IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 25 December 2024, accepted 6 February 2025, date of publication 14 February 2025, date of current version 20 February 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3542566

RESEARCH ARTICLE

Optimizing Pre-Trained Code Embeddings With Triplet Loss for Code Smell Detection

ALİ NİZAM[1], ERTUĞRUL İSLAMOĞLU[2], ÖMER KEREM ADALI[3], AND MUSA AYDIN[3]

[1]Software Engineering Department, Fatih Sultan Mehmet Vakıf University, 34445 İstanbul, Türkiye
[2]Artificial Intelligence and Data Engineering Department, Fatih Sultan Mehmet Vakıf University, 34445 İstanbul, Türkiye
[3]Computer Engineering Department, Fatih Sultan Mehmet Vakıf University, 34445 İstanbul, Türkiye

Corresponding author: Ali Nizam (ali.nizam@fsm.edu.tr)

This work was supported by the Scientific and Technological Research Council of Türkiye (TUBITAK) under Grant 123E020.

ABSTRACT Code embedding represents code semantics in vector form. Although code embedding-based systems have been successfully applied to various source code analysis tasks, further research is required to enhance code embedding for better code analysis capabilities, aiming to surpass the performance and functionality of static code analysis tools. In addition, standard methods for improving code embedding are essential to develop more effective embedding-based systems, similar to augmentation techniques in the image processing domain. This study aims to create a contrastive learning-based system to explore the potential of a generic method for enhancing code embedding for code classification tasks. A triplet loss-based deep learning network is designed to optimize in-class similarity and increase the distance between classes. An experimental dataset that contains code from Java, Python, and PHP programming languages and 4 different code smells is created by collecting code from open-source repositories on GitHub. We evaluate the proposed system's effectiveness with widely used BERT, CodeBERT, and GraphCodeBERT pretrained models to create code embedding for the code classification task of code smell detection. Our findings indicate that the proposed system may offer improvements in accuracy, an average of 8% and a maximum of 13% for models. These results suggest that incorporating contrastive learning techniques into the generation process of code representation as a preprocessing step can enhance performance in code analysis.

INDEX TERMS Code embedding, contrastive learning, triplet loss, code smell detection.

I. INTRODUCTION

Code embedding transforms code into distributed vector representations for source code analysis [1] and is employed in a variety of software engineering tasks such as code summarization and semantic labeling [1], code clone detection, the identification of bugs, the evaluation of software quality, the detection of code smells, the review of compiled code, and other related tasks [2], [3].

Discovering effective embedding is an important and novel research area. Software engineering researchers have developed several models of source code depending on various implementations that use a syntactic tree representation such as paths in the abstract syntax tree (AST) [4], [5], or process the code directly using a large language model (LLM) such as Bidirectional Encoder Representations from Transformers (BERT) [6], or specialized LLMs for understanding code semantics, such as CodeBERT [7], GraphCodeBERT [8] or low-level code structure such as compiler intermediate representations [9].

Various optimization techniques are used to enhance embedding performance. Fine-tuning has become the predominant optimization method in LLM studies for tailoring pre-trained models to specific tasks and enhancing their performance across various natural language processing applications. Although effective, fine-tuning the pre-trained parameters involves high resource usage with a huge computational cost [10]. Hyperparameter optimization represents another prominent approach to the refinement and tuning of the hyperparameters of

The associate editor coordinating the review of this manuscript and approving it for publication was Juan Wang.

© 2025 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License.
For more information, see https://creativecommons.org/licenses/by/4.0/

language models to optimize their performance on specific tasks [2].

Augmentation techniques are also proposed to generate and learn more effective representations [11]. However, they have not been fully explored in source code modeling and embedding optimization [12]. Preprocessing systems based on refactoring can impact code semantics. In addition, these methods employed to optimize code embeddings provide only modest performance gains. Augmentation improved accuracy in problem classification and bug detection, averaging 0.9% with base refactoring and 2.07% with enhanced augmentation, while some cases exceed these averages [13].

We aim to enhance code embedding by improving the distribution of embedding classes with a triplet loss-based deep neural network (DNN) to address the challenges above. It differs from the previous works that focused on designing a new code embedding architecture that creates a specific representation. We mainly explore the potential of the triplet loss (TL) technique from contrastive learning as a generic approach to enhance the performance of source code pretrained embeddings. The effectiveness of the proposed approach was validated using enhanced embeddings in the code classification task for detecting code smells.

Code smell indicates design flaws and violations of fundamental design principles in the source code that make software difficult to evolve, understand, and maintain. Detecting code smells is a crucial research area aimed at enhancing code quality. Despite the considerable advances made by existing approaches in detecting code smells, several challenges remain [14]. Firstly, the existing detection approaches rely heavily on fine-tuning pre-trained language models built on large-scale datasets. Fine-tuning and data collection are time-consuming and effort-driven processes. Secondly, they require various code metrics, and subjective factors can influence these metrics' selection. Finally, the traditional methodologies employ word vectors derived from code text data without fully leveraging the insights gleaned from the pretrained language models (PLMs). The optimal approach to code smell detection presents a significant challenge due to the inherent reproducibility issues [15].

Previous research has explored metric, machine-learning (ML), and DNN-based techniques for identifying and classifying code smells in source code. Code metrics designers cannot predict all code variations affecting metric definitions. Thus, ML and DNN have emerged as a particularly effective approach for code smell classification, identification, and detection [16], [17]. However, ML techniques have drawbacks such as the reliability of code features that require parsing the entire project and the dependability of some metrics [15]. The development of a classifier that is capable of effectively detecting different code smells has not yet been achieved [18]. In addition, code smell detection is typically regarded as a supervised learning problem requiring the availability of huge, labeled datasets; however, such comprehensive datasets are lacking [19], [20].

Many existing studies focus on a certain context, specific programming language, or even a particular smell of code type [16]. In addition, some studies on code smell detection have developed specific systems or DNNs configured to detect individual smell types, allowing the analysis of single smell [20], [21]. Some approaches can not be applied to code smells that do not rely on source code metrics [22]. Our methodology employs a unified system that analyzes five code classes from three programming languages, including four distinct code smell types and one without any smell.

The inputs to our model are a method-level code snippet sourced and collected from GitHub using the SonarCloud tool. The corresponding smell tags are also retrieved from SonarCloud. The output of the proposed classification system is the smell type of the method. We preferred multiclass classification that provides to detect a wide range of code smell [17]. We applied our preprocessing technique to existing PLMs to generate code embeddings. Based on our aim, the definitions of research questions are:

A. RQ1

How much triplet-based contrastive learning can improve the classification performance of code-embedding vectors compared to other optimization methods? We compared hyperparameter optimization with contrastive learning regarding their ability to enhance code embeddings. A comprehensive analysis of triplet generation methods, DNN structures, and parameters was conducted.

B. RQ2

Is it possible to generalize the triplet-based embedding enhancement method to improve the performance of different embeddings? Our objective is to examine the generalization capabilities of triplet loss-based optimization, hereafter referred to as triplet optimization (TO). We conducted a comparative analysis of the performance of the proposed model with base versions of BERT, CodeBERT, and GraphCodeBERT in Java, Python, and PHP programming languages to evaluate improvements and answer research questions. In addition, we analyzed the system performance when all languages were combined into a single dataset.

The following is a summary of our contributions: (1) We constructed a code smell dataset by collecting methods with code smells from Java, Python, and PHP languages using the SonarCloud tool (2) Hyperparameter optimization for pretrained embedding models is implemented for code smell detection, fine-tuned with our source-target pair dataset, (3) Optimizing embedding using triplet loss to increase the accuracy of code smell detection. In addition, we analyze the source of advantages offered by the proposed triplet loss methods. This study stands out from many others by utilizing a general embedding optimization system that supports multiple programming languages, PLMs, and various code classes (smell types).

The remainder of this paper is structured as follows: Section II presents an overview of related work, including an examination of pretrained code embedding generation techniques, embedding optimization, contrastive learning, and their application in code smell detection. Section III outlines the methodology employed for classifying code smell with triplet loss-based network and DNN. Section IV presents the result of the proposed model on the dataset. Section V discusses the efficacy of the proposed triplet approach and explains its underlying rationale. Section VI interprets the result with an assessment of its future application.

II. BACKGROUND AND RELATED WORKS

This section provides an overview of the background and applications of code embedding optimization, with a comparison of several existing approaches. In addition, we explore deep learning (DL) applications to understand how they are utilized for addressing issues related to code quality, code embedding optimization, and code smell detection.

A. CODE EMBEDDING TECHNIQUES

The application of ML and DL to code analysis is a relatively new research domain, requiring the transformation of code into a numerical representation. Code embeddings represent code snippets as numerical vectors in a continuous space while preserving semantic meaning. In early applications, code embedding techniques or models based on Word2Vec [23] treated source code as a series of textual tokens [1]. Word2Vec organizes the vocabulary as a Huffman binary tree with shorter binary codes assigned to more frequent words, and this approach reduces the number of evaluated output units and complexity [23]. Word2Vec can create high-quality word vectors; however, there are differences between software source code and natural language [1]. The code must adhere to the formal syntax and semantics to be executed [24].

Some code embedding techniques utilize graph embedding, which involves transforming a graph into a low-dimensional space while ensuring the preservation of the graph data [25]. These methods used the AST, data, or control flow graph of code. AST represents the complex structure of the source code using different semantic units [26]. Alon et al. [4] proposed Code2Vec that decomposed the tokens of code components into AST path collections, a DNN assigned weights to each path and aggregated the paths to create embeddings. PathPair2Vec [27] used AST with a short path concept. Code2Seq [28]employed the decoder architecture of transformers [29] to select the relevant paths.

Pre-training is another technique that facilitates the training of larger and more effective models [30]. In early natural-language-based pre-trained models, code-specific characteristics may not be properly considered, such as the syntactic and semantic structures and their inadequate performances [31]. Recently, with the emergence of large-scale PLMs, the various models for source code analysis and development have been proposed and significantly outperformed previous models [11]. RNN [32] and transformers-based [29] systems are employed for code embedding-based DL solutions. The advantages of the transformer structure have increased research efforts in this area. Many pretrained code representation models are successfully utilized in code analysis such as CodeBERT [7], and GraphCodeBERT [8]. In addition, recently developed text-based methods such as BERT [6] based on transformers have been shown to provide successful results in code representation. Eventually, researchers developed several pre-trained source code models using transformers. The semantic structure of software code and applications comprises structured combinations of reserved words and identifiers [1].

B. EMBEDDING OPTIMIZATION TECHNIQUES

The fine-tuning process adapts pretrained models to specific tasks to improve code embedding performance by arranging the network weights. Pretraining creates representations to extract universal code properties, and then fine-tuning adapts embeddings to various downstream applications [33]. The transfer learning and multi-task learning with fine-tuning strategies outperform the single-task-based models across all tasks. Different methods based on feature extraction are also used for embedding optimization. Preprocessing eigenvectors with the k lowest eigenvalues before using them as positional encodings is another method to obtain a structure-aware transformer [34]. Some works proposed information revealed from supplementary resources such as program test cases to explore the dynamic of programs and integrate it into the feature representations of code as supplementary elements [35] or code history. In addition, contrastive learning has been employed for code analysis tasks such as code retrieval and summarization [36].

C. CONTRASTIVE LEARNING

Contrastive learning extracts meaningful representations by contrasting positive and negative instance pairs. Its main principle is to position similar instances closer together in a learned embedding space, and dissimilar instances further apart. The method maximizes agreement between differently augmented views of the same sample using a contrastive loss in the latent space [37]. Contrastive learning is widely applied in various domains, such as visual representations of face detection [38] and the improvement of sentence embeddings by combining unsupervised contrastive learning and supervised learning [39]. FaceNet was an early implementation of contrastive learning, utilizing a novel online triplet mining method to optimize embeddings by generating triplets of approximately aligned, similar, and non-similar face matches, thereby enhancing face recognition performance [40].

Triplet loss, a contrastive learning technique originating from image processing, is widely applied in code analysis tasks such as code clone detection and augmentation. Various studies have used contrastive learning in code analysis

using supervised and unsupervised methods. Supervised contrastive learning utilizes labeled data to train models explicitly to increase the distance between similar and dissimilar instances using a trained model on pairs of data points with their labels. For unsupervised training, Bui et al. [36] proposed a self-supervised contrastive learning framework that generates multiple versions of a code without changing its semantics to learn without using labeled data. To generate the enhanced representation of code embeddings, the method applied two randomly selected transformation operators to produce distinct transformed code snippets, which are then processed by the same encoder.

Contrastive learning is well-suited to code search because its learning objective is simultaneously separating negative query-code pairs and pulling together positive pairs [11]. Using a mixture sampling strategy during the training phase to obtain hard negative samples supports the selection of data that is challenging for existing models [41]. Another use of contrastive learning is the augmentation task to learn better representations [11]. Zhang et al. [37] used hierarchical features from code to enhance code search performance, instead of increasing the amount of training data. This approach involved the direct input of a positive or negative sample pair into the model by exploiting hierarchical features and reorganizing original training data during training into hierarchical-uncorrelated feature pairs based on hierarchical features. Li et al. [11] proposed to create representation-level augmentation by employing InfoNCE loss to maximize the mutual information between positive pairs. Fan et al. [41] used a mixture sampling strategy to obtain hard negative samples for training to avoid the influence of noisy synthetic data. The code snippets and their generated queries were closely and diversely matched with hard negative samples. Together, these steps ensured that the model reached a reasonable level of convergence, even in the presence of noise.

Triplets were used in code searches to augment the code samples without changing the original semantics. The in-batch augmentation method used in-batch data, where a query and a randomly sampled code were accepted as dissimilar and forced away. It only created dissimilar pairs in a mini-batch, which ignores augmenting similar pairs for learning positive relations. Positive examples were added by rewriting queries [42]. Another augmentation approach was to generate more positive pairs instead of searching for equivalences. The compiler transformations were applied to unlabeled code to create different variants with equivalent functionality [43]. Data augmentation methods often suffer from inefficiency since they require substantial resources to generate data and process it through the large language model [44] and models must embed the data again for the augmented data [11]. To address this, representation-level augmentation created negative samples by choosing interpolation and stochastic perturbation and augmenting the original queries [11].

D. CODE SMELL DETECTION APPROACHES

Conventional techniques for detecting and classifying code smells utilize software metrics, manually defined code metrics based on static rules and thresholds, ML, and DL. Recently, structural code metrics have become a common input for learning models. However, code metrics represent only structural features, and information about the deeper semantic features is often insufficient [17].

Recently, several studies used DL to detect code smells. A CNN-based system achieved 95-97% accuracy for the detection of brain class and brain method smells on thirty open-source Java projects [45]. Barbez et al. [46] employed historical values of source code metrics to increase the precision for detecting Super Class anti-smell, and the performance was evaluated on three software systems. It outperforms existing static machine-learning classifiers by analyzing evolving metrics over time when changes are applied to the system. Another study utilized diverse measurement criteria and DNN systems for various types of smell in Java [20]. In this study, distance metrics with word2Vector embeddings were used for *feature envy* detection, achieving an average F1 score of 51.91%, and *lines of code*, *lack of cohesion of methods cohesion,* and *class cohesion*metrics with DNN network for long method smell detection, with an average accuracy 76.35%

The utilization of PLM for the analysis of code smells is becoming increasingly popular. The embedding generated by pre-trained models can be used alone or combined with other code metrics. An ML model was used to detect smells employing static analysis tools to extract structural features and a BERT model to extract textual features [47]. Modular approaches integrated pretrained models in a smell detection pipeline and each of these representations were trained with ML classifiers [48] or DNNs [17]. For example, the DistilBERT embeddings representing classes and methods, combined with numerical metrics, were used to feed into a convolutional neural network (CNN) to identify long parameter lists and switch statement smells [49]. Transformers also combined unsupervised semantic feature learning to detect several code smells [50]. Pretrained embedding models CodeT5 and CuBERT were used to detect *feature envy* and *data class* code smells by encompassing the metrics extracted using static analysis tools [15].

Some of the code smell detection work has also attempted to increase the performance of embedding, such as prompt learning with CodeBERT [51] and fine-tuning of RoBERTA [52]. Prompt Smell constructed the input of LLMs to detect *long parameter lists*and *long method smells* by combining code snippets with natural language prompts and mask tokens [14]. However, this approach necessitates incorporating supplementary textual data into the code as inputs of a dual-stream model. Another approach used graph data with graph network and code metrics as sequence data employing the transformers to learn interrelationships from

code metrics [22]; however, this method was only effective for smells based on code metrics.

Complex methods have high cyclomatic complexities complex conditional, feature envy, and multifaceted abstraction smells. Sharma et al. [21] implemented a transfer learning system for detecting each complex method smell type separately, with an accuracy rate of approximately 60%, using a network architecture comprising CNN, long short-term memory networks (LSTM), and a densely connected classifier network.

Our literature review revealed that a significant number of existing methods only detect one or a few code smell types. In addition, some methods require the output from static analysis tools or supplementary prompt information. Thus, enhancing PLMs is crucial for better code analysis and smell detection. This study aims to develop a high-performance model that relies entirely on information extracted from the code and can detect multiple code smells simultaneously.

III. METHODOLOGY

This section presents an overview of the research methods, including a description of the data-gathering process, the code vectorization method, and the DNN architecture. Fig. 1 describes the architecture of the proposed system. Firstly, we generate a vector-based numerical representation of the code using code embedding techniques to prepare code for the DNN classifier. Subsequently, we developed a triplet loss-based DNN system to optimize embeddings by bringing instances of the same class closer together while pushing instances of different classes farther apart. We then used a dense neural network to classify both the original and enhanced embeddings and compared their respective performances.

To establish a baseline for comparison, we initially determined the hyperparameters of the classifier using the embeddings without TO. After that, the most suitable triplet selection method was identified using the classifier. We evaluated the system's performance and the resulting improvement for each smell type and language.

A. DATA COLLECTION

We collected a source code dataset from open-source projects including the source code and smell information to train and evaluate our model using the SonarCloud tool from GitHub. The steps of data collection:

- The SonarCloud REST API was employed to generate the datasets necessary for examining the problematic code. This API enabled the identification of problematic code instances across various code repositories, their analysis according to specified parameters and filters, and their transfer to the examination environment in JSON format.
- The selected REST API parameters are *component key* (repository name), *ps* (number of smells for each request), *p* (number of requests), *language* (programming language), *resolution type*, *rules* (smell type), and *additional fields* (the details about return values).

To assess the quality of the collected data, we investigated random samples. GitHub fork and branch operations create multiple instances of the same code snippet. We eliminated the duplications and used a single representative example in the training process. After the elimination, the remaining number of method-level smell examples was relatively reduced.

B. PRETRAINED CODE EMBEDDING MODELS

PLMs have been preferred as the code representation method because they work without requiring comprehensive parsing operations and code compilation. We examined the transformers-based BERT model to enable an analysis of code text with pretraining approaches. CodeBERT extends the BERT structure by separating code and comments. GraphCodeBERT, advancing the CodeBERT technique, enables the investigation of the inherent structure of code by using control flow information. We assessed the effectiveness of the proposed embedding improvement method by applying it to text-based, text-and-code-based, and code-specific optimization techniques and comparing the results.

1) BERT

BERT pre-trains deep bidirectional representations from an unlabeled text by joint considering on both left and right context in all layers to improve unidirectional standard language models [6]. Pre-trained representations reduce the need for task-specific architectures that are heavily engineered. BERT convert input vectors $[C_{[CLS]}, C_1, C_1, \ldots, C_{[SEP]}]$ to $[T_{[CLS]}, T_1, T_1, \ldots, T_{[SEP]}]$ outputs after passing through a number of transformer encoder layers [53]. Input vectors are formed through the element-wise summation of token, segment, and position embeddings. BERT attention layer enriches each token embedding vector. It handles complex and higher-order relationships with relevant relational and contextual information by comparing each token in the sequence with all the others. BERT masks a token from the input sequence and asks the model to guess. The model can simultaneously use a masked token's left and right contexts to make predictions.

The unlabeled data over different pre-training tasks determines the model's parameters during pre-training. In the self-supervised learning phase, the word is substituted with the [MASK] token in 80% of cases, replaced by a randomly selected word 10% of the time, and left unaltered in the remaining 10%. The BERT model is initially set with pre-trained parameters for fine-tuning, and all parameters are subsequently refined using labeled data from downstream tasks, such as sentence pairs in a token sequence. A *sentence* can be any contiguous text, rather than an actual linguistic phrase. In many embedding techniques, only sentence embeddings are conveyed to downstream tasks. In contrast, The BERT model transfers all parameters to initialize

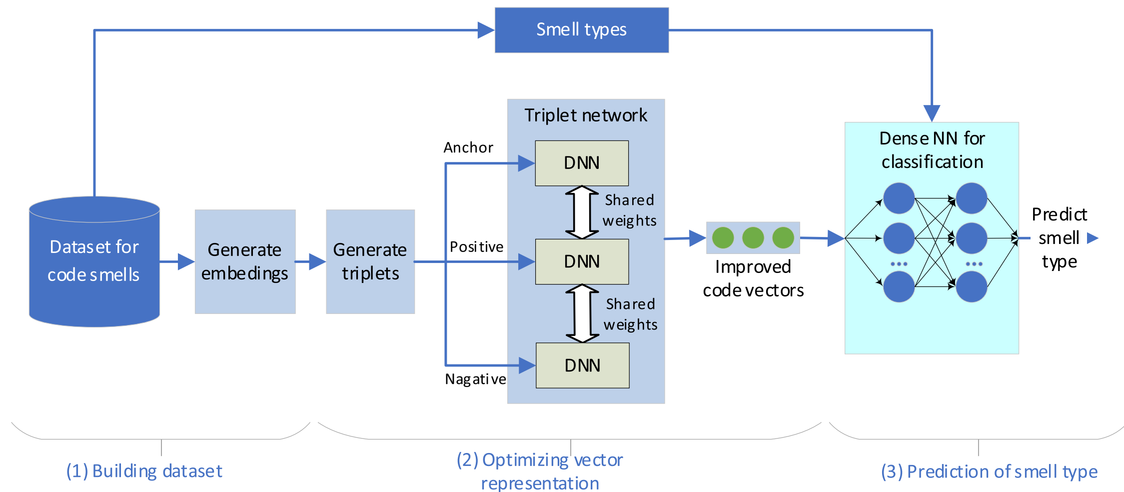


FIGURE 1. **The representation of the proposed system model.**

domain-specific task model parameters. BERT uses special tokens to provide additional information about the input sequence.

- *Word Token*: These are the actual words or sub-words in the input text. They are tokenized using the *WordPiece* tokenizer, which breaks down words into smaller units and represents them as a vocabulary of sub-word tokens.
- *Special classification token* [CLS]: This token is added to the beginning of each input sequence to obtain a fixed-size representation of the entire sequence for downstream tasks such as classification or regression.
- *Special Separator Token* [SEP]: This token separates and distinguishes two sequences when multiple sequences are provided as input. It indicates the end of one sequence and the beginning of another.

For tasks involving multiple sequences, such as question answering or natural language inference, each sequence is assigned a unique segment ID usually 0 or 1.

The model applies N transformer layers over the input vectors to produce code embedding $H^n = transformers_n\left(H^{n-1}\right), n \in [1, N]$ where H^{n-1} refers to nth layers input and $(n-1)$th layers input. The BERT architecture is formulated standard transformers [29]. Equations are as follows:

$$G^n = \mathrm{LN}(\mathrm{MultiAttn}(H^{n-1}) + H^{n-1}) \quad (1)$$

$$H^n = \mathrm{LN}(\mathrm{FFN}\left(G^n\right) + G^n) \quad (2)$$

where G^n is the multi-headed self-attention layer's output where MultiAttn is the multi-headed self-attention mechanism, FFN is a two layers feed forward network, and LN is the layer normalization operation.

2) CodeBERT

CodeBERT uses a hybrid objective function that incorporates the pre-training task of replaced token detection to capture the semantic connection between natural language and programming language [7]. It implements the masked language modeling and replaces token detection functions with a substantial amount of unimodal data [7]. CodeBERT takes the comment, source code, and the set of variables as the sequence of the input X and then converts the sequence into input vectors H^0. Source code $C = \{c_1, c_2 \ldots c_n\}$, comments $W = \{w_1, w_2 \ldots w_n\}$, and variable sequence matrix $V = \{v_1, v_2 \ldots v_n\}$ are concatenated to input as $X = \{[\mathrm{CLS}], W, [\mathrm{SEP}], C, [\mathrm{CLS}], V\}$.

3) GraphCodeBERT

GraphCodeBERT is a pre-trained model that considers both the inherent structure of code and its textual content. Instead of taking the syntactic-level structure of code like AST, it leverages semantic-level data flow information for pre-training [8]. Data flow is a graph including nodes for variables and edges for the flow of data between variables using data assignment [8]. GraphCodeBERT was implemented using BERT and the multi-layer bidirectional Transformer [29]. GraphCodeBERT takes and converts comments, source code, and the set of variables to the input vector. In addition, multi-headed self-attention is computed using V and a special position embedding for all variables to indicate that they are data flow nodes.

The last layer of the GraphCodeBERT network performs additional pooling and normalization (LN). Our experiments show that using the last hidden layer output gives better results for some smell types. Thus, H^n, was also used in the classification process. The typical output of GraphCodeBERT is expressed as the GraphCodeBERT pooler.

C. ENHANCING EMBEDDINGS USING TRIPLET LOSS

Triplet Loss uses a DNN trained to learn how to optimize the embedding itself. Since triplet loss is used mostly for image similarity, usually CNN networks are preferred [40]. However, we used a multi-layer dense network since code

similarity would be analyzed as shown in Fig. 2. Our experiments show that three layers were sufficient for adequate performance. The first layer is designed to adapt different embedding output sizes to the classifier network. An input with N features is represented in Fig. 2. Gemm stands for General Matrix Multiplication which is represented as a Linear layer in PyTorch. The model consists of 3 layers with *Leaky ReLU* activation, containing 1000, 500, N respectively. We preserved the embedding structure by making the size of the last layer the same as the embedding.

Triplet-loss uses a special loss function as follows:

$$\text{Loss} = \max(d(E_{\text{anchor}}, E_{\text{positive}}) - d(E_{\text{anchor}}, E_{\text{negative}}) + m, 0) \quad (3)$$

E represents embedding and d is the distance function. We employed Euclid distance in experiments. m is a margin value to keep negative samples apart.

Generating all possible triplets leads to quadratic or cubic growth in the number of examples, making it infeasible to process them all. To ensure fast convergence, quality, and diversity, creating triplets and selecting the most distinctive triplet pairs is crucial. Different methods were used to develop triplets in augmentation without changing the original semantics.

The main triplet sampling strategies are nucleus, negative, and random. *Nucleus sampling*, or top-p sampling, uses a threshold to restrict the sampling to the most probable tokens with a cumulative probability less than the specified threshold. *Negative sampling* filters negative samples while filtering samples that are identical to the positive samples [41]. *Random sampling* samples negative codes with equal chance as a uniform distribution. Despite its simplicity, the uniform negative has proven effective in enhancing retrieval performance, especially when combined with other negative sampling strategies [54]. However, its effectiveness may be limited if the negative codes are too dissimilar to the query. Incorporating cloned code snippets into the negative samples can reduce model accuracy. To overcome this, similarity metrics were employed to detect and remove cloned code snippets from the negative samples [41].

We selected code embeddings of the same smell type as anchor-positive and embeddings of different smell types as negative in a triplet sample. The triplet creation process is examined under two main categories: online and offline. In the offline approach, anchor, positive, and negative examples are fed into the system using three identical networks with shared weight as shown in Fig. 2. In online mining, the loss function of a single network calculates the loss using similar and dissimilar examples in the same batch.

All triplets are generated before training begins in the offline random sampling strategy. In experiments involving a limited sample size, we observed a marginal decline in performance, suggesting that the insufficient sample number hindered the achievement of optimal results. Consequently, the sample size was expanded by generating an n-ary Cartesian product across smell types using the following equation:

$$\text{CE}_1 x \text{CE}_2 \ldots x \text{CE}_n = \{(\text{ce}_1, \text{ce}_2 \ldots \text{ce}_n) : \text{ce}_i \in \text{CE}_i \text{ for every } i \in \{1, 2, \ldots n\}\} \quad (4)$$

samples using index sets as $(\text{CE}_1 x \text{CE}_2 \ldots x \text{CE}_n)_i, i \in \text{random}\,(1, 2..n)$ where n is the total size of cartesian set. We generate pairs by selecting indexes instead of directly creating pairs, allowing the Cartesian multiplication to produce a significantly large number of samples without compromising performance.

In online triplet selection, a network is fed using standard mini batches [40]. The loss function is responsible for the hard triplets' selection within the mini-batch. Negative samples close to the anchor and positive samples far away are called "hard" because they are difficult to distinguish. When the positive samples are selected close (inside a margin) and the negative samples are selected at a distance, a zero-value triplet loss is obtained, resulting in small gradient values and slow convergence. Conversely, the opposite way of selection (distant positive and close negative) creates a high triplet loss value leading to large gradient values and an update of model parameters [55]. The selection strategies:

- Batch hard triplet mining: Calculates the triplet loss for each anchor-positive pair in a batch using only the nearest negative.
- Batch random hard negative: Involves randomly creating triplets from anchor, positive, and negative examples within a batch, calculating the triplet loss for all combinations, and discarding triplets with zero loss.
- Batch semi-hard triplet mining: Involves randomly selecting triplets where the negative example is closer to the anchor than the positive example but is still within the margin. The margin parameter defines the minimum acceptable distance between anchor-positive and anchor-negative pairs. This approach permits the model to use examples that are difficult to learn, yet not the most difficult during training.

An inappropriate triplet selection strategy may result in inefficient training or, more seriously, model collapse in which all embeddings converge the same value [55].

D. CLASSIFIER ARCHITECTURE

Fig. 3. represents our DNN architecture for classifying the generated embeddings from various models. $B(\ldots)$ and $C(\ldots)$ represent the input and output dimensions of the relevant level. The first layer adapts embedding output sizes to the classifier network. An example input with 128 samples and 768 features for CodeBERT is represented in Fig. 2. The model consists of 4 layers with *Leaky ReLU* activation, containing 256, 128, 128, and 5 neurons, respectively. We added a dropout function before the final layer to reduce overfitting and improve the model's generalization ability. The number of layers and neurons was chosen empirically until a satisfactory performance was achieved.

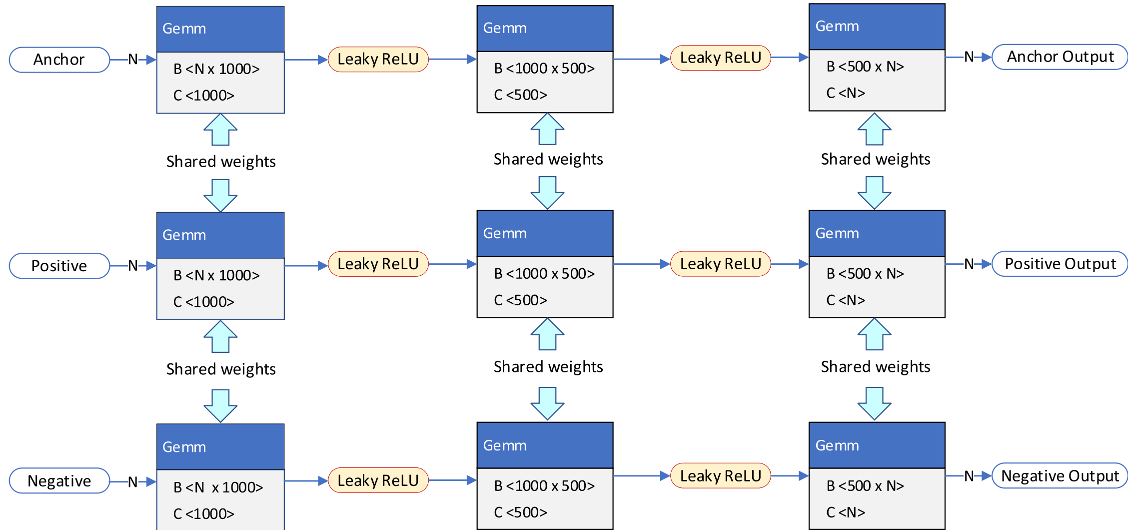


FIGURE 2. Triplet loss network structure (Offline triplet mining).

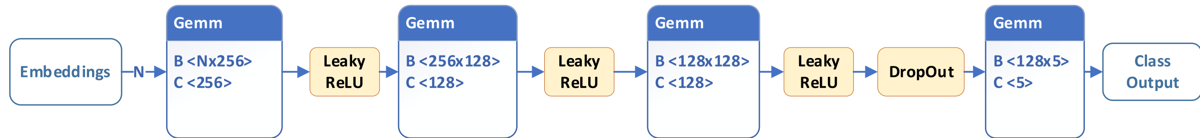


FIGURE 3. DNN architecture for classification.

Using the same classifier model for different embeddings enables a fair comparison of their performance, as fine-tuning the classifier for each embedding may introduce bias toward a specific embedding. In the output layer, the PyTorch CrossEntropyLoss function which combines SoftMax, and the cross-entropy loss was used to produce output probabilities for the multi-class classification. Subsequently, the accumulated positive and negative samples are employed as training data for each batch.

To compare cases where TO is used and not used, we implement two distinct data preparation strategies for training and testing. *When TO is used*, the dataset is randomly divided into 80% for training and 20% for testing. The triplet network model is trained using the training data with a triplet selection method, and the resulting model is used to optimize the embeddings of the test dataset. Optimized test data is used as input to the classifier network. Thus, a clear partitioning between the training and test data is maintained, preventing any potential bias resulting from prior learning on the test data. *When TO is not used*, TO operations are omitted, and the entire dataset is used as input for the classifier network.

The classifier network is trained using the k-fold cross-validation method, with a k value of 5, on the classifier input data. During each fold, 20% of the training data is selected as a validation set. The training process is conducted over 2000 epochs.

E. HYPERPARAMETER OPTIMIZATION

The model architecture, tokenization, and training procedure involve many hyperparameters that must be tuned to maximize predictive performance [56]. The optimization of transformer neural networks and the selection of effective hyperparameters are a computationally intensive problem requiring the exploration of a high-dimensional space. It involves conducting complete model training and inference [57]. Initially, we experimented with various configurations to determine the optimal DNN, including adjustments to the number of network layers, nodes, and dropout functions. The evaluation metrics are accuracy, precision, recall, F1-score, and loss.

We conducted a grid search using the following parameters and ranges to determine optimal hyperparameters:

- An optimizer is an algorithm that adjusts the neural network attributes, such as weights and learning rates update rules, and learning rates in the direction of the steepest descent of the loss function. The Stochastic Gradient Descent (SGD) and Adam optimizers were tested respectively for hyperparameter optimization.
- The learning rate (LR) determines the size of weight updates during gradient optimization. We varied the LR values as 10^{-4}, 10^{-5}, and 10^{-6}.
- Batch size (BS) refers to the number of training samples used to update the model's parameters in one iteration. It affects the speed and stability of training and impacts memory usage. Smaller batch sizes offer more frequent updates but can result in noisy gradients and slower convergence. Larger batch sizes provide smoother gradients but may require more memory and slower computation. Choosing an optimal batch size involves trade-offs between these factors and depends on

TABLE 1. Code smell definitions and statistic.

Smell Type	Description	Severity	Java LOC	PHP LOC	Python LOC	Total LOC
0 (Without smell)	Does not contain any smell	-	1143	1723	2086	4952
S107	Methods should not have too many parameters	Major	3013	6927	10450	20390
S112	General or reserved exceptions should never be thrown	Major	2573	5187	4779	12539
S1172	Unused method parameters should be removed to prevent confusion	Major	1483	1980	2445	5908
S3776	Code cognitive complexity of a function is above a certain threshold	Critical	6205	12471	9801	28477
		Total	14417	28288	29561	72266

the specific dataset and model architecture [58]. We varied the BS values as 16, 32, 64, 128, 256, and 512.

IV. RESULTS

This section presents the properties of the prepared dataset and provides a quantitative analysis of the results of the proposed approach applied to embeddings generated using BERT, CodeBERT, and GraphCodeBERT models. We analyzed the results of different embeddings for cases with and without triplet usage with accuracy, precision, recall, and F1-score metrics. We also investigate the class distributions of pre-triplet and post-triplet loss networks to explain the reason behind the performance increase. All code can be found at https://github.com/FSMVU-Tubitak-1001-Kod-Analiz/Triplet_Based_Embedding_Optimization link. We used NVIDIA GPU RTX 6000, equipped with 10,752 CUDA cores, 336 Tensor cores, and 48GB GDDR6 memory to evaluate the models and conduct all experiments.

We used pre-trained models provided by the *Hugging Face* service. We performed the necessary pre-processing, such as removing comments for each language model to work. The tokenizer significantly affects the performance of the PLM. We selected the recommended tokenizer for the best performance for each language model as *bert-base-nli-mean-tokens*for BERT, *codebert-base*for CodeBERT, and *graphcodebert-base*for GraphCodeBERT.

The dimensions of the code-embedding vectors produced by PLMs vary. BERT, CodeBERT, and GraphCodeBERT each produced 768-dimensional outputs. The size of the last hidden layer of GraphCodeBERT was 320 × 768. The DNN structure was adjusted to work for embeddings with different sizes by adding a flattened layer at the beginning of the model and then flexibly assigning the number of input nodes to the first layer.

A. DATASET

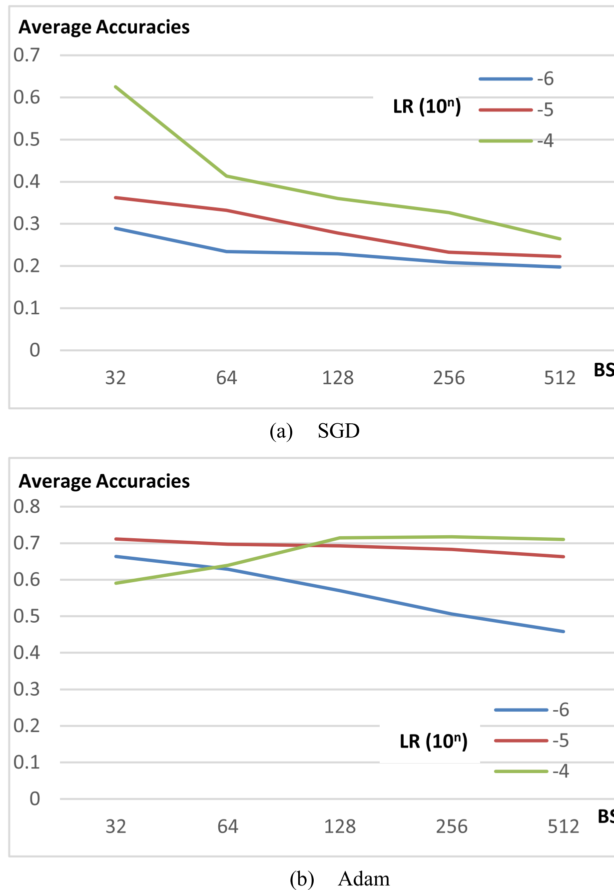
We examined 1,813 GitHub code repositories with open-source licenses such as Apache v2 and MIT during the data collection. We needed to find the same bad smells in different programming languages. For some languages, the default tokenizers of the selected PLMs were not implemented. Thus, we selected Java, PHP, and Python languages that have common 4 smells and tokenizers. Following these criteria, we collected data from Java, PHP, and Python repositories. Then, we combined the data sets from the different programming languages into a large training dataset.

Analysis and testing performed on the unbalanced data set gave low performance as the literature suggests similar results [16]. Therefore, we randomly selected 4 smell classes and 600 samples from each smell class for each language to ensure a balanced dataset distribution as summarized in Table 1. 600 code samples without code smell were also randomly collected to perform a baseline comparison. Thus, a total of 3,000 samples were collected from each language, resulting in a dataset containing 9,000 samples. We shared the dataset publicly at IEEE DataPort with doi:10.21227/j0rn-ht76 to allow further research.

Table 1 lists all the SonarQube code smell rules used in experiments. The selected code smells were examined in the relevant literature and classified as either critical or major. S3776 is a critical code smell that was studied in many studies with many neural network systems such as DNN, BiLSTM [22], and CNN and Recurrent Neural Networks [21]. The base metric of this smell (cyclomatic complexity) is also important for assessing other code smells [17]. S112 was observed in a high number of cases where catching the generic exception and empty catch block co-occurred with the maintainability smells forming the patterns [59]. This anti-pattern corresponds to more than 70% of all found smells [60]. In addition, S3776, S1172, and S112 are among the top fixed 20 code smell rules [61]. Another study identified S3776 and S112 rules as among the ten most frequently violated SonarQube rules [62]. Nearly 10% of rule violations were caused by 19 smells due to exceeding complexity measures including cognitive complexity and S107 smell in Java. [63]. Furthermore, S107 and S1172 are among the 40 most introduced *technical debt* items referred to as code smell in SonarCloud [64].

B. THE EVALUATION OF HYPERPARAMETERS EFFECT

To identify the optimal hyperparameters for various embedding models, we monitored the accuracy and loss of BERT, CodeBERT, and GraphCodeBERT embeddings across iterations for the code classification task until a common convergence point was reached with different hyperparameters. After determining the convergence point, we reexamined the accuracy of networks to select the best combination of hyperparameters before triplet tests, as illustrated in Fig. 4.



(a) SGD

(b) Adam

FIGURE 4. Accuracy values for different hyperparameters.

Thus, the Adam optimizer gave more consistent results than the SGD and achieved the highest accuracy. As a result, the parameter set Adam optimizer, 10^{-4} LR, and 256 BS were identified as the optimal choice.

Higher learning rates are required to effectively update the model for larger batch sizes, while small batch sizes may benefit from lower learning rates to prevent overfitting or instability [58]. Small batch sizes are computationally expensive and large-batch training is an efficient approach for DNN performance.

C. THE CLUSTERING PERFORMANCE OF TRIPLET LOSS EMBEDDINGS

After determining the optimal hyperparameter set, we conducted experiments with online and offline triplet generation models to identify the most suitable triplet selection method. We also compared the results with those obtained without using the TO, as shown in Table 2. Since the effect of BS on performance in the online triplet generation process is significant, the classification performance was tested across a range of BSs. The performance of the offline triplet selection method was higher than online methods. Therefore, detailed evaluations in the classification phase were conducted with the offline method.

TABLE 2. The accuracy values for triplet selection methods.

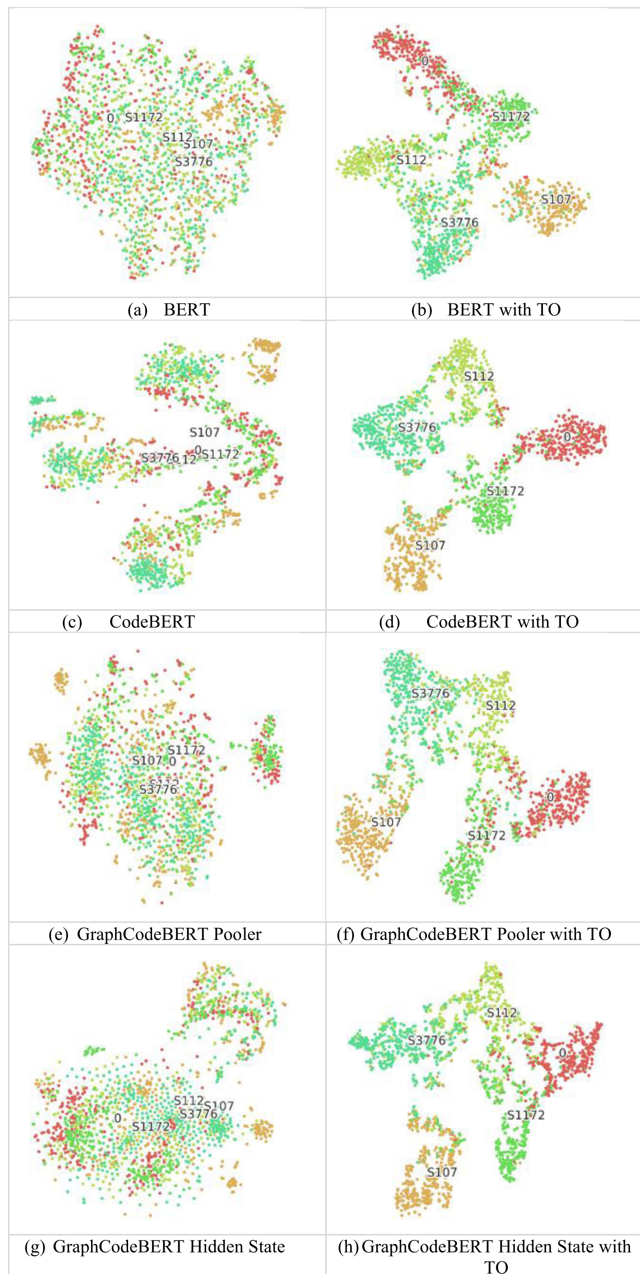
Triplet Selection Method	Pre-trained Model	BS	Accuracy
Without TO	BERT	*	0.6361
	CodeBERT	*	0.7561
	GraphCodeBERT Pooler	*	0.7072
Online Random Negative	BERT	50	0.6127
	CodeBERT	300	0.6828
	GraphCodeBERT Pooler	150	0.7472
Online Semi-hard Negative	BERT	450	0.4161
	CodeBERT	300	0.5256
	GraphCodeBERT Pooler	450	0.6317
Offline	BERT	*	**0.7211**
	CodeBERT	*	**0.8444**
	GraphCodeBERT Pooler	*	**0.7678**

We used the silhouette coefficient (SC) to evaluate quantitatively the correlation between the clustering outcomes of the training phase and the model's performance in the test phase when TO was employed. The formula for SC of each sample is $(b - a)/max(a, b)$ where a refers to the mean intra-cluster distance and b refers to the mean nearest-cluster distance. The mean SC across all samples ranges from -1 to 1, where 1 indicates the optimal and -1 indicates the poorest clustering performances. Values close to 0 indicate overlapping clusters and negative values indicate wrong cluster assignment.

As shown in Table 3, CodeBERT for Java achieves an SC of -0.013 and 0.7783 accuracy without the triplet and improves to 0.47 and 0.8850 with the TO, the highest SC observed. This highest SC also corresponds to the highest classification accuracy. Applying TO on the combined dataset, which includes all languages, exhibits similar SC results across the different models. CodeBERT achieves an SC of 0.37 and 0.8444 accuracy with TO, increased from -0.037 and 0.7561, corresponding to the highest accuracy, while BERT yields an SC of 0.21 and 0.7211 with TO, from -0.049 and 0.6361, representing the lowest accuracy. The results indicate that the increase in SC directly reflects the increase in classification performance.

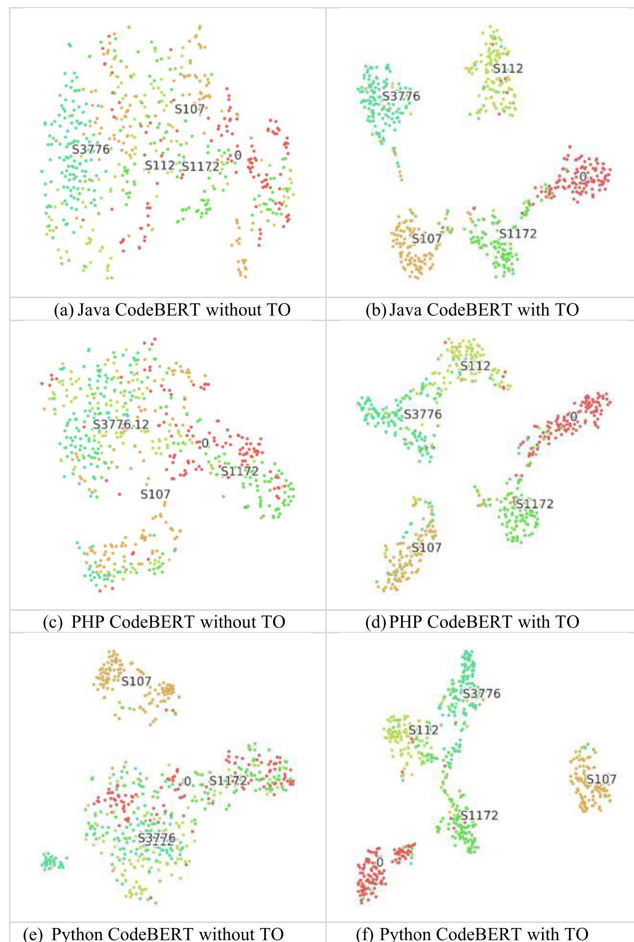
In addition to SC, we employed the t-SNE technique [65] to evaluate visually the effectiveness of triplet loss-based clustering on classification performance. t-SNE visualizes high-dimensional code vectors by mapping them onto a two-dimensional space, each class represented by a distinct color. Fig. 5a-h illustrates the methods' performance in separating class embeddings using triplet loss across four different embedding models for the combined dataset, which includes samples from all languages.

The results indicate that the data have a complex distribution in each direction, with lower inter-statement relations for direct use of language models. TO separates the different classes from each other, significantly improving classification performance. It also increases the quality of the clusters and provides high cohesion for each smell type.



(a) BERT (b) BERT with TO (c) CodeBERT (d) CodeBERT with TO (e) GraphCodeBERT Pooler (f) GraphCodeBERT Pooler with TO (g) GraphCodeBERT Hidden State (h) GraphCodeBERT Hidden State with TO

FIGURE 5. The impact of triplet loss on the distribution of embeddings for the combined dataset.



(a) Java CodeBERT without TO (b) Java CodeBERT with TO (c) PHP CodeBERT without TO (d) PHP CodeBERT with TO (e) Python CodeBERT without TO (f) Python CodeBERT with TO

FIGURE 6. The impact of triplet loss on the distribution of CodeBERT embeddings.

CodeBERT, with triplet loss on the Java language, has demonstrated the best performance among the other languages and PLMs. The t-SNE graphs of the CodeBERT in Figure 6 a-f reveal the rationale behind this enhancement and facilitate an examination of the impact of the data clustering quality of the triplet network on the classification performance. The clear separation between the classes allows for enhanced classification performance. The t-SNE graphs and confusion matrices of the models for other software languages are provided in Supplementary Material A, to avoid undue lengthening of the article.

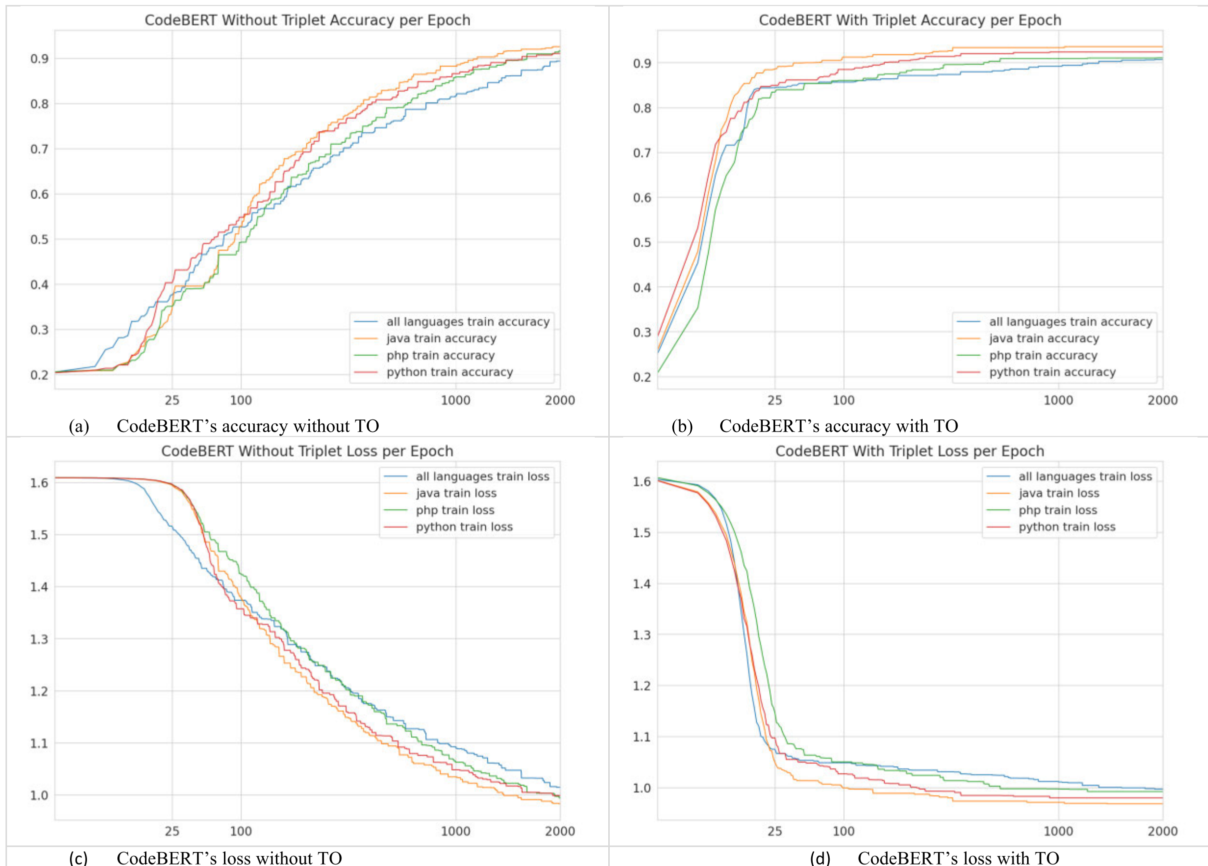
Fig. 7 a-d shows accuracy and loss with the number of iterations in the training phase for the CodeBERT model exhibiting optimal performance with and without triplet loss-based enhancements. The other models also show similar accuracy and loss characteristics.

The triplet loss-based model demonstrates a high convergence rate and low loss values, with similar improvements observed across different embedding models. Direct usage of embeddings achieved optimal results around the 2000th iteration during validation and continued improving over an extended training process.

Table 3 summarizes the performance metrics for all models. The proposed triplet loss approach significantly improves accuracy, precision, recall, and F1 score for nearly all models. CodeBERT gives the best performance when using or not using TO.

The GraphCodeBERT Hidden State has demonstrated the second-best performance in numerous instances. Nevertheless, the complex structure of this model, due to its huge embedding dimension, limits the ability to achieve a notable performance enhancement through TO. The BERT model, which has been designed for text analysis, also exhibits enhanced performance with TO.

In the offline triplet selection training phase, the loss function convergences in 16 iterations on average for (pooler)



(a) CodeBERT's accuracy without TO

(b) CodeBERT's accuracy with TO

(c) CodeBERT's loss without TO

(d) CodeBERT's loss with TO

FIGURE 7. Accuracy and loss change with original embeddings and embeddings obtained from Triplet loss.

embeddings, resulting in a relatively short duration of approximately 1 hour and 20 minutes. The hidden state of GraphCodeBERT required 6 hours to complete 4 iterations due to its substantial embedding size. Conversely, in online triplet selection, the convergence was reached over a longer duration depending on the batch size. A batch size of 75 required approximately 1 hour 20 minutes, while a batch size of 450 required approximately 10 hours. The time required for the TO system to create an embedding is less than one second. Thus, the classification performance remained consistent throughout the test process with or without using TO.

Table 4 shows the accuracies for different embeddings by class for the combined dataset. These results help to identify code smell types that are the most challenging to classify.

V. DISCUSSION

In this section, we evaluated the results and answered the research questions.

A. ANSWERS TO THE RESEARCH QUESTIONS

1) RQ1: How much triplet-based contrastive learning can improve the classification performance of code-embedding vectors compared to other optimization methods?

2) ANSWER 1: The comparative analysis results indicate that using triplet loss-based enhancement provides 1%-13% better performance for different embedding models in addition to hyperparameter optimization. When SC scores and t-SNE graphs are examined in Fig. 5, the main source of the performance increase is the smoother distribution between classes when TO is applied. We employed *t-test* to analyze the statistical differences introduced by TO. The results revealed a significant and substantial difference between the before ($M = 0.70$, $SD = 0.05$) and after TO ($M = 0.80$, $SD = 0.04$), $t(15) = 10$, $p < .001$.

Our study outperformed previous literature in several key aspects. A study employed a masked reserved words approach to the code smell detection task and achieved the detection performance of a score of F1 between 92.4-88.87 for single-type method level (feature envy and long method) smell detection on average [50]. Another study used the pretrained model's CodeBERT, CodeGPT, and CodeT5 to extract semantic relationships between code snippets to detect feature envy in Java language and achieved 64.94%, 65.91 and 81.89 F1 scores respectively [66].

The results of our study compared with those reported in the literature reveal that our approach can lead to a notable enhancement in performance even in multi-class and multi-language environments.

TABLE 3. Summary of the performance metrics for all models.

Language	TO	Embedding	Accuracy	Precision	Recall	F1-Score
All	Without	BERT Nli Mean	0.6361	0.6362	0.6361	0.6354
		CodeBERT	0.7561	0.7551	0.7561	0.7536
		GraphCodeBERT Hidden State	0.7706	0.7721	0.7706	0.7686
		GraphCodeBERT Pooler	0.7072	0.7056	0.7072	0.7057
	With	BERT Nli Mean	0.7211	0.7229	0.7211	0.7213
		CodeBERT	**0.8444**	**0.8451**	**0.8444**	**0.8442**
		GraphCodeBERT Hidden State	0.7811	0.7845	0.7811	0.7801
		GraphCodeBERT Pooler	0.7678	0.7661	0.7678	0.7667
Java	Without	BERT Nli Mean	0.6383	0.6409	0.6383	0.6376
		CodeBERT	0.7783	0.7749	0.7783	0.7758
		GraphCodeBERT Hidden State	0.7417	0.7483	0.7417	0.7385
		GraphCodeBERT Pooler	0.6583	0.6586	0.6583	0.6581
	With	BERT Nli Mean	0.7683	0.7687	0.7683	0.7672
		CodeBERT	**0.8850**	**0.8851**	**0.8850**	**0.8843**
		GraphCodeBERT Hidden State	0.8050	0.8105	0.8050	0.8040
		GraphCodeBERT Pooler	0.7283	0.7296	0.7283	0.7279
PHP	Without	BERT Nli Mean	0.6350	0.6344	0.6350	0.6325
		CodeBERT	0.7400	0.7497	0.7400	0.7407
		GraphCodeBERT Hidden State	0.7717	0.7858	0.7717	0.7712
		GraphCodeBERT Pooler	0.6800	0.6807	0.6800	0.6799
	With	BERT Nli Mean	0.7517	0.7552	0.7517	0.7513
		CodeBERT	0.8017	0.8035	0.8017	0.8010
		GraphCodeBERT Hidden State	**0.8050**	**0.8162**	**0.8050**	**0.8087**
		GraphCodeBERT Pooler	0.7750	0.7758	0.7750	0.7738
Python	Without	BERT Base Nli Mean	0.6483	0.6510	0.6483	0.6495
		CodeBERT	0.7067	0.7050	0.7067	0.7048
		GraphCodeBERT Hidden State	0.7483	0.7493	0.7483	0.7407
		GraphCodeBERT Pooler	0.7200	0.7189	0.7200	0.7179
	With	BERT Base Nli Mean	0.7467	0.7461	0.7467	0.7455
		CodeBERT	**0.8317**	**0.8361**	**0.8317**	**0.8316**
		GraphCodeBERT Hidden State	0.8117	0.8117	0.8117	0.8102
		GraphCodeBERT Pooler	0.8067	0.8099	0.8067	0.8067

TABLE 4. Class accuracies for the combined dataset.

TO	Embedding	0	S107	S112	S1172	S3776
Without	GraphCodeBERT Hidden State	0.8111	0.9028	0.6472	0.6500	0.8417
	CodeBERT	0.8056	0.8917	0.6361	0.6472	0.8000
	GraphCodeBERT Pooler	0.7306	0.8333	0.6250	0.6000	0.7472
	Bert Nli Mean	0.6333	0.6833	0.6972	0.5250	0.6417
	Average	0.7451	0.8278	0.6514	0.6056	0.7576
With	GraphCodeBERT Hidden State	0.6806	0.9500	0.6972	0.7389	0.8389
	CodeBERT	0.9083	0.9111	0.7833	0.7639	0.8556
	GraphCodeBERT Pooler	0.8194	0.8917	0.6528	0.7028	0.7722
	Bert Nli Mean	0.7472	0.8056	0.7111	0.6278	0.7139
	Average	0.7889	0.8896	0.7111	0.7083	0.7951

3) RQ2: Is it possible to generalize the triplet-based embedding enhancement method to improve the performance of different embeddings?

4) Answer 2: The enhanced metric scores of all models have proved that triplet loss-based contrastive learning can be generalized on the enhancement of pretrained code embedding in the code classification tasks. The results also demonstrate the adaptability of the proposed models in various pre-trained embedding techniques.

We also observed that as the embedding complexity increases, the effectiveness of triplet loss decreases. Utilizing triplet loss has enhanced even the performance of the BERT model to a level comparable to that of complex code analysis models without TO.

Our analysis of smell detection performance, categorized by class, indicates that S107 achieves the highest detection performance across all embedding models for the combined dataset as shown in Table 4. This performance may depend on the fact that all the problems causing the smell occur at the same point within the method. On the other hand, establishing a clear reason for the detection performance of S1172 and S112 can be challenging. They exhibit the weakest detection capabilities for the combined dataset while lacking any clear distinguishing features. The most likely explanation is that they impact fewer lines of code. In addition, S112 is defined by specific code terms. The detection performance for other smell classes remains relatively consistent. TO improved per-

formance across all classes, even though the improvement rates varied.

Our approach involves training the triplet network by generating numerous random matches between classes, for positive pairs consisting of samples from the same smell class and negative pairs from different classes. This process requires the utilization of significant system resources during the training phase. However, using an additional network in the test process did not significantly decrease performance. Thus, for code smell classes, an additional performance increase can be achieved with a small performance loss by re-using PLMs with the transfer learning method.

B. THREATS TO VALIDITY

To increase the generalizability of our models, we conducted a test using data collected from GitHub repositories written in multiple languages such as Java, PHP, and Python. The pre-trained models with TO operate effectively across various programming languages. However, the data set size remained relatively small. Thus, enlarging the dataset and experimenting with different software languages may help increase external validity and generalize the system.

We have only evaluated pre-trained embedding models for code classification or smell detection. To increase the generalizability of our models, contrastive learning-based embedding enhancement should be assessed on other code analysis tasks, such as clone detection, code generation, software quality evaluation, and code review. The methodology selected during the triplet-selection process significantly impacts performance. Therefore, the selection methods employed in different software engineering tasks should be chosen carefully.

In addition, hyperparameters significantly affect pre-trained models' performance and internal validity. The selection of appropriate hyperparameters is crucial for achieving optimal results. The necessity of fine-tuning by identifying the optimal combination of hyperparameters and triplet selection methods across various software engineering tasks may create limitations in real-world scenarios, hindering achieving optimal performance.

VI. CONCLUSION

In this research, we proposed a two-stage DL system leveraging contrastive learning to enhance embeddings generated using PLM for the code classification task, specifically targeting code smells. First, the embeddings were enhanced using a triplet-based network, and then the impact of this enhancement was evaluated by detecting code smells with a classifier DNN. Our experimental results indicate that this approach provides better results in performance metrics for various PLMs.

The recommended system can be applied as a preprocessing step for ML or DL-based code classification tasks by increasing inter-class distances while reducing intra-class distances. Therefore, it could improve the performance of other code-related classification tasks such as code language detection, bug detection, code comment classification, and technical debt classification. The high degree of accuracy demonstrated by the results, particularly in the context of the Java language, suggests the potential for the development of DL-based or integrated code analysis tools. In addition, the strong performance demonstrated on the integrated dataset containing various programming languages can help the development of unified code smell detection tools for programming languages. Code analysis with DL can be performed directly on code text, offering a significant advantage over some static code analysis tools that require compilation and can only function on fully compiled projects.

Although the results of the developed system are promising, there are potential areas for future improvement in applying contrastive learning systems to code analysis tasks. A potential research topic is to restructure and optimize contrastive learning by the specific requirements of the code analysis domain, such as developing specialized loss functions and enhanced triplet network models. An additional research area could be the incorporation of contrastive learning into the internal structure of the pre-trained model, thereby facilitating the development of novel pre-trained models.

ACKNOWLEDGMENT

The authors thank TUBITAK for their support.

REFERENCES

[1] Y. Sui, X. Cheng, G. Zhang, and H. Wang, "Flow2 Vec: Value-flow-based precise code embedding," *Proc. ACM Program. Lang.*, vol. 4, pp. 1–27, Nov. 2020, doi: 10.1145/3428301.

[2] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 8, pp. 1–79, Nov. 2024.

[3] O. O. Büyük and A. Nizam, "Deep learning with class-level abstract syntax tree and code histories for detecting code modification requirements," *J. Syst. Softw.*, vol. 206, Dec. 2023, Art. no. 111851, doi: 10.1016/j.jss.2023.111851.

[4] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, pp. 1–29, Jan. 2019, doi: 10.1145/3290353.

[5] E. Yahav and O. Levy, "Code2Seq: Generatıng sequences from structured representations of code," in *Proc. ICLR*, 2019, pp. 1–19.

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Human Lang. Technol.*, vol. 1, Jun. 2019, pp. 4171–4186.

[7] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Proc. Findings Assoc. Comput. Linguistics, EMNLP*, 2020, pp. 1536–1547, doi: 10.18653/v1/2020.findings-emnlp.139.

[8] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," in *Proc. ICLR*, 2021, pp. 1–18.

[9] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 3585–3597.

[10] E. Shi, Y. Wang, H. Zhang, L. Du, S. Han, D. Zhang, and H. Sun, "Towards efficient fine-tuning of pre-trained code models: An experimental study and beyond," 2023, *arXiv:2304.05216*.

[11] H. Li, C. Miao, C. Leung, Y. Huang, Y. Huang, H. Zhang, and Y. Wang, "Exploring representation-level augmentation for code search," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2022, pp. 4924–4936, doi: 10.18653/v1/2022.emnlp-main.327.

[12] T. Y. Zhuo, Z. Yang, Z. Sun, Y. Wang, L. Li, X. Du, Z. Xing, and L. David, "Data augmentation approaches for source code models: A survey," 2023, *arXiv:2305.19915*.

[13] Z. Dong, Q. Hu, Y. Guo, M. Cordy, M. Papadakis, Z. Zhang, Y. L. Traon, and J. Zhao, "MixCode: Enhancing code classification by mixup-based data augmentation," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Mar. 2023, pp. 379–390.

[14] H. Liu, Y. Zhang, V. Saikrishna, Q. Tian, and K. Zheng, "Prompt learning for multi-label code smell detection: A promising approach," 2024, *arXiv:2402.10398*.

[15] M. Škipina, J. Slivka, N. Luburić, and A. Kovačević, "Automatic detection of feature envy and data class code smells using machine learning," *Expert Syst. Appl.*, vol. 243, Jun. 2024, Art. no. 122855, doi: 10.1016/j.eswa.2023.122855.

[16] M. A. A. Hilmi, A. Puspaningrum, Darsih, D. O. Siahaan, H. S. Samosir, and A. S. Rahma, "Research trends, detection methods, practices, and challenges in code smell: SLR," *IEEE Access*, vol. 11, pp. 129536–129551, 2023, doi: 10.1109/ACCESS.2023.3334258.

[17] Y. Zhang, C. Ge, H. Liu, and K. Zheng, "Code smell detection based on supervised learning models: A survey," *Neurocomputing*, vol. 565, Jan. 2024, Art. no. 127014, doi: 10.1016/j.neucom.2023.127014.

[18] A. Alazba and H. Aljamaan, "Code smell detection using feature selection and stacking ensemble: An empirical investigation," *Inf. Softw. Technol.*, vol. 138, Aug. 2020, Art. no. 106648, doi: 10.1016/j.infsof.2021.106648.

[19] A. Alazba, H. Aljamaan, and M. Alshayeb, "Deep learning approaches for bad smell detection: A systematic literature review," *Empirical Softw. Eng.*, vol. 28, no. 3, p. 77, May 2023, doi: 10.1007/s10664-023-10312-z.

[20] H. Liu, J. Jin, Z. Xu, Y. Zou, Y. Bu, and L. Zhang, "Deep learning based code smell detection," *IEEE Trans. Softw. Eng.*, vol. 47, no. 9, pp. 1811–1837, Sep. 2021, doi: 10.1109/TSE.2019.2936376.

[21] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, "Code smell detection by deep direct-learning and transfer-learning," *J. Syst. Softw.*, vol. 176, Jun. 2021, Art. no. 110936, doi: 10.1016/j.jss.2021.110936.

[22] D. Yu, Q. Yang, X. Chen, J. Chen, S. Wang, and Y. Xu, "Actionable code smell identification with fusion learning of metrics and semantics," *Sci. Comput. Program.*, vol. 236, Sep. 2024, Art. no. 103110.

[23] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, May 2013, pp. 1–12.

[24] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv.*, vol. 51, no. 4, pp. 1–37, Jul. 2018.

[25] T. Sharma and D. Spinellis, "A survey on software smells," *J. Syst. Softw.*, vol. 138, pp. 158–173, Apr. 2018, doi: 10.1016/j.jss.2017.12.034.

[26] W. Wang, G. Li, S. Shen, X. Xia, and Z. Jin, "Modular tree network for source code representation learning," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 4, pp. 1–23, Oct. 2020, doi: 10.1145/3409331.

[27] K. Shi, Y. Lu, J. Chang, and Z. Wei, "PathPair2 Vec: An AST path pair-based code representation method for defect prediction," *J. Comput. Lang.*, vol. 59, Aug. 2020, Art. no. 100979, doi: 10.1016/j.cola.2020.100979.

[28] N. Tsantalis and A. Chatzigeorgiou, "Ranking refactoring suggestions based on historical volatility," in *Proc. 15th Eur. Conf. Softw. Maintenance Reeng.*, Mar. 2011, pp. 25–34, doi: 10.1109/CSMR.2011.7.

[29] A. Vaswani, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, Dec. 2017, pp. 5999–6009.

[30] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *Proc. 37th Int. Conf. Mach. Learn. (ICML)*, 2020, pp. 5066–5077.

[31] C. Niu, C. Li, V. Ng, D. Chen, J. Ge, and B. Luo, "An empirical comparison of pre-trained models of source code," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, May 2023, pp. 2136–2148, doi: 10.1109/ICSE48619.2023.00180.

[32] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[33] L. Liu, H. Nguyen, G. Karypis, and S. Sengamedu, "Universal representation for code," in *Advances in Knowledge Discovery and Data Mining* (Lecture Notes in Computer Science, Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 12714, 2021, pp. 16–28, doi: 10.1007/978-3-030-75768-7_2.

[34] S. Geisler, Y. Li, D. J. Mankowitz, A. T. Cemgil, S. Günnemann, and C. Paduraru, "Transformers meet directed graphs," in *Proc. Int. Conf. Mach. Learn.*, vol. 202, Jul. 2023, pp. 11144–11172.

[35] J. Huang, J. Zhao, Y. Rong, Y. Guo, Y. He, and H. Chen, "Code representation pre-training with complements from program executions," in *Proc. Conf. Empirical Methods Natural Lang. Process., Ind. Track*, 2023, pp. 267–278.

[36] N. D. Q. Bui, Y. Yu, and L. Jiang, "Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations," in *Proc. 44th Int. ACM SIGIR Conf. Res. Develop. Inf. Retr.*, Jul. 2021, pp. 511–521, doi: 10.1145/3404835.3462840.

[37] F. Zhang, M. Peng, Y. Shen, and Q. Wu, "Hierarchical features extraction and data reorganization for code search," *J. Syst. Softw.*, vol. 208, Feb. 2024, Art. no. 111896, doi: 10.1016/j.jss.2023.111896.

[38] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton. (2020). *A Simple Framework for Contrastive Learning of Visual Representations*. [Online]. Available: https://github.com/google-research/simclr

[39] T. Gao, X. Yao, and D. Chen, "SimCSE: Simple contrastive learning of sentence embeddings," 2021, *arXiv:2104.08821*.

[40] F. Schroff, D. Kalenichenko, and J. Philbin, "FaceNet: A unified embedding for face recognition and clustering," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 815–823, doi: 10.1109/CVPR.2015.7298682.

[41] G. Fan, S. Chen, C. Gao, J. Xiao, T. A. O. Zhang, and Z. Feng, "RAPID: Zero-shot domain adaptation for code search with pre-trained models," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 5, pp. 1–35, 2024, doi: 10.1145/3641542.

[42] J. Huang, D. Tang, L. Shou, M. Gong, K. Xu, D. Jiang, M. Zhou, and N. Duan, "CoSQA: 20,000+ Web queries for code search and question answering," 2021, *arXiv:2105.13239*.

[43] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. E. Gonzalez, and I. Stoica, "Contrastive code representation learning," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, Jul. 2020, pp. 5954–5971, doi: 10.18653/v1/2021.emnlp-main.482.

[44] S. Jeong, J. Baek, S. Cho, S. Ju Hwang, and J. C. Park, "Augmenting document representations for dense retrieval with interpolation and perturbation," 2022, *arXiv:2203.07735*.

[45] A. K. Das, S. Yadav, and S. Dhal, "Detecting code smells using deep learning," in *Proc. TENCON-IEEE Region 10 Conf. (TENCON)*, Oct. 2019, pp. 2081–2086, doi: 10.1109/TENCON.2019.8929628.

[46] A. Barbez, F. Khomh, and Y.-G. Guéhéneuc, "Deep learning anti-patterns from code metrics history," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2019, pp. 114–124, doi: 10.1109/ICSME.2019.00021.

[47] I. Latin and A. Transactions. (2024). *Code Smell Detection Research Based on Pre-Training and Stacking Models*. [Online]. Available: https://github.com/Lansforever/SCSmell.git/

[48] M. Hadj-Kacem and N. Bouassida, "Application of deep learning for code smell detection: Challenges and opportunities," *SN Comput. Sci.*, vol. 5, no. 5, p. 614, 2024.

[49] A. Bhave and R. Sinha, "Deep multimodal architecture for detection of long parameter list and switch statements using DistilBERT," in *Proc. IEEE 22nd Int. Work. Conf. Source Code Anal. Manipulation (SCAM)*, Oct. 2022, pp. 116–120, doi: 10.1109/SCAM55253.2022.00018.

[50] A. Alazba, H. Aljamaan, and M. Alshayeb, "CoRT: Transformer-based code representations with self-supervision by predicting reserved words for code smell detection," *Empirical Softw. Eng.*, vol. 29, no. 3, p. 59, May 2024, doi: 10.1007/s10664-024-10445-9.

[51] Y. Ma, S. Luo, Y.-M. Shang, Y. Zhang, and Z. Li, "Enhancing source code classification effectiveness via prompt learning incorporating knowledge features," *Sci. Rep.*, vol. 14, no. 1, p. 20220, Dec. 2024, doi: 10.1038/s41598-024-69402-7.

[52] G. Yang, "DeepSCC: Source code classification based on fine-tuned RoBERTa," in *Proc. Int. Conf. Softw. Eng. Knowl. Eng.*, Jul. 2021, pp. 499–502, doi: 10.18293/SEKE2021-005.

[53] J. Lin, R. Nogueira, and A. Yates, "Pretrained transformers for text ranking: BERT and beyond," 2020, *arXiv:2010.06467*.

[54] J. Zhan, J. Mao, Y. Liu, J. Guo, M. Zhang, and S. Ma, "Optimizing dense retrieval model training with hard negatives," 2021, *arXiv:2104.08051*.

[55] G. Sumbul, M. Ravanbakhsh, and B. Demir, "Informative and representative triplet selection for multilabel remote sensing image retrieval," *IEEE Trans. Geosci. Remote Sens.*, vol. 60, 2022, Art. no. 5405811, doi: 10.1109/TGRS.2021.3124326.

[56] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "IntelliCode compose: Code generation using transformer," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Nov. 2020, pp. 1433–1443, doi: 10.1145/3368089.3417058.

[57] K. C. Swarna, N. S. Mathews, D. Vagavolu, and S. Chimalakonda, "On the impact of multiple source code representations on software engineering tasks—An empirical study," *J. Syst. Softw.*, vol. 210, Apr. 2024, Art. no. 111941, doi: 10.1016/j.jss.2023.111941.

[58] L. Balles, J. Romero, and P. Hennig, "Coupling adaptive batch sizes with learning rates," 2016, *arXiv:1612.05086*.

[59] A. Oliveira, J. Correia, L. Sousa, W. K. G. Assunção, D. Coutinho, A. Garcia, W. Oizumi, C. Barbosa, A. Uchôa, and J. A. Pereira, "Don't forget the exception!: Considering robustness changes to identify design problems," in *Proc. IEEE/ACM 20th Int. Conf. Mining Softw. Repositories (MSR)*, May 2023, pp. 417–429, doi: 10.1109/MSR59073.2023.00064.

[60] D. B. C. de Sousa, P. H. M. Maia, L. S. Rocha, and W. Viana, "Studying the evolution of exception handling anti-patterns in a long-lived large-scale project," *J. Brazilian Comput. Soc.*, vol. 26, no. 1, pp. 1–24, Dec. 2020, doi: 10.1186/s13173-019-0095-5.

[61] M. T. Baldassarre, V. Lenarduzzi, S. Romano, and N. Saarimäki, "On the diffuseness of technical debt items and accuracy of remediation time when using SonarQube," *Inf. Softw. Technol.*, vol. 128, Dec. 2020, Art. no. 106377, doi: 10.1016/j.infsof.2020.106377.

[62] M. Odermatt, D. Marcilio, and C. A. Furia, "Static analysis warnings and automatic fixing: A replication for C# projects," in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, Mar. 2022, pp. 805–816, doi: 10.1109/SANER53432.2022.00098.

[63] M. Schütz and R. Plösch, "A practical failure prediction model based on code smells and software development metrics," in *Proc. 4th Eur. Symp. Softw. Eng.*, Dec. 2023, pp. 14–22, doi: 10.1145/3651640.3651644.

[64] V. Lenarduzzi, N. Saarimaki, and D. Taibi, "On the diffuseness of code technical debt in Java projects of the apache ecosystem," in *Proc. IEEE/ACM Int. Conf. Tech. Debt (TechDebt)*, May 2019, pp. 98–107, doi: 10.1109/TECHDEBT.2019.00028.

[65] L. Van der Maaten and G. Hinton, "Visualizing data using t-SNE," *J. Mach. Learn. Res.*, vol. 9, no. 11, pp. 2579–2605, 2008.

[66] W. Ma, Y. Yu, X. Ruan, and B. Cai, "Pre-trained model based feature envy detection," in *Proc. IEEE/ACM 20th Int. Conf. Mining Softw. Repositories (MSR)*, May 2023, pp. 430–440, doi: 10.1109/MSR59073.2023.00065.

ALİ NİZAM was born in Fatih, İstanbul, Türkiye, in 1976. He received the B.S. degree in electronic engineering from Yıldız Technical University, İstanbul, in 1997, and the M.S. and Ph.D. degrees in electronic-biomedical engineering from İstanbul Technical University, İstanbul, in 2000 and 2009, respectively.

From 1997 to 2011, he was with ISKI, as a Software Engineer, the Project Manager, and the Management Information Systems Manager. Since 2011, he has been an Assistant Professor with the Computer/Software Engineering Department, Fatih Sultan Mehmet Vakıf University, İstanbul. He is the author of four books, one chapter, and seven articles. His research interests include software engineering, relational database concepts, and data science.

Dr. Nizam's awards and honors include Türkiye Academy of Science (TÜBA) and the University Textbooks Award Program Best Original Book Award with Software Project Management Book.

ERTUĞRUL İSLAMOĞLU was born in Ankara, Türkiye, in 1997. He received the B.S. degree in computer engineering from Istanbul University, in 2021, and the M.S. degree in computer engineering from Fatih Sultan Mehmet Vakıf University, in 2024, where he is currently pursuing the Ph.D. degree in computer engineering..

He was a Research Assistant with the Software Engineering Department, Fatih Sultan Mehmet Vakıf University, from 2022 to 2024. Since then, he has been a Research Assistant with the Artificial Intelligence and Data Engineering Department, Fatih Sultan Mehmet Vakıf University. He is the co-author of two proceedings. His research interests include deep learning, large language models, and transformer neural networks.

ÖMER KEREM ADALI was born in Kadıköy, İstanbul Türkiye, in 2000. He received the B.S. degree in computer engineering from Fatih Sultan Mehmet Vakıf University, İstanbul, in 2024.

From 2023 to 2024, he was involved on a TUBITAK 1001 Project that aimed to detect code smells using deep learning. His research interests include software development and deep learning.

MUSA AYDIN was born in Giresun, Türkiye, in 1987. He received the B.S. degree from the Computer Control Department, Marmara University, in 2010, the M.S. degree in computer control technology from the Institute of Science, Marmara University, in 2012, and the Ph.D. degree from the Department of Computer Engineering, Institute of Science, Marmara University, in 2020.

From 2022 to 2023, he was a Postdoctoral Researcher with the University of California at Los Angeles (UCLA), for one year on deep learning microscopy and computational imaging. Since 2020, he has been an Assistant Professor with the Computer Engineering Department, Fatih Sultan Mehmet Vakıf University.