C4: Contrastive Cross-Language Code Clone Detection

Chenning Tao*
Zhejiang University
Hangzhou, China
tcn@zju.edu.cn

Qi Zhan*
Zhejiang University
Hangzhou, China
qizhan@zju.edu.cn

Xing Hu†
School of Software Technology, Zhejiang University
Ningbo, China
xinghu@zju.edu.cn

Xin Xia
Software Engineering Application Technology Lab,
Huawei
China
xin.xia@acm.org

ABSTRACT

During software development, developers introduce code clones by reusing existing code to improve programming productivity. Considering the detrimental effects on software maintenance and evolution, many techniques are proposed to detect code clones. Existing approaches are mainly used to detect clones written in the same programming language. However, it is common to develop programs with the same functionality but in different programming languages to support various platforms. In this paper, we propose a new approach named ***C4***, referring to **C**ontrastive **C**ross-language **C**ode **C**lone detection model. It can detect cross-language clones with learned representations effectively. ***C4*** exploits the pre-trained model CodeBERT to convert programs in different languages into high-dimensional vector representations. In addition, we fine tune the ***C4*** model through a constrastive learning objective that can effectively recognize clone pairs and non-clone pairs. To evaluate the effectiveness of our approach, we conduct extensive experiments on the dataset proposed by CLCDSA. Experimental results show that ***C4*** achieves scores of 0.94, 0.90, and 0.92 in terms of precision, recall and F-measure and substantially outperforms the state-of-the-art baselines.

KEYWORDS

Code Clone Detection, Neural Networks, Cross-Language, Contrastive Learning

ACM Reference Format:
Chenning Tao, Qi Zhan, Xing Hu, and Xin Xia. 2022. C4: Contrastive Cross-Language Code Clone Detection. In *30th International Conference on Program Comprehension (ICPC '22), May 16–17, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524610.3527911

*Both authors contributed equally to this research.
†Corresponding Author.

ICPC '22, May 16–17, 2022, Virtual Event, USA

ACM ISBN 978-1-4503-9298-3/22/05…$15.00
https://doi.org/10.1145/3524610.3527911

1 INTRODUCTION

In a software system, similar or identical code snippets that achieve the same functionality are called code clones. Previous research has shown that in large software systems, almost 20-30% programs are cloned [1]. The existence of code clones can be an issue considering the following reasons. Duplicated code blocks in source code improve the complexity to the program thus resulting in more difficult software maintenance [36]. The change to the code fragments will also needed to be implemented on other clone blocks as well and will lead to fault propagation if changes are done inconsistently. Code clones are also responsible for unnecessarily big code size, causing inconvenience to both the developers and maintainers. Thus, it requires to be detected and managed despite code clones enable fast software development by allowing programmers to copying and pasting the source code.

In recent years, cross language clone detection have attracted attention from researchers. Cross language clones are caused by different platforms and programming languages. Developers have to develop the same functionality programs written in different programming languages for different platforms, such as Apps in C/C# for a Windows phone, Java for an Android phone and Objective-C for an iPhone [4, 22]. Different from traditional clones, cross language clones are created intentionally by developers and are often unavoidable and cannot be removed [5, 22]. Detecting cross language clone automatically is helpful for developers to manage cross-language software systems in an easy, time-effective and cost-effective way.

According to existing studies [2], it is a common taxonomy to group code clones into four types: Type-I, Type-II, Type-III, and Type-IV. The first three types fit in the category of syntactic clone while the fourth fits in the category of semantic clone. Thus, cross language clones are Type-IV clones which have the same functionality but are quite different in implementations.

Many traditional approaches such as CCFinder [14] , Deckard [13], and SourcerCC [27] focus on detecting and analyzing Type-I to Type-III clones but limited in Type-IV clones. Recently, deep learning approaches are proposed to detect Type-IV clones. Approaches like CCLearner [17], ASTNN [45], and FA-AST [39] combined static analysis methods with deep learning achieve some success in detecting Type-IV clones.

However, only a few models are focused on cross-language clone detection. The most relevant studies are LICCA [35], CLCMiner [5],

and CLCDSA [22]. LICCA needs the SSQSA [25] platform to generate common intermediate representations for different languages and needs the code to be of equal length, which seriously limits its application in the real scenario. CLCMiner does not require those but it can only work with code with revision history, thus also having serious limitations. CLCDSA does not require the intermediate representation or revision history, but it requires an Abstract Syntax Tree (AST) to extract features manually and its performance is not very well. The limited application range and low precision and recall make it hard to be applied in real-world scenarios.

Compared to a single language clone detect, cross-language clone detection is much more difficult. To detect cross-language code clones effectively, our task is challenging since:

- **Programs are heterogeneous**. Basic grammar and APIs can be very different across different programming languages. It is hard for static analysis tools to illustrate different internal mechanisms and characteristics in programs. Deep learning based approaches are effective in mapping code snippets into high-dimensional vector space. However, many existing deep learning approaches are proposed to detect single language code clones [40, 41, 45].
- **Distinguish clone/non-clone pairs.** Distinguishing clone or non-clone pairs in high-dimensional vector space is difficult. Existing studies usually measure the vector distances (e.g., Cosine Similarity) to detect clone pairs. However, two code fragments having identical functionality but different in lexical and syntax may not be recognized as similar by the existing models. Likewise, these models cannot distinguish between two code fragments that differ in functionalities but share a close syntactic resemblance.

To address the above issues, we propose a new approach named ***C4*** that exploits the pre-trained model CodeBERT [9] with a contrastive learning objective. ❶ First, CodeBERT is a pre-trained model on six different programming languages and achieves success in programming language processing. It can map cross-language programs into the same high-dimensional vector space effectively. Therefore, we exploit the CodeBERT to get program representations. ❷ Second, we exploit the contrastive learning to make functionally equivalent code closer and distinct code further. Existing contrastive learning works [3, 32, 43] usually employ the following framework: pull together an anchor and a "positive" sample in the embedding space, and push apart the anchor from many "negative" samples. Since no labels are available, a positive pair often consists of data augmentations of the sample, and negative pairs are formed by the anchor and randomly chosen samples from the minibatch [15]. However, this framework is not ideal for our task since the randomly selected negative sample may actually be a positive clone pair. It will mislead the model to recognize clone pairs as non-clone pairs. To address this problem, we use contrastive learning in a supervised way and manually select negative samples from the batch.

To evaluate the effectiveness of our proposed approach ***C4***, we conduct experiments on the CLCDSA dataset proposed by Nafi et al. [22]. Experimental results show that our approach outperforms the state-of-the-art approaches significantly. Compared to the state-of-the-art cross-language clone detection models, our approach improves from 0.65 to 0.92 which improves about 41.5% in terms of the F1 score. In addition, compared with CodeBERT without contrastive learning, our approach improves 13.3% and 3.4% in terms of the Precision and F1 score. We also show program representations learned from ***C4***, the distance between clone pairs is closer and non-clone pairs are further. It further proves the effectiveness of ***C4*** on the cross-language clone detection.

In summary, the main contributions of this paper are summarized as follows:

- We propose a new cross-language clone detection method named ***C4***. It exploits the CodeBERT with contrastive learning to learn code representations effectively.
- We evaluate the effectiveness of ***C4*** on the CLCDSA dataset. The experimental results show that ***C4*** outperforms the state-of-the-art clone detection approaches.

The remainder of the paper is organized as follows: Section 2 presents the background information of our paper. Section 3 discusses the architectural design of the model. Section 4 and Section 5 briefly discusses the evaluation and validation steps and results of ***C4***. Section 6 describes the probable threats to validity and how those were addressed. Section 7 describes closely related work and Section 8 concludes the paper.

2 BACKGROUND

2.1 Code Clone

In software development, similar fragments that share the same functionality are called clones. Software clones are detrimental most time since they increase maintenance costs and make the system difficult to evolve, which lead to the development of clone detection.

Roy et al. [26] summarized four type of code clones:

- **Type-I Clone**: Identical code fragments except for variations in white-spaces and comments.
- **Type-II Clone**: Structurally/syntactically identical fragments except for variations in the names of identifiers, literals, types, layout and comments.
- **Type-III Clone**: Code fragments that exhibit similarity as of Type-II clones and also allow further differences such as additions, deletions or modifications of statements are known as Type-III clones.
- **Type-IV Clone**: Code fragments that exhibit identical functional behaviour but can be implemented through very different syntactic structures.

In this paper, we detect programs with the same functionality but written in different programming languages. Different from single language code clones, cross-language clones are created intentionally because of multi-platforms development. Thus, cross-language clone pairs referred in this paper are Type-IV clone pairs. As the cross-language clones are unavoidable, we need to detect and manage them for developers. Figure 1 shows an example of cross-language clone pair. We can find that the two programs aim to "count vowels permutations" but written in C++ and Python, respectively. The C++ language code and Python language code

Solutions for "Count Vowels Permutation"

C++ Solution:

```cpp
int countVowelPermutation(int n) {
    long long mod = 1e9 + 7;
    vector<long long> dp(5, 1);
    vector<long long> ndp(5);
    for (int i = 2; i <= n; ++i) {
        ndp[0] = (dp[1] + dp[2] + dp[4]) % mod;
        ndp[1] = (dp[0] + dp[2]) % mod;
        ndp[2] = (dp[1] + dp[3]) % mod;
        ndp[3] = dp[2];
        ndp[4] = (dp[2] + dp[3]) % mod;
        dp = ndp;
    }
    return accumulate(dp.begin(), dp.end(), 0LL) % mod;
}
```

Python Solution:

```python
def countVowelPermutation(self, n: int) -> int:
    factor = np.mat([(0, 1, 0, 0, 0),
                     (1, 0, 1, 0, 0),
                     (1, 1, 0, 1, 1),
                     (0, 0, 1, 0, 1),
                     (1, 0, 0, 0, 0)], np.dtype('O'))
    res = np.mat([(1, 1, 1, 1, 1)], np.dtype('O'))
    n -= 1
    while n > 0:
        if n % 2 == 1:
            res = res * factor % 1000000007
        factor = factor * factor % 1000000007
        n = n // 2
    return res.sum() % 1000000007
```

Figure 1: Cross-language clone example

represent same functionality whereas very different in syntax. Compared with clone detection in the same programming languages, cross-languages clone detection has more challenges:

- Basic grammar and characteristic are very different in different programming languages, it is difficult to measure code similarity through syntax analysis.
- There are no general intermediate representation for different programming languages, we need to propose a new representation for all languages in order to measure code similarity.

Therefore, we propose ***C4*** to address these challenges. ***C4*** uses the CodeBERT as pre-trained model to convert different languages code fragments into the same vector representations so that we can unify process code snippets in different programming languages based on the same intermediate representations .

2.2 Pre-trained Model

Pre-trained models have shown to be effective for NLP tasks and achieved the state-of-the-art results among many tasks [8, 19, 24, 38]. Recently, pre-trained models are also exploited in programming language processing tasks and achieve great results. For example, Liu et al. [18] apply the BERT model to the code completion task. Gao et al. [11] exploit the BERT to identify whether a TODO comment is obsolete and should be removed. In the following subsections, we will introduce the background information of Transformer, BERT, and CodeBERT.

2.2.1 Transformer. Transformer [34] is a model originally designed for machine translation. Later, it is proven useful in other areas of natural language processing and also boosts the use of pre-trained models, such as BERT.

The overall architecture consists of an encoder and a decoder. The Encoder consists of a self-attention layer and a feed-forward layer. The attention layer assists the encoder in looking at additional words in the input text while encoding a certain word. The output of this is then fed into a feed-forward layer. Both of these layers are included in the decoder, but between them is an encoder-decoder attention layer that assists the decoder in focusing on key sections of the input sequence.

2.2.2 BERT. Bidirectional Encoder Representations from Transformer (BERT) [8] applies bidirectional training of Transformer into language modeling. It has a deeper sense of language context and is trained on large-scale plain texts with self-supervised learning objectives. It can be reused for multiple related tasks through fine-tuning and achieving state-of-the-art performance.

BERT was pre-trained on two unsupervised tasks: Masked LM and Next Sentence Prediction (NSP). Masked LM masked 15% of words in each sentence randomly and let the model predict the masked word. NSP tries to improve the model's understanding of sentence relationship by letting the model predict whether these two languages are continuous.

The original paper described two model architecture: $\text{BERT}_{\text{base}}$ and $\text{BERT}_{\text{large}}$. They all consist of stacks of Transformer encoders. The base model has 12 layers, the embedding size is 768 and the number of attention head is 12. The large model doubled layer number and changed embedding size to 1024 and attention head to 16.

2.2.3 CodeBERT. CodeBERT [9] is a bimodal pre-trained model for programming language (PL) and natural language (NL). It is based on RoBERTa [20] and utilizes both bimodal instances of NL-PL pairs and a large amount of PL pairs by using standard masked language modeling [8] and replaced token detection [7].

It is trained on 2.1M bimodal datapoints and 6.4M unimodal code snippets across six programming languages, including Ruby, JavaScript, Go, Python, Java, and PHP. In the pre-training and fine tuning phase, the model takes the input format as $[CLS], \omega_1, \omega_2, ...\omega_n, [SEP]$ and outputs a contextual vector representation of each token and the representation of $[CLS]$, which could be deemed as sentence representation.

The experimental results show that CodeBERT achieves the state-of-the-art results on various software engineering tasks, for example, code search and code documentation generation. Considering that CodeBERT is pre-trained on the huge code corpus and learns universal representations for programs in different languages, we adopt CodeBERT to map programs in our corpus into high-dimensional vectors. We exploit it as an initialization of our approach ***C4*** and fine-tune it with the contrasive learning.

2.3 Contrastive Learning

Contrastive learning is first introduced by Hadsell et al. [12] for classification. Unlike loss functions that sum over samples, its loss function runs over pairs of examples. The original contrastive learning is defined as follows. Assume X_1, X_2 is a pair of input vectors. Let Y be the label assigned to this pair. $Y = 1$ means X_1 and X_2 are similar, $Y = 0$ means they are different. The distance between two vector is defined by $D_w(X_1, X_2)$. Then the loss function can be written as $L(W, (Y, X_1, X_2)^i) = (1 - Y)L_S(D^i_W) + YL_D(D^i_W)$. L_S is the partial loss function for a pair of similar pairs, L_D is the partial loss function for a pair of different pairs.

Loss function is one of the most important part in contrastive learning. Since Hadsell et al. [12], a variety of loss functions have been proposed for different situations such as Triplet Loss [28] and N-pair Loss [29]. They improve the loss function so that it can process multiple negative examples at the same time.

The overall framework of contrastive learning can be described as follows:

- An encoder $f(\cdot)$ that can convert a given sample into vector representation.
- A batch of examples $\{x_1, ..., x_n\}$ where n represents the batch size.
- A similarity function sim(x, y) which can represent the similarity of two vector, for example, many works [5, 22] use cosine as the similarity function.

To train a model with contrastive loss objective, for each example x_i in the batch, we need to use the encoder to get the vector representation of the example $v_i = f(x_i)$. Next, for each v_i, we have to construct the positive examples and negative examples for this particular sample. After the selection, we will use the similarity function sim(x, y) to get the distances between positive/negative examples and put it into the loss function.

Many approaches achieve the state-of-the-art results in their fields using contrastive learning, such as text generation [44], sentence embedding [42], and pre-trained Language Models [23].

3 PROPOSED APPROACH

In this section, we present the overall design and implementation details of our proposed model.

3.1 Overall Framework

The overall framework of ***C4*** is illustrated in Figure 2. We process all programs into batches where each batch consists of code snippets that belong to the same task but used different programming languages. For each code snippet, we exploit the CodeBERT to get its representation. Then, we use contrastive learning for every code snippet in the batch. It can make clone pairs much closer and non-clone pairs further in the high-dimensional vector space. Our approach mainly consists of three parts, including a) Data Process; b) Get Code Representation; c) Contrastive Learning. We will introduce these three parts in the following subsections.

3.2 Data Process

Data Processing is the first step to build our approach. We need to construct clone pairs and non-clone pairs. The details can be found in Section 4.2.

3.3 Code Representation

We use the CodeBERT as our pre-trained model to get code representations. In other words, we exploit the CodeBERT to encode programs into vector representations, i.e., $c \xrightarrow{f} v$ where f means CodeBERT, c means the source code, and v means the vector embedding for the corresponding source code.

We fine tune the CodeBERT model to get a better vector embedding for the code clone downstream task [8]. The code snippets is first split into a list of words, then it is put in the CodeBERT's tokenizer. The output of the tokenizer is then concatenated with CLS token in the front and SEP token in the back. The format of this output is $[CLS], \omega_1, \omega_2, ..\omega_n, [SEP]$. Then, we use the tokenizer's function to convert them into ids before put it into the model. Note that we limit each code snippet length to be less than 512 since CodeBERT cannot process input longer than that. During the fine tuning, the output of CodeBERT includes the contextual vector representation of each token and the representation of $[CLS]$ (sentence representation). The parameters of CodeBERT are learned to minimize the contrastive loss.

3.4 Contrastive Learning

The loss function we used is shown in Equation 1. Different from existing works, we use a combination of N-pair loss and Soft-Nearest Neighbor loss [10].

$$L_N = -\log \frac{\exp\left(\text{sim}(\text{x}, \text{x}^+)/\tau\right)}{\exp\left(\text{sim}(\text{x}, \text{x}^+)/\tau\right) + \sum_{j=1}^{N-1} \exp\left(\text{sim}(\text{x}, \text{x}_\text{j}^-)/\tau\right)} \quad (1)$$

x is the anchor sample, x^+ is the positive sample (i.e., x and x^+ is a clone pair), and x^- is the negative sample (i.e., x and x^- is a non-clone pair). τ is a hyperparameter that we tune to scale the penalties on negative samples.

As shown in Figure 2, ***C4*** exploits the contrastive learning to learn effective representation by pulling clones together and pushing apart non-clones. Assume we have a similarity function $\text{sim}(\text{v}_\text{i}, \text{v}_\text{j})$ and its codomain is $[0, 1]$. The details of pulling and pushing are shown as follows:

- **Pull** Pulling pairs together means getting the result of similarity function closer to 1.
- **Push** Pushing pairs apart means getting the result of similarity function closer to 0.

For the similarity evaluation we use

$$\text{sim}(\text{u}, \text{v}) = \frac{\boldsymbol{u} \cdot \boldsymbol{v}}{\|\boldsymbol{u}\|\|\boldsymbol{v}\|} \quad (2)$$

where $\text{sim}(\text{u}, \text{v}) \in [-1, 1]$.

For term $\sum_{j=1}^{N-1} \exp\left(\text{sim}(\text{x}, \text{x}_\text{j}^-)/\tau\right)$, if $\text{sim}(\text{x}, \text{x}^-) < 0$, the τ will decrease the sensitivity of loss function. Therefore, we transform the codomain of sim(u, v) from $[-1, 1]$ to $[0, 1]$ using a linear transformation.

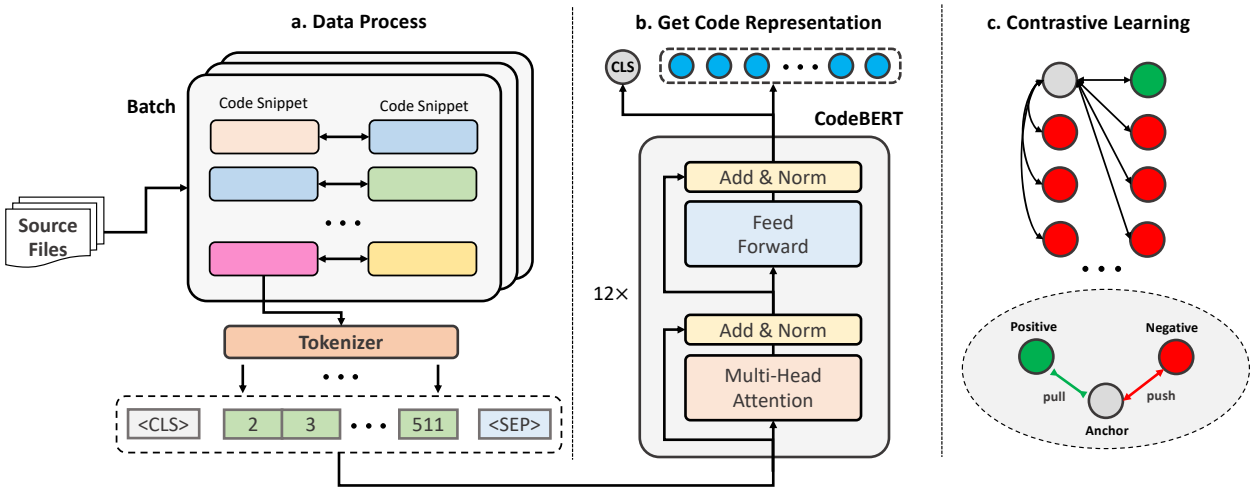


Figure 2: Overview of our approach

A detailed description of how to construct negative samples and compute contrastive loss is illustrated in Algorithm 1.

Algorithm 1: Contrastive loss computation

Input: A batch of clone pairs, B;
Output: Contrastive loss, L_N;

foreach *pair* P_i *in* B **do**
  $x, x^+ \leftarrow P_i.code$;
  $S_{pos} \leftarrow \text{sim}(x, x^+)$;
  foreach *pair* $P_j (i \neq j)$ *in* B **do**
    if $P_i.task \neq P_j.task$ **then**
      $x^-_{j_1}, x^-_{j_2} \leftarrow P_j.code$;
      $S_{nag_{j_1}} \leftarrow \text{sim}(x, x^-_{j_1})$;
      $S_{nag_{j_2}} \leftarrow \text{sim}(x, x^-_{j_2})$;
    end
  end
  $L_{N_i} = -\log(\frac{S_{pos}}{S_{pos}+\sum S_{nag}})$;
end
$L_N = \sum_{i=1}^{len(B)} L_{N_i}$

For each batch we process, assume the batch size is B. It means that we have B clone pairs in this batch. We will then loop over the batch. For each pair, we take the first one as an anchor example (x), the second one as a positive example (x^+). Ideally, the rest clone pairs will all be negative examples (x^-). Thus, the number of negative examples will be $2(B-1)$, i.e., all programs ($2B$) excluding the corresponding clone pair. However, the rest of programs may come from the same problem. In other words, these programs are clones of the anchor example. Therefore, we should filter out those code snippets. After constructing positive example and negative examples, we compute their cosine similarity with anchor example. Then, ***C4*** exploits the derived loss function (i.e., Equation 1) to get the loss and train the model.

In the fine tuning phase, we applied the same hyperparameter used in the CodeBERT paper. Then, we fine tune ***C4*** optimized by our proposed contrastive loss.

4 EXPERIMENTAL SETUP

We evaluate the effectiveness of our approach following three research questions:

- **RQ1:** How effective is our approach at detecting cross-language code clones compared to the state-of-the-art baselines?
- **RQ2:** How effective is our approach when detecting specific language pairs?
- **RQ3:** How effective is each component in our approach?

We address RQ1 by comparing our approach with CLCMiner [5] and CLCDSA [22]. CLCMiner is a traditional text-based tool for detecting code clones while CLCDSA is neural networks based approach. We then address RQ2 by analyzing our results for different programming language pairs. Finally, we address RQ3 by comparing results with or without fine-tuning and with or without contrastive learning.

4.1 Baselines

To evaluate the effectiveness of our approach, we compare our model with the following baselines:

4.1.1 CLCMiner [6]. CLCMiner is a state-of-art text-based code clone detection tool. It uses bag-of-tokens and revision histories to detect clones. As CLCMiner does not require training, we use it to detect clones in the test set.

4.1.2 CLCDSA [22]. CLCDSA is a cross-language detection tool that uses syntactical features and API documentation. It uses a deep neural network based feature vector learning model to learn the features and detect cross-language clones.

Table 1: Statistics for specific language code blocks

Language	Average Lines	Number of Blocks	Tokens
C++	66	32,281	616
C#	111	16,359	939
Java	106	18,836	734
Python	26	18,331	212

4.2 Datasets

CLCDSA [22] provides three dataset for cross-language clone detection, including Google CodeJam[1] data, AtCoder[2] data, and CoderByte data. As CoderByte data collected by CLCDSA is not available, we conduct experiments on the other two datasets, i.e., Google CodeJam and AtCoder.

CodeJam is a Google's programming competition and AtCoder is a programming competition website in Japan. There are different programming language solutions for one problem so that we can take them as cross-language clone pairs. There are 86,807 code blocks and 1,380 problems in our dataset. For one problem, there are about 63 solutions on average. We split these problems 8:1:1 for train, valid, test sets.

For ***C4*** with contrastive learning, we do not need the non-clone pairs in the training set since it can search different problems' code fragments in a batch as negative samples. We construct a clone pair by randomly selecting one program and the other one program in the same problem. In total, we get 86,695 cross-language clone pairs, in which 69,278 pairs in the training set, 8,742 pairs in the validation set, and 8,675 pairs in the test set.

For CLCDSA and CLCMiner, we follow the experiments setup of CLCDSA [22], randomly select from different problems to create non-clone pairs for one clone pair. As a result, we have 138,804 pairs in the training set, 17,332 pairs in the validation set, and 17,349 pairs in the test set where true clone pairs count for 50%.

The statistics of different language code blocks is given in Table 1. The number of lines and tokens in C# is the highest while the number in Python is the lowest since the amount of code in Python is usually smaller to achieve the same functionality than other programming languages in the dataset. The number of code blocks in C++ is the highest, nearly as twice as the number of other language blocks.

4.3 Metrics

In clone detection, we usually use Precision (P), Recall (R), and F1-Score (F1) to measure the effectiveness of different approaches.

Precision stands for the accuracy of detection, is the ratio of the number of true code clones detected to the number of all retrieved code pairs. We define R_R as the number of true code clones retrieved, I_R as the number of false code clones retrieved, then Precision is:

$$P = \frac{R_R}{R_R + I_R} \tag{3}$$

Recall is the ratio of the number of actual code clones detected to the number of actual code clones. We define R_R as the number of true code clones retrieved, R_N as the number of all true code clones in dataset, then Recall is:

$$R = \frac{R_R}{R_R + R_N} \tag{4}$$

F1-Score stands for the overall performance of detection, is the harmonic mean of precision and recall. If P is the precision and R is the recall, then F1 is:

$$F1 = \frac{2PR}{P + R} \tag{5}$$

Table 2: Effectiveness of our approach vs. baselines

Approach	Precision	Recall	F1
CLCMiner	0.36	0.57	0.44
CLCDSA	0.50	**0.92**	0.65
C4	**0.94**	0.90	**0.92**

4.4 Experimental Setting

We choose the hyperparameters for our approach as follows: we train the model for five epochs and use the Adam optimizer with batch size of 36. The threshold for predicting the test set is the same as the best threshold of the validation set for all RQs while the threshold for CLCMiner is 0.5. τ for contrastive learning is set to 12. Lastly, we use PyTorch to implement our models. For the training of this model, we use three RTX3090. For each epoch, it requires about one hour.

4.5 Replication Package

Our code and dataset we used is publicly available[3].

5 EXPERIMENTAL RESULTS

5.1 RQ1: *C4* vs. Baselines

We compare our approach with two state-of-the-art approaches, including CLCMiner [5] and CLCDSA [22]. We use the same experimental settings to evaluate our approach against the other approaches in terms of precision, recall, and F1-score.

As shown in Table 2, our approach outperforms all state-of-the-art baselines in terms of precision and F1-score. CLCDSA performs best considering the recall. We find that CLCDSA prefers to predict programs as clone pairs.

CLCMiner only considers a bag of tokens and neglects all other information when detects code clones. However, cross-language code clones are different lexically and syntactically. CLCMiner fails to detect them through similar tokens. In addition, different reserved tokens in different languages also increase the difficulty for CLCMiner to detect cross-language clones.

Considering CLCDSA that is designed for cross-language detection, it performs better than CLCMiner. It analyzes the similarity of nine syntactic source code features that have almost similar values if two source code fragments from two different programming languages have similar functionality.

[1] http://code.google.com/codejam
[2] https://atcoder.jp/
[3] https://github.com/Chenning-Tao/C4

Our approach outperforms both CLCMiner and CLCDSA in terms of precision, recall, and F1-Score, which achieve 0.94, 0.90, and 0.92 respectively.

5.2 RQ2: Specific language pairs

Table 3 shows results of approaches on different language combination. Comparing results of different programming language pairs with CLCMiner and CLCDSA, we have the following findings:

- ***Finding 1***. All evaluation metrics (Precision, Recall, F1-Score) in our approach is greatly higher than measures with CLCMiner, which achieve an increase of 0.35 in Precision, 0.56 in Recall, and 0.48 in F1-Score on average.
- ***Finding 2***. Compared with CLCDSA, the only metric our approach worse than CLCDSA is Recall for C# & Java pair. The Recall scores of our approach and CLCDSA are similar, but F1-Score of our approach is much higher than CLCDSA since Precision of our approach is higher significantly.
- ***Finding 3***. Variances of Precision, Recall, and F1-Score is 0.03304, 0.0367, and 0.03236 respectively for different language combinations results in our approach, which indicate that the deviation of results in our approach is small.

5.3 RQ3: Ablation Study

In this paper, we exploit the CodeBERT to lean the representations of source code. Then, we use the contrastive learning technique to fine tune the model. We want to investigate the impacts of these variants on the performance of our approach. To illustrate the importance of each component, we compare our approach with two of its incomplete variants:

- CodeBERT without fine-tuning. We directly use the pre-trained CodeBERT to detect cross-language clone pairs without fine-tuning and contrastive learning.
- CodeBERT with fine-tuning. We fine tune the CodeBERT without contrastive learning. During the fine-tuning process, we use Mean Square Error (MSE) as the loss function that is used in the original CodeBERT.

Table 4 shows the results of these two variants and ***C4***. Comparing results of CodeBERT with and without fine-tuning, we find that our approach benefits from fine-tuning. F1-Score is only 0.67 if we directly use the representations in CodeBERT to detect clone pairs. We can find that it regards all pairs as clone pairs. We achieve an increase of 0.22 in F1-Score when we fine-tune with the train set. Similarly, we also achieve an increase in F1-Score when we fine-tune with contrastive learning. Compared to fine-tuning CodeBERT, contrastive learning has great advantage in improving the Precision.

To evaluate the effectiveness of the contrastive learning, we also give the detailed results (shown in Table 5) of CodeBERT with MSE loss fine-tuning. Compared to results of ***C4*** shown in Table 3, Precision and F1-Score with contrastive learning is higher than models without contrastive learning. Our approaches achieve an average increase of 0.10 in precision and 0.03 in F1-Score, which indicates that our approach is effective for all different language combinations.

In addition, we also visualize the code vectors to help understand and explain why the vectors produced by ***C4*** are better than the vectors produced by others. We randomly select four problems and get embedding of code snippets that belongs to these problems, then we use t-SNE [33] to reduce dimensionality from 768 to 2 for visualization. As shown in Figure 3, points in the same color represent programs from the same problems. Points with the same color in Figure 3(a) are scattered across the graph, which means the embedding of code is irregular without fine-tuning. Most points with the same color in Figure 3(b) converge together but their boundary is not very clear and some of them tangled together, which means it can detect clone pairs but hard to distinguish several non-clone pairs. It explains the relatively high recall (0.95) and low precision (0.83) in RQ3. Points with same color in Figure 3(c) stay closer than Figure 3(b) while different color points stay far apart. Our results show that contrastive learning can keep dissimilar pairs away while pulling similar pairs together, which increases Precision and F1-Score greatly in detecting semantic code clones.

6 DISCUSSION

6.1 Can *C4* achieve good performance on a small dataset?

Considering the long training time for ***C4*** and CodeBERT, we further conduct an experiment on a smaller dataset that is more energy efficient. We randomly select 30% pairs from the dataset and compare ***C4*** with CodeBERT. The smaller dataset includes 20,773 in the training set, 2,619 in the validation set, and 2,572 in the test set. For fine-tuned CodeBERT, we need to construct negative samples in its dataset. Therefore, the fine-tuned CodeBERT has twice the amount of data includes 41,646 in the training set, 5,200 in the validation set, and 5,105 in the test set. Table 6 describe the results on the small dataset setting. Compared to Table 4, we find that $C4_{SMALL}$ has the same performance as ***C4***. However, the performance of the fine-tuned CodeBERT has an obvious decline. The small data setting further demonstrates the effectiveness of our approach ***C4***.

6.2 Proportion of Clone Pairs in the Test Set

Since the CLCDSA dataset only consists of clone pairs, we need to construct negative samples by selecting samples from different tasks.

Therefore, the amount of non-clone pairs may effect the performance of different approaches. In the experimental setting, we follow the CLCDSA and construct same number of non-clone pairs as the positive pairs.

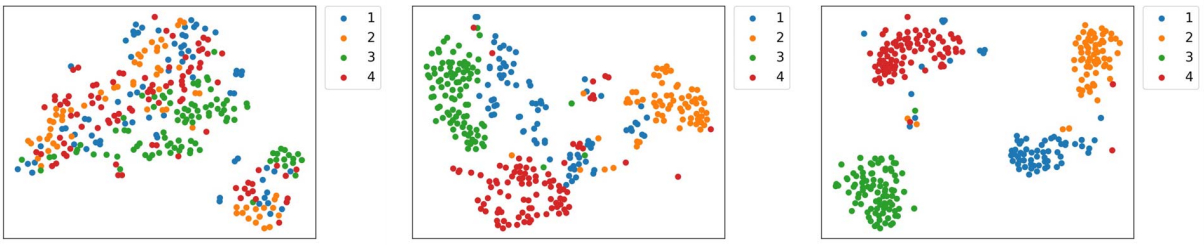
But in BigCloneBench [30], one of the most famous benchmarks for code clone, we find that though in the train dataset, the number of clone pairs is almost identical to the non-clone pairs, in the valid and test dataset, the number of non-clone pairs versus clone pairs is almost $1:6$. So we want to see how well our model is when the data distribution is different. As we need to construct much more non-clone pairs in this discussion, we follow the small data setting above to save training time.

We retrain the model using the same amounts of clone pairs but different amounts of non-clone pairs. Besides the $1:1$ (clone pair : non-clone pair) ratio we use in Section 4.1, we use $1:2, 1:3$ and $1:6$ for training. And for each model we trained, we test them on test dataset with $1:1, 1:2, 1:3, 1:6$ ratio. The detailed F1 results are shown in Table 7.

Table 3: Results on different language combinations

Language Combination	*CLCMiner*			*CLCDSA*			*C4*		
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
C++ & C#	0.57	0.36	0.44	-	-	-	0.91	0.88	0.89
C++ & Java	0.58	0.37	0.45	-	-	-	0.93	0.90	0.91
C++ & Python	0.59	0.36	0.45	-	-	-	0.94	0.92	0.93
C# & Java	0.59	0.34	0.43	0.67 (0.47*)	0.96 (0.91*)	0.79 (0.62*)	0.95	0.93	0.94
C# & Python	0.56	0.36	0.44	0.63 (0.54*)	0.89 (0.95*)	0.73 (0.69*)	0.92	0.95	0.94
Java & Python	0.59	0.36	0.45	0.58 (0.46*)	0.90 (0.85*)	0.705 (0.60*)	0.95	0.93	0.94

* We replicated the CLCDSA according to the replication package they provided in the GitHub. We re-train their models and show the results from our training (in the brackets). Results outside the brackets are taken from CLCDSA paper.



(a) w/o fine-tune (b) w/ fine-tune (c) ***C4***: w/ fine-tune and contrastive learning

Figure 3: Visualization of the vector representations of the code snippets

Table 4: Effectiveness of each incomplete variant of our approach

Approach	Precision	Recall	F1
without fine-tuning	0.50	**1.00**	0.67
with fine-tuning	0.83	0.95	0.89
C4	**0.94**	0.90	**0.92**

Table 5: Results on different language combinations of *C4* without contrastive learning

Language Combination	Precision	Recall	F1
C++ & C#	0.83	0.95	0.89
C++ & Java	0.84	0.94	0.88
C++ & Python	0.82	0.96	0.88
C# & Java	0.83	0.97	0.90
C# & Python	0.84	0.98	0.90
Java & Python	0.84	0.98	0.90

Table 6: Effectiveness of *C4* and fine-tuned CodeBERT on a small dataset setting

Approach	Precision	Recall	F1
fine-tuned CodeBERT	0.82	0.92	0.87
$C4_{SMALL}$	**0.94**	0.90	**0.92**

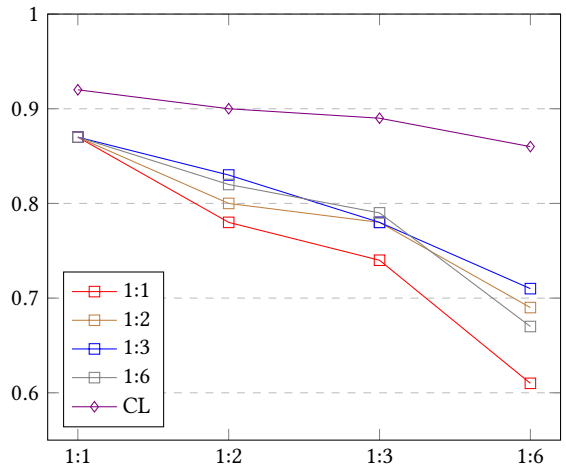


Figure 4: F1 results when using different proportions of clone pairs to train and test.

From Table 7 we can see, when only using CodeBERT with fine-tuning, simply adding more non-clone pairs will not help the model to perform better. The F1 result when using $1:2, 1:3, 1:6$ for training is almost the same while the number of negative samples almost tripled. This indicates the traditional method cannot make full use of a large number of non-clone pairs while the methods with contrastive learning can benefit a lot from a large number of non-clone pairs.

Table 7: F1 results of different train and test dataset

Train \ Test	1:1	1:2	1:3	1:6
1:1	0.87	0.78	0.74	0.61
1:2	0.87	0.80	0.78	0.69
1:3	0.87	0.83	0.78	0.71
1:6	0.87	0.82	0.79	0.67
CL	**0.92**	**0.90**	**0.89**	**0.86**

Table 8: CodeBERT baseline when training with same number of clone pairs and non-clone pairs

Test	Precision	Recall
1:1	**0.82**	0.92
1:2	0.70	0.91
1:3	0.64	0.91
1:6	0.45	**0.92**

Figure 4 depicted the result in the diagram. X-axis represents the test dataset ratio, each line represent different training method. We can see when using CodeBERT with fine tuning, when the ratio changed from 1 : 1 to 1 : 2, the F1 score dropped significantly from 0.04 up to 0.09. And when the ratio changed to 1 : 6, the result for our baseline method dropped up to 0.26 compared to 1 : 1 while CodeBERT fine tuning with contrastive learning only dropped 0.06.

From Table 8 we can see as the number of non-clone pairs in the dataset increases, recall barely changes while precision deteriorates rapidly. It seems to be the result of the model's tendency to deem non-clone pairs as clone pairs. When the ratio is 1 : 6, the precision drops to 0.45, which is almost like the model is just guessing. We can safely assume that the method with contrastive learning has learnt a better code representation thus can handle different data distribution better.

6.3 Investigating why *C4* fail

We are also curious about why our approach fail to detect clones. We find that ***C4*** may mistake non-clone pairs as clones. Figure 5 shows an example of a non-clone pair that is mistaken as a clone pair. We can find that the two code fragments have same variable name *time* and similar variable name *time_to_goal*, *bestTime*. ***C4*** is confused when most variable names are similar. In addition, we can observe that their interprocedural control flow is similar. In the main function, they both call another method in a loop and print the answer. In the function they call, there are the same *while* loop and some assignment statements in them. However, statements in the basic block are totally different and two code fragments achieve different functionality. Our approach fails to predict them as a non-clone pairs. In the future, we plan to combine the control flow and data flow to achieve more precise detection.

6.4 Threats to Validity

We have identified the following threats to validity among our study:

6.4.1 Internal Validity. In this paper, we exploit the pre-trained model CodeBERT to convert programs written in different languages into the high-dimensional code vector space and fine-tune the model. Pre-trained knowledge from CodeBERT may introduce bias of the effectiveness of our approach. First, CodeBERT is pre-trained on Go, Python, Java, JavaScript, Php, and Ruby. In this paper, we detect clones in Java, Python, C++, and C# in which C++ and C# programs are not learned by the CodeBERT. The representations for these programs may not effective. Second, CodeBERT can only handle 512 tokens while the lengths of most code fragments are over 512 tokens, which influence ***C4***'s effectiveness.

6.4.2 Dataset Validity. The dataset used in our experiments takes solutions written in different programming languages for the same task as clone pairs. CoderByte dataset collected by CLCDSA is not available so that we use other two datasets (i.e. AtCoder dataset, Google CodeJam dataset) to train and test, which lead to the different results between paper and our experiments for CLCDSA.

7 RELATED WORK

7.1 Traditional Approaches

7.1.1 Single language. Most of the clone detection approaches using traditional methods are targeted at Type-I, Type-II, and weak Type-III clones detection. Existing studies mainly include token-based techniques and syntactic-based techniques.

CCFinder [14] is a tool that transforms the input source text into a regular form and uses a token-by-token matching algorithm to compute clone metrics. But it requires a language-dependent lexical analyzer and transformation rules. Therefore, it only supports a limited amount of programming languages.

SourcerCC [27] is a token-based clone detector that tokenizes the code blocks and creates a partial inverted index for detecting clone pairs. Its ability is limited as it can only capture lexical level information.

Deckard [13] is an AST-based tool that characterizes subtrees with numerical vectors in the Euclidean space and clusters these vectors using the Euclidean distance metric. Subtrees with vectors in one cluster are considered similar. Similar to CCFinder, it requires pre-defined rules for each language.

In addition, a large amount of works also focus on clone detection. They can be divided into five categories [37]: textual, token, syntactic, semantic, and learning. Textual based approaches compare code in the form of text and are only found to be cloned if the two code fragments are literally identical in terms of textual content. Token based apporaches like CCFinder and SourcerCC split code into tokens and matched them. Syntactic approaches use tree-based techniques (i.e. AST) or metric-based techniques to compare code similarity. Semantic approaches detect code fragments that have different structure but perform same function. This can be done using static analysis such as PDG [16]. But they are often faced by the problem of being very difficult to scale. We find that most traditional tools that detect clone pairs such as CCFinder [14] and SourcererCC [27] can be applied to different languages but each pair must be the same programming language.

7.1.2 Cross Language. A very few models can be applied to cross-language clone detection. The most representative ones are LICCA [35]

C++:

```
double cookies(void) {
    double cost, f, goal;
    double time = 0;
    scanf("%lf%lf%lf", &cost, &f, &goal);
    double cookie_rate = 2.0;
    double time_to_farm, time_to_goal, time_after_farm;
    while(true) {
        time_to_goal = goal/cookie_rate;
        time_to_farm = cost/cookie_rate;
        if(time_to_goal < time_to_farm) {
            time += time_to_goal;
            break;
        }
        time_after_farm = goal/(cookie_rate+f);
        if(time_after_farm + time_to_farm < time_to_goal) {
            cookie_rate += f;
            time += time_to_farm;
        } else {
            time += time_to_goal;
            break;
        }
    }
    return time;
}
int main(void) {
    int cases;
    scanf("%d", &cases);
    for (int i = 1; i <= cases; i++) {
        double answer = cookies();
        printf("Case #%d: ", i);
        printf("%lf\n", answer);
    }
}
```

Python:

```
memo = dict()
def canBuy(C,F,n):
    if (C,F,n) in memo:
        return memo[(C,F,n)]
    if n==1:
        return C / 2.0
    memo[(C,F,n)] = canBuy(C,F,n-1) + C/ (2.0 + F* (n-1))
    return canBuy(C,F,n)


def best(C,F,X):
    factories =0
    bestTime = X/2.0

    while True:
        factories+=1
        time = canBuy(C,F,factories)
        time += X / (factories* F + 2.)
        if time < bestTime:
            bestTime = time
        else:
            return bestTime

f = open('input.in', 'r')
data = f.read().split()

instances = int(data[0])
for x in xrange(instances):
    line = map(float, data[x+1].split(" "))
    C, F, X = line[0], line[1], line[2]
    print "Case #%d: %.7f" % (x+1, best(C,F,X))
```

Figure 5: An example of a non-clone pair that is mistaken as a clone pair by *C4*

and CLCMiner [5]. LICCA relies on SSQSA's high-level representation of code but is only able to be applied to 5 languages. Also, it requires the same source code length and code steps and flow of functionality of two code blocks. CLCMiner mines clones from revision histories thus limiting its application. Those approaches have many limitations thus are unsuitable in real world scenarios.

7.2 Deep learning Approaches

With the emergence of difficult datasets such as BigCloneBench [31] which contains Type-IV clone pairs and the advances of deep learning, the number of approaches using deep learning to detect code clone pairs has increased a lot.

7.2.1 Single language. Most approaches use static analysis methods and can be categorized into token-based, syntax-based, and semantic-based.

The token-based methods like CCLearner [17] use deep learning methods to acquire token-level information. CCLearner extracts tokens from known method-level code clones and non-clones to form a feature vector. They use this feature vector to train a classifier and then use the classifier to detect clones in a given codebase. They also extracted some syntax tree information and use them in training. But this method ignores the structure information of the program and is very sensitive to changes like code orders.

The syntax-based methods usually use AST to obtain information regarding its code structure and grammar. CDLH [41] treats this problem as a supervised-learning problem of the source code's hash features. The CDLH approach uses certain encoding rules to encode the abstract syntax tree representation of the source code. Then it uses an LSTM network to combine these representations into a binary vector. But it only utilizes the structure information and ignores other information like the node type. ASTNN [45] utilized node type in AST and split the entire AST into a sequence of small statement trees. Then use a statement encoder to capture the lexical and syntactical knowledge of the tree and turn it into a vector. A bidirectional RNN model is used to generate the vector representation of the code.

The semantic-based methods usually involve more complex static analyses such as CFG, DFG and PDG. But some analysis like PDG requires the code to be compatible and is unable to detect clones in incomplete code fragments. Wang et al. [39] build a graph representation of programs called flow-augmented abstract syntax tree (FA-AST) by adding control and data flow edges. Mehrotra et al. [21] utilize the program dependency graph to encode code fragment and use a graph-based Siamese network to detect clone pairs.

7.2.2 Cross language. Very few researchers have focused on cross-language clone detection due to its complexity and lack of cross-language clone detection datasets. The most famous work is CLCDSA [22] which provides a dataset for cross-language clones and implemented a deepNN network that can detect cross-language clones.

8 CONCLUSION

There are numerous approaches to detect code clones in the same language, but few approaches are able to detect cross-languag code

clones effectively. In our paper, we propose a new approach, ***C4***, to detect cross-language code clones. ***C4*** takes the pre-trained model CodeBERT to convert programs in different languages into representations and finetune the model using contrasitive objective so that it can effectively recognize clone pairs and non-clone pairs. Experimental results show that our approach outperforms the state-of-the-art approaches significantly in terms of precision, recall, and F1-score. In addition, ***C4*** also performs well in small dataset settings and outperforms other approaches.

9 ACKNOWLEDGMENTS

This research was supported by the National Science Foundation of China (No. 62141222 and No. U20A20173)

REFERENCES

[1] B.S. Baker. 1995. On finding duplication and near-duplication in large software systems. In *Proceedings of 2nd Working Conference on Reverse Engineering*. 86–95. https://doi.org/10.1109/WCRE.1995.514697

[2] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering* 33, 9 (2007), 577–591.

[3] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A Simple Framework for Contrastive Learning of Visual Representations. arXiv:2002.05709 [cs.LG]

[4] Xiao Cheng, Zhiming Peng, Lingxiao Jiang, Hao Zhong, Haibo Yu, and Jianjun Zhao. 2016. Mining revision histories to detect cross-language clones without intermediates. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 696–701.

[5] Xiao CHENG, Zhiming PENG, Lingxiao Jiang, Hao Zhong, Haibo Yu, and Jianjun Zhao. 2017. CLCMiner: Detecting Cross-Language Clones without Intermediates. *IEICE Transactions on Information and Systems* E100.D (02 2017), 273–284. https://doi.org/10.1587/transinf.2016EDP7334

[6] Xiao Cheng, Zhiming Peng, Lingxiao Jiang, Hao Zhong, Haibo Yu, and Jianjun Zhao. 2017. CLCMiner: detecting cross-language clones without intermediates. *IEICE TRANSACTIONS on Information and Systems* 100, 2 (2017), 273–284.

[7] Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. arXiv:2003.10555 [cs.CL]

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

[9] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155 [cs.CL]

[10] Nicholas Frosst, Nicolas Papernot, and Geoffrey Hinton. 2019. Analyzing and Improving Representations with the Soft Nearest Neighbor Loss. arXiv:1902.01889 [stat.ML]

[11] Zhipeng Gao, Xin Xia, David Lo, John Grundy, and Thomas Zimmermann. 2021. Automating the removal of obsolete TODO comments. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 218–229.

[12] Raia Hadsell, Sumit Chopra, and Yann LeCun. 2006. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, Vol. 2. IEEE, 1735–1742.

[13] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *29th International Conference on Software Engineering (ICSE'07)*. 96–105. https://doi.org/10.1109/ICSE.2007.30

[14] T. Kamiya, S. Kusumoto, and K. Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670. https://doi.org/10.1109/TSE.2002.1019480

[15] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. 2021. Supervised Contrastive Learning. arXiv:2004.11362 [cs.LG]

[16] Raghavan Komondoor and Susan Horwitz. 2001. Using slicing to identify duplication in source code. In *International static analysis symposium*. Springer, 40–56.

[17] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara G. Ryder. 2017. CCLearner: A Deep Learning-Based Clone Detection Approach. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2017), 249–260.

[18] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 473–485.

[19] Yinhan Liu, Jiatao Gu, Naman Goyal, Xian Li, Sergey Edunov, Marjan Ghazvininejad, Mike Lewis, and Luke Zettlemoyer. 2020. Multilingual Denoising Pre-training for Neural Machine Translation. *CoRR* abs/2001.08210 (2020). arXiv:2001.08210 https://arxiv.org/abs/2001.08210

[20] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692 (2019). arXiv:1907.11692 http://arxiv.org/abs/1907.11692

[21] Nikita Mehrotra, Navdha Agarwal, Piyush Gupta, Saket Anand, David Lo, and Rahul Purandare. 2020. Modeling Functional Similarity in Source Code with Graph-Based Siamese Networks. arXiv:2011.11228 [cs.SE]

[22] Kawser Wazed Nafi, Tonny Shekha Kar, Banani Roy, Chanchal K Roy, and Kevin A Schneider. 2019. Clcdsa: cross language code clone detection using syntactical features and api documentation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1026–1037.

[23] Yujia Qin, Yankai Lin, Ryuichi Takanobu, Zhiyuan Liu, Peng Li, Heng Ji, Minlie Huang, Maosong Sun, and Jie Zhou. 2020. ERICA: Improving Entity and Relation Understanding for Pre-trained Language Models via Contrastive Learning. *CoRR* abs/2012.15022 (2020). arXiv:2012.15022 https://arxiv.org/abs/2012.15022

[24] XiPeng Qiu, TianXiang Sun, YiGe Xu, YunFan Shao, Ning Dai, and XuanJing Huang. 2020. Pre-trained models for natural language processing: A survey. *Science China Technological Sciences* 63, 10 (Sep 2020), 1872–1897. https://doi.org/10.1007/s11431-020-1647-3

[25] Gordana Rakić. 2015. *Extendable and adaptable framework for input language independent static analysis*. Ph. D. Dissertation. University of Novi Sad (Serbia).

[26] Chanchal K Roy, Minhaz F Zibran, and Rainer Koschke. 2014. The vision of software clone management: Past, present, and future (keynote paper). In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 18–33.

[27] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC. *Proceedings of the 38th International Conference on Software Engineering* (May 2016). https://doi.org/10.1145/2884781.2884877

[28] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. FaceNet: A unified embedding for face recognition and clustering. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (Jun 2015). https://doi.org/10.1109/cvpr.2015.7298682

[29] Kihyuk Sohn. 2016. Improved Deep Metric Learning with Multi-class N-pair Loss Objective. In *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett (Eds.), Vol. 29. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2016/file/6b180037abbebea991d8b1232f8a8ca9-Paper.pdf

[30] Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 476–480.

[31] Jeffrey Svajlenko and Chanchal K. Roy. 2015. Evaluating clone detection tools with BigCloneBench. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 131–140. https://doi.org/10.1109/ICSM.2015.7332459

[32] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. 2019. Representation Learning with Contrastive Predictive Coding. arXiv:1807.03748 [cs.LG]

[33] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008).

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[35] Tijana Vislavski, Gordana Rakić, Nicolás Cardozo, and Zoran Budimac. 2018. LICCA: A tool for cross-language clone detection. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 512–516. https://doi.org/10.1109/SANER.2018.8330250

[36] Christian Wagner. 2014. *Model-Driven Software Migration: A Methodology: Reengineering, Recovery and Modernization of Legacy Systems*. Springer Science & Business Media.

[37] Andrew Walker, Tom Černý, and Eunjee Song. 2020. Open-source tools and benchmarks for code-clone detection: past, present, and future trends. *ACM SIGAPP Applied Computing Review* 19 (01 2020), 28–39. https://doi.org/10.1145/3381307.3381310

[38] Shuohuan Wang, Yu Sun, Yang Xiang, Zhihua Wu, Siyu Ding, Weibao Gong, Shikun Feng, Junyuan Shang, Yanbin Zhao, Chao Pang, Jiaxiang Liu, Xuyi Chen, Yuxiang Lu, Weixin Liu, Xi Wang, Yangfan Bai, Qiuliang Chen, Li Zhao, Shiyong Li, Peng Sun, Dianhai Yu, Yanjun Ma, Hao Tian, Hua Wu, Tian Wu, Wei Zeng, Ge Li, Wen Gao, and Haifeng Wang. 2021. ERNIE 3.0 Titan: Exploring Larger-scale Knowledge Enhanced Pre-training for Language Understanding and Generation. *CoRR* abs/2112.12731 (2021). arXiv:2112.12731 https://arxiv.org/abs/2112.12731

[39] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting Code Clones with Graph Neural Networkand Flow-Augmented Abstract Syntax Tree. arXiv:2002.08653 [cs.SE]
[40] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting Code Clones with Graph Neural Networkand Flow-Augmented Abstract Syntax Tree. *CoRR* abs/2002.08653 (2020). arXiv:2002.08653 https://arxiv.org/abs/2002.08653
[41] Huihui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. In *IJCAI*.
[42] Xing Wu, Chaochen Gao, Liangjun Zang, Jizhong Han, Zhongyuan Wang, and Songlin Hu. 2021. ESimCSE: Enhanced Sample Building Method for Contrastive Learning of Unsupervised Sentence Embedding. *CoRR* abs/2109.04380 (2021). arXiv:2109.04380 https://arxiv.org/abs/2109.04380
[43] Zhirong Wu, Yuanjun Xiong, Stella Yu, and Dahua Lin. 2018. Unsupervised Feature Learning via Non-Parametric Instance-level Discrimination. arXiv:1805.01978 [cs.CV]
[44] Dejiao Zhang, Feng Nan, Xiaokai Wei, Shang-Wen Li, Henghui Zhu, Kathleen R. McKeown, Ramesh Nallapati, Andrew O. Arnold, and Bing Xiang. 2021. Supporting Clustering with Contrastive Learning. *CoRR* abs/2103.12953 (2021). arXiv:2103.12953 https://arxiv.org/abs/2103.12953
[45] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 783–794. https://doi.org/10.1109/ICSE.2019.00086