Deep Learning Based Identification of Suspicious Return Statements

Guangjie Li, Hui Liu*, Jiahao Jin, and Qasim Umer
School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China
3120150428@bit.edu.cn, liuhui08@bit.edu.cn, jinjiahao1993@gmail.com, qasimumer667@hotmail.com

***Abstract*—Identifiers in source code are composed of terms in natural languages. Such terms, as well as phrases composed of such terms, convey rich semantics that could be exploited for program analysis and comprehension. To this end, in this paper we propose a deep learning based approach, called *MLDetector*, to identifying suspicious return statements by leveraging semantics conveyed by the natural language phrases that are used as identifiers in the source code. We specially design a deep neural network to tell whether a given return statement matches its corresponding method signature. The rationale is that both method signature and return value should explicitly specify the output of the method, and thus a significant mismatch between method signature and return value may suggest a suspicious return statement. To address the challenge of lacking negative training data, i.e., incorrect return statements, we generate negative training data automatically by transforming real-world correct return statements. To feed code into neural network, we convert them into vectors by *Word2Vec*, an unsupervised neural network based learning algorithm. We evaluate the proposed approach in two parts. In the first part, we evaluate it on 500 open-source applications by automatically generating labeled training data. Results suggest that the precision of the proposed approach varies from 83% to 90%. In the second part, we conduct a case study on 100 real-world applications. Evaluation results suggest that 42 out of 65 real-world incorrect return statements are detected (with precision of 59%).**

***Index Terms*—Program Analysis, Code Quality, Bug Detection, Identification, Deep Learning, Return Value**

I. Introduction

Identifiers in source code are composed of terms in natural languages. Such terms, as well as phrases composed of such terms, convey rich semantics that could be exploited for program analysis and comprehension. It has been demonstrated that natural language information embedded in identifiers can be used to decrease the cost of software maintenance [1] [2]. Despite the importance of identifiers, many program analysis tools ignore the information conveyed in identifiers [3] [4].

To exploit information embedded in identifiers, researchers propose name-based analysis to conduct code recommendation [3] [5], to mine code idioms [6], to infer API specifications [7] [8], to detect argument defects [9] [4], to detect the mismatch between entity names and entity bodies [10] [11] [12] [13]. However, existing bug detection tools don't fully exploit the rich semantics conveyed in identifiers [4]. They often ignore information in identifiers [14] [15], or only rely on the lexical information in identifiers by designing manually tuned heuristics [3] [11] [16] [17] [18]. Such heuristics-based approaches are often ad-hoc, and cannot detect valid but uncommon patterns from source code. Consequently, it could be challenging for existing tools to identify some bugs that are obvious to a human.

In this paper, we leverage a deep learning-based approach to detect incorrect return statements in method declarations by leveraging semantics conveyed by the natural language phrases that are used as identifiers in source code. The key idea of the proposed approach is that suspicious return statements could be detected based on the mismatch between return statements and method signatures. The insight of the proposed approach is that vector representations of identifiers preserve the semantic similarities between identifiers and deep learning techniques learn generic quantitative rules of distinguishing incorrect code from the correct one. Neural networks and deep learning techniques have been proved good at building complex mappings from input into output automatically, and they have been successfully applied in different domains, e.g., natural language processing tasks [19], video processing [20], and program analysis [10] [21] [22]. Furthermore, to filter out false positives and improve the precision of the proposed approach, we extract features from source code and combine them with natural language information embedded in identifiers to train the neural network.

To feed textual data into the neural network, we convert identifiers (i.e., method names, return statements, and candidate return value) into vectors by *Word2Vec* [23] [24], an unsupervised neural network based learning algorithm. With the resulting vectors, the proposed approach trains a deep learning based classifier to distinguish incorrect code from the correct one.

We train the deep learning neural network with both correct code and incorrect code that makes the neural network efficient in learning and classifying. The proposed approach mines the semantic similarities between identifiers by representing them in high dimensional vector space, where similar identifiers are assigned to similar locations. It is the first work to detect incorrect return statements based on the semantic similarities between identifiers. However, one of the biggest challenges in training the neural network effectively is to collect a big dataset containing both correct and incorrect code. Consequently, in this paper, we download a large amount of (500) popular open-source Java applications from the well-known community GitHub [25], and hypothesize that original code from popular applications is correct. Meanwhile, since

*Hui Liu is the corresponding author (liuhui08@bit.edu.cn).

978-1-7281-5143-4/20/$31.00 © 2020 IEEE

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on August 12,2024 at 01:08:54 UTC from IEEE Xplore. Restrictions apply.

it is challenging to collect incorrect code examples from real-world code, we propose to artificially generate incorrect code examples by simply swapping the order of return statements and candidate return values of correct code.

To investigate how often incorrect return statements that exist in real-world applications could be identified by the proposed approach, we conduct a case study on 100 popular open-source Java application by parsing commits containing terms "return value", "return bug", "return error", "return bug", or "fix return", and manually comparing the results against the warnings identified by the proposed approach. The results of the case study suggest that the proposed approach identifies 42 out of 65 incorrect return statements with high precision.

We make the following main contributions in this paper:

- A deep learning-based approach to identify incorrect return statements. To the best of our knowledge, we are the first one to apply deep learning techniques to detect suspicious return value.
- Evaluation of the proposed approach on real-world applications suggests that the proposed approach is effective in identifying incorrect return statements, with a precision of 59%.
- We implement the proposed approach into a prototype *MLDetector*, and make it publicly available together with our dataset.

The rest of the paper is structured as follows. Section II presents the approach. Section III evaluates the proposed approach. Section IV presents a case study of application. Section V discusses the threats. Section VI introduces related works. Section VII makes conclusion and envision future work.

II. Approach

In this section, we propose a deep learning-based approach to identify incorrect return statements. The key rationale of the proposed approach is that both method signature and return value should explicitly specify the output of the method, and thus a significant mismatch between method signature and return statement may suggest a suspicious return value. Consequently, we propose a deep learning-based approach to distinguish suspicious incorrect return statements from the correct ones by leveraging semantics conveyed in identifiers extracted from the method signature and features extracted from source code.

A. Overview

An overview of the proposed approach is presented in Fig. 1. We first statically extract code examples from a large corpus of software applications and use them as positive items. Second, we generate negative items by swapping the order of return statements and candidate return values of positive items. Third, we train a *Word2Vec* neural network to map identifiers in natural languages into vector representations. Fourth, we train a deep learning-based classifier to distinguish incorrect code from correct one. At the prediction phase, we query the learned neural network to predict whether a given input code is correct or not. The rationale of the proposed approach is that the deep learning-based classifier can identify the mismatch between return statement and method signature based on their similarity in high-dimensional vector space.

B. Feature Selection

To distinguish incorrect return statements from correct ones, we empirically extract the following names from the method signatures and features from source code as the input of our approach:

$$\begin{aligned} input &=< names, features > \\ names &=< n_m, n_r, n_c > \\ features &=< f_t, f_{sim}, f_{len} > \\ f_t &=< t_r, t_c > \\ f_{sim} &=< lexSim(n_m, n_r), lexSim(n_m, n_c) > \\ f_{len} &=< len(n_m), len(n_r), len(n_c) > \end{aligned} \tag{1}$$

From the equation 1, we observe that the selected features are composed of two parts: one is the *names* extracted from the method signatures, the other is the *features* extracted from source code. *names* includes n_m, n_r, and n_c, which represents method name, return value, and the most similar candidate return statement, respectively. The $features$ part includes three kinds of features (i.e., f_t, f_{sim}, and f_{len}) empirically extracted from source code, which represents the types, similarities, and length information, respectively. t_r and t_c represents the type of return statement (e.g., 1 stands for a field, 2 represents method invocation) and the type of most similar candidate return statement, respectively. $lexSim(n_m, n_r)$ and $lexSim(n_m, n_c)$ are the lexical similarity between n_m and n_r, and between n_m and n_c. $len(n_m)$, $len(n_r)$, and $len(n_c)$ represent the number of tokens in n_m, n_r, and n_c, respectively.

For the return statements, we extract n_r as Liu et al. [3] did as follows:

- if the return statement is a field access expression, we extract the field name.
- if the return statement is a method invocation expression, we extract the invoked method name.
- if the return statement is a variable, we extract the name of the variable.
- if the return statement is an array element expression *arr[i]*, we extract the name of the array *arr*.
- if the return statement is a literal expression, we extract the value as a string.
- if the return statement is the *this* expression, we extract the name of the class in which the method is declared.
- if the return statement is a member expression *a.b*, we extract the *b*.
- if the return statement is any other expression, we ignore it.

The most similar candidate return statement is the candidate return statement that is lexically most similar to the method name, whereas a candidate return statement is an expression (e.g., local variable, global variable, method invocation, or

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on August 12,2024 at 01:08:54 UTC from IEEE Xplore. Restrictions apply.

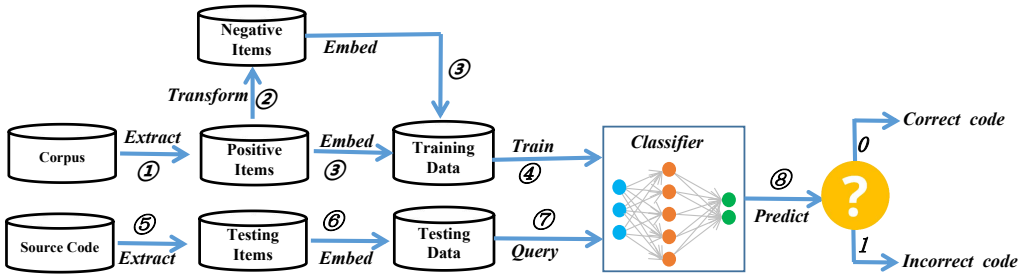

Figure 1. Overview of the Proposed Approach

default value) that could be accessed in the method declaration and replace the return statement without introducing any compiling errors. Inspired by Liu et al. [3], we consider the following expressions as candidate return statements:

- Formal parameters of the enclosing method where the call site appears.
- Local variables of the enclosing method where the call site appears.
- Global variables and method declarations available in the class where the method is declared.
- Default values of corresponding data types, e.g., "true" or "false" for boolean type.
- For a variable (or formal parameter) v, we consider not only v itself but also its members, e.g., $v.id$.

We exploit *Jaccard* similarity [26] to compute the lexical similarity between n_1 and n_2. First, we decompose n_1 and n_2 into a list of tokens based on underscores and capital letters, assuming that the name follows the popular camel case naming convention. Second, we compute the *Jaccard* similarity between n_1 and n_2 as follows:

$$lexSim(n_1, n_2) = \frac{|len(n_1) \bigcap len(n_2)|}{|len(n_1) \bigcup len(n_2)|} \tag{2}$$

where $len(n_1)$ and $len(n_2)$ represent the number of tokens in n_1 and n_2, respectively. For example, $lexSim(\text{“}httpProxy\text{”}, \text{“}getHttpProxy\text{”}) = \frac{2}{3} = 0.67$.

Take the method *getHttpsProxy* in Fig. 2 as example. We extract the following information:

$$\begin{aligned}
&n_m = \text{“}getHttpsProxy\text{”}, n_r = \text{“}httpProxy\text{”},\\
&n_c = \text{“}httpsProxy\text{”}, t_r = 1, t_c = 1,\\
&lexSim(\text{“}getHttpsProxy\text{”}, \text{“}httpProxy\text{”}) = 0.3333,\\
&lexSim(\text{“}getHttpsProxy\text{”}, \text{“}httpsProxy\text{”}) = 0.6667,\\
&len(n_m) = 3, len(n_r) = 2, len(n_c) = 2
\end{aligned}$$

Consequently, we get the following final input:

$$input =< \text{“}getHttpsProxy\text{”}, \text{“}httpProxy\text{”}, \text{“}httpsProxy\text{”}, 1, 1, 0.3333, 0.6667, 3, 2, 2 >$$

```
1 public class Proxy {
2   //..
3   private String ftpProxy;
4   private String httpProxy;
5   private String httpsProxy;
6   private String noProxy;
7   //...
8   public Proxy setNoProxy(String noProxy) {
9     this.noProxy = noProxy;
10    return this;
11  }
12  public String getNoProxy() {
13    return noProxy;
14  }
15  public String getFtpProxy() {
16    return ftpProxy;
17  }
18  public String getHttpsProxy() {
19    return httpProxy;
20  }
21  //..
22 }
```

Figure 2. Example Code

C. Vector Representation of Identifiers

To feed the context into the neural network, we convert identifiers into numerical vectors. We exploit *Word2Vec* [24] to embed identifier sequences. *Word2Vec* is a well-known neural network based encoder, which has been proved efficient in learning semantic word relationships by placing semantically similar words in adjacent high-dimensional space. It predicts a word based on the sequence of words (called *window*) before and after the word. In our experiments, we empirically set the *window* to *20*.

For a given identifier, we first decompose it into a sequence of tokens according to underscores and capital letters, assuming that the identifiers follow the popular camel case naming convention. As a result, an identifier *name(m)* is split into the following sequence of tokens:

$$\begin{aligned}
name(m) &=< t_1, t_2, \cdots, t_k >\\
&=< V(t_1), V(t_2), \cdots, V(t_k) >
\end{aligned} \tag{3}$$

where $< t_1, t_2, \cdots, t_k >$ is a sequence of tokens in the identifier, and $V(t_i)$ is the vector representation of token i. In our experiments, we empirically set the length of the vector to 128. For efficiency, we train the vector model by exploiting the most frequently used 10,000 tokens from the training dataset as

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on August 12,2024 at 01:08:54 UTC from IEEE Xplore. Restrictions apply.

the vocabulary. Our analysis results (Section III) suggest that the vocabulary covers 92% of tokens involved in the dataset. Tokens beyond the vocabulary are replaced with placeholder "unknown".

D. Generation of Training Data

Deep learning-based approaches require sufficient training data to adjust the parameters of the neural work and as a result effectively distinguish incorrect return statements from correct ones. We generate training data as follows. First, we extract names from the method signatures and features from a large amount of open-source code as depicted in Section II-B. Second, we take the code examples extracted from the original code as positive items (correct code) directly, based on the hypothesis that existing code are mostly correct [27] [28]. Third, since it is challenging to collect incorrect code from real-world corpus, we generate negative items by simply swapping the order of return statements and candidate return values in positive items, which is inspired by Michael et al. [4] who automatically generate incorrect code to train a deep learning neural network and conduct mutation test [29] by simply changing the order of arguments of correct code. Consequently, for a given method declaration extracted from the subject applications, we create a positive example

$$\begin{aligned} input_p = <& n_m, n_r, n_c, t_r, t_c, \\ & lexSim(n_m, n_r), lexSim(n_m, n_c), \\ & len(n_m), len(n_r), len(n_c) > \end{aligned}$$

and a negative example

$$\begin{aligned} input_n = <& n_m, n_c, n_r, t_c, t_r, \\ & lexSim(n_m, n_c), lexSim(n_m, n_r), \\ & len(n_m), len(n_c), len(n_r) > \end{aligned}$$

Take the code in Fig. 2 as example, we generate the following positive and negative training items:

$$\begin{aligned} input_p = <& \text{“}getHttpsProxy\text{”}, \text{“}httpProxy\text{”}, \text{“}httpsProxy\text{”}, \\ & 1, 1, 0.3333, 0.6667, 3, 2, 2 > \\ input_n = <& \text{“}getHttpsProxy\text{”}, \text{“}httpsProxy\text{”}, \text{“}httpProxy\text{”}, \\ & 1, 1, 0.6667, 0.3333, 3, 2, 2 > \end{aligned}$$

E. Deep Learning-based Classifier

The structure of deep learning-based classifier for identifying suspicious return statements is presented in Fig. 3. The input layer is composed of two parts, i.e., *names* and *features*. *Names* are first split into tokens, concatenated into token sequences, encoded into *Word2Vec* vectors as depicted in Section II-C, and then fed into a two-layer one dimensional *CNNs*. *Features*, represented as numerical values, are fed into another two-layer one dimensional *CNNs* directly.

The parameters for the *CNN* layers in our experiments are set as follows: *filters=128, kernel size=1*, and *activiation=tanh*. The *kernel size=1* is used to convolve metrics as well as tokens split from identifiers one by one sequentially. It has been proved to be effective in detecting code smell based on identifiers and metrics information extracted from source code [30].

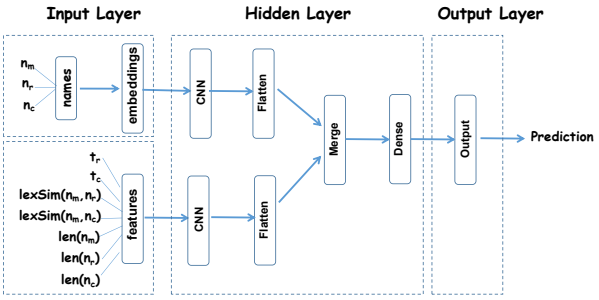

Figure 3. The architecture of the *MLDetector* Neural Network

The output of the two *CNNs* is converted into one-dimensional vectors by the *flatten* layers and then merged by the *merge* layer. Finally, the output of *merge layer* is mapped into the prediction of probability of being correct or incorrect, with a dense layer (*activation=tanh* and *neuro=128*) and a output layer (*activation=sigmoid* and *neuro=2*). Since the neural network acts as a binary classifier, we choose *loss=binary_crossentropy* in the model.

We exploit *CNNs* as the hidden layers because of the following reasons. First, *CNN* is well known as one of the most useful deep learning models [19]. Second, *CNN* has been proved to be effective in capturing the semantics of source code [10] [21] [22]. Third, we have replace CNN with other types of neural network (e.g., FNN, LSTM) as an initial try, and it fails to improve the performance of the approach.

F. Prediction

Once the deep learning-based classifier is trained with a large amount of training data, we can query it to predict whether an unseen testing code is correct (output is 0) or not (output is 1). We generate the testing dataset in the same way as generating the training dataset (i.e., extracting items from the corpus, embedding identifiers into *Word2Vec* representation).

It should be noted that the training of the neural network could be done on special machines in advance. The training of deep neural networks, even with advanced deep learning algorithms, is usually time-consuming and requires special devices, e.g., powerful GPU. However, in the prediction phase, it often takes neural networks only a few milliseconds to generate output for a given input. Consequently, with the resulting neural network trained on special devices in advance, the proposed approach can response instantly, which makes it practical to integrate the proposed approach into IDEs.

III. Evaluation

In this section, we evaluate the proposed approach with dataset extracted from 500 Java open-source applications and artificially generated incorrect return statements.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on August 12,2024 at 01:08:54 UTC from IEEE Xplore. Restrictions apply.

A. Research Questions

The evaluation investigates the following research questions:

- **RQ1**: How effective is the proposed approach to distinguish incorrect return statements from correct ones?
- **RQ2**: How do different deep learning techniques influence the performance of the proposed approach?
- **RQ3**: How do different vectorization approaches influence the performance of the proposed approach? Can we improve the performance of the proposed approach by replacing the current vectorization approach with alternative vectorization approaches?
- **RQ4**: How long does it take to train the neural network and to identify incorrect return statements? Will IDEs become unresponsive if the approach is implemented as a plug-in of such IDEs?

Research question **RQ1** concerns the performance (e.g., precision and recall) of the proposed approach in identifying incorrect return statements. We compare the proposed approach against the *similarity-based approach* [31]. The *similarity-based* approach is selected as the baseline because of the following reasons. First, it is one of the state-of-the-art name-based approaches to investigate information conveyed in identifiers. Second, the findings of it have been applied to detect anomalies and recommend arguments. Third, although some related approaches have been proposed to detect bugs, their implementation is not publicly available. Note that, in this paper, we implement the *similarity-based* approach by recommending the candidate return statement that is lexically most similar to the method name. To answer the research question, we evaluate 500 open-source Java applications and artificially generate incorrect code.

Research question **RQ2** investigates whether replacing the *CNN* model in the proposed approach with other deep learning models could result in further improvement in the performance of the proposed approach. To this end, we replace the *CNN* model with Fully-connected Neural Network (*FNN*) and Long Short Neural Network (*LSTM*), and repeat the evaluation. Evaluation results are presented in Section III-F2.

Research question **RQ3** concerns the influence of vector representations. In Section II-C, we convert identifiers into vectors by the well known *Word2Vec*. To answer research question **RQ3**, we replace the *Word2Vec* representation with one-hot representation [32], and repeat the evaluation.

Research question **RQ4** concerns the efficiency of the proposed approach, i.e., time complexity. It usually takes a long time to train deep learning models on specific machines, whereas it responses instantly to conduct testing and prediction based on learned models. Answering this question helps to validate whether the proposed approach can be applied to identify incorrect return statements in practice.

B. Subject Applications

We first search for open-source Java applications from GitHub [25] that have at least 5 releases and can be imported into Eclipse. Second, we sort the resulting applications according to their stars in descending order, download the top 600 applications, and randomly select 500 out of 600 as our subject applications[1]. The size of our subject applications varies from 935 LOC to 717,732 LOC (excluding blank lines and comment lines), and 11,910,156 LOC in total. We extract 84,082 Java files, and 1,967,471 methods from the subject applications. We select such applications because of the following reasons. First, the source code of such applications is publicly available, which helps other researchers to repeat the evaluation. Second, such applications are popular and released several versions, and thus it is likely that most of the incorrect return statements, if there is any, have been identified and fixed. It is important because the evaluation is based on the assumption that the return statements in the source code are correct.

It should be noted that although the proposed approach can be applied to different programming languages, as an initial try, we only confine the prototype implementation *MLDetector* to Java language.

C. Process

We carry out ten-fold cross-validation on subject applications in Section III-B to validate the proposed approach. The evaluation process is conducted as follows:

- We generate positive items (correct code) by extracting information from each subject application.
- We generate negative items (incorrect code) by transforming each positive item, i.e., exchanging the return statement with candidate return value and updating corresponding features.
- We generate our dataset by combining positive items and negative items together and shuffle them randomly.
- We split the dataset into ten equally-sized folds noted as f_i ($i = 1 \cdots 10$). For each fold cross-validation (e.g., f_i), we employ all data items from other folds (except f_i) as the corpus of training data, and those items within the current fold (f_i) as the testing data.
- We convert each identifier (i.e., method name, return value, and candidate return value) into vectors. The conversion is done by *Word2Vec* as introduced in Section II-C. Note that, we train the *Word2Vec* representation of identifiers using only the training data.
- We train the neural network as introduced in Section II-E with the training data represented as vectors.
- We feed each item in testing data to the learned resulting network, and compare the labeled output of each item with the prediction of the network.
- Finally, we compute the precision and recall of the recommendation.

D. Measurements

To measure the performance of the proposed approach, we use precision, recall, and the F1-score, defined as follows:

$$precision = \frac{T_{pos}}{T_{pos} + F_{pos}} \tag{4}$$

[1]April 18, 2019

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on August 12,2024 at 01:08:54 UTC from IEEE Xplore. Restrictions apply.

$$recall = \frac{T_{pos}}{T_{pos} + F_{neg}} \quad (5)$$

$$F1 = \frac{2 * precision * recall}{precision + recall} \quad (6)$$

where the T_{pos}, F_{pos}, F_{neg} represents the number of true positive, false positive, and false negative, respectively. Informally, precision measures how much percentage the proposed approach makes correct classification, and recall represents how much percentage of incorrect return statements could be identified by the proposed approach. To evaluate the precision and recall of the proposed approach, we train the classifier with a positive dataset extracted 500 Java open-source applications and a negative dataset generated artificially by transforming each positive item. Consequently, the ratio of positive items and negative items in the training dataset is 50% to 50%.

Besides those metrics, we also compute Area Under Curve (AUC) for performance comparison using the well-known machine learning tool scikit-learn[2].

E. Implementation

We parse the ASTs of Java files and extract positive data using the Eclipse plug-in tool *JDT*. The neural network for suspicious code identification is implemented based on an open-source generic implementation of neural networks, called *Keras* [33]. It is a high-level library that may run on top of either TensorFlow [34] or Theano [35] [36]. We use *Keras* because it is fast and simple with rich documents. It is also one of the most popular open-source neural network libraries. We implement the proposed approach as a prototype, called *MLDetector* (Machine Learning-based Detector).

We evaluate the proposed approach on 1,967,471 methods extracted from 500 open-source Java applications. We focus on Java applications in the implementation because Java is the most popular programming language according to the well-known TIOBE Index[3]. To this end, as an initial try we implement the proposed approach to handle Java applications only. In future, it would be interesting to extend *MLDetector* to handle applications in other languages.

F. Results

1) ***RQ1:*** *Identification of Incorrect Return Statements:* Results of the ten-fold cross-validations on the proposed approach and the similarity-based baseline approach are presented in Table I. In the table, the first column presents the cross-validation fold id, the 2-9 columns present precision, recall, F1 measure, and AUC of the proposed approach and the similarity-based approach, respectively. The last row of the table presents the average performance.

From this table, we make the following observation:

- The proposed approach outperforms the state-of-the-art similarity-based approach in all metrics (precision, recall, F1, and AUC).
- The performance of the proposed approach varies significantly when the training and testing dataset change. For example, during the ten-fold cross-validation, the precision of the proposed approach ranges from 83.87% to 90.05% (standard deviation: 0.025).
- The proposed approach identifies most of incorrect return statement correctly. The average precision and F1 is 86.25% and 70.56%, respectively.

We conclude from the results that the proposed approach outperforms the similarity-based baseline approach. Compared to the manually drawn rules (exploited by the lexical similarity-based approach), the deep learning-based approach can learn better rules to identify incorrect code.

2) ***RQ2:*** *Influence of Learning Models:* We empirically compare the *CNN-based* approach with Full-connected Neural Network (FNN) and Long Short Neural Network (LSTM) based approaches. The *FNN-based* approach employs two dense layers with the following parameters: *units=128*, *dropout=0.2*, and *activation=softmax*. The *LSTM-based* approach employs two LSTM layers with the following parameters: *hidden_size=128*, *dropout=0.2*, and *activation=softmax*. Results of ten-fold cross-validations on different neural networks are presented in Table II.

From this table, we make the following observations:

- First, *CNN-based* approach significantly outperforms *FNN-based* approach. It improves precision significantly from 47.98% to 86.25%.
- Second, *CNN-based* approach slightly outperforms *LSTM-based* approach as well. It improves precision from 82.02% to 86.25%.
- Third, despite the variation of performance on different training and testing datasets, the proposed approach outperforms alternative neural network models in all of the ten folds.

The results suggest that just applying a deep learning solution directly whereas without sophisticated adaption to the problem may result in poor performance (e.g., the *FNN-based* approach).

We conclude from the results that different deep learning techniques do influence the performance of the proposed approach, and the *CNN-based* approach outperforms *FNN-based* and *LSTM-based* approaches on each fold.

3) ***RQ3:*** *Influence of Vectorization Approaches:* To investigate the influence of vectorization approaches, we replace the *Word2Vec* employed in Section III-F1 with one-hot vectorization and repeat the evaluation. Evaluation results are presented in Table II. We also conduct Analysis of Variance (ANOVA) on the resulting precision, and present the results in Figs. 4. From Table I and Figs. 4, we make the following observations:

- First, *Word2Vec* vectorization results in better performance than one-hot. Replacing *Word2Vec* with one-hot reduces the average precision by 9.5% (=(86.25%-78.05%)/86.25%). One of the reason for the reduction in performance is that *Word2Vec* can learn the order of the words and the semantics of the words, whereas one-hot is a bag-of-words approach.

[2]https://scikit-learn.org/dev/index.html

[3]October 2019, https://www.tiobe.com/tiobe-index/

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on August 12,2024 at 01:08:54 UTC from IEEE Xplore. Restrictions apply.

Table I
PERFORMANCE OF THE PROPOSED APPROACH

# Ten-fold	Proposed Approach				Similarity-based Approach			
	Precision	Recall	F1	AUC	Precision	Recall	F1	AUC
1#	86.24%	58.12%	69.44%	89.13%	5.02%	20.75%	8.08%	30.41%
2#	88.77%	59.83%	71.49%	90.29%	6.43%	22.84%	10.03%	36.94%
3#	83.36%	65.74%	73.51%	83.34%	4.52%	19.17%	7.32%	40.28%
4#	90.05%	62.80%	74.00%	87.32%	6.26%	23.46%	9.88%	35.20%
5#	86.75%	60.24%	71.10%	85.33%	5.42%	17.59%	8.29%	42.97%
6#	87.25%	56.57%	68.64%	89.15%	5.11%	18.60%	8.02%	36.45%
7#	83.87%	62.32%	71.51%	88.66%	4.82%	15.32%	7.33%	47.38%
8#	85.78%	55.62%	67.49%	82.43%	6.90%	18.35%	10.03%	42.66%
9#	84.95%	54.66%	66.52%	80.26%	6.78%	19.23%	10.03%	45.83%
10#	85.43%	61.16%	71.29%	81.59%	6.45%	13.25%	8.68%	35.27%
Avg	**86.25%**	**59.71%**	**70.56%**	**85.75%**	**5.77%**	**18.86%**	**8.77%**	**39.33%**

Table II
INFLUENCE OF DIFFERENT LEARNING MODELS

Ten-fold	FNN-based Approach			LSTM-based Approach			One-hot Approach		
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
1#	44.64%	31.77%	37.12%	82.52%	50.23%	62.45%	79.43%	43.17%	55.94%
2#	46.03%	30.67%	36.81%	84.02%	52.43%	64.57%	82.24%	45.20%	58.34%
3#	42.52%	38.23%	40.26%	79.43%	46.27%	58.48%	75.45%	40.93%	53.07%
4#	56.26%	32.73%	41.38%	88.60%	41.52%	56.54%	80.57%	40.15%	53.59%
5#	45.32%	33.25%	38.36%	81.01%	44.38%	57.34%	74.69%	42.34%	54.04%
6#	49.11%	34.91%	40.81%	80.03%	54.22%	64.64%	76.34%	47.18%	58.32%
7#	54.82%	36.65%	43.93%	80.54%	49.14%	61.04%	73.47%	44.26%	55.24%
8#	46.90%	31.48%	37.67%	81.71%	47.52%	60.09%	79.52%	45.63%	57.99%
9#	45.78%	32.72%	38.16%	82.08%	43.26%	56.66%	78.62%	41.32%	54.17%
10#	48.45%	37.41%	42.22%	80.24%	45.82%	58.33%	80.17%	43.63%	56.51%
Avg	**47.98%**	**33.98%**	**39.79%**	**82.02%**	**47.48%**	**60.14%**	**78.05%**	**43.38%**	**55.77%**

- Second, the analysis on variance on precision (F $>F_{crit}$ and P-value $<$0.05 where α = 0.05) suggests that different vectorization approaches lead to significantly different precision. As presented in Fig. 4, the SS (sum of squares of deviation from mean) within groups (0.011338) is much smaller than that between groups (0.033579).

We conclude from the results that different vectorization approaches influence the performance of the proposed approach significantly, and that *Word2Vec* vectors learn the semantic relationship between identifiers better than one-hot representation.

4) ***RQ4:*** *Efficiency of the Proposed Approach:* To evaluate the efficiency of the proposed approach, we investigate the time require for training and testing, respectively. It usually takes long time to train a neural network, consequently we conduct model training on a specific workstation with the following GPU installed: 2.6GHz Intel Xeon E5-2680 processor, 64G RAM, NVIDIA Tesla P4 GPU, Ubuntu 16.04.2 LTS, and Keras. However, the identification of incorrect return statements is often conducted by developers when they write program or maintain projects. Consequently, to simulate the identification of suspicious code, we conduct testing of the proposed approach on a common personal computer with Intel Core i7-6700 cpu.

Table III shows the efficiency of the proposed approach. Evaluation results suggest that it takes approximately 10 hours on average to train and test the neural network model on the 500 open-source applications. However, it will not make the IDEs where the approach is integrated unresponsive because the training could be done in advance outside the IDEs. What does matter is the time the approach takes to make a prediction of whether the code is correct or not. It takes on average 0.62 minutes to make a prediction for each application, and around 15 milliseconds on average to make prediction for a given method. We conclude that the proposed approach is efficient.

Table III
MINUTES SPENT ON TRAINING AND TESTING THE PROPOSED APPROACH

MLDetector	Training	Prediction
Data Extraction	347	35
Learning	554	62

IV. CASE STUDY

In Section III, we evaluate the proposed approach on artificially generated incorrect return statements. In this section, we evaluate the proposed approach on real-world applications without artificially injected code, and compare the reported warnings against source code change history by inspecting commit logs manually.

A. Subject Applications

From the 600 most popular Java open-source applications downloaded from Github in Section III-B, we employ the remaining 100 applications (i.e., excluding those involved in the evaluation in Section III-B) as the subject applications of our case study. The resulting subject applications are listed online [37].

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on August 12,2024 at 01:08:54 UTC from IEEE Xplore. Restrictions apply.

ANOVA analysis on precision

SUMMARY

Groups	Count	Sum	Average	Variance
Word2Vec Approach	10	8. 6245	0. 86245	0. 000429334
One-hot Approach	10	7. 805	0. 7805	0. 000830458

ANOVA

Source of Variation	SS	df	MS	F	P-value	F crit
Between Groups	0. 033579	1	0. 03357901	53. 30883413	0. 00000088	4. 41387342
Within Groups	0. 011338	18	0. 0006299			
Total	0. 044917138	19				

Figure 4. Anova Analysis on Precision

B. Process

We conduct the following case study to evaluate the effectiveness of the identification of incorrect return statements in real-world applications. First, we train the classifier based on the subject applications in Section III-B. Second, we query the trained classifier to predict whether a given method from the subject applications of section IV-A is correct or not, and record reported warnings. Third, we parse the evolution history of subject applications for extracting problems related to incorrect return statements and manually check the real incorrect return statements. We use *git log* command to parse commit logs that change the return statements, and manually collect those commits that replace an incorrect return statement with a correct one. Such a process ensures that each considered change is indeed a fix to the incorrect return statement. Finally, we manually compare incorrect code identified by the classifier against those identified by manually commits inspection, and compute the precision and recall of the proposed approach in identifying real-world bugs.

C. Results

The results of the case study are listed in the online appendix [37]. In total, we found that 65 incorrect return statements are fixed in the software evolution history by parsing the commits. The proposed approach reports 71 suspicious return statements, out of which 42 incorrect return statements point to real incorrect return statements that have been changed by commits. Consequently, the precision and recall of the proposed approach is 59% and 65%, respectively.

We also compare the proposed approach against the *similarity-based approach* [31]. Results suggest that the *similarity-based* approach finds 9 out of 65 bugs in the subject applications, with 213 warning reports. All the 9 incorrect return statements are identified by the proposed approach. Consequently, we conclude that compared to the manually drawn rule (lexical similarity-based approach), the deep learning-based approach learns better rules of distinguishing incorrect code from correct ones, and the proposed approach outperforms the *similarity-based* approach significantly.

D. Example of Incorrect Return Statements

Fig. 5 lists two incorrect return statements identified by the proposed approach. The first example is from the application *SeleniumHQ* (commit id *d919114*). The method *getHttpsProxy* should return the field *httpsProxy*, whereas developers mistakenly return the field *httpProxy*.

The second example is from the application *jenkins* (commit id *69a246*). The method *getTransitiveDownstreamProjects* should invoke the method *getTransitiveDownstream* from another class as the return value, whereas it mistakenly invoked the method *getTransitiveUpstream*.

V. Threats to Validity

A threat to external validity is that the conclusions are drawn only from a set of sample applications, which might not hold on other applications. To reduce such threat, we collect 500 popular open-source applications in various domains from the well-known open-source community *GitHub*, which finally yields a 3,934,942 training dataset. We further reduce this threat via ten-fold cross validations as described in Section III-C. We find that the conclusions keep stable across validations while training and testing data change.

Another threat to external validity is that conclusions are drawn on Java applications, which may not hold for applications written in other programming languages. In the future, it would be interesting to investigate whether the proposed approach works for other languages, e.g., C, Python.

A threat to construct validity is that the vector representation and neural network structure may not be optimized, consequently the performance may be inaccurate. To reduce such threat, we carefully design the neural network architecture and select *CNNs* as the hidden layer because *CNN* has been proved effective in learning semantical relationship in identifiers [21]. Furthermore, we extract a huge amount of datasets to optimize the parameter of the neural network.

The second threat to construct validity is that existing return statements in the involved application are supposed to be correct in the process of *CNNs* model training. We take such an assumption because of the following reasons. First, it is challenging, if not impossible, to distinguish incorrect code from correct ones manually. Second, subject applications involved in the evaluation are popular and well-known, and thus it is likely that most of the return statements are correct. However, it is also likely that some of them are incorrect. Consequently, we may make mistakes by simply supposing current return statements are correct. To reduce such threat, we select well-known and popular applications where incorrect code (if any) are more likely to be fixed.

The third threat to the construct validity is that the negative training dataset is generated artificially, which may not represent the real-world incorrect code. To reduce such threat, we conduct a case study to check whether the proposed

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on August 12,2024 at 01:08:54 UTC from IEEE Xplore. Restrictions apply.

1) Example from SeleniumHQ (commit d919114b9bed12ade27b873093e14a30ab4c60e5)

```
2 java/client/src/org/openqa/selenium/Proxy.java
@@ -198,7 +198,7 @@ public Proxy setHttpProxy(String httpProxy) {
198 198      * @return the HTTPS proxy hostname if present, or null if not set
199 199      */
200 200     public String getHttpsProxy() {
201     -     return httpProxy;
    201 +     return httpsProxy;
202 202     }
```

2) Example from jenkins (commit 69a2461d7d2e6fc9dd5702315afe6bdec1394982)

```
2 core/src/main/java/hudson/model/AbstractProject.java
@@ -692,7 +692,7 @@ void removeFromList(Descriptor<T> item, List<T> collection) throws IOException {
692 692      * @since 1.138
693 693      */
694 694     public final Set<AbstractProject> getTransitiveDownstreamProjects() {
695     -         return Hudson.getInstance().getDependencyGraph().getTransitiveUpstream(this);
    695 +         return Hudson.getInstance().getDependencyGraph().getTransitiveDownstream(this);
696 696     }
```

Figure 5. Example of incorrect return statements identified by the proposed approach

approach can identify real-world incorrect return statements by comparing commit messages and warnings of the proposed approach, and results suggest that the proposed approach identifies 42 real-world incorrect code.

A threat to internal validity is that the features selection may strongly depend on the quality of codebases, such as the naming convention of a function or variable, etc. We tokenize identifiers based on the assumption that they follow the camel case or underscore naming convention. As future work, we will validate the proposed approach on a more large dataset, extend the dataset to codebases written in other languages, and exploit more advanced tokenization approaches to split identifiers.

Finally, the case study in Section IV could be inaccurate. We compute the recall of the case study based on the assumption that we get all incorrect return statements by manually investigating the commits that only change the return statements. However, it may miss some incorrect return statements fixed in complex commits. In the future, it would be interesting to reduce this validity by investigating the entire commit logs of subject applications.

VI. Related Work

A. Bug Detection

Many approaches have been proposed to detect bugs in source code. One branch of such approaches relies on formal heuristic rules and static program analysis. *FindBugs* [15] and its successor *SpotBugs*[4] find bug patterns in Java applications and AIP libraries based on complicated program analysis (e.g., dataflow). Infer [38] detects software bugs statically based on formal verification techniques and is applied in Facebook[5]. Error Prone [39] automatically detects and repairs errors at compile time based on compiler-level analysis. These statically heuristics-based tools detect bugs by mining rules from source code and identify bugs against such ad-hoc rules. They cannot distinguish uncommon but valid code from bugs.

The second branch of bug detection approaches relies on dynamic techniques (e.g., testing, assertions) to detect the runtime bugs. Almasi et al. [40] find real bugs based on unit test generation. Shamshiri et al. [41] apply three test generation tools to detect real bugs in the *Defect4j* dataset [42]. Selakovic et al. [43] investigate performance issues in Javascript programs.

The third branch of bug detection approaches exploits the rich information conveyed in identifiers to detect bugs. Liu et al. [44] [31] [12] detect incorrect arguments in method invocations based on lexical similarity between arguments and parameters. Michael et al. [9] propose a graph matching based approach to reduce false positive their previous approach [12] in detecting argument defects. Li et al. [45] propose *PR-Miner* approach to extract implicit programming rules from a large corpus of code, which can be used to detect method invocation problems.

The fourth branch of bug detection approaches exploits deep learning techniques to automatically learn and predict bugs. Murali et al. [46] detect incorrect API usage by training a recurrent neural network based on positive method invocation examples. Choi et al. [47] detect buffer overruns by training memory networks [48] with raw source code. Wang et al. [49] predict code defect by training a deep belief network to learn semantic features from the *ASTs* of source code. Michael et al. [4] propose deep learning and name-based approaches to detect accidentally swapped arguments defects.

The proposed approach differs from previous approaches in that it leverages deep learning techniques and semantic information conveyed in source code identifiers to distinguish

[4]https://github.com/spotbugs/spotbugs
[5]https://www.facebook.com/

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on August 12,2024 at 01:08:54 UTC from IEEE Xplore. Restrictions apply.

incorrect return statements from correct ones. To the best of our knowledge, it is the first work to detect incorrect returns statements with deep learning techniques.

B. Machine Learning on Static Code Analysis

The *n-gram* model has been proved to be effective in predicting repeated features of source code [50]. Hindle et al. [51] train a *n-gram* model to predict the next token in source code based on the preceding *n* tokens. Allamanis et al. [52] [5] exploit *n-gram* models to recommend variable names, method names, and class names. Raychev et al. [28] exploit *n-gram* models and conditional random fields [53] to predict variable names in JavaScript. Hellendoorn et al. [54] propose a nested and cached n-gram based approach called *SLP-Core* to modeling and completing source code. By adding a cache mechanism to n-gram models, the approach assigns a higher probability to tokens most recently used based on the findings that identifiers in source code have a high degree of localness.

Neural networks and deep learning are also applied to facilitate software engineering tasks, e.g., code completion, code smell detection, etc. White et al. [55] introduce deep learning to model software and illustrate how it outperforms n-gram models. Murali et al. [46] generate API-dependent Java code by training a neural generator on program sketches. It can predict the entire body of a method given just a few API calls or data types that appear in the method. Wang et al. [56] propose a novel semantic program embedding that is learned from program execution traces. They exploit recurrent neural networks to model natural fit of program semantic and use the model for program repair.

Liu et al. [21] [22] propose deep learning based approaches to detect feature envy and other code smells. Allamanis et al. [57] [58] generate code summarization by exploiting an attentional neural network to capture long-distance attention features in source code. Liu et al. [10] detect inconsistent method names based on *Word2Vec* and *CNNs* deep learning model.

Mou et al. [59] propose a tree-based neural network to capture structure of source code, which can be used in program classification. White et al. [60] [60] detect code clones by combining source code identifiers and structures (i.e., ASTs) with recursive neural network and convolutional neural network. Xhang et al. [61] learn lexical and structural information from source by training a neural network based on a sequence of sub-trees of ASTs at the statement level, which could be used to code classification and clone detection.

The proposed approach differs from such approaches in that it effectively train the deep learning neural network by artificially generating a large amount of negative data (incorrect code).

VII. Conclusion and Future Work

In this paper, we propose the first deep learning-based approach to identify incorrect return statements. Our framework *MLDetector* distinguishes incorrect return statements from correct ones by leveraging semantics conveyed by the natural language phrases that are used as identifiers in the source code. The rationale is that both method signature and return value should explicitly specify the output of the method, and thus a significant mismatch between method signature and return value may suggest a suspicious return statement. We conduct two separate experiments to evaluate the proposed approach. In the first experiment, we train the neural network classifier with both positive data extracted from 500 open-source applications and artificially generated negative data. To feed code into a neural network, we convert them into vectors by *Word2Vec*. In the second experiment, we query the trained classifier to predict incorrect return statements in real-world code, and manually inspect the prediction by parsing the commits. Evaluation results suggest that the accuracy of the proposed approach ranges from 83% to 90% in the first experiment, and identify 42 out of 65 real-world incorrect return statements with the precision of 59% in the second experiment.

A potential way to improve the proposed approach in the future is to replace the *Word2Vec* with other advanced vector representations (e.g., *Paragraph Vector* [62], BERT[6] [63], Glove [64]), and compute the similarity between identifiers based on other similarity metrics (e.g., Euclidean distance [65]). Although we have tried some alternative metrics, e.g., cosine similarity, and found that it has little influence on the performance, it is likely that replacing the metrics with more advanced metrics may improve the performance of the proposed approach.

It would be interesting in the future to extend the proposed approach to applications in other programming languages (e.g., C, Python), which may help to reveal whether the proposed approach can be generalized.

Acknowledgment

The authors would like to say thanks to the anonymous reviewers for their valuable suggestions.

The work is partially supported by the National Natural Science Foundation of China (61772071, 61690205).

References

[1] L. M. Eshkevari, V. Arnaoudova, M. Di Penta, R. Oliveto, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of identifier renamings," in *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM, 2011, pp. 33–42.

[2] D. Shepherd, L. Pollock, and K. Vijay-Shanker, "Case study: supplementing program analysis with natural language analysis to improve a reverse engineering task," in *ACM Sigplan-Sigsoft Workshop on Program Analysis for Software TOOLS and Engineering*, 2007, pp. 49–54.

[3] H. Liu, Q. Liu, C.-A. Staicu, M. Pradel, and Y. Luo, "Nomen est omen: exploring and exploiting similarities between argument and parameter names," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 1063–1073. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884841

[4] M. Pradel and K. Sen, "Deepbugs: A learning approach to name-based bug detection," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 147:1–147:25, Oct. 2018. [Online]. Available: http://doi.acm.org/10.1145/3276517

[6] https://github.com/google-research/bert

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on August 12,2024 at 01:08:54 UTC from IEEE Xplore. Restrictions apply.

[5] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 38–49. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786849

[6] M. Allamanis and C. Sutton, "Mining idioms from source code," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 472–483. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635901

[7] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, "Inferring method specifications from natural language API descriptions," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 815–825. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337319

[8] H. Zhong, L. Zhang, T. Xie, and H. Mei, "Inferring resource specifications from natural language API documentation," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 307–318. [Online]. Available: https://doi.org/10.1109/ASE.2009.94

[9] A. Rice, E. Aftandilian, C. Jaspan, E. Johnston, M. Pradel, and Y. Arroyo-Paredes, "Detecting argument selection defects," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 104:1–104:22, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3133928

[10] K. Liu, D. Kim, T. F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. L. Traon, "Learning to spot and refactor inconsistent method names," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 1–12. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00019

[11] E. W. Høst and B. M. Østvold, *ECOOP 2009 – Object-Oriented Programming: 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, ch. Debugging Method Names, pp. 294–317.

[12] M. Pradel and T. R. Gross, "Detecting anomalies in the order of equally-typed method arguments," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 232–242. [Online]. Available: http://doi.acm.org/10.1145/2001420.2001448

[13] M. Pradel and T. Gross, "Name-based analysis of equally typed method arguments," *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1127–1143, Aug. 2013. [Online]. Available: http://dx.doi.org/10.1109/TSE.2013.7

[14] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan, "Building useful program analysis tools using an extensible java compiler," in *IEEE International Working Conference on Source Code Analysis & Manipulation*, 2012.

[15] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, Dec. 2004. [Online]. Available: http://doi.acm.org/10.1145/1052883.1052895

[16] G. Ammons, R. Bodik, and J. R. Larus, "Mining specification," in *Acm Sigplan-sigact Symposium on Principles of Programming Languages*, 2002.

[17] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross, "Statically checking API protocol conformance with mined multi-object specifications," in *International Conference on Software Engineering*, 2012.

[18] B. Liang, B. Pan, Z. Yan, W. Shi, and C. Yan, "Antminer: mining more bugs by reducing noise interference," in *IEEE/ACM International Conference on Software Engineering*, 2017.

[19] Y. Kim, "Convolutional neural networks for sentence classification," *Eprint Arxiv*, 2014.

[20] Y. Pan, M. Tao, T. Yao, H. Li, and R. Yong, "Jointly modeling embedding and translation to bridge video and language," in *Computer Vision & Pattern Recognition*, 2016.

[21] H. Liu, Z. Xu, and Y. Zou, "Deep learning based feature envy detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 385–396. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238166

[22] H. Liu, J. Jin, Z. Xu, Y. Bu, Y. Zou, and L. Zhang, "Deep learning based code smell detection," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[23] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'13. USA: Curran Associates Inc., 2013, pp. 3111–3119. [Online]. Available: http://dl.acm.org/citation.cfm?id=2999792.2999959

[24] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *Computer Science*, 2013.

[25] Github. https://github.com/.

[26] W. W. Cohen, P. D. Ravikumar, and S. E. Fienberg, "A comparison of string distance metrics for name-matching tasks," in *Proceedings of IJCAI-03 Workshop on Information Integration on the Web (IIWeb-03)*, 2003, pp. 73–78.

[27] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 858–868. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818858

[28] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from "big code"," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15. New York, NY, USA: ACM, 2015, pp. 111–124. [Online]. Available: http://doi.acm.org/10.1145/2676726.2677009

[29] J. Yue and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2010.

[30] H. Liu, J. Jin, Z. Xu, Y. Bu, Y. Zou, and L. Zhang, "Deep learning based code smell detection," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[31] G. Li, H. Liu, Q. Liu, and Y. Wu, "Lexical similarity between argument and parameter names: An empirical study," *IEEE Access*, vol. 6, pp. 58 461–58 481, 2018.

[32] one-hot. (2018) https://en.wikipedia.org/wiki/one-hot.

[33] Keras. (2019) https://keras.io/.

[34] Google. (2019) http://www.tensorflow.org/.

[35] Theano Development Team. (2019) https://github.com/theano/theano/.

[36] ——, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016. [Online]. Available: http://arxiv.org/abs/1605.02688

[37] https://github.com/d12126977/saner.

[38] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, "Moving fast with software verification," 2015.

[39] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan, "Building useful program analysis tools using an extensible java compiler," in *Proceedings of the 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 14–23. [Online]. Available: https://doi.org/10.1109/SCAM.2012.28

[40] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ser. ICSE-SEIP '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 263–272. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2017.27

[41] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults?: An empirical study of effectiveness and challenges," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 201–211. [Online]. Available: https://doi.org/10.1109/ASE.2015.86

[42] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 437–440. [Online]. Available: http://doi.acm.org/10.1145/2610384.2628055

[43] M. Selakovic and M. Pradel, "Performance issues and optimizations in javascript: An empirical study," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 61–72. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884829

[44] H. Liu, Q. Liu, C. A. Staicu, M. Pradel, and Y. Luo, "Nomen est omen: exploring and exploiting similarities between argument and parameter

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on August 12,2024 at 01:08:54 UTC from IEEE Xplore. Restrictions apply.

names," in *ACM 38th IEEE International Conference on Software Engineering*, 2016.

[45] Z. Li and Y. Zhou, "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," *Acm Sigsoft Software Engineering Notes*, vol. 30, no. 5, pp. 306–315, 2005.

[46] V. Murali, S. Chaudhuri, and C. Jermaine, "Bayesian specification learning for finding API usage errors," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 151–162. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106284

[47] M. J. Choi, S. Jeong, H. Oh, and J. Choo, "End-to-end prediction of buffer overruns from raw source code via neural memory networks," 2017.

[48] J. Weston, S. Chopra, and A. Bordes, "Memory networks," *arXiv preprint arXiv:1410.3916*, 2014.

[49] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 297–308. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884804

[50] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," 11 2014, pp. 269–280.

[51] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Commun. ACM*, vol. 59, no. 5, pp. 122–131, Apr. 2016. [Online]. Available: http://doi.acm.org/10.1145/2902362

[52] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 281–293. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635883

[53] J. Lafferty, A. McCallum, and F. C. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data," in *Proceedings of the 18th International Conference on Machine Learning*, ser. ICML 2001. New York, NY, USA: ACM, 2001, pp. 282–289. [Online]. Available: http://portal.acm.org/citation.cfm?id=655813

[54] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Joint Meeting on Foundations of Software Engineering*, 2017, pp. 763–773.

[55] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk,

[63] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language

"Toward deep learning software repositories," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 334–345. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820559

[56] K. Wang, R. Singh, and Z. Su, "Dynamic neural program embedding for program repair," *CoRR*, vol. abs/1711.07163, 2017. [Online]. Available: http://arxiv.org/abs/1711.07163

[57] M. Allamanis, H. Peng, and C. A. Sutton, "A convolutional attention network for extreme summarization of source code," *CoRR*, vol. abs/1602.03001, 2016. [Online]. Available: http://arxiv.org/abs/1602.03001

[58] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 38–49. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786849

[59] L. Mou, G. Li, Z. Jin, L. Zhang, and T. Wang, "TBCNN: A tree-based convolutional neural network for programming language processing," *CoRR*, vol. abs/1409.5718, 2014. [Online]. Available: http://arxiv.org/abs/1409.5718

[60] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sep. 2016, pp. 87–98.

[61] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, May 2019, pp. 783–794.

[62] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, 2014, pp. 1188–1196.
understanding," *CoRR*, vol. abs/1810.04805, 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[64] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *Conference on Empirical Methods in Natural Language Processing*, 2014, pp. 1532–1543.

[65] R. J. Kuoa, Z. Y. Chen, and F. C. Tien, "Integration of particle swarm optimization and genetic algorithm for dynamic clustering," *Information Sciences*, vol. 195, no. 13, pp. 124–140, 2012.

Authorized licensed use limited to: BEIJING INSTITUTE OF TECHNOLOGY. Downloaded on August 12,2024 at 01:08:54 UTC from IEEE Xplore. Restrictions apply.