May 6th, 2021

Dear Prof. Pezze and Reviewers,

Thank you for your helpful and insightful comments! We have greatly benefited from all the comments and suggestions. In revising this paper, we have conducted additional experiments and make major changes to address the comments and suggestions raised by each reviewer. Here is a summary of the major changes:

1. We apply the abstraction technology for BigCloneBench to abstract the OJClone dataset, and conduct multiple experiments on the abstracted OJClone dataset.

2. We rewrite some too strong claims about AbsBigCloneBench, and once again explain our contribution in detail.

3. We delete the repetitive parts in the article and carefully correct the grammatical mistakes in the article .

4. We clarify that identifier abstraction is not a novel technique we are proposing. We give a detailed introduction to the BigCloneBench and the abstraction techniques it used, as well as the abstraction techniques in other software engineering tasks.

5. We once again explain our contributions in detail and we point out the possible application scenarios of models that rely less on identifiers.

6. We clearly delineate which dataset is being used for each research question.

In the following sections, we provide detailed responses in blue to the comments from each reviewer in their appearance order in the reviews.

Thank you very much for your time and kind consideration.

Best regards,

Hao, Xing, Ge, Tao, Ying and Qianxiang

## Reviewer: 1

==========

**Comment 1.0:**

Comments:

---Summary

The general finding of the paper, i.e., that BigCloneBench artificially oversaturates performance through errors/issues in its construct validity, is well taken and quite useful. The work does a good job at showing this through three different means: dataset analysis, model ablation, contrast with improved data. What hampers this message are three broad issues:

- The paper is far too verbose. The above point could be safely made in half the pages. Please consider several of the reductions proposed above (and more).

- Several claims, related especially to AbsBigCloneBench, are too strong. Abstracting identifiers does not make for a "better" dataset, that captures semantic similarity more; identifiers are perfectly important to semantics. Stick to the key problem: BCB was poorly constructed, making identifiers to obvious a cue to clones. Abstracting them solves that particular problem on this particular dataset, and indeed makes for better models \*on this dataset\*, but not in general (at least, not provably so). Proving that would require something like a comparative evaluation with models trained on OJClone, which does not suffer from these issues, possibly involving human annotators. See examples in my soundness comments.

- The writing needs to be improved drastically. As is, it contains many typos, grammatical mistakes, problems with textual outline (e.g. sentences not ending with a period, or with too many; random uppercases midway through a sentence), and awkward or downright incomprehensible phrases.

**Response 1.0:**

Thanks for the very useful feedback. We are glad that you like the work.

**Comment 1.1:**

Several claims, related especially to AbsBigCloneBench, are too strong. Abstracting identifiers does not make for a "better" dataset, that captures semantic similarity more; identifiers are perfectly important to semantics. Stick to the key problem: BCB was poorly constructed, making identifiers to obvious a cue to clones. Abstracting them solves that particular problem on this particular dataset, and indeed makes for better models \*on this dataset\*, but not in general (at least, not provably so). Proving that would require something like a comparative evaluation with models trained on OJClone, which does not suffer from these issues, possibly involving human annotators.

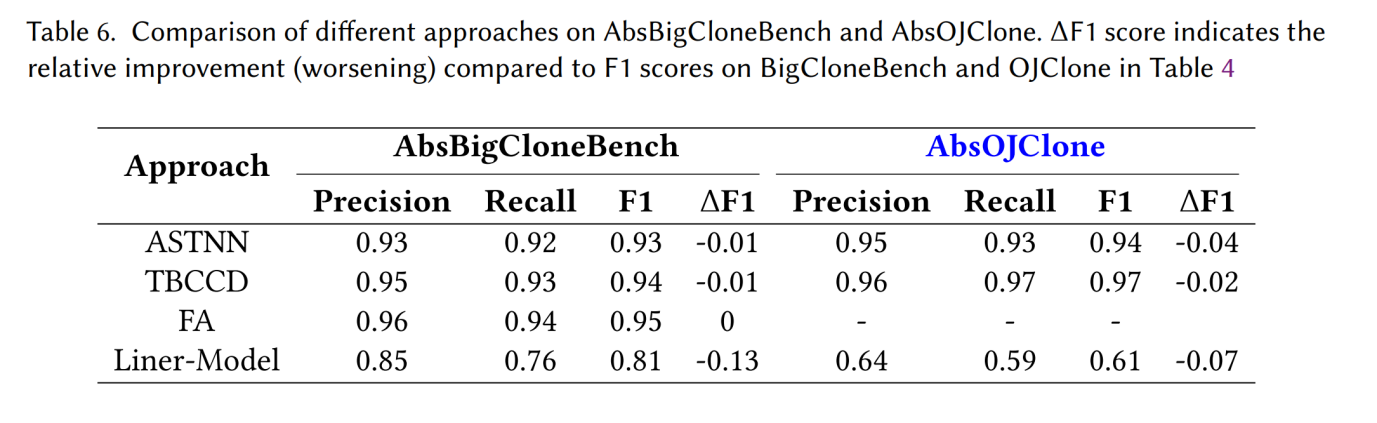
**Response 1.1:**

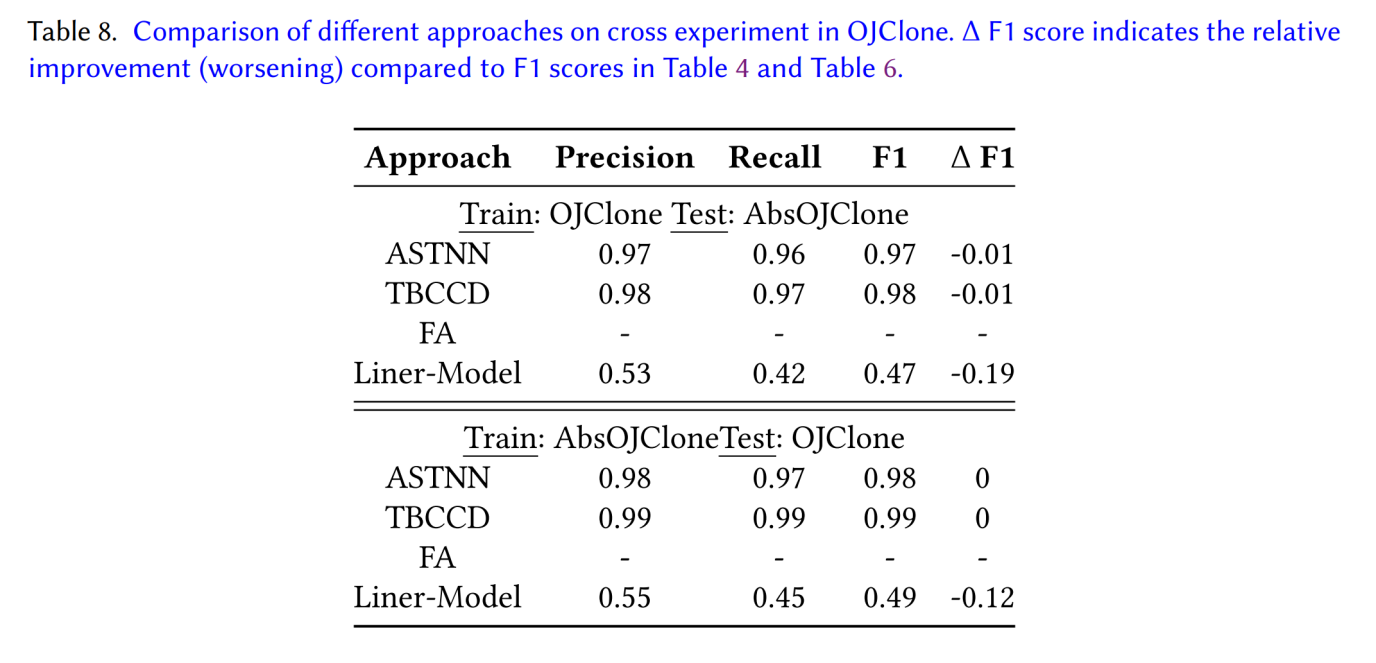
1. The abstract identifier technology is only valid for datasets that rely on identifiers (e.g., BigCloneBench), and is invalid for datasets that do not rely on identifiers (e.g., OJClone).
2. In order to further elaborate on the point we want to express, we use the same abstraction technology to abstract the OJClone dataset, and conduct multiple experiments (Table 6, 8) on the abstracted OJClone dataset. Experiment results show that the performance of the approaches is the same on AbsOJClone and OJClone. Both AbsOJClone and and OJClone can effectively verify the effectiveness of the approaches for semantic code clone detection. The cross-experiment results of the models on the OJClone and AbsOJClone are almost the same.
3. At the same time, we add three practical assessment techniques to check weather a dataset can be used for validating the effectiveness of the models of semantic code clone detection.

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 liner 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.





2. Some claims of AbsBigCloneBench in the article are indeed too strong.

(1) We rewrite "AbsBigCloneBench is more desirable than BigCloneBench on deep learning approaches validation" as “AbsBigCloneBench can help BigCloneBench to better verify the effectiveness of semantic code clone detection model“.

(2) We rewrite "Models trained with AbsBigCloneBench are more generalizable than those trained with BigCloneBench." as “Models trained on the AbsBigCloneBench are still valid on the real-world dataset (i.e., BigCloneBench)”.

3. According to the definition of code clone, AbsBigCloneBench will not change the labeling of semantic code clones and non-semantic code clones in the BigCloneBench. A good semantic code clone detection model should still be valid on the AbsBigCloneBench, so AbsBigCloneBench is reasonable from the perspective of verifying whether the model of semantic code clone detection is effective.

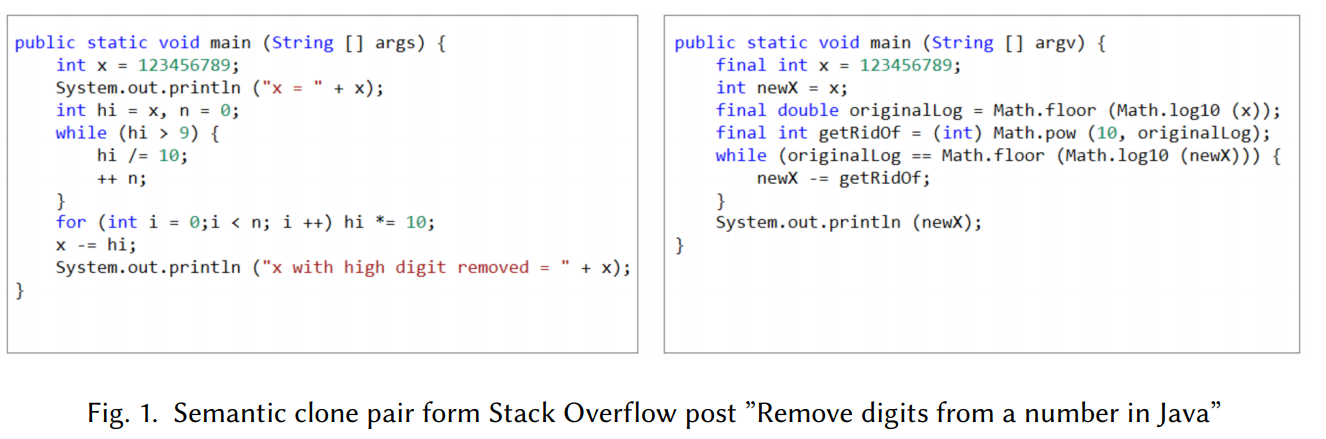
**Comment 1.2:**

Introduction: the work makes the argument for the hazards of code duplication, but focuses on the challenges in detecting the hardest, least syntactically similar kinds of semantic clones. While these two are related, it needs to be better substantiated that this work addresses the problems attributed to the former: does code duplication in practice often manifest with (very) different identifiers? (the cited linux duplication number, for instance, was based on lexical similarity). If so, substantiate with references; if not, please characterize the contributions accordingly.

**Response 1.2:**

1.The article focuses on claiming that the use of the BigCloneBench cannot effectively verify the effectiveness of the semantic code clone detection approaches based on deep learning. The linux duplication problem highlighted in the introduction part is not the focus of the article. We have removed the linux duplication part in the introduction.

1. Although the OJClone dataset is not mined from real-world development projects, OJClone is also obtained from real-world programming training scenarios. The identifiers in the semantic code clones in the OJClone are not similar, because students will define different identifiers when solving programming problems. This paper also uses the Jaccard similarity coefficient to prove that the OJClone has a weak dependence on identifiers. In addition to the OJClone dataset, we also add an example (Figure 1) of semantic code clone pair on StackOverflow, which does not rely on identifiers.



3. BigCloneBench is currently the largest benchmark of code clone detection that is mined from existing real-world development projects. As emphasized in the article, the semantic code clone in BigCloneBench relies heavily on identifiers. Finding semantic code clones with different identifiers is a challenge. The construction of BigCloneBench is explained in section 2.2, which is newly added in the article. 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. BigCloneBench has reduced the impact of identifiers to a certain extent when it is constructed, but the identifiers between the semantic code clone pairs in it are still very similar. We think this does not mean that the identifiers of semantic code clones in real-world are very similar, because it is difficult to find a large number of semantic code clone pairs with different identifiers through an effective way.

**Comment 1.3:**

The writing needs to be improved drastically. As is, it contains many typos, grammatical mistakes, problems with textual outline (e.g. sentences not ending with a period, or with too many; random uppercases midway through a sentence), and awkward or downright incomprehensible phrases.

**Response 1.3:**

We have carefully corrected the grammatical mistakes in the article.

**Comment 1.4:**

The paper frequently discusses "One Node Convolutions", but concretely, the model appears to be just a simple linear layer (eq. 4) applied to the node embeddings (due to the absence of hidden layers and convolution dimension of 1). - 3.2: The first few sentences are rather repetitive.

**Response 1.4:**

We rewrite section 3.2 in a brief way. The expression of ONCCD is removed and replaced with the expression of Linear-Model, and the model diagram is changed from Figure 2 to Figure 3.

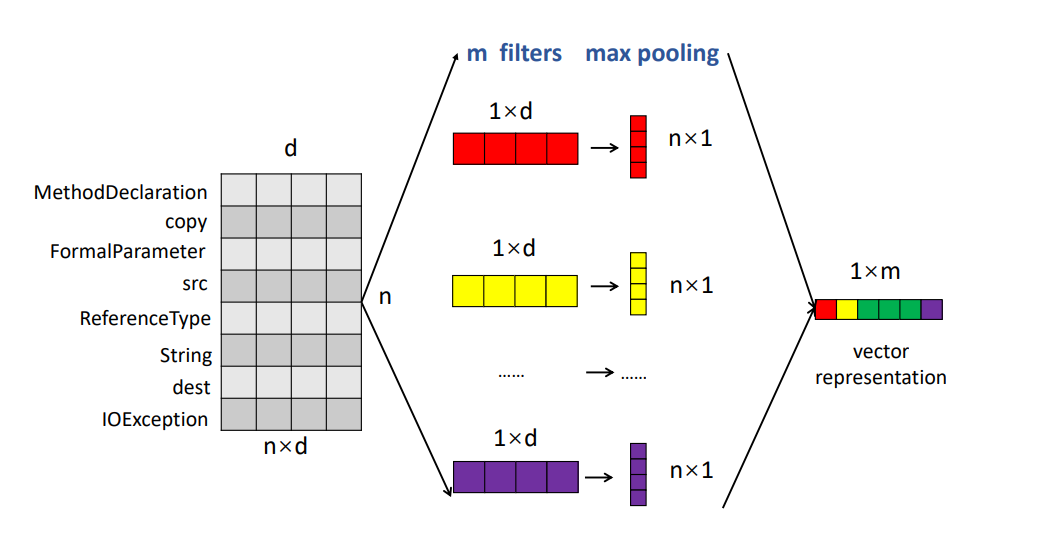


Figure2: Original ONCCD

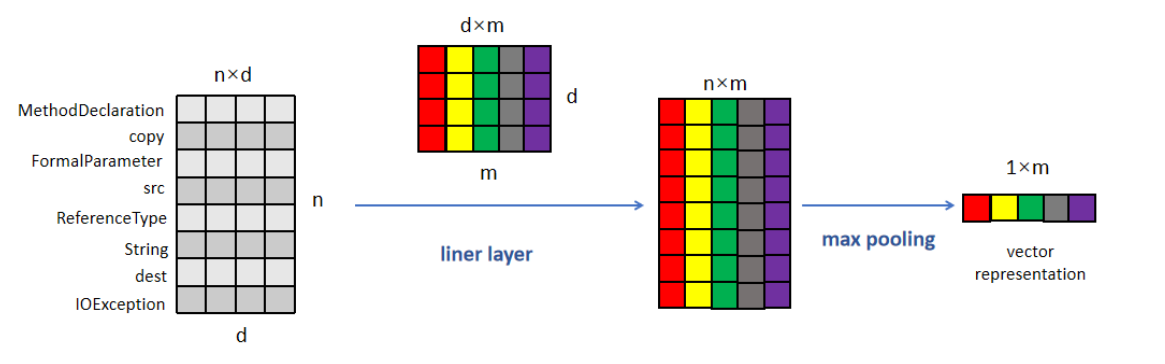


Figure3: New Liner Model

**Comment 1.5:**

The introduction is quite repetitive, especially in terms of stating the contributions.

The introduction to section 4 summarizes results that are then repeated in the results section (and are also repeated in the introduction and conclusion). Consider removing.

- 4.1.1 seems redundant.

- There are many repeated/redundant statements throughout the text.

- 5.2.1 is largely redundant, at least the first paragraph. This rationale is pretty well established by now.

- The renaming in 5.2.2 is quite commonly used and may not need as long an explanation (nor Algorithm 2).

- 5.3.2 seems quite redundant; several reasons are mentioned in the preceding sub-subsection. Consider merging them.

**Response 1.5:**

1. We carefully reduce the length of the repetitive part.
2. We combine the first two paragraphs in introduction and remove the linux duplication.
3. We rewrite the contribution in introduction, and remove the repeated stating of results in introduction and conclusion. For example, “We find 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. Therefore, a reasonable benchmark dataset for the task of detecting semantic code clones should not heavily rely on identifier information when detecting semantic clones.” 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” in the contribution of section 1 are removed. “Even an undesirable-by-design deep learning approach (i.e., Linear-Model) performs well on the semantic clone detection by considering only the identifier information. ” in section 4 are removed.
4. we remove 4.1.1, we remove the redundant statements such as “we are not saying that the identifiers of two code fragments belonging to the same semantic clone should not be similar”, “researchers need to pay attention to the identifier name in BigCloneBench when using BigCloneBench for validating and comparing deep learning models for detecting semantic clones”.
5. We clarify that identifier abstraction is not a novel technique we are proposing. We remove Algorithm 2 in the article.
6. We merge 5.2.1 with 5.2.2, and introduce the existing abstract identifier work in section 6.

**Comment 1.6:**

P12: the explanation for why OJClone & BigCloneBench have different degrees of overlap in identifiers, while it may be correct, should be substantiated with numbers: are 'i,j,k' really responsible for so much overlap in the former?

**Response 1.6:**

We use the Jaccard similarity coefficient to prove that when only identifiers are considered, the identifier similarity coefficient between different problems in the OJClone dataset is relatively high. We counted the proportion of tokens used for Jaccard similarity calculation in the OJClone dataset, and variable names accounted for 93%. Therefore, we believe that the reason for the overlap of OJClone and BigCloneBench on the identifier is the variable name, such as i, j, k.

**Comment 1.7:**

5.2.3, L20 states: "AbsBigCloneBench can effectively validate whether a deep learning model can learn semantic information or not when detecting semantic clones". This seems mistaken; AbsBigCloneBench only helps in identifying the importance of identifiers to the model. "Semantic information" is not equivalent to "code without identifiers".

**Response 1.7:**

We rewrite "AbsBigCloneBench can effectively validate whether a deep learning model can learn semantic information or not when detecting semantic clones" as "AbsBigCloneBench can help BigCloneBench to better verify the effectiveness of semantic code clone detection approach".

**Comment 1.8:**

- Minor note: "identifier names" is often used in the paper, e.g. referring to what abstracted, but this should probably be made more specific (at least once) to refer to "local variable and method declaration names" (or the likes), since API names are preserved and are also 'identifiers' from a parse tree perspective.

**Response 1.8:**

We rewrite the “identifier” to “part-identifier”

## Reviewer: 2

==========

**Comment 2.0:**

Comments:

---Summary

Good job at exposing this identifier-bias in BigCodeBench. I like the work and the systematic way of showing it, with different experimental settings. One concern I have is regarding a false sense of novelty which one gets when reading the paper. The authors should clearly state that identifier abstraction or normalization is not a novel technique they are proposing.

They should give credit to other works which already use it. Identifier normalization is well-known and is used in other deep learning tasks such as vulnerability detection. See references 1 and 2. This and a potential existence in other DL - BigCode tasks should be cited. Also, references in the paper 5/16/32 do normalization but not on BigCloneBench. This should be mentioned. Also, the paper introducing BigCodeBench, cited as reference 40 in the paper, itself talks of some sort of identifier normalization. But the fact that the identifier-bias still remains in the released dataset is interesting. Differences should be highlighted.

Similarly, I believe the authors would have already done this due diligence, but they should perhaps recheck prior work on BigCloneBench to see if others use normalization / abstraction as a regular preprocessing step. Explicitly mentioning that this does not happen will make your case stronger.

**Response 2.0:**

Thanks for the very useful feedback. We are glad that you like the work.

**Comment 2.1:**

1. The authors should clearly state that identifier abstraction or normalization is not a novel technique they are proposing.

They should give credit to other works which already use it.

Identifier normalization is well-known and is used in other deep learning tasks such as vulnerability detection. See references 1 and 2.

This and a potential existence in other DL - BigCode tasks should be cited.

**Response 2.1:**

We clarify that identifier abstraction is not a novel technique we are proposing. We explain the other existing works of abstract identifiers, and explain the section 5.2.1 and section 6.1 in the article.

1. “The abstraction technology is similar to Harer et al. [24], Liet al. [9], which we discuss the abstraction technology used by Harer et al. [24], Liet 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.”
2. “Abstract identifier technology has been applied in many approaches based on deep learning for the tasks of software engineering. In vulnerability detection tasks, 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.”

**Comment 2.2:**

Also, references in the paper 5/16/32 do normalization but not on BigCloneBench. This should be mentioned.

**Response 2.2:**

We add the normalization of other code clone detection approach in section 6.1.

“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.”.

**Comment 2.3:**

The paper introducing BigCloneBench, cited as reference 40 in the paper, itself talks of some sort of identifier normalization. But the fact that the identifier-bias still remains in the released dataset is interesting. Differences should be highlighted.

**Response 2.3:**

We add a new section (section 2.2) to detail the construction of BigCloneBench.

“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.”

**Comment 2.4:**

- Need explanation regarding how does 913 code fragments gets converted to 416238 pairs in Section 4.3.

- Line 41 page 12, should it not be "12 times MORE"

- Figure 6(a) should maybe be titled ONCCD and not CNN

- Line 30, page 21 should be "the F1 value of ONCCD on BigCloneBench can reach 0.94"

**Response 2.4:**

1. We carefully correct the grammatical mistakes in the article.
2. We replace "12 times less" with "12 times more".
3. We replace the title “CNN” with “Linear-Model”.
4. We replace the “with "the F1 value of SVM on BigCloneBench can reach 0.94” with "the F1 value of Linear-Model on BigCloneBench can reach 0.94".
5. We introduce the construction of code pairs in each dataset. For example, we explain how to get 416,238 pairs in section 4.3: 913\*912/2=416,238.

## Reviewer: 3

==========

**Comment 3.0:**

Comments:

Summary:

This paper presents a study that examines the mechanisms by which Deep Learning-related techniques for code clone detection tested on the BigCloneBench dataset might be learning to distinguish between semantic code clones. The author's main hypothesis for conducting this investigation is that after investigating several pairs of semantic clones in the BigCloneBench dataset, they noticed that such semantic pairs tended to share semantically similar identifier names. The authors view this as a potential limitation of the BigCloneBench dataset that authors should be aware of when training DL models for code clone detection. To further illustrate this point, the authors train a simple DL model that purposefully overfits to the identifier names exhibited by these code clones and show that it performs well on the BigCloneBench dataset. The authors also generate a new dataset that abstracts identifier names and illustrates their simple DL performs more poorly on this dataset, whereas past code-clone detection techniques can still perform well on it while being less dependent on identifier names.

**Response 3.0:**

Thanks for the very useful feedback. We are glad that you like the work.

**Comment 3.1:**

My fundamental struggle with this paper is that the authors are essentially pointing out a natural occurrence in the BigCloneBench benchmark – that semantic code clones tend to have similar identifier names. While this is an interesting phenomenon to note, I am not sure I completely understand the implications for future work. If a large dataset of naturally occurring semantic code clones exhibit similar identifier names, then I am not sure there is an inherent danger for these DL techniques to be leveraging this information to help to identify such clones. In essence, this is part of the automated feature engineering process the DL models undertake.

**Response 3.1:**

If a large dataset of naturally occurring semantic code clones exhibit similar identifier names, it is reasonable to use the identifier information to make the DL model converge. But when the same test set only changes the identifier information (according to the definition of semantic code clone detection, changing the identifier does not change the original semantic code clone type), the model will become invalid. Therefore, for the task of semantic code clone detection, it is not enough for the model to learn only the identifier information. In order to enable the model to learn more non-identifier information and to reduce the model's dependence on identifiers, researchers should also pay attention to the use of abstract identifier technology.

Like the new example (i.e., Figure 1) we added in the introduction, it is unreasonable to distinguish dogs and wolves only based on the surrounding background. When it is difficult to improve the model, consider improving the background color of the dataset may avoid this problem. Similarly, the identifier in the BigCloneBench dataset pointed out in this article should not affect the model's judgment of semantic code clones.

**Comment 3.2:**

The authors claim such models are “more generalizable” however if their artificial abstracted dataset is purposefully “unnatural” I think this claim might be somewhat unfounded, as it is unclear what models trained on their abstracted dataset are more generalizable to, and why this matter for specific SE-related tasks.

**Response 3.2:**

Since our experimental results can only support models trained on the AbsBigCloneBench dataset and are still valid on the BigCloneBench dataset, we rewrite “Models trained with AbsBigCloneBench are more generalizable than those trained with BigCloneBench.” as “Model trained on the AbsBigCloneBench dataset is still valid on the real-world dataset (i.e., BigCloneBench)”.

We have rewritten section 5.3.

**Comment 3.3:**

I see the main findings of this paper being important for code clone detections domains where one might want models to focus less on identifier names, however, I’m not sure where this would appropriate or applicable. I think this is the primary question that the authors need to answer for this paper and that I struggled to understand from the current form of the manuscript.

**Response 3.3:**

1. By pointing out the identifier problem in BigCloneBench, the article mainly wants to express two views:

(1) Verification problem: BigCloneBench cannot effectively verify whether a semantic code clone detection approach is really effective. The addition of AbsBigCloneBench for assistance will more effectively verify whether the semantic code clone detection approach is really effective.

(2) The model trained with the AbsBigCloneBench dataset can rely less on identifier information, and the model is still valid on the BigCloneBench dataset.

2. We rewrite the contributions of our article:

**“Validation**. We design an undesirable-by-design deep learning approach named Linear-Model, 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 improves over BigCloneBench when used to validate and compare deep learning models of detecting semantic clones.

**Training**. The models trained on AbsBigCloneBench has less reliance on identifier information. Through cross-validation between BigCloneBench and AbsBigCloneBench, we find that models trained with AbsBigCloneBench are also effective on BigCloneBench, but models trained on BigCloneBench are not effective than models trained on AbsBigCloneBench.”.

3. In addition, when a model trained on a dataset that is less dependent on identifiers, the trained model will be less dependent on the identifier. When the identifier information changes, the model will remain valid.

4. 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, in the task of malware and vulnerable detection, models need to learn the behavior of the malware and vulnerable, only using identifier information is not enough. To assist programming learners to search more different implement can be another application scenario (As shown in Figure 1). We add this explanation in section 7.1.

5. We add another example in introduction to illustrate the limit of models trained on the task of image classification.

“As reported in Shen [38], some deep learning models distinguishes dogs and wolves based on the surrounding background, instead of their different looks, which leads to their poor performance on new data.”.

**Comment 3.4:**

Why did the authors only select 300 thousand pairs for the training sets from both OJClone and BigCloneBench? Is this consistent with past work? Could it be possible that the Jaccard similarities that they observed are simply due to sampling bias in these cases? Also, is the analysis performed in RQ1 performed on the entire datasets, or only the sampled versions of the datasets?

**Response 3.4:**

Similarly as ASTNN, TBCCD, FA, the article randomly selected 300 thousand pairs for the training sets from both OJClone and BigCloneBench. The data used to calculate the Jaccard coefficient similarity in the article is complete. Both CDLH and TBCCD use 9,134 code fragments in BigCloneBench and 7,500 code fragments in OJClone.

**Comment 3.5:**

I would urge the authors to discuss all of the datasets, including the Abstracted dataset, in Section 4.2, and clearly delineate which dataset is being used for which research question. I think this would also clear up some of the other methodological questions that I have above.

**Response 3.5:**

1. We clarify in Section 4.2 that BigCloneBench and OJClone are used to research our RQ1.
2. We add the descriptions of the AbsBigCloneBench and AbsOJClone. AbsBigCloneBench is used to verify our RQ1 and RQ2.

“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 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. ”

1. We add the descriptions of Violent-AbsBigCloneBench and Violent-AbsOJClone in Section 4.2. Violent-AbsBigCloneBench is used to research RQ2 in section 5.2.1, to assist in the introduction of AbsBigCloneBench.

“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 in section 3.1 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 ViolentAbsOJClone 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).”