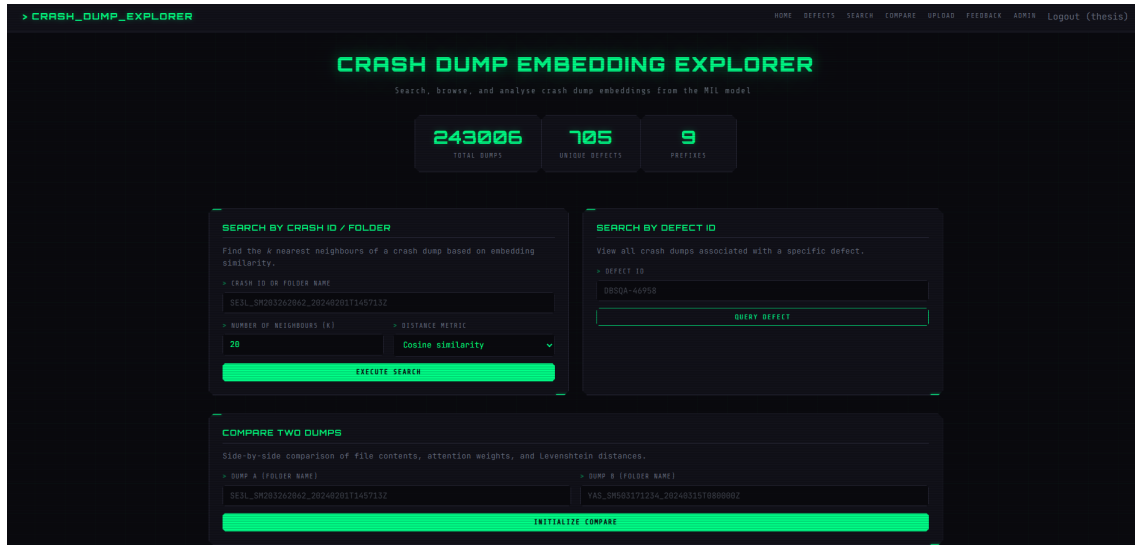


Figure D.1: This is the initial view when visiting the website.

QUERY: YAS_3N164032005_20240409T151830Z

YAS **LINUX** **DB5QA-43640** 18 FILES

TOP 20 NEAREST NEIGHBOURS **COSINE**

#	SIMILARITY	NAME	PREFIX	OS	FILES	DEFECT	FILE ATTENTION	COMPARE
1	0.999943	YAS_SM202112118_20250331T115346Z	YAS	Linux	18	DB5QA-43640	package_name_version.txt(0.3137), ...	COMPARE
2	0.999937	YAS_SM192912078_20260122T072523Z	YAS	Linux	18	DB5QA-43640	package_name_version.txt(0.3117), ...	COMPARE
3	0.999922	YAS_SM201552236_20250225T104652Z	YAS	Linux	18	DB5QA-43640	package_name_version.txt(0.3074), ...	COMPARE
4	0.999898	YAS_SM181432347_20240813T120218Z	YAS	Linux	18	DB5QA-43640	system_info.txt(0.2807), package_n...	COMPARE
5	0.999886	YAS_SM190852069_20251211T092932Z	YAS	Linux	18	DB5QA-43640	package_name_version.txt(0.3156), ...	COMPARE
6	0.999864	YAS_SM181432347_20250221T140613Z	YAS	Linux	18	DB5QA-43640	system_info.txt(0.2810), package_n...	COMPARE
7	0.999857	YAS_SM203532026_20240621T013816Z	YAS	Linux	18	DB5QA-43640	package_name_version.txt(0.2821), ...	COMPARE
8	0.999848	YAS_SM192212046_20240617T155030Z	YAS	Linux	18	DB5QA-43640	system_info.txt(0.2853), package_n...	COMPARE
9	0.999839	YAS_3N174442207_20250407T170340Z	YAS	Linux	18	DB5QA-43640	system_info.txt(0.2822), package_n...	COMPARE
10	0.999838	YAS_SM200142002_20241115T133451Z	YAS	Linux	18	DB5QA-43640	package_name_version.txt(0.3158), ...	COMPARE
11	0.999836	YAS_SM204132042_20250224T072813Z	YAS	Linux	18	DB5QA-43640	package_name_version.txt(0.3096), ...	COMPARE
12	0.999835	YAS_3N174642155_20250407T194217Z	YAS	Linux	18	DB5QA-43640	system_info.txt(0.2697), package_n...	COMPARE
13	0.999833	YAS_SM204722025_20241115T133451Z	YAS	Linux	18	DB5QA-59272	system_info.txt(0.2705), package_n...	COMPARE
14	0.999832	YAS_SM201552236_20241124T161622Z	YAS	Linux	18	DB5QA-43640	package_name_version.txt(0.3072), ...	COMPARE
15	0.999825	YAS_SM202122219_20240713T104314Z	YAS	Linux	18	DB5QA-43640	package_name_version.txt(0.3107), ...	COMPARE

Figure D.2: A screenshot of the k-NN view from a specific crash dump. Here it is possible to see that it correctly retrieves many of the same defect (green) but has one likely false positive (row 13).



Figure D.3: A screenshot of the file attention comparison between the crash dumps and their overall similarity measure.

PER-FILE COMPARISON								
FILE	Q	T	EMB COSINE	EMB EUCLIDEAN	Q ATTN	T ATTN		
df.txt	•	•	0.9986	3.79	0.51%	0.52%	DIFF	
dmesg.txt	•	•	0.9945	2.98	0.55%	0.55%	DIFF	
dump_nsp_servers.log	•	•	0.9475	8.98	5.91%	5.8%	DIFF	
log-gdb.txt	•	•	0.9766	5.99	2.13%	2.58%	DIFF	
log-setup-debug.txt	•	•	0.9788	6.69	0.68%	0.63%	DIFF	
mem_info.txt	•	•	0.9991	1.17	1.0%	0.99%	DIFF	
mmc_info.txt	•	•	0.9520	8.58	1.64%	1.38%	DIFF	
package_name_version.txt	•	•	0.9483	9.59	28.97%	25.46%	DIFF	
packages_installed.txt	•	•	0.9980	1.74	1.54%	1.47%	DIFF	
platform_variables.txt	•	•	0.9955	2.62	4.05%	4.41%	DIFF	
resetcpu_log.txt	•	•	0.9638	7.54	4.27%	3.67%	DIFF	
stacktrace-full.txt	•	•	0.9981	3.91	4.35%	5.42%	DIFF	
stacktrace-stripped.txt	•	•	0.9999	0.33	14.29%	12.8%	DIFF	
supercap_status.txt	•	•	0.9842	4.93	1.3%	1.32%	DIFF	
system_info.txt	•	•	1.0000	0.01	30.26%	27.85%	DIFF	
system_log.txt	•	•	0.9584	8.73	1.32%	1.4%	DIFF	
trace.log	•	•	0.9218	18.95	1.51%	1.2%	DIFF	
uptime.txt	•	•	0.8678	14.29	3.57%	3.25%	DIFF	

Figure D.4: More information dense view showing cosine similarities and attentions between files with the same type between the two dumps.

FILE CONTENTS

df.txt CODE: 0.9586 TOGGLE SIDE-BY-SIDE

Filesystem	Size	Used	Available	Use%	Mounted on
/dev/root	471.3M	159.2M	276.9M	37%	/
devtmpfs	249.8M	0	249.8M	0%	/dev
tmpfs	258.8M	0	258.8M	0%	/dev/shm
tmpfs	258.8M	292.0K	240.7M	0%	/run
/dev/mmcblk0p7	2.5G	504.9M	1.9G	20%	/var/sbo
/dev/mmcblk0p1	181.6M	4.1M	168.2M	2%	/boot
/dev/mmcblk0p5	177.7M	12.3M	151.4M	8%	/var/Log

Filesystem	Size	Used	Available	Use%	Mounted on
/dev/root	471.3M	147.2M	288.9M	34%	/
devtmpfs	249.6M	0	249.6M	0%	/dev
tmpfs	249.8M	0	249.8M	0%	/dev/shm
tmpfs	249.8M	344.0K	249.5M	0%	/run
/dev/mmcblk0p1	181.6M	4.2M	168.1M	2%	/boot
/dev/mmcblk0p7	2.5G	482.2M	1.9G	20%	/var/sbo
/dev/mmcblk0p5	177.7M	32.4M	131.3M	20%	/var/Log

Figure D.5: Example of the file diff view.

EXAMENSARBETE Hierarchical Self-Supervised Learning for Semantic Embeddings of Crash Dumps**STUDENTER** Danny Tang, Ludvig Lindholm**HANDLEDARE** Görel Hedin (LTH) Emil Eriksson (SE) Jörgen Malmberg (SE)**EXAMINATOR** Niklas Fors (LTH)

Att lära en AI förstå krascher. Ny metod hittar rätt bugg bland miljontals krascher!

POPULÄRVETENSKAPLIG SAMMANFATTNING Danny Tang, Ludvig Lindholm

När ett dataprogram kraschar skapas en rapport, som kan innehålla flera filer som beskriver detaljer från kraschen. Att manuellt sortera dessa för att hitta det faktiska felet är ett enormt pussel. Genom att använda tekniker inom språkteknologi har vi skapat ett AI-system som automatiskt läser och grupperar kraschrapporter som beror på samma underliggande buggar.

Tänk dig att du får in tusentals felrapporter varje dag från maskiner över hela världen, din uppgift är att lista ut vilka av dessa krascher som beror på samma underliggande fel. Detta är vardagen på ett företag som Schneider Electric.

Idag på Schneider Electric använder man oftast enkla metoder som letar efter exakta kopior bland rapporterna. Problemet är att om en enda liten siffra eller tidsstämpel ändras tror det nuvarande systemet att det är en helt ny bugg. Resultatet blir att hundratusentals krascher hamnar i en osorterad hög, och ingenjörer måste slösa tid för att utreda och gruppera samma fel om och om igen.

I detta examensarbete har vi tagit fram ett smartare sätt. Vi har tränat en AI-modell baserad på moderna språkmodeller. Målet är att träna modellen så att den kan läsa hela kraschrapporter och förstå deras underliggande betydelse, istället för att stirra sig blint på exakta ord. AI:n omvandlar varje kraschrapport till en matematisk vektor och krascher som beror på samma grundfel får liknande matematiska vektorer. En väldigt stor ut-

maning i projektet var bristen på ett facit. Av alla miljontals krascher fanns det enbart ett fåtal fall där vi kunde koppla kraschen till en specifik bugg. Därför byggde vi ett självlärande system. AI:n fick titta på enorma mängder data och på egen hand lära sig att känna igen mönster och gruppera liknande fel, utan något riktigt facit.

På Schneider Electric är en kraschrapport ofta extremt lång. Vi lät AI:n läsa texten i kraschrapporterna i mindre delar, ungefär som att läsa en bok ett kapitel i taget, och sedan sammanfatta hela boken. Vi tränade också modellen på ett sådant sätt att den lär sig vilka delar som är viktigast. Precis som vissa kapitel i en bok är mer avgörande för handlingen, lärde sig modellen vilka delar av kraschrapporten som var viktigast. Resultatet blir en matematisk sammanfattning där det verkliga felet är i fokus.

I framtiden kan denna AI-lösning fungera som en smart sökassistent, vilket gör så att ingenjörer lättare och snabbare kan hitta krascher som kommer från samma underliggande fel.