Assessing and Improving an Evaluation Dataset for Detecting Semantic Code Clones via Deep Learning

HAO YU, School of Software and Microelectronics, Peking University, Beijing, China
XING HU, School of Software Technology, Zhejiang University, Ningbo, China
GE LI, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, China
YING LI, National Research Center of Software Engineering, Peking University, Beijing, China
QIANXIANG WANG, Huawei Technologies Co., Ltd., China
TAO XIE, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, China

In recent years, applying deep learning to detect semantic code clones has received substantial attention from the research community. Accordingly, various evaluation benchmark datasets, with the most popular one as BigCloneBench, are constructed and selected as benchmarks to assess and compare different deep learning models for detecting semantic clones. However, there is no study to investigate whether an evaluation benchmark dataset such as BigCloneBench is properly used to evaluate models for detecting semantic code clones. In this article, we present an experimental study to show that BigCloneBench typically includes semantic clone pairs that use the same identifier names, which however are not used in non-semantic-clone pairs. Subsequently, we propose an undesirable-by-design Linear-Model that considers only which identifiers appear in a code fragment; this model can achieve high effectiveness for detecting semantic clones when evaluated on BigCloneBench, even comparable to state-of-the-art deep learning models recently proposed for detecting semantic clones. To alleviate these issues, we abstract a subset of the identifier names (including type, variable, and method names) in BigCloneBench to result in AbsBigCloneBench and use AbsBigCloneBench to better assess the effectiveness of deep learning models on the task of detecting semantic clones.

CCS Concepts: • **Software and its engineering → Maintaining software;**

Additional Key Words and Phrases: Code clone detection, deep learning, dataset collection

This work was supported by the Key-Area Research and Development Program of Guangdong Province (No. 2020B010164003) and National Natural Science Foundation of China (Grant No. 62161146003). This work was also supported by the Tencent Foundation or XPLORER PRIZE.

Authors' addresses: H. Yu, School of Software and Microelectronics, Peking University, No.5 Yiheyuan Road Haidian District, Beijing, China 10087; email: yh0315@pku.edu.cn; X. Hu, School of Software Technology, Zhejiang University, No. 1689 Jiangnan Road, Ningbo, Zhejiang, China, 315048; email: xinghu@zju.edu.cn; G. Li (corresponding author) and T. Xie (corresponding author), Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, No. 5 Yiheyuan Road Haidian District, Beijing, China 10087; emails: lige@pku.edu.cn, taoxie@pku.edu.cn; Y. Li (corresponding author), National Research Center of Software Engineering, Peking University, No. 5 Yiheyuan Road Haidian District, Beijing, China 10087; email: li.ying@pku.edu.cn; Q. Wang, Huawei Technologies Co., Ltd., Building 1 and 4, No. 18, Muhe Road, Haidian District, Beijing, China, 100094; email: wangqianxiang@huawei.com.

1049-331X/2022/07-ART62 $15.00
https://doi.org/10.1145/3502852

ACM Reference format:
Hao Yu, Xing Hu, Ge Li, Ying Li, Qianxiang Wang, and Tao Xie. 2022. Assessing and Improving an Evaluation Dataset for Detecting Semantic Code Clones via Deep Learning. *ACM Trans. Softw. Eng. Methodol.* 31, 4, Article 62 (July 2022), 25 pages.
https://doi.org/10.1145/3502852

1 INTRODUCTION

Code clones [32] (in short as clones in the rest of this article) are similar code snippets that share the same semantics but may differ syntactically to various degrees. There is a common agreement that clones should be detected and managed [20, 34] for three main reasons. First, clones unnecessarily increase system size. As a system increases in size, more software maintenance efforts are needed. Second, changes to a code segment, such as fault fixing, need to be made to its clones as well, thereby increasing maintenance efforts. Also, if changes are performed inconsistently, faults could be introduced. Third, duplicating a code snippet that contains faults leads to fault propagation.

To detect and manage clones, researchers have established a common taxonomy to group clones into multiple types [2, 32], which encompass *semantic clones*, the most difficult-to-detect ones. Type I-III clones are clone pairs that differ at token and statement levels. Type-IV clones are code snippets with similar functionalities but with different implementations. To clarify the differences between Type-III and Type-IV clones, previous work [39] divides these two types into the following four categories based on their syntactical similarity (sorted from the easiest to most difficult to detect): Very-Strong Type-III, Strong Type-III, Moderately Type-III, and Weak Type-III/Type-IV. The two most difficult-to-detect categories of clones, i.e., Moderately Type-III and Weak Type-III/Type-IV, are referred to as semantic clones [2, 32]. Semantic clones are difficult to detect because they are quite different in implementations, and are not amenable for detection based on the lexical and structural information [45, 48].

Since the emergence of clones as a research field, various traditional approaches [15, 16, 33, 36] have focused on detecting and analyzing Type-I to Type-III clones but have had limited success with semantic clones (i.e., Moderately Type-III and Weak Type-III/Type-IV clones). Without prior knowledge, these approaches cannot identify semantic clones being dissimilar in lexical/syntactical-level implementations (e.g., bubble sort and quick sort) without considering functional behaviors of code snippets. For example, SourcererCC [36] treats the source code under analysis as tokens and compares subsequences to detect clones. SourcererCC is a typical lexical-based approach that considers the similarity only in the lexical level of code snippets and ignores the structural information. Deckard [15] uses only the structural information without considering lexical information of code snippets.

Based on a large labeled dataset such as BigCloneBench [39] (one of the most popular benchmarks), deep learning approaches [45, 46, 48, 49] have been proposed to detect semantic clones in recent years. In the rest of this article, we refer a deep learning architecture before training as a deep learning approach, and refer a deep learning model after training as a deep learning model. Existing research has taken great efforts on proposing complicated deep learning approaches [44, 45, 48, 49] to improve detection effectiveness with respect to evaluation metrics (e.g., precision and recall) on BigCloneBench. These approaches usually split the dataset into the training and validation/test sets. Their experimental results show that deep learning approaches can effectively detect semantic clones by learning features from big data without manually extracting features. Deep learning approaches perform well on detecting semantic clones because the deep learning approaches can learn the semantic information in semantic clones from the training set.

Despite the quality of a labeled dataset being highly critical for these deep learning approaches, there exists no study for investigating limitations of BigCloneBench when being used to assess and

compare different deep learning approaches for detecting semantic clones, in the face of various data quality issues increasingly reported for other software engineering tasks [1, 26]. If researchers do not pay attention to data quality issues in the dataset, the reported effectiveness of models on the dataset is not convincing [1, 26]. For example, code duplication issues exist (resulted from identical and similar files) in both the training/validation and test sets used to train and assess deep learning approaches for source code [1]. Code duplication affects all evaluation metric values, and the effects observed by end-users are often significantly worse than the effects reported by evaluations conducted based on the dataset. Here is another example to illustrate the impact of the dataset on a deep learning approach. As the keynote presentation in Frontiers in AI and Robotics (FAIR 2020) [37], some deep learning approaches distinguish dogs and wolves based on the surrounding background instead of their different looks, leading to the model's poor effectiveness on new data. Although many research efforts have found the limitations of the dataset used by them, there is no study to investigate the limitations of BigCloneBench when used as the benchmark to assess and compare different deep learning approaches on semantic clone detection.

To fill this gap of lacking empirical investigation, in this article, we conduct an experimental study to find that many semantic clone pairs from BigCloneBench use the same identifier names, which, however, are not used in non-clone pairs. Based on this finding, we hypothesize that deep learning approaches can achieve high metric values on BigCloneBench by considering only the identifier name information. To validate this hypothesis, we develop an undesirable-by-design approach to detect semantic clones by utilizing only the identifier name information. This approach is undesirable purposely because code segments from a semantic clone pair can have quite different identifier names by definition and in practice, and detecting whether code segments are semantic clones shall be independent of how the identifier names in these code segments are named. We find that even this undesirable approach can achieve high effectiveness comparable to the effectiveness of state-of-the-art approaches on BigCloneBench. Note that the undesirable model trained on OJClone [27] fails to effectively detect semantic clones in OJClone, another major evaluation dataset popularly used by the research community.

To alleviate the identified issue in BigCloneBench, we abstract a subset of the identifier names in BigCloneBench to better assess the effectiveness of deep learning approaches (we denote the resulting new dataset as AbsBigCloneBench) that have less reliance on the identifier name information. Our experimental results show that the undesirable approach fails to detect semantic clones on AbsBigCloneBench effectively. However, the state-of-the-art approaches used in our experiments still perform well on AbsBigCloneBench by learning semantic features such as the lexical and structural information from code snippets.

We also empirically assess the cross-effectiveness of deep learning approaches on BigCloneBench and AbsBigCloneBench. We conduct an experiment to explore whether models trained with BigCloneBench (or AbsBigCloneBench) are also effective on AbsBigCloneBench (or BigCloneBench). The experimental results show that models trained with AbsBigCloneBench perform well when applied on BigCloneBench. However, models trained with BigCloneBench cannot be effectively applied to AbsBigCloneBench for detecting semantic clones. These results indicate that models trained with BigCloneBench fail to detect semantic clones if the identifier names are changed.

This article makes the following main contributions:

- **Assessment.** We design an undesirable-by-design approach named Linear-Model, which can achieve high effectiveness on BigCloneBench by utilizing only the identifier name information. Thus, deep learning approaches with high effectiveness evaluated on BigCloneBench may not really be effective in general. Researchers need to pay attention to the identifier naming in BigCloneBench when using BigCloneBench to assess and compare

deep learning approaches for detecting semantic clones. We abstract the identifier names in BigCloneBench to produce AbsBigCloneBench, which can be used to better assess the effectiveness of deep learning approaches on the task of detecting semantic clones. The experimental results show that abstracting the identifier names for BigCloneBench can help better assess the effectiveness of an approach on the task of detecting semantic clones. Researchers should strive to assess the effectiveness of their approaches on datasets that do and do not abstract the identifier names, respectively, in order to provide a more comprehensive view of approach effectiveness.

- **Training.** The models trained on AbsBigCloneBench have less reliance on the identifier name information. Through cross-experiments between BigCloneBench and AbsBigCloneBench, we find that the models trained with AbsBigCloneBench are also effective on BigCloneBench, but the models trained with BigCloneBench are not effective on AbsBigCloneBench.

The remainder of this article is structured as follows. Section 2 introduces the preliminary. Section 3 details the undesirable-by-design approach named Linear-Model. Sections 4 and 5 describe the experiments and result analysis, respectively. Section 6 introduces related work. Section 7 discusses our work. Finally, Section 8 concludes.

2 PRELIMINARY

In this section, we first introduce the problem definition of code clone detection. Then, we detail the construction of BigCloneBench. Finally, we introduce the motivating examples of this article.

2.1 Problem Definition

Given two code snippets C_i and C_j, we set their label $y_{i,j}$ to 1 if (C_i, C_j) is a clone pair or -1 otherwise. Then a set of training data of n code snippets $\{C_1, \ldots, C_n\}$ can be represented as $D = \{(C_i, C_j, y_{i,j}) | i, j \in [1, n], i < j\}$. Our goal is to train a Linear-Model to learn a function ϕ that maps any code snippet C to a feature vector $\boldsymbol{v}$ so that for any pair of code snippets (C_i, C_j), the cosine similarity $s_{i,j}$ of the two feature vectors v_i and v_j is as close to the corresponding label $y_{i,j}$.

We use Equation (1) to calculate the cosine similarity of two vectors of the same dimension:

$$Cosin\ Similarity(\boldsymbol{u}, \boldsymbol{v}) = \frac{\boldsymbol{u} \cdot \boldsymbol{v}}{\|\boldsymbol{u}\|\|\boldsymbol{v}\|} \tag{1}$$

Thus, we have the following equation (Equation (2)):

$$s_{i,j} = \frac{\phi(C_i) \cdot \phi(C_j)}{\|\phi(C_i)\|\|\phi(C_j)\|} \tag{2}$$

where $s_{i,j} \in [-1, 1]$.

To determine whether a pair of code snippets (C_i, C_j) is a clone pair or not during inference, we need to set a threshold value σ such that (C_i, C_j) is a clone pair if $s_{i,j} \geq \sigma$. We choose σ empirically based on the validation set.

2.2 BigCloneBench Dataset

2.2.1 Construction of BigCloneBench. BigCloneBench is mined from a big-data inter-project repository IJaDataset 2.0 [13]. It covers 10 functionalities. For each functionality, its mining steps are mainly divided into seven steps. (1) Select Target Functionality. The authors of BigCloneBench (in short as authors in the rest of this section) select a commonly needed functionality in open-source Java projects as its target functionality. (2) Identify Possible Implementations. The authors review Internet discussion (e.g., Stack Overflow) and API documentation (e.g., JavaDoc) to identify

the common implementations of the target functionality. (3) Create Specifications. The authors create a specification of the functionality. (4) Create Sample Snippet. After obtaining the possible implementations and a specification of the functionality in the second and third steps, the authors then create a sample snippet, which will be used to later search for code snippets. (5) Search for Code Snippets. The authors search for possible code snippets for the target functionality based on the sample snippet. When searching for similar code snippets to the sample snippet, to avoid introducing too many dissimilar code snippets and causing a lot of manual annotation burden, from the sample snippet, the authors identify the keywords and source code patterns intrinsic to the sample snippet with respect to implementing the functionality. Although searching with source code patterns may result in code snippets with dissimilar identifier names to a certain extent, searching with keywords may result in code snippets most of which have similar identifier names. (6) Build Candidate Set. Based on the resulting code snippets in the search result from the fifth step, the authors obtain the candidate possible code clone snippets. (7) Manual Tagging. Finally, the authors manually confirm the candidate code snippets.

2.2.2 Typifying the Clone Types in BigCloneBench. After the authors manually mark whether two code snippets are clone pairs, the clone types of the clone pairs are automatically marked. Type-I normalization includes the removal of comments and a strict pretty-printing. Type-II normalization expands Type-I normalization to include the systematic renaming of identifier names and replacement of literals with default values. To identify Type-III and Type-IV clones, the authors measure the syntactical similarity of the clones using a line-based metric after full normalization, including the removal of comments, a strict pretty-printing, the renaming of all identifier names to a common value, and the change of all literal values to a common value.

Although the authors use an abstract technique to distinguish between semantic clones (Type-IV) and non-semantic clones, it is not guaranteed that semantic clones do not rely on the identifier names. In the fifth step of the previous section, to avoid introducing a large number of false positives, the authors search for the clone code based on the same keywords and source code patterns. This process will lose a lot of true positives (i.e., code snippets with dissimilar identifier names), so the semantic clones in BigCloneBench are highly dependent on the identifier names.

2.3 Motivating Example

2.3.1 Analyzing BigCloneBench. We carefully read clone pairs in BigCloneBench and OJClone. We find that many clone pairs from BigCloneBench use the same identifier names, which are not used in non-clone pairs. However, this observation does not exist in OJClone. In addition, we further collect identifier names statistics by the Jaccard similarity coefficient to evaluate the similarity of the identifier names among different functionalities of two datasets. The Jaccard similarity coefficient is used to measure the similarity between two sets of data. It compares members for two sets to see which members are shared and which are distinct. The value varies from 0 to 1. The higher the value, the more similar the two sets. It is calculated as follows:

$$J(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}$$

where X and Y are two sets, $|X \cap Y|$ is the size of the intersection of X and Y, and $|X \cup Y|$ is the size of the union of X and Y.

To evaluate the similarity of the identifier names among different functionalities, we first get the Top 20 frequent identifier names in each functionality. Then, we compute the Jaccard similarity coefficient for every two functionalities, namely, $J(F_i, F_j)$ in which F_i and F_j are sets of Top 20 used identifier names in functionalities i and j. At last, we get the average Jaccard similarity coefficient

Table 1. Top 20 Frequent Identifier Names of Each Functionality on BigCloneBench

Functionality	Top 20 frequent identifier names
Web Download	InputStreamReader BufferedReader readLine URL url close IOException openStream line toString in InputStream append openConnection int getInputStream reader substring add length
Secure Hash (MD5)	getInstance MessageDigest digest update getBytes NoSuchAlgorithm Exception length md toString append toHexString i UnsupportedEncodingException StringBuffer md5 hash reset password encode text
Copy a File	close FileOutputStream IOException File FileInputStream String int write out read in InputStream copy IOUtils size FileChannel getChannel exists OutputStream length
Decompress Zip	getName File close isDirectory IOException FileOutputStream out in ZipArchiveEntry InputStream length entry IOUtils mkdirs getParentFile FileInputStream getNextEntry write exists copy
FTP Authenticated Login	login connect FTPClient IOException disconnect getReplyCode FTPReply isPositiveCompletion ftp logout setFileType reply BINARY_FILE_TYPE password changeWorkingDirectory enterLocalPassiveMode FTP close isConnected File
Bubble Sort	i length j temp a bubbleSort tmp sort n swapped size list t print k get field permut Index_value addLast
Init. SGV With Model	setContents ScrollingGraphicalViewer viewer setEditPartFactory setRootEditPart createControl ScalableFreeformRootEditPart SWT GraphicalViewer setEditDomain shell run Shell Composite getContents parent getShell open flush getWorkbench
SGV Selection Event Handler	ISelectionChangedListener selectionChanged SelectionChangedEvent addSelectionChangedListener event getSelection viewer setContents setEditPartFactory setRootEditPart setContextMenu setSelectionProvider Composite IStructuredSelection instanceof SWT getFirstElement getSite parent createControl
Create Java Project(Eclipse)	create setOutputLocation setNatureIds JavaCore setRawClasspath IProjectDescription open Path IJavaProject getWorkspace Resources Plugin IClasspathEntry setDescription NATURE_ID getProject CoreException getFullPath getDescription javaProject IProject
SQL Update and Rollback	executeUpdate rollback commit SQLException close prepareStatement PreparedStatement setAutoCommit Connection getConnection setString createStatement next setInt sql ResultSet conn executeQuery log stmt

among different functionalities J. The details to get the average Jaccard similarity coefficient are shown in Algorithm 1.

The Jaccard similarity coefficient among different functionalities of BigCloneBench is 0.038, it indicates the huge differences in the identifier names among different functionalities in BigCloneBench. For better illustration, Table 1 shows the top 20 frequent identifier names in each functionality. For example, programs in functionality "copy a file" usually use "file" related identifier names, but rarely used in other functionalities. On the other hand, the Jaccard similarity coefficient of OJClone is 0.469, indicating that about half of the frequent identifier names are the same among different functionalities. The difference of the Jaccard similarity coefficients between the two datasets indicates that the identifier name similarity among different functionalities in OJClone is much higher than that in BigCloneBench. The reason for the difference is the different construction of the two datasets. The programs of OJClone come from a programming environment

```java
public void actionPerformed(java.awt.event.ActionEvent e) {
    ...
    File file = fc.getSelectedFile();
    ...
    try{
        FileInputStream fis = new FileInputStream(file);
        FileOutputStream fos = new FileOutputStream(rutaGlobal, true);
        FileChannel canalFuente = fis.getChannel();
        FileChannel canalDestino = fos.getChannel();
        canalFuente.transferTo(0, canalFuente.size(), canalDestino);
        fis.close();
        fos.close();
    }
    catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

```java
public static void copyFile1(File srcFile, File destFile) throws IOExce
ption {
    if(!destFile.exists()) {
    destFile.createNewFile();
    }
    FileInputStream fis = new FileInputStream(srcFile);
    FileOutputStream fos = new FileOutputStream(destFile);
    FileChannel source = fis.getChannel();
    FileChannel destination = fos.getChannel();
    destination.transferFrom(source, 0, source.size());
    source.close();
    destination.close();
    fis.close();
    fos.close();
}
```

Fig. 1. Semantic clone pair with the similar identifier names from BigCloneBench.

ALGORITHM 1: Jaccard Similarity Coefficient Statistics

```
1:  procedure JACCARD(𝒟)                                   ▷ 𝒟 is the dataset
2:      ls ← list()
3:      for f in functionalities do
4:          l ← dict()                          ▷ Count list of identifier names in a functionality
5:          for c in codes do                              ▷ codes are all snippets belong to f
6:              s ← set()                                  ▷ s is the set of identifier names in code
7:              for id in identifiernames do
8:                  if id not in s then
9:                      s.add(id)
10:                     if id in l then l[id] ← l[id] + 1
11:                     else l[id] ← 1
12:         top ← l.most_common(20)                        ▷ Top 20 identifier names
13:         ls.add(top)
14:     sum, cnt ← 0, 0
15:     for F_i, F_j in ls and i ≠ j do
16:         J_ij ← computeJaccard(F_i, F_j)
17:         sum ← sum + J_ij
18:         cnt ← cnt + 1
19:     return sum/cnt
```

where students write solutions from scratch for various problems. Therefore, even for different problems, programmers may define the same identifier names. For example, students typically define the loop variable as i when implementing a single loop and define multiple loop variables as j, k when implementing nested loops. However, BigCloneBench is mined from a large open-source repository where the identifier names are more related to a specific functionality. Thus, the identifier names of code snippets that do not belong to the same functionality are quite dissimilar.

The two code snippets in Figure 1 are numbered 21,754,660 and 23,677,114, respectively, in BigCloneBench. These two code snippets are labeled as semantic clones in BigCloneBench. From the figure, we can find that although they are labeled as semantic clones, their identifier names are very similar, such as "File", "FileInputStream", "FileOutputStream", "IOException", "fis", and "fos".

Based on the preceding finding, we hypothesize that deep learning approaches can achieve high metric values on BigCloneBench by considering only the identifier name information. Abstracting the identifier names in BigCloneBench may provide a different lens for researchers to assess the effectiveness of deep learning approaches on the task of detecting semantic clones.

```java
public static void main (String [] args) {
    int x = 123456789;
    System.out.println ("x = " + x);
    int hi = x, n = 0;
    while (hi > 9) {
        hi /= 10;
        ++ n;
    }
    for (int i = 0;i < n; i ++) hi *= 10;
    x -= hi;
    System.out.println ("x with high digit removed = " + x);
}
```

```java
public static void main (String [] argv) {
    final int x = 123456789;
    int newX = x;
    final double originalLog = Math.floor (Math.log10 (x));
    final int getRidOf = (int) Math.pow (10, originalLog);
    while (originalLog == Math.floor (Math.log10 (newX))) {
        newX -= getRidOf;
    }
    System.out.println (newX);
}
```

Fig. 2. Semantic clone pair from Stack Overflow post "Remove digits from a number in Java".

2.3.2 An Example of Semantic Clone Pair with the Different Identifier Names in Real World. The previous section analyzes in detail that the semantic clones in BigCloneBench are highly dependent on the identifier names. Many research efforts [45, 48] have pointed out that semantic clones are difficult to detect because semantic clones may be different in lexical implementation. Figure 2 shows an example of semantic clone pairs from Stack Overflow post "Remove digits from a number in Java."

3 UNDESIRABLE-BY-DESIGN MODEL

In this section, we first introduce the overall structure of our proposed undesirable-by-design approach named Linear-Model and then explain the technical details of Linear-Model and max pooling [31].

3.1 Approach Overview

Figure 3 shows the architecture of Linear-Model. To process a code snippet, we first use Javalang[1] to parse code snippets into ASTs, as shown in Figure 4. Then we traverse all the nodes in the AST and remove the node duplicates determined based on their IDs (e.g., "MethodDeclaration", "copy", "FormalParameter"). Here we use Depth-First-Search (the way to traverse AST does not affect the traversal result). Finally, the set of the node IDs that we get is "MethodDeclaration", "copy", "FormalParameter", "src", "ReferenceType", "String", "dest", and "IOException". State-of-the-art approaches use the program information from AST nodes, including the lexical and structural information (represented as terminal nodes identified with dashed-line boxes). Therefore, for Linear-Model, we also leverage the sets of the traversed node IDs from ASTs to get the same lexical information as other state-of-the-art approaches. Still we use only little structural information of AST (e.g., "ForStatement" and "IfStatement") for Linear-Model. A detailed explanation for Linear-Model is shown in the next section. Then we use PACE (proposed by Yu et al. [48]) to initialize the embeddings of node IDs before feeding the vectorized node IDs into Linear-Model with a max-pooling layer. To detect clones, Linear-Model uses two linear matrices in parallel to process a pair of code snippets simultaneously. The two linear matrices share the same parameters. Linear-Model takes the output of the max-pooling layer as the feature vector of a code snippet and calculates the cosine similarity of the two vectors. At last, Linear-Model is trained through gradient descent back-propagation to minimize the MSE loss function:

$$\sum_{i}\sum_{j}(s_{i,j} - y_{i,j})^2 \tag{3}$$

$s_{i,j}$ and $y_{i,j}$ are the cosine similarity and the corresponding label of a pair code snippets, respectively. In summary, Linear-Model learns to make the cosine similarity of non-clone pairs as close to -1 as possible and the cosine similarity of clone pairs as close to 1 as possible.

[1] https://github.com/c2nes/javalang.

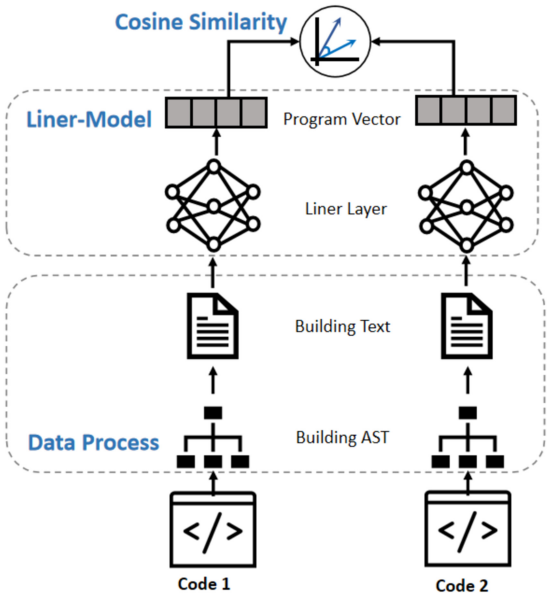

Fig. 3. Overall architecture.

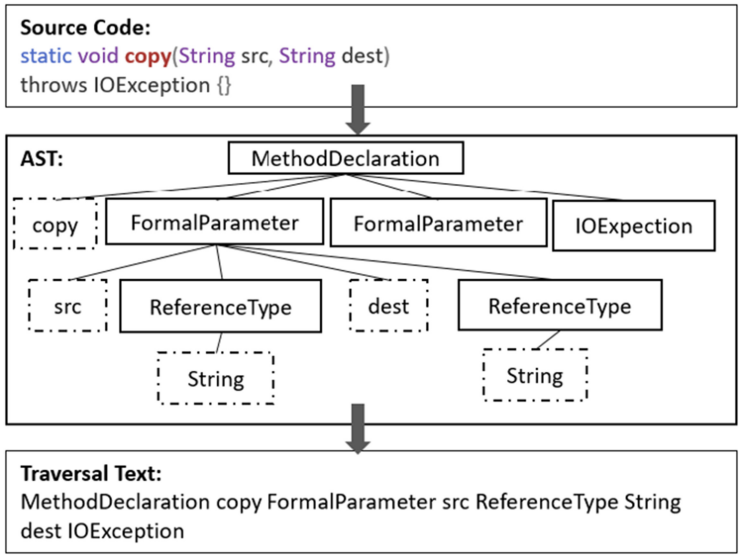

Fig. 4. An example of a code snippet, its AST, and the traversal node IDs of the AST.

3.2 Linear Operation and Max Pooling

Figure 5 shows the linear operation of Linear-Model, where there are n tokens in the set of node IDs, and the encoding length of each token is d. Linear-Model uses a $d \times m$ matrix for the linear operation, and then obtains a $n \times m$ matrix. After the linear operation, the number of tokens in the set of node IDs is the same as before. Then max pooling is applied to each dimension in the output vectors; each dimension corresponds to one feature detector, thereby reducing a set of node IDs

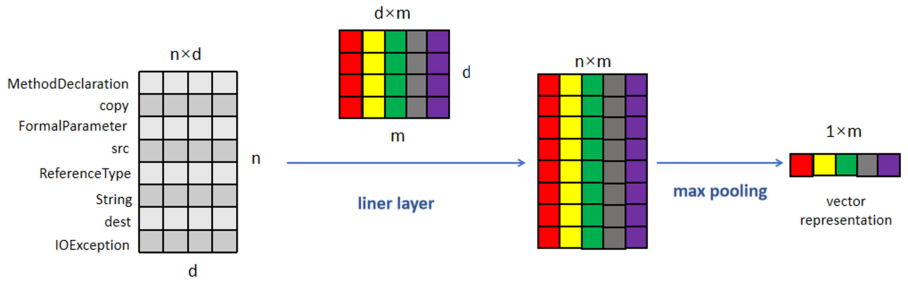

Fig. 5. Linear operation.

with any number of tokens to a $1 \times m$ vector. Linear-Model uses only little structural information of AST as Linear-Model knows only which statements (e.g., "ForStatement" and "IfStatement") appear in source code, but Linear-Model does not know the relationship among these statements. In addition, Linear-Model does not consider the sequential information of source code as it tries to see only which tokens appear in the source code.

Formally, there is 1 matrix with the set of node IDs represented by $W_{set} \in \mathbb{R}^{n \times d}$, where n and d respectively represent the number of tokens in the set of node IDs and the vector dimension of each token; then the output of Linear-Model is

$$W_{output} = W_{set} \cdot W_{linear} + b \tag{4}$$

where $W_{output} \in \mathbb{R}^{n \times m}, W_{linear} \in \mathbb{R}^{d \times m}, b \in \mathbb{R}^{m}$.

Note that Linear-Model uses only one simple linear layer. It does not use a hidden layer to capture the hidden features.

4 EXPERIMENTAL STUDIES

In this section, we describe the state-of-the-art approaches and datasets that we use in our experimental studies. According to Yu et al. [48] and Wei et al. [45], more than 98% of clone pairs in BigCloneBench [39] are semantic clones, and clone pairs in the OJClone dataset [27] belong to at least Type-III clones. Both datasets are widely used in detecting semantic clones via deep learning [45, 46, 48, 49]. Therefore, we conduct experiments on BigCloneBench and OJClone to explore whether and why "effective" approaches (assessed on BigCloneBench) based on deep learning are not really effective in general. We also conduct experiments on AbsBigCloneBench to address that abstracting a set of identifier names in BigCloneBench can help better assess the effectiveness of deep learning approaches on BigCloneBench for tasks of detecting semantic clones. In addition, we also design cross-experiments, i.e., (1) train models on BigCloneBench and test them on AbsBigCloneBench; and (2) train models on AbsBigCloneBench and test them on BigCloneBench.

We conduct the experiments to answer the following research questions:

- RQ1: Can the undesirable-by-design Linear-Model (by utilizing only a subset of the identifier names) achieve high effectiveness on BigCloneBench?
- RQ2: How effective are state-of-the-art approaches and Linear-Model on AbsBigCloneBench?
- RQ3: How effective are state-of-the-art approaches and Linear-Model on the cross-experiments?

We address RQ1 to point out that for the task of detecting semantic clones, the approaches based on deep learning assessed to be effective on BigCloneBench may not be really effective in general. We then address RQ2 to get an improved evaluation dataset named AbsBigCloneBench, which is

Table 2. Percentage of Clone Types in BigCloneBench

Clone Type	T1	T2	VST3	ST3	MT3	WT3/T4
Ratio	0.46%	0.06%	0.05%	0.19%	1.01%	98.23%

derived from BigCloneBench. Experimental results show that AbsBigCloneBench is more desirable for differentiating state-of-the-art approaches and the undesirable approach. At last, we address RQ3 to show that models trained with AbsBigCloneBench are also effective on BigCloneBench.

4.1 State-of-the-art Approaches

In this article, we select three state-of-the-art approaches of detecting semantic clones, including ASTNN [49], TBCCD [48], and FA [44]. In addition, to illustrate the impact of the identifier names in datasets, we design an undesirable-by-design approach named Linear-Model to detect semantic clones by utilizing only the identifier name information. Linear-Model is detailed in Section 3.

4.1.1 ASTNN. ASTNN [49] is an AST-based neural network for source code representation. It encodes the small statement trees split from entire trees and captures both lexical and structural knowledge of statements. Then, it applies a bidirectional RNN to produce the representation of the entire AST from the representation vectors of statements.

4.1.2 TBCCD. TBCCD [48] is recent work for semantic clone detection. It exploits the tree-based convolutional neural network to detect semantic clones. In addition, TBCCD includes a position-aware character embedding technique to eliminate the impacts of unseen code tokens.

4.1.3 FA. FA [44] uses graph neural networks for detecting semantic clones. FA first adds edges to the program's AST to form a graph named FA-AST. Then FA applies two different types of **graph neural networks (GNNs)** on FA-AST to measure the similarity of code pairs. As FA is customized for the Java language, we do not replicate it on OJClone, which is in the C language.

4.2 Datasets

We conduct experiments on two public datasets (the other two datasets are variants of these two datasets): BigCloneBench [40] and OJClone [27]. They are widely used benchmarks for clone detection [48, 49, 51]. Table 3 shows the detailed information of the two datasets that we use.

4.2.1 BigCloneBench. BigCloneBench is a large clone detection benchmark proposed by Svajlenko et al. [39]. BigCloneBench is mined from IJaDataset [13] and confirmed by three experts. IJaDataset [13] contains a total of 25,000 subject systems and 365M LOC. BigCloneBench covers 10 functionalities and contains six million tagged true clone pairs, and 260,000 tagged false clone pairs. Unlike the common taxonomy that groups clones into four types, BigCloneBench divides the Type-III clone type into weak, medium, strong, and very strong. Table 2 describes the proportion of each clone type in BigCloneBench. Semantic clone pairs (i.e., Moderately Type-III and Weak Type-III/Type-IV) account for more than 98% among all types of clone pairs. Therefore, many approaches exploit BigCloneBench to assess the effectiveness of approaches for detecting semantic clones.

We use the BigCloneBench dataset to answer our RQ1.

4.2.2 OJClone. OJClone is another widely used benchmark to assess the effectiveness of approaches for detecting semantic clones. OJClone[2] was proposed by Mou et al. [27]. OJClone is

[2]http://poj.openjudge.cn.

Table 3. Overall Information of BigCloneBench and OJClone

Datasets	Language	Code snippet	%Clone pair	AVG Len
BigCloneBench	Java	9,134	13.7	28.60
OJClone	C	7,500	6.7	32.25

derived from programming assignments submitted by students. It was originally used for code classification tasks. Later, CDLH [45], TBCCD [48], and ASTNN [49] used the OJClone dataset for the task of detecting semantic clones. As code snippets submitted by students while solving a specific problem share the same functionality, true clone pairs in OJClone are at least Type-III clones. Following state-of-the-art approaches, we also select the first 15 problems of the OJClone dataset in which each problem contains 500 solutions. In OJClone, two different code solutions that solve the same programming problem are considered as a true clone pair; otherwise, they are considered a false clone pair. Compared with the state-of-the-art approaches on the OJClone dataset, the undesirable-by-design Linear-Model proposed in this article is obviously invalid. Compared with BigCloneBench, OJClone can effectively assess the effectiveness of approaches for detecting semantic clones based on deep learning. In agreement with the experimental results, the Jaccard similarity of the identifier names among the different problems of OJClone that we analyze shows that OJClone is little dependent on the identifier names.

We use the OJClone dataset to answer our RQ1.

4.2.3 AbsBigCloneBench. AbsBigCloneBench is abstracted from BigCloneBench. AbsBigCloneBench abstracts a subset of the identifier names in BigCloneBench, such as type, variable, and method names, and retains other tokens such as operators, basic types, and member variables. According to the definition of clone type [2, 32, 35], the abstraction of AbsBigCloneBench will not change the labeling of semantic clones and non-semantic clones in BigCloneBench.

We use AbsBigCloneBench to answer our RQ2 and RQ3. We aim to illustrate that abstracting a subset of identifier names in BigCloneBench can assist BigCloneBench to more effectively assess the effectiveness of approaches for detecting semantic clones based on deep learning. Models trained on the AbsBigCloneBench dataset are also effective on the BigCloneBench dataset.

4.2.4 Over-AbsBigCloneBench. Over-AbsBigCloneBench is mentioned in Section 5.2.1. Like Mou et al. [27] and Yu et al. [48], we also explore the effectiveness of each approach when only non-terminal AST nodes are retained. In Figure 4, the terminal nodes are identified with dashed-line boxes. Over-AbsBigCloneBench remains only the tokens in solid-line boxes. This article does not emphasize Over-AbsBigCloneBench because the abstraction of Over-AbsBigCloneBench will change the labeling of semantic clones and non-semantic clones in BigCloneBench.

We use Over-AbsBigCloneBench to show that state-of-the-art approaches on such an over-abstraction dataset are also more effective than Linear-Model (Table 5).

4.3 Experimental Setting

For the BigCloneBench dataset, we use the same data as CDLH [45] and TBCCD [48], including 9,134 code snippets in total. We divide the dataset into the training set, validation set, and test set according to the ratio of 8: 1: 1. Specifically, we randomly select 913 code snippets to construct the test set, 913 code snippets to construct the validation set, and the remaining 7,308 code snippets to construct the training set. Finally, we get $\frac{913\times912}{2} = 416{,}328$ pairs in the test and validation sets, respectively, and $\frac{7{,}308\times7{,}307}{2} = 26{,}699{,}778$ pairs in the training set. Since the training set is enormous, we randomly select 300,000 pairs as the training set.

Table 4. Comparison of Different Approaches on BigCloneBench and OJClone

Approach	BigCloneBench			OJClone		
	P	R	F1	P	R	F1
ASTNN	0.93	0.94	0.93	0.98	0.98	0.98
TBCCD	0.94	0.95	0.95	0.99	0.99	0.99
FA	0.98	0.94	0.96	–	–	–
Linear-Model	0.91	0.94	0.93	0.70	0.71	0.70

P and R represent Precision and Recall, respectively. (The FA is customized for Java programs and cannot be used to detect clones written in C. Therefore, its results on OJClone are neglected.)

OJClone has a total of 7,500 code snippets. We also divide the training set, validation set, and test set according to the ratio of 8: 1: 1. We randomly select 750 code snippets to construct the test set and 750 code snippets to construct the validation set. The remaining 6,000 code snippets are used to construct the training set. We have $\frac{750\times749}{2} = 280{,}875$ pairs in the test and the validation sets, and $\frac{6{,}000\times5{,}999}{2} = 17{,}997{,}000$ pairs in the training set; since the training set is enormous, finally we randomly select 300,000 pairs as the training set.

Our sampling of the training set is completely random. We construct the training set, validation set, and test set according to 8:1:1 since this sampling is consistent with existing related work [44, 48, 49]. We replicate ASTNN, TBCCD, and FA by running the source code that they provide.

The parameters for training Linear-Model are as follows: m is 100, the number epochs that we train Linear-Model is 10, the optimizer is SGD, and the batch size is 1. The threshold of Linear-Model for prediction is determined with the validation set. We use TensorFlow[3] to implement Linear-Model. Our code for Linear-Model and the dataset of AbsBigCloneBench are publicly available.[4]

5 RESULTS

This section shows our experimental results and analysis.

5.1 RQ1: Can the Undesirable-by-design Linear-Model Achieve High Effectiveness on BigCloneBench by Utilizing only A Subset of the Identifier Names?

5.1.1 Precision and Recall on BigCloneBench and OJClone. To assess the aforementioned hypothesis, we compare Linear-Model, an undesirable-by-design approach, with state-of-the-art approaches for detecting semantic clones, i.e., ASTNN, TBCCD, and FA. Table 4 shows the Precision, Recall, and F1 score of different approaches on BigCloneBench and OJClone.

Table 4 shows that Linear-Model can achieve equivalent effectiveness to other approaches. Both Linear-Model and ASTNN have achieved 0.93 F1 score on the BigCloneBench dataset. We can see that even this undesirable approach can achieve equally high effectiveness comparable to state-of-the-art approaches on BigCloneBench.

However, the results of Linear-Model on OJClone are quite different from those on BigCloneBench. This undesirable approach cannot effectively detect semantic clones on OJClone. Desirable approaches (i.e., TBCCD, ASTNN, FA) reported by existing literature still perform well on both BigCloneBench and OJClone datasets. According to the analysis in their papers, these

[3] http://www.tensorflow.org.
[4] https://github.com/yh1105/AbsBigCloneBench.

approaches can detect semantic clones by capturing both the lexical and structural information in the source code.

Experimental results show that Linear-Model can effectively detect semantic clones in BigCloneBench, whereas all metrics are worsened on OJClone dramatically, and the F1 score of Linear-Model is 0.70, about 0.23 lower than that on BigCloneBench. Due to the great differences among top used identifier names in different functionalities of BigCloneBench, Linear-Model can achieve equivalent effectiveness as other approaches by learning these identifier names. For OJClone, it is difficult for Linear-Model to effectively learn the similarity among code snippets by learning the identifier names in source code, thus failing to detect semantic clones. However, state-of-the-art approaches can learn other features, e.g., structural features in programs.

5.1.2 Researchers Need to Pay Attention to the Identifier Names in BigCloneBench When Using BigCloneBench to Assess and Compare Deep Learning Approaches for Detecting Semantic Clones. We design an undesirable-by-design linear model named Linear-Model to detect semantic clones by utilizing only the identifier name information. This model is undesirable because code snippets from a semantic clone pair can have quite different identifier names by definition, and detecting whether code snippets are semantic clones can be independent of how the identifiers in these code snippets are named. Therefore, it is unreasonable to use Linear-Model to detect semantic clones because Linear-Model detects semantic clones by looking at only which tokens appear in the code snippets. As shown in Table 4, the huge difference of the identifier names among different functionalities in BigCloneBench helps Linear-Model achieve high effectiveness on the task of detecting semantic clones. However, when the difference disappears (e.g., OJClone), Linear-Model fails to detect semantic clones.

When researchers use the BigCloneBench dataset to assess the effectiveness of their approaches, they should pay attention to mitigating the impact of the identifier names. The next section illustrates that the abstraction technique for a subset of the identifier names in BigCloneBench can help better assess the effectiveness of approaches for detecting semantic clones. We introduce the abstraction technique in detail in the next section. In summary, we find that

Finding 1. An undesirable Linear-Model can achieve high effectiveness based on the identifier name information on BigCloneBench but fails to effectively detect semantic clones in OJClone. It is problematic to directly use only BigCloneBench to assess the effectiveness of approaches for detecting semantic clones, calling for further improvement before being used.

5.2 RQ2: How Effective are State-of-the-art Approaches and Linear-Model on AbsBigCloneBench?

To alleviate the issue in BigCloneBench discussed in RQ1, in this section, we abstract a subset of the identifier names in BigCloneBench to better assess the effectiveness of deep learning approaches. We denote the dataset after abstracting the identifier names as AbsBigCloneBench, such that the identifier names among code snippets with different functionalities in AbsBigCloneBench are much more similar. This section first compares the Recall, Precision, and F1 score of Linear-Model and state-of-the-art approaches on Over-AbsBigCloneBench. Next, we introduce how we modify BigCloneBench to get AbsBigCloneBench. Then, we compare the Recall, Precision, and F1 score of Linear-Model and the state-of-the-art approaches on AbsBigCloneBench. Finally, we discuss why AbsBigCloneBench is more desirable than BigCloneBench for assessing the effectiveness of deep learning approaches.

Table 5. Comparison of Different Approaches on Over-AbsBigCloneBench

Approach	Precision	Recall	F1	ΔF1
ASTNN	0.80	0.67	0.73	−0.20
TBCCD	0.62	0.71	0.66	−0.29
FA	0.77	0.76	0.77	−0.19
Linear-Model	0.42	0.25	0.44	−0.49

The experimental results and Jaccard similarity coefficient show that abstracting the identifier names in BigCloneBench can help better assess the effectiveness of an approach for detecting semantic clones.

5.2.1 Recall, Precision, and F1 Score of Different Approaches on Over-AbsBigCloneBench. Like Mou et al. [27] and Yu et al. [48], to further explore the impacts of code tokens on clone detection, we first conduct an experiment on AST without the lexical information. Noticing that the lexical information is located in the terminal nodes of the ASTs, we remove all terminal nodes from the ASTs to generate partial ASTs. We feed state-of-the-art approaches with partial ASTs for this experiment. We apply the traversed text from partial ASTs as input for Linear-Model to detect semantic clones. Figure 4 shows that the terminal node is identified with dashed-line boxes. For the experiment on Over-AbsBigCloneBench, a partial AST used for state-of-the-art approaches includes only identifiers with solid-line boxes, and the text used for Linear-Model is "MethodDeclaration", "FormalParameter", "ReferenceType", and "IOException". As all the code token information is removed, the partial AST mainly contain the structural information, the text traversed from ASTs contain only little structural information (e.g., ForStatement and IfStatement). Table 5 shows that when more information in the dataset is removed (all terminal node information is removed), the state-of-the-art approaches are more effective than Linear-Model by capturing the program's structural information. However, if we remove all code tokens (remove all terminal nodes and keep only non-terminal nodes in the program's AST), this removal will lose a lot of semantic information in the program.

5.2.2 How We Modify BigCloneBench to Get AbsBigCloneBench. Considering the common taxonomy of clone types [2, 32, 35], we abstract a subset of the identifier names in the source code and obtain AbsBigCloneBench to better assess the effectiveness of deep learning approaches. In particular, we abstract a subset of the identifier names, including type, variable, and method names in the code snippets to obtain AbsBigCloneBench. The other tokens, such as operators, basic types, and member variables are kept as the original ones to retain programs semantics. API invocations are essential for implementing the functionality of programs, so we keep these API invocations in AbsBigCloneBench. The abstraction technique is similar to Harer et al. [24] and Li et al. [9]. We discuss the abstraction technique used by Harer et al. and Li et al. in Section 6. Finally, our abstraction technique abstracts the type names, variable names, and method name for a method in the code snippets. Figure 6 shows an example of code snippet in BigCloneBench and AbsBigCloneBench.

After building the AbsBigCloneBench dataset, we compare the Jaccard similarity coefficient of BigCloneBench and AbsBigCloneBench. The Jaccard similarity coefficients of the identifier names among different functionalities in BigCloneBench and AbsBigCloneBench are 0.038 and 0.484, respectively, indicating that AbsBigCloneBench is less dependent on the identifier names than BigCloneBench.

5.2.3 Recall, Precision, and F1 Score of Different Approaches on AbsBigCloneBench. Similar to RQ1, we compare different approaches on AbsBigCloneBench to investigate whether

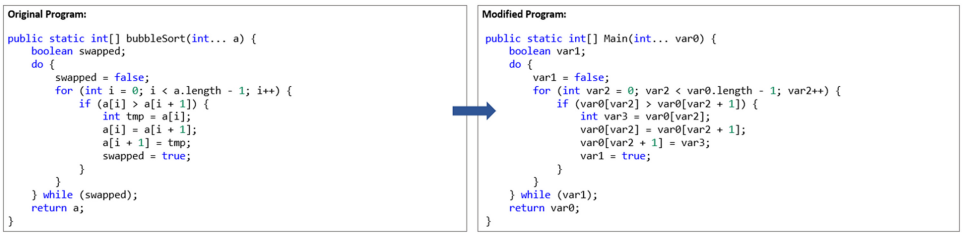

Fig. 6. An example of code snippet in BigCloneBench and AbsBigCloneBench.

Table 6. Comparison of Different Approaches on AbsBigCloneBench

Approach	Precision	Recall	F1	ΔF1
ASTNN	0.94	0.82	0.93	0
TBCCD	0.92	0.96	0.94	−0.01
FA	0.96	0.95	0.96	0
Linear-Model	0.81	0.88	0.83	−0.10

ΔF1 score indicates the relative improvement (worsening) compared to F1 scores on BigCloneBench in Table 4.

AbsBigCloneBench can be used as an evaluation dataset to assess the effectiveness of deep learning approaches on detecting semantic clones. The results are shown in Table 6, where ΔF1 is the relative improvement (worsening) compared to the F1 on BigCloneBench in Table 4.

Table 6 shows a substantial decrease in both the Precision and the Recall of Linear-Model compared to state-of-the-art approaches on AbsBigCloneBench. Compared to the results shown in Table 4, the state-of-the-art approaches can achieve equivalent effectiveness on AbsBigCloneBench. Although the identifier names are abstracted, the approaches can perform well by learning other semantic features such as the structural information from code snippets. However, it is limited for Linear-Model to learn other features while detecting semantic clones on AbsBigCloneBench. As we abstract a subset of the identifier names, including type, variable, and method names, the identifier names among code snippets with different functionalities in AbsBigCloneBench are much more similar than that in BigCloneBench. Therefore, Linear-Model may recognize some false clone pairs in AbsBigCloneBench as true clone pairs, thus decreasing the Precision on AbsBigCloneBench. On the other hand, it is also difficult for Linear-Model to learn features in true clone pairs, so the Recall also drops substantially.

We also compare **Precision-Recall (PR)** curves and AUC values of different approaches on BigCloneBench and AbsBigCloneBench, respectively. As shown in Figure 7, the AUC value of Linear-Model on AbsBigCloneBench is substantially lower than that on BigCloneBench. The AUC values of state-of-the-arts approaches on AbsBigCloneBench are almost the same as those on BigCloneBench. This result further shows that biases introduced from the subset of the identifier names can be reduced by abstracting them.

5.2.4 Why Can AbsBigCloneBench Help BigCloneBench Better Assess the Effectiveness of Approaches for Detecting Semantic Clones? AbsBigCloneBench is derived from BigCloneBench by abstracting a subset of the identifier names, including type, variable, and method names. The subset of the identifier names can be totally different according to the definition of semantic clones. As semantic clones are functionally similar, we keep API invocations essential for implementing the functionalities of programs. After abstracting a subset of the identifier names, the Jaccard

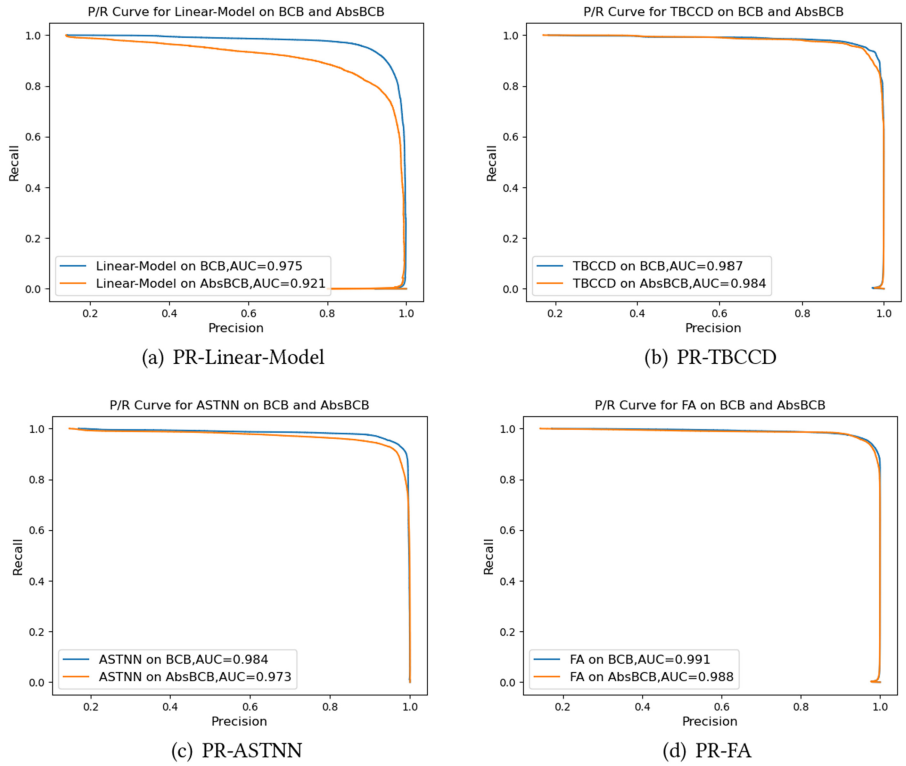

Fig. 7. PR curves and AUC of different approaches on BigCloneBench and AbsBigCloneBench.

similarity coefficient of AbsBigCloneBench improves to 0.484 from 0.038. The great improvement demonstrates the great increase of the similarity of the identifier name usage among different functionalities.

To show whether AbsBigCloneBench can be used to assess different deep learning approaches, we first compare Linear-Model with other approaches on AbsBigCloneBench. Table 6 shows that state-of-the-art approaches still perform well on AbsBigCloneBench, whereas the undesirable-by-design Linear-Model has a substantial effectiveness decline. Additionally, instead of achieving equivalent effectiveness to other approaches on BigCloneBench, Linear-Model is 0.10-0.13 lower than others in terms of F1. The Precision-Recall curve and AUC value in Figure 7 also shows that Linear-Model cannot detect semantic clones well on AbsBigCloneBench. The experimental results show that AbsBigCloneBench can better differentiate state-of-the-art approaches and the undesirable approach in terms of their effectiveness demonstrated on the evaluation dataset.

In summary, after comparing the results of many deep learning approaches in BigCloneBench and AbsBigCloneBench, we find that

Finding 2. Desirably abstracting the identifier names for BigCloneBench can help differentiate the state-of-the-art approaches and the undesirable approach in terms of their effectiveness demonstrated on the evaluation dataset. Abstracting the identifier names for BigCloneBench can help better assess the effectiveness of an approach for detecting semantic clones. In other words, abstracting the identifier names for BigCloneBench can help illustrate an approach's reliance on the identifier names for detecting semantic clones.

Table 7. Comparison of Different Approaches on the Cross-experiments

Approaches	Precision	Recall	F1	Δ F1
Train: BigCloneBench Test: AbsBigCloneBench				
ASTNN	0.69	0.63	0.66	−0.27
TBCCD	0.63	0.72	0.67	−0.28
FA	0.87	0.74	0.80	−0.16
Linear-Model	0.58	0.60	0.59	−0.34
Train: AbsBigCloneBenchTest: BigCloneBench				
ASTNN	0.93	0.88	0.91	−0.02
TBCCD	0.91	0.92	0.92	−0.02
FA	0.95	0.92	0.94	−0.02
Linear-Model	0.75	0.70	0.73	−0.10

Δ F1 score indicates the relative improvement (worsening) compared to F1 scores in Tables 4 and 6.

5.3 RQ3: How Effective are State-of-the-art Approaches and Linear-Model on the Cross-experiments?

In this section, we conduct cross-experiments to explore whether models trained with BigCloneBench (AbsBigCloneBench) are also effective on AbsBigCloneBench (BigCloneBench). The cross-experiments are divided into two groups: (1) train models on BigCloneBench and test them on AbsBigCloneBench; (2) train models on AbsBigCloneBench and test them on BigCloneBench. In the end of this section, we provide three practical assessment techniques to check whether a dataset is reasonable for assessing the effectiveness of an approach for detecting semantic clones based on deep learning.

5.3.1 Different Models Trained on BigCloneBench and Tested on AbsBigCloneBench. Table 7 shows the different models on the cross-experiments in BigCloneBench and AbsBigCloneBench. The F1 score and Δ F1 score (i.e., scores in the upper part) in Table 7 show that all models trained on BigCloneBench do not work well on AbsBigCloneBench. The experimental results indicate that these models are not effective on another dataset (i.e., AbsBigCloneBench) with different identifier names (e.g., type, variable, and method names). Models trained on BigCloneBench do not work well on AbsBigCloneBench because models trained on BigCloneBench can achieve convergence quickly by using the identifier name information. Using BigCloneBench is limited for all approaches (i.e., ASTNN, TBCCD, FA, and Linear-Model) to effectively learn the lexical and structural information. Therefore, when the identifier names in the test set are different from the training set, the state-of-the-art approaches cannot effectively detect semantic clones. In summary, we find that

Finding 3. Models trained on BigCloneBench are not effective on AbsBigCloneBench, but models trained on AbsBigCloneBench (except Linear-Model) are still effective on BigCloneBench. The experimental results of the cross-experiments suggest that AbsBigCloneBench provides a more comprehensive view of an approach's effectiveness that is less reliant on the identifier names than BigCloneBench.

5.3.2 Different Models Trained on AbsBigCloneBench and Tested on BigCloneBench. As shown in Table 7, models (except Linear-Model) trained on AbsBigCloneBench still perform well on BigCloneBench. From the Δ F1 score in Table 7, we find that the results of Linear-Model have a substantial decline, i.e., 0.10 in terms of the Δ F1 score compared to other approaches (i.e., 0.01). As

Linear-Model captures features from the identifier names, the effectiveness of Linear-Model decreases when the identifiers in the test set are different from the training set. However, the state-of-the-art approaches show only a slight decrease. After abstracting the subset of the identifier names in the training set, the state-of-the-art approaches mainly rely on the structural information to learn functional similarities between semantic clone pairs. The effectiveness of the state-of-the-art approaches would not be affected when applied to other datasets with different identifier names. In summary, we find that

Finding 4. Models trained on AbsBigCloneBench are also effective on BigCloneBench.

5.3.3 Three Dataset Properties. Tables 4 and 6 show that both AbsBigCloneBench and OJClone can be used to better assess an approach's effectiveness of detecting semantic clones than BigCloneBench. Here we propose three dataset properties to check whether a dataset can be used for assessing the effectiveness of an approach based on deep learning. We denote the dataset (consisting of the training, validation, and test sets) before abstraction as *pre-abstraction dataset*, and the dataset (consisting of the training, validation, and test sets) after abstraction as *post-abstraction dataset*. In addition, we use *mixed dataset* to denote a dataset that consists of the training and validation sets from the *pre-abstraction dataset* and the test set from *post-abstraction dataset*.

- **Property 1: Preserving effectiveness ranking of the given approaches assessed across the *pre-abstraction dataset* and the *post-abstraction dataset*.**
  The relative effectiveness ranking of the given approaches assessed on the *pre-abstraction dataset* shall be the same or similar as the ranking of the given approaches assessed on the *post-abstraction dataset*.
- **Property 2: Preserving effectiveness of the given approach assessed across the *pre-abstraction dataset* and the *mixed dataset*.** The effectiveness of the given approach assessed on the *pre-abstraction dataset* shall be the same or similar as the effectiveness of the approach assessed on the *mixed dataset*.
- **Property 3: Achieving ineffectiveness with an undesirable approach assessed on the *post-abstraction dataset*.**
  An undesirable approach (such as Linear-Model, which relies upon only the identifier names) shall be ineffective on the *post-abstraction dataset*, performing worse than state-of-the-art approaches assessed on the *post-abstraction dataset*.

6 RELATED WORK

In this section, we introduce the related work of identifier normalization, code clone detection, and learning from big code.

6.1 Identifier Normalization in Software Engineering

The techniques of abstracting identifier names have been applied in many approaches based on deep learning for software engineering tasks. In the task of vulnerability detection, Li et al. [24] abstract identifier names to reduce their approach's dependence on the identifier names. Li et al. first remove the comment information in the given code snippets, then abstract away the variable names in the code snippets, and finally abstract away the custom method names in the code snippets. In addition to the variable and method names, Harer et al. [9] also abstract constants in code snippets. Kim et al. [17] abstract formal parameter names, local variable names, data type names, and method names to detect vulnerabilities. They normalize the method body by removing the comments, whitespaces, tabs, and line feed characters and converting all characters into lowercase.

In the task of code completion, Cummins et al. [5] automatically synthesize a large number of OpenCL benchmarks by learning a character-level LSTM over valid OpenCL code. The goal of the OpenCL benchmarks is to generate reasonable-looking code rather than synthesizing a program that complies with a specification. To ease the task of synthesizing the OpenCL benchmarks, Cummins et al. normalize the code by consistently renaming variables and method names. For Python code, Bhoopchand et al. [3] use a token sparse pointer-based neural model that learns to copy recently declared identifier names to capture long-range dependencies of the identifier names. Bhoopchand et al. normalize the identifier names before feeding the resulting token stream to their approach. Also, Bhoopchand et al. replace each identifier name with an anonymous identifier name indicating the identifier group (class, variable, argument, attribute, or function) concatenated with a random number that makes the identifier names unique in the scope of the identifier group.

In the task of clone detection, many researchers pay attention to the abstraction of identifier names. Roy and Cordy [33] propose a tool to use flexible code normalization, which is not simply limited to global replacement (e.g., replacement of all identifier and literal names), or simple abstraction (e.g., abstraction of loop bodies). Tool users can choose to normalize the specific parts that the users expect to vary. Kamiya et al. [16] design multiple transformation rules (e.g., "Remove namespace attribution", "Remove template parameters", and "Remove initialization lists") for C++ code, and rules ("Remove package names", "Supplement callees", and "Separate class definitions") for Java code to abstract the identifier names. Although many approaches for code clone detection have applied identifier abstraction techniques, they are all different from our application scenarios. First, they are not based on deep learning. Second, they focus on improving the effectiveness of clone detection through abstracting the identifier names, while we focus on assessing the effectiveness of deep learning approaches for detecting semantic clones.

6.2 Code Clone Detection

Code clone detection is an important research problem in the field of software engineering. Detecting clones can help reduce the cost of software maintenance and prevent faults. Clones are similar code snippets that share the same semantics but may differ syntactically to various degrees. One common taxonomy among researchers is to group clones into four types, i.e., Type-I to Type-IV [2, 32, 35]. Type I-III clones are clone pairs that differ at the token and statement levels. Type-I clones are identical code snippets in addition to variations in comments and layout. Apart from Type-I clones, Type-II clones are identical code snippets except for different identifier names and literal values. Apart from Type-II clones, Type-III clones are syntactically similar code snippets that differ at the statement level. Apart from Type-III clones, Type-IV clones are functionally similar code snippets with different implementations. The difference in the identifier names does not affect the Type-II, Type-III, and Type-IV clones.

Many approaches have been proposed to detect clones. These approaches mainly measure the similarity among code representations varying from lexical-based [16, 18, 28, 33, 36, 43], structural-based [6–8, 15], and graph-based [19, 21, 25, 29] representations. CCFinder [16] and SourcererCC [36] detect clones according to the comparison of tokens. Deckard [15] detects clones by comparing the structural similarity between the ASTs.

Deep learning approaches [23, 47–49, 51] have recently received substantial attention for clone detection. White et al. [47] propose to learn latent features for source code via a recursive auto-encoder over ASTs. To detect Type-IV clones effectively, Wei et al. [45] formulate code clone detection as a supervised learning task. They propose an end-to-end deep learning approach named CDLH to learn hash codes from the lexical and structural information. Zhao et al. [51] propose DeepSim to measure functional similarity. DeepSim encodes the program's control flow graph and data flow graphs into matrices and then uses them to train a deep learning model to

compute the similarity between two code snippets. Yu et al. [48] propose TBCCD to detect semantic clones using tree-based convolutional neural network capturing both the lexical and structural information of code snippets.

6.3 Learning from Big Code

Large open-source software systems [42, 50] have arisen and been ubiquitous in recent years. They provide billions of code tokens for various software engineering tasks, such as code clone detection [44, 48], bug fixes [30], and code summarization [10–12]. The availability of "Big Code" suggests new data-driven approaches for learning statistical distributions from source code. These approaches estimate distributional properties over large and representative software corpora.

Deep learning approaches have a powerful ability to generalize from "Big Code" and handle noise in code examples. Various deep learning approaches have been proposed to address software engineering tasks. Research in the "Big Code" area relies on a large corpora of code. However, a few studies address issues with the "Big Code". Allamani [1] describes the impact of code duplication on deep learning approaches, and finds that code duplication can cause the evaluation to overestimate deep learning approaches. LeClair et al. [22] address the problem of identifier names in summarizing source code by using a project dataset without words from source code.

7 DISCUSSION

In this section, we first discuss the implications and limitations of our work. Then we discuss why we choose Linear-Model to compare with state-of-the-art approaches, and make clarification: we do not emphasize that the identifier names of semantic clones should not be similar. Finally, we show the effectiveness of SVM on BigCloneBench, OJClone, and AbsBigCloneBench.

7.1 Implications

7.1.1 Assessing the Effectiveness of Deep Learning Approaches for Detecting Semantic Clones. We point out an essential issue in BigCloneBench, widely used in clone detection; other researchers have not noticed this issue. We notice that BigCloneBench is used to assess the effectiveness of many approaches based on deep learning. However, researchers do not notice that the undesirable-by-design approach (Linear-Model) can still achieve high effectiveness on BigCloneBench due to BigCloneBench's substantial dependence on the identifier information. We point out this issue, which has not been noticed by researchers before, and propose to alleviate this issue through abstracting a subset of the identifier names. The experimental results also show that abstracting the identifier names in BigCloneBench can help better assess the effectiveness of approaches for detecting semantic clones based on deep learning. In addition, the models trained on AbsBigCloneBench are also effective on BigCloneBench.

7.1.2 Possible Scenarios. Not only AbsBigCloneBench can be used to better assess the effectiveness of approaches for detecting semantic clones, models trained on AbsBigCloneBench also have meaningful application scenarios. For example, for the task of malware and vulnerability detection, approaches need to learn the behavioral characteristics of the malware and vulnerabilities; such learning is not feasible when using only the identifier information. Assisting programming learners to search for more diverse implementations can be another application scenario.

7.2 Limitations

BigCloneBench is mined from real world projects. In many real world projects, the identifier names in the code snippets that belong to a semantic clone pair may be similar. If we abstract the identifier names in BigCloneBench to result in AbsBigCloneBench, AbsBigCloneBench may be

different from the real world scenario, and AbsBigCloneBench may also increase the difficulty of detecting semantic clones. Here we argue that if an approach is assessed to be effective on AbsBigCloneBench with the identifier names abstracted away, the approach will still be effective in the real world project (i.e., BigCloneBench).

7.3 Why Do We Choose Linear-Model to Compare with State-of-the-art Approaches?

We design an undesirable-by-design approach named Linear-Model to point out the negative impact of the identifier names on semantic clone detection via deep learning. Section 3 introduces the configuration of Linear-Model in detail. We ensure that Linear-Model does not consider the sequential information of source code as it tries to see only which tokens appear in the source code. Compared to Linear-Model, state-of-the-art approaches deeply capture the structural information of a code snippet while detecting semantic clones. For example, TBCCD captures the structural information of a code snippet and a token's position in the AST to detect semantic clones. CDLH uses AST-based LSTM to capture both lexical and structural information of the code snippet. ASTNN divides the AST of a code snippet into multiple subtrees; after the vector representation of the subtree is obtained, the structural information of the code snippet is captured through the bidirectional GRU. We choose the undesirable Linear-Model to assess whether it can perform well by using only the identifier name information to detect semantic clones on BigCloneBench. Linear-Model can be replaced by other deep learning approaches that utilize only the identifier name information.

7.4 Do We Emphasize That the Identifier Names of Semantic Clones Should Not Be Similar?

Some researchers may argue that the identifier name information is meaningful for detecting semantic clones in the real world scenarios. In this article, we are not saying that the identifier names of two code snippets belonging to a semantic clone pair should not be similar; we just emphasize that the identifier names of the semantic clone pair can be different, and the identifier names in the code snippets can cause confusion when the effectiveness of deep learning approaches is assessed. According to the common taxonomy of clone types [2, 32, 35], the identifier names can differ if two code snippets belong to a semantic clone pair. AbsBigCloneBench can eliminate the impact of the identifier name information on detecting semantic clones as much as possible, so abstracting the identifier names in BigCloneBench can help better assess the effectiveness of deep learning approaches on the task of detecting semantic clones.

7.5 Will a Traditional Machine Learning Approach Achieve Results as Good as Linear-Model?

Before deep learning algorithms became popular, **Support Vector Machine (SVM)** [4, 38, 41] were one of the most popular machine learning models. For the clone detection task, Jadon [14] proposes to detect clones by SVM. We also compare the effectiveness of SVM (a non-deep learning approach) on BigCloneBench and OJClone. We first perform lexical analysis on code snippets to obtain the tokens of the code snippets. Similar to the text classification task with SVM, we stitch two code snippets into a text and treat the clone detection task as a binary classification task. The F1 score of SVM on BigCloneBench can reach 0.75, while reaching 0.11 on OJClone (the F1 score of Linear-Model on BigCloneBench can reach 0.93, while reaching 0.70 on OJClone). The F1 scores of SVM on BigCloneBench and OJClone are lower than Linear-Model; the reason is that SVM is still inferior to Linear-Model in terms of the ability to capture features. One of the advantages of a deep learning approach is that it captures hidden features. Consistent with Linear-Model, the experimental results show that SVM works on BigCloneBench, but does not work on OJClone.

8 CONCLUSION

In this article, we have found that the semantic clones in BigCloneBench heavily rely on the identifier name information. According to the definition of clone type, the identifier names of semantic clones can be different. We have conducted an experimental study on BigCloneBench to compare the effectiveness of an undesirable-by-design approach named Linear-Model and other state-of-the-art deep learning approaches. The experimental results show that it is questionable to use only BigCloneBench to assess the effectiveness of approaches for detecting semantic clones. To alleviate the impact of the identifier names in BigCloneBench, we abstract a subset of the identifier names in BigCloneBench to produce a new dataset named AbsBigCloneBench. The experimental results show that abstracting the identifier names for BigCloneBench can help differentiate the state-of-the-art approaches and the undesirable approach in terms of their effectiveness demonstrated on the evaluation dataset. Models trained with AbsBigCloneBench are desirably less dependent on the identifier names than those trained with BigCloneBench. In addition, models trained with AbsBigCloneBench are also effective on BigCloneBench, but models trained with BigCloneBench are not effective on AbsBigCloneBench.

We hope that our work can inspire future research on using deep learning for semantic clone detection; researchers need to pay attention to whether using only the identifier name information in the dataset can achieve good effectiveness. Researchers also need to pay attention to whether similar problems exist in the used evaluation benchmarks for other software engineering tasks.

REFERENCES

[1] Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. 143–153. https://doi.org/10.1145/3359591.3359735

[2] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering* 33, 9 (2007), 577–591. https://doi.org/10.1109/TSE.2007.70725

[3] Avishkar Bhoopchand, Tim Rocktäschel, Earl Barr, and Sebastian Riedel. 2016. Learning Python Code Suggestion with a Sparse Pointer Network. (2016). arXiv:cs.NE/1611.08307

[4] Pai-Hsuen Chen, Chih-Jen Lin, and Bernhard Schölkopf. 2005. A tutorial on ν-support vector machines: Research articles. *Appl. Stoch. Model. Bus. Ind.* 21, 2 (2005), 111–136.

[5] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. Synthesizing benchmarks for predictive modeling. In *Proceedings of the 15th IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 86–99. https://doi.org/10.1109/CGO.2017.7863731

[6] Jianglang Feng, Baojiang Cui, and Kunfeng Xia. 2013. A code comparison algorithm based on AST for plagiarism detection. In *Proceedings of the 4th International Conference on Emerging Intelligent Data and Web Technologies (EIDWT)*. 393–397. https://doi.org/10.1109/EIDWT.2013.73

[7] Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable detection of semantic clones. In *Proceedings of the 30th ACM/IEEE International Conference on Software Engineering (ICSE)*. 321–330. https://doi.org/10.1145/1368088.1368132

[8] Yi Gao, Zan Wang, Shuang Liu, Lin Yang, Wei Sang, and Yuanfang Cai. 2019. TECCD: A tree embedding approach for code clone detection. In *Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 145–156. https://doi.org/10.1109/ICSME.2019.00025

[9] Jacob A. Harer, Louis Y. Kim, Rebecca L. Russell, Onur Ozdemir, Leonard R. Kosta, Akshay Rangamani, Lei H. Hamilton, Gabriel I. Centeno, Jonathan R. Key, Paul M. Ellingwood, Erik Antelman, Alan Mackay, Marc W. McConley, Jeffrey M. Opper, Peter Chin, and Tomo Lazovich. 2018. Automated Software Vulnerability Detection with Machine Learning. (2018). arXiv:cs.SE/1803.04497

[10] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th IEEE/ACM International Conference on Program Comprehension (ICPC)*. 200–210. https://doi.org/10.1145/3196321.3196334

[11] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering* 25, 3 (2020), 2179–2217. https://doi.org/10.1007/s10664-019-09730-9

[12] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing source code with transferred API knowledge. In *Proceedings of the 27th AAAI International Joint Conference on Artificial Intelligence (IJCAI)*. 2269–2275. https://doi.org/10.24963/ijcai.2018/314

[13] IJaDataset2.0. January 2013. Ambient Software Evolution Group. (January 2013). http://secold.org/projects/seclone.
[14] Shruti Jadon. 2016. Code clones detection using machine learning technique: Support vector machine. In *Proceedings of the 2nd IEEE International Conference on Computing, Communication and Automation (ICCCA)*. 399–403. https://doi.org/10.1109/CCAA.2016.7813733
[15] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th ACM/IEEE International Conference on Software Engineering (ICSE)*. 96–105. https://doi.org/10.1109/ICSE.2007.30
[16] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670. https://doi.org/10.1109/TSE.2002.1019480
[17] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A scalable approach for vulnerable code clone discovery. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*. 595–614. https://doi.org/10.1109/SP.2017.62
[18] Egambaram Kodhai and Selvadurai Kanmani. 2014. Method-level code clone detection through LWH (light weight hybrid) approach. *Software Engineering Research & Development* 2, 1 (2014), 12. https://doi.org/10.1186/s40411-014-0012-8
[19] Raghavan Komondoor and Susan Horwitz. 2001. Using slicing to identify duplication in source code. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Static Analysis (SAS)*. 40–56. https://doi.org/10.1007/3-540-47764-0_3
[20] Rainer Koschke, Raimar Falke, and Pierre Frenzel. 2006. Clone detection using abstract syntax suffix trees. In *Proceedings of the 13th IEEE Working Conference on Reverse Engineering (WCRE)*. 253–262. https://doi.org/10.1109/WCRE.2006.18
[21] Jens Krinke. 2001. Identifying similar code with program dependence graphs. In *Proceedings of the 8th IEEE Working Conference on Reverse Engineering (WCRE)*. 301–309. https://doi.org/10.1109/WCRE.2001.957835
[22] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE)*. 795–806. https://doi.org/10.1109/ICSE.2019.00087
[23] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. 2017. CCLearner: A deep learning-based clone detection approach. In *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 249–260. https://doi.org/10.1109/ICSME.2017.46
[24] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A deep learning-based system for vulnerability detection. *Proceedings of the 26th Network and Distributed System Security Symposium (NDSS)*. https://doi.org/10.14722/ndss.2018.23158
[25] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. 2006. GPLAG: Detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 872–881. https://doi.org/10.1145/1150402.1150522
[26] Zhongxin Liu, Xin Xia, Ahmed E. Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural-machine-translation-based commit message generation: How far are we?. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 373–384. https://doi.org/10.1145/3238147.3238190
[27] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI)*. 1287–1293.
[28] Hiroaki Murakami, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. 2013. Gapped code clone detection with lightweight source code analysis. In *Proceedings of the 21st IEEE International Conference on Program Comprehension (ICPC)*. 93–102. https://doi.org/10.1109/ICPC.2013.6613837
[29] Annamalai Narayanan, Mahinthan Chandramohan, Lihui Chen, Yang Liu, and Santhoshkumar Saminathan. 2016. SubGraph2Vec: Learning Distributed Representations of Rooted Sub-graphs from Large Graphs. (2016). arXiv:cs.LG/1606.08928
[30] Michael Pradel and Koushik Sen. 2018. DeepBugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*. 2, OOPSLA, Article 147 (2018). https://doi.org/10.1145/3276517
[31] Marc'Aurelio Ranzato, Y-Lan Boureau, and Yann LeCun. 2007. Sparse feature learning for deep belief networks. In *Proceedings of the 20th International Conference on Neural Information Processing Systems (NIPS)*. 1185–1192.
[32] Chanchal Kumar Roy and James R. Cordy. 2007. A survey on software clone detection research. *Queen's School of Computing TR* 541, 115 (2007), 64–68.
[33] Chanchal Kumar Roy and James R. Cordy. 2008. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC)*. 172–181. https://doi.org/10.1109/ICPC.2008.41

[34] Chanchal Kumar Roy and James R. Cordy. 2009. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *Proceedings of the 3rd IEEE International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*. 157–166. https://doi.org/10.1109/ICSTW.2009.18

[35] Chanchal Kumar Roy and James R. Cordy. 2018. Benchmarks for software clone detection: A ten-year retrospective. In *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 26–37. https://doi.org/10.1109/SANER.2018.8330194

[36] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling code clone detection to big-code. In *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering (ICSE)*. 1157–1168. https://doi.org/10.1145/2884781.2884877

[37] Harry Shum. 2020. From deep learning to deep understanding. In *Frontiers in AI and Robotics*. https://www.bilibili.com/video/av754058420.

[38] Alex J. Smola and Bernhard Schölkopf. 2004. A tutorial on support vector regression. *Statistics and Computing* 14, 3 (2004), 199–222. https://doi.org/10.1023/B:STCO.0000035301.49549.88

[39] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME)*. 476–480. https://doi.org/10.1109/ICSME.2014.77

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

[41] V. David Sánchez A. 2003. Advanced support vector machines and kernel methods. *Neurocomputing* 55, 1 (2003), 5–20. https://doi.org/10.1016/S0925-2312(03)00373-4

[42] Suresh Thummalapenta and Tao Xie. 2007. PARSEWeb: A programmer assistant for reusing open source code on the web. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 204–213. https://doi.org/10.1145/1321631.1321663

[43] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K. Roy. 2018. CCAligner: A token based large-gap clone detector. In *Proceedings of the 40th IEEE/ACM International Conference on Software Engineering (ICSE)*. 1066–1077. https://doi.org/10.1145/3180155.3180179

[44] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 261–271. https://doi.org/10.1109/SANER48275.2020.9054857

[45] Hui-Hui Wei and Ming Li. 2017. 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 (IJCAI)*. 3034–3040. https://doi.org/10.24963/ijcai.2017/423

[46] Hui-Hui Wei and Ming Li. 2018. Positive and unlabeled learning for detecting software functional clones with adversarial training. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI)*. 2840–2846. https://doi.org/10.24963/ijcai.2018/394

[47] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 87–98. https://doi.org/10.1145/2970276.2970326

[48] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. 2019. Neural detection of semantic code clones via tree-based convolution. In *Proceedings of the 27th IEEE International Conference on Program Comprehension (ICPC)*. 70–80. https://doi.org/10.1109/ICPC.2019.00021

[49] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE)*. 783–794. https://doi.org/10.1109/ICSE.2019.00086

[50] Yuxia Zhang, Minghui Zhou, Audris Mockus, and Zhi Jin. 2021. Companies' participation in OSS development–An empirical study of OpenStack. *IEEE Transactions on Software Engineering* 47, 10 (2021), 2242–2259. https://doi.org/10.1109/TSE.2019.2946156

[51] Gang Zhao and Jeff Huang. 2018. 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 (ESEC/FSE)*. 141–151. https://doi.org/10.1145/3236024.3236068

Received January 2021; revised September 2021; accepted November 2021