computers

MDPI

Article

CLCD-I: Cross-Language Clone Detection by Using Deep Learning with InferCode

Mohammad A. Yahya and Dae-Kyoo Kim *

Computer Science and Engineering Department, Oakland University, Rochester, MI 48309, USA
* Correspondence: kim2@oakland.edu; Tel.: +1-248-370-2863

Abstract: Source code clones are common in software development as part of reuse practice. However, they are also often a source of errors compromising software maintainability. The existing work on code clone detection mainly focuses on clones in a single programming language. However, nowadays software is increasingly developed on a multilanguage platform on which code is reused across different programming languages. Detecting code clones in such a platform is challenging and has not been studied much. In this paper, we present CLCD-I, a deep neural network-based approach for detecting cross-language code clones by using InferCode which is an embedding technique for source code. The design of our model is twofold: (a) taking as input InferCode embeddings of source code in two different programming languages and (b) forwarding them to a Siamese architecture for comparative processing. We compare the performance of CLCD-I with LSTM autoencoders and the existing approaches on cross-language code clone detection. The evaluation shows the CLCD-I outperforms LSTM autoencoders by 30% on average and the existing approaches by 15% on average.

Keywords: abstract syntax tree; code clone detection; cross language; deep neural network; Java; Python

Citation: Yahya, M.A.; Kim, D.-K. CLCD-I: Cross-Language Clone Detection by Using Deep Learning with InferCode. *Computers* **2023**, *12*, 12. https://doi.org/10.3390/computers12010012

Academic Editors: Phivos Mylonas, Katia Lida Kermanidis and Manolis Maragoudakis

Received: 11 December 2022
Revised: 27 December 2022
Accepted: 31 December 2022
Published: 4 January 2023

1. Introduction

Code clones are code fragments that have the same or similar syntax and semantics [1]. They are often practiced in software development as part of reuse effort [2]. However, code clones are in general considered to hamper the quality of software due to misfit and modifications made during reuse which often introduce errors [3]. They also require that when one clone is changed, all other clones are to be updated for consistency [4]. Code clones are typically categorized into Type I, II, III, and IV where Type I is textual clones (i.e., identical codes), Type II is lexical clones (e.g., different identifiers), Type III is syntactic clones (e.g., additional statements), and Type IV is semantic clones (i.e., the same logic) [5,6].

Deep learning has been increasingly adopted in code clone detection (e.g., [7–12]). The general approach is that source code is represented in abstraction such as abstract syntax trees, tokens, and control flow graphs and embedded by using embedding techniques, such as word2vec, node2vec, and graph2vec to learn structural similarity. Most existing work focuses on code clone detection in the same programming language [13]. However, nowadays software is increasingly developed on a multilanguage platform on which similar functionalities are used across multiple programming languages. For example, common APIs for big data processing such as Apache Spark have similar naming and call patterns written in different programming languages [14,15]. Such an environment often involves clones in multiple languages and requires consistent updates across clones when a change is made on one clone. There exists some work (e.g., [14,15]) on cross-language code clone detection. However, their approaches suffer from poor performance mainly due to low-quality features used in learning and prediction.

In this paper, we present CLCD-I, a cross-language clone detection approach by using InferCode [16], which is an embedding technique for source code. The model takes as

input pretrained embeddings from InferCode to learn abstract features of source code by grouping related embeddings. The embeddings are then fed into a Siamese architecture [17] for comparative processing of source code in Java and Python. InferCode embeddings capture only syntactic features, and thus, this work focuses on clones of Type III. We evaluate CLCD-I by using a dataset containing more than 40,000 pairs of Java and Python code snippets. We compare the performance of CLCD-I with LSTM autoencoders, which are widely used for the translation of languages. We also compare CLCD-I with the existing approaches on cross-language code clone detection. Specifically, we seek to answer the following research questions.

1. Is CLCD-I more effective than LSTM autorencoders?
2. Is CLCD-I more effective than existing approaches?

The remainder of the paper is organized as follows. Section 2 discusses the existing work on code clone detection by using machine learning, Section 4 describes source code embedding by using InferCode, Section 5 presents CLCD-I, Section 7 evaluates CLCD-I and discusses the results of the evaluation, and Section 8 concludes the paper with discussion on future work.

2. Related Work

There exists much work on code clone detection by using deep learning [13]. The general approach is that source code is represented in abstraction and vectorized by using an embedding technique. Then, the embeddings are input to a deep learning model for learning and prediction. For source code abstraction, abstract syntax tree (AST) [7,18–20], control flow graph (CFG) [8,9], data flow graph (DFG) [21], program dependency graph (PDG) [22], bytecode dependency graph (BDG) [22], application user interface (API) call similarity [14,23], function call graph (FCG) [24], and size-based sharding (SBS) [25] were used with dominance by AST and CFG. Regarding embedding techniques, word2vec [26], graph2vec [27], node2vec [28], doc2vec [29], position-aware character embedding (PACE) [30], one-hot embedding [31], continuous-valued vector (CVV) [18], and recursive autoencoder (RAE) [10] were used.

Various deep learning models have been used for code clone detection, including deep neural network (DNN) [14,19,21,24,25,32,33], recurrent neural network (RNN) [7–9,18,20,34], graph neural network (GNN) [11,35,36], long short-term memory (LSTM) [12,15], and convolutional neural network (CNN) [22,23,30,37]. To improve performance, some work [10,14,18,19,25] used filters to filter out data and some other work [9,12,34] used supplementary training strategies such as attention mechanisms and adversarial training. For evaluation, BigCloneBench [38] and OJClone [39] were commonly used as primary benchmarks. Others include Apache Ant, ArgoUML, JHotDraw, IJaDataset, NetBeans-Javadoc, Eclipse-Ant, Apache Software Foundation, GitHub, Google Code Jam, JDK, and Maven. With regard to programming languages, C, Java, C#, C++, Python, and Go were considered where C and Java were more common. With respect to clone types, Type IV was more focused.

The majority of the existing work dealt with clone detection in the same language. Only a few studies (e.g., [14,15]) looked into cross-language clone detection. Perez and Chiba [15] proposed an LSTM-based approach for detecting code clones in Java and Python programs. They used a Siamese architecture [17] for comparative processing of Java and Python programs. An AST transformer was used to transform the AST of source code into a vector whose elements are the indices of AST nodes. Then, a skip-gram model (unsupervised learning) was used to generate vectors of nodes based on the indices, which is the uniqueness of their approach. These vectors were fed into a stacked bidirectional LSTM encoder to produce a single vector in a high-dimensional space. A hash layer was used to take the output of the LSTM to reduce the dimension. The output of the hash layer was then input to a feed-forward neural network (classifier) to produce a similarity score. Nafi et al. [14] proposed a DNN-based approach for detecting code clones in Java, C#, and Python programs. Similar to Perez and Chiba's work, they also made use of a Siamese architecture for comparative processing of code in two different languages. Source code was

preprocessed (e.g., removing comments) and represented in an AST. From the AST, features of the source code were extracted based on the feature set proposed by Saini et al. [25]. The features were then input to two identical subnetworks for vector transformation. The outputs of the subnetworks were concatenated and input to a comparator. They used API call mapping to learn similarity of code in two different languages in which each API of one language is mapped to the corresponding API in the other language based on API call documentation, which was vectorized by using word2vec. An action filter was used to filter out nonprobable clone pairs based on similarity. Only filtered pairs were input to the comparator. Finally, the output of the comparator was input to the classification unit to determine the cloneness of a pair. A major drawback in their approach is the manual mapping of API calls between languages, which is time consuming and error prone, as acknowledged in their work.

The existing work on cross-language clone detection suffers from poor performance. Perez and Chiba reported an F1 score of 0.23, a precision of 0.19, and a recall of 0.90. Corresponding measures of Nafi et al.'s work were 0.57, 0.68, and 0.5, which is slightly better than Perez and Chiba's work, but still not viable because the improvement was mainly aided by the heavy manual process for API mapping. CLCD-I notably outperforms the existing work without requiring any manual handling.

3. Data Preparation

We conducted the experiments based on a dataset from a programming competition website (https://atcoder.jp [accessed: 13 May 2022]), which collects Java and Python programs to solve various problems (e.g., Fibonacci series) and classifies them per problem title. Each submission was made in only one language (either Python or Java) per problem title. Thus, it can be ensured that the submissions under the same title are clones. Figure 1 shows an example of a clone in Java and Python that solves the selection sort. The two programs exhibit similar structures of nested loops with a comparison construct at the end, which leads to similar ASTs.

```java
public void selectionSort(int data[]) {
  for (int i = 0; i < data.length - 1; i++) {
    int min = i;
    for (int j = i + 1; j < data.length; j++) {
      if (data[j] < data[min]) {
        min = j;
      }
    }
    int temp = data[i];
    data[i] = data[min];
    data[min] = temp;
  }
}
```

(a) Selection Sort in Java

```python
def selectionSort(data, size):
    for i in range(size):
        min = i
        for j in range(i + 1, size):
            if data[j] < data[min]:
                min = j
        (data[i], data[min]) = (data[min], data[i])
```

(b) Selection Sort in Python

Figure 1. Clone example: Selection sort in Java and Python.

For this work, we collected 22,318 Python programs submitted for 461 problems and 20,828 Java programs submitted for 504 problems and paired them for 445 common problems. In our experiment, we split the dataset into 70% for training and 30% for validation. Figure 2 shows the embedding process by using InferCode. For contrastive learning, we augmented the collected pairs of Java and Python programs by adding a label of "1" for clones and "0" for disclones. The labeled pairs are then passed to InferCode to produce embeddings.

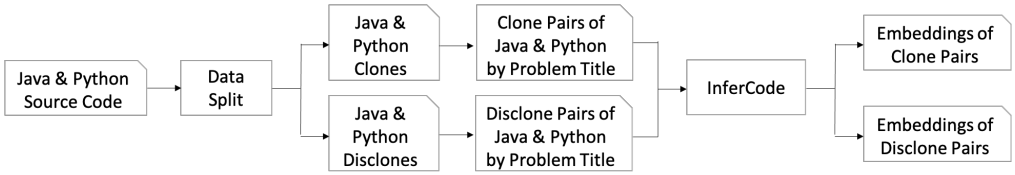


Figure 2. Source code embedding by using InferCode.

In order to avoid too many clone pairs of Java and Python code, we merge the Python code set and the Java code set by using the inner join operation with indexing. Figure 3 illustrates the merging process. In each set, the programs that solve the same problem are indexed. Then, the programs in the two sets that solve the same program and the same index are merged. Algorithm 1 describes the merging process. Disclone pairs are created in a similar process by using different problems.

Python Code	Problem Title	Index
p1	t1	1
p2	t1	2
p3	t2	1
p4	t2	2
...	...	...

Java Code	Problem Title	Index
j1	t1	1
j2	t1	2
j3	t2	1
j4	t2	2
...	...	...

Python Code	Java Code	Problem Title	Index
p1	j1	t1	1
p2	j2	t1	2
p3	j3	t2	1
p4	j4	t2	2
...	...	...	...

Figure 3. Merging Java set and Python set.

Algorithm 1 Create Clone Pairs

1: $createClonePairs(dataset) : clonePairs$
2: $< dataset\_Python, dataset\_Java > \leftarrow split(dataset)$
3: $grouped\_Python \leftarrow groupByProblemTitle(dataset\_Python)$
4: $grouped\_Java \leftarrow groupByProblemTitle(dataset\_Java)$
5: $indexed\_Python \leftarrow index(grouped\_Python)$
6: $indexed\_Java \leftarrow index(grouped\_Java)$
7: $clonePairs \leftarrow join(indexed\_Python, indexed\_Java)$

4. Source Code Embedding by Using InferCode

We use InferCode [16] for embedding source code. InferCode is a self-supervised learning technique for embedding source code based on the subtrees of an AST. A strong benefit of InferCode is that it requires no human labeling for unlabeled datasets. In our work, it learns to label code snippets by using AST subtrees. A label is generated for each code snippet based on the subtrees of the AST representing the code. After learning based on the label, prediction is then made on the probability of the subtrees existing in given source code by minimizing the loss between the subtrees and the code. The more data are labeled, the higher accuracy of prediction.

InferCode extends tree-based convolutional neural network (TBCNN) [39,40], a source code encoder, by including the textual information (i.e., tokens) in node initialization in addition to its type information and replacing the dynamic max pooling with an attention mechanism to aggregate nodes into one fixed embedding. Code clone detection requires

exhaustive feature engineering [41] which is even more demanding in cross-language clone detection. The existing work [14,15] on cross-language clone detection relies on manual feature engineering, which is prone to bias and inaccurate in feature selection. The self-supervised learning in TBCNN makes manual feature engineering obsolete. There also exist other embedding techniques, such as code2vec [42] and code2seq [43], which are designed for a specific task, such as unsupervised learning code clustering (code2vec) and supervised name prediction(code2seq). Unlike these techniques, TBCNN is meant to be more general supporting various downstream tasks, which makes it suitable for code clone detection in different languages.

InferCode takes three steps to learn source code representations—(i) learning node representations, (ii) aggregating node embeddings into a single embedding, and (iii) learning the probability of a subtree for prediction. For the AST of given source code, the information of the AST is accumulated from bottom to top where a parent node contains information about its descendants. Each node v_i in a subtree of AST is associated with D-dimensional real-valued vector $x_v \in \mathbb{R}^D$ represented as $x_v = x_{type} + x_{token}$ (token and text of v_i) as an input feature. A convolution window over the AST is created and moves via a binary tree. Figure 4 illustrates the convolution window moving over an AST from bottom to top. The weight matrix of each node is computed by the weighted sum of W^t, W^l, W^r $\in \mathbb{R}^{D\times D}$each representing the weight of top, left, and right in the binary tree and a bias term $b \in \mathbb{R}^D$. The convolutional output y of the window is then computed as Equation (1) where α represents the activation function and n_i^t, n_l^t, and n_r^t are weights reflecting the depth and position of each node [16]. We have

$$y = \alpha\left(\sum_{n=1}^{K}[n_i^t W^t + n_i^l W^l + n_i^r W^r]x_i + b\right). \tag{1}$$

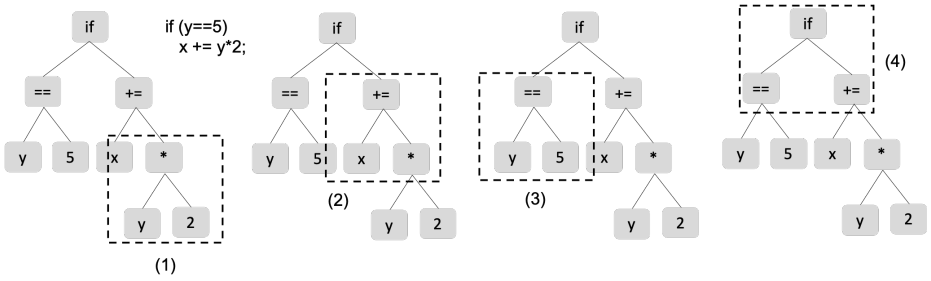


Figure 4. Convolution window moving over an AST in TBCNN.

Node embeddings are combined into one fixed embedding representing the whole AST by using an attention mechanism. The attention weight α_i of each node $\vec{h}_i$ is computed as Equation (2) where $\vec{a}$ is the global attention vector, which is randomly initialized and learned as networks are updated. This assigns a higher weight to a similar subtree. We have

$$\alpha_i = \frac{exp(\vec{h}_i \cdot \vec{a})}{\sum_{j=1}^{n} exp(\vec{h}_j \cdot \vec{a})}. \tag{2}$$

Given that, the aggregated vector $\vec{v}$ is computed as Equation (3):

$$\vec{v} = \sum_{i=1}^{n} \alpha_i \cdot \vec{h}_i. \tag{3}$$

The aggregated vector with attention weight makes the Siamese architecture in CLCD-I much lighter in terms of the amount of data needed for training. Given the aggregated vector, the probability $q(l_i)$ of a subtree $l_i \in L$ (the set of subtrees) to appear in the given

code snippet is computed as Equation (4) where $W_i^{subtrees}$ is the embedding of a subtree l_i [16]:

$$q(l_i) = \frac{exp(\vec{v} \cdot W_i^{subtrees})}{\sum_{l_j \in L}^{n} exp(\vec{v} \cdot W_j^{subtrees})}. \quad (4)$$

5. CLCD-I: Deep Learning Approach

We present CLCD-I, a deep learning-based approach for cross-language code clone detection. The collection of Java and Python code pairs is split into a clone set and a disclone set. The sets are then input to InferCode to generate embeddings. The embeddings are fed into a Siamese architecture for comparative process of Java and Python code. Figure 5 shows the Siamese architecture. It consists of two equivalent networks that have the same architecture and trainable parameters. The model takes as input pairs of similar embeddings for learning clones and dissimilar embeddings for learning disclones. The model has an alternation of dense and dropout layers for deep learning while reducing overfitting. Unlike Perez et al.'s work [15] where different weights were used between the two networks of their Siamese model, we use the same weight in both networks of our model. They considered that the two networks have different domains as each network takes a different programming language. However, we argue that the domains are of the same nature as both networks take embeddings at the statement-level and thus, the same weight should be used. The weight is updated consistently between the networks during backpropagation. We measure similarity between the output of the two networks by using the pairwise contrastive loss function [44], which calculates the Euclidean distance between the embeddings in a pair. If their similarity is above the preset threshold [17], then the pair is considered as similar and labeled with "1". Otherwise, the pair is considered as dissimilar and labeled with "0". The output of the loss function is then backpropagated for learning. The model is updated only for similar clones to reinforce the distinction of similar clones from dissimilar clones.

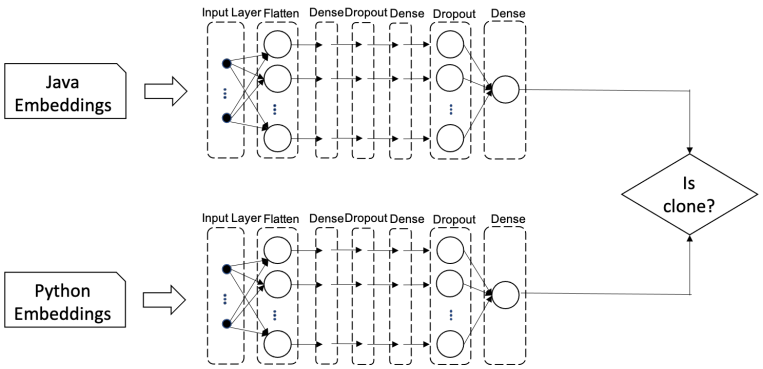


Figure 5. The Siamese architecture in CLCD-I.

For a pair of Python embedding py_{emb} and Java embedding $java_{emb}$, the Siamese model produces the weighted output $O_W(py_{emb})$ from the Python network and the weighted output $O_W(java_{emb})$ from the Java network. Given those outputs, their Euclidean distance approximating the similarity of the two embeddings is computed as Equation (5):

$$D_W(py_{emb}, java_{emb}) = \| O_W(py_{emb}) - O_W(java_{emb}) \|_2 . \quad (5)$$

Given the distance, the loss function is defined as Equation (6), where $Y \in \{0,1\}$ (0 indicating dissimilarity and 1 indicating similarity) and m is a margin to limit the contribution of dissimilar pairs to the loss function. That is, only the pair whose distance is within the margin is allowed to contribute to the function. If the distance is higher than the margin ($D_W(x_i) > m$), $m - D_W$ becomes negative and thus, 0 is chosen, which makes the

distance ignored. That is, for a given pair x_i, the equation results in $(1-Y)\frac{1}{2}\left(D_W(x_i)^2\right)+0$ if the pair is similar and $0+(Y)\frac{1}{2}\left(\max\{0,\ m-D_W(x_i)\}^2\right)$ if dissimilar. We have

$$L(Y, W, py_{emb}, jv_{emb}) = (1-Y)\frac{1}{2}(D_W)^2 + (Y)\frac{1}{2}\{\max(0,\ m-D_W)\}^2. \quad (6)$$

Given Equation (6), the total loss for k number of similar pairs and k number of dissimilar pairs (a total of $2k$ number of pairs) is computed as Equation (7):

$$L(W) = \sum_{i=1}^{2k} L(W, Y, py^i_{emb}, jv^i_{emb}). \quad (7)$$

6. Compared Models

For a comparative study, we compare CLCD-I with three LSTM models—a vanilla LSTM autoencoder (AE), LSTM AE with Loung attention (LA) [45], and LSTM AE with Bahdanau attention (BA) [46]. A vanilla LSTM AE is an implementation of the encoder–decoder LSTM architecture for processing sequential data. It was chosen because the cross-language clone detection problem can be viewed as a translation problem where a Python code snippet is translated into its Java counterpart (or vice versa) if they are clones. Cross-language clone detection is also a sequence to sequence (seq2seq) problem where each AST token is processed one at a time step by the encoder and decoder, which makes LSTM AE suitable for the problem.

Figure 6 shows the vanilla LSTM AE used in this work. It consists of an encoder and a decoder. The encoder takes Java embeddings as input and produces a context vector. The decoder takes the context vector together with Python embeddings as input and produces the probability of the Java embeddings being similar to the Python embeddings. The cell state of an LSTM unit in the encoder accumulates hidden states from all previous units. That is, the cell state of the last unit contains the hidden state of all the units in the encoder. However, when the input sequence is long, information loss might occur in the last unit, leading to low accuracy, which is a downside of vanilla LSTM AE.

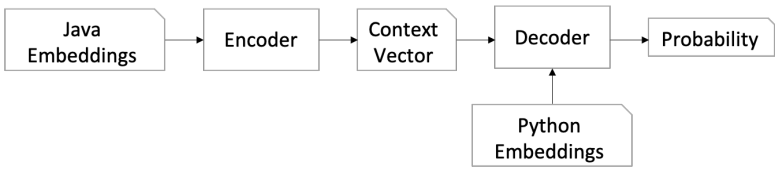


Figure 6. Vanilla LSTM AE.

LSTM with BA (LSTM+BA) addresses the limitation of vanilla LSTM AE by using BA. Figure 7 shows the LSTM+BA model used in this work. Unlike vanilla LSTM AE where only the last hidden state of the encoder is used, LSTM+BA uses all the hidden states of the encoder along with the decoder's hidden states generated at all the time steps. The decoder states are initialized by using all the hidden states of the encoder. At each decoding time step—(1) the encoder's all hidden states and the previous decoder's output are used to calculate a global context vector by using BA, and (2) the context vector is concatenated with the previous decoder's output to create the input to the current decoder. BA scores how well the hidden states h_s of the encoder matches the previous hidden state h_{t-1} of the decoder (previous time step output). Equation (8) defines the alignment score function of BA [46] where h_{t-1} is the previous hidden state of the decoder, h_s is all the hidden states of the encoder, T is the transpose of a matrix, and W is the weight matrix for parameterizing the calculations. Due to the use of the addition operation in the function, BA is also referred to as additive attention. We have

$$a(h_t, h_s) = v_a^T \tanh(W_1 h_{t-1} + W_2 h_s). \quad (8)$$

The score is then input to the softmax function to calculate the attention weight as Equation (9) where S is the length of the input sequence:

$$\alpha_{ts} = \frac{\exp(score(h_t, h_s))}{\sum_{s'=1}^{S} \exp(score(h_t, h_{s'}))}. \tag{9}$$

The context vector c_t is built by applying the attention weight to the hidden states of the decoder as Equation (10):

$$c_t = \sum_{s=1}^{S} \alpha_{ts} h_s. \tag{10}$$

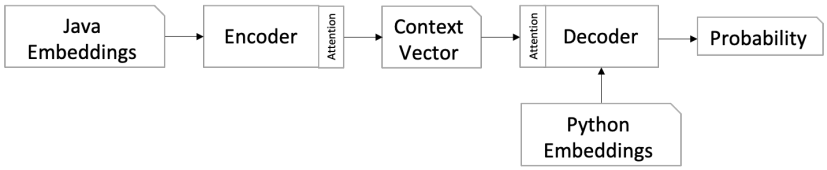


Figure 7. LSTM+BA.

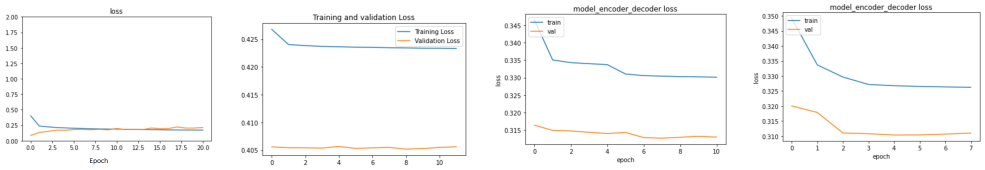
LSTM with LA (LSTM+LA) is similar to LSTM+BA from an architectural perspective. A major difference lies in the computation of alignment scores. Unlike BA, where the score function uses the previous hidden state of the decoder, LA's score function uses the current hidden state of the decoder. Moreover, whereas BA uses an addition operation in the score function, LA uses a multiplicative operation in the function. Equation (11) shows three alternative functions of LA for computing alignment scores [45]. In this work, we use the general function. We have

$$score(h_t, h_s) = \begin{cases} h_t^T h_s & dot \\ h_t^T W h_s & general \\ h_a^T tanh(W[h_t; h_s]) & concat. \end{cases} \tag{11}$$

7. Evaluation

We compared CLCD-I with the three LSTM models in Section 5 for their performance. For consistency, we configured the three LSTM models to have the same 16 LSTM units in their encoder and decoder. The input of the LSTM models was split into four time steps. For a fair comparison, we set the number of epochs for all the four models to 25 with the same patience level in the early stopping mode which terminates the training when validation and training loss no longer converge. The vanilla LSTM AE stopped at the 25th epoch, whereas LSTM+BA and LSTM+LA stopped at the 20th and 21st epoch respectively. The SGD optimizer was used in all models. About 17,000 pairs of Python and Java programs were used. The models were trained with only similar pairs to learn cloneness.

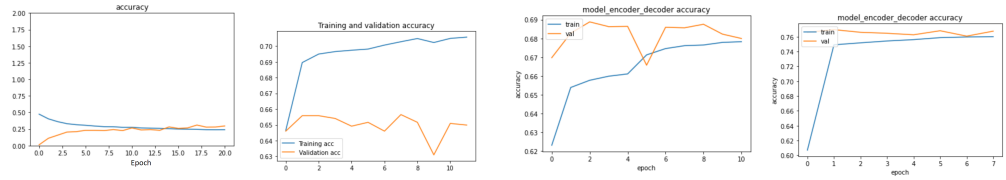
Figure 8 shows the loss of the four models. It demonstrates a decent convergence on training loss except the vanilla LSTM AE. CLCD-I outperformed the others in both training and validation loss, which indicates its well fitting with the training and validation data. The narrow gap between the training and validation loss of CLCD-I indicates better generalizability to unseen data.



(**a**) CLCD-I (**b**) Vanilla LSTM AE (**c**) LSTM+BA (**d**) LSTM+LA

Figure 8. Training and validation loss.

Figure 9 shows the accuracy of the four models. LSTM+BA and LSTM+LA demonstrated competitive accuracy to each other in both training and validation. The vanilla LSTM shows competitive training accuracy, but lower validation accuracy. CLCD-I demonstrated the worst accuracy. However, CLCD-I outperformed the others on F1 measure by 30% on average as shown in Table 1. This answers "Yes" to the research question (1) posed in Section 1. This indicates that CLCD-I is more effective on false negatives and false positives.



(**a**) CLCD-I (**b**) Vanilla LSTM (**c**) LSTM+BA (**d**) LSTM+LA

Figure 9. Training and validation accuracy.

Table 1. Performance comparison.

Models	Precision	Recall	F1-Measure	Accuracy
CLCD-I	**0.99**	**0.63**	**0.78**	0.40
Vanilla LSTM	0.60	0.45	0.53	**0.76**
LSTM+BA	0.59	0.45	0.54	0.75
LSTM+LA	0.62	0.53	0.56	0.74

We also compared CLCD-I with the existing cross-language code clone approaches—CLCDSA [14] and Perez and Chiba's work [15]. Table 2 shows the result. CLCD-I outperformed the existing approaches on F1 measure by 15% on average, which answers the research question (2) posed in Section 1. We think that it is likely due to the improved efficiency by the contrastive loss function (cf. Section 5).

Table 2. Perfomance comparison of CLCD-I with other models.

Models	Precision	Recall	F1-Measure
CLCD-I	**0.99**	0.63	**0.78**
CLCDSA [14]	0.67	0.65	0.66
Perez and Chiba's work [15]	0.66	**0.83**	0.66

8. Conclusions

We have presented CLCD-I for detecting cross-language code clones. CLCD-I uses InferCode for source code embedding and a Siamese architecture with the quadratic contrastive loss function to improve performance. A main advantage of CLCD-I comes from the use of InferCode which self-learns labeling code snippets by using AST subtrees, which removes the burden of manual labelling. CLCD-I was experimented for comparative performance with vanilla LSTM AE, LSTM+BA, and LSTM+LA. CLCD-I underperformed

on accuracy, but outperformed on precision, recall, and F1 measure, which indicates CLCD-I is more efficient on false negatives and false positives. In comparison to the existing work, CLCD-I outperformed on F1 measure. CLCD-I is limited to Type III clones. We plan to further investigate semantic clones of Type IV by mapping source code to control flow graphs and generating embeddings from control flow graphs by using graph2vec. We also plan to look at clones in other languages such as C++ and C#.

Author Contributions: conceptualization, M.A.Y.; methodology, M.A.Y.; validation, M.A.Y.; investigation, M.A.Y.; writing—original draft preparation, M.A.Y.; writing–review and editing, D.-K.K.; supervision, D.-K.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: All data were presented in main text.

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

References

1. Baxter, I.D.; Yahin, A.; Moura, L.; Sant'Anna, M.; Bier, L. Clone detection using abstract syntax trees. In Proceedings of the International Conference on Software Maintenance, Bethesda, MD, USA, 16–19 March 1998; pp. 368–377.
2. Di Lucca, G.A.; Di Penta, M.; Fasolino, A.R. An approach to identify duplicated web pages. In Proceedings of the 26th Annual International Computer Software and Applications, Oxford, UK, 26–29 August 2002; pp. 481–486.
3. Monden, A.; Nakae, D.; Kamiya, T.; Sato, S.; Matsumoto, K. Software quality analysis by code clones in industrial legacy software. In Proceedings of the 8th IEEE Symposium on Software Metrics, Ottawa, ON, Canada, 4–7 June 2002; pp. 87–94.
4. Krinke, J. A Study of Consistent and Inconsistent Changes to Code Clones. In Proceedings of the 14th Working Conference on Reverse Engineering, Vancouver, BC, Canada, 28–31 October 2007; pp. 170–178.
5. 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]
6. Roy, C.K.; Cordy, J.R. A survey on software clone detection research. *Queen's Sch. Comput. TR* **2007**, *541*, 64–68.
7. 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 41st IEEE/ACM International Conference on Software Engineering, Montreal, QC, Canada, 25–31 May 2019.
8. Yuan, Y.; Kong, W.; Hou, G.; Hu, Y.; Watanabe, M.; Fukuda, A. From Local to Global Semantic Clone Detection. In Proceedings of the 6th IEEE International Conference on Dependable Systems and Their Applications, Harbin, China, 3–6 January 2020.
9. 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]
10. Zeng, J.; Ben, K.; Li, X.; Zhang, X. Fast Code Clone Detection Based on Weighted Recursive Autoencoders. *IEEE Access* **2019**, *7*, 125062–125078. [CrossRef]
11. 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, London, ON, Canada, 18–21 February 2020.
12. Meng, Y.; Liu, L. A Deep Learning Approach for a Source Code Detection Model Using Self-Attention. *Complexity* **2020**, *2020*, 5027198. [CrossRef]
13. Lei, M.; Li, H.; Li, J.; Aundhkar, N.; Kim, D.K. Deep learning application on code clone detection: A review of current knowledge. *J. Syst. Softw.* **2022**, *184*, 111141. [CrossRef]
14. 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, San Diego, CA, USA, 10–15 November 2019.
15. Perez, D.; Chiba, S. Cross-language clone detection by learning over abstract syntax trees. In Proceedings of the 16th IEEE/ACM International Conference on Mining Software Repositories, Montreal, QC, Canada, 26–27 May 2019; pp. 518–528.
16. Bui, N.D.; Yu, Y.; Jiang, L. InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees. In Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering, Virtual, 25–28 May 2021; pp. 1186–1197.
17. Bromley, J.; Guyon, I.; LeCun, Y.; Säckinger, E.; Shah, R. Signature verification using a "siamese" time delay neural network. In Proceedings of the 6th International Conference on Neural Information Processing Systems, Denver, CO, USA, 29 November–2 December 1993; pp. 737–744.
18. White, M.; Tufano, M.; Vendome, C.; Poshyvanyk, D. Deep Learning Code Fragments for Code Clone Detection. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Singapore, 3–7 September 2016.

19. 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, Shanghai, China, 17–22 September 2017.
20. Wei, H.; Li, M. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. In Proceedings of the 26th International Joint Conference on Artificial Intelligence, Melbourne, Australia, 19–25 August 2017.
21. 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 4–9 November 2018.
22. Sheneamer, A. CCDLC Detection Framework-Combining Clustering with Deep Learning Classification for Semantic Clones. In Proceedings of the 17th IEEE International Conference on Machine Learning and Applications, Orlando, FL, USA, 17–20 December 2018.
23. Chen, L.; Wei, Y.; Zhang, S. Capturing Source Code Semantics via Tree-Based Convolution over API-Enhanced AST. In Proceedings of the 16th ACM International Conference on Computing Frontiers, Alghero, Italy, 30 April–2 May 2019.
24. Fang, C.; Liu, Z.; Shi, Y.; Huang, J.; Shi, Q. Functional Code Clone Detection with Syntax and Semantics Fusion Learning. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual, 18–22 July 2020.
25. Saini, V.; Farmahinifarahani, F.; Lu, Y.; Baldi, P.; Lopes, C. Oreo: Detection of Clones in the Twilight Zone. 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, 4–9 November 2018.
26. Mikolov, T.; Chen, K.; Corrado, G.; Dean, J. Efficient Estimation of Word Representations in Vector Space. *arXiv* **2013**, arXiv:abs/1301.3781
27. Narayanan, A.; Chandramohan, M.; Venkatesan, R.; Chen, L.; Liu, Y.; Jaiswal, S. Graph2vec: Learning Distributed Representations of Graphs. *arXiv* **2017**, arXiv:abs/1707.05005.
28. Grover, A.; Leskovec, J. node2vec: Scalable Feature Learning for Networks. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, 13–17 August 2016.
29. Le, Q.; Mikolov, T. Distributed Representations of Sentences and Documents. In Proceedings of the 31st International Conference on Machine Learning, Beijing, China, 21–26 June 2014.
30. Yu, H.; Lam, W.; Chen, L.; Li, G.; Xie, T.; Wang, Q. Neural Detection of Semantic Code Clones Via Tree-Based Convolution. In Proceedings of the 27th IEEE/ACM International Conference on Program Comprehension, Montreal, QC, Canada, 25 May 2019; pp. 70–80.
31. Koehren, W. Neural Network Embeddings Explained. 2018. Available online: https://towardsdatascience.com/neural-network-e\protect\discretionary{\char\hyphenchar\font}{}{}mbeddings-explained-4d028e6f0526 (accessed on 20 March 2020).
32. Wang, C.; Gao, J.; Jiang, Y.; Xing, Z.; Zhang, H.; Yin, W.; Gu, M.; Sun, J. Go-Clone: Graph-Embedding Based Clone Detector for Golang. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, Beijing, China, 15–19 July 2019.
33. Gao, Y.; Wang, Z.; Liu, S.; Yang, L.; Sang, W.; Cai, Y. TECCD: A Tree Embedding Approach for Code Clone Detection. In Proceedings of the IEEE International Conference on Software Maintenance and Evolution, Cleveland, OH, USA, 29 September–4 October 2019.
34. Wei, H.; Li, M. Positive and Unlabeled Learning for Detecting Software Functional Clones with Adversarial Training. In Proceedings of the 27th International Joint Conference on Artificial Intelligence, Stockholm, Sweden, 13–19 July 2018.
35. Li, Y.; Tarlow, D.; Brockschmidt, M.; Zemel, R. Gated Graph Sequence Neural Networks. *arXiv* **2015**, arXiv:abs/1511.05493.
36. Scarselli, F.; Gori, M.; Tsoi, A.C.; Hagenbuchner, M.; Monfardini, G. The Graph Neural Network Model. *IEEE Trans. Neural Netw.* **2009**, *20*, 61–80. [CrossRef] [PubMed]
37. Guo, C.; Yang, H.; Huang, D.; Zhang, J.; Dong, N.; Xu, J.; Zhu, J. Review Sharing via Deep Semi-Supervised Code Clone Detection. *IEEE Access* **2020**, *8*, 24948–24965. [CrossRef]
38. Svajlenko, J.; Islam, J.F.; Keivanloo, I.; Roy, C.K.; Mia, M. Towards a Big Data Curated Benchmark of Inter-Project Code Clones. In Proceedings of the International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, 29 September–3 October 2014.
39. Mou, L.; Li, G.; Zhang, L.; Wang, T.; Jin, Z. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In Proceedings of the 30th AAAI Conference on Artificial Intelligence, Phoenix, AZ, USA, 12–17 February 2016; pp. 1287–1293.
40. Mou, L.; Peng, H.; Li, G.; Xu, Y.; Zhang, L.; Jin, Z. Discriminative neural sentence modeling by tree-based convolution. *arXiv* **2015**, arXiv:1504.01106.
41. Chilowicz, M.; Duris, E.; Roussel, G. Syntax tree fingerprinting for source code similarity detection. In Proceedings of the 17th IEEE International Conference on Program Comprehension, Vancouver, BC, Canada, 17–19 May 2009; pp. 243–247.
42. Alon, U.; Zilberstein, M.; Levy, O.; Yahav, E. code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.* **2019**, *3*, 1–29. [CrossRef]
43. Alon, U.; Brody, S.; Levy, O.; Yahav, E. code2seq: Generating sequences from structured representations of code. *arXiv* **2018**, arXiv:1808.01400.

44. Hadsell, R.; Chopra, S.; LeCun, Y. Dimensionality reduction by learning an invariant mapping. In Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, New York, NY, USA, 17–22 June 2006; pp. 1735–1742.
45. Luong, M.T.; Pham, H.; Manning, C.D. Effective approaches to attention-based neural machine translation. *arXiv* **2015**, arXiv:1508.04025.
46. Bahdanau, D.; Cho, K.; Bengio, Y. Neural machine translation by jointly learning to align and translate. In Proceedings of the 3rd International Conference on Learning Representations, San Diego, CA, USA, 7–9 May 2015; pp. 1–15.