HINDBR: Heterogeneous Information Network Based Duplicate Bug Report Prediction

Guanping Xiao*†⋆, Xiaoting Du‡, Yulei Sui§, Tao Yue*
*College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, China
†State Key Laboratory of Novel Software Technology, Nanjing University, China
‡School of Automation Science and Electrical Engineering, Beihang University, China
§School of Computer Science, University of Technology Sydney, Australia
{gpxiao, taoyue}@nuaa.edu.cn, xiaoting_2015@buaa.edu.cn, yulei.sui@uts.edu.au

***Abstract*—Duplicate bug reports often exist in bug tracking systems (BTSs). Almost all the existing approaches for automatically detecting duplicate bug reports are based on text similarity. A recent study found that such approaches may become ineffective in detecting duplicates in bug reports submitted after the just-in-time (JIT) retrieval, which is now a built-in feature of modern BTSs (e.g., Bugzilla). This is mainly because the embedded JIT feature suggests possible duplicates in a bug database when a bug reporter types in the new summary field, therefore minimizing the submission of textually similar reports. Although JIT filtering seems effective, a number of bug report duplicates remain undetected. Our hypothesis is that we can detect them using a semantic similarity-based approach.**

This paper presents HINDBR, a novel deep neural network (DNN) that accurately detects semantically similar duplicate bug reports using a heterogeneous information network (HIN). Instead of matching text similarity alone, HINDBR embeds semantic relations of bug reports into a low-dimensional embedding space where two duplicate bug reports represented by two vectors are close to each other in the latent space. Results show that HINDBR is effective.

***Index Terms*—heterogeneous information network, duplicate bug report prediction, deep learning.**

I. INTRODUCTION

Bug tracking systems (BTSs), e.g., Bugzilla and JIRA, inherently suffer from the duplicate problem, i.e., the same bug is reported multiple times, leading to unnecessary maintenance effort such as repeatedly discussing the same bug. Quite a few automatic approaches for detecting duplicate bug reports have been proposed during the past few years [1]–[7]. Almost all of them, including the state-of-the-art algorithms BM25F [8] and REP [9], heavily rely on the text similarity calculated with information retrieval (IR) techniques such as Term Frequency-Inverse Document Frequency (TF-IDF), to detect duplicate bug reports [10].

However, with the advent of the just-in-time (JIT) retrieval feature in modern BTSs, the above-mentioned textual-based approaches become ineffective in detecting after-JIT duplicate bug reports [10]. This is because many recent BTSs (e.g., Bugzilla 4.0 [11]) offer the JIT feature for suggesting a bug reporter possible duplicates when s/he is filing a bug (i.e., typing in the summary field), thereby reducing chances for submitting duplicate reports in the first place. Evidence from a recently reported empirical study [10] also reveals that the state-of-the-art practices work well on bug reports submitted before employing JIT in BTSs, but perform poorly on reports submitted thereafter. Though employing JIT improves the quality of bug reporting by avoiding textually similar reports being submitted to a certain extent, there is still a substantial proportion of duplicate reports that are more semantically similar but less textually similar are failed to be detected by textual-based techniques. For example, after applying the JIT filtering, there are still 10,312 (18%) 11,931 (10%) and 1,399 (12%) duplicate bug reports in the Mozilla-Firefox, Mozilla-Core, and Eclipse-Platform projects, remain undetected [10]. This is because the current JIT technique only relies on contents of the summary field of bug reports, and it does not consider rich semantic information contained in bug report attributes such as products, components, versions, and severity and priority levels.

Observations and Insights. Figure 1a presents the heterogeneous information related to bug reporting and processing in Bugzilla. A bug (*BID*) reported by a reporter, occurs (*O*) in a specific version (*VER*). The bug can have (*H*) a priority (*PRI*) to be processed and has impacts (*I*) on users to a certain severity (*SEV*) level, e.g., high, low, normal, or blocking. The bug can be located (*L*) in a functional component (*COM*), which belongs (*B*) to a product (*PRO*). These attributes along with bug reports, like other data or information objects, are semantically connected [12], [13]. For example, two bug reports are related if they are attributed with the same component, version, priority or severity level. Such relations can be abstracted as a heterogeneous information network (HIN) [14].

Unlike homogeneous networks that have only one type of nodes and edges in a network, HIN leverages diverse and semantically rich information of typed nodes and edges in a network. Recently, representation learning [15] is a promising branch of deep learning and shown its power in learning heterogeneous information networks. Semantic relations learned with these models can be used to precisely conduct classification [16], clustering [17], and link prediction [18] tasks. Hence, we believe HIN together with representation learning is a very promising solution for identifying semantically similar duplicate bug reports in the post-JIT era. By reasoning and

⋆Corresponding author: Guanping Xiao.

2332-6549/20/$31.00 ©2020 IEEE
DOI 10.1109/ISSRE5003.2020.00027

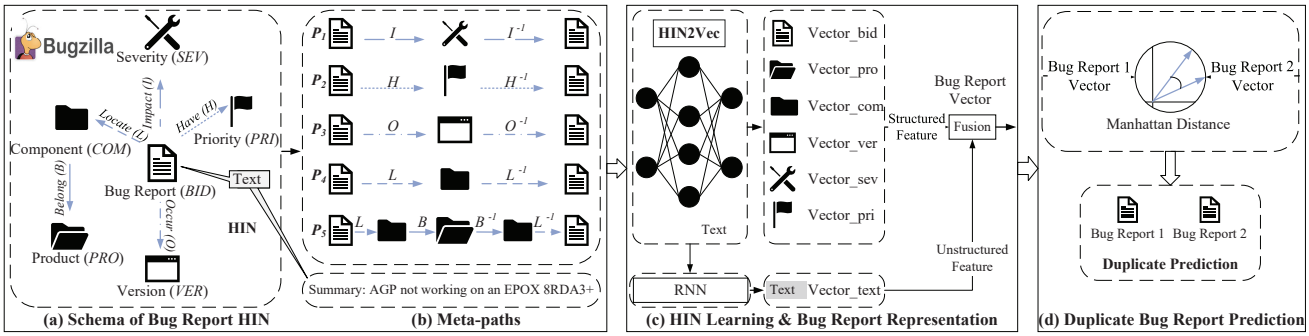


Fig. 1. Overview of HINDBR.

learning heterogeneous relations in a network, we can improve existing textual-based approaches by capturing underlying semantic correlations among different attributes in BTSs.

Our Solution. In this paper, we build an HIN for detecting semantically similar duplicate bug reports by considering comprehensive yet heterogeneous information extracted from BTSs. The HIN has six types of nodes, i.e., product (*PRO*), component (*COM*), version (*VER*), severity (*SEV*), priority (*PRI*), and bug report (*BID*), which are connected via five types of meta relations: *BID* $\leftrightarrow$ *COM*, *BID* $\leftrightarrow$ *SEV*, *BID* $\leftrightarrow$ *PRI*, *BID* $\leftrightarrow$ *VER*, and *COM* $\leftrightarrow$ *PRO*. Consequently, five meta-paths can be formulated to represent correlations between two bug reports, as illustrated in Figure 1b. Semantic relations of the bug report HIN are learned by HIN2Vec [19], a network representation learning model. The text information of a node *BID* in the HIN, i.e., the summary field of a bug report, is extracted as a text sequence. We use a recurrent neural network (RNN) for text sequence embedding, as depicted in Figure 1c. To represent bug reports and predict duplicates, we develop a deep neural network (DNN) by embedding the learned semantic relations and the text information into a low-dimensional vector space. In the DNN, the Manhattan distance is calculated between vectors of two bug reports to measure their semantic similarity (Figure 1d). We evaluate the model on 2,038,675 bug reports from nine real-world open-source projects (i.e., Eclipse, Freedesktop, GCC, GNOME, KDE, LibreOffice, Linux kernel, LLVM, and OpenOffice), using both before-JIT and after-JIT reports.

In summary, the paper has the following key contributions:

- We present HINDBR, a new deep representation learning based approach to detect semantically similar duplicate bug reports in the post-JIT setting.
- HINDBR, for the first time, introduces HIN as the backbone representation to facilitate the learning and prediction of duplicate bug reports;
- We have evaluated HINDBR by using 5-fold cross-validation with a total of 1,050,175 generated bug report pairs, including both before-JIT and after-JIT duplicates. Results show that HINDBR achieves up to 98.83% accuracy and 97.08% F1 score across the open-source projects, and outperforms the state-of-the-art deep learning-based classification model [5].
- We made our dataset and source code publicly available at https://github.com/hindbr.

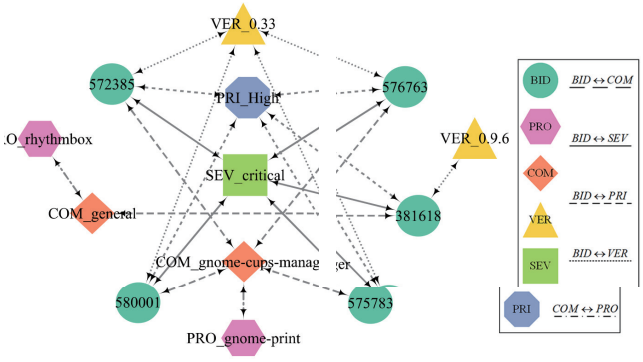


Fig. 2. An excerpt of a bug report HIN for the GNOME project.

TABLE I
SIMILARITIES OF DUPLICATE AND NON-DUPLICATE PAIRS WITH HIN VECTORS

Duplicate	Similarity	Non-Duplicate	Similarity
(576763, 575783)	1	(381618, 576763)	2.14E-224
(580001, 575783)	0.98	(381618, 580001)	4.51E-225
(576763, 572385)	0.96	(381618, 572385)	1.28E-225
(572385, 575783)	0.76	(381618, 575783)	0
(580001, 572385)	0.51		
(580001, 576763)	0.41		

Organization. The rest of this paper is organized as follows. Section II shows a motivating example. Section III briefly introduces the problem definition, HIN and its representation learning. Section IV presents our HINDBR approach. Section V describes data collection and aggregation. Section VI evaluates the effectiveness of HINDBR, while Section VII discusses the main threats to the validity. Section VIII introduces related work. Finally, Section IX concludes the paper.

II. A MOTIVATING EXAMPLE

We present a real-world scenario of duplicate bug reports in the GNOME project and illustrate how HINDBR precisely identifies that bug IDs-580001, 572385, 576763, and 575783 are duplicate, while bug ID-381618 is not similar to any of them using HIN. Figure 2 depicts a part of the HIN constructed from GNOME bug reports following the relations described in Figure 1a. We use six different colors and five different lines to differentiate the six node types and five edge types on HIN. Bug reports in Figure 2 are formulated with the five types of meta-paths (Figure 1b), based on which the employed HIN2Vec [19] learns the latent semantic relations

and represent them as a low-dimensional vector space. To represent a bug report, we concatenate the vectors of the six nodes: *BID*, *PRO*, *COM*, *VER*, *SEV*, and *PRI*. Then, we calculate the similarity between two bug reports using their vectors. TABLE I shows the similarities of the duplicate and non-duplicate pairs calculated by using a Manhattan distance with normalization.

Since duplicate bug reports have tight relations in the constructed HIN, we can see that the duplicate pairs have significantly higher similarities than those of non-duplicate ones, e.g., the distance of the HIN vectors of bug IDs-580001, 572385, 576763, and 575783 are very close (the 2nd column of TABLE I), while they are far from the vector of bug ID-381618 (the 4th column of TABLE I), showing that the constructed HIN is effective in detecting duplicate bug reports.

III. Background

A. Duplicate Bug Report Prediction

Duplicate bug reports are those that describe the same failure. Figure 3 shows an example of a duplicate bug report in the Linux kernel project. Once a newly submitted bug report is identified duplicated to an existing one, the <*resolution*> will be marked as *DUPLICATE*. In the report, the <*dup_id*> records a reference to an existing bug that the current bug is duplicate of. Usually, bug reports can be organized into groups [20]. According to the <*dup_id*>, all reports representing the same bug are arranged into the same group. In each group, one bug report that all other duplicate reports refer to would be considered as the *master bug report*. Note that if no duplicate of a new bug report is found, the bug report is the *master bug report* and forms a new group. A bug group from the GNOME project is presented as an example in TABLE II. In this group, three bugs (i.e., IDs-575783, 576763, and 580001) are duplicates of the master bug report ID-572385. After obtaining all the bug groups, pairs of duplicate and non-duplicate bug reports can be generated.

Consequently, the problem of identifying duplicate bug reports can be formulated as a simple binary prediction problem instead of a supervised ranking problem [5], [10], [20]. When a new bug report is submitted, it can be paired with all the *master bug reports* and a trained prediction model is used to predict whether these pairs are duplicates.

B. Heterogeneous Information Network

Many real-world complex systems ranging from nature to human society can be abstracted as information networks [21], where entities (or relations) are denoted by nodes (or edges). Such an abstraction not only represents and stores the essential information about a complex system but also provides a useful perspective to mine knowledge from it. We define information networks [14] and related concepts as follows.

Definition 1 (Information Network): An *information network* is a directed graph $G = (V, E)$ with a node type mapping function τ: $V \rightarrow A$ and an edge type mapping function ϕ: $E \rightarrow R$, where each node $v \in V$ belongs to one particular node type $\tau(v) \in A$, each edge $e \in E$ belongs

```
<bugzilla maintainer="helpdesk@kernel.org" urlbase="https://bugzilla.kernel.org/"
version="5.1.1">
<bug>
<bug_id>200389</bug_id>
<creation_ts>2018-07-02 01:59:58 +0000</creation_ts>
<short_desc>iwlmvm: 7265: stops working after kernel warning / trace</short_desc>
<product>Drivers</product>
<component>network-wireless</component>
<version>2.5</version>
<bug_status>CLOSED</bug_status>
<resolution>DUPLICATE</resolution>
<dup_id>199967</dup_id>
<priority>P1</priority>
<bug_severity>normal</bug_severity>
...
</bug>
</bugzilla>
```

Fig. 3. Linux bug report ID-200389 (XML format).

TABLE II
AN EXAMPLE OF BUG GROUP IN GNOME PROJECT

Type	Bug ID	Summary
Master	572385	crash in Printing: Just clicked the gnome-c...
Duplicates	575783	crash in Printing:
	576763	crash in Printing: launching gnome-cups-man...
	580001	crash in Printing: Checking to see why I co...

to a particular relation $\phi(e) \in R$. When the types of nodes $|A| > 1$ or the types of relations $|R| > 1$, the network is called a *heterogeneous information network*; otherwise, it is a *homogeneous information network*.

The meta-structure of an HIN is described as *network schema* as follows.

Definition 2 (Network Schema): The *network schema* $T_G = (A, R)$, is a meta template for a given HIN $G = (V, E)$ with the node type mapping τ: $V \rightarrow A$ and the edge type mapping ϕ: $E \rightarrow R$, which is a directed graph defined over node types A, with edges as relations from R.

Given a specific network schema, the relations between two nodes are defined by *meta-paths* as follows.

Definition 3 (Meta-path): A *meta-path* P is a path defined on the graph of network schema $T_G = (A, R)$, and is denoted as:

$$P = A_1 \xrightarrow{R_1} A_2 \xrightarrow{R_2} \dots \xrightarrow{R_l} A_{l+1}, \quad (1)$$

which defines a composite relation $R = R_1 \cdot R_2 \cdot \dots \cdot R_l$ between node types A_1 and A_{l+1}, where $\cdot$ denotes the composition operator on relations. Note that a path p, which goes through nodes $v_1, v_2, \dots, v_{i+1}$, is an instance of the meta-path P, if $\forall i = 0, \dots, l$, $A_i = \tau(v_i)$ and $R_i = \phi(v_i, v_{i+1})$.

C. Representation Learning on HIN

Several representation learning models on network data have been proposed, such as node2vec [15] and LINE [22]. Here, we focus on introducing a recent representation learning technique, named HIN2Vec [19], particularly developed for heterogeneous networks. It is formulated as below.

Definition 4 (Representation learning on HIN): Given an HIN $G = (V, E)$. The *representation learning* aims to learn a function f: $V \rightarrow \mathbb{R}^d$ that projects each node $v \in V$ to a vector in a d-dimensional space $\mathbb{R}^d$, where $d \ll |V|$.

Figure 4 shows the internal working of the HIN2Vec model. It reduces the tasks of predicting the probabilities of rela-

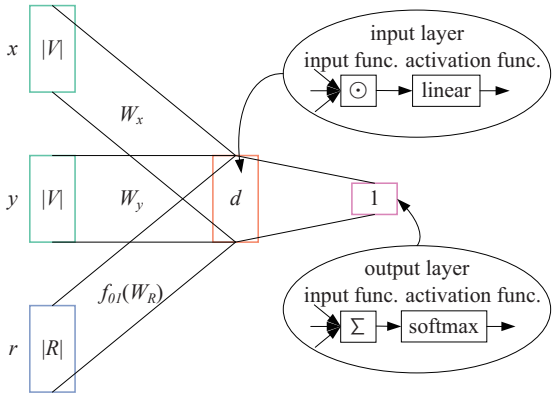


Fig. 4. The HIN2Vec neural network model.

tionships between two nodes into new prediction tasks, i.e., whether two nodes x and y, have a specific relationship r [19]. The advantage of the model is that it avoids scanning for all relationships in the data preparation and examining/updating for all relationships during training. The HIN2Vec model is a binary classifier, which takes a pair of nodes x and y, and a relationship $r \in R$ as the input to predict whether nodes x and y have the relationship r.

In the input layer, three one-hot vectors, $\vec{x}$, $\vec{y}$, and $\vec{r}$, are transformed into latent vectors $W_X^{'}\vec{x}$, $W_Y^{'}\vec{y}$, and $f_{01}(W_R^{'}\vec{r})$, in the latent layer. Note that the treatment on r is different from x and y since they have different semantics and implications in the learning process. Thus, a regularization function $f_{01}(.)$ is used to restrict the values of latent vector for r to be between 0 and 1, aiming to avoid negative values in W_R and to prevent values in W_R from becoming too large. The three latent vectors are aggregated by the Hadamard function, i.e., element-wise multiplication, denoted by $W_X^{'}\vec{x} \odot W_Y^{'}\vec{y} \odot f_{01}(W_R^{'}\vec{r})$, and then the identify function is applied for activation. For the output layer, it takes a Summation function as input and a Sigmoid function for activation, computing $sigmoid(\sum W_X^{'}\vec{x} \odot W_Y^{'}\vec{y} \odot f_{01}(W_R^{'}\vec{r}))$ to realize logistic classification. Note that W_X and W_Y are made to be identical, so that W_X and W_R consist of learned node vectors and meta-path vectors, respectively.

IV. Our HINDBR Approach

We formulate the duplicate bug report prediction as a representation learning problem on HIN. This section first introduces a new method for constructing HIN for bug reports, then presents detailed of HINDBR.

A. Constructing HIN for Bug Reports

Figure 1a shows the schema of the HIN for bug reports. It has six types of nodes: bug report (*BID*), component (*COM*), product (*PRO*), version (*VER*), priority (*PRI*), and severity (*SEV*). We only consider bug reports from Bugzilla. Reports from other BTSs may have different attributes. The text information (e.g., summary) of a bug report is considered as the content of node *BID*. To describe relations between bug reports, the five relations below are preserved in the network:

R1: Bug-Component. Each bug report records a bug that is located in a specific functional component of a software project. $\boldsymbol{L}$ and $\boldsymbol{L}^{-1}$ describe relations between bug reports and components: $BID \xrightarrow{L} COM$ and $COM \xrightarrow{L^{-1}} BID$.

R2: Component-Product. For Bugzilla, a component is a functional decomposition belonged to a product, which is further resulted from the functional decomposition of a software project (e.g., *Drivers* in the Linux kernel project) or a specific software product (e.g., *gedit* in the GNOME project). We use $\boldsymbol{B}$ and $\boldsymbol{B}^{-1}$ to denote relations between components and products, i.e., $COM \xrightarrow{B} PRO$ and $PRO \xrightarrow{B^{-1}} COM$.

R3: Bug-Version. To describe the case that a bug occurs in a specific version as a relation, we use $\boldsymbol{O}$ and $\boldsymbol{O}^{-1}$ to represent relations between bug reports and versions, i.e., $BID \xrightarrow{O} VER$ and $VER \xrightarrow{O^{-1}} BID$.

R4: Bug-Priority. Each bug report has a priority to be processed by developers. Then, we use $\boldsymbol{H}$ and $\boldsymbol{H}^{-1}$ to denote relations between bug reports and priorities, i.e., $BID \xrightarrow{H} PRI$ and $PRI \xrightarrow{H^{-1}} BID$.

R5: Bug-Severity. Each bug is also characterized with a severity (e.g., high, low, normal, or blocking) level from the perspective of its reporter. We use $\boldsymbol{I}$ and $\boldsymbol{I}^{-1}$ to describe relations between bug reports and severity, i.e., $BID \xrightarrow{I} SEV$ and $SEV \xrightarrow{I^{-1}} BID$.

Given a network schema with various types of nodes and relations, we generate five types of meta-paths, i.e., $P_1 - P_5$ as depicted in Figure 1b, for capturing their semantic correlations among bug reports in Bugzilla. Different meta-paths characterize relations between two bug reports from different aspects. For example, a typical one to formulate relationships between bug reports in Bugzilla is: P_5: $BID \xrightarrow{L} COM \xrightarrow{B} PRO \xrightarrow{B^{-1}} COM \xrightarrow{L^{-1}} BID$, which means that two bug reports are connected as they are located in the components of the same product. After constructing the HIN, HIN2Vec (Section III-C) is then used to learn latent vectors of nodes in the HIN. The learned node vectors are then used as inputs for the proposed HIN-based neural network.

B. HINDBR: Representation Learning on HIN

To predict duplicate bug reports, we propose a novel DNN named HINDBR. Figure 5 shows the detailed structure of HINDBR. The neural network consists of two parts: a bug report representation module and a similarity module that measures the degree of similarity of a bug report pair, which are described in detail below.

1) Feature Representation and Fusion: Given a constructed HIN containing semantic relations among all bug reports, the next step is to extract its features for representation learning. An input bug report is represented by seven features from the constructed HIN: *BID*, *PRO*, *COM*, *VER*, *SEV*, *PRI*, and $Text_{BID}$. The first six features are the six types of nodes in the HIN, while $Text_{BID}$ is the text information of node *BID*, i.e., summary. The seven features are divided into two categories:

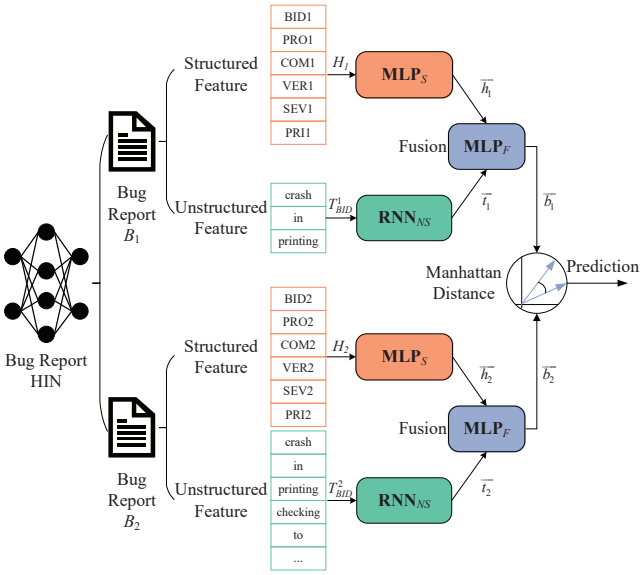


Fig. 5. Detailed structure of HINDBR.

the structured and unstructured features. Their details are given below.

Structured Feature. *BID*, *PRO*, *COM*, *VER*, *SEV*, and *PRI*, are treated as the structured features. HINDBR, first, maps these features to a d_1-dimensional vector $h_i \in \mathbb{R}^{d_1}$ where $i = BID, PRO, COM, VER, SEV$ or PRI learned by HIN2Vec based on the structure of the HIN. Then, the six features are concatenated into a whole structured feature, denoted as $H = [h_{BID}, h_{PRO}, h_{COM}, h_{VER}, h_{SEV}, h_{PRI}]$, $H \in \mathbb{R}^{6d_1}$. Since H has no strict order, it is simply embedded by a multilayer perceptron (MLP, i.e., the conventional fully connected layer [23]), denoted as $\mathbf{MLP}_S$:

$$h = tanh(W^H H), \tag{2}$$

where H represents the concatenated structured feature, $W^H \in \mathbb{R}^{k_1 \times 6d_1}$ is the matrix of trainable parameters in the $\mathbf{MLP}_S$ (k_1 is the number of hidden units of the $\mathbf{MLP}_S$), $tanh$ is the activation function used in the $\mathbf{MLP}_S$, while $h \in \mathbb{R}^{k_1}$ is the final vector of the whole structured feature.

Unstructured Feature. In addition to the structured features, text information $Text_{BID}$ is also embedded in HINDBR. We use an RNN for the embedding of sentences [24], [25]. Figure 6 shows an example of sequence embedding for the summary of a bug report using an RNN. It reads the words in the sentence one by one and outputs the hidden states h_1, h_2, and h_3. The first word "crash", denoted as word vector x_1, is read by the RNN. Then, the current hidden state h_1 is computed by considering x_1 and initial hidden state h_0. Likewise, the RNN reads the second word x_2 (i.e., "in") and updates the hidden state h_1 to h_2 using x_2. Finally, the last word x_3 (i.e., "printing") is read and the final state is updated as the output. Typically, the last state h_3 is used as the embedding vector of the whole sentence.

Given a *BID*'s summary $T_{BID} = \{x_1, x_2, \ldots, x_{N_T}\}$, where x_i is a word token of a sentence with the length of N_T,

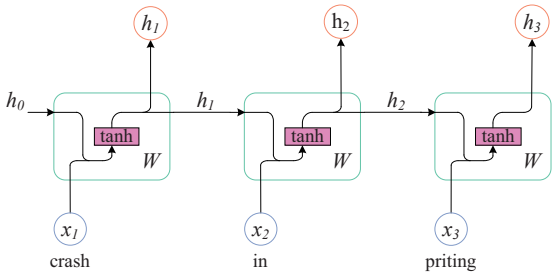


Fig. 6. Sequence embedding using an RNN.

HINDBR embeds the sequence of split word tokens using an RNN (denoted as $\mathbf{RNN}_{NS}$):

$$t_i = tanh(W^T[x_i, t_{i-1}]), \forall i = 1, 2, \ldots, N_T, \tag{3}$$

where $x_i \in \mathbb{R}^{d_2}$ is the embedding vector of word token x_i learned by Word2Vec [26] on bug report corpus, $t_i \in \mathbb{R}^n$ is the hidden state at time step i (n is the number of hidden units of the $\mathbf{RNN}_{NS}$), $[x_i, t_{i-1}] \in \mathbb{R}^{d_2+n}$ is the concatenation of two vectors, $W^T \in \mathbb{R}^{n \times (d_2+n)}$ is the matrix of trainable parameters in the $\mathbf{RNN}_{NS}$, $tanh$ is the activation function. The unstructured feature is thus embedded as an n-dimensional vector t ($t = t_{N_T}$).

Feature Fusion. After obtaining the structured feature vector h and the unstructured feature vector t, they are fused into one final vector by an MLP (denoted as $\mathbf{MLP}_F$):

$$b = tanh(W^B[h, t]), \tag{4}$$

where $[h, t] \in \mathbb{R}^{k_1+n}$ describes the concatenation of two vectors, $W^B \in \mathbb{R}^{k_2 \times (k_1+n)}$ is the matrix of trainable parameters in the $\mathbf{MLP}_F$ (k_2 is the number of hidden units of the $\mathbf{MLP}_F$), while $tanh$ is the activation function of the $\mathbf{MLP}_F$. The final embedding of the bug report is represented by the output vector $b \in \mathbb{R}^{k_2}$.

2) Similarity Module: The module is used to measure the similarity of two bug reports with their vectors obtained by the representation module. In this work, we use the Manhattan distance as the measure [27], which is defined as:

$$S(b_1, b_2) = exp(-||b_1 - b_2||_1), \tag{5}$$

where b_1 and b_2 are the vectors of two bug reports, while $exp(.)$ normalizes the distance value between 0 and 1.

3) Model Training: To train HINDBR, we construct each training instance as $< B_1, B_2 >$. If bug reports B_1 and B_2 are duplicate of each other, label y of the instance is set to 1, and 0 otherwise. When using the set of $< B_1, B_2 >$ for training, HINDBR predicts the Manhattan distance similarities $\hat{y}$ and minimizes the binary cross entropy loss function [28]:

$$\mathcal{L}(\theta) = -(ylog(\hat{y}) + (1-y)log(1-\hat{y})), \tag{6}$$

$$\hat{y} = \begin{cases} 1, & S(B_1, B_2) \geq 0.5 \\ 0, & S(B_1, B_2) < 0.5 \end{cases}, \tag{7}$$

where θ denotes the model parameters, y is the true label, $\hat{y}$ is the predicted label calculated by a given Manhattan distance similarity threshold (i.e., 0.5).

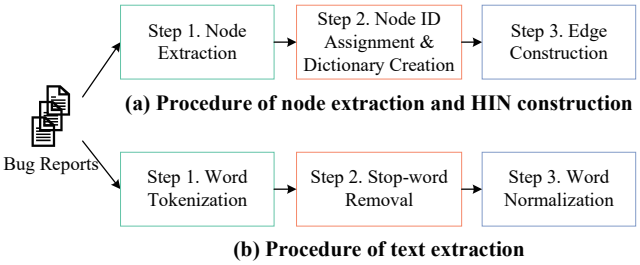
V. Data Collection and Aggregation

A. Data Collection

TABLE III shows the detailed information of the collected dataset, including project types, names, time frames, the year of employing the JIT feature, and the number of bug reports. Most previous studies evaluated their approaches only using before-JIT reports, which have different characteristics as post-JIT reports [10]. In our experiments, we collected 2,038,675 bug reports from nine popular open-source projects, i.e., Eclipse, Freedesktop, GCC, GNOME, KDE, LibreOffice, Linux kernel, LLVM, and OpenOffice, covering both before-JIT and after-JIT bug reports. We collected all the bug reports by downloading their XML files (provided by Bugzilla) via a web crawler that we developed. The collected bug data and the web crawler are available at https://github.com/hindbr.

TABLE III
Collected Bug Reports

Project Type	Project	Time Frame	JIT Year	# of Reports
Development Tool	Eclipse	10/10/01 - 09/30/18	2011 [29]	528,862
	GCC	08/03/99 - 09/30/18	2011 [30]	81,463
	LLVM	10/07/03 - 09/30/18	Unknown	38,107
Desktop Environment	Freedesktop	01/09/03 - 09/30/18	2011 [31]	106,065
	GNOME	02/05/99 - 09/30/18	Unknown	673,301
	KDE	01/21/99 - 09/30/18	2012 [32]	388,711
Office Suite	LibreOffice	08/03/10 - 09/30/18	Unknown	62,029
	OpenOffice	10/16/00 - 09/30/18	2012 [33]	127,797
Operating System	Linux kernel	11/06/02 - 09/30/18	2012 [34]	32,340
Total				2,038,675



(a) Procedure of node extraction and HIN construction

(b) Procedure of text extraction

Fig. 7. Procedure of constructing HIN and extracting text.

B. Feature Extraction

HIN features are extracted from collected bug reports by following the steps in Figure 7 and discussed in detail below.

HIN Construction. For each bug report, we extract the nodes and construct bug report HIN by the following procedure shown in Figure 7a.

Step 1. Node Extraction. As described in Section IV-A, six types of attributes (e.g., *BID*, *PRO*) are extracted by regular expressions. For example, `<product>(.*)</product>` is used to extract *PRO* (product) from the XML file (e.g., the one shown in Figure 3). To avoid potential duplicate names of nodes, each extracted content is concatenated with a prefix, e.g., 'PRO_' for the product node *PRO*. Similar regular expressions are used for extracting other types of HIN nodes.

Step 2. Node ID Assignment & Dictionary Creation. To store the nodes, we create an empty node dictionary `node_dict`, where `key` is the node's name and `value` is a tuple ($node_{id}$, $node_{type}$). The $node_{id}$ is assigned to a number, i.e., `len(node_dict) + 1`, while $node_{type}$ denotes the type of the node, e.g., *COM*. The node dictionary is also served as an index for mapping the node's name to its corresponding vector when constructing the model.

Step 3. Edge Construction. According to the relations defined in Section IV-A, we construct the edges of the bug report HIN as the following format: $node1_{id}$ $node1_{type}$ $node2_{id}$ $node2_{type}$ $edge_{type}$, separated by "\t". Note that $edge_{type}$ is the relationship between nodes.

After processing all the bug reports of a project by these steps, an HIN is constructed for the report.

Text Extraction. We extract the text, i.e., summary of each bug report, from its XML file by `<short_desc>(.*)</short_desc>`. Then, the sequence of the text is processed by the three steps shown in Figure 7b.

Step 1. Word Tokenization. The text extracted by the regular expression is first divided into a stream of words. This step removes, from the text, all numbers, punctuation, and other non-alphabetic characters such as "+", "-", and "=", with spaces. For example, given bug report ID-575783 "crash in Printing:", after this step, we obtain ["crash","in","Printing"].

Step 2. Stop-word Removal. This step removes stop-words from the tokens of words. Stop-words are frequently used words containing no information, such as "and", "the", and "to". These words do not contribute to information retrieval and text mining. For instance, after removing stop-words of bug report ID-575783, we obtain ["crash", "Printing"].

Step 3. Word Normalization. This final step converts each word token to its lower case. For example, the final output of the summary of bug report ID-575783 is ["crash", "printing"].

We also extracted descriptions and comments of bug reports by `<thetext>(.*?)</thetext>` to form a text corpus. Word2Vec [26] then uses it to train word embedding vectors.

C. Bug Pairs Generation

1) Bug Pairs for Model Training and Testing: TABLE IV shows the generated bug pairs of each project for training and testing HINDBR. As discussed in Section III-A, reports representing the same bug are put into the same bug group. For duplicate pairs, we generate combinations of duplicate bug pairs from a bug group. As the example shown in TABLE II, six pairs of bug reports can be generated: (572385, 575783), (572385, 576763), (572385, 580001), (575783, 576763), (575783, 580001), and (576763, 580001). Same as the ratio setting of the duplicate pairs and the non-duplicate pairs in [20], we randomly generated non-duplicate pairs, the number of which is four times larger than the total number of duplicate pairs, by combining two master reports from different bug groups.

2) Evaluating Bug Pairs under the Before-JIT and After-JIT Settings: To evaluate the performance of HINDBR in terms of handling before-JIT and after-JIT duplicates, two kinds of

TABLE IV
NUMBER OF BUG PAIRS FOR MODEL TRAINING AND TESTING

Project	Duplicate Pair	Non-Duplicate Pair	Pair
Eclipse	54,742	218,968	273,710
Freedesktop	11,316	45,264	56,580
GCC	7,819	31,276	39,095
GNOME	69,381	277,524	346,905
KDE	41,094	164,376	205,470
LibreOffice	6,771	27,084	33,855
Linux kernel	2,998	11,992	14,990
LLVM	3,093	12,372	15,465
OpenOffice	12,821	51,284	64,105
Total	210,035	840,140	1,050,175

TABLE V
NUMBER OF BUG PAIRS FOR BEFORE-JIT AND AFTER-JIT EVALUATION

Project	Before-JIT		After-JIT	
	Duplicate	Non-Duplicate	Duplicate	Non-Duplicate
Eclipse	5,474	21,896	5,474	21,896
Freedesktop	1,131	4,524	1,131	4,524
GCC	781	3,124	781	3,124
KDE	4,109	16,436	4,109	16,436
Linux kernel	299	1,196	299	1,196
OpenOffice	1,282	5,128	1,282	5,128

bug pairs are generated: before and after JIT, as shown in TABLE V. Note that we ignored reports that were submitted in the year when the JIT was deployed (e.g., 2012 for Linux kernel). This is because reporters may need time to get familiar with the usage of the new JIT retrieval feature [10]. Again, the number of non-duplicate pairs is also four times larger than that of the duplicate pairs.

VI. EXPERIMENT SETUP AND EVALUATION

A. Implementation Details

1) Settings of Pre-trained Embeddings: The implementations of two pre-trained embeddings, i.e., HIN and word embeddings, use HIN2Vec [19] and Word2Vec [26], respectively.

HIN Embedding. As recommended in [19], we set the HIN2Vec parameters as follows: dimension of node representation d_1 is 128; context window w is 4; length of random walks l is 1280; number of negative samples per positive sample n is 5. We perform training on each project's HIN constructed by the procedures described in Section V-B.

Word Embedding. Parameter settings of the word embedding are as follows: dimension of word vectors d_2 is 100; context window w is 10; training algorithm is Skip-Gram; min_count is 5. Training was conducted on the text corpus extracted from all the 2,038,675 collected bug reports.

2) Settings of Neural Networks in HINDBR*:* To investigate the impact of feature settings on the performance of HINDBR, we build three models under different bug report representations, denoted as Text, HIN1 (no Text), and HIN2 (with Text).

Text. In this model, we use the text of node *BID* to represent a bug report. Thus, the $\mathbf{MLP}_S$ and $\mathbf{MLP}_F$ networks are removed. The $\mathbf{RNN}_{NS}$ is implemented by the bi-directional long short-term memory (Bi-LSTM) [35], a state-of-the-art variant of RNN. The number of hidden units of Bi-LSTM is 100 in each direction. So, the embedded unstructured feature t is a 200-dimensional vector (two directions in Bi-LSTM).

HIN1 (no Text). In this model, we only use the structured feature for bug report representation. The number of hidden units in the $\mathbf{MLP}_S$ is 32.

HIN2 (with Text). For the default feature setting, HINDBR represents a bug report by fusing structured and unstructured features. The settings of the $\mathbf{RNN}_{NS}$ for the unstructured feature and the settings of the $\mathbf{MLP}_S$ for the structured feature are the same as aforementioned. The fusion layer, i.e., $\mathbf{MLP}_F$, has 64 hidden units.

3) Settings of Model Training: We built our model on Keras [36], an open-source deep learning framework based on TensorFlow. All the experiments were conducted on a DELL Precision Tower with a 3.60 GHz Intel i9-9900K CPU, 32 GB memory, 512GB SSD and 2TB HDD storage, and an NVIDIA RTX 2080Ti GPU, running Ubuntu 18.04.

Training Parameters. Training settings are as follows: number of epochs is 100; batch size is 128; gradient clipping value is 1.25; ratio of validation dataset split from training dataset during training process is 0.2. Optimizations of the parameters are enabled using the Adadelta method along with gradient clipping [37], [38].

Stratified Cross-Validation Evaluation. We use stratified 5-fold cross-validation to evaluate the effectiveness of HINDBR, i.e., four folds are randomly generated to train HINDBR, while the remaining one fold is used for testing. To ensure that each fold can be tested, we perform five iterations. Thus, there are five values for each evaluation metric and we report their average values. Note that the ratios of duplicate pairs and non-duplicate pairs in each fold are the same as those in the original dataset, when using stratified k-fold cross-validation technique.

Dealing with Imbalanced Data. Due to the imbalanced data of duplicate and non-duplicate pairs, we perform oversampling during cross-validation [39], i.e., we only oversampled the duplicate pairs in the training folds, but keep the ratio of duplicate and non-duplicate pairs in the testing fold, so that to avoid overfitting that affects the model's performance. We use a popular oversampling algorithm, i.e., Synthetic Majority Oversampling Technique (SMOTE) with Tomek Links (SMOTE + TL) as the oversampling technique during the cross-validation procedure [39].

4) Evaluation Metrics: To evaluate our model, we use the following four metrics.

Accuracy. It is calculated as the proportion of correctly predicted duplicate and non-duplicate bug report pairs relative to the total number of bug report pairs, and defined as:

$$Accuracy = \frac{TP + TN}{TP + FN + FP + TN}, \tag{8}$$

where TP is true positives (i.e., duplicate pair), TN is true negatives (i.e., non-duplicate pair), FP is false positives, and FN is false negatives.

Precision. It is calculated as the proportion of instances correctly predicted to belong to a class relative to all instances that are predicted to belong to this class. For duplicate pairs, *precision* is defined as:

TABLE VI
PREDICTION RESULTS OF HINDBR COMPARED WITH BASELINE APPROACH DLDBR

Project	Accuracy			Precision			Recall			F1 Score		
	DLDBR	HINDBR	Impro.	DLDBR	HINDBR	Impro.	DLDBR	HINDBR	Impro.	DLDBR	HINDBR	Impro.
Eclipse	0.8910	**0.9489**	**6.51%**	0.8196	**0.9005**	**9.87%**	0.7930	**0.8374**	**5.60%**	0.8037	**0.8671**	**7.89%**
Freedesktop	0.9161	**0.9621**	**5.01%**	0.8503	**0.9184**	**8.01%**	0.8519	**0.8891**	**4.36%**	0.8504	**0.9035**	**6.24%**
GCC	0.9061	**0.9587**	**5.81%**	0.8523	**0.9205**	**8.01%**	0.8306	**0.8721**	**5.01%**	0.8392	**0.8957**	**6.73%**
GNOME	0.9843	**0.9883**	**0.42%**	0.9620	**0.9709**	**0.93%**	**0.9769**	0.9707	-0.63%	0.9693	**0.9708**	**0.15%**
KDE	0.9639	**0.9834**	**2.02%**	0.9363	**0.9651**	**3.08%**	0.9312	**0.9508**	**2.11%**	0.9333	**0.9579**	**2.64%**
LibreOffice	0.8440	**0.9277**	**9.91%**	0.7708	**0.8538**	**10.76%**	0.7022	**0.7703**	**9.69%**	0.7259	**0.8096**	**11.53%**
Linux kernel	0.8943	**0.9578**	**7.10%**	0.8242	**0.8961**	**8.72%**	0.8321	**0.8925**	**7.26%**	0.8274	**0.8942**	**8.08%**
LLVM	0.8388	**0.9296**	**10.82%**	0.7500	**0.8423**	**12.31%**	0.7033	**0.7903**	**12.38%**	0.7115	**0.8154**	**14.61%**
OpenOffice	0.8432	**0.9487**	**12.51%**	0.7508	**0.8969**	**19.45%**	0.7464	**0.8369**	**12.13%**	0.7454	**0.8658**	**16.16%**

$$Precision = \frac{TP}{TP + FP}. \tag{9}$$

Recall. It is the proportion of correctly predicted instances of a class relative to all actual instance of that class. For the duplicate pairs, it is calculated as:

$$Recall = \frac{TP}{TP + FN}. \tag{10}$$

F1 Score. The *F1 score* combines *precision* and *recall* by the harmonic average, calculated as:

$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}. \tag{11}$$

B. Model Evaluation

Our evaluation aims to answer the following three research questions (RQs):

- **RQ1**: How does HINDBR perform in predicting duplicate bug reports compared with the state-of-the-art deep learning-based approach?
- **RQ2**: How do different feature settings impact HINDBR's performance?
- **RQ3**: How is the performance of HINDBR in handling duplicate bug reports before and after JIT deployed?

1) Comparison Method: We compare the effectiveness of HINDBR with a recently-proposed state-of-the-art model (we named it DLDBR) [5]. DLDBR detects duplicate bug reports by using DNNs, which learns rich information from text information of bug reports (e.g., a LSTM for embedding short descriptions and convolutional neural networks (CNNs) for embedding long descriptions). DLDBR consists of two different models, i.e., retrieval model and classification model. Because our model is designed for the duplicate bug report prediction problem instead of a supervised ranking problem, we compare HINDBR with the classification model of DLDBR. For the experiments, we implemented DLDBR based its description in [5], and used the same cross-validation and imbalanced data processing strategy as those in HINDBR.

2) Results and Analysis: ***RQ1:*** **HINDBR** ***Effectiveness.*** We first present the prediction results of HINDBR (i.e., HIN2), as shown in TABLE VI. In the table, the 1st column lists all the studied projects; the 2nd column lists the accuracy and the improvement of HINDBR over DLDBR, calculated as $(Per_{\text{HINDBR}} - Per_{\text{DLDBR}})/Per_{\text{DLDBR}}$; the 3rd-5th columns present the precision, recall, and F1 score for the duplicate class. We can find that HINDBR achieves up to 98.83% accuracy, 97.09% precision, 97.07% recall rate, and 97.08% F1 score across the evaluated projects. For example, for the GNOME and KDE projects, the model obtains more than 95% in terms of accuracy and F1 score. The result clearly demonstrates the practicality of our model.

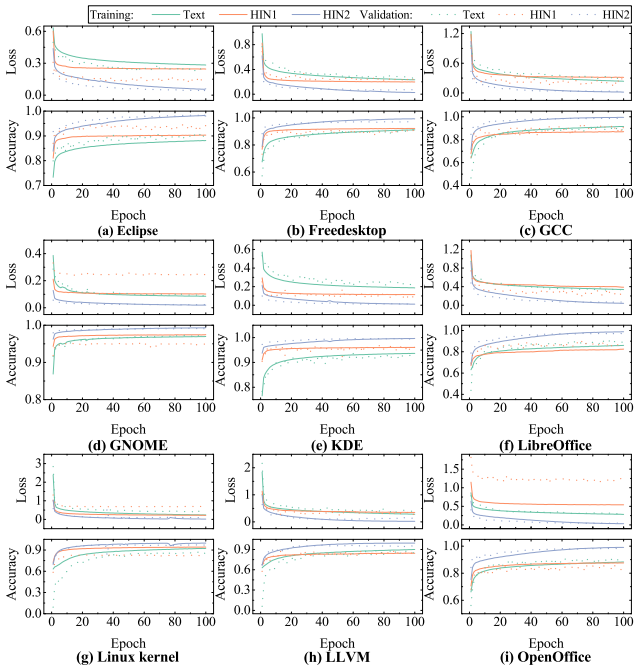


Fig. 8. Comparison of training history under different feature settings.

In addition, it can be observed from TABLE VI that HINDBR achieves better performance in all the projects than DLDBR. For example, the improvements in the accuracy and the F1 score are 10.82% and 14.61% respectively for the LLVM projects, and 12.51% and 16.16% respectively for the OpenOffice project. Moreover, HINDBR needs fewer texts compared with DLDBR. Only the short description (i.e., summary) is used in HINDBR. However, both short and long descriptions (i.e., summaries and descriptions) are needed in DLDBR.

RQ2: Impacts of Feature Settings. HINDBR represents bug reports using two categories of features in the constructed HIN,

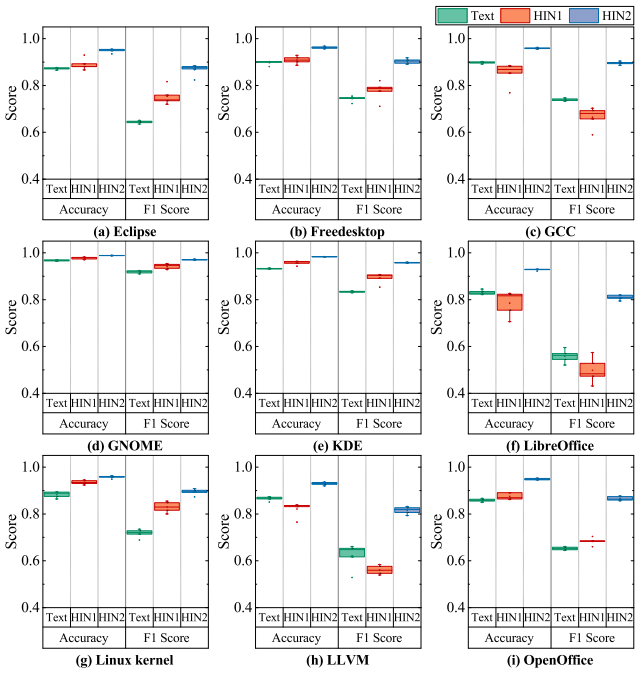


Fig. 9. Comparison of performance under different feature settings.

i.e., the structured feature and unstructured feature. To explore the impacts of different features on HINDBR's performance, we compare the training history and predicted performance of HINDBR under different feature settings, as shown in Figures 8 and 9 respectively. From the two figures, we have the following three findings:

(1) HINDBR (i.e., HIN2, using both structured and unstructured features) achieves the best performance during model training. Figure 8 shows that the loss and accuracy of the training and validation for HIN2 are the best (i.e., the smallest loss and the highest accuracy values) across all projects. (2) Using the structured feature alone (HIN1) outperforms the setting when using the unstructured feature alone (Text) for most projects, i.e., Eclipse, Freedesktop, GNOME, KDE, Linux kernel, and OpenOffice, as shown in Figure 9 (tested by the Mann–Whitney U test with $\alpha = 0.05$ [40]). However, using the structured feature alone is not effective in some projects (e.g., LibreOffice). To further investigate this, we use t-SNE with PCA [41] to visualize the structured feature vectors of bug report pairs from the GNOME and LibreOffice projects, as shown in Figure 10. We can see that the duplicate and non-duplicate pairs are very clearly clustered for the GNOME project, while it is not clear for the LibreOffice project. In addition, Figure 9 shows that the interquartile range (IQR) of the model (HIN1) using the structured feature alone is significantly larger than that of using unstructured feature alone (Text). This implies that the performance of the model using the structured feature alone is not stable. (3) Considering all the features for bug report representation learning improves the stability (i.e., a smaller IQR) and the prediction performance significantly.

RQ3: Impacts of Before-JIT and After-JIT Duplicates.

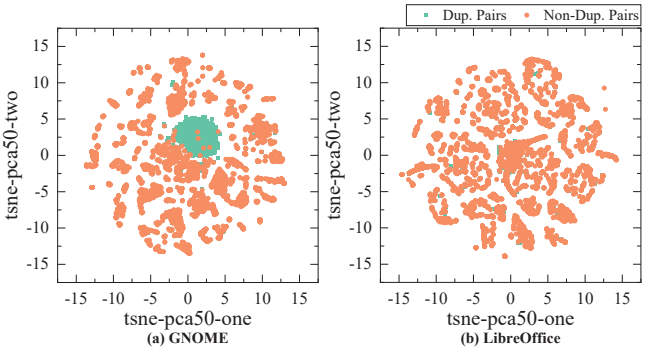


Fig. 10. t-SNE visualization of structured feature vectors of bug pairs.

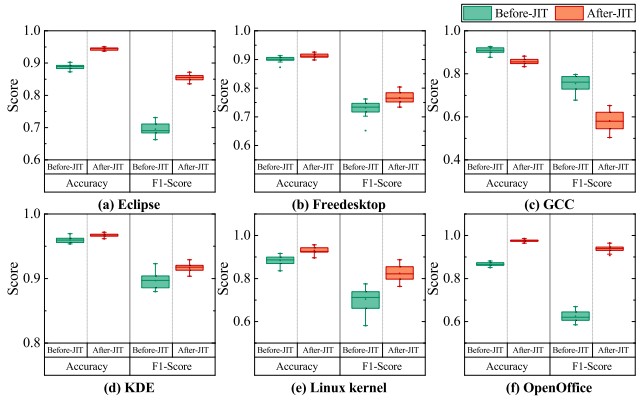


Fig. 11. Comparison of performance on before-JIT and after-JIT datasets.

In this RQ, we evaluate the effectiveness of HINDBR on duplicates before and after the JIT feature is deployed. Figure 11 shows the distributions of performance of HINDBR in the before-JIT and after-JIT datasets in TABLE V. We can find that for most projects (5/6), the performance of HINDBR on the after-JIT dataset is significantly higher than that on the before-JIT dataset (tested by the Mann–Whitney U test with $\alpha = 0.05$ [40]). The result is reasonable, as our model uses learned semantic relations to detect duplicates, although the after-JIT duplicates have lower text similarity compared with before-JIT duplicates [10]. In addition, for the GCC project (Figure 11c), the lower performance of HINDBR on the after-JIT dataset is due to the reason that the structured feature of the HIN of GCC bug reports is less effective than the unstructured feature. It can be observed from Figure 9c that the model (HIN1) using structured feature alone has the lowest performance. This may be due to the weakly connected attribute nodes in the bug report HIN.

VII. Threats to Validity

Threats to Internal Validity. The empirical data collection in our feature extraction and model implementation can be considered as a threat to the internal validity. To reduce this threat, we implemented our technique carefully and utilized state-of-the-art tools and frameworks, such as Keras and Gensim. The field reassignment of bug reports [42], i.e., changing of bug report fields, is another threat to the internal validity. When constructing bug report HINs, we followed DLDBR [5]

and used the final field values of the bug reports. We found that the field reassignment of the bug reports may have an impact on HINDBR's performance. In the future, we plan to investigate such an impact, by constructing bug report HINs through initially submitted fields and final fields, respectively.

Threats to External Validity. First, for simplicity, we ignored the temporal nature of bug reports in the repository when generating bug report pairs. We acknowledge that this setting is a main threat to the external validity (i.e., the generalization of our model). Second, the 5-fold cross-validation setting for model evaluation is also a threat to the external validity, since only a small subset of dataset was used for our evaluation. HINDBR is applicable for any project with BTSs using Bugzilla, and it is easy to be adopted to other BTSs, such as JIRA and MantisBT. In the future, we plan to consider the temporal nature of bug reports and using lager-scale dataset to evaluate HINDBR.

Threats to Construct Validity. The choice of the evaluation metrics of prediction performance can be a threat to the construct validity. To reduce this threat, we used accuracy, precision, recall, and F1 score, which are widely used in other studies (e.g., [7] and [43]) to assess the performance of binary prediction models.

VIII. Related Work

Duplicate Bug Report Detection. Over the past decades, several approaches have been proposed to automatically detect duplicate bug reports [4], [8], [9], [44]–[49]. Among them, BMF25 [8] and REP [9] are two state-of-the-art algorithms. BM25F calculates the similarity between two bug reports based on common words shared between the bug reports. Additionally, REP extends BM25F by also considering bug report attribute information (e.g., product, priority). There are also many studies that take into account new techniques and more document features, such as using topic modeling with Latent Dirichlet Allocation (LDA) [50], incorporating software contextual features [51], combining word embedding technique [52], [53], considering execution information (e.g., stack traces) [54], and detecting duplicates just-in-time [55]. Deshmukh et al. [5] proposed a deep learning-based approach (i.e., DLDBR), which mainly relies on the textual feature to detect duplicate bug reports. Later, several deep learning-based approaches have been proposed, such as DBR-CNN [7], DWEN [56], and DC-CNN [43]. These approaches leverage text similarity between two bug reports to detect duplicates with word embeddings and DNNs.

With the advent of the JIT retrieval feature in modern BTSs (e.g., Bugzilla and JIRA), remaining duplicates in BTSs may become difficult to be detected with text similarity techniques [10]. Because many textually similar duplicates have already been excluded by the built-in JIT in BTSs. HINDBR differs from existing approaches in that it uses HIN as the fundamental representation for learning and predicting duplicate bug reports.

HIN in Software Engineering. Due to the advantages of storing rich information and representing diverse relations of objects, HIN and its related techniques have been adopted in many applications, such as recommendation [57] and anomaly detection [58]. The adoption of HIN in the software engineering field mainly relates to malware detection [59] and developer relation analysis [12].

Hou et al. [60] proposed an Android malware detection by representing relationships between Android malware as a structured HIN. A meta-path based approach is used to characterize the semantic relatedness of apps and their APIs. Each meta-path is used to measure the similarity over Android apps. The HIN based technique has also been adopted to detect insecure code snippets in Stack Overflow [61]. Moreover, HIN has also been used to investigate social relations among developers during software development, such as developers' collaboration and contributions in bug repositories [12], [13]. To the best of our knowledge, we are the first to explore the application of HIN for duplicate bug report prediction.

IX. Conclusion

This paper presents HINDBR, a deep representation learning approach for duplicate bug report prediction based on HIN. We construct a bug report HIN by the relations of attributes in bug reports. The latent semantic relations are revealed through a representation learning method. Bug reports are represented by embedding the learned structured and unstructured features into a low-dimensional vector space. Then, the Manhattan distance is used to calculate the similarity between the vectors of the two bug reports. We have used 5-fold cross-validation to evaluate our model with a total of 1,050,175 bug report pairs generated from nine popular open-source projects (i.e., Eclipse, Freedesktop, GCC, GNOME, KDE, LibreOffice, Linux kernel, LLVM, and OpenOffice), covering both before-JIT and after-JIT duplicates. Our results showed that HINDBR is effective in predicting duplicate bug reports in the post-JIT ear. Our source code and data are available at https://github.com/hindbr.

Although the performance of HINDBR has been improved compared with DLDBR, the results are valid for our limited experiments. Therefore, the results cannot be generalized to other dataset and other approaches for duplicate bug detection. In the future, we plan to investigate the performance of retrieval models built upon HINDBR using more recent real-world projects which have Bugzilla or JIRA, etc.

Acknowledgment

This work was supported in part by the Start-up Fund for New Faculty of NUAA under Grant YAH20026, the Open Research Fund of State Key Laboratory of Novel Software Technology under Grant KFKT2020B20, the National Natural Science Foundation of China under Grants 61772055, 61872169, and 61872182, the Technical Foundation Project of Ministry of Industry and Information Technology of China under Grant JSZL2016601B003, the Equipment Preliminary R&D Project of China under Grant 41402020102, and the Australian Research Council under Grant DP200101328.

REFERENCES

[1] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," in *Proc. ASE*. ACM, 2012, pp. 70–79.

[2] K. Aggarwal, T. Rutgers, F. Timbers, A. Hindle, R. Greiner, and E. Stroulia, "Detecting duplicate bug reports with software engineering domain knowledge," in *Proc. SANER*. IEEE, 2015, pp. 211–220.

[3] M. S. Rakha, W. Shang, and A. E. Hassan, "Studying the needed effort for identifying duplicate issues," *Empirical Software Engineering*, vol. 21, no. 5, pp. 1960–1989, 2016.

[4] S. Banerjee, Z. Syed, J. Helmick, M. Culp, K. Ryan, and B. Cukic, "Automated triaging of very large bug repositories," *Information and Software Technology*, vol. 89, pp. 1–13, 2017.

[5] J. Deshmukh, S. Podder, S. Sengupta, and N. Dubash, "Towards accurate duplicate bug retrieval using deep learning techniques," in *Proc. ICSME*. IEEE, 2017, pp. 115–124.

[6] M. S. Rakha, C.-P. Bezemer, and A. E. Hassan, "Revisiting the performance evaluation of automated approaches for the retrieval of duplicate issue reports," *IEEE Transactions on Software Engineering*, vol. 44, no. 12, pp. 1245–1268, 2017.

[7] Q. Xie, Z. Wen, J. Zhu, C. Gao, and Z. Zheng, "Detecting duplicate bug reports with convolutional neural networks," in *Proc. APSEC*. IEEE, 2018, pp. 416–425.

[8] S. Robertson, H. Zaragoza, and M. Taylor, "Simple bm25 extension to multiple weighted fields," in *Proc. CIKM*. ACM, 2004, pp. 42–49.

[9] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *Proc. ASE*. IEEE, 2011, pp. 253–262.

[10] M. S. Rakha, C.-P. Bezemer, and A. E. Hassan, "Revisiting the performance of automated approaches for the retrieval of duplicate reports in issue tracking systems that perform just-in-time duplicate retrieval," *Empirical Software Engineering*, vol. 23, no. 5, pp. 2597–2621, 2018.

[11] "Bugzilla 4.0 Release Notes," 2019. [Online]. Available: https://www.bugzilla.org/releases/4.0/release-notes.html#v40_feat_dup

[12] S. Wang, W. Zhang, Y. Yang, and Q. Wang, "Devnet: Exploring developer collaboration in heterogeneous networks of bug repositories," in *Proc. ESEM*. IEEE, 2013, pp. 193–202.

[13] W. Zhang, S. Wang, Y. Yang, and Q. Wang, "Heterogeneous network analysis of developer contribution in bug repositories," in *Proc. CSC*. IEEE, 2013, pp. 98–105.

[14] Y. Sun and J. Han, "Mining heterogeneous information networks: principles and methodologies," *Synthesis Lectures on Data Mining and Knowledge Discovery*, vol. 3, no. 2, pp. 1–159, 2012.

[15] A. Grover and J. Leskovec, "Node2vec: Scalable feature learning for networks," in *Proc. KDD*. ACM, 2016, pp. 855–864.

[16] M. Ji, J. Han, and M. Danilevsky, "Ranking-based classification of heterogeneous information networks," in *Proc. KDD*. ACM, 2011, pp. 1298–1306.

[17] T. Opsahl and P. Panzarasa, "Clustering in weighted networks," *Social Networks*, vol. 31, no. 2, pp. 155–163, 2009.

[18] D. Liben-Nowell and J. Kleinberg, "The link-prediction problem for social networks," *Journal of the American Society for Information Science and Technology*, vol. 58, no. 7, pp. 1019–1031, 2007.

[19] T.-y. Fu, W.-C. Lee, and Z. Lei, "Hin2vec: Explore meta-paths in heterogeneous information networks for representation learning," in *Proc. CIKM*, 2017, pp. 1797–1806.

[20] A. Lazar, S. Ritchey, and B. Sharif, "Generating duplicate bug datasets," in *Proc. MSR*. ACM, 2014, pp. 392–395.

[21] Z. Zheng and G. Xiao, "Evolution analysis of a uav real-time operating system from a network perspective," *Chinese Journal of Aeronautics*, vol. 32, no. 1, pp. 176–185, 2019.

[22] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "Line: Large-scale information network embedding," in *Proc. WWW*. WWW Committee, 2015, pp. 1067–1077.

[23] D. J. Montana and L. Davis, "Training feedforward neural networks using genetic algorithms." in *Proc. IJCAI*, vol. 89, 1989, pp. 762–767.

[24] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, "Recurrent neural network based language model," in *Proc. INTERSPEECH*. ISCA, 2010, pp. 1045–1048.

[25] H. Palangi, L. Deng, Y. Shen, J. Gao, X. He, J. Chen, X. Song, and R. Ward, "Deep sentence embedding using long short-term memory networks: Analysis and application to information retrieval," *IEEE/ACM Transactions on Audio, Speech and Language Processing*, vol. 24, no. 4, pp. 694–707, 2016.

[26] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[27] J. Mueller and A. Thyagarajan, "Siamese recurrent architectures for learning sentence similarity," in *Proc. AAAI*, 2016, pp. 2786–2792.

[28] K. P. Murphy, *Machine learning: A probabilistic perspective*. MIT press, 2012.

[29] "Eclipse Bugzilla," 2019. [Online]. Available: https://bugs.eclipse.org/bugs/show_bug.cgi?id=359299

[30] "GCC Bugzilla," 2019. [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=49935

[31] "Freedesktop Bugzilla," 2019. [Online]. Available: https://bugzilla.freedesktop.org/show_bug.cgi?id=30376

[32] "KDE Bugzilla," 2019. [Online]. Available: https://bugs.kde.org/show_bug.cgi?id=196285

[33] "OpenOffice Bugzilla," 2019. [Online]. Available: https://bz.apache.org/ooo/show_bug.cgi?id=117266

[34] "Linux Bugzilla," 2019. [Online]. Available: https://bugzilla.kernel.org/show_bug.cgi?id=30682

[35] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. NIPS*, 2014, pp. 3104–3112.

[36] A. Géron, *Hands-on machine learning with Scikit-Learn, keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.

[37] M. D. Zeiler, "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[38] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *Proc. ICML*, 2013, pp. 1310–1318.

[39] M. S. Santos, J. P. Soares, P. H. Abreu, H. Araujo, and J. A. C. Santos, "Cross-validation for imbalanced datasets: Avoiding overoptimistic and overfitting approaches," *IEEE Computational Intelligence Magazine*, vol. 13, no. 4, pp. 59–76, 2018.

[40] D. J. Sheskin, *Handbook of parametric and nonparametric statistical procedures*. crc Press, 2003.

[41] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008.

[42] X. Xia, D. Lo, M. Wen, E. Shihab, and B. Zhou, "An empirical study of bug report field reassignment," in *Proc. CSMR-WCRE*. IEEE, 2014, pp. 174–183.

[43] J. He, L. Xu, M. Yan, X. Xia, and Y. Lei, "Duplicate bug report detection using dual-channel convolutional neural networks," in *Proc. ICPC*, 2020.

[44] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *Proc. ICSE*. IEEE, 2007, pp. 499–510.

[45] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," in *Proc. DSN*. IEEE, 2008, pp. 52–61.

[46] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proc. ICSE*. ACM, 2008, pp. 461–470.

[47] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proc. ICSE*. ACM, 2010, pp. 45–54.

[48] A. Sureka and P. Jalote, "Detecting duplicate bug report using character n-gram-based features," in *Proc. APSEC*. IEEE, 2010, pp. 366–374.

[49] T. Prifti, S. Banerjee, and B. Cukic, "Detecting bug duplicate reports through local references," in *Proc. Promise*, 2011, pp. 1–9.

[50] J. Zou, L. Xu, M. Yang, X. Zhang, J. Zeng, and S. Hirokawa, "Automated duplicate bug report detection using multi-factor analysis," *IEICE Transactions on Information and Systems*, vol. 99, no. 7, pp. 1762–1775, 2016.

[51] A. Hindle, A. Alipour, and E. Stroulia, "A contextual approach towards more accurate duplicate bug report detection and ranking," *Empirical Software Engineering*, vol. 21, no. 2, pp. 368–410, 2016.

[52] X. Yang, D. Lo, X. Xia, L. Bao, and J. Sun, "Combining word embedding with information retrieval to recommend similar bug reports," in *Proc. ISSRE*. IEEE, 2016, pp. 127–137.

[53] A. Budhiraja, R. Reddy, and M. Shrivastava, "Poster: Lwe: Lda refined word embeddings for duplicate bug report detection," in *Proc. ICSE-Companion*. IEEE, 2018, pp. 165–166.

[54] N. Ebrahimi, A. Trabelsi, M. S. Islam, A. Hamou-Lhadj, and K. Khanmohammadi, "An hmm-based approach for automatic detection and

classification of duplicate bug reports," *Information and Software Technology*, vol. 113, pp. 98–109, 2019.
[55] A. Hindle and C. Onuczko, "Preventing duplicate bug reports by continuously querying bug reports," *Empirical Software Engineering*, vol. 24, no. 2, pp. 902–936, 2019.
[56] A. Budhiraja, K. Dutta, R. Reddy, and M. Shrivastava, "Dwen: Deep word embedding network for duplicate bug report detection in software repositories," in *Proc. ICSE-Companion*. ACM, 2018, pp. 193–194.
[57] H. Zhao, Q. Yao, J. Li, Y. Song, and D. L. Lee, "Meta-graph based recommendation fusion over heterogeneous information networks," in *Proc. KDD*. ACM, 2017, pp. 635–644.
[58] N. T. Tam, M. Weidlich, B. Zheng, H. Yin, N. Q. V. Hung, and B. Stantic, "From anomaly detection to rumour detection using data streams of social platforms," *Proc. VLDB Endowment*, vol. 12, no. 9, pp. 1016–1029, 2019.
[59] Y. Fan, S. Hou, Y. Zhang, Y. Ye, and M. Abdulhayoglu, "Gotcha-sly malware!: Scorpion a metagraph2vec based malware detection system," in *Proc. KDD*. ACM, 2018, pp. 253–262.
[60] S. Hou, Y. Ye, Y. Song, and M. Abdulhayoglu, "Make evasion harder: An intelligent android malware detection system," in *Proc. IJCAI*, 2018, pp. 5279–5283.
[61] Y. Ye, S. Hou, L. Chen, X. Li, L. Zhao, S. Xu, J. Wang, and Q. Xiong, "Icsd: An automatic system for insecure code snippet detection in stack overflow over heterogeneous information network," in *Proc. ACSAC*. ACM, 2018, pp. 542–552.