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
TAO XIE†, 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

In recent years, applying deep learning to detect semantic code clones (in short as semantic clones) has received substantial attention from the research community. Accordingly, various evaluation datasets, with the most popular one as BigCloneBench, are constructed and selected as benchmarks to validate 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 approaches for detecting semantic code clones. In this article, we present an experimental study to show that (1) BigCloneBench typically includes semantic clone pairs for using the same identifier names, which however are not used in non-semantic-clone pairs (according to a common taxonomy of clone types, the identifiers of the semantic clones can be different, and the identifier information in BigCloneBench affects the validity of the evaluation of the semantic clone detection approach) and (2) subsequently, we propose an undesirable-by-design deep learning model that considers only which identifier appears in code fragment (even not considering the sequence relationship between identifiers); 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. Based on this result, researchers need to pay attention to the identifiers in BigCloneBench when using BigCloneBench (as it is) for validating and comparing deep learning models for detecting semantic clones. To alleviate this issue, we abstract part-identifier names in BigCloneBench to result in AbsBigCloneBench, and use AbsBigCloneBench to better evaluate the effectiveness of deep learning approaches on the task of detecting semantic code clones. The experimental results show that models trained with AbsBigCloneBench are less dependent on identifier names than those models trained with BigCloneBench, and models trained on AbsBigCloneBench are also effective on BigCloneBench.

CCS Concepts: • **Software and its engineering → Software creation and management**.

Additional Key Words and Phrases: Code Clone Detection, Deep Learning, Dataset Collection

ACM Reference Format:
Hao Yu, Xing Hu, Ge Li, Tao Xie, Ying Li, and Qianxiang Wang. 2021. Assessing and Improving an Evaluation Dataset for Detecting Semantic Code Clones via Deep Learning. In *Assessing and Improving an Evaluation*

*Corresponding author
†Corresponding author

TOSEM, ,

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Dataset for Detecting Semantic Code Clones via Deep Learning. ACM, New York, NY, USA, 25 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Code clones [31] (in short as clones in the rest of this article) are similar code fragments 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, 33] 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 fragment that contains faults leads to fault propagation.

To better detect and manage clones, researchers have established a common taxonomy to group clones into multiple types [2, 31], 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 fragments with similar functionalities but with different implementations. To clarify the differences between Type-III and Type-IV clones, previous work [40] 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, 31]. Semantic clones are difficult to detect because they are quite different in implementations, not amenable for detection based on lexical and syntactical information [43, 47].

Since the emergence of clones as a research field, various traditional approaches [5, 15, 17, 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 fragments. For example, SourcererCC [36] treats the source code under analysis as tokens and compares subsequences to detect clones. SourcererCC is a typical lexicon-based approach that considers the similarity only in the lexical level of code fragments and ignores the syntactical information. Deckard [15] uses only structure information without considering the lexicon information of code fragments.

Based on a large labeled dataset such as BigCloneBench [40] (one of the most popular benchmarks), deep learning approaches [43, 44, 47, 48] are proposed to detect semantic clones in recent years. Existing research has taken great efforts on building complicated deep learning models [42, 43, 47, 48] 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 detect semantic clones effectively by learning features from big data without manually extracted features. The reason why deep learning approaches perform well on detecting semantic clones is that 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 validate 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 approaches

on the dataset is not convincing [1, 26]. For example, there exist code duplication issues (resulted from identical and similar files) in both the training and validation/test sets used to train and validate deep learning models for source code [1]. Code duplication affects all evaluation metric values, and the effects observed by end-users is often significantly worse than the effects reported by the evaluation conducted based on the dataset. Here is another example to illustrate the impact of the dataset on the model. As reported in Shen [38], some deep learning models distinguish dogs and wolves based on the surrounding background, instead of their different looks, which leads to their poor performance on new data. However, there is no study to investigate the limitations of BigCloneBench when used as the benchmark to validate 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 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 information. To validate this hypothesis, we develop an undesirable-by-design deep learning approach to detect semantic clones by utilizing only identifier information. This approach is undesirable purposely because code segments from a semantic clone pair can have quite different identifier names by definition, and detecting whether code segments are semantic clones shall be independent of how identifiers in these code segments are named. We find that even this undesirable approach can achieve high effectiveness comparable to the effectiveness by state-of-the-art approaches on BigCloneBench. Note that this undesirable approach fails to effectively detect semantic clones in OJClone [27], another major evaluation dataset popularly used by the research community. The result highlights that because the semantic clones in BigCloneBench rely on identifier information heavily, it is problematic to directly only use BigCloneBench to evaluate the effectiveness of detecting semantic clones via deep learning, calling for further improvement before being used.

To alleviate the identified issue in BigCloneBench, we abstract the identifiers in BigCloneBench to better evaluate the effectiveness of deep learning approaches (we denote the new dataset as AbsBigCloneBench) that has less reliance on identifier information. The experimental results show that the undesirable approach fails to detect semantic clones on AbsBigCloneBench effectively. However, the state-of-the-art approaches still perform well on AbsBigCloneBench by learning semantic features such as lexical and structural information from code fragments. In other words, desirably AbsBigCloneBench can help differentiate the state-of-the-art approaches and the undesirable approach in terms of their effectiveness demonstrated on the evaluation dataset. When the state-of-the-art approaches are applied, models trained with AbsBigCloneBench are desirably less dependent on identifier names than those models trained with BigCloneBench.

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 illustrate 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 identifiers are changed.

This article makes the following main contributions:

- **Validation**. We design an undesirable-by-design deep learning approach named LinearModel, which can achieve high effectiveness on BigCloneBench by utilizing only identifier information. Thus, deep learning approaches with effectiveness evaluated on BigCloneBench

may not really be effective, and researchers need to pay attention to the identifiers in BigCloneBench when using BigCloneBench for validating and comparing deep learning approaches of detecting semantic clones. We abstract the identifier names in BigCloneBench to produce AbsBigCloneBench, which can be used to better evaluate the effectiveness of deep learning approaches on the task of detecting semantic code clones. The experimental results show that AbsBigCloneBench outperforms BigCloneBench when used to validate and compare deep learning models of detecting semantic clones.

- **Training**. The models trained on AbsBigCloneBench have less reliance on identifier information. Through cross-validation between BigCloneBench and AbsBigCloneBench, we find that models trained with AbsBigCloneBench are also effective on BigCloneBench, and are more effective than models trained on BigCloneBench.

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

2 PRELIMINARY

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

2.1 Problem Definition

Given two code fragments 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 fragments $\{C_1, ..., C_n\}$ can be represented as $D = \{(C_i, C_j, y_{i,j}) | i, j \in n, i < j\}$. Our goal is to train a deep learning model to learn a function ϕ that maps any code fragment C to a feature vector $\boldsymbol{v}$ so that for any pair of code fragments (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:

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

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 fragments (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 7 steps. (1) Select Target Functionality. It selects a commonly needed functionality in open-source Java projects as its target functionality. (2) Identify Possible Implementations. The authors of BigCloneBench review Internet discussion (e.g., Stack Overflow) and API documentation (e.g., JavaDoc) to identify the common implementations of the target functionality.

(3) Create specification. BigCloneBench creates a specification of the functionality, including the minimum steps or features a snippet must realize to be a true positive of the target functionality. (4) Create Sample Snippet. After obtaining the possible implementations and a specification of the functionality in the second and third steps, BigCloneBench then creates a Sample Snippet, and uses the Sample Snippet to search for a large number of code clone fragments in the IJaDataset [13]. (5) Create Search Heuristic. This step searches for possible code fragments for the target functionality. When searching for similar code snippets, in order to avoid introducing too many dissimilar code snippets and causing a lot of artificial annotation burden, BigCloneBench identifies the keywords and source code patterns that are intrinsic to the identified implementations of the functionality. Although identifying source code patterns will also search for code fragments with dissimilar identifiers to a certain extent, another condition with similar keywords will make most of the code fragments found have similar identifiers. The keyword similarity here refers to the referenced third-party libraries and standard system library functions. (6) Build Candidate Set. After the fifth step of the search is over, the candidate possible code clone fragments are obtained. (7) Manual Tagging. Finally, the candidate code fragments are manually confirmed.

2.2.2 Typifying the clones in BigCloneBench. After manually marking whether the two code fragments are clone pairs, BigCloneBench automatically marks the clone types of the code clone pairs. Type-1 normalization includes removing comments and a strict pretty-printing. Type-2 normalization expands Type-1 normalization to include the systematic renaming of identifiers, and the replacing of literals with default values. To identify Type-3 and Type-4 clones, BigCloneBench measures the syntactical similarity of the clones using a line-based metric after full normalization which includes the removal of comments, a strict pretty-printing, the renaming of all identifiers to a common value and the change of all literal values to a common value.

Although BigCloneBench uses abstract technology to distinguish between semantic code clones (Type-4) and non-semantic code clones, this is not guaranteed that semantic code clones do not rely on identifiers. In the fifth step of the previous section, in order to avoid introducing a large number of false positives, BigCloneBench searches for the clone code based on similar keywords (e.g., class or method from the Java standard library or third party library) and source patterns. This process will lose a lot of true positives (i.e., code fragments with dissimilar identifiers), so the semantic code clones in BigCloneBench are highly dependent on identifiers.

2.3 Motivation 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 conduct identifier statistics by the Jaccard similarity coefficient to evaluate the similarity of identifiers between different functionalities of two datasets. Jaccard similarity coefficient 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 populations. 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, $|X \cap Y|$ is the size of the union of X and Y.

To evaluate the similarity of identifiers between different functionalities, we first get the Top 20 frequently used identifiers in each functionality. Then, we compute the Jaccard similarity coefficient

Table 1. Top 20 frequent used identifiers of each functionality on BigCloneBench.

Functionality	Top 20 frequently used identifiers
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 NoSuchAlgorithmException 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 ResourcesPlugin 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

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

The Jaccard similarity coefficient between different functionalities of BigCloneBench and OJClone is 0.038 and 0.469, respectively. We also compute that the Jaccard similarity coefficient between the same functionality of BigCloneBench and OJClone are 0.192 and 0.414, respectively. The Jaccard similarity coefficient of BigCloneBench indicates the huge differences in identifier usage between different functionalities in BigCloneBench. For better illustration, Table 1 shows the Top 20 frequent identifiers in each functionality. For example, programs in functionality "copy a file" usually use "file" related identifiers, but these identifiers are rarely used in other functionalities. On the other hand, the Jaccard similarity coefficient of OJClone means that about half of the frequently used identifiers are the same between different functionalities. It is about 12 times more than

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 identifiers in a functionality
5:          for c in codes do                          ▷ codes are all fragments belong to f
6:              s ← set()                              ▷ s is the set of identifiers in code
7:              for id in identifiers 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 identifiers
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
```

that of BigCloneBench, i.e., 0.038. The difference indicates that the identifier similarity between different functionalities in OJClone is much higher than it in BigCloneBench. The reason for the difference is the different construction of the two datasets. The programs of OJClone come from a programming environment where students write solutions from scratch for various problems. Therefore, even for different problems, programmers may define the same identifiers. For example, students typically define the loop variable as i when implementing a single-layer for loop, and define multiple loop variables as j, k, etc. when implementing a multi-layer for loop. We count the proportion of tokens used for Jaccard similarity calculation in the OJClone dataset, and the result shows that variable names account for 93%. Analyzing the reason why variable names account for such a high proportion of identifiers, we believe the reason is that the OJClone dataset is written in C language, so the only function calls involved are system functions and custom functions whose proportion is very small. However, BigCloneBench is mined from a large open-source repository where identifiers are more related to specific functionality. Thus, identifiers of code fragments that do not belong to the same functionality are quite dissimilar.

Based on this finding, we hypothesize that deep learning approaches can achieve high metric values on BigCloneBench by considering only the identifier information.

2.3.2 An example of semantic clone pair with different identifier in real-world. The previous section analyzes in detail that the semantic code clones in the BigCloneBench are highly dependent on identifiers. Much existing work has pointed out that semantic code clones are difficult to detect because semantic code clones are very different in lexical implementation. Figure 1 shows an example of semantic clone pairs from Stack Overflow post "Remove digits from a number in Java".

3 UNDESIRABLE-BY-DESIGN APPROACH

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

```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. 1. Semantic clone pair form Stack Overflow post "Remove digits from a number in Java"

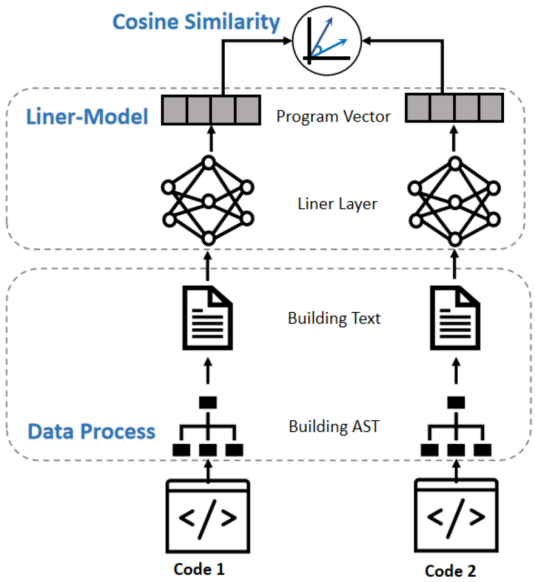


Fig. 2. Overall Architecture

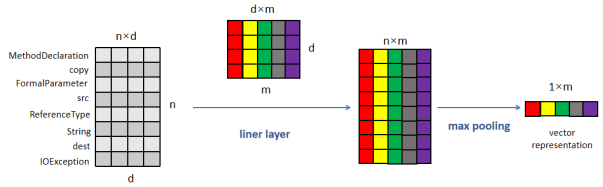


Fig. 3. Linear-Model

3.1 Approach Overview

Figure 2 shows the architecture of undesirable-by-design approach named Linear-Model. To process a code fragment, we first use Javalang [1] to parse code fragments into ASTs, as shown in Figure 4. Then we traverse all the nodes in the AST and remove the duplicates, here we use Depth-First-Search(the way to travel AST not affect the travel result). Finally, the text we get is "MethodDeclaration copy FormalParameter src ReferenceType String dest IOException". Baselines

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

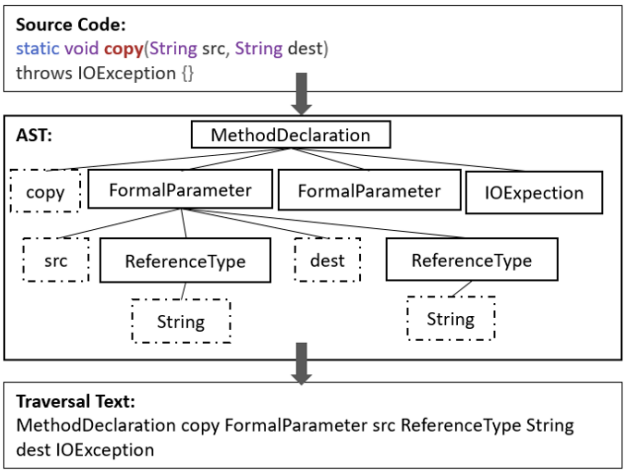


Fig. 4. The example of a code fragment, the AST of the code fragment, the traversal text of the AST.

use program information from AST nodes, including structure information and code token information(represented as terminal nodes, the terminal nodes are identified with dashed-line boxes). Therefore, for Linear-Model, we also leverage traversed texts from ASTs to get the same nodes information as baselines, but we only use the less AST's structure information for Linear-Model. The detail explanation shows in the next section. Then we use PACE (proposed by Yu et al. [47]) to initialize the embeddings of Text tokens before feeding the vectorized Text into One Node Convolution with a max-pooling layer To detect code clones, Linear-Model uses two neural networks in parallel to process a pair of code fragments at the same time. The two neural networks share the same parameters. Linear-Model takes the output of the max-pooling layer as the feature vector of a code fragment and calculates the cosine similarity of the two vectors. At last, the neural networks are trained through gradient descent back-propagation to minimize the MSE loss function:

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

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.

3.2 Linear-Model and Max Pooling

Figure 3 shows the Linear-Model, where there are n tokens in the text and the encoding length of each token is d, using a $d \times m$ matrix for the linear operation, a $n \times m$ matrix is finally obtained. After the linear operation, the number of tokens in the text is the same as before, and then max pooling is applied to each dimension in the output vectors; each dimension corresponds to one feature detector, thereby reducing a text of any tokens to a $1 \times m$ vector. Linear-Model only uses less structural information as Linear-Model only know which statements (e.g., ForStatement and IfStatement) appear in source code, but Linear-Model does not know the relationship between these statements. In addition, Linear-Model does not consider the sequential information of source code as it only tries to see which tokens appear in the source code. Formally, there is 1 matrix with text representation W_{text}, where $W_{text} \in \mathbb{R}^{m \times d}$, m and n respectively represent the number of tokens in the text and the vector dimension of each token, then the output of Linear-Model is

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

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 only uses one simple linear layer. It does not use a hidden layer to capture the hidden features.

4 EXPERIMENTAL STUDIES

In this section, we describe deep learning approaches and datasets that we used in our experimental studies. According to Yu et al. [47] and Wei et al. [43], more than 98% of clone pairs in BigCloneBench [40] are semantic clones, and the data in the OJClone dataset [27] belong to at least Type-III clones. Both datasets are widely used in semantic code clone detection via deep learning [43, 44, 47, 48]. Therefore, we conduct experiments on BigCloneBench and OJClone to explore whether and why "effective" approaches based on deep learning verified on BigCloneBench are not really effective.

We conduct experiments to answer the following research questions:

- RQ1: Can the undesirable-by-design approach Linear-Model achieve high metrics on BigCloneBench by utilizing only identifiers information?
- RQ2: How effective are deep learning approaches on AbsBigCloneBench?
- RQ3: How effective are deep learning approaches on cross experiment?

We address RQ1 to point out that for the task of detecting semantic code clones, the deep learning based approaches verified effective on BigCloneBench may not be really effective. We then address RQ2 to get an improved evaluation dataset AbsBigCloneBench that is derived from BigCloneBench. Experiment results show that AbsBigCloneBench is more desirable for differentiating the 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 Deep learning approaches

In this article, we select three state-of-the-art deep learning approaches for semantic clone detection, including ASTNN [48], TBCCD [47], and FA [42]. In addition, to illustrate the impact of identifier names in datasets, we design an undesirable-by-design deep learning approach Linear-Model to detect semantic clones by utilizing only identifier information. Linear-Model is detailed in the third Section 3.

4.1.1 ASTNN. ASTNN [48] 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 syntactical knowledge of statements. Then, it applies a bidirectional RNN to produce the representation of the entire code snippets from the representation vectors of statements.

4.1.2 TBCCD. TBCCD [47] is another recent work for code clone detection. It exploits the tree-based convolutional neural network to detect semantic clones. In addition, TBCCD proposes a position-aware character embedding technique to eliminate the impacts of unseen code tokens.

4.1.3 FA. FA [42] uses graph neural networks for code clone detection. FA first adds edges to the program's AST to form a graph. Then FA applies two different types of graph neural networks (GNN) 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 that is C language.

4.2 Datasets

We conduct experiments on two public datasets (The other four datasets are variants of these two datasets): BigCloneBench [41] and OJClone [27]. They are widely used benchmarks for clone

Table 2. Percentage of clone types in BigCloneBench.

Clone Type	T1	T2	VST3	ST3	MT3	WT3/T4
Ratio	0.455%	0.058%	0.053%	0.19%	1.014%	98.23%

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

detection [47, 48, 50]. Table 3 shows the detailed information of the two datasets we used. Besides the

4.2.1 BigCloneBench. BigCloneBench is a large code clone benchmark that is proposed by Svajlenko et al. [40]. BigCloneBench dataset is mined from IJaDataset2.0 [13] and confirmed by 3 experts. IJaDataset2.0 [13] contains a total of 25,000 subject systems and 365M LOC. BigCloneBench covers 10 functionalities and contains 6 million tagged true clone pairs and 260,000 tagged false clone pairs. Unlike the common taxonomy among researchers that groups clones into four types, BigCloneBench dataset divides Type-III clones into weak, medium, and strong. Table 2 describes the proportion of each type in BigCloneBench dataset. 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 dataset to validate the effectiveness on semantic clone detection.

We use the BigCloneBench dataset to verify our RQ1.

4.2.2 OJClone. OJClone[2] is proposed by Mou et al. [27]. OJClone is derived from programming assignments submitted by students. It is originally used for code classification tasks. Later, CDLH [43], TBCCD [47], and ASTNN [48] use OJClone dataset for code clone tasks. As code fragments submitted by students while solving a specific problem share similar functionality, true clone pairs in OJClone are at least Type-III clones. Following existing approaches, we also select the first 15 problems of OJClone dataset in which each problem contains 500 solutions. In OJClone, two different code solutions solving the same programming problem are considered as a true clone pair; otherwise, they are considered as a false clone pair.

OJClone is another widely used benchmark to verify the effectiveness of semantic code clone detection problems. Compared with other state-of-the-art approaches on the OJClone dataset, the undesirable-by-design approach Linear-Model proposed in this article is obviously invalid. On the BigCloneBench it has the same effect as other state-of-the-art approaches. Compared with BigCloneBench, OJClone can effectively verify the effectiveness of the semantic code cloning detection method based on deep learning. In agreement with the experimental results, the Jaccard similarity of the identifiers between the different problems of the OJClone we analyzed shows that the OJClone is less dependent on the identifier.

We use the OJClone dataset to verify our RQ1.

4.2.3 AbsBigCloneBench and AbsOJClone. AbsBigCloneBench and AbsOJClone are abstracted from BigCloneBench and OJClone, respectively. They abstract part-identifier information in BigCloneBench and OJClone, such as class names and variable names, and retain other tokens such as

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

operators, basic types, and member variables. The abstraction of AbsBigCloneBench and AbsOJClone will not change the labeling of semantic code clones and non-semantic code clones in the BigCloneBench and OJClone.

We use AbsBigCloneBench and AbsOJClone to verify our RQ2 and RQ3. We aim to illustrate that AbsBigCloneBench can assist BigCloneBench to more effectively verify the effectiveness of the semantic code clone detection approaches based on deep learning, and the model trained on the AbsBigCloneBench dataset is also effective on BigCloneBench.

4.2.4 Violent-AbsBigCloneBench and Violent-AbsOJClone. Violent-AbsBigCloneBench is mentioned in section 5.2.1. Like Mou et al. [27], Yu et al. [47], this article also explores the effectiveness of each approach when only non-terminal AST nodes are retained. Figure 4 shows the terminal node is identified with dashed-line boxes. Violent-AbsBigCloneBench and Violent-AbsOJClone only remain the token in real-line boxed. This article does not emphasize Violent-AbsBigCloneBench and Violent-AbsOJClone because the abstraction of Violent-AbsBigCloneBench and Violent-AbsOJClone will change the labeling of semantic code clones and non-semantic code clones in the BigCloneBench.

We only use Violent-AbsBigCloneBench and Violent-AbsOJClone to verify that on such a violent abstraction dataset, the state-of-the-art approaches are more effective than Linear-Model (Table 5).

4.3 Experimental Setting

For BigCloneBench dataset, we use the same data as CDLH [43] and TBCCD [47] that has 9,134 code fragments totally. 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 fragments to construct the test set, 913 code fragments to construct the validation set, and the remaining 7,308 code fragments to construct the training set. Finally, we get $\frac{913\times912}{2} = 416,328$ pairs in the test and validation set 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 thousand pairs as the training set.

OJClone has a total of 7,500 code fragments. We also divide the training set, validation set, and test set according to the ratio of 8: 1: 1. We randomly select 750 code fragments to construct the test set and 750 code fragments to construct the validation set. The remaining 6,000 code fragments are used to construct the training set. We have $\frac{750\times749}{2} = 280,875$ pairs in the test set and the validation set, 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 thousand pairs as the training set.

The parameters for training Linear-Model are as follows: m is 100; we train the model for ten epochs and use the SGD optimizer with batch size 1. The threshold of our approach for prediction is determined with the validation set. We use Tensorflow[3] to implement Linear-Model. We replicate ASTNN, TBCCD, and FA by running the source code they provide.

We will public our source code and dataset, and we also plan to turn our data into archived open data in case of acceptance.

5 RESULTS

5.1 RQ1: Can the undesirable-by-design approach Linear-Model achieve high metrics on BigCloneBench by utilizing only identifiers information?

5.1.1 Precision and Recall on BigCloneBench and OJClone. To validate the hypothesis mentioned above, we compare Linear-Model that is undesirable purposely, with state-of-the-art deep learning approaches on semantic clone detection, i.e., ASTNN, TBCCD, and FA.

[3]http://www.tensorflow.org

Table 4. Comparison of different approaches on BigCloneBench and OJClone. P and R represent Precision and Recall respectively. (The FA approach is customized for Java language and cannot used to detect C programming clones, thus its results on OJClone are missing.)

Approaches	BigCloneBench			OJClone		
	P	R	F1	P	R	F1
ASTNN	0.95	0.93	0.94	0.98	0.98	0.98
TBCCD	0.94	0.96	0.95	0.99	0.99	0.99
FA	0.95	0.95	0.95	-	-	-
Linear-Model	0.96	0.92	0.94	0.68	0.67	0.68

Table 4 shows the Precision and Recall of different approaches on BigCloneBench and OJClone. We also give the overall score F1 of Precision and Recall. Recall means the fraction of similar method pairs retrieved by an approach. Precision means the fraction of retrieved truly similar method pairs in the results reported by an approach.

Table 4 illustrates that Linear-Model outperforms other approaches in terms of precision on BigCloneBench. When it comes to Recall, the TBCCD achieves the best result, and Linear-Model is slightly worse than other approaches. The F1 score indicates that Linear-Model can achieve equivalent performance to other approaches. Both Linear-Model and ASTNN have achieved 0.94 F1 score on BigCloneBench dataset. We can see that even this undesirable approach can achieve high effectiveness comparable to the effectiveness by the 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 fails to detect semantic clones on OJClone effectively. Desirable approaches(e.g., TBCCD, ASTNN, FA) reported by existing literature still perform well on both BigCloneBench and OJClone datasets. According to the analysis in their papers, these approaches can detect semantic clones by capturing both the structure information and lexical information in the source code. However, Linear-Model can detect clones in BigCloneBench effectively, whereas all metrics are worsened on OJClone dramatically, the F1 score of Linear-Model is 0.68 that is about 30% lower than that on BigCloneBench. Due to the great differences between top used identifiers in different functionalities of BigCloneBench, Linear-Model can achieve equivalent performance as other approaches by learning these identifiers. For OJClone, it is difficult for Linear-Model to learn the similarity between programs well by learning the identifiers in source code, thus making it fails to detect semantic clones. However, state-of-the-art approaches can learn other features, e.g., lexical and structural features in programs.

5.1.2 Researchers need to pay attention to the identifier name in BigCloneBench when using BigCloneBench for validating and comparing deep learning approaches for detecting semantic clones. We use an undesirable-by-design deep learning approach Linear-Model to detect semantic clones by utilizing only identifier information. This approach is undesirable because code segments from a semantic clone pair can have quite different identifier names by definition, and detecting whether code segments are semantic clones can be independent of how identifiers in these code segments are named. Therefore, it is unreasonable to use Linear-Model to detect semantic clones due to Linear-Model detect semantic clones by only looking at which tokens appear in the code snippets. As Table 4 shows, the huge difference of identifier names between different functionalities in BigCloneBench makes Linear-Model achieve high performance on semantic clone detection. However, when the difference disappears (e.g., OJClone), Linear-Model fails to detect semantic clones.

When researchers use the BigCloneBench dataset to verify the effectiveness of their approach, they should pay attention to mitigating the impact of identifiers. In the next section, this article

Table 5. Comparison of different approaches on Violent-AbsBigCloneBench and Violent-AbsOJClone.

Approach	Violent-AbsBigCloneBench			Violent-AbsOJClone		
	Precision	Recall	F1	Precision	Recall	F1
ASTNN	0.83	0.67	0.74	0.94	0.88	0.91
TBCCD	0.75	0.63	0.69	0.95	0.95	0.95
FA	0.78	0.54	0.64	-	-	-
Linear-Model	0.46	0.50	0.48	0.22	0.36	0.28

abstracts part-identifier in BigCloneBench to better verify the effectiveness of the semantic code clone detection approach. We will introduce it in detail in the next section. In summary, we find that:

Finding 1. An undesirable approach Linear-Model can achieve high effectiveness from identifiers on BigCloneBench but fails to detect semantic clones in OJClone effectively. It is problematic to directly use BigCloneBench to validate the effectiveness of detecting semantic clones via deep learning, calling for further improvement before being used.

5.2 RQ2: How effective are deep learning approaches on AbsBigCloneBench?

To alleviate the issue in BigCloneBench discussed in RQ1, in this section, we abstract the part-identifier names in BigCloneBench to better validate the effectiveness of deep learning approaches. We denote the dataset after abstracting identifiers as AbsBigCloneBench, such that the used identifiers between code segments with different functionalities in the AbsBigCloneBench are much more similar. This section first introduces how we modify BigCloneBench to get AbsBigCloneBench. Then, we compare the recall, precision, and F1 values of Linear-Model and other state-of-the-art approaches on AbsBigCloneBench. Finally, we discuss why AbsBigCloneBench is more desirable than BigCloneBench on deep learning approaches validation. The experimental results and Jaccard similarity coefficient prove that AbsBigCloneBench can better validate the effectiveness of deep learning approaches than BigCloneBench.

In addition to abstracting BigCloneBench, we also abstract OJClone to get AbsOJClone. The experimental results prove that the AbsOJClone can still verify the effectiveness of the approaches of semantic code clone detection.

5.2.1 How we modify BigCloneBench to get AbsBigCloneBench. Like Mou et al. [27], Yu et al. [47], to further explore the impacts of code tokens on clone detection, we take an experiment on the AST without token information. The token information is contained in the terminal nodes of the AST. For baseline approaches based on AST directly, we remove all terminal nodes from the ASTs partial ASTs are fed into these models. For Linear-Model, we apply the traversed text from partial ASTs as input to detect semantic code clones. Figure 4 shows the terminal node is identified with dashed-line boxes. For the without code token experiment, the AST used for baseline approaches with only real-line boxed, and the text used for Linear-Model is "MethodDeclaration, FormalParameter, ReferenceType, IOException". As all the code token information is removed, the ASTs mainly consist of structure information, the texts traversed from ASTs only consist of weak structure information.

Table 5 illustrate that when more information in the dataset is violently removed (all terminal information is removed), the existing state-of-the-art approaches are more effective than Linear-Model. After removing the token information in the AST, baselines can still achieve significantly

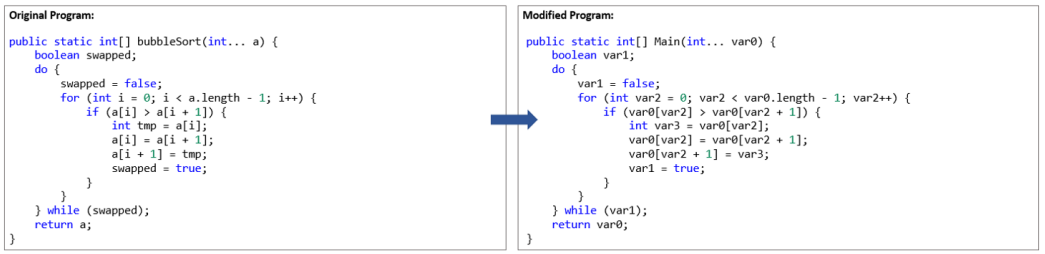


Fig. 5. An example of code fragment in BigCloneBench and AbsBigCloneBench

Table 6. Comparison of different approaches on AbsBigCloneBench and AbsOJClone. ΔF1 score indicates the relative improvement (worsening) compared to F1 scores on BigCloneBench and OJClone in Table 4

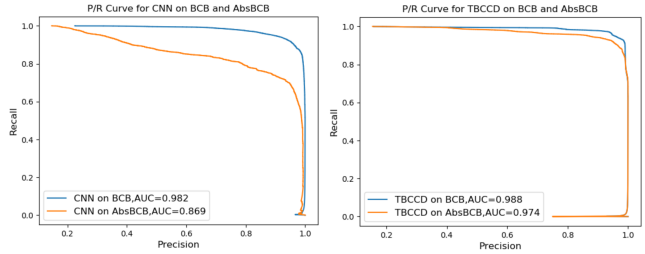
Approach	AbsBigCloneBench				AbsOJClone			
	Precision	Recall	F1	ΔF1	Precision	Recall	F1	ΔF1
ASTNN	0.93	0.92	0.93	-0.01	0.96	0.96	0.96	-0.01
TBCCD	0.95	0.93	0.94	-0.01	0.96	0.97	0.97	-0.02
FA	0.96	0.94	0.95	0	-	-	-	
Linear-Model	0.85	0.76	0.81	-0.13	0.64	0.59	0.61	-0.07

outperforms 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), it will lose a lot of semantic information in the program. Considering this phenomenon and the common taxonomy of clone types [2, 31, 34], we abstract part-identifier names in the source code and build AbsBigCloneBench to better validate the effectiveness of deep learning approaches. In particular, we abstract identifier names, including class names and variable names in the code fragments to obtain AbsBigCloneBench. The other tokens, such as operators, basic types, and member variables, are kept as original ones. API invocations are essential information for implementing the functionality of programs. Therefore, we keep these API invocations in AbsBigCloneBench. The abstraction technology is similar to Harer et al. [24], Li et al. [9], we discuss the abstraction technology used by Harer et al. [24], Li et al. [9] in Section 6. Finally, this article abstracts the variable names, class names in the code snippets. We use the same abstraction technology to abstract OJClone.

The Jaccard similarity coefficients of the identifiers of the BigCloneBench and the AbsBigCloneBench are 0.038 and 0.484, respectively, indicating that AbsBigCloneBench is less dependent on identifiers than the BigCloneBench. The Jaccard similarity coefficient of identifiers of the OJClone and the AbsOJClone are 0.469 and 0.472, respectively, indicating that OJClone and AbsOJClone are all weakly dependent on identifiers.

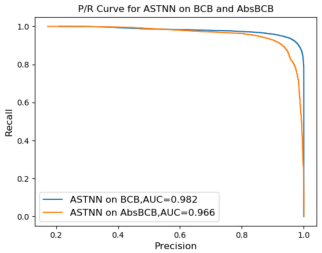
5.2.2 Recall, Precision, and F1 values of different approaches on AbsBigCloneBench. Similar to RQ1, we compare different approaches on AbsBigCloneBench to investigate whether AbsBigCloneBench can be used as an evaluation dataset to validate the effectiveness of deep learning approaches on semantic clone detection. The result is shown in Table 6 in which ΔF1 is the relative improvement (worsening) compared to the F1 on BigCloneBench in Table 4.

Table 6 illustrates a significant decrease in both the precision and the recall of Linear-Model compared to other approaches on AbsBigCloneBench and AbsOJClone. Compared to the results

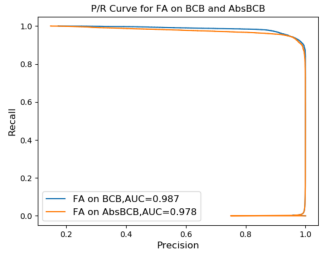


(a) PR-Linear-Model

(b) PR-TBCCD



(c) PR-ASTNN



(d) PR-FA

Fig. 6. PR curves and AUC of different approaches on BigCloneBench and AbsBigCloneBench

shown in Table 4, state-of-the-art approaches can achieve equivalent performance on the AbsBigCloneBench. Although part-identifier names are abstracted, they can perform well by learning other semantic features such as the structural information from code fragments. However, it is limited for Linear-Model to learn other features while detecting semantic clones on the AbsBigCloneBench. As we abstract part-identifier names, including variable names and class names, used identifiers between code segments with different functionalities in AbsBigCloneBench are much more similar. Therefore, Linear-Model will recognize some false clone pairs in AbsBigCloneBench as true clone pairs. It causes the precision decrease on the AbsBigCloneBench. On the other hand, it is difficult for Linear-Model to learn features in true clone pairs, so the recall also drops significantly.

We also compare Precision-Recall (PR) curves and AUC values of different approaches on BigCloneBench and AbsBigCloneBench, respectively. As shown in Figure 6, the AUC value of Linear-Model on the AbsBigCloneBench is significantly lower than that on BigCloneBench. AUC values of other approaches on the AbsBigCloneBench are almost the same as those on BigCloneBench. It further proves that biases introduced from part-identifier names can be reduced by abstracting them.

Table 6 also illustrates the performance of the approaches is the same on AbsOJClone and OJClone. Both AbsOJClone and OJClone can effectively verify the effectiveness of the approaches of semantic code clone detection.

5.2.3 Why AbsBigCloneBench is more desirable than BigCloneBench on deep learning approaches validation? AbsBigCloneBench is derived from BigCloneBench by abstracting part-identifier names, including variable names and class names. These part-identifier names can totally different according to the definition of semantic clones. As semantic clones are functionally similar, we keep API invocations that are essential for implementing functionalities of programs. After abstracting part-identifier names, the Jaccard similarity coefficient of AbsBigCloneBench improves to 0.484 from

0.038. The great improvement demonstrates the difference of identifier usage between different functionalities decreases.

To prove whether AbsBigCloneBench can validate different deep learning approaches, we first compare Linear-Model with other approaches on AbsBigCloneBench. Table 6 illustrates that state-of-the-art approaches still perform well on AbsBigCloneBench, whereas the undesirable-by-design model Linear-Model has a significant performance decline. In addition, instead of achieving equivalent performance to other approaches on BigCloneBench, Linear-Model is 0.12-0.14 lower than others in terms of F1. The Precision-Recall curve and AUC value shown in Figure 6 also prove Linear-Model cannot detect semantic clones well on AbsBigCloneBench. The experimental results prove that AbsBigCloneBench can better differentiate the 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 found that:

Finding 2. Desirably AbsBigCloneBench can help differentiate the state-of-the-art approaches and the undesirable approach in terms of their effectiveness demonstrated on the evaluation dataset. AbsBigCloneBench can help BigCloneBench better verify the effectiveness of semantic code clone detection approach.

5.2.4 Three practical assessment techniques. Table 4 and Table 6 show that AbsBigCloneBench, OJClone and AbsOJClone can all verify the effectiveness of the model of semantic code clone detection. This section proposes three practical assessment techniques to check weather a dataset can be used for validating the effectiveness of models.

- **Property 1: Preserving effectiveness ranking of models across part-identifier abstraction**. The relative effectiveness ranking of the given models trained and validated/compared against the given dataset shall be the same or similar as the ranking of the given models trained and validated/compared against the part-identifier abstraction dataset.
- **Property 2: Preserving model effectiveness validated across part-identifier abstraction**. The effectiveness of each given model trained and validated against the original dataset shall be the same or similar as the effectiveness of the model trained on the given dataset but validated against the part-identifier abstraction dataset.
- **Property 3: Achieving ineffectiveness with the undesirable model trained on only identifiers** The undesirable model trained with a simple linear model upon only the identifier in the given dataset shall be ineffective when validated against the given dataset, performing worse than the given models trained and validated on the given dataset.

5.3 RQ3: How effective of deep learning approaches on cross experiment?

This section will conduct cross-validation to explore weather the models trained on AbsBigCloneBench are also effective on BigCloneBench. 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. The experiment aims to explore whether models trained with BigCloneBench (AbsBigCloneBench) are also effective on AbsBigCloneBench (BigCloneBench). As Table 7 shows, values in the upper part of the table represent the results of models trained on BigCloneBench and tested on AbsBigCloneBench. At the same time, values in the lower part of the table represent the results of models trained on AbsBigCloneBench and tested on BigCloneBench. Δ F1 score indicates the relative improvement (worsening) compared to F1 scores in Table 4 and Table 6.

Table 7. Comparison of different approaches on cross experiment in BigCloneBench. Δ F1 score indicates the relative improvement (worsening) compared to F1 scores in Table 4 and Table 6.

Approach	Precision	Recall	F1	Δ F1
Train: BigCloneBench Test: AbsBigCloneBench				
ASTNN	0.88	0.68	0.77	-0.17
TBCCD	0.66	0.74	0.70	-0.25
FA	0.89	0.60	0.71	-0.24
Linear-Model	0.56	0.72	0.63	-0.31
Train: AbsBigCloneBench Test: BigCloneBench				
ASTNN	0.97	0.84	0.90	-0.03
TBCCD	0.95	0.88	0.92	-0.02
FA	0.93	0.92	0.91	-0.04
Linear-Model	0.76	0.66	0.70	-0.11

Table 8. Comparison of different approaches on cross experiment in OJClone. Δ F1 score indicates the relative improvement (worsening) compared to F1 scores in Table 4 and Table 6.

Approach	Precision	Recall	F1	Δ F1
Train: OJClone Test: AbsOJClone				
ASTNN	0.97	0.96	0.97	-0.01
TBCCD	0.98	0.97	0.98	-0.01
FA	-	-	-	-
Linear-Model	0.53	0.42	0.47	-0.19
Train: AbsOJClone Test: OJClone				
ASTNN	0.98	0.97	0.98	0
TBCCD	0.99	0.99	0.99	0
FA	-	-	-	-
Linear-Model	0.55	0.45	0.49	-0.12

5.3.1 Different models trained on BigCloneBench and test on AbsBigCloneBench. The F1 score and Δ F1 score (i.e., scores in the upper part) show that approaches trained on BigCloneBench do not work well on AbsBigCloneBench. It indicates that these models are not effective on another dataset with other identifier names, e.g., variable names and class names. They fail to detect semantic clones if identifier names are changed. As they heavily rely on the identifier names to infer functionalities while detecting semantic clones. The reason for models trained on BigCloneBench do not work on AbsBigCloneBench is models trained on BigCloneBench can achieve convergence easily by using the identifier information. It is limited for deep learning approaches (e.g., TBCCD) to learn the lexical and structural information well. Therefore, when identifier names in the test set are different from the training set, they fail to detect clones effectively. In conclusion, we find that:

Finding 3. Models trained on BigCloneBench are not effective on AbsBigCloneBench, a reasonable dataset for semantic code clone detection.

5.3.2 Different models trained on AbsBigCloneBench and test on BigCloneBench. As Table 7 shows, models(except Linear-Model) trained on AbsBigCloneBench still perform well on BigCloneBench.

From Δ F1 scores of models trained on AbsBigCloneBench, we find that the results of Linear-Model have a significant decline, i.e., 0.11 in terms of Δ F1 score compared to other approaches (i.e., 0.03, 0.02, and 0.04). As Linear-Model captures features from identifiers, it decreases when abstracting the class names and variable names. However, other approaches only show a slight decrease. They mainly rely on lexical and structural information to learn functional similarity between semantic clone pairs after abstracting identifier names. Their performance would not be affected when applied to other datasets with different identifier names.

In conclusion, we find that:

> **Finding 4.** Models trained on AbsBigCloneBench are also effective on the real-world dataset(i.e., BigCloneBench).

5.3.3 Cross experiment for OJClone. We do the same cross experiment on the OJClone and AbsOJClone. As Table 8 shows, except for Linear-Model, the model trained on OJClone is valid on AbsOJClone, and the model trained on AbsOJClone is also valid on OJClone. As analyzed in the previous section, since the OJClone does not have the problem of identifier dependence, the cross-experiment results of the models on the OJClone and AbsOJClone are almost the same.

6 RELATED WORK

6.1 Identifier Normalization in Software Engineering

Abstract identifier technology has been applied in many approaches based on deep learning for the tasks of software engineering. In the task of vulnerability detection, Li et al. [24] use abstract identifier technology to reduce their approach's dependence on identifiers. They first remove the comment information in the code snippets, then abstract away the variables in the program, and finally abstract away the custom function names in the program. In addition to abstracting variables and function names in programs, Harer et al. [9] also abstract constants in code fragments. They abstract variable names and function names to the same generic identifier, but each unique variable name within a single function gets a separate index (for the purposes of keeping track of where variables re-appear). Kim et al. [18] abstract formal parameters, local variables, data types, and function calls to detect Vulnerabilities. They normalize the function body by removing the comments, whitespaces, tabs, and line feed characters, and by converting all characters into lowercase.

For the task of code completion, Cummins et al. [6] synthesize automatically a large number of OpenCL benchmarks by learning a character-level LSTM over valid OpenCL code. Their goal is to generate reasonable-looking code rather than to synthesize a program that complies with a specification. To ease their task, they normalize the code by consistently alpha-renaming variables and method names. Bhoopchand et al. [3] use a token sparse pointer-based neural model of Python that learns to copy recently declared identifiers to capture very long-range dependencies of identifiers, outperforming standard LSTM models. They normalize identifiers before feeding the resulting token stream to their models. That is, they replace every identifier name with an anonymous identifier indicating the identifier group (class, variable, argument, attribute or function) concatenated with a random number that makes the identifier unique in its scope.

In the field of code clone detection, there are also many approaches that pay attention to the abstraction of identifiers. For example, CK and JR. [5] propose to use flexible code normalization, which is not simply limited to global replacement, for example of all identifiers and literals, or simple abstraction, for example of loop bodies. They can choose to normalize the specific parts that they expect to vary. Kamiya et al. [17] have designed several transformation rules for *c++* code and *java* code to abstract identifiers. For example, "Remove namespace attribution, Remove template parameters, Remove initialization lists" for *c++* code, and "Remove package names,

Supplement callees, Separate class definitions" for *java* code. However, they are all different from our application scenarios. First, they are not based on deep learning approaches. Second, they focus on improving the effectiveness of code clone detection through abstract identifiers, while our focus is on verifying the effectiveness of deep learning models.

6.2 Code Clone Detection

Code clone detection is a significant research problem in the field of software engineering. It helps reduce the cost of software maintenance and prevent faults. Clones are similar code fragments 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, 31, 34]. Type I-III clones are clone pairs that differ at the token and statement levels. Type-I clones are identical code fragments in addition to variations in comments and layout. Apart from Type-I clones, Type-II clones are identical code fragments except for different identifier names and literal values. Apart from Type-II clones, Type-III clones are syntactically similar code fragments that differ at the statement level. Apart from Type-III clones, Type-IV clones are functionally-similar code fragments with different implementations. The difference in identifier does not affect the Type II-IV clones.

Many approaches have been proposed to detect clones. These approaches mainly measure the similarity between code representations varying from lexical [7, 17, 28, 32, 35, 36], syntactical [8, 15, 16, 46], and graph-based [19, 21, 25, 29] representations. CCFinder [17] and SourcererCC [36] detect clones according to the comparison of tokens. Deckard [15] detects clones by comparing the syntactical similarity between the ASTs.

Deep learning approaches [23, 45, 47, 48, 50] for clone detection have recently received great attention. White et al. [45] propose to learn latent features for source codes via a recursive auto-encoder over ASTs. To detect Type-IV clones effectively, Wei and Li [43] formulate code clone detection as a supervised learning task to hash problem. They propose an end-to-end deep learning approach named CDLH to learn hash codes from lexical and syntactical information. Zhao and Huang [50] propose DeepSim to measure the functional similarity. It encodes the program's control flow graph and data flow graphs into matrices and then uses the matrices to train a deep learning model to compute the similarity between two functions. Yu et al. [47] propose TBCCD to detect semantic clones using tree-based convolutional neural networks capturing both the structural and lexical information of code fragments. ASTNN proposed by Zhang et al. [48] is another recent approach that learns small statement trees split from the entire trees.

6.3 Learning from Big Code

Large open-source software systems have been arisen and ubiquitous in recent years [49]. They provide billions of code tokens for various software engineering tasks, such as code clone detection [42, 47], 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 deal with software engineering tasks. Research in the "Big Code" area relies on large corpora of code. However, a few studies address issues with the "Big Code". Allamanis [1] describes the impact of code duplication on deep learning approaches. They find that code duplication can cause the evaluation to overestimate deep learning approaches. LeClair et al. [22] address the problem in summarizing source code by using a project dataset.

7 DISCUSSION

7.1 Implications

7.1.1 Verify the effectiveness of the approach on semantic code clone detection. In this article, we point out an important issue in BigCloneBench, which is widely used in code clone detection; this issue has not been noticed by other code clone detection researchers. We noticed that many approaches based on deep learning use BigCloneBench to validate the effectiveness of their model, but they did not notice that the undesirable-by-design approach (Linear-Model) can still achieve high performance on BigCloneBench due to BigCloneBench very large dependence on identifier information. We not only point out this problem that has not been noticed by researchers before but also propose to alleviate this problem through abstract identifiers. The experimental results also prove that after abstracting identifier names, the AbsBigCloneBench can validate the effectiveness of the semantic code clone detection approach based on deep learning, and the models trained on AbsBigCloneBench are also effective on BigCloneBench.

7.1.1 Verify the effectiveness of the approach on semantic code clone detection. In addition to being able to better assist the BigCloneBench dataset to verify the effectiveness of the semantic code clone detection approaches, abstract identifiers also have meaningful application scenarios for models trained on datasets that do not rely on identifiers. For example, for the task of malware and vulnerable detection, models need to learn the behavior of the malware and vulnerable, and only using identifier information is not enough. To assist programming learners to search more different implementations can be another application scenario.

7.2 Limitations

The BigCloneBench is mined from real-world projects. In many cases in real-world projects, the identifier names in the code fragments that belong to the same semantic clones will be similar. If we abstract the identifier in BigCloneBench, it will be different from the real-world scenario, and it will also increase the difficulty of semantic code clone detection. Here we argue that if the model is verified on AbsBigCloneBench with the identifier abstracted away, the model will still be effective in the real-world project(i.e., BigCloneBench). From the verification of semantic code clone detection, it is also more reasonable to choose the AbsBigCloneBench.

7.3 Why do we choose Linear-Model to compare with other deep learning approaches?

To point out the negative impact of identifier names on semantic clone detection via deep learning, we develop an undesirable-by-design approach Linear-Model. 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 only tries to see which tokens appear in the source code. Compared to Linear-Model, other deep learning approaches deeply capture the semantic information of the program while detecting semantic clones. For example, TBCCD [47] captures the structural information of the code fragment and the position of the token in the AST to detect semantic clones. CDLH [43] uses AST-based LSTM to capture both lexical and syntactical information of the code fragment. ASTNN [48] divides the program AST into multiple subtrees; after obtaining the vector representation of the subtree, the semantic information of the code fragment is captured through the bidirectional GRU. We choose the undesirable Linear-Model to validate whether it can perform well using only identifier information to detect semantic clones on BigCloneBench. Linear-Model can be replaced by other deep learning approaches that utilize only identifier information.

7.4 Does this article emphasize that the identifiers of semantic clones should not be similar?

Some researchers may argue that the identifier information is meaningful for semantic code clone detection in the real-world. In this article, we are not saying that the identifiers of two code fragments belonging to the same semantic clone should not be similar; we just emphasize that the identifiers of the semantic clones can be different, the identifier in code fragment can cause confusion when validating the effectiveness of the deep learning approach. According to the common taxonomy of clone types [2, 31, 34], if two code fragments belong to semantic clones, their identifiers can differ. AbsBigCloneBench can eliminate the impact of identifier information on semantic code clone detection as much as possible, so it can better validate the deep learning model's effectiveness on tasks of semantic code clone detection.

7.5 Will traditional machine learning approaches get as good results as Linear-Model?

In this article, we find that an undesirable-by-design deep learning approach (Linear-Model) can achieve high effectiveness on BigCloneBench by utilizing only identifier information. Some researchers may question whether machine learning methods can achieve the same good results as Linear-Model? Before deep learning algorithms become popular, SVM [4, 37, 39] is one of the most popular machine learning algorithms. For the code clone detection task, Jadon [14] propose to detect code clones by the Support Vector Machine (SVM) [4, 37, 39]. We also compare the performance of non-deep learning SVM models on BigCloneBench and OJClone. We first perform lexical analysis on the code snippet to obtain the token of the code snippet. Similar to the text classification task with SVM, we stitch the two code snippets into a text and treat code clone detection as a binary classification task. The F1 value of SVM on BigCloneBench can reach 0.75, whereas 0.11 on OJClone(the F1 value of SVM on BigCloneBench can reach 0.94, whereas 0.68 on OJClone). The F1 value of SVM on BigCloneBench and OJClone is lower than Linear-Model; the reason is that SVM is still inferior to Linear-Model in terms of its ability to capture features. One of the advantages of deep learning is to capture hidden features. But through the experimental results, we can also see that consistent with Linear-Model's performance; the result shows that the SVM works on BigCloneBench, but it does not work on the OJClone.

8 CONCLUSION

In this article, we have found that the semantic clones in BigCloneBench heavily rely on identifier information. According to the definition of code clone type, the identifiers 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 the undesirable approach can achieve high effectiveness by considering only identifier name information, even comparable to the state-of-the-art deep learning approaches for detecting semantic clones. However, the undesirable approach fails to detect semantic clones in OJClone effectively. Therefore, when using BigCloneBench to verify and compare deep learning models to detect semantic clones, researchers need to pay attention to the identifier names in BigCloneBench. To alleviate the impact of identifier names in BigCloneBench, we abstract part-identifier in BigCloneBench to get a new dataset named AbsBigCloneBench. The experimental results show that AbsBigCloneBench can help differentiate the state-of-the-art approaches and the undesirable approach in terms of their effectiveness demonstrated on the evaluation dataset. In addition, models trained with AbsBigCloneBench are desirably less dependent on identifier names than those trained with BigCloneBench. Models trained with AbsBigCloneBench are also effective on BigCloneBench.

Our work inspires future research on the use of deep learning for semantic clone detection; we need to pay attention to whether using only the identifier information in the dataset can achieve good performance. For other software engineering tasks, researchers also need to pay attention to whether similar problems exist in the used benchmark.

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*. 143–153.

[2] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and Evaluation of Clone Detection Tools. *IEEE Trans. Software Eng.* 33, 9 (2007), 577–591.

[3] Avishkar Bhoopchand, Tim Rocktäschel, Earl Barr, and Sebastian Riedel. 2016. Learning python code suggestion with a sparse pointer network.. In *arXiv Preprint arXiv:1611.08307*.

[4] P. H. Chen, C. J. Lin, and B. Schölkopf. 2005. A tutorial on vsupport vector machines. In *Appl. Stoch. Models. Bus. Ind.* 111–136.

[5] Roy CK and Cordy JR. 2008. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization.. In *In: Proc. of the 16th IEEE Int'l Conf. on Program Comprehension*. 172–181.

[6] Chris Cummins, Pavlos Petoumenos, Zheng Wang, , and Hugh Leather. 2017. Synthesizing benchmarks for predictive modeling.. In *In Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*. 86–99.

[7] Selvadurai Kanmani Egambaram Kodhai. 2014. Method-level code clone detection through lwh (light weight hybrid) approach. *Software Engineering Research & Development* 2, 1 (2014), 12.

[8] Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*. IEEE Computer Society, 321–330.

[9] Jacob A. Harer, Louis Y. Kim, Rebecca L. Russell, Onur Ozdemir, Leonard R. Kosta, Akshay Rangamani, and Lei H. Hamilton. 2018. Automated software vulnerability detection with machine learning. In *arXiv Preprint arXiv:1611.08307*.

[10] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*. ACM, 200–210.

[11] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2019. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering* (2019), 1–39.

[12] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing source code with transferred api knowledge. (2018).

[13] IJaDataset2.0. January 2013. Ambient Software Evoluton Group. http://secold.org/projects/seclone

[14] S. Jadon. 2016. Code clones detection using machine learning technique: Support vector machine. In *2016 International Conference on Computing, Communication and Automation (ICCCA)*. 299–303.

[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 international conference on Software Engineering*. IEEE Computer Society, 96–105.

[16] Kunfeng Xia Jianglang Feng, Baojiang Cui. 2013. A code comparison algorithm based on ast for plagiarism detection. In *Fourth International Conference on Emerging Intelligent Data and Web Technologies (EIDWT)*. 393–397.

[17] 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.

[18] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*. 595–614. https://doi.org/10.1109/SP.2017.62

[19] R. Komondoor and S. Horwitz. 2001. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*. 40–56.

[20] Rainer Koschke, Raimar Falke, and Pierre Frenzel. 2006. Clone Detection Using Abstract Syntax Suffix Trees. In *Proceedings of the 13th Working Conference on Reverse Engineering*. 253–262.

[21] Jens Krinke. 2001. Identifying similar code with program dependence graphs. In *Proceedings. Eighth Working Conference on Reverse Engineering*. 301–309.

[22] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 795–806.

[23] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. 2017. Cclearner: A deep learning-based clone detection approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 249–260.

[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.. In *arXiv Preprint arXiv:1611.08307*.

[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*. 872–881.

[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*. ACM, 373–384.

[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 Thirtieth AAAI Conference on Artificial Intelligence* (Phoenix, Arizona) *(AAAI'16)*. AAAI Press, 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 21th international conference on Program Comprehension*. IEEE Computer Society, 93–102.

[29] Annamalai Narayanan, Mahinthan Chandramohan, Lihui Chen, Yang Liu, and Santhoshkumar Saminathan. 2016. subgraph2vec: Learning distributed representations of rooted sub-graphs from large graphs.

[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 (2018), 1–25.

[31] 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.

[32] Chanchal K 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 international conference on Program Comprehension*. IEEE Computer Society, 172–181.

[33] Chanchal K. Roy and James R. Cordy. 2009. A Mutation/Injection-Based Automatic Framework for Evaluating Code Clone Detection Tools. In *Proceedings of the 2nd International Conference on Software Testing Verification and Validation*. 157–166.

[34] C. K. Roy and J. R. Cordy. 2018. Benchmarks for software clone detection: A ten-year retrospective. In *SANER*. 26–37.

[35] Chanchal K Roy and James R Cordy. 2018. Ccaligner: a token based large-gap clone detector. In *Proceedings of the 40th International Conference on Software Engineering*. IEEE Computer Society, 1066–1077.

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

[37] V. D. Sanchez. 2003. Advanced support vector machines and kernel methods. In *Stat. Comput.* 5–20.

[38] Xiangyang Shen. 2020. From Deep Learning to Deep Understanding. In *Grand Challenges and Opportunities*.

[39] A. J. Smola and B. Schölkopf. 2004. A tutorial on support vector regression. In *Stat. Comput.* 199–222.

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

[41] Jeffrey Svajlenko and Chanchal K Roy. 2015. Evaluating clone detection tools with bigclonebench. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 131–140.

[42] 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 International Conference on Software Analysis, Evolution, and Reengineering*. IEEE Press.

[43] Huihui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code.. In *IJCAI*. 3034–3040.

[44] Huihui Wei and Ming Li. 2018. Positive and Unlabeled Learning for Detecting Software Functional Clones with Adversarial Training.. In *IJCAI*. 2840–2846.

[45] 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*. ACM, 87–98.

[46] Z. Wang Y. Gao, S. Liu, and W L. Yang. 2019. A Tree Embedding Approach for Code Clone Detection. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (Cleveland, OH, USA). 145–156.

[47] 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 International Conference on Program Comprehension*. IEEE Press, 70–80.

[48] 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 International Conference on Software Engineering*. IEEE Press, 783–794.

[49] Yuxia Zhang, Minghui Zhou, Audris Mockus, and Zhi Jin. 2019. Companies' Participation in OSS Development-An Empirical Study of OpenStack. *IEEE Transactions on Software Engineering* (2019).

[50] Gang Zhao and Jeff Huang. 2018. Deepsim: deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 141–151.