applied sciences

MDPI

Article

TCCCD: Triplet-Based Cross-Language Code Clone Detection

Yong Fang, Fangzheng Zhou, Yijia Xu * and Zhonglin Liu

College of Cybersecurity, Sichuan University, Chengdu 610065, China; fangyongscu@gmail.com (Y.F.); 2021226240007@stu.scu.edu.cn (F.Z.); jungleforsa@gmail.com (Z.L.)
* Correspondence: xuyijia@stu.scu.edu.cn

Abstract: Code cloning is a common practice in software development, where developers reuse existing code to accelerate programming speed and enhance work efficiency. Existing clone-detection methods mainly focus on code clones within a single programming language. To address the challenge of code clone instances in cross-platform development, we propose a novel method called TCCCD, which stands for Triplet-Based Cross-Language Code Clone Detection. Our approach is based on machine learning and can accurately detect code clone instances between different programming languages. We used the pre-trained model UniXcoder to map programs written in different languages into the same vector space and learn their code representations. Then, we fine-tuned TCCCD using triplet learning to improve its effectiveness in cross-language clone detection. To assess the effectiveness of our proposed approach, we conducted thorough comparative experiments using the dataset provided by the paper titled CLCDSA (Cross Language Code Clone Detection using Syntactical Features and API Documentation). The experimental results demonstrated a significant improvement of our approach over the state-of-the-art baselines, with precision, recall, and F1-measure scores of 0.96, 0.91, and 0.93, respectively. In summary, we propose a novel cross-language code-clone-detection method called TCCCD. TCCCD leverages the pre-trained model UniXcode for source code representation and fine-tunes the model using triplet learning. In the experimental results, TCCCD outperformed the state-of-the-art baselines in terms of the precision, recall, and F1-measure.

Keywords: code clone detection; cross-language; pre-trained model; triplet learning

Citation: Fang, Y.; Zhou, F.; Xu, Y.; Liu, Z. TCCCD: Triplet-Based Cross-Language Code Clone Detection. *Appl. Sci.* **2023**, *13*, 12084. https://doi.org/10.3390/app132112084

Academic Editors: Antonio Fernández-Caballero and João M. F. Rodrigues

Received: 18 September 2023
Revised: 16 October 2023
Accepted: 2 November 2023
Published: 6 November 2023

1. Introduction

Code cloning refers to code fragments that implement the same functionality and have identical or similar syntax and semantics. Code clone instances are common in software development and maintenance. In general software projects, there are 5–20% of duplicated code [1], while high-quality software systems like Linux core code have even higher rates of code reuse, up to 15–25% [2].

With the development of software engineering and open-source communities, the proportion of code clones in software systems will continue to increase [3]. While benefits such as reduced development time can be provided by code clones, they can also have significant negative impacts on software quality and maintenance [4]. The primary harm of code clone instances is that the complexity in software systems is increased, which, in turn, leads to an increase in maintenance costs. Additionally, the number of potential defects and vulnerabilities in the software system can be increased by code clone instances, thus decreasing software quality and reliability [5].

According to the research [1], code clones are typically classified into four types: Type-I, Type-II, Type-III, and Type-IV. Type-I, Type-II, and Type-III belong to the syntactic clone class, meaning that they have similarity in code structure and syntax. Type-IV belongs to the semantic clone class, meaning that they have the same functionality, but may differ greatly in implementation, involving different programming languages, libraries, or algorithms.

Code-clone-detection technology has made significant progress in the detection of Type-I, Type-II, and Type-III clones, with methods such as CCFinder [6], Deckard [7], NiCad [8], and SourcererCC [9] being proposed. However, for Type-IV clone detection, which is semantic clone detection, there are still challenges to be faced. These challenges include the difficulty of fully capturing and comprehending the intricate semantics of code, as well as the insufficient performance and efficiency in handling large-scale code repositories. To address these challenges, many researchers have combined static detection methods with deep learning and applied them to Type-IV clone detection, achieving promising results, such as CCLearner [10], CCGraph [11], ASTNN [12], and DeepSim [13].

Prior research in the field of code clone detection primarily concentrated on identifying clones within the single programming language, with limited attention given to cross-language clone detection. Recent studies on cross-language code clone detection encompass LICCA [14], CLCDSA [15], and C4 [16]. LICCA is a tool based on the SSQSA [17] architecture for cross-language code clone detection. However, it exhibits certain limitations, such as requiring equal source code lengths and identical code steps and functional flow for code blocks, restricting its applicability in real-world scenarios. CLCDSA utilizes Abstract Syntax Trees (ASTs) to extract features for detecting cross-language code clones. Nonetheless, its precision and recall rates are relatively low, impeding its effectiveness in real-world scenarios. C4 leverages the pre-trained model CodeBERT [18] to convert code from diverse languages into high-dimensional vector representations, facilitating the detection of cross-language code clones. However, C4's use of CodeBERT's pre-trained models for languages beyond the six it was trained on may lead to less-accurate code representation. Additionally, C4's pre-trained approach solely focuses on modeling natural language, neglecting the valuable structural information present in the code. This limitation restricts C4's ability to fully exploit the code's structural and data flow features.

In the research on cross-language code clone detection, several challenges are encountered:

- **Source code representation:** Cross-language source code representation faces several challenges. Firstly, there are syntax and semantic differences between different programming languages, resulting in potentially different representations in the vector space for code with similar functionality. Secondly, different programming languages have distinct code structures and conventions, such as variable naming rules and function call styles, which also impact the vector representation of code [19,20].
- **Identification of clone/non-clone code pairs**: The difficulty in determining whether code pairs are clones or non-clones in vector space mainly arises from several factors. First, the vector representation of code may have a high dimensionality, leading to a significant amount of redundant information, which can impact the effectiveness and efficiency of similarity calculations. Additionally, due to the diversity and complexity of code, even if the code functionality is similar, their representations in the vector space may differ substantially, making the similarity calculation of clone code pairs complex [12,19].

In response to the aforementioned challenges, we propose a novel approach named Triplet-based Cross-language Code Clone Detection (TCCCD). Our approach leverages the pre-trained model UniXcoder [21] and adopts triplet learning. Firstly, we employed the UniXcoder pre-trained model to map clone code from different languages into a shared high-dimensional vector space, addressing the challenge of source code representation. Secondly, we used triplet learning [22,23] to tackle the challenge of clone/non-clone code pair identification and to learn effective feature representations. Triplet learning demonstrates flexibility in selecting positive, anchor, and negative samples, adapting to intricate cross-language code structures. Moreover, triplet learning enhances the discriminative capacity of vector representations, thereby improving detection effectiveness.

To validate the effectiveness of our experiments, we conducted comparative experiments on the same dataset, comparing our approach with CLCMiner, CLCDSA, and C4. The experimental results demonstrated that our approach outperformed state-of-the-art ap-

proaches significantly. Compared to the current state-of-the-art models for cross-language clone detection, our approach achieved an improvement from 0.92 to 0.93 in the F1-score, an increase from 0.94 to 0.96 in precision, and a rise from 0.90 to 0.91 in recall. Furthermore, compared to UniXcoder without triplet learning, our approach achieved a 15.3% increase in precision and a 3.4% improvement in the F1-score.

In conclusion, our main contributions can be summarized as follows:

1. A novel and effective method, TCCCD, for cross-language code clone detection, which leverages UniXcoder to efficiently represent code.
2. The triple-based learning method in the context of cross-language code clone detection and validating its effectiveness and applicability.
3. The evaluation of the effectiveness of TCCCD on the CLCDSA dataset. The experimental results demonstrated that TCCCD outperformed the state-of-the-art clone-detection methods.

2. Related Work

In this section, we explore existing work and research advancements in the field of code clone detection. Existing studies are analyzed and compared to to gain a better understanding of our contributions and the innovation behind our approach.

2.1. Code Clone Detection

We categorize the existing research in code clone detection into three parts based on the techniques employed and specific scenarios. These parts include traditional approaches, deep learning approaches, and cross-language code clone detection. Subsequent content will provide detailed descriptions.

2.1.1. Traditional Approaches

In traditional approaches, researchers typically rely on static features of the code, such as code tokens, abstract syntax trees, textual analysis, and metrics, to capture the semantic similarity between code fragments, primarily focusing on Type-I, Type-II, and weak Type-III clone detection.

Deckard [7] employs a tree-based representation to encode code fragments and utilizes both the structural and content information of the tree to identify cross-language code clones. Deckard's study relies on specific programming language parsers and code representations, limiting its support to only a few programming languages for clone detection.

CCFinder [6] proposes a clone-detection method based on token sequence comparison. The method converts code fragments into token sequences and utilizes the similarity of token sequences to identify clone code. However, CCFinder relies solely on the similarity of token sequences to detect code clones, which may not capture more-fine-grained code similarities.

Boreas [24] proposes a token-level code representation, constructs a structured representation of the code, and introduces a matching algorithm based on similarity measurement to identify code clones by comparing the similarity of token sequences. Boreas shares similar limitations with CCFinder.

In addition, NiCad [8] treats source code as character sequences and performs clone detection based on textual analysis. This kind of method is easy to implement and is language-independent. Shawky et al. [25] and CMCD [26] perform fixed-granularity clone detection, utilizing metric-based approaches. These methods offer a fast detection speed. The effectiveness of traditional methods in code clone detection is not particularly prominent, as they exhibit limitations in terms of both effectiveness and speed.

2.1.2. Deep Learning Approaches

In recent years, there have been emerging approaches in clone detection that leverage deep learning techniques. These methods harness the power of neural networks to auto-

matically learn meaningful representations of code, leading to more-accurate and -effective clone detection.

CCLearner [10] employs a Convolutional Neural Network (CNN) and a Recurrent Neural Network (RNN) as its primary techniques to handle the structural and sequential information of the code, thereby capturing the semantics and contextual relationships of the code.

TBCNN [27] utilizes an AST to transform code into a tree-like structure and performs clone detection based on the shape and content information of the tree. This method defines a set of convolutional kernel functions to perform convolutions on the nodes of the tree, capturing the similarity between code fragments.

DeepSim [13] utilizes a CNN and an RNN as the main techniques to process the structural and sequential information of the code. By converting code fragments into vector representations and encoding them through neural networks, DeepSim effectively captures the semantic relationships between code segments.

With the advancement of deep learning techniques, the Graph Neural Network (GNN) has been widely applied to code clone detection with significant results. FCCA [28] transforms code snippets into an AST and Program Dependency Graph (PDG). It employs a hybrid code representation approach based on a Graph Convolutional Network (GCN) and attention mechanisms to model and detect code clones using these representations. Wang [20] et al. proposed a method that combines a GNN and a Flow-Augmented Abstract Syntax Tree (FA-AST) to detect code clones by integrating syntax trees with data flow information.

2.1.3. Cross-Language Code Clone Detection

In recent years, there has been an increasing amount of research on cross-language code clone detection due to the wide variety of programming languages used in software development.

LICCA [14] incorporates various techniques and methods such as code parsing, AST construction, code feature extraction, and similarity comparison for cross-language code clone detection. CLCMiner [29] introduces a novel technique for cross-language clone detection based on revision histories, eliminating the need for an intermediate language. CLCDSA [15] proposes a method that utilizes cross-language API call similarity to analyze the syntactic features of the source code in different programming languages for cross-language code clone detection. C4 [16] utilizes the pre-trained model CodeBERT [18] to convert programs in different languages into vector representations in the same space. Through contrastive learning, it effectively identifies clone pairs and non-clone pairs.

2.2. Source Code Representation Learning

Source code representation learning is a significant research direction in the field of software engineering, aiming to transform source code into machine-interpretable vector representations. With the advancement of deep learning techniques, an increasing number of researchers have embarked on exploring the use of deep learning methods to learn meaningful representations of source code. Source-code-representation-learning models can be mainly categorized into token-based and graph-based approaches.

2.2.1. Token-Based Approaches

The token-based source-code-representation-learning method is a significant research direction in the field. White et al. [30] utilize an RNN, Long Short-Term Memory (LSTM), a Gated Recurrent Unit (GRU), and other models to extract vector representations of tokens. Bhoopchand et al. [31] utilize an RNN to extract token context and employed a sparse pointer network to capture long-range dependencies. On the other hand, Dam et al. [32] use the LSTM model to extract token embeddings, introducing gate mechanisms to control information flow and, thus, exhibiting a stronger ability than an RNN in capturing long-range dependencies among tokens.

Utilizing pre-training techniques significantly improves the predictive effectiveness of NLP models. Mikolov et al. [33,34] employ word2vec [33] and BERT [35] for token pre-training, leading to enhanced model effectiveness. BERT, a bidirectional Transformer encoder, adjusts model parameters through two pre-training objectives: masked language model and next sentence prediction [35]. Feng et al. [18] propose a multimodal pre-trained model that leverages the complementary interactions among different modalities to enhance the overall representation capability of the model. CodeBERT [18], based on documents and source code, conducts pre-training using BERT in the bimodal context of natural language and programming language. It captures semantic connections between the two different types of languages to provide universal representation vectors for downstream tasks [16,36,37].

2.2.2. Graph-Based Approaches

In graph-based source code representation, the source code is represented as a graph structure, where various elements of the code, such as functions, variables, and statements, are treated as nodes in the graph. The relationships between these elements, such as function calls and variable dependencies, are represented as edges in the graph. Ref. [38] converts the AST of the source code into a graph structure, where each node and edge in the AST is mapped to nodes and edges in the graph. The Inst2vec [39] model is based on the Low-Level Virtual Machine (LLVM) framework [40] to construct semantic flow graphs and uses the skip-gram algorithm for training to obtain vector representations of the graphs. The BRGCN [41] model extracts instruction-level heterogeneous data flow graphs, where edges include various relationships such as data flow, variable adjacency, and read–write relationships. It utilizes an R-GCN [42] for heterogeneous representation learning. Wang et al. [43] adopt the Transformer framework for heterogeneous network representation learning. The attention mechanism of the Transformer is related to the types of source and target nodes, with key values and query values associated with the nodes' types. This enables the model to learn the interactive features of heterogeneous nodes.

3. Background

We delve into three essential parts of our research background: code clone, pre-trained models, and triplet learning. A more-detailed exploration of these parts will be presented in the following content.

3.1. Code Clone

Code cloning refers to the existence of similar or identical code fragments in a software system, which achieve the same function [1]. According to the research [1], code clones can be classified into the following categories:

Type-I: code fragments that have the same syntax except for whitespace and comments.

Type-II: code fragments that have the same syntax as Type-I clones, but also include variations in identifier names and literal values.

Type-III: code fragments that exhibit more syntax changes compared to Type-I and Type-II clones, such as added, modified, or deleted statements.

Type-IV: code fragments that have different syntax, but still perform the same functionality.

The focus of our research on cross-language code clones primarily revolves around Type-IV clones, which are characterized by code fragments that exhibit different syntax, but possess similar functionality [44]. The provided Figure 1 showcases a cross-language code clone example depicting the implementation of the linked list reversal functionality using Java and Python. It is evident from Figure 1 that distinct programming languages exhibit notable differences in syntax and structure. In particular, significant disparities are observed in syntax rules, keyword usage, class and method definitions, variable declarations, and overall statement composition between Java and Python. These variations

highlight the inherent distinctions encountered in syntax and structure when comparing code clones across different programming languages.

Solution for reversing a link list

Python

```python
def reverseList(self, head:ListNode) -> ListNode:
    def recur(cur, pre):
        if not cur: return pre;
        res = recur(cur.next, cur)
        cur.next = pre
        return res
    return recur(head, None)
```

Java

```java
public ListNode reverseList(ListNode head){
    if (head == null || head.next == null){
        return head;
    }
    ListNode newHead = reverseList(head.next);
    head.next.next = head;
    head.next = null;
    return newHead;
}
```

Figure 1. Cross language clone example.

When performing clone detection across different programming languages, a major challenge lies in establishing a unified intermediate representation. This unified intermediate representation should possess the following characteristics:

1. **Cross-language compatibility:** Due to the differences in syntax, rules, and control symbols between different programming languages, the intermediate representation must overcome these diversities and be compatible with various programming languages.
2. **Information preservation:** The intermediate representation needs to accurately preserve important information from the source code, including variables, functions, classes, and control flow structures.
3. **Comparability:** The intermediate representation should be comparable, allowing code written in different programming languages to be compared at the level of the intermediate representation. This requires the intermediate representation to capture both the syntax and semantic features of the code, enabling effective matching and comparison operations.

To address these challenges, we chose the pre-trained model UniXcoder for code representation, transforming code from different programming languages into a shared vector space for effective representation. This approach enables cross-language code comparison and analysis as the foundation.

3.2. Pre-Trained Models

Pre-trained models have emerged as a powerful approach in Natural Language Processing (NLP) by utilizing large-scale unlabeled textual data to learn rich language representations and patterns. By leveraging techniques such as self-supervised learning and Transformer architectures, state-of-the-art pre-trained models have achieved remarkable success in the NLP field [18,21,35].

3.2.1. BERT

BERT [35] is based on the Transformer architecture and adopts a bidirectional approach to better capture the context of words in a sentence. Unlike traditional language models, which process words sequentially, BERT leverages the power of self-attention mechanisms to simultaneously model the relationships between all words in the input sentence.

BERT comprises two main training objectives: Masked Language Modeling (MLM) and Next Sentence Prediction (NSP). In MLM, a certain percentage of words in each input sentence is randomly masked, and the model is trained to predict the masked words based on the surrounding context. This enables BERT to develop a deep understanding of the relationships between words and their context. NSP, on the other hand, involves training

BERT to predict whether two sentences are consecutive or not. This helps BERT capture the semantic relationships between sentences and improve its ability to handle tasks that involve sentence-level understanding.

To train BERT, a massive amount of unlabeled text data are utilized, enabling the model to learn contextualized representations of words. This unsupervised training process helps BERT capture various levels of language information, including syntax and semantics.

3.2.2. CodeBERT

CodeBERT [18] is an advanced code-representation-learning method proposed by Microsoft Research Asia, which draws inspiration from the successful NLP model BERT. Similar to BERT, CodeBERT adopts the Transformer [45] architecture, a deep neural network structure based on attention mechanisms, capable of capturing long-range dependencies and contextual information. Through pre-training on large-scale code data, CodeBERT aims to learn universal code representations, providing powerful feature representations for various code-related tasks [16,36,37].

During the pre-training phase, CodeBERT performs the masked language modeling [35] on abundant code data by masking parts of code snippets and predicting their original content. This process enables the model to understand the structure and semantics of code effectively. The pre-training procedure captures abstract semantics and contextual information in the code, endowing CodeBERT with the ability to deeply comprehend and represent code.

3.2.3. UniXcoder

UniXcoder [21] is a novel unified cross-modal pre-trained model for programming languages. It addresses the limitations of existing encoder–decoder frameworks by introducing mask attention matrices with prefix adapters, which allow fine-grained control over the model's behavior. The model leverages cross-modal contents, such as ASTs and code comments, to enhance the representation of the source code. Compared to CodeBERT, UniXcoder expands the amount of programming language data it learns from, pre-training on nine programming languages, Go, Python, Java, JavaScript, Php, Ruby, C, C++, and C#. As a result, it achieves better performance in representing cloned code in C++ and C#.

When using the encoder-only mode, UniXcoder processes the input sequence by adding a special token (such as [Enc]) as a prefix and setting all elements of the attention mask matrix to 0, allowing all tokens to attend to each other. This processing method enables UniXcoder to consider all tokens in the input sequence simultaneously, capturing their associations and semantic information. By utilizing the self-attention mechanism, UniXcoder models the global dependencies of the input sequence and generates corresponding encoding representations.

UniXcoder stands as the most-state-of-the-art pre-trained model, exhibiting remarkable effectiveness in various code-related tasks. We employed UniXcoder to transform code fragments from various programming languages into a unified high-dimensional vector space. Subsequently, we utilized the triplet learning method to further explore the learning features within these transformed vectors, effectively capturing the semantic relationships and similarities between code fragments.

3.3. Triplet Learning

Triplet learning is a fundamental concept in metric learning, first introduced and applied in the field of image recognition by researchers [46] at Google in 2015. Triplet learning utilizes triplets composed of an anchor sample X_i^a, a positive sample X_i^p (similar to the anchor), and a negative sample X_i^n (dissimilar to the anchor) to train the model.

The distance function between two vector, X_1, X_2, is defined as $D(X_1, X_2)$. The triplet loss function is defined as Equation (1):

$$L = \sum_{i=1}^{N} [D(X_i^a, X_i^p) - D(X_i^a, X_i^n) + \alpha]_+. \quad (1)$$

where N is cardinality of train set. α is a margin that is enforced between positive and negative pairs.

By optimizing this loss function, a discriminative embedding representation is learned by the model, where similar samples are brought closer together and dissimilar samples are pushed farther apart in the embedding space. This leads to improved similarity measurement and sample classification performance.

Researchers have proposed various improved triplet loss functions to better capture the relative relationships between features. These improvements include cosine–margin–triplet loss [23], weighted triplet loss [47,48], adaptive triplet loss [49], and others [49–51]. Among them, the cosine–margin–triplet loss, based on cosine similarity, restricts the embedding vectors of samples within a specific cosine boundary. This helps enhance the similarity between positive samples while ensuring that the embedding vectors of negative samples are located outside the cosine boundary. This approach can strengthen the discriminative power of embedding vectors. The goal of these methods is to increase the separation of samples in the embedding space, thereby enhancing the model's effectiveness. Weighted triplet loss takes into account the importance of different samples for model training by introducing sample weights for optimization. Adaptive triplet loss allows the model to adaptively adjust the loss function based on the current training state and sample distribution.

In triplet learning, the construction of appropriate negative samples plays a crucial role in effectively training the model. Researchers have proposed various approaches and strategies to address this key issue, such as soft triple loss [52] and distance-weighted sampling loss [22].

Due to the effectiveness of triplet learning in capturing the relative relationships between features and learning discriminative embedding representations, it is combined with other loss functions to enhance learning effectiveness. Specifically, the distance-weighted sampling loss and triplet loss are integrated, and the cosine–margin–triplet loss [23] is employed as an improved approach to enhance triplet learning.

The triplet learning approach we employed can be summarized in the following steps:

1. Using an encoder $f(.)$ to transform code fragments into high-dimensional vectors.
2. Selecting an appropriate sampling method called $S(.)$ for triplet learning; some research has been effective, such as [22]. Treating each vector in the batch as an anchor X, according to sampling method $S(.)$ to chose positive samples and negative samples.
3. Putting the triplets into the cosine–margin–triplet loss function.

Recent research on triplet loss has shown state-of-the-art results in the fields of image recognition and text classification [23,51,53].

4. Proposed Approach

In this section, we describe TCCCD, a cross-language code-clone-detection approach that utilizes the pre-trained model UniXcoder and triplet learning.

4.1. Overall Framework

Figure 2 presents an overview of our approach. In our study, we began by performing data processing as the initial step. The data were preprocessed using UniXcoder [21], which was fine-tuned, acting as a tokenizer to transform each code snippet from the source code into aggregated sequences. Then, these sequences were grouped into batches, where each batch consisted of code pairs belonging to the same task, but in different programming languages, as well as code pairs from diverse tasks using different programming languages.

Following that, we employed UniXcoder to convert each processed sequence into its corresponding code representation. Lastly, we applied triplet learning to encode the code snippets into a high-dimensional embedding space, thereby ensuring that the anchor and positive samples were brought closer together, while the anchor and negative samples were pushed further apart. Our approach can be summarized into main three parts: data processing, code representation, and triplet learning.

Detailed descriptions of these components will be provided in the subsequent sections.

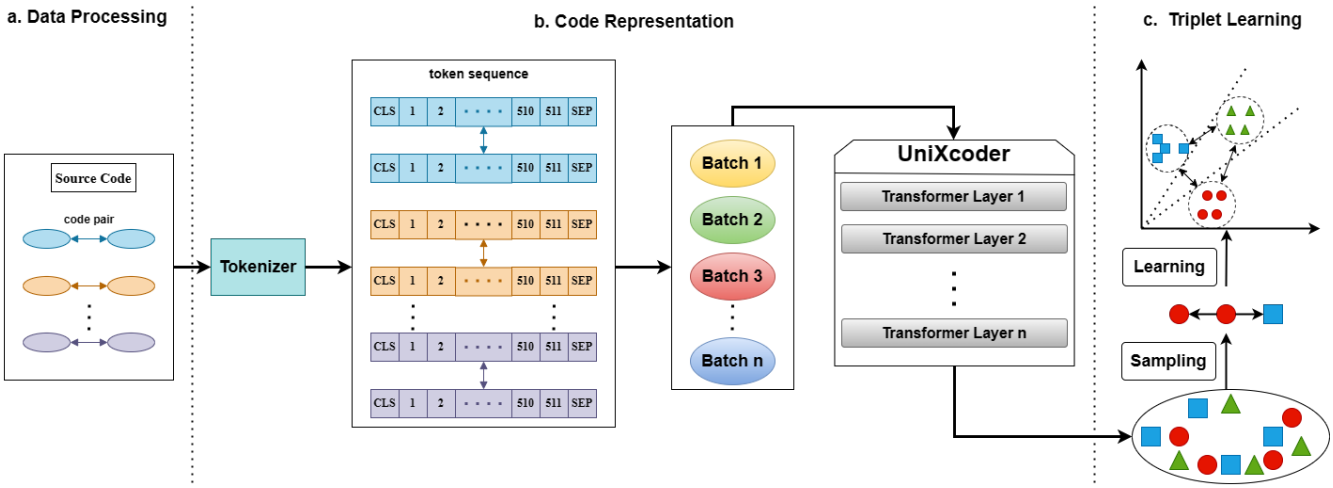

Figure 2. Overview of TCCCD. (In Process (**a**), the same color pairs represent the code pairs belonging to the same task, but in different programming languages. In Process (**b**), the same color token sequences represent the sequences belonging to the same task, but in different programming languages. After tokenization, UniXcoder converts the sequences into code representation. In Process (**c**), the symbols with the same color and shape stand for code representation vectors belonging to the same task. Next, they are selected to form triplets, and after triplet learning, they are effectively classified in the vector space).

4.2. Data Processing

The data-processing stage involved providing detailed information about the dataset and describing the specific data-handling procedures. A comprehensive description of the dataset and code clone pairs is presented in Section 4.1. The dataset we utilized is composed of code clone pairs, where code clone pairs refer to combinations of two code fragments that exhibit similar structures and functionalities. Code clone pairs were used as the inputs for our proposed approach.

4.3. Code Representation

After the data processing step, we fine-tuned the UniXcoder pre-trained model to obtain more-effective code representation. Our fine-tuning procedure included the following steps:

1. **Tokenization:** Code snippets were subjected to a tokenization process, whereby UniXcoder was utilized as the tokenizer to segment the source code snippets into subwords, resulting in the transformation of the source code into a words list. When UniXcoder tokenizes the source code, it segments source code snippets into subwords. These subwords can include identifiers, keywords, operators, and more from the source code. This tokenization process helps to illustrate the abstract hierarchy of the source code, transforming it into smaller semantic units for subsequent representation and learning. The list of these subwords forms an abstract representation of the source code, providing a more-informative input for our method.
2. **Sequence aggregation:** After tokenization, special tokens [CLS] (abbreviation of Classification) and [SEP] (abbreviation of Separator) are incorporated into the vocabulary

list, with the [CLS] token marking the beginning of the source code and the [SEP] token indicating the end of the source code, respectively. Finally, the tokenized source code is represented as a sequence of tokens, denoted as $[CLS, \omega_1, \omega_2, \omega_3, ..., \omega_n, SEP]$. Then, we replaced the subwords of each code snippet with their corresponding IDs using the tokenizer function and, then, aggregated them into sequences. Because the maximum input token sequence length for the pre-trained model UniXcoder is 512, the token sequence in our approach was also limited to a maximum length of 512.

3. **Batch creation:** To enhance efficiency in processing and training, we created batches comprising the aggregated sequences. These sequences were grouped into batches, where each batch consisted of pairs of sequences belonging to the same task, but in different programming languages, as well as pairs of sequences from diverse tasks using different programming languages.
4. **Vector generation:** We took the sequences within a batch as the inputs to the UniXcoder model, and the output of UniXcoder included both the contextual vector representations for each token and the sentence representation corresponding to the [CLS] token, which represents the entire code snippet.

During the fine-tuning process, we applied various data-processing techniques and adjusted the model parameters to obtain more-effective code representation. We preprocessed the dataset to consist of clone code pairs from the same task, but implemented in different programming languages, making it better suited for our proposed approach. Then, we fine-tuned the parameters of UniXcoder to achieve optimal effectiveness, including parameters such as the learning rate and batch size. More-specific details can be found in Section 4.3. Additionally, by learning to minimize the triplet loss, the UniXcoder model can enhance its representational capacity, thereby improving its effectiveness.

4.4. Triplet Learning

We propose an effective triplet-learning method that combines distance-weighted sampling [22] in triplet learning with the cosine–margin–triplet loss [23]. Our objective was to select the most-suitable negative samples for each anchor sample and, then, feed the selected triplets into the cosine–margin–triplet loss function.

The selection probability of negative samples is determined by Equation (1), which utilizes the softmax function to convert the distance between negative samples and anchor samples into a probability distribution. The calculation of the selection probability was based on the Euclidean distance between samples, where larger distances correspond to higher probabilities.

Our probability of negative samples selected can be expressed as Equation (2):

$$P_{negative}(i) = \frac{E(D_{pos}(X_i^a, X_i^n))}{\sum_{j=1}^{r} E(D_{pos}(X_i^a, X_j^n))}. \quad (2)$$

In this equation, $P_{negative}$ is the probability of negative samples selected and i is the number of the currently selected samples.

r is the total count of all negative samples in a batch.
X^a and X^n are anchor samples and negative samples, respectively (i.e., X^a and X^n form a non-clone pair).
The D_{pos} function represents the square of the Euclidean distance between samples. The E function is the softmax function.

The distance metric function D_{pos} used to measure the square of the Euclidean distance between two samples (x and y represent two samples) is defined as Equation (3):

$$D_{pos}(x, y) = \|\mathbf{x} - \mathbf{y}\|^2. \quad (3)$$

The triplet loss function is applied after selecting the negative samples based on the probabilities and finding the positive samples from the same task for each anchor sample.

The cosine–margin–triplet loss function calculates the cosine similarity between embedding vectors and introduces a margin parameter (m) and a scaling factor (s) to penalize positive and negative samples that do not meet the desired similarity boundary. This encourages the embedding vectors to cluster and separate samples more effectively in the embedding space. The cosine-margin–triplet loss function is defined as Equation (4):

$$Loss = -\frac{1}{N}\sum_{i}^{N} log\frac{E(s(cos(X_i^a, X_i^p) - m))}{E(s(cos(X_i^a, X_i^n))) + E(s(cos(X_i^a, X_i^p) - m))}. \quad (4)$$

In Equation (4), N represents the number of all triples.

m represents the minimum angular distance between positive and negative samples.
s is the scaling factor used to adjust the scaling of the exponential term.
X^p is the positive sample (i.e., X^a and X^p form a clone pair).

The following Algorithm 1 provides a detailed description of the process for selecting negative samples and constructing the loss function.

Algorithm 1: Calculation of triplet loss.

input : A batch of clone code pairs, B
output: Triplet loss, L_T

for *each pair* E_i *in* B **do**
  $X_i^a, X_i^p \leftarrow E_i$;
  $S_{cos_p} = E(s(cos(X_i^a, X_i^p) - m))$;
  for *each pair* $E_j (i \neq j)$ *in* B **do**
    $X_j^{n_1}, X_j^{n_2} \leftarrow E_i$;
    $D_{sum} = D_{sum} + E(D_{pos_j}(X_i^a, X_j^{n_1})) + E(D_{pos_j}(X_i^a, X_j^{n_2}))$;
    $Set_{neg} \leftarrow X_j^{n_1}, X_j^{n_2}$;
  for *each negative sample* e **do**
    $P_{neg}(i) = \frac{E(D(X_i^a, h_i))}{D_{sum}}$;
    $X_i^n = S(p_{neg})$;
    $S_{cos_n} = E(s(cos(X_i^a, X_i^n)))$;
    $L_{T_i} = -log(\frac{S_{cos_p}}{S_{cos_n} + S_{cos_p}})$;
$L_T = \frac{\sum_{i=1}^{len(B)} L_{T_i}}{len(B)}$

For each processed batch, B denotes the number of token sequence instances converted from cloned code within the batch. The traversal of this batch is conducted iteratively for processing purposes. In each pair, the initial instance is designated as the anchor sample, while the subsequent instance assumes the role of the positive sample. Ideally, the remaining pairs of cloned code instances are utilized as negative samples, resulting in an aggregate count of 2*(B-1) negative examples. Subsequently, an assessment of the distances is carried out between all negative samples and one anchor sample. These distances are then subjected to a softmax transformation, yielding probabilities. The probability of selecting a negative sample instance increases proportionally with the magnitude of the distance between the negative sample instance and the anchor sample. Ultimately, the selection of a negative sample instance is contingent upon these probabilities.

After selecting the anchor samples, positive samples, and negative samples, we combined them to form triplets and processed them using the loss function. Finally, the loss values obtained for each triplet were summarized or averaged to form the overall loss for the entire batch. By following this process of handling triplets, we can effectively handle the anchor sample, select the positive sample, apply the negative-sample-selection strategy,

and train the model using the loss function, thereby improving the feature representation capability of the model.

During the fine-tuning process, we conducted experiments using the same hyperparameters as those used in the paper on UniXcoder [21]. Subsequently, we optimized TCCCD using the triplet loss, and the results demonstrated a significant improvement in effectiveness.

5. Experiment Setup

When it comes to cross-language code clone detection, we propose several Research Questions (RQs) to evaluate the effectiveness of our approach:

1. **RQ1: baselines' effectiveness comparison.** To gauge our approach's competitiveness in the cross-language code-clone-detection domain, we aimed to assess its effectiveness relative to state-of-the-art methods using the metrics described in Section 4.2 as the criteria. Hence, we formulated **RQ1**: "How does our approach perform compared to the state-of-the-art approaches in cross-language code clone detection?"
2. **RQ2: effectiveness on specific language pairs.** In practical development scenarios, developers often require code clone detection across various programming languages. To assess the effectiveness of our approach in detecting code clones across specific, distinct programming languages, we pose **RQ2:** "Does our approach effectively detect code clones across different programming languages?"
3. **RQ3: impact of different components.** Our approach comprises multiple components, including triplet learning and distance-weighted sampling. By analyzing the influence of these diverse components on the effectiveness of our approach, we can gain a better understanding of which components are most crucial for cross-language code clone detection. This understanding, in turn, guided our selection and optimization. Therefore, we present **RQ3**: "What is the impact of different components in our approach on the effectiveness of cross-language code clone detection?"

5.1. Baselines

CLCMiner [29] is an advanced technology for cross-language code clone detection that utilizes token sequences to represent the syntactic and structural information of code fragments. It leverages modification history and window algorithms to detect clones effectively.

CLDCDSA [15] is a tool that utilizes techniques involving the analysis of syntactic features of source code and measuring cross-language API call similarity to detect cross-language code clones.

C4 [16] is a state-of-the-art approach for clone detection that effectively detects clone code. It leverages the pre-trained model CodeBERT to convert programs in different languages into high-dimensional vector representations. Additionally, the C4 model is fine-tuned using a contrastive learning objective, enabling it to accurately identify clone pairs and non-clone pairs.

CLCDSA [15] proposes three datasets for cross-language clone detection, sourced from three open-source programming competition websites (AtCoder, Google CodeJam, and CoderByte), comprising over 78,000 solutions. To construct these datasets, researchers collect at least 20 fully accepted solutions for each problem statement, spanning the Java, C#, C++, and Python programming languages. Accepted solutions for the same problem statement are considered functional clones, while solutions for different problem statements are considered non-clone pairs. These datasets are regarded as validated cross-language clone databases in various aspects.

In the construction of triplets for TCCCD, we selectively chose code pairs representing solutions for the same problem in different programming languages as positive and anchor samples in the triplets, without explicitly constructing negative samples. During training, we adopted a sampling strategy to select a code segment from a different problem within the same batch as the negative sample in the triplets. As a result, our dataset consisted

of clone pairs comprising positive and anchor samples in the triplets. The training set contained 69,424 clone pairs, the testing set 8692 clone pairs, and the validation set 8556 clone pairs.

The same dataset as used in TCCCD was also employed in C4 since it utilizes the N-pair approach, which only requires clone pairs as the inputs.

For CLCDSA and CLCMiner, we followed the dataset setup described in CLCDSA. We needed to augment the dataset consisting only of clone pairs by adding non-clone pairs. Finally, our training set contained 138,848 code pairs, the validation set 17,384 code pairs, and the test set 17,112 code pairs, with a positive clone pair ratio of 50%.

The precise statistics of the experimental dataset can be observed in Table 1. According to the statistical data, C++ code snippets had the highest count, reaching 32,281, which was roughly twice the count of code snippets in other programming languages. Python code exhibited the lowest average number of lines, which can be attributed to the language's inherent simplicity. The average number of lines and tokens in C# code blocks was the highest among all programming languages, due to the more-complex syntax structures and a limited number of standard library functions in the C# language.

Table 1. Statistics for specific language code snippets.

Language	Code Snippets	Average Lines	Average Tokens
Java and Python	18,836	104	686
Python	18,331	26	191
C++	32,281	66	556
C#	16,359	111	855

5.2. Metrics

In our experiment, the Precision (P), Recall (R), and F-measure (F1) were used as the metrics to evaluate the effectiveness of code clone detection.

True Positive (TP) is the count of correctly classifying positive class samples as positive. True Negative (TN) is the count of correctly classifying negative class samples as negative. False Positive (FP) is the count of incorrectly classifying negative class samples as positive. False Negative (FN) is the count of incorrectly classifying positive class samples as negative.

Precision measures the proportion of correctly predicted positive samples by the classifier. It is calculated as the ratio of TPs to the sum of TPs and FPs. In code clone detection, precision represents the percentage of accurately detected clones among the samples that include both clones and non-clones. If T_p is a TP and F_p is an FP clone, Equation (5) for the Precision (P) is:

$$P = \frac{T_p}{T_p + F_p}. \tag{5}$$

Recall measures the proportion of actual positive samples that are correctly predicted as positive by an approach. In code clone detection, recall represents the percentage of detected clone code out of all actual clone code instances. If T_p is the number of detected TP clones and F_n is the number of detected FP clones, Equation (6) for the Recall (R) is:

$$R = \frac{T_p}{T_p + F_n}. \tag{6}$$

The F-measure (F1-score) is a comprehensive evaluation metric that takes into account both the precision and recall. If P represents precision and R represents recall, Equation (7) for the F-measure (F_1) is:

$$F_1 = \frac{2PR}{P + R} \tag{7}$$

In the context of code clone detection, the precision, recall, and F1-score serve as vital metrics for measurement. Precision indicates the accuracy of labeling genuine code

clone fragments, ensuring that the identified code segments indeed exhibit duplication and merit the attention of developers. This aids in keeping the development team focused on addressing real issues, without being distracted by false clone alerts. Recall measures the proportion of genuinely cloned code instances successfully detected by the model among all true clones. A low recall implies that the model may have missed numerous genuinely existing code clones, leaving these clones unattended and unaddressed. The F1-score strikes a balance between precision and recall while considering both FPs and FNs, providing a critical and versatile metric for comprehensive effectiveness evaluation.

In the field of cross-language clone detection, ensuring high precision is of paramount importance. False positives are a particularly concerning issue as they can lead to developers spending a significant amount of time and effort examining and handling code that is not actually cloned. When balancing high precision against recall and the F1-score, we believe that high precision takes precedence, and a slight compromise in recall and the F1-score may be acceptable in the pursuit of achieving high precision.

By computing these metrics, the effectiveness of a approach in clone detection tasks can be assessed. Higher values of the precision, recall, and F-measure indicate the better effectiveness of the approach.

5.3. Experiment Settings

All experiments were conducted on a server using the Linux system with 32 cores running at 2.1 GHz CPU and an RTX 3090 graphics card. For the training process of our approach, we set the batch size to 16, and for the cosine–margin–triplet loss, we set S to 16.0 and m to 0.2. For distance-weighted sampling, we selected four times from the negative sample weight set within the same batch to form the negative samples for the triplets. Additionally, we used the Adam optimizer for training, performing 10 epochs, and each epoch took approximately 40 min. The learning rate was set to 0.00004.

As for the settings of the baseline approaches, we followed the descriptions provided in their original papers or published code [15,16,29].

6. Experimental Results

In this section, we analyze the experimental results to gain a comprehensive understanding of the effectiveness of our approach in cross-language code clone detection.

6.1. RQ1: Baselines' Effectiveness Comparison

Under the same experimental environment and settings, we conducted experiments on several state-of-the-art baseline approaches, including CLCMiner [29], CLCDSA [15], and C4 [16], and compared them with our proposed approach. To evaluate the experimental results, the precision, recall, and F1-score were used as the effectiveness metrics.

The experimental results are presented in Table 2, which clearly shows that our proposed approach outperformed the baseline methods in terms of the precision and F1-score, and CLCDSA tended to consider most code pairs as clones.

Table 2. Effectiveness comparison of TCCCD and baselines.

Approach	Precision	Recall	F1
CLCMiner	0.37	0.59	0.46
CLCDSA	0.53	0.6	0.67
C4	0.93	0.90	0.92
TCCCD	**0.96**	0.91	**0.93**

Due to significant syntactic and structural differences between programming languages, CLCMiner failed to fully capture the similarity between cross-language code fragments using simple token sequences. Furthermore, the adaptation of the modification history and window algorithms in CLCMiner may suffer from inadequate adaptability to different programming languages, leading to suboptimal detection effectiveness.

CLCDSA utilizes syntactical features and API documentation information to represent code snippets, which provides stronger expressive power and discrimination in code representation compared to the simple token sequence representation method used in CLCMiner.

Compared to CLCMiner and CLCDSA, C4 demonstrated significant advancements in code representation and vector similarity contrastive learning. By leveraging the pre-trained model CodeBERT and contrastive learning, C4 can more comprehensively capture the semantic information of code in cross-language code clone detection, resulting in significant effectiveness in the precision and F1-score.

Our approach clearly outperformed the baseline approaches, with an improvement in the F1-score from 0.92 to 0.93, precision from 0.94 to 0.96, and recall from 0.90 to 0.91.

6.2. RQ2: Effectiveness on Specific Language Pairs

The results presented in Table 3 demonstrate the effectiveness of various methods on specific cross-language pairs, including CLCMINER, CLCDSA, C4, and TCCCD. Among these methods, TCCCD exhibited the best effectiveness.

Table 3. Effectiveness on specific cross-language pairs.

Language	CLCMiner			CLDSA			C4			TCCCD		
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
Java and Python	0.36	0.57	0.44	0.49	0.93	0.60	0.95	0.93	0.94	0.96	0.92	0.94
Java and C#	0.38	0.60	0.47	0.58	0.97	0.72	0.95	0.93	0.94	0.98	0.90	0.94
Python and C#	0.35	0.57	0.43	0.48	0.98	0.64	0.92	0.95	0.94	0.95	0.91	0.92
Python and C++	0.36	0.56	0.44	-	-	-	0.94	0.92	0.93	0.94	0.92	0.93
Java and C++	0.38	0.59	0.46	-	-	-	0.93	0.90	0.91	0.96	0.90	0.93
C++ and C#	0.39	0.61	0.48	-	-	-	0.91	0.88	0.89	0.97	0.90	0.93

In comparison to CLCMiner, TCCCD exhibited significant effectiveness in all specific cross-language code pairs, with an average increase of 0.59 in precision, 0.33 in recall, and 0.48 in the F1-score.

Compared to CLCDSA, our approach showed similar recall on Java and Python clone pairs, but CLCDSA had a higher average recall, indicating that it tended to classify more input code pairs as clones. However, our approach outperformed CLCDSA in the precision and F1-score metrics, with an average increase of 0.44 in the precision and 0.27 in the F1-score.

Based on the experimental results, we observed that our approach consistently outperformed C4 in terms of the precision, recall, and F1-score on each specific cross-language pair, with a notable 0.06 increase in precision for the C++ and C# cross-language pairs. This improvement can be attributed to the fact that UniXcoder was pre-training on data that include C++ and C# code, while CodeBERT lacked such pre-training, making UniXcoder more effective in representing C++ and C# code. Additionally, the adoption of UniXcoder and triplet learning in TCCCD contributed to its superior effectiveness over the baseline methods across all specific cross-language pairs.

6.3. RQ3: Impact of Different Components

Our proposed approach in this paper consisted of two steps: the first step involved processing the source code into sequences using UniXcoder, and the second step involved fine-tuning the model using triplet learning. We designed the following ablation experiments to signify the importance of each module of TCCCD.

We designed the following experiments to compare with our approach.

6.3.1. Fine-Tuning of the Model

In Table 4, we present the results of model fine-tuning. We designed two methods to validate the effectiveness of model fine-tuning. The first method directly uses the UniXcoder model to detect code clones without fine-tuning. The second method fine-tunes

the UniXcoder model on the training set, representing the source code and using the Mean-Squared Error (MSE) as the loss function. Upon observing the experimental results, we found that fine-tuning had a significant positive impact on our approach. In the method that directly uses the UniXcoder model to detect cross-language code clones, the F1-score was only 0.67, with the precision and recall being 0.5 and 1, respectively. This indicated that the method considered all input code pairs as clones. However, with fine-tuning of UniXcoder, the effectiveness greatly improved, with an increase of 0.23 in the F1-score and 0.35 in the precision.

Table 4. Fine-tuning of the model.

Approach	Precision	Recall	F1
Without fine-tuning	0.50	**1.00**	0.67
Fine-tuning	**0.85**	0.95	**0.90**

6.3.2. Distance Metric Learning

Based on our analysis of the experimental results in Table 5, we concluded that triplet learning demonstrated excellent effectiveness in cross-language code clone detection. This was attributed to triplet learning's ability to select a challenging negative sample and train it alongside an anchor sample and a positive sample, forming a triplet. This approach effectively facilitated the learning of subtle distinctions between clones and non-clones.

Table 5. Distance metric learning.

Approach	Precision	Recall	F1
MSE	0.85	**0.95**	0.90
Contrastive learning	0.94	0.91	0.92
Triplet learning	**0.96**	0.91	**0.93**

In Table 5, we demonstrate the impact of distance metric learning on cross-language code clone detection. We conducted experiments by replacing the triplet loss function used in TCCCD with the Mean-Squared Error (MSE) loss function and the contrastive loss function separately. For cross-language code clone detection, triplet learning showed the best effectiveness in terms of the precision and F1-score, achieving 0.96 in the precision, 0.93 in the F1-score, and 0.91 in the recall.

6.3.3. Triple Sampling Method in Triplet Learning

We observe a significant improvement in the effectiveness of triplet learning when using distance-weighted sampling in Table 6. We attributed this improvement to the fact that distance-weighted sampling allowed the model to select samples with varying degrees of similarity based on the distances between them. This included selecting challenging samples (those with high similarity between positive and negative instances), as well as normal samples (those with low similarity). The selection of these samples was crucial for model training as it enhanced the model's discriminative capabilities.

Table 6. Triple sampling method.

Approach	Precision	Recall	F1
Random sampling	0.94	**0.92**	0.92
Distance-weighted sampling	**0.96**	0.91	**0.93**

In Table 6, we present the impact of the triplet sampling methods in triplet learning. For the triplet learning strategy used in TCCCD, we conducted experiments with two different sampling methods: random sampling and distance-weighted sampling. We compared

their respective effects on the model's effectiveness. In TCCCD, the distance-weighted sampling method was more effective in identifying code pairs as triplets, outperforming the random sampling method in terms of the precision and F1-score.

7. Discussion

In this section, we discuss the effectiveness of TCCCD on a small dataset and the reasons for choosing UniXcoder for the code representation. Through these discussions, we aimed to gain a deeper understanding of the limitations and advantages of our approach.

7.1. Effectiveness of TCCCD on a Small Dataset

We conducted an effectiveness analysis of TCCCD on datasets of various sizes, including 10%, 30%, 50%, 70%, and 90% of the original data, and investigated the impact of dataset size on its results. Figure 3 shows the results of the effectiveness of TCCCD on a small dataset. When using 70% of the original data, TCCCD's precision, recall, and F1-score tended to stabilize. On the reduced dataset containing 30% of the original data, its effectiveness was slightly lower than that on the full dataset, but still surpassed the effectiveness of existing baseline methods.

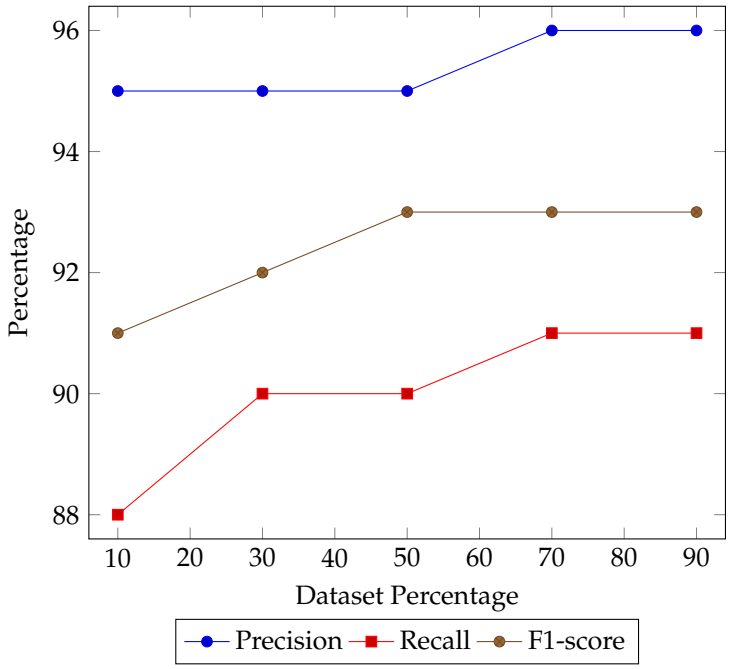

Figure 3. Effectiveness on different dataset percentages.

When evaluating on a small-scale dataset containing only 10% of the original data, TCCCD achieved relatively lower effectiveness with a precision, recall, and F1-score of 0.95, 0.91, and 0.88, respectively. However, as the training data were increased to include 30% of the original data, the precision remained unchanged, while the recall and F1-score both improved, reaching 0.90 and 0.92, respectively. This indicated that TCCCD can maintain a high level of effectiveness and recall even with reduced data, highlighting its robustness and generalization capability on small-scale datasets.

When the data were further increased to include 70% of the original data, the method's precision, recall, and F1-score became relatively stable. It is worth noting that our approach's precision showed relatively stable effectiveness across various percentages of the original dataset. The F1-score gradually improved as the dataset percentage increased from 10% to 50%, while the recall exhibited a significant improvement in two ranges: from 10% to 30% and from 50% to 70% of the original dataset.

The ability of TCCCD to perform well on small-scale datasets is crucial. This indicates that our approach does not excessively rely on large amounts of data, making it applicable in scenarios where data collection resources are limited or challenging. Furthermore, the consistent effectiveness on the reduced datasets demonstrated that TCCCD remained a reliable solution even in situations with limited data availability.

Overall, these results demonstrated the efficacy and robustness of TCCCD, making it a promising approach for cross-language code clone detection even in resource-constrained environments.

7.2. Why We Chose UniXcoder for Code Representation

One significant advantage of using UniXcoder [21] for code representation lies in its exceptional cross-language generalization capability. UniXcoder is a pre-trained code representation model that has been trained on vast amounts of code data, allowing it to learn the syntactic structures and semantic information of multiple programming languages. Since UniXcoder has already encompassed diverse programming languages during its training, it can automatically adapt to different languages when performing code representation, eliminating the need for prior uniform and rigorous data preprocessing. UniXcoder effectively captures the distinctive features and structures of code, enabling consistent representation and transformation across various programming languages.

We employed CodeBERT for code representation and fine-tuned it using the same distance-weighted sampling-triplet-learning technique. Subsequently, we performed an effectiveness comparison with TCCCD based on the results. Regarding the dataset, we trained CodeBERT on the dataset described in Section 4.1, which was the same dataset used in TCCCD.

In our experiments, as presented in Table 7, we conducted comparative trials between CodeBERT and UniXcoder, employing triplet learning for optimization. The results highlighted UniXcoder's superiority in terms of the precision and F1-score. UniXcoder demonstrated impressive effectiveness in multiple downstream tasks related to code processing. As a result, we chose UniXcoder as our pre-trained model for code representation.

Table 7. Pre-trained model effectiveness.

Approach	Precision	Recall	F1
CodeBERT	0.94	**0.91**	0.92
UniXcoder	**0.96**	0.91	**0.93**

The emergence of TCCCD has a multifaceted impact on both researchers and industry practitioners. Here are some of the primary impacts:

1. Innovative approach: TCCCD introduces a novel and highly efficient method for addressing cross-language code clones, providing researchers with a platform to explore new technologies and algorithms.
2. Enhanced effectiveness: For industry professionals, TCCCD significantly improves the accuracy of code clone detection, thereby reducing false positives and false negatives. This has profound implications for software development and maintenance.
3. Multilingual support: TCCCD's cross-language capabilities open up new possibilities for research in multilingual software projects, encouraging researchers to investigate clone relationships across different programming languages. For multilingual software development teams, TCCCD offers an effective solution for addressing clone-related challenges among diverse languages, ultimately reducing the complexity of software maintenance.

8. Threats to Validity

In our study, several threats to the validity pose potential risks to the effectiveness of our research, categorically falling into two domains: internal validity and external validity.

Internal validity pertains to factors within the study that could influence the results, potentially leading to internal inaccuracies or unreliability. On the other hand, external validity relates to the generalizability of the research findings to broader contexts.

Specifically, within the domain of internal validity, we contend with the following threats: the challenge of handling untrained programming languages and the limitation of input length. In terms of external validity, our study faces the threat of having a limited set of programming languages within the experimental dataset.

8.1. Internal Validity

1. Challenge of handling untrained programming languages: The potential issue arises when the pre-training of the model does not cover certain specific programming languages. In such cases, the pre-trained model may struggle to effectively comprehend the syntax and semantics of these untrained programming languages, resulting in a decline in the quality of code representations for those languages. To mitigate this issue, it is advisable to consider incorporating a more-diverse range of untrained programming languages into the pre-trained model to expand its language coverage.
2. Limitation of input length: The limitation of pre-trained models to handle input code sequences with a maximum length of 512 tokens can pose challenges when using them for code representation. When processing longer code sequences, the truncation of the code may be necessary, leading to potential loss or incompleteness of code information. This limitation could impact the quality and effectiveness of code representations, especially for tasks or dataset that involve longer code sequences. When confronted with lengthy code sequences, a practical approach is to partition them into shorter segments, which can then be individually processed by the model.

8.2. External Validity

Limited programming languages in the experimental dataset: The experimental dataset included only four programming languages. As a result, we cannot fully assess the effectiveness of the method on other programming languages. It remains uncertain how well the method would generalize across a broader set of programming languages. In the real world, different programming languages exhibit diverse code features and structures, and a dataset restricted to a few programming languages may not sufficiently represent the method's effectiveness across various languages.

To address this issue, one potential approach is to consider augmenting the existing dataset by utilizing synthetic data or leveraging multi-language solutions available on programming competition websites.

9. Conclusions

In this paper, we proposed the TCCCD model to address the challenges of cross-language code clone detection. Leveraging the UniXcoder pre-trained model, we mapped programs written in different programming languages into the same vector space and learned their code representations. Through fine-tuning using triplet learning, we further enhanced the effectiveness of the TCCCD model in cross-language clone detection.

In summary, our main contributions can be outlined as follows: (i) Firstly, we proposed a novel and effective method, TCCCD, for cross-language code clone detection, leveraging UniXcoder to efficiently represent code. (ii) Secondly, we introduced the triplet-based learning approach in the context of cross-language code clone detection and validated its effectiveness. (iii) Lastly, we evaluated the effectiveness of TCCCD on publicly available datasets, with experimental results demonstrating its outstanding effectiveness in the field of clone detection, surpassing existing methods.

The experimental results demonstrated that TCCCD outperformed existing baselines significantly, achieving precision, recall, and F-measure scores of 0.96, 0.92, and 0.93, respectively, showcasing its outstanding effectiveness in this domain.

In future work, we will focus on exploring novel and effective pre-trained models and evaluating their effectiveness when fine-tuned using the triplet learning method we proposed. Additionally, our research will involve expanding the diversity of programming languages in existing datasets and ensuring a balanced representation of samples from different languages. This will enable us to conduct a more-comprehensive assessment of the generalization capabilities of the methods in the field of cross-language code clone detection.

Author Contributions: Conceptualization, F.Z.; Methodology, F.Z.; Software, F.Z.; Validation, Y.F.; Formal analysis, Y.X.; Investigation, Y.F.; Resources, Y.F.; Data curation, Y.X.; Writing—original draft, F.Z.; Supervision, Z.L.; Project administration, Z.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the National Science Foundation of China (U20B2045).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Roy, C.K.; Cordy, J.R. A survey on software clone detection research. *Queen's Sch. Comput. TR* **2007**, *541*, 64–68.
2. Antoniol, G.; Villano, U.; Merlo, E.; Di Penta, M. Analyzing cloning evolution in the linux kernel. *Inf. Softw. Technol.* **2002**, *44*, 755–765. [CrossRef]
3. Dang, Y.; Ge, S.; Huang, R.; Zhang, D. Code clone detection experience at Microsoft. In Proceedings of the 5th International Workshop on Software Clones, Waikiki, HI, USA, 23 May 2011; pp. 63–64.
4. Bellon, S.; Koschke, R.; Antoniol, G.; Krinke, J.; Merlo, E. Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.* **2007**, *33*, 577–591. [CrossRef]
5. Juergens, E.; Deissenboeck, F.; Hummel, B.; Wagner, S. Do code clones matter? In *Proceedings of the IEEE 31st International Conference on Software Engineering*; IEEE: Piscataway, NJ, USA, 2009; pp. 485–495.
6. Kamiya, T.; Kusumoto, S.; Inoue, K. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.* **2002**, *28*, 654–670. [CrossRef]
7. Jiang, L.; Misherghi, G.; Su, Z.; Glondu, S. Deckard: Scalable and accurate tree-based detection of code clones. In Proceedings of the 29th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA, 20–26 May 2007; pp. 96–105.
8. Roy, C.K.; Cordy, J.R. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In Proceedings of the 16th IEEE International Conference on Program Comprehension, Amsterdam, The Netherlands, 1–13 June 2008; pp. 172–181.
9. Sajnani, H.; Saini, V.; Svajlenko, J.; Roy, C.K.; Lopes, C.V. SourcererCC: Scaling code clone detection to big-code. In Proceedings of the 38th International Conference on Software Engineering, Austin, TX, USA, 14–22 May 2016; pp. 1157–1168.
10. Li, L.; Feng, H.; Zhuang, W.; Meng, N.; Ryder, B. CClearner: A deep learning-based clone detection approach. In Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), Shanghai, China, 17–22 September 2017; pp. 249–260.
11. Zou, Y.; Ban, B.; Xue, Y.; Xu, Y. CCGraph: A PDG-based code clone detector with approximate graph matching. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, Virtual, 21–25 December 2020; pp. 931–942.
12. Zhang, J.; Wang, X.; Zhang, H.; Sun, H.; Wang, K.; Liu, X. A novel neural source code representation based on abstract syntax tree. In Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, Canada, 25–31 May 2019; pp. 783–794.
13. Zhao, G.; Huang, J. Deepsim: Deep learning code functional similarity. In Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Lake Buena Vista, FL, USA, 4–11 November 2018; pp. 141–151.
14. Vislavski, T.; Rakić, G.; Cardozo, N.; Budimac, Z. LICCA: A tool for cross-language clone detection. In Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Campobasso, Italy, 20–23 March 2018; pp. 512–516.
15. Nafi, K.W.; Kar, T.S.; Roy, B.; Roy, C.K.; Schneider, K.A. Clcdsa: Cross language code clone detection using syntactical features and api documentation. In Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 11–15 November 2019; pp. 1026–1037.
16. Tao, C.; Zhan, Q.; Hu, X.; Xia, X. C4: Contrastive cross-language code clone detection. In Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, Pittsburgh, PA, USA, 16–17 May 2022; pp. 413–424.

17. Rakić, G. Extendable and Adaptable Framework for Input Language Independent Static Analysis. Ph.D. Thesis, University of Novi Sad, Novi Sad, Serbia, 2015.
18. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. Codebert: A pre-trained model for programming and natural languages. *arXiv* **2020**, arXiv:2002.08155.
19. Wang, K.; Yan, M.; Zhang, H.; Hu, H. Unified abstract syntax tree representation learning for cross-language program classification. In Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, Pittsburgh, PA, USA, 16–17 May 2022; pp. 390–400.
20. Wang, W.; Li, G.; Ma, B.; Xia, X.; Jin, Z. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In Proceedings of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), London, ON, Canada, 18–21 February 2020; pp. 261–271.
21. Guo, D.; Lu, S.; Duan, N.; Wang, Y.; Zhou, M.; Yin, J. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv* **2022**, arXiv:2203.03850.
22. Wu, C.-Y.; Manmatha, R.; Smola, A.J.; Krahenbuhl, P. Sampling matters in deep embedding learning. In Proceedings of the IEEE International Conference on Computer Vision, Venice, Italy, 22–27 October 2017; pp. 2840–2848.
23. Unde, A.S.; Rameshan, R.M. MOTS R-CNN: Cosine-margin-triplet loss for multi-object tracking. *arXiv* **2021**, arXiv:2102.03512.
24. Yuan , Y.; Guo , Y. Boreas: An accurate and scalable token-based approach to code clone detection. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, Essen, Germany, 3–7 September 2012; pp. 286–289.
25. Shawky, D.M.; Ali, A.F. An approach for assessing similarity metrics used in metric-based clone detection techniques. In Proceedings of the 3rd International Conference on Computer Science and Information Technology, Chengdu, China, 9–11 July 2010; pp. 580–584.
26. Yuan , Y.; Guo , Y. CMCD: Count matrix based code clone detection. In Proceedings of the 18th Asia-Pacific Software Engineering Conference, Chi Minh, Vietnam, 5–8 December 2011; pp. 250–257.
27. Mou, L.; Li, G.; Jin, Z.; Zhang, L.; Wang, T. TBCNN: A tree-based convolutional neural network for programming language processing. *arXiv* **2014**, arXiv:1409.5718.
28. Hua, W.; Sui, Y.; Wan, Y.; Liu, G.; Xu, G. FCCA: Hybrid code representation for functional clone detection using attention networks. *IEEE Trans. Reliab.* **2020**, *70*, 304–318. [CrossRef]
29. Cheng, X.; Peng, Z.; Jiang, L.; Zhong, H.; Yu, H.; Zhao, J. CLCMiner: Detecting cross-language clones without intermediates. *IEICE Trans. Inf. Syst.* **2017**, *100*, 273–284. [CrossRef]
30. White, M.; Vendome, C.; Linares-Vásquez, M.; Poshyvanyk, D. Toward deep learning software repositories. In Proceedings of the IEEE/ACM 12th Working Conference on Mining Software Repositories, Florence, Italy, 16–17 May 2015; pp. 334–345.
31. Bhoopchand, A.; Rocktäschel, T.; Barr, E.; Riedel, S. Learning python code suggestion with a sparse pointer network. *arXiv* **2016**, arXiv:1611.08307.
32. Dam, H.K.; Tran, T.; Pham, T. A deep language model for software code. *arXiv* **2016**, arXiv:1608.02715.
33. Mikolov, T.; Chen, K.; Corrado, G.; Dean, J. Efficient estimation of word representations in vector space. *arXiv* **2013**, arXiv:1301.3781.
34. Pennington, J.; Socher, R.; Manning, C.D. Glove: Global vectors for word representation. In Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP), Doha, Qatar, 25–29 October 2014; pp. 1532–1543.
35. Devlin, J.; Chang, M.-W.; Lee, K.; Toutanova, K. BERT: Pre-training of deep bidirectional Transformers for language understanding. *arXiv* **2018**, arXiv:1810.04805.
36. Mashhadi, E.; Hemmati, H. Applying codebert for automated program repair of java simple bugs. In Proceedings of the IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), Madrid, Spain, 17–19 May 2021; pp. 505–509.
37. Zhou, X.; Han, D.; Lo, D. Assessing generalizability of codebert. In Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), Luxembourg, 27 September–1 October 2021; pp. 425–436.
38. Wang, Y.; Li, H. Code completion by modeling flattened abstract syntax trees as graphs. In Proceedings of the AAAI Conference on Artificial Intelligence, Vancouver, BC, Canada, 2–9 February 2021; Volume 35, pp. 14015–14023.
39. Ben-Nun, T.; Jakobovits, A.S.; Hoefler, T. Neural code comprehension: A learnable representation of code semantics. *arXiv* **2018**, arXiv:1806.07336.
40. Lattner, C.; Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In Proceedings of the International Symposium on Code Generation and Optimization, Palo Alto, CA, USA, 20–24 March 2004.
41. Wang, Z.; Yu, L.; Wang, S.; Liu, P. Spotting Silent Buffer Overflows in Execution Trace through Graph Neural Network Assisted Data Flow Analysis. *arXiv* **2021**, arXiv:2102.10452.
42. Schlichtkrull, M.; Kipf, T.N.; Bloem, P.; Van Den Berg, R.; Titov, I.; Welling, M. Modeling relational data with graph convolutional networks. In Proceedings of the Semantic Web: 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, 3–7 June 2018; pp. 593–607.
43. Zhang, K.; Wang, W.; Zhang, H.; Li, G.; Jin, Z. Learning to represent programs with heterogeneous graphs. In Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, Pittsburgh, PA, USA, 16–17 May 2022; pp. 378–389.
44. Yuan, D.; Fang, S.; Zhang, T.; Xu, Z.; Luo, X. Java code clone detection by exploiting semantic and syntax information from intermediate code-based graph. *IEEE Trans. Reliab.* **2022**, *72*, 511–526. [CrossRef]

45. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems 30*; NEURips: San Diego, CA, USA, 2017.
46. Schroff, F.; Kalenichenko, D.; Philbin, J. Facenet: A unified embedding for face recognition and clustering. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Boston, MA, USA, 7–12 June 2015; pp. 815–823.
47. Ge, W. Deep metric learning with hierarchical triplet loss. In Proceedings of the European Conference on Computer Vision (ECCV), Munich, Germany, 8–14 September 2018; pp. 269–285.
48. Yu, J.; Zhu, C.; Zhang, J.; Huang, Q.; Tao, D. Spatial pyramid-enhanced NetVLAD with weighted triplet loss for place recognition. *IEEE Trans. Neural Netw. Learn. Syst.* **2019**, *31*, 661–674. [CrossRef] [PubMed]
49. Zhao, X.; Qi, H.; Luo, R.; Davis, L. A weakly supervised adaptive triplet loss for deep metric learning. In Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops, Seoul, Republic of Korea, 27–28 October 2019.
50. Seo, S.; Kim, D.; Ahn, Y.; Lee, K.-H. Active learning on pre-trained language model with task-independent triplet loss. *Proc. AAAI Conf. Artif. Intell.* **2022**, *36*, 11276–11284. [CrossRef]
51. Chen, W.; Chen, X.; Zhang, J.; Huang, K. Beyond triplet loss: A deep quadruplet network for person re-identification. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Honolulu, HI, USA, 21–26 July 2017; pp. 403–412.
52. Qian, Q.; Shang, L.; Sun, B.; Hu, J.; Li, H.; Jin, R. Softtriple loss: Deep metric learning without triplet sampling. In Proceedings of the IEEE/CVF International Conference on Computer Vision, Seoul, Republic of Korea, 27 October–2 November 2019; pp. 6450–6458.
53. Dong, X.; Shen, J. Triplet loss in siamese network for object tracking. In Proceedings of the European Conference on Computer Vision (ECCV), Munich, Germany, 8–14 September 2018; pp. 459–474.