# Data Quality Matters: A Case Study of Obsolete Comment Detection

Shengbin Xu[1], Yuan Yao[1], Feng Xu[1], Tianxiao Gu[2], Jingwei Xu[1], Xiaoxing Ma[1]
[1]State Key Lab for Novel Software Technology, Nanjing University, China
[2]Tiktok, USA
kingxu@smail.nju.edu.cn, {y.yao, xf, jingweix, xxm}@nju.edu.cn, tianxiaogu@bytedance.com

***Abstract*—Machine learning methods have achieved great success in many software engineering tasks. However, as a data-driven paradigm, how would the data quality impact the effectiveness of these methods remains largely unexplored. In this paper, we explore this problem under the context of just-in-time obsolete comment detection. Specifically, we first conduct data cleaning on the existing benchmark dataset, and empirically observe that with only 0.22% label corrections and even 15.0% fewer data, the existing obsolete comment detection approaches can achieve up to 10.7% relative accuracy improvement. To further mitigate the data quality issues, we propose an adversarial learning framework to simultaneously estimate the data quality and make the final predictions. Experimental evaluations show that this adversarial learning framework can further improve the relative accuracy by up to 18.1% compared to the state-of-the-art method. Although our current results are from the obsolete comment detection problem, we believe that the proposed two-phase solution, which handles the data quality issues through both the data aspect and the algorithm aspect, is also generalizable and applicable to other machine learning based software engineering tasks.**

***Index Terms*—Obsolete comment detection, machine learning for software engineering, data quality**

## I. Introduction

In the past years, machine learning has been widely used in many software engineering tasks, including defect detection [1]–[3], program repair [4]–[6], and code summarization [7]–[9]. The main focus of existing work is on developing customized machine learning models that can unleash the value of the available data in specific tasks. In addition to the model aspect, the quality of the training data itself has been recently emphasized in the machine learning community [10]. However, how would the data quality affect machine learning based software engineering tasks remains largely unknown.

In this work, we aim to fill this gap by investigating the data quality issue in the just-in-time obsolete comment detection problem. On one hand, when code changes happen, code comments (as one of the most important software artifacts) are likely to be ignored by developers and thus become outdated [11]–[13]. Such obsolete comments may provide confusing or even misleading information to developers, leading them to write vulnerable code [14] and thus degrading the quality of the software. On the other hand, some recent approaches [15], [16] have adopted machine learning techniques to detect obsolete comments based on a collected dataset (named OCDDATA hereinafter) from GitHub. Although millions of data samples were collected in OCDDATA [15], their quality aspect remains largely unchecked. Specifically, the supervision labels (i.e., whether the comment is obsolete or not) were determined according to some simple heuristics that may lead to incorrectness.

Several data samples with improper or incorrect labels are given in Fig. 1. There are two types of data quality issues, i.e., *false positives* and *false negatives*. Fig. 1(a) includes two false positives, where the changes from the old comment to the new comment (i.e., *a* to *an* and *lose* to *loses*) are grammar corrections and do not affect the actual semantics of the comments. Marking this as a positive label would mislead the learning model to capture grammar corrections instead of actual semantics. In Fig. 1(b) we show two false negatives, which are much more prevalent in the dataset due to the fact that the developer may easily forget or neglect to update the comment. According to the code changes, *TitleView* and *key group index* should be changed to *View* and *key group range*, respectively. However, the old comment and the new comment are identical, and the label is negative. Such false negatives may make the learning model less sensitive to the case when the comment should be updated as well as prevent the model effectiveness from being accurately assessed.

These observations raise the following critical questions:

$RQ_1$ – *Due to their data-driven nature, to what extent are the existing machine learning based obsolete comment detection approaches affected by the data quality issue?*

$RQ_2$ – *As large-scale manual labeling is extremely expensive, can we automate the detection of false negatives so as to mitigate the impact of quality issues?*

$RQ_3$ – *Which design choices are especially useful for improving the performance?*

**Our Work –** In this paper, we present an obsolete comment detection approach ADVOC with data cleaning and adversarial learning techniques to answer the above questions.

$RA_1$ – To answer $RQ_1$, we first analyze the samples in the existing dataset and propose a set of rules for data cleaning. By applying our rules, we reduce the size of OCDDATA by 15.0% and correct the labels of 5,790 samples (0.22% of the training set). We then apply two recent machine learning based methods OCD [15] and Just-In-Time [16] to OCDDATA both before and after our data cleaning, and evaluate them on the same test set. The experimental results show that their per-

formance does indeed improve with the cleaned training data. Specifically, when trained with the cleaned dataset (with even 15.0% fewer data samples and only 0.22% label corrections), the performance of the two methods relatively improves by 10.7% and 2.3% in F1-score.

$RA_2$ – To answer $RQ_2$, we propose to adopt the adversarial learning framework to handle the false negatives. Specifically, we first select a small set of reliable, high-quality data samples from the training data based on the sample characteristics and the correlations between samples. These reliable samples act as the seeds for adversarial learning, and we denote the rest of the training data as unreliable samples. We then design a hierarchical encoding method that captures the complex semantics within each data sample, and transforms it into feature vectors. Specifically, we consider multiple edit representations of each code change as well as the connection between a code change to its corresponding comment. After obtaining the feature vectors of data samples, we train a generative adversarial network (GAN) [17] with a classifier and a discriminator against each other. The goal of the classifier is to assign labels (positive or negative) to unreliable samples, trying to make it impossible for the discriminator to distinguish between reliable and unreliable samples; the goal of the discriminator is to distinguish between reliable and unreliable samples. Through such adversarial learning, we can simultaneously estimate the reliability degree of all the unreliable samples (including those potential false negatives) and make the final predictions. The experimental results show that, on the cleaned dataset, our approach can further achieve up to 18.1% relative improvement compared to the best competitor Just-In-Time [16].

$RA_3$ – To answer $RQ_3$, we conduct an ablation study by deleting each of the design choices, including the adversarial learning framework and each of the three edit representations (i.e., edit sequence, edit tree, and edit script). The results show that the adversarial learning framework and the edit tree representation are especially important for the obsolete comment detection problem, relatively improving the F1-score by 8.1% and 11.5%, respectively.

**Contribution –** Note that although we study the obsolete comment detection problem in this work, the proposed data cleaning and adversarial learning ideas are potentially applicable to many other similar problems. In summary, this paper makes the following contributions:

- We propose data cleaning and adversarial learning techniques to tackle the data quality issues in the obsolete comment detection problem.
- We design a hierarchical encoding method for representing the complex semantics within each data sample, including a set of methods for various perspectives of code changes as well as the connections between comments and code changes.
- Experiment evaluations confirm the effectiveness of the proposed approach and that both our data cleaning and adversarial learning techniques can improve the effectiveness of obsolete comment detection.

```
  public SLStatementNode createBreak(Token breakToken) {
    final SLBreakNode breakNode =new SLBreakNode(srcFromToken(breakToken));
-   if (prober != null) {
-     return prober.probeAsStatement(breakNode);
-   }
    return breakNode;
  }
```

**old comment**: Returns a {@link SLBreakNode} for the given token.
**new comment**: Returns an {@link SLBreakNode} for the given token.
**label: true**

```
    public void revokeLeadership() {
      runAsync(new Runnable() {
        public void run() {
          log.info("ResourceManager {} was revoked leadership.",
getAddress());
          clearState();
+         leaderSessionID = null;
        }
      });
    }
```

**old comment**: Callback method when current resourceManager lose leadership.
**new comment**: Callback method when current resourceManager loses leadership.
**label: true**

(a) Two false positive samples.

```
-  public TitleView getTitleView() {
+  public View getTitleView() {
     return mTitleView;
   }
```

**old comment**: Returns the {@link TitleView}
**new comment**: Returns the {@link TitleView}
**label: false**

```
-  public int getKeyGroupIndex() {
-    return keyGroupIndex;
+  public KeyGroupRange getKeyGroupRange() {
+    return keyGroupRange;
   }
```

**old comment**: Returns the key group index the KvState instance belongs to.
**new comment**: Returns the key group index the KvState instance belongs to.
**label: false**

(b) Two false negative samples.

Fig. 1. Some data samples with improper labels in OCDDATA dataset.

**Data Availability –** The original OCDDATA dataset can be found at [18]. The cleaned dataset, tool, and source code are released at [19].

**Roadmap –** The remainder of this paper is organized as follows. Section II introduces the background knowledge. Section III shows the overview of our approach, and Sections IV, V, and VI are devoted to data cleaning, data sample encoding, and adversarial learning, respectively. Section VII presents the experimental results. Section VIII discusses the threats to validity and the implications, Section IX covers the related work, and Section X concludes.

## II. BACKGROUND

### A. *The* OCDDATA *Dataset*

The current benchmark dataset for obsolete comment detection is collected by Liu *et al.* [15]. Specifically, Liu *et al.* cloned 1,496 Java repositories from GitHub that were manually verified by Wen *et al.* [13] to be popular real software projects. Then, they iterated through each repository's non-merge commits and extracted modified methods and their corresponding java-docs. The method-doc instances with abstract methods or docs without description sections were

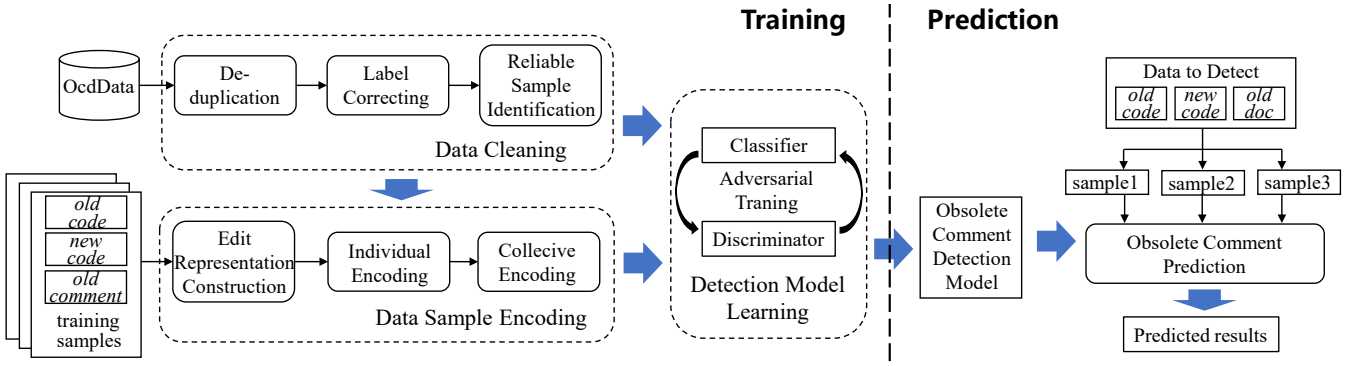



Fig. 2. Overview of the proposed ADVOC approach.

purged, and finally 4,357K+ qualified instances in the form of ⟨*old code, new code, old doc, new doc*⟩ are extracted. To further decide which sentence in the doc is obsolete, Liu *et al.* matched the sentences between the old doc and the new doc, and divided each method-doc instance into multiple tuples in the form of ⟨*old code, new code, old comment, new comment*⟩. Then, the label of each tuple is determined by comparing the old comment with the new comment. That is, the label is *negative* if the old and new comments are the same, and *positive* otherwise. Finally, each *data sample* is in the form of ⟨*old code, new code, old comment, label*⟩, where the former three serve as the input and the label serves as the output. After a few filtering steps, Liu *et al.* successfully curated a dataset consisting of about 4,086K data samples. Liu *et al.* also did data splitting based on the commit creation time, i.e., the samples corresponding to the earliest 80% commits in each repository were used as the training set and the remaining samples were randomly split into the validation and test sets.

### B. Generative Adversarial Networks

GAN [17] is an adversarial learner, consisting of a generator $G$ and a discriminator $D$. The two networks play the following minimax game with the value function $V(G, D)$:

$$\begin{aligned}&\min_G \max_D V(D,G)\\ =&\mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1-D(G(\boldsymbol{z})))].\end{aligned} \tag{1}$$

where $p_{\text{data}}$ is the data generating distribution of the training data, and $p_{\boldsymbol{z}}$ is the prior distribution of the input noise variables. The generator $G$ takes random noises as input and tries to generate data instances that approximate the training data to fool the discriminator, while the discriminator $D$ takes both instances from the training data and those generated by $G$ as input and tries to distinguish between these two types of instances. Through an iterative and adversarial process, $G$ can eventually generate instances that $D$ can hardly distinguish.

## III. Approach Overview

In this section, we present the overview of ADVOC. As shown in Fig. 2, ADVOC consists of two stages: the *training stage* and the *prediction stage*.

In the training stage, there are three modules: (1) *data cleaning*, (2) *data sample encoding*, and (3) *detection model learning*. The first module is mainly responsible for false positives and it performs data cleaning before feeding the data into the following pipeline. We reuse the OCDDATA dataset as it is large, carefully-crafted, and open-sourced. However, there are still quality issues with the dataset and we perform several data cleaning steps accordingly. Specifically, we first de-duplicate the existing samples to ensure that all samples in the dataset are unique and that there are no samples with conflicting labels. We next identify and correct false positive samples by defining a set of heuristic rules. We also add a data reliability estimation step which divides the existing data into reliable and unreliable sets. This division facilitates the detection model learning module, as will be later shown.

In the data sample encoding module, we propose a hierarchical encoding method to transform the data samples into feature vectors. We need to deal with the ⟨*old code, new code, old comment*⟩ terms in each data sample. Specifically, we first tokenize the *old comment*, and represent the *old code* and *new code* into three different forms (i.e., edit sequence, edit tree, and edit script, as will be later shown). We then propose to first individually encode each of them, and fuse them afterwards by taking into account the connections between the code and the comment.

In the detection model learning module, we mainly deal with the false negatives. We take both the reliable set and the unreliable set as input, and train an adversarial learning model that simultaneously estimates the reliability of each data sample and makes the final predictions. Unlike traditional GANs, we replace the generator with a classifier to predict a more appropriate label for each unreliable sample. In other words, we train two deep neural networks, i.e., a classifier and a discriminator, against each other to better unveil the label information of the samples in the unreliable set. Both classifier and discriminator use the encoded features from the previous module.

In the prediction stage, ADVOC takes the changed method and the corresponding doc (i.e., *old code, new code, old doc*) as input, and then separates the doc into sentences to form multiple data samples. Next, each data sample is transformed

TABLE I
STATISTICS OF THE ADVOC DATASET.

| | Train | Valid | Test | Total |
|---|---|---|---|---|
| Original | 3,194,930 | 437,042 | 454,383 | 4,086,355 |
| Original Positive | 85,469 | 9,846 | 9,647 | 104,962 |
| Cleaned | 2,676,160 | 410,258 | 385,258 | 3,471,676 |
| Cleaned Positive | 66,754 | 8,417 | 7,359 | 82,530 |

into features and fed into the trained classifier from the detection model learning module,[1] and the classifier returns either *positive* or *negative* for each sample. A positive feedback means that the method change causes inconsistency between the code and the comment. In this way, ADVOC can tell developers which sentences in the doc need to be updated after method changes.

## IV. DATA CLEANING

In the data cleaning module, a typical data cleaning workflow [20] (i.e., discovery, error detection, and error repair) is adopted. This module encourages the approach to focus on the actual semantics instead of the grammar corrections.

### A. Sample De-duplication

Despite the fact that the OCDDATA dataset is carefully constructed, we observe a large number of duplicate samples, some of which even have conflicting labels. After manual inspection, we find that this is mainly due to the existence of identical commits (caused by *git merge* or *git cherry-pick*, with the same diff but different commit hashes) in the same project. We consider two samples as duplicates if they are essentially the same input (i.e., the same *old code*, *new code*, and *old comment*), regardless of whether the *new comments* or *labels* are the same. For duplicate samples, we choose to keep a random one of them and we keep the positive sample when the labels are conflicting. After de-duplication, there are 3,471K+ samples left (decreased by $15.0\%$) and the detailed statistics can be found in the "Cleaned" row of Table I.

### B. Label Correction

Next, we address the label inaccuracy due to false positives. Specifically, one of the authors first manually checks over 500 randomly sampled positive samples and discovers that a small part ($\approx 10\%$) of the instances are false positives. We then analyze the causes of these false positives and detect them with a few manually summarized rules. For example, we find that format changes (e.g., adding an in-line tag to a token: *TitleView* $\rightarrow$ {*@link TitleView*}, or changing the verb form: *Get* $\rightarrow$ *Gets*) in the comments account for most of the false positive samples. However, there are various types of format changes and it is difficult to exhaust them. We adopt the trial-and-error method to further refine the detection rules. In detail, we first generate rough rules based on the manually checked false positives and use the current rules to make the detection. Then, we manually check a small portion of the "detected false positives". If the results do not meet our expectations (e.g., some of the "detected false positives" are not true false positives), we further refine our rules and detect again until satisfactory results are observed. We finally obtain the following rules to detect the five types of format changes:

- *In-line tag changing*. We remove all the in-line tags from both old and new comments before comparing them.
- *Case changing*. We compare the lowercase of both old and new comments.
- *Stopword changing*. If the changed words are in [*a, an, the, in, on, at*], we treat the two comments as the same one.
- *Lexical translation*. If the root format (after lemmatization) of the changed words are the same, the two comments are treated as the same one.
- *Typo fixing*. If only one word in the comments is changed, the minimum edit distance is no greater than 2, and the word before edit is not in the old code, the sample is viewed as a typo fixing.

Based on the above rules, 7,304 false positive samples (0.22%) are identified from the whole cleaned dataset, and 5,790 of them are found from the cleaned training set. We modify the labels of these detected false positives into negatives. After this step, there are 82K+ positive samples left (decreased by $21.4\%$) and the specific statistics can be viewed in the "Cleaned Positive" row of Table I.

### C. Reliable Sample Identification

In this step, we identify a small set of reliable samples from the cleaned training set. The proposed approach will make use of both the reliable subset and the remaining relatively unreliable training set during adversarial learning. Specifically, we consider the samples corresponding to docs that have been checked by developers to be relatively reliable, and whether a developer has checked a doc is determined by whether the developer has updated any of the sentences in the doc. In other words, we assume that if the developer has changed any sentence in a method doc, he/she would have probably checked all the sentences in this doc. Therefore, we mark all the samples corresponding to changed docs as reliable samples regardless of the labels. Similarly, the docs corresponding to the samples identified as false positives in the cleaned training set are also considered to be reliable. Finally, we identify 116,496 reliable samples from the training set (66,754 positive samples and 49,742 negative samples).

## V. DATA SAMPLE ENCODING

Data sample encoding transforms each data sample into feature vectors. Here, we present a hierarchical encoding method that considers both the code change and the connection between the code change and the comment.

### A. Edit Representation Construction

In the field of code modeling, simultaneously using semantically equivalent but formally different representations (e.g., code token sequence and AST) as input has achieved

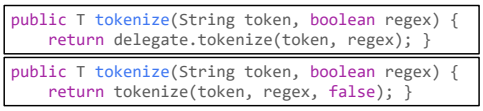
[1]Although both classifier and discriminator are trained in the detection model learning module, only the classifier is used in the prediction stage.

excellent performance [9], [21], [22]. In this paper, we propose to simultaneously use three different forms of code changes, i.e., *edit sequence*, *edit tree*, and *edit script*. An example is shown in Fig. 3. Essentially, different representations can help neural networks "understand" code changes from different perspectives. For example, the *edit sequence* is for the naturalness of code, the *edit tree* reveals more of the syntax, and the *edit script* directly delivers what entities have been changed. Each representation is described in detail below.

The *edit sequence* is formed by merging two code sequences. Specifically, we first tokenize both *old code* and *new code* with a lexer. Then, the compound words in identifiers and string literals are further split into multiple tokens according to naming conventions. Finally, we use difflib [23] to obtain the edit sequence and represent it as the form proposed by Panthaplackel *et al.* [24]. An example of edit sequence is shown in Fig. 3(b).

The *edit tree* is obtained by parsing both *old code* and *new code* into ASTs and then merging them. To be specific, we denote the AST of *old code* and *new code* as $T_{old}$ and $T_{new}$, and use GumTree [25] to calculate the difference between $T_{old}$ and $T_{new}$. The output includes a set of *node actions*, specifying matched, inserted, deleted, moved, and updated nodes on $T_{old}$ and $T_{new}$. Next, $T_{old}$ and $T_{new}$ are merged by consolidating the matched and moved nodes and keeping the remaining nodes and edges. An example edit tree is shown in Fig. 3(c), where node `RECEIVER` is deleted and node `BooleanLiteral` is inserted.

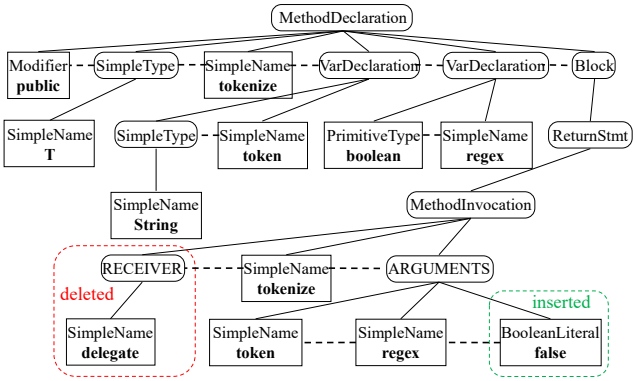
The *edit script* is derived from the edit tree, but discards the AST parts that are not affected by the code change. As shown in Fig. 3(d), each edit script is a bag of *individual changes* and each individual change consists of five elements, i.e., (`action_type`, `old_node`, `old_path`, `new_node`, `new_path`). There are four types of *actions* in edit scripts, i.e., delete, insert, move, and update. The `old_node` and `old_path` are obtained from $T_{old}$, and `new_node` and `new_path` are obtained from $T_{new}$. The `path` denotes the path from root node (`MethodDeclaration` in Fig. 3(c)) to the changed node on AST, which implies the location of the changed node.




(a) The old code and new code.

```
<KEEP> public t tokenize ( string token ,
boolean regex ) { return <KEEP_END>
<DELETE> delegate . <DELETE_END>
<KEEP> tokenize ( token , regex <KEEP_END>
<INSERT> , false <INSERT_END>
<KEEP> ) ; } <KEEP_END>
```

(b) The edit sequence.




(c) The edit tree.

```
(delete, [RECEIVER],
[MethodDeclaration,Block,ReturnStmt,MethodInvocation], [], []),
(insert, [], [], [BooleanLiteral,false],
[MethodDeclaration,Block,ReturnStmt,MethodInvocation,ARGUMENTS])
```

(d) The edit script.

Fig. 3. An illustrative example of edit representations.

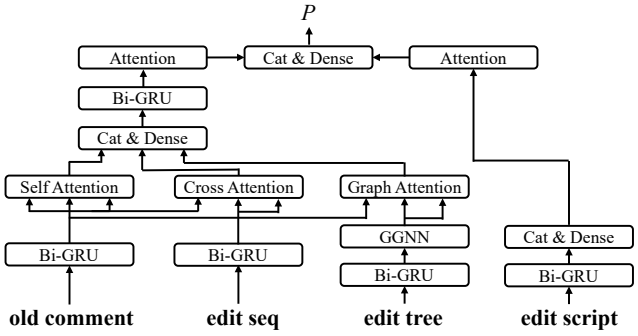



Fig. 4. The data sample encoding architecture.

## B. Individual Encoding

We next individually encode each edit representation as well as the comment. The encoding architecture is shown in Fig. 4.

*1) Edit Sequence Encoder:* Following the standard way, we use Bi-GRU layers [26] to encode the edit sequence:

$$h_i^{(s)} = \text{Bi-GRU}(h_{i-1}^{(s)}, h_{i+1}^{(s)}, e_{s_i}), \tag{2}$$

where $s_i$ is the $i$-th token in edit sequence, $e_{s_i}$ is its embedding, and $h_i^{(s)}$ is its contextual vector.

*2) Edit Tree Encoder:* For an edit tree, we treat it as a heterogeneous graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, X)$, where $\mathcal{V}$ and $\mathcal{E}$ refer to the node set and edge set, respectively, and $X$ includes the node attributes. For each node $v$, we merge its *type* and *value* into a sequence $x_v$. Take the inserted node in Fig. 3(c) as an example, its attribute sequence $x_v$ is [`BooleanLiteral`, `false`]. The Bi-GRU used for $x_v$ is similar to Eq. (2) and thus omitted for brevity. To obtain the vector representation for $x_v$, we also consider the node action embedding $e_{a_v}$,

$$h_v = W^{(n)} \left[ h_{|x_v|}; e_{a_v} \right], \tag{3}$$

where $W^{(n)}$ is a matrix to resize $h_v$, and $h_{|x_v|}$ is the last hidden state of the Bi-GRU.

We then use a gated graph neural network (GGNN) [27], which has been widely used for code graph encoding, to encode the edit tree. The GGNN models relationships (edges) between nodes with message passing operations. There are five types of directed edges in edit trees, including four commonly

used in AST, i.e, child, parent, next_sibling, and prev_sibling, as well as one derived from the update relation identified by GumTree. We use $k$ to denote the type index and $K$ to denote the number of types. Specifically, in GGNN layer $l$, the messages sent by each node $v$ to its neighbors through the edges of type $k$ are computed through a fully-connected layer,

$$m_{k,v}^{(l)} = W^{(l)} h_v^{(l)} + b^{(l)}. \tag{4}$$

Next, node $v$ aggregates all the incoming messages from its neighbors and updates itself,

$$\begin{aligned} m_v^{(l)} &= \sum_{(u,v)\in\mathcal{E}_k, 1\le k\le K} m_{k,u}^{(l)}, \\ h_v^{(l+1)} &= \text{GRU}\left(m_v^{(l)}, h_v^{(l)}\right). \end{aligned} \tag{5}$$

The node vector inputted to the first layer is $h_v$, and the vector outputted by the last layer $h_v^{(L)}$ is used for the next stage.

*3) Edit Script Encoder:* We first use Bi-GRUs to encode `old_node`, `old_path`, `new_node`, and `new_path` for each individual change. Then, the resulting vectors in the same individual change are concatenated and fed into a fully-connected layer, which is a common way of feature fusion. Finally, we aggregate multiple individual changes with attention, which facilitates the model to focus on important changes.

Take the $i$-th individual change for example,

$$h_i^{(cc)} = W^{(cc)} \left[e_{t_i}; h_{on_i}; h_{op_i}; h_{nn_i}; h_{np_i}\right], \tag{6}$$

where $h_i^{(cc)}$ is the individual change vector, $W^{(cc)}$ is used for resizing, $e_{t_i}$ is the embedding of `action_type`, and $h_{on_i}$, $h_{op_i}$, $h_{nn_i}$, and $h_{np_i}$ are the last hidden states of Bi-GRUs given `old_node`, `old_path`, `new_node`, and `new_path`, respectively.

Given a bag of vectors $\{h_1^{(cc)}, ..., h_n^{(cc)}\}$, we calculate the attention weight $\alpha_i$, and obtain the edit script vector $v^{(e)}$ by a linear combination,

$$\begin{aligned} \alpha_i &= \frac{\exp\left(h_i^{(cc)T} \cdot a\right)}{\sum_{j=1}^{n} \exp\left(h_j^{(cc)T} \cdot a\right)}, \\ v^{(e)} &= \sum_{i=1}^{n} \alpha_i \cdot h_i^{(cc)}, \end{aligned} \tag{7}$$

where $a$ is a learnable vector that has the same size with $h_i^{(cc)}$.

*4) Comment Encoder:* Finally, we use a Bi-GRU as shown in Eq. (2) to encode the comment. Comments are first tokenized and parsed following standard NLP steps. The hidden state of the $i$-th comment token is referred to as $h_i^{(c)}$.

### C. Collective Encoding

We next use the attention mechanism (e.g., self-attention, cross-attention, and graph-attention) to model the connections between inputs.

First, to better capture the global contextual information of comments, we use self-attention [28] to update $H^{(c)}$ (which is obtained by stacking $h_i^{(c)}$),

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V, \tag{8}$$

where $Q$, $K$, $V$ are obtained by performing different linear transformations on the comment vectors $H^{(c)}$. We denote the result by $H^{(cg)}$.

Next, to connect comment to edit sequence, we use cross-attention. The equations are similar to Eq. (8) except that $K$ and $V$ are computed from edit sequence vectors $H^{(s)}$ (by stacking $h_i^{(s)}$). The result is denoted by $H^{(cs)}$.

Then, we connect comment to edit tree via graph-attention. Specifically, we use graph relational embedding attention [29], which, unlike directly calculating the attention score using $(q_i k_j^T)/\sqrt{d_k}$, adds a bias term $b_{i,j}$ into the computation and the equation becomes $(q_i + b_{i,j})k_j^T/\sqrt{d_k}$. This bias term is computed as follows,

$$b_{i,j} = w_e^T e_{(i,j)} + b_e, \tag{9}$$

where $w_e$ is a learnable vector, $b_e$ is a learnable scalar, and $e_{(i,j)}$ is the embedding of the *edge type* connecting comment token $i$ and edit tree node $j$. We define three edge types including *original match*, *token match*, and *stem match*:

- *Original match*: the original form of comment token $i$ is identical to the value of node $j$.
- *Token match*: comment token $i$ is identical to one subtoken of node $j$'s value.
- *Stem match*: the stem word of token $i$ (after lemmatization) is identical to one subtoken of node $j$'s value.

The other computational steps of graph relational embedding attention are the same with Eq. (8), with the exception that $K$ and $V$ are computed from edit tree vectors $H^{(L)}$ (by stacking $h_v^{(L)}$). The result of graph-attention is denoted by $H^{(ct)}$.

We combine the above $H^{(cg)}$, $H^{(cs)}$, and $H^{(ct)}$ through

$$h_i = W^{(c)} \left[h_i^{(cg)}; h_i^{(cs)}; h_i^{(ct)}\right], \tag{10}$$

and feed them into a Bi-GRU layer (refer to Eq. (2)). Then, the enhanced comment vector $v^{(c)}$ is calculated from the output of the Bi-GRU layer using equations similar to Eq. (7).

Finally, we concatenate $v^{(c)}$ and $v^{(e)}$ to form a vector $v$ to represent the final embedding of a data sample.

*Remarks*. Note that one of our baselines, Just-In-Time [16], also encodes multiple forms of code changes. The key differences are two-fold. First, we model *old comment*, *edit seq*, *edit tree*, and *edit script*, whereas Just-In-Time models only the first three. Second, we additionally use Bi-GRU to fuse the attribute information of each node on the edit tree, and use graph-attention to explicitly encode the connections between old comment and edit tree.

## VI. Detection Model Learning

In this section, we present our adversarial learning framework for dealing with false negatives.

### A. Adversarial Learning

Here, we denote the reliable set from Section IV-C as $\mathcal{R}$, and the rest unreliable set as $\mathcal{U}$. We then adopt the adversarial learning idea to build a classifier $C$ and a discriminator $D$. The reliable set $\mathcal{R}$ is analogous to the $p_{\text{data}}$ in Eq. (1), and

the unreliable set $\mathcal{U}$ is analogous to the $p_{\boldsymbol{z}}$. The classifier $C$ takes unreliable samples as input and predicts labels for each sample with the purpose of making the discriminator unable to distinguish between reliable and unreliable samples. The purpose of the discriminator $D$ is to distinguish between reliable and unreliable samples. Through such adversarial learning, we can estimate the reliability of the unreliable samples based on the knowledge of the reliable samples.

For simplicity, we let $D$ and $C$ use the same structure as shown in Fig. 4. The encoded vector $v$ for each data sample is then fed into a fully-connected layer with sigmoid activation,

$$p = \text{sigmoid}\left(Wv + b\right), \tag{11}$$

where $p$ denotes the probability that the old comment is obsolete. The adversarial learning is built on top of the probability $p$ outputted by $D$ and $C$. Specifically, for each unreliable sample, discriminator $D$ tries to make the distance between $p^D$ and $p^C$ as large as possible to identify the unreliable sample, where $p^D$ is the probability predicted by $D$ and $p^C$ is the probability predicted by $C$. Classifier $C$ tries to make $p^C$ and $p^D$ as close as possible to fool $D$. We use binary cross-entropy to measure the distance as obsolete comment detection is a binary classification problem. The minimax game played by $C$ and $D$ can be formulated as follows:

$$\begin{aligned}\min_C \max_D V(D,C) = &- \sum_{(x^r,y^r)\in\mathcal{R}} \text{BCE}\left(p^D_{x^r}, y^r\right) \\ &- \alpha \sum_{(x^u,y^u)\in\mathcal{U}} \text{BCE}\left(p^D_{x^u}, y^u\right) \\ &+ \beta \sum_{(x^u,y^u)\in\mathcal{U}} \text{BCE}\left(p^D_{x^u}, p^C_{x^u}\right),\end{aligned} \tag{12}$$

where $x$ and $y$ correspond to the sample and its label; $\alpha$ and $\beta$ are hyper-parameters. The binary cross-entropy is calculated as:

$$\text{BCE}(p_1, p_2) = -p_2 \cdot \log p_1 - (1-p_2)\cdot\log(1-p_1). \tag{13}$$

Note that our adversarial learning framework is loosely built upon our edit encoding method (it can admit an output vector of any encoding method). Instead of directly using the output vector to make predictions as existing work did, it further employs a small subset of reliable seeds, and tries to estimate the reliability of other data samples from these reliable seeds.

### B. Training Algorithm

Alg. 1 shows the training algorithm of ADVOC. In each training iteration, the parameters of $D$ are updated first, and then the parameters of $C$. Eq. (14) and Eq. (15) for calculating the gradients are derived from Eq. (12). The $\gamma$ in Eq (14) is a constant that represents the ratio of the number of samples in the reliable set to the number in the unreliable set, which is $0.0455$ in our case. The meanings of $\alpha$ and $\beta$ are the same as in Eq. (12), and the function $\phi$ is calculated as follows:

$$\phi(x) = \begin{cases} 1.0 & \text{if } x < 0.5, \\ 0.0 & \text{if } x \geq 0.5, \end{cases} \tag{16}$$

where $0.0 \leq x \leq 1.0$.

**Algorithm 1** Adversarial training algorithm of ADVOC.

**Input:** reliable training data $\mathcal{R}$, unreliable training data $\mathcal{U}$;
**Output:** well-trained classifier $C$;

1: Randomly initialize discriminator $D$ and classifier $C$;
2: **for** each iteration **do**
3: Sample a minibatch of $m$ unreliable samples $\{(x^u_1, y^u_1), ..., (x^u_m, y^u_m)\}$ from $\mathcal{U}$.
4: Sample a minibatch of $m$ reliable samples $\{(x^r_1, y^r_1), ..., (x^r_m, y^r_m)\}$ from $\mathcal{R}$.
5: Update the discriminator by descending its stochastic gradient:

$$\begin{aligned}\nabla_{\theta_d} \frac{1}{m}\sum_{i=1}^{m}[&\gamma \cdot \text{BCE}\left(D(x^r_i), y^r_i\right) + \alpha \cdot \text{BCE}\left(D(x^u_i), y^u_i\right) \\ &+ \beta \cdot \text{BCE}\left(D(x^u_i), \phi(C(x^u_i))\right)]\end{aligned} \tag{14}$$

6: Update the classifier by descending its stochastic gradient:

$$\nabla_{\theta_c} \frac{1}{m}\sum_{i=1}^{m} \text{BCE}\left(C(x^u_i), D(x^u_i)\right) \tag{15}$$

7: **end for**
8: **return** $C$

## VII. EVALUATIONS

In this section, we present the experimental results. The experiments are designed to answer the following questions.

- **RQ1: Data Cleaning.** To what extent can data cleaning improve the existing obsolete detection methods?
- **RQ2: Overall Performance.** How does the proposed ADVOC perform compared to the existing obsolete comment detection approaches?
- **RQ3: Ablation Study.** How does each individual module of ADVOC impact its performance?

### A. Experimental Setup

*1) Data Preparation:* As mentioned earlier, we use OCDDATA [15] in our experiment. More precisely, we use two versions of OCDDATA: one is the *Original* version and the other is the *Cleaned* version after de-duplication and label correction. The statistics of these two versions OCDDATA are shown in Table I. We first follow the original training/test data split *by time* (i.e., the earliest 80% data samples are used as the training set) used by Liu *et al.* [15] to ensure a fair comparison. Considering that even the labels of the cleaned OCDDATA test set (denoted by *Full Test Set*) may not be completely correct (e.g., false negatives cannot be corrected in the label correction stage), we randomly select samples from the test set and manually label them for more reliable evaluation. Specifically, two of the authors first manually label each sample independently. When there are conflicting results, the third person (another author) participates in the discussion, and the three authors reach a consensus by voting. Finally, we obtain 500 positive samples and 500 negative samples, and refer to these 1,000 samples as the *Verified Test Samples*.

To further verify the effectiveness of ADVOC under the cross-project setting, we additionally re-partition the cleaned

dataset *by project*. That is, we sort the 1,496 GitHub repositories by the creation time and then select the samples corresponding to the most recently created 300 ($\approx 20\%$) repositories as the validation and test sets and the rest as the training set. The validation set and the test set are randomly divided, each containing samples corresponding to 150 repositories.

*2) Compared Approaches:* We compare our approach ADVOC with the following four baselines:

- **Fraco** [30]. Fraco is a rule-based tool to detect fragile comments caused by identifier renaming. The tool itself exists as an Eclipse plugin and is triggered whenever the user uses Eclipse's *rename refactoring* features.
- **RandomForest** [31]. RandomForest is a machine learning based approach for detecting outdated block/line comments during code changes. Specifically, RandomForest constructs 64 features for each sample and then trains a random forest classifier for prediction.
- **OCD** [15]. OCD is a deep learning based approach to detect obsolete comments after method changes. OCD represents each changed method as an edit triple sequence and then uses Bi-LSTM and Co-Attention layers to model the changed method, the old comment, and the relationships between them.
- **Just-In-Time** [16]. Just-In-Time is another deep learning based approach. It models old comment, edit sequence, and edit tree, as well as the relations between them.

We choose OCD and Just-In-Time as they are the most recent deep learning based methods and with better reported results. We choose RandomForest as it is a typical method using traditional machine learning techniques. We choose Fraco as it is the latest rule-based representative that does not apply machine learning.

*3) Evaluation Metrics:* To measure the performance, we use *Precision*, *Recall*, and *F1-score* metrics as they are well-known metrics for binary classification. Furthermore, considering the imbalance between positive and negative samples in the test set, we choose a metric commonly used for imbalanced datasets, the area under the precision-recall curve (*AUPRC*) [32]. Precision-recall curve shows what happens to precision and recall as the decision threshold changes and AUPRC is a general measure of the corresponding precision-recall curve irrespective of any particular threshold. We do not report the AUPRC results for the *Verified Test Samples* as this test set is balanced.

*4) Implementations:* For data sample encoding in ADVOC, the maximum number of tokens in the *edit sequence* and *old comment* are set to 380 and 160, respectively, the maximum number of nodes in the *edit tree* is set to 380, the maximum number of individual changes in the *edit script* is set to 10, and the maximum number of tokens in the *node* and nodes in the *path* are both set to 8. Statistics shows that the above values work for more than 95% of the samples in the dataset and we truncate the part that exceeds these limits. We set the embedding size of all tokens except `action_type` to 64, and the size of `action_type` is 8. Likewise, we set the hidden size of all Bi-GRUs to 64, resulting in an output hidden state size of 128 ($64 \times 2$). The layer number of Bi-GRUs in the edit sequence encoder and comment encoder is set to 2, while the layer number of Bi-GRUs in the edit tree encoder and edit script encoder is set to 1. The layer number and hidden size of GGNN are set to 8 and 128. The hidden size and head number of self attention, cross attention, and graph attention are set to 128 and 4. The dropout rate is set to 0.5.

For detection model learning, we use the Adam optimizer [33] with a 0.0001 learning rate for both the classifier and discriminator. The hyper-parameters $\alpha$ and $\beta$ in adversarial training are set to 0.4 and 0.1, respectively. The batch size is set to 128. We tune these hyper-parameters and select the best performing classifier based on the validation set.

For compared methods, we use the offline Fraco [34] provided by Liu *et al.* [15] as the original Fraco [30] is not applicable to our scenario. It first uses RefactoringMiner to detect rename refactorings and then uses Fraco to detect fragile comment phrases with respect to each rename refactoring. We re-implement the RandomForest [31] according to the specification in the paper and discard some unavailable features in our dataset (7 out of 64) since the source code is unavailable. We directly reuse the implementations of OCD [15] and Just-In-Time [16] as provided in the corresponding papers. The parameters are set as described in their papers when possible.

TABLE II
EFFECTIVENESS RESULTS OF THE TWO BASELINE APPROACHES TRAINED ON BOTH THE ORIGINAL DATASET AND THE CLEANED DATASET. BETTER RESULTS ARE OBSERVED ON THE CLEANED DATASET.

| | Training data | Precision | Recall | F1-score |
|---|---|---|---|---|
| OCD | Original | 95.8% | 27.6% | 42.9% |
| | Cleaned | 95.8% | **31.6%** | **47.5%** |
| Just-In-Time | Original | 95.8% | 49.8% | 65.5% |
| | Cleaned | **97.0%** | **51.2%** | **67.0%** |

### B. Results and Analysis

*1) RQ1: Data Cleaning:* To answer RQ1, we train the two latest baseline approaches (i.e., OCD and Just-In-Time) on both the original OCDDATA and the cleaned OCDDATA, and then evaluate them on the *Verified Test Samples*. Table II shows the results. We can observe that both approaches perform significantly better when trained on the cleaned data. Specifically, compared with the OCD model trained on the original data, the OCD model trained on the cleaned dataset relatively improves the Recall and F1-score by 14.5% and 10.7%, respectively. OCD's Precision does not change, and this is mainly because there are still many false negatives in the cleaned training data. On the other hand, less false positive noise in the training set makes the trained model better identify the features of true positives, thus improving the Recall.

Similarly, the Precision, Recall, and F1-score of Just-In-Time trained on the cleaned data are relatively improved by 1.3%, 2.8%, and 2.3%, respectively. We also observe that Just-In-Time works better than OCD. These two methods mainly differ in that OCD does not use the edit tree and thus cannot learn from the syntactic structure. This result also indicates

TABLE III
EFFECTIVENESS COMPARISON RESULTS ON CLEANED DATASET. THE PROPOSED ADVOC GENERALLY OUTPERFORMS THE EXISTING APPROACHES.

| | Full Test Set | | | | Verified Test Samples | | |
|---|---|---|---|---|---|---|---|
| | **Precision** | **Recall** | **F1-score** | **AUPRC** | **Precision** | **Recall** | **F1-score** |
| Fraco | 19.5% | 15.4% | 17.2% | 18.3% | 90.2% | 22.0% | 35.4% |
| RandomForest | **65.1%** | 13.5% | 22.4% | 29.4% | **99.1%** | 21.4% | 35.2% |
| OCD | 60.2% | 21.6% | 31.8% | 34.6% | 95.8% | 31.6% | 47.5% |
| Just-In-Time | 60.6% | 35.4% | 44.7% | 41.4% | 97.0% | 51.2% | 67.0% |
| ADVOC | 58.8% | **41.8%** | **48.8%** | **43.7%** | 98.7% | **58.8%** | **73.6%** |

that syntactic information can improve the performance of obsolete comment detection.

Overall, the above results indicate that the existing machine learning based obsolete comment detection approaches are indeed affected by the data quality issues. Additionally, with 0.22% corrected labels and even 15.0% fewer training data, we can have significantly better effectiveness.

*To conclude RQ1, we show that data cleaning can significantly improve the performance of existing obsolete comment detection methods.*

*2) RQ2: Overall Performance:* We next compare our approach ADVOC with the existing baseline approaches on the cleaned OCDDATA, and Table III shows the results.

We can first observe from the table that the proposed approach ADVOC generally outperforms the baselines on both the full test set and the *Verified Test Samples*. Specifically, on the full test set, ADVOC is better than all competitors in Recall, F1-score, and AUPRC metrics. Compared to the best competitor Just-In-Time, ADVOC relatively improves the Recall, F1-score, and AUPRC by 18.1%, 9.2%, and 5.6%, respectively. Similar results are observed on the *Verified Test Samples*, achieving 14.8% and 9.9% relative improvements on Recall and F1-score, respectively. Specifically, we observe that the Precision results on the *Verified Test Samples* are all close to 100%. Compared with the relatively lower Precision on the full test set, this result confirms that there may be many false negatives in the full test set. This is also the probable reason that the Precision of ADVOC is a little worse than Just-In-Time in the full test set. Additionally, since all the competitors are evaluated on the cleaned training data, and ADVOC mainly differs from OCD and Just-In-Time in the data sample encoding and adversarial learning methods, the results indicate that our algorithmic design could achieve better effectiveness results than the competitors.

In Table III, we notice that RandomForest performs well in the Precision metric but degrades much on the Recall metric. The reason is that RandomForest is designed for specific types of obsolete comments and cannot accurately detect general obsolete comments. To further show how different methods perform on different samples, we show the Venn diagram of the successfully detected obsolete comments on the *Verified Test Samples* in Fig. 5. In total, 344 out of 500 obsolete comments have been successfully detected. Among all the methods, ADVOC detects the most obsolete comments (294), including 37 comments that cannot be detected by other methods. Among the competitors, Fraco detects 20 obsolete

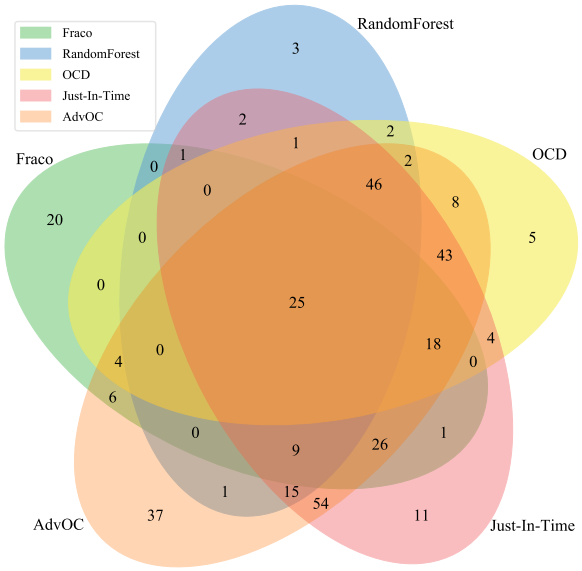



Fig. 5. The number of successfully detected obsolete comments by each approach.

TABLE IV
EFFECTIVENESS COMPARISON RESULTS ON DIVIDED-BY-PROJECT DATASET. ADVOC IS STILL BETTER.

| | **Precision** | **Recall** | **F1-score** | **AUPRC** |
|---|---|---|---|---|
| Fraco | 36.4% | 17.0% | 23.1% | 27.8% |
| RandomForest | **67.1%** | 14.4% | 23.7% | 32.1% |
| OCD | 59.4% | 14.0% | 22.7% | 31.0% |
| Just-In-Time | 53.1% | 34.5% | 41.8% | 38.5% |
| ADVOC | 53.0% | **39.6%** | **45.4%** | **42.3%** |

comments that cannot be detected by other methods. Since Fraco is built upon manually defined rules for specific types of comments, this result indicates the merit of traditional rule-based methods. How to complement the power of rule-based methods in specific scenarios with the versatility of learning-based methods in general scenarios is left as future work.

As mentioned above, we also test the case when the dataset is split *by project*. The results are shown in Table IV. Similar results to the *by time* data partition are observed. Specifically, compared with the best competitor Just-In-Time, ADVOC relatively improves it by 14.8%, 7.9%, and 9.9% w.r.t. Recall, F1-score, and AUPRC, respectively. We can also observe that deep learning based approaches, i.e., OCD, Just-In-Time, and ADVOC, perform worse in the *by project* setting than in the *by time* setting, probably because the code semantics and

TABLE V
ABLATION STUDY RESULTS ON CLEANED DATASET. ALL THE THREE EDIT REPRESENTATIONS AND THE ADVERSARIAL LEARNING FRAMEWORK IS HELPFUL IN IMPROVING THE EFFECTIVENESS.

| | Full Test Set | | | | Verified Test Samples | | |
|---|---|---|---|---|---|---|---|
| | **Precision** | **Recall** | **F1-score** | **AUPRC** | **Precision** | **Recall** | **F1-score** |
| ADVOC$^{-}_{adv}$ | **63.5%** | 34.9% | 45.1% | 44.0% | 98.5% | 52.0% | 68.1% |
| ADVOC$^{-}_{seq}$ | 59.3% | 39.4% | 47.3% | 43.2% | 98.3% | 56.4% | 71.7% |
| ADVOC$^{-}_{tree}$ | 58.4% | 36.1% | 44.6% | 40.4% | 98.0% | 49.8% | 66.0% |
| ADVOC$^{-}_{script}$ | 61.1% | 41.1% | **49.2%** | **45.0%** | 97.6% | 56.8% | 71.8% |
| ADVOC | 58.8% | **41.8%** | 48.8% | 43.7% | **98.7%** | **58.8%** | **73.6%** |

comments in different projects vary greatly. In addition, the F1-score of OCD decreased from 31.8% to 22.7%, which is the largest, and the difference between OCD and the other two methods lies in that OCD only encodes the sequence form of code changes. This indicates that considering both the edit tree and the edit script might help to better learn cross-project features.

*To conclude RQ2, with the proposed data sample encoding module and the adversarial learning module,* ADVOC *significantly outperforms the existing baselines.*

*3) RQ3: Ablation Study:* Finally, we analyze the performance gain of ADVOC, and the results are shown in Table V. To be specific, we evaluate the following variants of ADVOC. ADVOC$^{-}_{adv}$ is the variant without adversarial learning, i.e., we directly train the classifier on the full training set; ADVOC$^{-}_{seq}$ is the variant without modeling edit sequence; ADVOC$^{-}_{tree}$ is the variant without modeling edit tree; ADVOC$^{-}_{script}$ is the variant without modeling edit script.

As we can observe from Table V, ADVOC performs better than the four variants on the *Verified Test Samples*, indicating that the four components are all helpful in improving its effectiveness. Among the four variants, ADVOC achieves the most improvement compared to ADVOC$^{-}_{adv}$ and ADVOC$^{-}_{tree}$ (e.g., 8.1% and 11.5% relative improvements in F1-score). This indicates that our adversarial learning and edit tree representation are especially important for the obsolete comment detection problem. On the full test set, ADVOC$^{-}_{script}$ performs slightly better than ADVOC, and the main performance difference comes from the Precision metric. Meanwhile, ADVOC$^{-}_{adv}$ achieves the best Precision. These are probably, again, due to the false negatives in the full test set, and the false negative noise makes the results on the full test set less reliable than the results on the *Verified Test Samples*.

*To conclude RQ3, both the adversarial learning module and the sample encoding module (especially the edit tree representation) help to improve the effectiveness of* ADVOC.

## VIII. DISCUSSIONS

### A. Threats to Validity

The implementation of our approach and the compared approaches is the first threat to validity. To reduce this threat, we directly reuse the implementations provided by the authors and set the parameters as their papers describe when possible. For approaches that do not provide any implementation (i.e., RandomForest [31]), we re-implement the techniques strictly following their paper based on widely-used libraries (e.g., scikit-learn [35]). Meanwhile, we build our approach based on existing mature tools/libraries (e.g., GumTree and difflib).

The second threat comes from the proposed heuristic rules in the label correction stage. When the rules are too strict, only a small fraction of false positives are detected. Conversely, when the rules are not strict enough, samples that are not false positives are detected. This is a trade-off between quality and quantity, and we choose to first guarantee the quality of the detected false positives. Specifically, we iteratively perform rule refinement and manual checking until the samples from the detected false positives are true false positives.

The assumption based on which we identify reliable samples may also introduce a threat to validity. Specifically, to identify reliable samples, we assume that modified docs have been manually checked by the developer and thus are reliable, which might not always be true in practice. However, the proposed adversarial learning framework is quite robust and it does not require 100% reliable samples to achieve good results, as confirmed by our experimental results.

Another threat is the mislabeled samples in the test set. There could be many false negatives in the full test set, and thus we manually labeled the *Verified Test Samples*. We recommend focusing more on the results of the *Verified Test Samples*, as the label of each verified sample is endorsed by at least two of our authors. Within our affordable efforts, we labeled a balanced set of 1000 samples, and more extensive labeling work is left as future work.

### B. Implications

*Implications to other ML4Code tasks*: The two key steps of our approach, i.e., data cleaning and adversarial learning, are relatively general and have the potential to be applied to various machine learning for code tasks with data quality issues. Data cleaning has been extensively studied [20] and widely used. The error detection rules we summarized may not be applicable to all jobs, but the workflow (i.e., discovery, error detection, and error repair) we used and the ideas behind are universal. Moreover, the adversarial learning framework can admit vectors outputted by any encoding method. Thus, it can be used for various types of noisy datasets, as long as a relatively reliable subset of the full dataset can be identified (either by manual annotation or human-defined rules).

Use the code change quality estimation work [36] as an example. In this work, code changes that have been commented

on during the code review process are regarded as suspicious code that may have quality issues. Such labeling may introduce data quality issues (e.g., some comments do not involve code quality, leading to false positives; some low-quality changes are not commented on, resulting in false negatives). To this end, we may first inspect a small portion of the data to summarize the sources and types of data quality issues, and define rules to do some corrections. Further, we can apply the adversarial learning framework by manually identifying a small set of reliable samples from the whole dataset.

*Implications to practice*: ADVOC tells whether the comments corresponding to the changed methods are obsolete and which sentences in the comments need to be updated. Such capability enables ADVOC to improve developers' productivity and, ultimately, code quality to some extent. For example, one direct application scenario of ADVOC is code review. ADVOC can be integrated into the existing code review tool to warn reviewers of commits that result in obsolete comments, thus preventing the introduction of obsolete comments. Furthermore, ADVOC can be integrated into a standalone tool that traverses the project's git history and detects previously introduced obsolete comments. In this way, we may reduce the technical debt and improve the maintainability of the project.

## IX. Related Work

*Obsolete Comment Detection*. Various obsolete comment detection approaches have been proposed [14]–[16], [30], [31], [37]–[43]. Early proposals used manually defined rules to detect whether specific types of comments are obsolete (i.e., inconsistent with the code). For example, Tan *et al.* [37] extracted constraints from *locking* or *calling* related comments based on manually defined rules, and then determined obsolete comments based on whether the constrains held. Tan *et al.* [39] checked *null-pointer* related comments in a similar way but used dynamic analysis. Zhou *et al.* [14], [42] focused on *parameter usage constraints* in the comments and checked them by using manually defined rules and SMT solvers. Ratol and Robillard [30] proposed to detect fragile comments caused by identifier renaming. These rule-based methods are designed to detect specific types of obsolete comments, and how to adapt them to general comments is largely open. Our work can be complementary to the rule-based methods.

Recently, machine learning has been widely used to detect obsolete comments, and such methods are not limited to specific comment types. For example, Corazza *et al.* [40], Cimasa *et al.* [41], as well as Rabbi and Siddik [43] determined the consistency between code and comments with textual similarity. However, these methods treat code as text, and thus ignore the unique semantics of code. Later, the just-in-time obsolete comment detection problem was raised and studied [15], [16], [31]. These methods make the detection at the time when code change happens. For instance, Panthaplacke *et al.* [16] proposed a neural network model to detect whether a comment becomes inconsistent as a result of changes to the corresponding method. Liu *et al.* [15] also utilized neural networks to detect obsolete comments and they further updated the detected obsolete comments with another neural network model. However, these methods focus on developing more powerful neural networks, and ignore the quality issues of the training data from which the neural networks are built. There also exists research that focuses on the quality issues of machine learning code or framework [44], [45]. Our work differs from them and improves the learning-based methods with special focus on the data quality aspect.

*Code/Comment Inconsistency Analysis*. Prior studies [12], [13], [46]–[50] have investigated the inconsistencies between code and comment from different perspectives. For example, Fluri *et al.* [46], [49] investigated to what extent developers add or adapt comments when they evolve the code. Malik *et al.* [47] investigated what attributes lead to function's comment updates during evolution. Ibarahim *et al.* [50] studied the correlation between code-comment inconsistencies and software bugs. Linares-Vásquez *et al.* [12] mined a large set of open source projects, and observed that 17.2% method changes resulted in comment updates. Wen *et al.* [13] confirmed that 13%-20% of code changes trigger comment updates.

*Comment Generation and Update*. Our work is also related to comment generation and update [8], [9], [21], [24], [51]–[62]. For example, Hu *et al.* [8], [21] and LeClair *et al.* [9] combined AST information and neural networks to generate comments, and Chen *et al.* [57] proposed to integrate different comment generation techniques for different comment types. As for comment update, Liu *et al.* [58] updated comment with deep learning model, Lin *et al.* [59] proposed a heuristic-based approach, and Yang *et al.* [60] integrated deep learning based approach and heuristic-based approach.

## X. Conclusions

In this paper, we have proposed an obsolete comment detection approach ADVOC. The focus of ADVOC is on the data quality aspect, and it consists of a data cleaning module that corrects some false positive samples, a data sample encoding module that captures the complex semantics among code changes and comments, and a detection model learning module that adopts adversarial learning to handle false negative samples. Experimental evaluations show that: 1) existing comment detection methods can be significantly improved by simply applying them on the cleaned training data, and 2) ADVOC can further outperform the existing methods based on the proposed data sample encoding and adversarial learning methods. Future directions include conducting larger user studies on the predicted results of ADVOC, and applying the proposed framework to other machine learning based software engineering tasks.

## Acknowledgment

This work is supported by the National Natural Science Foundation of China (No. 62025202, 62172199), and the Collaborative Innovation Center of Novel Software Technology and Industrialization. Yuan Yao is the corresponding author.

## REFERENCES


[1] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, "Deepjit: an end-to-end deep learning framework for just-in-time defect prediction," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 34–45.

[2] Z. Cai, L. Lu, and S. Qiu, "An abstract syntax tree encoding method for cross-project defect prediction," *IEEE Access*, vol. 7, pp. 170 844–170 853, 2019.

[3] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "Cc2vec: distributed representations of code changes," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2020.

[4] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 341–353.

[5] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1161–1173.

[6] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1506–1518.

[7] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred api knowledge," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI)*, 2018, pp. 2269–2275.

[8] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *ICPC*, 2018.

[9] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 795–806.

[10] W. Liang, G. A. Tadesse, D. Ho, F.-F. Li, M. Zaharia, C. Zhang, and J. Zou, "Advances, challenges and opportunities in creating data for trustworthy ai," *Nature Machine Intelligence*, pp. 1–9, 2022.

[11] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *2013 21st international conference on program comprehension (icpc)*. Ieee, 2013, pp. 83–92.

[12] M. Linares-Vásquez, B. Li, C. Vendome, and D. Poshyvanyk, "How do developers document database usages in source code?(n)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 36–41.

[13] F. Wen, C. Nagy, G. Bavota, and M. Lanza, "A large-scale empirical study on code-comment inconsistencies," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 53–64.

[14] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing apis documentation and code to detect directive defects," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 27–37.

[15] Z. Liu, X. Xia, D. Lo, M. Yan, and S. Li, "Just-in-time obsolete comment detection and update," *IEEE Transactions on Software Engineering*, 2021.

[16] S. Panthaplackel, J. J. Li, M. Gligoric, and R. J. Mooney, "Deep just-in-time inconsistency detection between comments and source code," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 1, 2021, pp. 427–435.

[17] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," *Advances in neural information processing systems*, vol. 27, 2014.

[18] "The original OCDData," https://drive.google.com/drive/folders/1FKhZTQzkj-QpTdPE9f_L9Gn_pFP_EdBi.

[19] "The AdvOC Repo," https://github.com/SoftWiser-group/AdvOC.

[20] I. F. Ilyas and X. Chu, *Data cleaning*. Morgan & Claypool, 2019.

[21] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation with hybrid lexical and syntactical information," *Empirical Software Engineering*, vol. 25, no. 3, pp. 2179–2217, 2020.

[22] W. Cheng, P. Hu, S. Wei, and R. Mo, "Keyword-guided abstractive code summarization via incorporating structural and contextual information," *Information and Software Technology*, vol. 150, p. 106987, 2022.

[23] "The difflib documentation," https://docs.python.org/3/library/difflib.html.

[24] S. Panthaplackel, P. Nie, M. Gligoric, J. J. Li, and R. Mooney, "Learning to update natural language comments based on code changes," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020, pp. 1853–1868.

[25] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the IEEE/ACM international conference on Automated software engineering (ASE)*, 2014, pp. 313–324.

[26] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv*, 2014.

[27] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, "Gated graph sequence neural networks," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2016. [Online]. Available: http://arxiv.org/abs/1511.05493

[28] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems (NeurIPS)*, 2017, pp. 5998–6008.

[29] V. J. Hellendoorn, C. Sutton, R. Singh, P. Maniatis, and D. Bieber, "Global relational models of source code," in *International conference on learning representations*, 2019.

[30] I. K. Ratol and M. P. Robillard, "Detecting fragile comments," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 112–122.

[31] Z. Liu, H. Chen, X. Chen, X. Luo, and F. Zhou, "Automatic detection of outdated comments during code changes," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2018, pp. 154–163.

[32] K. Boyd, K. H. Eng, and C. D. Page, "Area under the precision-recall curve: point estimates and confidence intervals," in *Joint European conference on machine learning and knowledge discovery in databases*. Springer, 2013, pp. 451–466.

[33] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[34] "The offline Fraco Repo," https://github.com/Tbabm/FracoUpdater.

[35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[36] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu *et al.*, "Automating code review activities by large-scale pre-training," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1035–1047.

[37] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/* icomment: Bugs or bad comments?*," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007, pp. 145–158.

[38] N. Khamis, R. Witte, and J. Rilling, "Automatic quality assessment of source code comments: the javadocminer," in *International Conference on Application of Natural Language to Information Systems*. Springer, 2010, pp. 68–79.

[39] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@ tcomment: Testing javadoc comments to detect comment-code inconsistencies," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 260–269.

[40] A. Corazza, V. Maggio, and G. Scanniello, "Coherence of comments and method implementations: a dataset and an empirical investigation," *Software Quality Journal*, vol. 26, no. 2, pp. 751–777, 2018.

[41] A. Cimasa, A. Corazza, C. Coviello, and G. Scanniello, "Word embeddings for comment coherence," in *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2019, pp. 244–251.

[42] Y. Zhou, C. Wang, X. Yan, T. Chen, S. Panichella, and H. Gall, "Automatic detection and repair recommendation of directive defects in java api documentation," *IEEE Transactions on Software Engineering*, vol. 46, no. 9, pp. 1004–1023, 2020.

[43] F. Rabbi and M. S. Siddik, "Detecting code comment inconsistency using siamese recurrent network," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 371–375.

[44] D. Cheng, C. Cao, C. Xu, and X. Ma, "Manifesting bugs in machine learning code: An explorative study with mutation testing," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2018, pp. 313–324.

[45] X. Sun, T. Zhou, R. Wang, Y. Duan, L. Bo, and J. Chang, "Experience report: investigating bug fixes in machine learning frameworks/libraries," *Frontiers of Computer Science*, vol. 15, no. 6, pp. 1–16, 2021.

[46] B. Fluri, M. Wursch, and H. C. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *14th Working Conference on Reverse Engineering (WCRE 2007)*. IEEE, 2007, pp. 70–79.

[47] H. Malik, I. Chowdhury, H.-M. Tsou, Z. M. Jiang, and A. E. Hassan, "Understanding the rationale for updating a function's comment," in *2008 IEEE International Conference on Software Maintenance*. IEEE, 2008, pp. 167–176.

[48] S. Xi, S. Yang, X. Xiao, Y. Yao, Y. Xiong, F. Xu, H. Wang, P. Gao, Z. Liu, F. Xu *et al.*, "Deepintent: Deep icon-behavior learning for detecting intention-behavior discrepancy in mobile apps," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, pp. 2421–2436.

[49] B. Fluri, M. Würsch, E. Giger, and H. C. Gall, "Analyzing the co-evolution of comments and source code," *Software Quality Journal*, vol. 17, no. 4, pp. 367–394, 2009.

[50] W. M. Ibrahim, N. Bettenburg, B. Adams, and A. E. Hassan, "On the relationship between comment update practices and software bugs," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2293–2304, 2012.

[51] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering (ASE)*, 2010, pp. 43–52.

[52] E. Wong, T. Liu, and L. Tan, "Clocom: Mining existing source code for automatic comment generation," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 380–389.

[53] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proceedings of International conference on machine learning (ICML)*, 2016.

[54] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*, 2016.

[55] Y. Zhou, X. Yan, W. Yang, T. Chen, and Z. Huang, "Augmenting java method comments generation with context information based on neural networks," *Journal of Systems and Software*, vol. 156, pp. 328–340, 2019.

[56] S. Xu, Y. Yao, F. Xu, T. Gu, H. Tong, and J. Lu, "Commit message generation for source code changes," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI)*, 2019.

[57] Q. Chen, X. Xia, H. Hu, D. Lo, and S. Li, "Why my code summarization model does not work: Code comment improvement with category prediction," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 2, pp. 1–29, 2021.

[58] Z. Liu, X. Xia, M. Yan, and S. Li, "Automating just-in-time comment updating," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 585–597.

[59] B. Lin, S. Wang, K. Liu, X. Mao, and T. F. Bissyandé, "Automated comment update: How far are we?" in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 36–46.

[60] Z. Yang, J. W. Keung, X. Yu, Y. Xiao, Z. Jin, and J. Zhang, "On the significance of category prediction for code-comment synchronization," *ACM Transactions on Software Engineering and Methodology*, 2021.

[61] H. Zhu, X. He, and L. Xu, "Hatcup: Hybrid analysis and attention based just-in-time comment updating," *arXiv preprint arXiv:2205.00600*, 2022.

[62] S. Yang, Y. Wang, Y. Yao, H. Wang, Y. Ye, and X. Xiao, "Describectx: context-aware description synthesis for sensitive behaviors in mobile apps," in *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 685–697.