Learning to Spot and Refactor Inconsistent Method Names

Kui Liu†*, Dongsun Kim†*, Tegawendé F. Bissyandé†*, Taeyoung Kim‡, Kisub Kim†*, Anil Koyuncu†*, Suntae Kim‡, Yves Le Traon†
†Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg
‡{kui.liu, dongsun.kim, tegawende.bissyande, kisub.kim, anil.koyuncu, yves.letraon}@uni.lu
‡Department of Software Engineering, Chonbuk National University, South Korea
†{rlaxodud1200, jipsin08}@gmail.com

Abstract—To ensure code readability and facilitate software maintenance, program methods must be named properly. In particular, method names must be consistent with the corresponding method implementations. Debugging method names remains an important topic in the literature, where various approaches analyze commonalities among method names in a large dataset to detect inconsistent method names and suggest better ones. We note that the state-of-the-art does not analyze the implemented code itself to assess consistency. We thus propose a novel automated approach to debugging method names based on the analysis of consistency between method names and method code. The approach leverages deep feature representation techniques adapted to the nature of each artifact. Experimental results on over 2.1 million Java methods show that we can achieve up to 15 percentage points improvement over the state-of-the-art, establishing a record performance of 67.9% F1-measure in identifying inconsistent method names. We further demonstrate that our approach yields up to 25% accuracy in suggesting full names, while the state-of-the-art lags far behind at 11.1% accuracy. Finally, we report on our success in fixing 66 inconsistent method names in a live study on projects in the wild.

Index Terms—Code refactoring, inconsistent method names, deep learning, code embedding.

“If you have a good name for a method, you don’t need to look at the body.” — Fowler et al. [1]

I. INTRODUCTION
Names unlock the door to languages. In programming, names (i.e., identifiers) are pervasive in all program concepts, such as classes, methods, and variables. Descriptive names are the intuitive characteristic of objects being identified, thus, correct naming is essential for ensuring readability and maintainability of software programs. As highlighted by a number of industry experts, including McConnell [2], Beck [3], and Martin [4], naming is one of the key activities in programming.

Naming is a non-trivial task for program developers. Studies conducted by Johnson [5, 6] concluded that identifier naming is the hardest task that programmers must complete. Indeed, developers often write poor (i.e., inconsistent) names in programs due to various reasons, such as lacking a good thesaurus, conflicting styles during collaboration among several developers, and improper code cloning [7].

Method names are the intuitive and vital information for developers to understand the behavior of programs or APIs [8], [9].

*Corresponding authors.
findings suggest that developers are indeed striving to choose appropriate method names, often to address consistency with the contexts of their code.

To debug method name, Høst and Østvold [28] explored method naming rules and semantic profiles of method implementations. Kim et al. [7] relied on a custom code dictionary to detect inconsistent names. Allamanis et al. introduced the NATURALIZE framework [29] learning the domain-specific naming convention from local contexts to improve the stylistic consistency of code identifiers with n-gram model [30]. Then, building on this framework, they proposed a log-bilinear neural probabilistic language model to suggest method and class names with similar contexts [31]. The researchers leveraged attentional neural networks [32] to extract local time-invariant and long-range topical attention features in a context-dependent way to suggest names for methods.

Overall, their context information is limited to local identifier sub-tokens and the data types of input and output. While the state-of-the-art has achieved promising results, a prime criterion of naming methods has not been considered: the implementation of methods that is a first-class feature to assess method naming consistency since method names should be mere summaries of methods’ behavior [1]. Examples shown in Figure 1 illustrate the intuition behind our work:

Methods implementing similar behavior in their body code are likely to be consistently named with similar names, and vice versa. It should be possible to suggest new names for a method, in replacement to its inconsistent name, by considering consistent names of similarly implemented methods.

In this paper, we propose a novel automated approach to spotting and refactoring inconsistent method names. Our approach leverages Paragraph Vector [33] and Convolutional Neural Networks [34] to extract deep representations of method names and bodies, respectively. Then, given a method name, we compute two sets of similar names: the first one corresponds to those that can be identified by the trained model of method names; the second one, on the other hand, includes names of methods whose bodies are positively identified as similar to the body of the input method. If the two sets intersect to some extent (which is tuned by a threshold parameter), the method name is identified to be consistent, and inconsistent otherwise. We further leverage the second set of consistent names to suggest new names when the input method name is flagged as inconsistent.

To evaluate our proposed approach, we perform experiments with 2,116,413 methods of training data and 2,805 methods with changed names of test data, which are collected from 430 open source Java projects. Our experimental results show that the approach can achieve an F1-measure of 67.9% in the identification of inconsistent method names, representing an improvement of about 15 percentage points over the state-of-the-art. Furthermore, the approach achieves 34–50% accuracy on suggesting first sub-tokens and 16–25% accuracy on suggesting accurate full names for inconsistent method names, again outperforming the state-of-the-art. Finally, we report how our approach helped developers in fixing 66 inconsistent method names in 10 projects during a live study in the wild.

II. BACKGROUND

This section briefly describes three techniques from the field of neural networks, namely Word2Vec [35], Paragraph Vector [33] and Convolutional Neural Networks [34]. Our approach relies on these techniques to achieve two objectives: (1) embedding tokens from method names and bodies into numerical vector forms, and (2) extracting feature representations for accurately identifying similar method names and bodies.

1) Paragraph Vector: Paragraph Vector is an unsupervised algorithm that learns fixed-length feature representations from variable-length pieces of texts, such as sentences [33]. This technique was proposed to overcome the limitations of bag-of-word [36] features which are known to (1) lose the order of words and (2) ignore the word semantics. Recent studies provide evidence that paragraph vector outperforms other state-of-the-art techniques [36], [37] for text representations [33], [38], and can effectively capture semantic similarities among words and sentences [39]–[43].

In our work, we use Paragraph Vector for training a model to compute similarities among method names (considering sequences of method name sub-tokens as sentences). We expect this model to take into account not only the lexical similarity but also the semantic similarity: for example, function names containsObject and hasObject should be classified as similar names since both of them describe the functionality of code implementation to check whether a set contains a specific object put in argument(s). We detail in later parts of this paper how method names are processed in our approach to feeding the Paragraph Vector algorithm.

2) Convolutional Neural Networks (CNNs): CNNs are biologically-inspired variants of multi-layer artificial neural networks [34]. Initially developed and proven effectiveness in the area of image recognition, CNNs have gained popularity for handling various NLP tasks. For text classification, these deep learning models have achieved remarkable results [44],
[45] by managing to capture the semantics of sentences for relevant similarity computation. Recently, a number of studies [46]–[52] have provided empirical evidence to support the naturalness hypothesis, we treat source code, in particular, method bodies, as documents written in natural language and to which we apply CNNs for code embedding purpose. The objective is to produce a model that will allow to accurately identify similar method code. A recent work by Bui et al. [55] has provided preliminary results showing that some variants of CNNs are even effective to capture code semantics so as to allow the accurate classification of code implementations across programming languages. In this study, we use LeNet5 [56], a specific implementation of CNNs, which consists of lower-layers and upper-layers (see Figure 4 for its architecture).

3) Word2Vec: When feeding tokens of a method body to CNNs, it is necessary to convert the tokens into numerical vectors. Otherwise, the size of a CNN’s input layer would be too large if using one-hot encoding, or interpreting its output can be distorted if using numeric encoding (i.e., assigning a single integer value for each token). The machine learning community often uses vector representation for word tokens [33], [44], [57]. This offers two advantages: (1) a large number of (unique) tokens can be represented as a fixed-width vector form (dimensionality reduction) and (2) similar tokens can be located in a vector space so that the similar tokens can be dealt with CNNs in a similar way. Our approach uses Word2Vec [35] to embed tokens of method bodies. 

Word2Vec [58] is a technique that encodes tokens into n-dimensional vectors [35], [59]. It is basically a two-layered neural network dedicated to process token sequences. The neural network takes a set of token sequences (i.e., sentences) as inputs and produces a map between a token and a numerical vector. The technique not only embeds tokens into numerical vectors but also places semantically similar words in adjacent locations in the vector space.

III. Approach

This section presents our approach to debugging inconsistent method names. As illustrated in Figure 3, it involves two phases: (1) training and (2) identification & suggestion. In the training phase, taking as input a large number of methods from real-world projects, it uses Paragraph Vector for method names and Word2Vec + CNNs for method bodies to embed them into numerical vectors (hereafter simply referred to as vectors), respectively. Eventually, two distinct vector spaces are produced and will be leveraged in the next phase. The objective of the training phase is thus to place similar method names and bodies into adjacent locations in each vector space.

The identification & suggestion phase determines whether a given method has a name that is consistent with its body by comparing the overlap between the set of method names that are close in the name vector space and the set of methods names whose bodies are close in the body vector space. When the overlap is Ø, the name is considered to be inconsistent with the body code and suggested with alternative consistent names.

Before explaining the details of these two phases, we first describe an essential step of data processing that adapts to the settings of code constructs.

A. Data Preprocessing

This step aims at preparing the raw data of a given method to be fed into the workflow of our approach. We consider the textual representations of code and transform them into tokens (i.e., basic data units) which are suitable for the deep representation learning techniques described in Section II. Given that method names and bodies have different shapes (i.e., names are about natural language descriptions while bodies are focused on code implementations of algorithms), we propose to use tokenization techniques adapted to each:

- **Method name tokenization:** Method names are broken into sub-token sequences based on camel case and underscore naming conventions, and the obtained sub-tokens are brought to their lowercase form. This strategy has been proven effective in prior studies [7], [28], [29], [31], [32], [60]. For example, method names `findField` and `find_field` are tokenized into the same sequence `[find, field]`, where `find` and `field` are respectively the first and second sub-tokens of the names.

- **Method body tokenization:** Method bodies are converted into textual token sequences by following the code parsing method proposed in our previous study [61]: this method consists in traversing the abstract syntax tree (AST) of a method body code with a depth-first search algorithm to collect two kinds of tokens: AST node types and raw code tokens. For example, the declaration statement “int a;” will be converted into a four-token sequence: `[PrimitiveType, int, Variable, a]`. Since noisy information of code (e.g., non-descriptive variable names such as `a`, `b`) can interfere with identifying similar code [62], all local variables are renamed as the concatenation of their data type with the string `Var`. Eventually, the previous declaration code will be represented by the sequence: `[PrimitiveType, int, Variable, intVar]`.

B. Training

This phase takes tokens of method names and bodies in a code corpus to produce two numerical vector spaces that are leveraged to compute similarities, among method names, on the one hand, and among method bodies, on the other hand, for eventually identifying inconsistent names and suggesting appropriate names. Note that the objective is not to train a classifier whose output will be some classification label given a method name or body. Instead, we adopt the idea of unsupervised learning [63] and lazy learning [64] to embed method names and bodies.

Token sequences of method names are embedded into vectors by the paragraph vector technique described in Section II-1 since token sequences of method names resemble sentences describing the methods. In contrast, all tokens in a method body are first embedded into vectors using Word2Vec. The embedded token vectors are then fed to CNNs to embed the whole method body into a vector, which will be used to represent each method body as a numerical vector.
1) Token Embedding for Method Bodies: As shown in Figure 3, tokens of method bodies are embedded into individual numerical vectors before they can be fed to the CNNs. To that end, the token embedding model is built as below:

\[ TV_B = E_W(T_B) \]  

(1)

where \( E_W \) is the token embedding function (i.e., Word2Vec \cite{mikolov2013efficient} in our case) taking as input a training set of method body token sequences \( T_B \). The output is then a token mapping function \( TV_B : TW_B \rightarrow BV_W \), where \( TW_B \) is a vocabulary of method body tokens, and \( BV_W \) is the vector space embedding the tokens in \( TV_B \).

After token embedding, a method body is eventually represented as a two-dimensional numerical vector. Suppose that a given method body \( b \) is represented by a sequence of tokens \( T_b = (t_1, t_2, t_3, \ldots, t_k) \), where \( t_i \in TW_B \), and \( V_b \) is a two-dimensional numerical vector corresponding to \( T_b \). Then \( V_b \) is inferred as follows:

\[ V_b = l(T_b, TV_B) \]  

(2)

where \( l \) is a function that transforms a token sequence of a method body into a two-dimensional numerical vector based on the mapping function \( TV_B \). Thus, \( V_b = (v_1, v_2, v_3, \ldots, v_k) \in BV_W \), where \( v_i = TV_B(t_i) \) and \( BV_W \) is a set of two-dimensional numerical vectors.

Since token sequences of method bodies may have different lengths (i.e., \( k \) could be different for each method body), the corresponding vectors must be padded to comply with a fixed-width input layer in CNNs. Our approach follows the workaround tested by Wang et al. \cite{wang2016} and appends PAD vectors (i.e., zero vectors) to make all vector sizes consistent with the size of the longest one (see Section IV-2 for how to determine the longest one). For example, the left side of Figure 4 (See Section III-B2 for its description) shows how a method body is represented by a two-dimensional \( n \times k \) numerical vector, where \( n \) is the vector size of each token and \( k \) is the size of the longest token sequence of bodies. Each row represents a vector of an embedded token, and the last two rows represent the appended zero vectors to make all two-dimensional vector sizes consistent.

2) Embedding Method Names and Bodies into Vectors: Vector spaces are built by embedding method names and bodies into corresponding numerical vectors. For method names, we feed the sub-token sequences (i.e., one sequence per method name) to a paragraph embedding technique. Specifically, we leverage the paragraph vector with distributed memory (PV-DM) technique \cite{le2014distributed}, which embeds token sequences into a vector space as follows:

\[ NV_{name} = E_{PV}(T_N) \]  

where \( E_{PV} \) is the paragraph vector embedding function (i.e., PV-DM), which takes as input a training set of method name sub-token sequences \( T_N \). The output is a name mapping function \( NV_{name} : T_N \rightarrow V_N \), where \( V_N \) is an embedded vector space for method names. This step is similar to classical word embedding with differences in the mapping relationships. The paragraph vector embeds a token sequence into a vector, while Word2Vec embeds a token into a vector.

For method bodies, we need another mapping function, where the input is a two-dimensional numerical vector for each method body. The output is a vector corresponding to a body. This mapping function is obtained by the formula below:

\[ NV_{body} = E_{BV}(V_B) \]  

(4)

where \( E_{BV} \) is an embedding function (i.e., CNNs) that takes the two-dimensional vectors of method bodies \( V_B \) as training data and produces a mapping function \( NV_{body} \). Note that \( V_B = \{V_{b_1}, V_{b_2}, V_{b_3}, \ldots, V_{b_m}\} \) is obtained by \( l \) (Equation 2), where \( V_{b_i} \ (i \in [1, m]) \) and \( m \) is the size of training data. \( NV_{body} \) is defined as \( NV_{body} : V_B \rightarrow V_{body}' \), where \( V_{body}' \) is an embedded vector space of method bodies. Based on \( NV_{body} \), we defined the body mapping function \( NV_{body} \) as:

\[ NV_{body} : T_B \rightarrow V_{body}' \]  

(5)

where \( NV_{body} \) is the composition of \( l \) and \( NV_{body} \) in Equation 2 and 4, respectively (i.e., \( NV_{body} = (NV_{body} \circ l)(T_b) = NV_{body}(l(T_b)) \)). \( NV_{body} \) takes a token sequence of a method body and returns an embedded vector representing it.

Our approach uses CNNs \cite{lecun1995convolutional} as the embedding function \( E_{BV} \) in Equation 5. Figure 4 shows the architecture of CNNs.
that our approach uses. The input is two-dimensional numeric vectors of method bodies as stated in Section III-B1. The two pairs of convolutional and subsampling layers are used to capture the local features of methods and decrease dimensions of input data. The network layers from the second subsampling layer to the subsequent layers are fully connected, which can combine all local features captured by convolutional and subsampling layers. We select the output of dense layer as the layer to the subsequent layers are fully connected, which two pairs of convolutional and subsampling layers are used to vectors of method bodies as stated in Section III-B1. The that our approach uses. The input is two-dimensional numeric

Note that vectors in the two vector spaces (i.e., $V_N$ and $V_B$) can be indexed by each method name. For a given body vector of a method, we can immediately find its corresponding name vector in the name vector space, and vice versa. This index facilitates the search of corresponding method names after locating similar method bodies for a given method.

C. Identification & Suggestion

This phase consists of two sub-steps. First, the approach takes a given method as a query to inconsistency identification. By leveraging the two vector spaces (i.e., $V_N$ and $V_B$) and the two embedding functions (i.e., $NV_{name}$ and $NV_{body}$), it identifies whether the name of the given method is consistent with its body. Second, if the name turns out to be inconsistent, the approach suggests potentially consistent names for it from the names of similarly implemented methods.

1) Inconsistency Identification: For a given method $m_i$, we can take a set of adjacent vectors for its name ($n_i$) and body ($b_i$), respectively (i.e., $adj(n_i)$ and $adj(b_i)$). After retrieving the actual names (i.e., $name(*)$) corresponding to vectors in $adj(n_i)$ and $adj(b_i)$, we can compute the intersection between the two name sets as $C_{full}$:

$$C_{full} = name(adj(n_i)) \cap name(adj(b_i))$$ (6)

If $C_{full}$ is $\emptyset$, we consider $b_i$ to be inconsistently named $n_i$.

However, $C_{full}$ in Equation 6 is too strict since it relies on exact matching. In other words, there should exist the same character sequences between two name sets. For example, suppose that there is $findField$ in $name(adj(n_i))$ and $findElement$ in $name(adj(b_i))$ with similar implementations. This relationship cannot be identified by $C_{full}$ even if they have similar behavior of looking up something.

In the Java naming conventions [66], the first sub-token often indicates the key behavior of a method [67] (e.g., `get[ ]().contains[ ]()`). Thus, if the key behavior of a given method is similar to those of other methods with similar bodies, we can regard that the name is consistent. Thus, we relax the condition of consistency. Instead of comparing the full name, we take the first sub-token of each name in the two name sets to get the intersection as below:

$$C_{relaxed} = first(name(adj(n_i))) \cap first(name(adj(b_i)))$$ (7)

where $first(*)$ is a function that obtains the first sub-token set by the same rule of method name tokenization described in Section III-A. Other subsequent tokens are often highly project-specific. Therefore, those subsequent tokens would be different across projects even if their bodies are highly similar.

Algorithm 1: Inconsistency identification and new names suggestion.

```
Input: target method (name and body): $m_i = (n_i, b_i)$
Input: threshold of adjacent vectors: $k$
Input: set of name vectors obtained from a training set: $V_N$
Input: set of body vectors obtained from a training set: $V_B$
Input: indexes of actual names from all vectors $V_N$ or $V_B$: $Idx_{name}: V \rightarrow N$
Input: function embedding a name to a vector: $NV_{name}$
Input: function embedding a body to a vector: $NV_{body}$
Output: pair of the consistency determinant of $m_i$ (Boolean) and a set of suggested names: $(c, SG_{n_i})$, where $SG_{n_i}$ is $\emptyset$ if $c$ is false.

1 Function identify($m_i$, $V_N$, $V_B$)
  2 // compute name and body vectors of $m_i$, $V_{n_i}$ and $V_{b_i}$
  3 $V_{n_i} = NV_{name}(n_i)$;
  4 $V_{b_i} = NV_{body}(b_i)$;
  5 // get adjacent name vectors similar to the name vector ($V_{n_i}$) of $m_i$
  6 Name$V_{adj} = getTopAdjacent(V_{n_i}, V_N, k)$;
  7 // get actual names for adjacent name vectors (Name$V_{adj}$).
  8 Names$^\prime_1 = NameV_{adj}.collect(Idx_{name}(\forall V \in NameV_{adj}))$;
  9 // get adjacent body vectors similar to the body vector ($V_{b_i}$) of $m_i$
 10 Body$V_{adj} = getTopAdjacent(V_{b_i}, V_B, k)$;
 11 // get actual names for adjacent body vectors (Body$V_{adj}$).
 12 Names$^\prime_2 = BodyV_{adj}.collect(Idx_{name}(\forall V \in BodyV_{adj}))$;
 13 // take the first tokens of actual names for adjacent name and body vectors.
 14 $fT_{V_{adj}} = Names^\prime_1.\{ adjacent.get\{tokenize\}\}(\forall N \in Names^\prime_1)$
 15 $fT_{V_{adj}} = Names^\prime_2.\{ adjacent.get\{tokenize\}\}(\forall N \in Names^\prime_2)$;  
 16 if $fT_{V_{adj}} \cap fT_{V_{adj}} = \emptyset$ then
 17   // $m_i$ has an inconsistent name and suggest new names.
 18   newNames := rankNames(Name$^\prime_1$, Body$V_{adj}$);
 19   $(c, SG_{n_i}) := (false, newNames)$;
 20 else
 21   // $m_i$ has a consistent name.
 22   $(c, SG_{n_i}) := (true, \emptyset)$;
```

Algorithm 1 details the precise routine for checking whether the name $n_i$ of a method is consistent with its body $b_i$ or not. Our approach computes the cosine similarity for a given method to search for similar methods. After retrieving the embedded vectors of the name $n_i$ and body $b_i$ (cf. lines 3 and 4), the approach looks up the top $k$ adjacent vectors in the respective vectors spaces for method names and bodies (cf. lines 6 and 10). Since threshold $k$ can affect the performance of identification, our evaluation described in Section V includes an experiment where $k$ values are varied.

After remapping the set of adjacent vectors to sets of the corresponding method names (cf. lines 8 and 12), the sets are processed to keep only first sub-tokens (cf. lines 14 and 15), since our approach uses $C_{relaxed}$ as specified in Equation 7 to compare the two sets of first tokens, $fT_{V_{adj}}$ and $fT_{V_{adj}}$ (cf. line 16). If their intersection is $\emptyset$, the approach suggests names for the given method body $b_i$ (cf. Section III-C2 for details). Otherwise, our approach assumes that $n_i$ is consistent with $b_i$.

2) Name Suggestion: Our approach suggests new names for a given method by providing a ranked list of the similar names (cf. line 18), with four ranking strategies as below:

- **R1**: This strategy purely relies on the similarities between method bodies. The names of similar method bodies ($Names^\prime_{adj}$) are ranked by the similarities to the given method body (between $V_{b_i}$ and Body$V_{adj}$).

- **R2**: This strategy first groups the same names in $Names^\prime_{adj}$ since there might be duplicates. It then ranks distinct names based on the size of the associated groups. Ties are broken based on the similarities between method bodies as R1.
• R3: Similarly to R2, this strategy groups the same names in Names$^{	ext{adj}}$. Then, the strategy computes the average similarity to $b_i$ of each group and ranks the groups based on the average similarity, but the group sizes are not considered.

• R4: To avoid having highly ranked groups with a small size as per strategy R3, this strategy eventually re-ranks all groups produced in R3 by downgrading all 1-size groups to the lowest position.

IV. EXPERIMENTAL SETUP

Empirical validation of the approach is performed through various experiments. Before describing the results and conclusions, we present the research questions and the data collection as well as details on the parameter settings in implementation to facilitate replication.

1) Research Questions: To evaluate the approach, we propose to investigate the following research questions (RQs):

• RQ1: How effectively does the approach identify inconsistent method names?
• RQ2: Can the approach suggest accurate method names?
• RQ3: How does the approach compare with the state-of-the-art in terms of performance?
• RQ4: To what extent applying the approach in the wild produces debugging suggestions that are acceptable to developers?

2) Data Collection: We collect both training and test data from open source projects from four different communities, namely Apache, Spring, Hibernate, and Google. We consider 430 Java projects with at least 100 commits, to ensure that these have been well-maintained.

Training data is constituted by all methods of these projects, after filtering out noisy data with criteria as below:

• main methods, constructor methods, and example methods\(^2\) are ignored since they have the less adverse effect on program maintenance and understanding, and can pollute the results of searching for similar methods.

• Empty methods (i.e., abstract or zero-statement methods) have no implementation and thus are filtered out.

• Methods names without alphabetic letters (e.g., some methods are named “−”) are removed as they are undescriptive.

As a result, 2,425,939 methods are collected.

In practice, we further limit the training data to methods with reasonable size, to avoid the explosion of code tokens which can degrade performance. The sizes of token sequences of collected method bodies range from 2 to 60,310. According to the sizes’ distribution shown in Figure 5, most methods have less than 100 tokens. We focus on building the training data with methods containing at most 94 tokens, which is set based on the upper whisker value\(^3\) from the boxplot distribution of method body token sequence sizes in Figure 5. The sizes beyond the upper whisker value are considered as outliers [68]. Eventually, 2,116,413 methods are selected to be the training data, as indicated in Table IV-2. Note that methods in the training data are not labeled as consistent or inconsistent since the objective of training is to construct a vector space of methods with presumable\(^4\) consistent names.

Test data is the oracle that we must constitute to assess the performance of our proposed approach. We build it by parsing the commit history of our subjects (i.e., 430 projects). Specifically, we consider:

• Methods whose names have been changed in a commit without any modification being performed on the body code; and

• the names and body code have become stable after the change (i.e., no more changes up to the current versions).

The first criterion allows to ensure that the change is really about fixing method names, and to retrieve their buggy and fixed versions. Overall, within commit changes, we identified 53,731 methods satisfying this criterion. The second criterion increases the confidence that the fixed version of the name is not itself found buggy later on. With this criterion, the number is reduced to 8,734 methods. We further observe that some method names are changed due to simple typos (cf. Figure 6). Such changes can constitute noise in the oracle. Given that our approach heavily relies on first sub-tokens of method names to hint at inconsistency, we conservatively ignore all change cases where this part is not changed. At this stage, the dataset still includes 4,445 buggy-fixed pairs of method names. The final selection follows the criterion used for collecting training data (i.e., no constructor or example methods, etc.). The final test data includes 2,805 distinct methods.

Fig. 5. Sizes’ distribution of collected method body token sequences.

| Classification | # Methods |
|----------------|-----------|
| All collected methods | 2,425,939 |
| Methods after filtering (for training) | 2,116,413 |
| Methods for testing | 2,805 |

| TABLE I |
| SIZE OF DATASETS USED IN THE EXPERIMENTS. |

\(^2\)The package, class or method name includes the keyword “example”, “sample” or “template”.

\(^3\)The upper whisker value is determined by 1.5 IQR (interquartile ranges) where IQR = 3rd Quartile − 1st Quartile, as defined in [68].

\(^4\)The majority of the methods in the world have names that are likely to be consistent with their bodies.
3) Implementation of Neural Network Models: The Paragraph Vector, Word2Vec, and CNNs models are implemented with the open source DL4J library [70], which is widely used across the research and practice in deep learning (800k+ people and 90k+ communities according to data on the Gitter networking platform [71]). These neural networks must be tuned with specific parameters. In this study, all parameters are set following the parameters setting proposed by Kim [44] and our previous work [61]. Their subjects and models are similar to ours, and their yielded models were shown to achieve promising results. Tables II, III, and IV show the parameters used in our experiment for each model.

| Parameters Setting of Paragraph Vector. | Parameters | Values |
|----------------------------------------|------------|--------|
| Min word frequency                     | 1          | Size of vector |
| Learning rate                           | 0.025      | Window size |
|                                       |            | 300     |
|                                       |            | 2       |

| Parameters Setting of Word2Vec. | Parameters | Values |
|---------------------------------|------------|--------|
| Min word frequency              | 1          | Size of vector |
| Learning rate                    | 1e-2       | Window size |
|                                 |            | 300     |
|                                 |            | 4       |

| Parameters Setting of CNNs. | Parameters | Values |
|----------------------------|------------|--------|
| # nodes in hidden layers    | 1000       | learning rate |
| activation (output layer)   | softmax    | 1e-2    |
| activation (other layers)   | ReLU       | max pool |
| optimization algorithm      | stochastic gradient descent | |
| loss function               | mean absolute error | |

V. EVALUATION

A. RQ1: Effectiveness of Inconsistency Identification

As the first objective of our approach is to identify inconsistent method names, we examine whether our approach effectively identifies methods with inconsistent names. We train the model with the collected training data and apply it to the separated test data whose collection was described in Section IV-2. Given that the performance of identification depends on the threshold value \( k \) representing the size of the sets of adjacent vectors (Lines 6 and 10 in Algorithm 1), we vary \( k \) as 1, 5, and \( n \times 10 \) with \( n \in [1, 10] \).

Inconsistent identification is a binary classification since the test data explained in Section IV-2 are labeled in two classes (IC: inconsistent; C: consistent). Thus, there are four possible outcomes: IC classified as IC (true positive=TP), IC classified as C (false negative=FN), C classified as IC (true negative=TN), and C classified as C (false positive=FP). We compute Precision, Recall, F1-measure and Accuracy for each class. The precision and recall for the class IC are defined as \( \frac{TP}{TP+FP} \) and \( \frac{TP}{TP+FN} \), respectively. Those for the class C are defined as \( \frac{TN}{TN+FN} \) and \( \frac{TN}{TN+FP} \), respectively. The F1-measure of each class is defined as \( \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \), while the Accuracy is defined as \( \frac{TP+TN}{TP+TN+FP+FN} \).

Table V provides the experimental results on the performance. Due to space limitation, we show the metrics for variations of \( k \) up to 40 (instead of 100). Overall, our approach yields an Accuracy metric ranging from 50.8% to 60.9% (for the presented results) and an F1-measure up to 67.9% for the inconsistent class. In particular, the approach achieves the highest performance when \( k=1 \) (i.e., the single most adjacent vector is considered). The general trend indeed is that the performance decreases as \( k \) is increased.

| Evaluation metrics | k = 1 | k = 5 | k = 10 | k = 20 | k = 30 | k = 40 |
|--------------------|-------|-------|--------|--------|--------|--------|
| Precision (%)      | 56.3  | 53.7  | 53.3   | 53.3   | 49.9   | 49.7   |
| Recall (%)         | 54.5  | 54.4  | 49.7   | 49.7   | 36.5   | 40.1   |
| F1-measure (%)     | 67.9  | 54.8  | 47.9   | 47.9   | 36.5   | 40.1   |
| Consistent         |       |       |        |        |        |        |
| Precision (%)      | 72.0  | 53.9  | 54.2   | 54.2   | 31.4   | 51.4   |
| Recall (%)         | 52.7  | 53.7  | 60.7   | 60.7   | 72.2   | 67.4   |
| F1-measure (%)     | 49.9  | 54.8  | 57.3   | 57.3   | 60.0   | 58.3   |
| Accuracy (%)       | 60.9  | 54.8  | 53.8   | 50.8   | 30.9   | 51.1   |

Since \( k \) determines the number of similar methods retrieving from the vector spaces of names and bodies in the training data, higher \( k \) value increases the probability of non-empty intersection (i.e., \( T_{adj}^{i} \cap T_{adj}^{b} \), cf. line 16 in Algorithm 1). Thus, the recall of the inconsistent class tends to decrease as \( k \) is getting higher. In contrast, the recall of the consistent class increases for higher values of \( k \).

Overall, the approach to identifying inconsistent method names can be tuned to meet the practitioners’ requirements. When the criteria are to identify as many inconsistent names as possible, \( k \) should be set to a low value.

B. RQ2: Accuracy in Method Names Suggestion

This experiment aims at evaluating the performance of our approach in suggesting new names for identified inconsistent names. The suggested names are ranked by a specified ranking strategy (either R1, R2, R3, or R4 as described in Section III-C2).

Prior studies compare the first tokens of suggested names [28], [72], [73] and oracles or token sets without considering token ordering [32], [74]. To ensure fair and comprehensive assessment, we consider three different scenarios to evaluate the performance of our approach. These scenarios are defined as follows:

- T1 (Inconsistency avoidance): In this scenario, we evaluate to what extent the suggested names are different from the input buggy name \( n_{i} \). The accuracy in this scenario is computed by \( 1 - \frac{\sum_{n \in D} D(n_{i}^{1}, f(T_{adj}^{b}))}{|n_{i}|} \), where \( n_{i}^{1} \) is the first token of \( n_{i} \). \( f() \) collects the first tokens of consistent method names corresponding to each vector in \( T_{adj}^{b} \) (similar body vectors shown at Line 10 in Algorithm 1). \( D(*) \) checks whether the items in the second argument contain the first argument (if so, returns 1. Otherwise 0). \( i \) is the set of inconsistent method names identified by our approach.

- T2 (First-token accuracy): In this scenario, we evaluate to what extent the suggested names are identical to the name that developers proposed in debugging changes. We remind the reader that our test data indeed include pairs of buggy/fixes names extracted from code changes history (cf. Section IV-2). The accuracy is computed as \( \frac{\sum_{n \in D} D(n_{i}^{1}, f(T_{adj}^{b}))}{|n_{i}|} \), where \( n_{i}^{1} \) is the first token of the actual developer-fixed version of the name.
• **T3** (Full-name accuracy): In this scenario, we evaluate to what extent the full name of each suggested name is identical to the name that developers proposed in debugging changes. The accuracy in this scenario is computed as \(\frac{\sum_{i=1}^{n} D(\text{rn}(n_i), \text{fn}(|\text{BodyV}_{adj}|))}{|\text{cn}|}\), where \(\text{rn}(n_i)\) is the actual fixed version of the name and \(\text{fn}(\cdot)\) retrieves the full names of methods corresponding to each vector in \(\text{BodyV}_{adj}\).

We perform different experiments, varying \(k\). For these experiments, the approach produces new names for all methods identified as inconsistent (i.e., true_positive + false_positive). Thus, the results include the performance of false positives (i.e., names that are already consistent). We compute the performance based on the number of suggested names (varying the threshold value \(\text{thr}\)). For the ranking strategy \(R_1\), \(k\) is set only to 1 or 5 since higher values do not affect the results when \(\text{thr} = 1\) or 5 (note that \(R_1\) produces the same number of suggested names with \(k\)). For other ranking strategies, \(k\) is set to 10, 20, 30, and 40 to have large numbers of suggested names that can be aggregated (as per ranking strategy working).

According to the results for \(T_1\) listed in Table VI, our approach is highly likely to suggest names that are different from the identified inconsistent names with ranking strategies, even when \(k\) is high (>90% if \(\text{thr}=1\) and >60% if \(\text{thr}=5\)). When matching the first tokens (cf. results for \(T_2\)) and the full name (cf. results for \(T_3\)), the ranking strategy \(R_4\) slightly outperforms others regardless of the \(k\) value, while its accuracy is ≈40%. \(\text{thr}=5\) gives a better probability to find consistent names than \(\text{thr}=1\). \(k=10\) yields the best performance for ranking strategies \(R_2\) and \(R_3\) while ranking strategy \(R_4\) performs best when \(k=20\) and \(\text{thr}=5\) and \(k=40\) and \(\text{thr}=1\).

| Accuracy (%) | \(k=\text{thr}\) | \(k=10\) | \(k=20\) | \(k=30\) | \(k=40\) |
|--------------|----------------|---------|---------|---------|---------|
| \(T_1\) | R1 | R2 | R3 | R4 | R1 | R2 | R3 | R4 | R1 | R2 | R3 | R4 |
| thr=1 | 90.0 | 91.2 | 91.3 | 97.6 | 92.2 | 92.2 | 75.9 | 93.0 | 92.9 | 75.8 | 93.7 | 93.6 | 75.9 |
| thr=5 | 69.0 | 66.4 | 66.4 | 66.1 | 64.0 | 64.0 | 61.8 | 64.9 | 64.9 | 61.9 | 66.3 | 66.1 | 61.5 |
| \(T_2\) | R1 | R2 | R3 | R4 | R1 | R2 | R3 | R4 | R1 | R2 | R3 | R4 |
| thr=1 | 23.4 | 23.2 | 23.0 | 24.0 | 21.5 | 21.5 | 24.1 | 19.3 | 19.2 | 24.0 | 17.2 | 17.2 | 24.2 |
| thr=5 | 35.7 | 39.4 | 39.4 | 39.7 | 38.5 | 38.6 | 40.8 | 37.3 | 37.2 | 40.6 | 36.5 | 36.3 | 40.1 |
| \(T_3\) | R1 | R2 | R3 | R4 | R1 | R2 | R3 | R4 | R1 | R2 | R3 | R4 |
| thr=1 | 10.7 | 11.0 | 10.9 | 10.9 | 10.9 | 10.9 | 11.1 | 10.6 | 10.5 | 11.3 | 10.2 | 10.1 | 11.5 |
| thr=5 | 17.0 | 18.7 | 19.0 | 19.2 | 17.7 | 17.8 | 19.5 | 16.9 | 16.9 | 19.4 | 16.6 | 16.6 | 19.2 |

The best case of each ranking strategy in each row is highlighted as **bold**.

The results above show that: (1) ranking strategies \(R_2\), \(R_3\), and \(R_4\) perform better than \(R_1\) but they need more candidates, (2) higher \(k\) values do not increase the accuracy of name suggestion, and (3) more number of suggested names (i.e., higher \(\text{thr}\) values) would improve the accuracy but users of our approach will need to look up more names.

Note that it is promising that achieving ≈20% and ≈40% accuracy (for first-token / \(T_2\)) when looking up only top-1 and top-5 suggestions, respectively. Suggesting exact first tokens of method names is challenging since there are a large number of available words for the first tokens of method names. Finding the exact full name of a method is even more challenging since full method names are often very project-specific [53]. Our approach achieves ≈10% and ≈20% accuracy, respectively for \(\text{thr}=1\) and \(\text{thr}=5\).

### C. RQ3: Comparison Against the State-of-the-Art Techniques

We compare our approach with two state-of-the-art approaches in the literature which are based on the n-gram model [75] and the convolutional attention network (CAN) model [32]. The latter includes two sub-models: copy_attention, which uses only the pre-trained vocabulary, and copy__attention, which can copy tokens of input vectors (i.e., tokens in a method body). These techniques are selected since they are the most recent approaches for method name debugging. Given that the n-gram model approach by Suzuki et al. [75] cannot suggest full names or even the first token of methods, we compare against them with respect to the performance of inconsistent name identification. The CAN model, on the other hand, does not explicitly identify name inconsistency. Instead, the model suggests names for any given method. Thus, in this experiment, we make the CAN model and our approach suggest names for all test data (2,805 buggy method names). For both techniques, we use the same training data described in Section IV-2. It should be noted that while the tool for the CAN model has been made available by the authors, we had to replicate the n-gram models approach, in a best effort way following the details available in [75].

Table VII shows the comparison results with the n-gram model [75]. While the performance of the n-gram model stays in a range from 51.5-54.2% for all measures, our approach outperforms the model when \(k=1\) and 5. With \(k=1\), the improvement is up to 33 percentage points. In particular, our approach achieves a higher F1-measure by 15 percentage points.

| Evaluation Metrics | Our Approach | n-gram Model |
|--------------------|--------------|--------------|
| \(k=1\) | \(k=5\) | \(k=10\) | \(k=5\) | \(k=10\) |
| Precision | 56.8% | 55.7% | 53.3% | 53.3% | 53.3% |
| Recall | 84.5% | 55.9% | 46.7% | 51.5% | 51.5% |
| F1-measure | 67.9% | 54.8% | 49.7% | 52.4% | 52.4% |
| Accuracy | 60.9% | 54.8% | 53.8% | 54.2% | 54.2% |

To compare our approach against the CAN model [32], we propose two evaluations. The first follows the evaluation strategy proposed by the authors themselves in their paper. The second evaluation is based on our own strategies already explored for RQ2 (cf. Section V-B).

Table VIII shows the performance based on the per-sub-token basis metric, which is the evaluation metric used by the authors originally to present the performance of the CAN model [32]. This metric estimates to what extent sub-tokens of method names can be correctly suggested without considering their order within the method names. We compute precision, recall, and F1-measure of correctly suggesting sub-tokens.

When applying the per-sub-token basis, our approach outperforms the CAN model in all configurations, except for the precision of copy_attention with \(\text{thr}=5\) (cf. Section V-B). While the precision of our approach can be higher by up to 7 percentage points, we achieve substantial performance improvement in terms of recall and F1-measure, with up to 15 percentage points margin.

### TABLE VIII

**ACCURACY OF SUGGESTING METHOD NAMES WITH THE FOUR RANKING STRATEGIES (i.e., R1, R2, R3, and R4).**

### TABLE VII

**COMPARISON RESULTS OF IDENTIFYING INCONSISTENT METHOD NAMES AGAINST THE N-GRAM MODEL [75].**
Table IX presents the comparison results when applying the three evaluation strategies of RQ2, described in Section V-B. Regardless of the evaluation strategy, our approach outperforms the CAN models. Notably, our approach achieves 16–25% accuracy for T3 (i.e., full name suggestion) while the CAN model is only successful for at most 1.1%. Note that specific values of accuracy in Table IX are different from Table VI since, in the experiment for RQ3, our approach suggests names for all test data.

D. RQ4: Live Study

To investigate practicability of our approach to debugging inconsistent method names (RQ4), we conduct a live study on active software projects: we submit pull requests of renaming suggestions from our approach, and assess acceptance rates. For this experiment, we randomly sample 10% of the training data to be used as test data. Indeed the labeled test data collected for previous experiments represent cases where developers debugged the method names. The remaining 90% of method names now constitute the training data for this phase. We apply this version of our approach to the target subjects to identify whether they have inconsistent names (using \( k=20 \) as per result of previous experiments). Overall, 4,430 methods among the 211,642 methods in the test set have been identified as inconsistent by our approach. Given that we cannot afford to spam project maintainers with thousands of pull requests, we randomly select 100 cases of identified inconsistent methods. We then collect the ranked list of suggested names for each of the 100 methods: we use \( \text{thr}=5 \) with ranking strategy R4 since these parameters show the best performance for full name suggestion (T3). From each ranked list of suggested names, we select the top-1 name and prepare a patch that we submit as a pull request to the relevant project repository.

As indicated in Table X, developers agreed to merge the pull requests for renaming 40 out of the 100 methods. 26 renaming suggestions have been validated and approved (based on developers’ reply) by developers, but the pull requests have not been merged (as of submission date) since some projects systematically apply unit test and complete review tasks of external changes before accepting them into the main branch. Four inconsistent method names have also been fixed after improving our suggested names. Interestingly, one developer used our suggestion as a renaming pattern to fix six (6) similar cases other than the ones submitted in our pull requests. Furthermore, some developers have welcomed our suggestions on inconsistent method names and showed interest in applying even more suggestions from our approach, given that it seems to provide more meaningful names than their current names.

We also report on cases where developers did not apply our suggested name changes. In 1 case, the developers could not merge the pull request as it would break the program: the method is actually an overridden method from another project. The developer nevertheless agreed that our suggested name was more intuitive. For 2 methods, developers agree that the suggested names are appropriate but they would not make the changes as the names are not in line with inner-project naming conventions. For nine methods, however, the pull requests are rejected since developers judge the original method names to be more meaningful than the suggested ones. The remaining 18 cases are simply ignored: we did not receive any reply up to the date of submission. We summarize developers’ feedback as follows:

1) Some method names should follow the naming convention of specific projects. This is a threat to the validity of our study since it is implemented in a cross-project context.
2) Some method should be named considering the class names. E.g., in a class named “XXXBuilder”, the developers do not want to name a method as “build”, although the method builds a new “XXXBuilder” object.

VI. DISCUSSION

1) Naming based on Syntactic and Semantic Information: As stated in Section I, our approach is based on the assumption that similar method implementations might be associated with similar method names. However, there could be several different definitions of similarity. While our approach relies on the syntactic similarity of method bodies (although, using AST tokens), one can use dynamic information (e.g., execution traces) to compare different method implementations as experimented for the detection of semantic (i.e., type-4) code clones [76], [77]. However, obtaining dynamic information is
Several lexical categories. Liblit can be decomposed into fragments and further classified into syntactical and semantic structure, and reported that identifiers Tonella [82] analyzed function identifiers from the lexical, make code harder to understand and maintain. Caprile and on code identifiers and concluded that inconsistent names can enlarge test data, but it could bias the assessing results. Thus, collecting more actual fixed method names from other projects are included in our future work.

VII. RELATED WORK

There have been several empirical studies [67], [79]–[81] investigating the impact of a naming scheme on program comprehension, readability, and maintainability. Takang et al. [12] and Lawrie et al. [14] conducted empirical studies on code identifiers and concluded that inconsistent names can make code harder to understand and maintain. Caprile and Tonella [82] analyzed function identifiers from the lexical, syntactical and semantic structure, and reported that identifiers can be decomposed into fragments and further classified into several lexical categories. Liblit et al. [13] examined how human cognition is reflected in naming things of programs.

Several approaches have been presented to detect inconsistent identifiers. Deissenboeck and Pizka [8], and Lawrie et al. [83] relied on the manual mapping between names and domain concepts to detect inconsistent identifiers in code. Binkley et al. [73] developed a tool with part-of-speech tagging to identify field identifiers that violate accepted patterns. Even after automatically detecting inconsistent names, developers may have difficulties in debugging or refactoring inconsistent names. The ultimate goal of debugging names is to automatically replace inconsistent names into consistent ones rather than just helping identifier naming. Haiduc et al. [84] used natural language summarization techniques and the lexical and structural context in code to improve code comprehension. Sridhara et al. [86] designed an automatic technique for summarizing code with the idioms and structure in a method. Lucia et al. [87] proposed an IR-based approach to improve program comprehension with the textual similarity between the code under development and related artefacts. Høst and Østvold [28] used method naming rules and semantic profiles of method implementations to debug method names.

Recently, Allamanis et al. [31], [32] leveraged deep learning techniques to suggest method names with local contexts, which are similar to this paper on embedding method names and bodies. Their work learns method body features from code sub-tokens, this paper further consider code nodes at abstract syntax tree level since they can capture code semantic information [48]. This paper just compared against the work [32] since both of them are from the same group and the work [32] presents two more advanced models. This paper performed various evaluations on the actual fixed method names which were not done in [31], [32], with a large sample of 430 projects against the 20/10 projects in their work. In addition, this paper tried various configurations and strategies and used various indicators for method name suggestions, which was not exactly the same as they did. Thus, our conclusions are likely to be more solid than those in their work. Furthermore, we performed a live study (not done in their work) and showed the technique has strong potential to be useful by actually fixing 66 inconsistent method names in the wild.

VIII. CONCLUSION

Method names are key to readable and maintainable code, but it is not an easy task to give an appropriate name to a method. Thus, many methods have inconsistent names, which can impede the readability and maintainability of programs and even lead to some defects. To reduce the manual efforts of resolving inconsistent method names, we propose a novel approach to debugging inconsistent method names by leveraging similar methods with deep learning techniques. Our experimental results show that the performance of our approach achieves an F1-measure of 67.9% on identifying inconsistent method names, improving about 15 percentage points over the state-of-the-art. On suggesting appropriate first sub-tokens and full names for inconsistent method names, it achieves 34%–50% and 16%–25% accuracy respectively, outperforming the state-of-the-art as well. We further report that our approach helps developers to fix 66 inconsistent method names in the wild. The tool and data of in our study are available at https://github.com/SerVal-DTF/debug-method-name.

ACKNOWLEDGEMENTS

This work is supported by the Fonds National de la Recherche (FNR), Luxembourg, under projects RECOMMEND 15/IS/10449467 and FIXPATTERN C15/IS/9964569.
