Two-Pass Technique for Clone Detection and Type Classification Using Tree-Based Convolution Neural Network †

Young-Bin Jo, Jihyun Lee * and Cheol-Jung Yoo

Department of Software Engineering, Jeonbuk National University, Jeonju 54896, Korea;
opwe37@gmail.com (Y.-B.J.); cjyoo@jbnu.ac.kr (C.-J.Y.)
* Correspondence: jihyun30@jbnu.ac.kr; Tel.: +82-63-270-4860
† This paper is based on dissertation research completed at Jeonbuk National University under the direction of Jihyun Lee and Cheol-Jung Yoo.

Abstract: Appropriate reliance on code clones significantly reduces development costs and hastens the development process. Reckless cloning, in contrast, reduces code quality and ultimately adds costs and time. To avoid this scenario, many researchers have proposed methods for clone detection and refactoring. The developed techniques, however, are only reliably capable of detecting clones that are either entirely identical or that only use modified identifiers, and do not provide clone-type information. This paper proposes a two-pass clone classification technique that uses a tree-based convolution neural network (TBCNN) to detect multiple clone types, including clones that are not wholly identical or to which only small changes have been made, and automatically classify them by type. Our method was validated with BigCloneBench, a well-known and wildly used dataset of cloned code. Our experimental results validate that our technique detected clones with an average rate of 96% recall and precision, and classified clones with an average rate of 78% recall and precision.

Keywords: clone detection; clone type classification; CNN; TBCNN; clone-and-own approach

1. Introduction

Developers often develop software using copied and modified existing code fragments, referred to as “code clone” or simply “clone” [1,2]. These code fragments can ultimately make the code larger, more complex, and more challenging to maintain. According to some research, between 20% and 59% of all source code is duplicated [3–5].

The ‘clone-and-own’ approach, in which preexisting code fragments are used to fulfill new requirements, is widely employed when a new software system is similar but not identical to an existing system. While this method does allow for the rapid and cost-effective development of software systems, code clones can render it more difficult to maintain a system, and they result in the copying of subtle errors across systems [6,7]. Once a bug is found in a clone, it has to be checked everywhere that clone was deployed, and any modifications have to be repeated across all systems. This can increase the costs associated with the development and maintenance of large-scale software systems. When cloned code fragments are not searched for, identified, and possibly refactored, the same bug (and adverse effects) will be repeated across software systems.

According to S. Bellon et al. [1] and K. Roy et al. [2], code clones can be divided into four types: “an exact copy without modifications (Type 1, T1),” “a structurally and syntactically identical fragments except variable, type, or function identifiers (Type 2, T2),” “a copied fragments with further modifications in which statements were changed, added, or removed (Type 3, T3),” and “code fragments that perform the same computation but are implemented by different syntactic variants (Type 4, T4).” Known clone detection techniques are text-based, token-based, tree-based, Program-Dependent Graph (PDG) based, or metric-based [8,9]. Existing clone detection techniques perform well when tested against types T1 and T2, but the detection rate is relatively low for types T3 and T4.
Furthermore, because these techniques are designed only for clone detection, they do not provide information about what category the detected clones fall into.

As actual code may contain various types of clone code, each technique’s performance must be confirmed to detect each clone type reliably. Furthermore, if clone detection is to lead to refactoring, all clone types must be identifiable. Clone-type information thus helps with decisions as about whether removing clones or process changes, such as modifying coding guidelines, should be made [10]. Finally, the ability to identify clones by type is an essential step towards automated refactoring systems. Though not all clones are suitable for refactoring as some clones may be beneficial [11], investigating clone types helps to visualize clone genealogies that need to be refactored and those that do not [12].

In his paper, we propose a two-pass technique for clone detection and type classification using a Tree-Based Convolutional Neural Network (TBCNN), which can capture structural features with short propagation path and achieves comparable performance to word-by-word attention models [13]. TBCNN is efficient for structural learning because its propagation length between input and output layers is fixed and independent of the tree size. We tested the performance of this technique against all four clone types and presented its accuracy at identifying each. Our technique does not classify the clone type directly with a code fragment’s Abstract Syntax Tree (AST). Rather, our technique detects clones on the first pass and then determines the clone type on the second pass. Both the first and second pass use the TBCNN to extract features from the code fragments’ ASTs to be compared [14]. To verify the performance of the two-pass technique, we used the BigCloneBench dataset [15], a well-known dataset widely used by researchers. The proposed technique’s clone detection performance was 96% (recall), 96% (precision), and 95% (F1 score). Our technique achieved 78% (recall and precision) and 75% (F1 score) on average for clone type classification.

The contributions of this paper are as follows:
• We present a novel approach for classifying the clone types, exceptionally well suited to T3 and T4 clones.
• Our technique uses the AST to capture characteristics that reflect common code patterns, which are both scalable and easily generated from code, which saves efforts for preprocessing process.
• We present a two-pass technique consisting of steps in clone detection and clone classification order to improve the performance of clone type classification.

In the next section, we describe related techniques for clone detection. Section 2 introduces AST and its vector representation as background knowledge. Section 3 presents our two-pass clone detection and type classification technique. The setup for the experiment and its results are presented briefly in Section 4 and our results are discussed in detail in Section 5. Section 6 offers the related work of clone detection. Finally, in Section 6 we conclude the paper and suggest directions for future research.

2. Background

A typical code clone detection approaches represent source code using different abstractions such as AST, Control Flow Graph (CFG), and PDG. In this section, we briefly discuss the AST with embedding technique used in this paper to transform AST into feature vectors.

2.1. Abstract Syntax Tree

An AST is a tree representation of a code fragment. Since this tree represents the actual structure of a code fragment, studies have used ASTs for code clone detection [16–18]. To extract the syntactic representation of a code fragment, code is converted into a set of tokens, and the list of tokens is turned into an AST. Figure 1a is a code fragment of a method signature part. Figure 1b is an AST for the method signature part. Each node of the AST tree structure has a type specifying what it is representing. For example, a type could be a “MethodDeclaration” representing a method definition or a “FormalParameter”
representing a parameter for a method declaration. There are two “FormalParameter” subtrees, each with a “ReferenceType” of “str”, that is, String.

A typical clone detection approach usually represents the code structure into different abstractions, such as AST, CFG, and PDG, and then compares code similarity between two code fragments. Graph-based techniques that generate CFGs or PDGs from code perform better in detecting code clones. However, PDG-based techniques are not scalable due to the complexity of graph isomorphism \[19\]. K. Chen et al. \[19\] improve accuracy and scalability simultaneously in detection clones, but they do not validate their technique for Type 4 clones. M. Gabel et al. \[17\] improve scalability by mapping the subgraphs in PDG back to AST forest and comparing syntactical feature vectors extracted from AST, but the results are imprecise due to the approximations \[20\]. In this paper, we use AST as our code representation because we find that AST can represent code patterns with significantly lower effort compared to CFG and PDG, and it is scalable to large amount of code.

2.2. Vector Representation of the AST

To facilitate datamining on code as well as the interpretation of the datamining results, syntax trees should be transformed into continuous vectors for representing the code. Vector representation of the code enables a much more comprehensive range of analysis. Figure 1c shows assigned code vectors for the AST of Figure 1b. Since machine learning algorithms take vectors as their inputs, we use vector embedding techniques \[21,22\] to transform AST into vectors. The code vectors capture properties of code fragments, such that similar code fragments have similar vectors.

![Figure 1](image_url). A sample AST and AST with generated feature vectors.
3. TBCNN-Based Two-Pass Clone Type Classification

Our technique consists of two steps performed in order. The first step is for detecting code clones, and the second step is for classifying their clone types. The clone type classification is performed only for the code pairs detected as clones at the first step. Thus, two classification steps are as follows:

- **The 1st-pass classification (clone detection)** determines whether a given piece of code is a clone. This pass detects structural features of a code fragment represented with an AST via TBCNN and applies max pooling to gather information over different parts of the tree. After pooling, the features that are aggregated into a fixed-size vector are fed to a fully-connected hidden layer before introducing the final output layer. For supervised classification, **softmax** is used as the output layer (see clone detection in Figure 2). The output layer consists of two neurons, clone or non-clone.

- **The 2nd-pass classification (clone type classification)** classifies the clone code. This pass targets only the code fragments classified as clones during the first pass. This pass again uses TBCNN and max pooling to detect and aggregate features of a clone code fragment. The features are fully connected to a hidden layer and then fed to the output layer (see clone type classification in Figure 2). In this step, the number of neurons of the output layer refers to the number of clone types.

In the remainder of Section 3 we describe the details of the AST embedding process associated with the code preprocessing step, as well as the clone detection and clone type classification steps.

**Figure 2.** Steps of the TBCNN-based two-pass clone detection and clone type classification.

### 3.1. Preprocessing

The preprocessing step consists of the two sub-steps denoted with the “Preprocessing” label in Figure 2. The first sub-step is the transformation of code fragments into ASTs to focus on the structure of code. The second sub-step requires the conversion of nodes in AST into vectors to produce AST represented by vectors. We used the ast2vec [22] to convert nodes in AST into vectors. The ast2vec encodes nodes in AST similar to how word2vec
encodes words for natural language processing, so AST’s nodes with similar functions have similar feature vectors.

3.2. Encoding Using TBCNN

The proposed technique applies TBCNN [10,13], which uses a tree-based convolution, max pooling, and a fixed-size neural layer during the 1st and 2nd classification. As with a CNN’s convolution layer, TBCNN has a tree-based convolution layer that explores the entire AST to generate a new tree that includes structural information associated with a code fragment. The nodes of the subtree are computed as one vector via the following formula:

\[ y = \tanh\left( \sum_{i=1}^{n} W_{conv,i} \cdot x_i + b_{conv} \right) \]  (1)

in which \( x_i \) is a vector representation of nodes in a tree, \( W_{conv,i} \) is the weight matrix of the node, and \( b_{conv} \) is the bias. AST performs poorly at establishing the size of \( W_{conv,i} \) because the number of children that a parent node can have is not fixed. To tackle this problem, TBCNN uses a “continuous binary tree”, a model that views any subtree as a “binary tree” regardless of its depth and degree. In this model, the weight matrix for a node \( W_{conv,i} \) is defined as a linear combination of the three weight matrices: \( W_{tconv}^l, W_{rconv}^r, \) and \( W_{lconv}^l \). \( W_{conv,i} \) is determined by multiplying each matrix by a factor containing the location information of the actual node (see [13] for details), as follows:

\[ W_{conv,i} = \eta_t^i W_{tconv}^l + \eta_r^i W_{rconv}^r + \eta_l^i W_{lconv}^l \]  (2)

TBCNN explores AST through such calculations, and upon reaching a leaf node, creates a hypothetical child node to perform the calculation by assigning “0” to the leaf node. Generating these virtual child nodes equalizes the form of the tree in the input and output of the tree-based convolution layer. The data is then down-sampled through the pooling layer. We used max pooling, which takes the maximum value of each dimension from features mined from the tree-based convolutional layers. With max pooling, features are pooled to one vector. Data is then aggregated through a pre-binding neural network and converted into a one-dimensional vector.

3.3. The 1st-Pass Classification: Clone Detection

Clone detection during the first pass classifies whether a given code fragment is a clone. Looking at the structure of the three code fragments in Figure 3, the code fragments in Figure 3a,b are similar in their functionality while those of Figure 3c perform a different function. The first two code fragments are logics for obtaining the factorial value of a given integer, while the last code fragment is a logic that changes the position of two elements in an integer array. Code similarity is low in all three cases because no structurally identical syntax is found. Yet, the code pair of Figure 3a,b should be classified as a T4 clone as they perform the same function (although their structural similarities are low). The remaining two code pairs are not clones, due to their low structural similarities and performance of different functions.
int factorial(int n) {
    int i, j = 1;
    for (i = 1; i <= n; i++)
        j = j*i;
    return j;
}

(a) Factorial program using iterative method

int factorial(int n) {
    if (n = 0) return 1;
    else return n * factorial(n-1);
}

(b) Factorial program using recursive method

void sort(int[] arr) {
    int i, max_index = 0;
    for (i = 0; i<arr.length; i++)
        if (arr[i] > arr[max_index]) swap(arr, i, max_index);
}

(c) Sort program using a for loop, if statement and swap method

Figure 3. Example code fragments of a T4 type and non-clone type.

Our technique performs this clone detection prior to clone classification because non-clone code pairs and their corresponding clone types are sometimes categorized as clones. Our experimental results confirmed that the accuracy of our clone type classification results varied significantly in accordance with whether this clone detection pass was performed. The main components of our technique include the vector value of AST obtained from AST embedding, tree-based convolution, max pooling, a fully-connected neural network, and an output layer. The output node value (the probability that the assessed code is a clone of another code fragment) was between 0 and 1.

3.4. The 2nd-Pass Classification: Clone Type Classification

In our system, clone-type classification is performed only on code fragments detected as clones, with this pass using the same TBCNN model used during the 1st pass. However, unlike the clone detection model’s output layer with two output nodes (clone or non-clone), the output of the classification model consists of four output nodes corresponding to the four possible clone types (lower right-hand of Figure 2). Each output node has a value between 0 and 1, with the specific value indicating the probability that the assessed code is of a particular clone type.

4. Experimental Evaluation

We conducted experiments to evaluate the performance of our technique. This section describes the dataset used for evaluation and the parameter values set in our training and experimentation models, as well as the experimental results.

4.1. Dataset

We used BigCloneBench for training and testing our model. BigCloneBench [15] is a real-world benchmark that contains over 6,000,000 tagged clone pairs across 43 functionalities, as well as 260,000 false clone pairs. BigCloneBench specifies a clone pair by triple, including a pair of similar code fragments and their clone-type. It contains both intra-project and inter-project clones of all four clone-types. Depending on the similarity ratio, these clones and clone-types have all been manually evaluated. BigCloneBench divides T3 and T4 clones into four categories: Very-Strongly Type 3 (VST3), Strongly Type 3 (ST3), Moderately Type 3 (MT3), and Weakly Type 3+4 (WT3/4).
BigCloneBench’s T1 clone refers to syntactically identical code fragments, excluding differences in white space, layout, and comments identified after applying T1 normalization. T2 refers to syntactically identical code fragments, excluding differences in identifier names and literal values, in addition to Type-1 clone differences identified after applying T2 normalization. T1 normalization refers to annotation removal and pretty-printing, while T2 normalization includes the addition or modification of identifier names and literal values, in addition to T1 normalization. BigCloneBench’s ST3, MT3, and WT3/4 clones were classified by measuring similarity as the minimum ratio of lines or token code fragments shared after the T1 and T2 normalizations were applied. As there is no consensus on classification criteria of a T3 clone or T4 clone, BigCloneBench defines the range of similarity of each clone-type to separate clones because of these types. ST3 clones are those clones with a similarity in the range of 70% (inclusive) to 90% (exclusive), and most syntactic clone detection tools use this criterion. MT3 clones refer to clones with a syntactic similarity of between 50 and 70%. Conventional syntactic clone detection tools classify MT3 clones as either a T3 clone or T4 clone. The final WT3/4 are those clones with a syntactic similarity of less than 50%, and most clone detection tools do not detect this type.

To compare the performance of our method with existing tools, we used the BigCloneBench’s dataset (Table 1). The selected dataset consists of 59,618 individual code fragments, paired in two to form 97,535 code pairs, including 20,000 false clone pairs. Clone pairs consist of 15,555 T1 clones, 3663 T2 clones, 18,317 ST3 clones, 20,000 MT3 clones, and 20,000 WT3/4 clones. However, the number of clone pairs of each clone-type is not balanced. Specifically, the number of T2 clone pairs is significantly less than other types. Because imbalanced datasets produce biased training results in classes with multiple instances, we conducted sampling from the BigCloneBench dataset to obtain an appropriate dataset for training and testing our technique.

### Table 1. BigCloneBench dataset.

| Dataset Statistics                  | Value |
|-------------------------------------|-------|
| Number of code fragments            | 59,618|
| Number of code pairs                | 97,535|
| Number of false clone pairs         | 20,000|
| Number of T1 clones                 | 15,555|
| Number of T2 clones                 | 3663  |
| Number of ST3 clones                | 18,317|
| Number of MT3 clones                | 20,000|
| Number of WT3/4 clones              | 20,000|

### 4.2. Implementation

#### 4.2.1. Training and Test Datasets

To cope with BigCloneBench’s class imbalance problem, we used an undersampling technique that randomly removes class instances from a class with many instances to balance the number of instances. We performed undersampling based on the number of T2 clone pairs to build a training and test dataset. Especially for the clone detector of the first pass, the total number of clone pairs and the number of false clone pairs were adjusted to be similar. Table 2 shows a training and test dataset that resulted after your undersampling. We performed the undersampling twice, once to generate a training dataset and again to generate a test dataset.
Table 2. Training and test dataset used.

| Clone Types | True Clone Pairs | False Clone Pairs |
|-------------|-----------------|------------------|
| Dataset     | T1   | T2  | ST3 | MT3 | WT3/4 | T1   | T2  | ST3 | MT3 | WT3/4 |
| Training    | 1099 | 1099| 1099| 1098| 1099  | 5492 |
| Test        | 2564 | 2564| 2564| 2563| 2564  | 12,823 |
| Total       | 3663 | 3663| 3663| 3661| 3663  | 18,315 |

4.2.2. Parameter Settings

We tuned the parameters and chose the set that achieved the highest F1 score on the dataset. Parameter tuning was applied to the second pass’s classification model. To determine the optimal parameters, parameter tunings proceeded in order of (1) AST embedding, (2) fully-connected neural network, and (3) both AST embedding and fully-connected neural network. We identified the optimal parameter settings for our technique (Table 3). We set the clone classification model parameters higher for the second pass than the first, considering the difficulty of clone-type classification performed during the second pass.

Table 3. Parameter settings for each classification pass.

| Classification Pass | Parameters |
|---------------------|------------|
| 1st pass TBCNN-based encoding | Convolution: 1, Pooling layer: 1, Output size: 50 |
| Fully-connected NN | Input size: 100, Output size: 2, Hidden layer: 2, First hidden layer size: 100, Second hidden layer size: 150, Activation function: LRelu |
| Learning rate: 0.001, Epoch: 100, Batch size: 5 |
| 2nd pass TBCNN-based encoding | Convolution: 4, Pooling layer: 1, Output size: 200 |
| Fully-connected NN | Input size: 400, Output size: 5, Hidden layer: 3, First hidden layer size: 400, Second hidden layer size: 400, Third hidden layer size: 300, Activation function: LRelu |
| Learning rate: 0.001, Epoch: 200, Batch size: 5 |

4.3. Results

Table 4 summarizes the recall and precision of our technique on the BigCloneBench dataset. Recall refers to ratio of clone pairs or clone-types within a dataset that our technique detect or classify, while precision is the ratio of clone pairs or their types reported by our technique that are true clone pairs or true clone-types (and not false positives). Overall, our technique achieved 95% precision in the first pass for clone detection. More specifically, our technique achieved 94% recall and 96% precision for true clones, and 97% recall and 95% precision for false clones. In the second pass for clone type classification, our technique achieved 78% recall and precision on average. Our technique achieved high precision (94%) for T1 clones and low precision (67%) for MT3 clones, while the highest recall is for T2 clones (93%) and the lowest is for MT3 clones (58%).
Table 4. Results on clone detection of the 1st pass and clone type classification of the 2nd pass.

| Clone Detection Result of the 1st Pass | Clone Type Classification Result of the 2nd Pass |
|---------------------------------------|-----------------------------------------------|
| Clone Pairs                         | Recall | Precision | F1 Score | Clone Type | Recall | Precision | F1 Score |
| true clone                          | 0.94   | 0.96       | 0.95     | T1        | 0.75   | 0.94       | 0.84     |
| false clone                         | 0.97   | 0.95       | 0.96     | T2        | 0.93   | 0.77       | 0.84     |
|                                     | -      | -          | -        | ST3       | 0.71   | 0.69       | 0.70     |
|                                     | -      | -          | -        | MT3       | 0.58   | 0.67       | 0.62     |
|                                     | -      | -          | -        | WT3/4     | 0.81   | 0.74       | 0.77     |

In our second experiment, we classified clone-types with and without the first pass. We built a model with the same architecture of our technique, excluding separated passes. Table 5 shows our comparative experimental results. Overall, the proposed two-pass technique achieved better recall and precision than the model without the first pass. With the two-pass technique, classification of T2 clones is significantly improved (see F1 score of Table 5).

Table 5. Comparing clone classification achievements with/without the 1st pass.

| Clone Type | Recall Without the 1st Pass | Recall With the 1st Pass | Precision Without the 1st Pass | Precision With the 1st Pass | F1 Score Without the 1st Pass | F1 Score With the 1st Pass |
|------------|-----------------------------|--------------------------|-------------------------------|----------------------------|-------------------------------|----------------------------|
| False clone| 0.73                        | 0.97                     | 0.78                          | 0.77                       | 0.76                          | 0.86                       |
| T1         | 0.76                        | 0.75                     | 0.90                          | 0.95                       | 0.82                          | 0.84                       |
| T2         | 0.89                        | 0.93                     | 0.77                          | 0.78                       | 0.73                          | 0.85                       |
| ST3        | 0.68                        | 0.73                     | 0.67                          | 0.72                       | 0.67                          | 0.72                       |
| MT3        | 0.54                        | 0.57                     | 0.63                          | 0.71                       | 0.58                          | 0.63                       |
| WT3/4      | 0.77                        | 0.72                     | 0.64                          | 0.77                       | 0.70                          | 0.74                       |

Table 6 compares the recall and precision of our technique against other approaches. Overall, our technique achieved 96% recall and precision for clone detection, a result significantly better than those of the other approaches, except DeepSim, against which our technique performed slightly lower worse. These results demonstrate that our technique can detect clones at a similar level and rate as DeepSim, despite its much simpler preprocessing. In the case of clone classification, our technique achieved 78% recall and precision.

Table 6. Comparison results of existing tools and our technique.

| Tools           | Clone Detection | Clone Type Classification |
|-----------------|-----------------|---------------------------|
|                | Recall | Precision | F1 Score | Recall | Precision | F1 Score | Recall | Precision | F1 Score |
| DECKARD [16]    | 0.02   | 0.93       | 0.3      | -      | -         | -       |
| RtvNN [23]      | 0.01   | 0.95       | 0.01     | -      | -         | -       |
| CDLH [24]       | 0.74   | 0.92       | 0.82     | -      | -         | -       |
| DeepSim [20]    | 0.98   | 0.97       | 0.98     | -      | -         | -       |
| Our Technique   | 0.96   | 0.96       | 0.96     | 0.78   | 0.78      | 0.75    |

In our final test, we investigated misclassified clones (Table 7). Many WT3/4 clones were incorrectly classified as false clones. The reason for this result was that the similarity ratio of WT3/4 clones was less than 50%. Clones tend to be incorrectly classified as adjacent types, i.e., ST3 clones are misclassified into T2 or MT3 clones.
Table 7. Distribution of misclassified clone types.

| Clone Type | False Clone | T1 | T2 | ST3 | MT3 | WT3/4 |
|------------|-------------|----|----|-----|-----|-------|
| false clone| -           | 2  | 1  | 138 | 165 | 330   |
| T1         | 7           | -  | 2903 | 160 | 13  | 11    |
| T2         | 1           | 61 | -  | 160 | 13  | 11    |
| ST3        | 922         | 318| 1283| -   | 2205| 407   |
| MT3        | 1687        | 69 | 68 | 3452| -   | 3495  |
| WT3/4      | 2962        | 71 | 15 | 465 | 1841| -     |

5. Discussion

We manually converted clone pairs to ASTs to analyze the reason that clones are misclassified as adjacent clone-types. We identified a common feature in clones classified as T2, ST3, and MT3 clones. For clones classified as T2, more than 95% of original code fragments were reused by cloned code fragments without any modifications. Clones classified as ST3 included several identical subtrees at the same level. For clones classified as MT3, there was a difference in the level at which the same subtree appeared, unlike clone pairs classified as ST3 clones. The first three subsections discuss each of the misclassified clones, and the final subsection discusses the threats that can affect the validity of the experiment results.

5.1. Clone Pairs Misclassified into T2 Clones

As reported in Table 7, most clone pairs misclassified into T2 clones were either T1 or ST3 clones. As for T1 clones misclassified as T2 clones, their ASTs were entirely identical; this is because a T2 clone modifies only identifiers or literals of an original code fragment. In the case of the ST3 clones misclassified into T2 clones, more than 95% of the original code fragment was completely identical. The ASTs of these clones resulted in some nodes being rearranged or a subtree increasing by one level due to the newly added code. Our technique tended to classify clone pairs into T2 clones when the difference between the two ASTs was slight, i.e., when the similarity between the codes was high.

Figure 4 presents an example clone pair that best illustrates one of these cases. The clone pair in Figure 4a was classified as an ST3 clone on BigCloneBench, but classified as a T2 clone by our technique. This clone pair is similar, but each code fragment uses different object names, with the code on the left-side using object names such as srcChannel and dstChannel, while the code on the right uses in and out instead (see the fourth line in italic). The order of arguments of a caller method is also different from that of on the left-side code, i.e., “srcChannel, 0, srcChannel.size()” and “0, in.size(), out”, respectively. The two caller methods are similar but have a semantic difference, making this clone pair an ST3 clone. In AST, this semantic difference is represented by two identical subtrees that differ only in their order (see the boxed subtrees of Figure 4b). Our technique detects this difference as small changes, so it was classified as a T2 clone.
1. public static void copyFile(File src, File dst) throws IOException {
2.     FileChannel srcChannel = new FileInputStream(src).getChannel();
3.     FileChannel dstChannel = new FileInputStream(dst).getChannel();
4.     dstChannel.transferFrom(srcChannel, 0, srcChannel.size());
5.     srcChannel.close();
6.     dstChannel.close();
7. }

(a) A clone pair with different order of arguments in a caller method

(b) ASTs of the clone pair in (a)

Figure 4. Example of ST3 clone classified as T2 clone.

The clone pair example in Figure 5a was classified as an ST3 clone by BigCloneBench but as a T2 clone by our technique. The clone pair of Figure 5a has method signature difference (see lines in italic). According to its ASTs, the signature difference between the two code fragments is expressed as the addition or deletion of subtrees such as the ‘set’ subtree of Figure 5b. Our technique did not detect this slight distinction.
Changes to a small number of nodes in AST have a small impact on type classification because they cause slight changes in the feature vectors used as inputs for training and testing in the model. Clones of the type in Figures 4 and 5 are close to T2 clones because they are highly similar codes that are functionally comparable. In particular, the code pair of Figure 5 can be refactored in the same way as T1 and T2 clones because all code lines, except for the method signature, are identical.

5.2. Clone Pairs Misclassified into ST3 Clones

The total number of MT3 clones on BigCloneBench was 20,000, among which 8771 MT3 clones were misclassified (as described in Tables 1 and 7). Almost 78% of MT3 clones were misclassified as ST3 clones. Examination of a sample code pair (Figure 6a) reveals that the difference between the two code fragments is that lines 2 and 3 of the right-side code are newly added, and line 5 of the left-side code has been changed to line 7 of its right-side code (see lines in italic). The ASTs of this clone pair are identical except for the boxed portion of Figure 6b, a piece of the AST for the right-side code fragment. In most cases where MT3 clones are misclassified as ST3 clones, 70–80% of the AST nodes had identical structures. However, the classification performed using the proposed technique can be regarded as reasonable, recalling that the ST3 clones (similarity of 70–90%) and MT3 clones (similarity of 50–70%) in BigCloneBench are distinguished only by their degree of similarity. Had we not divided T3 clones into ST3 and MT3 categories, our technique would be regarded as performing well on T3 clones.
5.3. Clone Pairs Misclassified into MT3 Clones

The clones misclassified as MT3 clones were ST3 and WT3/4 clones. Analysis results of these clones revealed that the ASTs had identical subtrees but that some of them were located at different levels. A sample clone pair in Figure 7a is a clone classified as an ST3 clone in BigCloneBench, but was classified as an MT3 clone in our technique. This clone is identical in its statements, the only difference being in the use of the ‘try-catch’ statement, i.e., line 4–9. When we investigated the ASTs of this clone pair, we found that subtrees located at level 1 in the AST of the left-side code fragment were located at level 2 in the AST of the right-side code fragment due to the ‘TryStatement’ node. We tested whether the number of statements within a try-catch statement affected clone-type classification, finding that only the level of subtrees in the AST had an effect, regardless of the number of statements with a try-catch statement. This demonstrates that the subtree’s level is an influential feature that distinguishes MT3 clones from others in our clone-type classification technique.

![Figure 6. Example of MT3 clone classified as ST3 clone.](image)
5.4. Threats to Validity

First, we did not apply cross-validation techniques to train and evaluate our model. Our model thus might fit to the training data and does not accurately work for the actual data. Second, we performed undersampling to overcome the imbalance problem, but we did not compare the results with those of oversampling. We did not perform oversampling because we could not guarantee that we could apply the same classification criteria as originally used in the BigCloneBench dataset. However, this is a threat to the internal validity of the results.

6. Related Work

There are, at present, six types of clone detection techniques: text-based, token-based, tree-based, PDG-based, matrix-based, and machine learning-based. A text-based clone detection technique treats the code as text and detects clones through direct comparison of code fragments. For example, Dup, a text-based and line-based clone detection tool, uses a parameterized matching algorithm [3], and J. Johnson [25] uses a Karp-Rabin string matching approach. These text-based techniques can find clones simply and quickly but are not capable of detecting clones that are not completely or near identical to the source code.

Token-based clone detection transforms the entire code into a sequence of tokens and then compares the sequences. CCFinder [26], for example, lexes, parses, and transforms identifiers related to variables, constants, and user-defined types into a sequence of tokens. CCFinder detects clones, even if only identifiers have been changed. CCFinder cannot detect clones, however, where the order of sentences has been modified or new code has...
been inserted. To overcome this limitation, a different token-based clone detection tool, CP-Miner [7], uses frequent pattern mining. CP-Miner can detect one or two code lines that have been inserted, deleted, or modified, but it is still unable to detect cloned codes where more extensive code modifications have been made.

Tree-based clone detection techniques seek to overcome the limitations of text-based techniques by converting code to AST and then using tree matching algorithms to detect clones. CloneDR [27] transforms AST to hash values via hash functions and then detects clones by comparing subtrees with the same hash value. V. Wahler et al. [28] have proposed a technique for clone detection using frequent pattern mining techniques based on AST expressed in XML. W. Evans and C. Fraser [29] have suggested a technique to detect clones using AST’s subtree where a particular pattern appears. While researchers have worked hard to refine these tree-based clone detection techniques, clone detection remains complicated in situations where two code fragments have different orders.

An alternative technique for detecting clones, including differently ordered statements, newly inserted statements, or removed statements, are PDG-based techniques. PDG-based techniques detect clones by matching PDGs that contain data flow and control flow information of a code fragment. The most well-known PDG-based clone detection technique is PDG-DUP, proposed by R. Krinke and S. Horwitz [30]. PDG-DUP finds identical PDG subgraphs using program slicing. J. Krinke [31] uses k-length patch matching to identify maximally similar subgraphs in PDGs. While PDG-based clone detection techniques overcome the limitations of the text-based, token-based, and tree-based techniques, this technique does not scale well to large-scale systems, as matching PDG subgraphs takes a while.

Metric-based clone detection techniques refer to various metric values derived from code for clone detection. Metrics are typically calculated from classes, techniques, and statements using fingerprinting algorithms. J. Mayrand et al. [32] transformed each function of source code to Intermediate Presentation Language (IRL) to calculate metrics values from the names, layout, expression, and control flow of functions. A pair of functions that has similar metric values is detected as a clone. Another metric-based clone detection technique proposed by K. Kontogiannis et al. [4] computes metric values that capture the data and control flow properties of source code. The technique annotates metric values to corresponding nodes of AST constructed from source code and detects clones by comparing the distances between annotated metric values of code fragment pairs using dynamic programming approach. Since metric-based techniques do not utilize the code itself, this method may be less precise than others, as two different codes with similar metric values may be identified as clones.

Recently, researchers have focused on extracting code features and detecting clones using machine learning [18,20,23,24,33–36]. CCLearner [33] uses a token analysis technique as well as a deep learning technique, in which tokens are used to generate inputs for a deep learning model, which then determines whether the given inputs are clones. DeepSim [20] encodes both control flows and data flows into a semantic matrix and uses a deep learning method to detect clones based on the semantic matrix. N. Bui et al. [34] is an AST-based approach and used a bilateral tree-based convolutional neural network for automated program classification. RtvNN [23], CDLH [24], and D. Perez et al. [35] are AST-based approaches that transform AST into feature vectors to use deep learning models. Whereas RtvNN uses recursive neural networks to detect both textual and functional similarity, CDLH and D. Perez et al. apply LSTM on ASTs to detect functional clones. C. Fang et al. [18] use both AST and CFG to extract syntactic and semantic information of code. They use a DNN classifier model by using fused syntactic and semantic feature vectors as input. CCLearner, RtvNN, and CDLH perform tolerably well in experiments using BigCloneBench but perform particularly poorly at detecting T3- and T4-type clones. DeepSim performs well at detecting all types of clones, but its preprocessing procedure is unwieldy. The experiments of D. Perez et al. are about clone detection between the two different programming languages. A. Sheneamer et al. [36] use both AST and PDG to
detect semantic information of code, and they compare T3- and T4-type clone detection performance of machine learning algorithms such as Rotation Forest, Random Forest, and Xgboost. They conclude that Xgboost is an excellent classification algorithm for the detection of T3- and T4-type clones.

7. Conclusions and Future Work

This paper proposed a TBCNN-based two-pass clone-type classification technique that detects clones and then classifies them by clone-type. We confirmed that the proposed technique performs well in terms of recall and precision on the first clone detection pass, which used only simple preprocessing. Experimentation revealed that the second pass classified the clone-types with 78% precision and recall.

This work makes three contributions to the field of clone detection and classification by using machine learning: (1) The proposed technique performs clone detection with high accuracy using only a straightforward preprocessing process. DeepSim [20], which performs similarly well to our technique’s clone detection phase, encodes data and control flow information of a program into a compact semantic feature matrix to capture a code’s semantic information, which made DeepSim unwieldy. Our technique, however, used only ASTs, which can be generated easily from code. Moreover, our technique encodes each AST’s nodes into a vector via AST2vec, which is a much more simple and direct process than that applied by DeepSim. (2) Unlike conventional machine learning techniques, our technique detects and classifies clones with relatively high accuracy. Clone-type information can be used to provide automated refactoring support in development environments. Specifically, T1 and T2 clones can be easily modified by consolidating clones into a single code and invoking the code, if necessary, without further analysis. As for T3 and T4 clones, developers can make decision whether they refactor the clones or not by considering the context in which the clones appear. (3) Our two-pass technique can be divided into two phases, each of which can be used alone or together as needed. Since our two-pass clone classification technique performs the first classification and the second classification sequentially, clone detection results can be obtained without associated classification data.

Our technique does have some deficiencies, specifically its lower recall of MT3 clones than other clone-types, and the fact that it does not consider small changes in AST as features for clone type classification. We can improve the current code vector representation method in a way that captures the subtle differences between code fragments. Alternatively, additional code information such as control flows and data flows can be considered to address the issue that AST provides only structural information. Our future work includes these directions.

Author Contributions: Conceptualization, Y.-B.J.; methodology, Y.-B.J. and J.L.; validation, Y.-B.J.; investigation, Y.-B.J.; writing—original draft preparation, Y.-B.J. and J.L.; writing—review and editing, J.L.; supervision, C.-J.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by a grant from the National Research Foundation of Korea funded by the Korean government (MSIT) (NRF-2020R1F1A1071650).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: The authors would like to thank the Writing Center at Jeonbuk National University for its skilled proofreading service.

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

1. 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]

2. Roy, C.K.; Cordy, J.R.; Koschke, R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.* 2009, 74, 470–495. [CrossRef]

3. Baker, B. On finding duplication and near-duplication in large software system. In Proceedings of the 2nd IEEE Working Conference on Reverse Engineering, Toronto, ON, Canada, 14–16 July 1995; pp. 86–95.

4. Kontogiannis, K.; Demori, R.; Merlo, E.; Galler, M.; Bernstein, M. Pattern matching for clone and concept detection. *Autom. Softw. Eng.* 1996, 3, 76–108. [CrossRef]

5. Lague, B.; Pronlx, D.; Mayrand, J.; Merlo, E.; Hudopohl, J. Assessing the benefits of incorporating function clone detection in a development process. In Proceedings of the 1st IEEE International Conference on Software Maintenance, Bari, Italy, 1–3 October 1997; pp. 314–321.

6. Chou, A.; Yang, J.; Chelf, B.; Hallem, S.; Engler, D.R. An empirical study of operating system errors. In Proceedings of the 18th ACM Symposium on Operating Systems Principles, Banff, AB, Canada, 21–24 October 2001; pp. 73–88.

7. Li, Z.; Lu, S.; Myagmar, S.; Zhou, Y. CP-Miner: Finding copy-paste and related bugs in operating system code. *IEEE Trans. Softw. Eng.* 2006, 32, 289–302. [CrossRef]

8. Roy, G.K.; Cordy, J.R. *A Survey on Software Clone Detection Research. Technical Report No. 2007-541*; School of Computing, Queen’s University at Kingston: Kingston, ON, Canada, 2007.

9. Gautam, P.; Saini, H. Various code clone detection techniques and tools: A comprehensive survey. In Proceedings of the International Conference on Smart Trends for Information Technology and Computer Communications, Jaipur, India, 6–7 August 2016; pp. 665–667.

10. Staron, M.; Meding, W.; Eriksson, P.; Nilsson, J.; Lövgren, N.; Österström, P. Classifying obstructive and nonobstructive code clones of type I using simplified classification scheme: A case study. *Adv. Softw. Eng.* 2015, 1–18. [CrossRef]

11. Murphy, G. An empirical study of code clone genealogies. In Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, Lisbon, Portugal, 5–9 September 2005; pp. 187–196.

12. Rattan, D.; Bhatia, R.; Singh, S. Software clone detection: A systematic review. *Inf. Softw. Technol.* 2003, 55, 1165–1199. [CrossRef]

13. Mou, L.; Jin, Z. *Tree-Based Convolutional Neural Networks Principles and Applications*; Springer: Singapore, 2018.

14. Mou, L.; Li, G.; Zhang, L.; Wang, T.; Jin, Z. Convolution 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.

15. Sväjlenko, J.; Roy, C.K. Evaluating clone detection tools with BigCloneBench. In Proceedings of the IEEE International Conference on Software Maintenance and Evolution, Bremen, Germany, 29 September–1 October 2015; pp. 131–140.

16. Jiang, L.; Misherghi, G.; Su, Z.; Glondu, S. DECKARD: Scalable and accurate tree-based detection of code clones. In Proceedings of the 29th International Conference on Software Engineering, Minneapolis, MN, USA, 20–26 May 2007; pp. 96–105.

17. Gabel, M.; Jiang, L.; Su, Z. Scalable detection of semantic clones. In Proceedings of the 30th International Conference on Software Engineering, Leipzig, Germany, 10–18 May 2008; pp. 321–330.

18. 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 Event. New York, NY, USA, 18–22 July 2020; pp. 516–527.

19. Chen, K.; Liu, P.; Zhang, Y. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, 31 May–7 June 2014; pp. 175–186.

20. 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; pp. 141–151.

21. Alon, U.; Zilberstein, M.; Levy, O.; Yahav, E. code2vec: Learning distributed representations of code. In Proceedings of the ACM on Programming Languages; 2019; 3, pp. 1–29.

22. Paaßen, B.; McBroom, J.; Jeffries, B.; Koprinska, I.; Yacef, K. ast2vec: Utilizing recursive neural encodings of python programs. In Proceedings of the Educational Data Mining, Paris, France, 29 June–2 July 2021; pp. 1–34.

23. White, M.; Tufano, M.; Vendome, C.; Poshivanyk, 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; pp. 87–98.

24. Wei, H.-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, VIC, Australia, 19–25 August 2017; pp. 3034–3040.

25. Johnson, J. Substring matching for clone detection and change tracking. In Proceedings of the 10th International Conference on Software Maintenance, Victoria, BC, Canada, 19–23 September 1994; pp. 120–126.

26. Kamiya, T.; Kusumoto, S.; Inoue, K. CCFinder: A multilingual token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.* 2002, 28, 654–670. [CrossRef]
27. Baxter, I.; Yahin, A.; Moura, L.; Anna, M.S. Clone detection using abstract syntax trees. In Proceedings of the 14th International Conference on Software Maintenance, Bethesda, MD, USA, 20 November 1998; pp. 368–377.

28. Wahler, V.; Seipel, D.; von Gudenberg, J.W.; Fischer, G. Clone detection in source code by frequent itemset techniques. In Proceedings of the 4th IEEE International Workshop Source Code Analysis and Manipulation, Chicago, IL, USA, 16 September 2004; pp. 128–135.

29. Evans, W.; Fraser, C. Clone detection via structural abstraction. Softw. Qual. J. 2009, 17, 309–330. [CrossRef]

30. Komondoor, R.; Horwitz, S. Using slicing to identify duplication in source code. In Proceedings of the 8th International Symposium on Static Analysis, Paris, France, 16–18 July 2001; pp. 40–56.

31. Krinke, J. Identifying similar code with program dependence graphs. In Proceedings of the 8th Working Conference on Reverse Engineering, Stuttgart, Germany, 2–5 October 2001; pp. 301–309.

32. Mayrand, J.; Leblanc, C.; Merlo, E. Experiment on the automatic detection of function clones in a software system using metrics. In Proceedings of the 12th International Conference on Software Maintenance, Monterey, CA, USA, 4–8 November 1996; pp. 244–253.

33. Li, L.; Feng, H.; Zhuang, W.; Meng, N.; Tyder, B. CC Learner: A deep learning-based clone detection approach. In Proceedings of the IEEE International Conference on Software Maintenance and Evolution, Shanghai, China, 17–24 September 2017; pp. 249–260.

34. Bui, N.D.Q.; Jiang, L.; Yu, Y. Cross-language learning for program classification using bilateral tree-based convolution neural networks. In Proceedings of the 32nd AAAI Conference on Artificial Intelligence Workshop on NLP for Software Engineering, New Orleans, LA, USA, 2–7 February 2018; pp. 758–761.

35. Perez, D.; Chiba, S. Cross-language clone detection by learning over abstract syntax trees. In Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories, Montreal Quebec, QC, Canada, 26–27 May 2019; pp. 518–528.

36. Sheneamer, A.; Kalita, J. Semantic clone detection using machine learning. In Proceedings of the 15th IEEE International Conference on Machine Learning and Applications, Anaheim, CA, USA, 18–20 December 2016; pp. 1024–1028.