



Article

Knowledge Bases and Representation Learning Towards Bug Triaging

Qi Wang ^{1,2}, Weihao Yan ², Yanlong Li ^{3,4,*}, Yizheng Ge ², Yiwei Liu ² , Peng Yin ² and Shuai Tan ⁴¹ College of Civil Aviation, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China; qiwang_2024@nuaa.edu.cn² Defence Industry Secrecy Examination and Certification Center, Beijing 100001, China; ywh0015@163.com (W.Y.); vernier2021@163.com (Y.G.); yiweiliu_disecc@163.com (Y.L.); yinpeng@iie.ac.cn (P.Y.)³ School of Materials Science and Engineering, Harbin Institute of Technology, Harbin 150001, China⁴ Beijing Huahang Radio Measurement Research Institute, Beijing 100039, China; 15110147295@163.com

* Correspondence: liyanlong0709@163.com

Abstract

A large number of bug reports are submitted by users and developers in bug-tracking system every day. It is time-consuming for software maintainers to assign bug reports to appropriate developers for fixing manually. Many bug-triaging methods have been developed to automate this process. However, most previous studies mainly focused on analyzing textual content and failed to make full use of the structured information embedded in the bug-tracking system. In fact, this structured information, which plays an important role in bug triaging, reflects the process of bug tracking and the historical activities. To further improve the performance of automatic bug triaging, in this study, we propose a new representation learning model, PTITransE, for knowledge bases, which extends TransE via enhancing the embeddings with textual entity descriptions and is more suitable for bug triaging. Moreover, we make the first attempt to apply knowledge base and link prediction techniques to bug triaging. For each new bug report, the proposed framework can recommend top-*k* developers for fixing the bug report by using the learned embeddings of entities and relations. Evaluation is performed on three real-world projects, and the results indicate that our method outperforms baseline bug triaging approaches and can alleviate the *cold-start* problem in bug triaging.

Keywords: software maintenance; bug triaging; knowledge base; representation learning; link prediction

Academic Editor: Mehmed
Kantardzic

Received: 22 April 2025

Revised: 2 June 2025

Accepted: 13 June 2025

Published: 19 June 2025

Citation: Wang, Q.; Yan, W.; Li, Y.; Ge, Y.; Liu, Y.; Yin, P.; Tan, S. Knowledge Bases and Representation Learning Towards Bug Triaging. *Mach. Learn. Knowl. Extr.* **2025**, *7*, 57.
<https://doi.org/10.3390/make7020057>

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

With the increasing scale and complexity of software systems, bugs are inevitable in the lifecycle of software development. Many software projects adopt a bug-tracking system (e.g., Bugzilla [1], JIRA [2] and GNATS [3]) to help developers manage bugs occurring in the projects [4]. Due to the large number of bug reports submitted to the bug-tracking system every day, bug triaging, i.e., assigning bugs to appropriate developers for fixing, has been a task too time-consuming and labour-intensive to be practical.

In order to reduce the cost of the software maintenance and manual bug triaging, many automatic bug-triaging methods have been proposed [5–8]. Most of these methods are text-based, which treat bug reports as documents and represent them as feature vectors by using Vector Space Model (VSM) [9]. However, these methods treat words as atomic units that

cannot capture terms with the same meaning but different expressions or terms carrying multiple meanings, i.e., synonyms and polysemants. To mitigate this problem, topic model and deep-learning-based methods have been proposed to determine the semantic content of bug reports, which can improve the accuracy of bug triaging by representing bug reports more accurately [10–12]. Although these methods successfully achieved the desired goals, capturing the semantic and syntactic information contained in textual fields of bug reports, there is still room for further improvement. This is because text-based methods typically focus on analyzing the textual content of bug reports but ignore the effects of the bug reports' interaction with other objects, especially the developers. In fact, these interactions, called structured information in this work, provide valuable information such as the historical activities of developers and the process of bug tracking, which will help to improve the performance of bug triaging.

In addition to the above text-based methods, a few graph-based methods (e.g., DREX [4] and KSAP [13]) utilized the structured information to improve the performance of bug triaging. For example, KSAP proposed by Zhang et al. [13] used historical bug reports and heterogeneous network to implement automatic bug report assignment. However, these methods rank the developers' expertise on resolving new bug reports through simple network analysis [4,13], which fail to make full use of richer information available in the interactions between different entities in bug repository and have difficulty in integrating textual information into their network-based developer ranking methods. Hence, the improvement is also limited.

In this study, to overcome the above limitations, we present a new idea to efficiently solve the bug triaging task by introducing knowledge bases and representation learning technologies. In addition, aiming to make full use of both structured and textual information of bug reports, we develop a new representation learning model, PTITransE (combining *Partial entities' Textual Information with TransE*), for knowledge bases, which extends TransE [14] via enhancing the embeddings with textual entity descriptions and is more suitable for bug triaging. More concretely, we view the bug triaging process as three steps. First, we extract structured information from the bug repository and textual information from the *summary* and *description* fields of all bug reports to construct a bug triaging knowledge base. Next, we learn the embeddings of both entities and relations in the knowledge base by training our proposed representation learning model PTITransE. Finally, we recommend the top-*k* developers for fixing new bug reports by using link prediction based on the embeddings learned in the previous step. Our experimental results show that PTITransE improves the performance of bug triaging when compared with the state-of-the-art methods.

This paper makes the following contributions:

- An automatic bug-triaging method based on knowledge base and link prediction is developed. To the best of our knowledge, our study is the first one to apply a knowledge base and link prediction to bug triaging.
- Knowledge bases are constructed for three projects by extracting both structured and textual information from bug repository. Hopefully, the constructed knowledge bases can provide insight into solutions of other software engineering issues, e.g., duplicate bug report detection.
- A new model PTITransE is proposed, specializing for learning the representation of knowledge base with both structured information and partial entities' textual information. When it is applied to bug triaging, the model can make full use of the interactions between bug reports and other entities such as developers, components and products in the historical activities and bug-tracking process.

- The cold start issue on bug triaging, which is not well studied in the previous literature by leveraging the semantic feature of textual information learnt from the deep-learning-based model, is analyzed in this paper, and the results support that the proposed framework can mitigate the cold-start problem for new fixer recommendations.
- An extensive set of experiments using three real-world large-scale datasets are performed, and the experimental results show the effectiveness of the proposed method over the state-of-the-art methods by a significantly large margin.

The structure of this paper is organized as follows. Section 3 presents the background and motivation. Section 4 presents the overall framework of PTITransE, and Section 5 details our method. Section 6 presents the overview of the *cold-start* problem in bug triaging. Section 7 describes the experimental evaluation and results. Section 8 discusses the threats to validity of our results and gives several valuable suggestions to help bug triagers apply our method in practice. Finally, Section 2 introduces related work, while Section 9 concludes this paper and outlines our future work.

2. Related Work

2.1. Automated Bug Triaging

Text classification techniques are widely used in bug triaging. Cubranic et al. [6] first applied machine learning and text classification to predict fixer for a given bug report. Anvik et al. [5] tried various machine learning algorithms, including Naive Bayes, SVM and C4.5, to build the predictive models for solving this problem. Ahsan et al. [7] converted the clean textual fields of bug reports into a dimensional reduced term-to-document matrix based on TF-IDF and latent semantic indexing and then leveraged different machine learning methods for bug report classification. Xia et al. [15] used LDA and multi-label classification techniques to improve the accuracy of bug triaging. Xuan et al. [16] introduced the semi-supervised learning technique into bug triaging, which aims to solve the deficiency of labeled bug reports in supervised learning techniques. These classification-based methods treat bug reports as documents and represent them as feature vectors by using discrete word expression techniques (e.g., one-hot and TF-IDF). However, most of these methods cannot capture higher-level semantic information encoded in bug reports since words in their research are treated as atomic units. Therefore, the accuracy of recommending fixers could be adversely affected.

The rank-based methods are also proposed to find the most suitable fixers for new bug reports. Naguib et al. [17] created an activity profile of developers to rank and recommend the suitable developers for new bug reports. Tamrawi et al. [8] proposed a novel approach Bugzie based on fuzzy set and cache-based modeling of the bug-fixing capability of developers to improve the performance of bug triaging. Zhang et al. [18] provided a hybrid algorithm to rank all candidate developers by combining a probability model with an empirical model. Xie et al. [19] proposed a topic-based matching method called DRETOM to model developers' interest and expertise on bug resolving activities, and Xia et al. [10] proposed a new topic model called MTM to improve the performance of bug triaging by considering product and component information. In addition, some studies leveraged auxiliary information, such as code information [20–22], stack overflow [23] and commit information [24,25], to help address this problem. Most of these rank-based methods are performed by calculating the textual similarity among bug reports to analyze the developers' experience on fixing the historical bugs so that the most appropriate developers are recommended to fix the new bug reports. However, a few of these methods introduce the developers' interaction with other entities into the bug triaging process. Hence, the accuracy of bug triaging can be further improved by incorporating the structured information.

There have been a number of studies that use network analysis to rank and recommend developers for fixing new bug reports. Jeong et al. [26] introduced a tossing graph model based on the Markov property, which can help to better assign developers to bug reports. Wu et al. [4] constructed a homogeneous developer network based on the comment relationship between developers and then ranked candidate developers according to various metrics. Zhang et al. [13] adopted a similar approach, but the difference is that they constructed a heterogeneous network based on the structured information extracted from bug reports. These methods concentrate on utilizing developers' interaction to match developers for fixing new bug reports by means of simple network analysis. However, such methods typically cannot fully incorporate textual information of bug reports into their network based developer ranking methods due to the limitation of their used techniques.

In recent years, deep-learning-based approaches have been emerging to extract the semantic features of textual fields of bug reports to improve the performance of bug triaging. For example, both Lee et al. [12] and Mani et al. [11] took textual fields of bug reports as inputs of the deep learning model and output the probability of assigning bug reports to each developer. The difference is that Lee et al. employed the CNN network to extract textual semantic features of bug reports, while Mani et al. adopted bi-directional LSTM with attention mechanism. For the bug-triaging task, deep-learning-based methods often beat the common text classification methods due to the fact that they can capture the high-level latent semantic information of textual content in bug reports.

Unlike previous related works, both the structured and textual information are fully utilized in our proposed method by jointly learning latent representations of these information. The extensive evaluation in this study also shows the proposed work can effectively improve the performance of bug triaging.

2.2. Knowledge Bases for the Software Engineering Domain

Increasingly more researchers are showing interest in applying knowledge base techniques to tackle software engineering tasks, including source code representation, bug localization and vulnerability detection. For example, studies in [27,28] proposed to use knowledge graphs to represent code context and learn context embedding based on code knowledge graphs. Since both aims are to capture deep semantics of source code via knowledge graphs, it can be applied to different but related tasks, i.e., bug localization [27] and issue-commit link recovery [28]. In addition, researchers have also leveraged large amounts of knowledge from different software resources (e.g., Github, Bugzilla, StackOverflow and Wikipedia) to assist specific software engineering tasks. More concretely, Liu et al. constructed an *API knowledge graph* (API KG) for JDK and Android, which takes code-related information from external sources (e.g., Wikipedia) into account to create programming task-specific summaries [29]. Zhou et al. [30] proposed a bug knowledge graph construction framework, which aims to better integrate bug knowledge from multi-source software data (e.g., Github, Bugzilla and StackOverflow) and then intends to provide the possibility of intelligent bug fixing through precise classification of bugs and accurate recommendation for bug-fixing knowledge question and answering. Lin et al. [31] details the concepts and the process to construct the knowledge graphs of specific-domain software projects, which can be leveraged to achieve intelligent assistance for software development.

In essence, our work differs from these previous studies in that our effort is the first attempt to use knowledge base techniques for automated bug triaging, and the extensive experimental results also demonstrate its effectiveness in coping with large-scale open-source projects.

3. Background and Motivation

Before explaining our approach, we first describe the motivation example and fundamental concepts about knowledge base and representation learning. Then, as our innovation is to combine the partial entities' textual information with TransE for bug triaging, we also describe the techniques for text representations.

3.1. Motivation Example

Table 1 shows two bug reports of Eclipse. Both of them belong to the same project "JDT" and the same component "Debug" and are fixed by the same developer "Joe Szurszewski". The first bug report, i.e., Bug report #6447, describes a break point error: when a user debugs code, the inner class breakpoints are not triggered by Debug component, while the second bug report, i.e., Bug report #7004, describes a feature-interaction bug [32] that arises as a result of feature interactions: removed specified step filters when a user deselects all buttons on preference page.

Observations and Implications. From the two bug reports shown in Table 1, we have the following observations:

Table 1. Two bug reports of eclipse.

Field	Bug Report 1 [33]	Bug Report 2 [34]
Bug_id	6447	7004
Summary	Inner class breakpoints not hit	DCR: Deselect All button on step filter preference page
Description	From the newsgroup: I just tried it on 1127, and it wouldn't break for me. I tested the following code:...	It would be nice to have a deselect all button that would not remove specified step filters, but would deselect all of them
Reporter	Darin Swanson	Darin Swanson
Product	JDT	JDT
Component	Debug	Debug
Bug severity	Critical	Normal
Fixer	Joe Szurszewski	Joe Szurszewski

(1) The textual descriptions (i.e., the terms used in the summary and description fields) of these two bug reports are different as there are almost no shared terms that appear in both of them.

(2) Semantics of the two bug reports are entirely different since, according to their description, they are used to triage bugs of different types and severity.

(3) Both of the two bug examples are fixed successfully by the developer "Joe Szurszewski", who seems to have the expertise to fix many different types of bugs.

(4) The two bug reports belong to the same product "JDT" or even the same component "Debug", and both of them are fixed by the same developer "Joe Szurszewski", which indicates that developers have high affinity towards the product–component combination. This finding is in line with the previous work [10].

(5) "Darin Swanson" and "Joe Szurszewski" may be in the same tight knit community of developers, which contains a set of potential links between different developers and thus is useful for analyzing developers' behavior.

The above observations tell us that although each bug report has its own fields, different bug reports may share some commonalities, e.g., both of the two bug examples are in the same product JDT, or even the same component Debug, and the bug reporters as well as the bug resolvers are also the same. These commonalities could help to find the most

suitable developers to assign the given bug reports to, while they are not fully exploited by previous work [10–12]. In addition, most of the previous methods are based on the observing that a developer can resolve the bug reports with similar textual description since the bug reports with high textual similarity often correspond to the same type of bugs [10–13]. However, these methods mostly ignore the fact that many developers have the expertise to fix many different types of bugs, which results in the bug reports, fixed by the same developers, often having totally different textual descriptions. The reason behind this phenomenon can be explained from two perspectives: 1. Different types of bugs often exhibit distinct characteristics and manifestations, which leads to varied textual descriptions in bug reports. 2. This occurs because experienced developers typically work on multiple areas of a project or assume diverse roles, causing them to fix bugs that differ greatly in context and terminology. As a result, even though the same developer addresses these bugs, the descriptions in the reports can be quite different. Therefore, it is not appropriate to only consider the textual information of bug reports for bug triaging due to narrowing the search of possible candidates when recommending developers for new bug reports. To alleviate this limitation, the proposed approach considers both the candidate fixers' behaviors in the community of developers and their relations with other entities (e.g., bug, product, component and the other developers) to make good use of the mentioned commonalities via knowledge base and knowledge base embedding techniques. More than that, the proposed approach also benefits a lot from the combination of the textual information of bug reports.

3.2. Knowledge Bases

Knowledge bases (KBs) such as Freebase [35], Dbpedia [36], YAGO [37] and WordNet [38] have been widely used in information retrieval (IR) and question answering (Q&A) as they provide abundant structured information [14]. A typical knowledge base consists of millions of entity–relation–entity triplets ($h-r-t$), which can be represented as a labeled, directed heterogeneous graph, in which each entity h is a node, and each learning-dependency relation $R(h, t)$ is an edge labeled r between entity h and t [39]. The entities in the knowledge base can have different types, and the relations represent the facts involving two entities. A bug-tracking system contains a large amount of structured information (e.g., interactions between bug reports and developers) that can be organized in the form of knowledge base. Such structured information is important for improving the performance of bug triaging. This is the motivation that we apply knowledge base to bug triaging.

3.3. Knowledge Base Embedding

With the increasing scale of knowledge base, the sparsity and computational inefficiency become the main obstacles to its application in practice [40]. In this situation, representation learning (RL) methods are emerging to address the limitations by projecting both entities and relations into a continuous low-dimensional semantic space, which have been successfully applied in a wide range of applications, e.g., knowledge completion, acquisition and inference [40–42]. Among these methods, translation-based knowledge base embedding techniques, such as TransE [14], TransH [43], TransR [44] and TransD [45], are becoming more widely used due to their very simple structures and efficient implementations with the state-of-the-art performance. TransE is one of the most successful and efficient models [46]; however, it fails to deal well with reflexive/one-to-many/many-to-one/many-to-many relations [43]. Thus several modifications (e.g., TransH, TransR and TransD) to TransE have been proposed to address such challenges. To help understand the differences among existing works, we briefly summarize these methods as follows:

- TransE [14] learns the embeddings of h, r, t that satisfy the constraint $\mathbf{h} + \mathbf{r} \approx \mathbf{t}$ when (h, r, t) holds. The scoring function of TransE to measure the plausibility of the triplet is defined as

$$f_r(\mathbf{h}, \mathbf{t}) = \|\mathbf{h} + \mathbf{r} - \mathbf{t}\| \quad (1)$$

where $\mathbf{h}, \mathbf{r}$ and $\mathbf{t}$ are the embeddings of h, r and t , respectively.

- TransH [43] projects the embeddings of entities onto a hyperplane determined by relation r before translation, which overcomes the flaws of TransE in dealing with reflexive/one-to-many/many-to-one/many-to-many relations [43]. The scoring function is

$$f_r(\mathbf{h}, \mathbf{t}) = \|\mathbf{h}_\perp + \mathbf{r} - \mathbf{t}_\perp\| \quad (2)$$

where $\mathbf{h}_\perp$ and $\mathbf{t}_\perp$ are the *projections* of $\mathbf{h}$ and $\mathbf{r}$ on the hyperplane, which are expected to be connected by the translation vector of $\mathbf{r}$ with low error.

- TransR [44] projects the embeddings of entities from entity space to corresponding relation space and then builds translations between projected embeddings. The difference with TransH is the projection operator is a matrix that is more general than an orthogonal projection to a hyperplane [46]. The scoring function is

$$f_r(\mathbf{h}, \mathbf{t}) = \|\mathbf{h}_r + \mathbf{r} - \mathbf{t}_r\| \quad (3)$$

where $\mathbf{h}_r$ and $\mathbf{t}_r$ are projections of $\mathbf{h}$ and $\mathbf{r}$ by multiplying a projection matrix.

- TransD [45] improves the projection operation of TransR by replacing the projection matrix with the inner product of two distinct vectors of entity–relation pairs.

In addition, to our knowledge, representation learning has not been applied to bug triaging yet. In this paper, we aim to develop a novel representation learning approach for extracting interaction features among developers to help improve the performance of bug triaging.

3.4. Link Prediction

Link prediction is a common technique in knowledge base completion, which attempts to estimate the probability of the existence of a link between entities [47,48]. Formally, performing link prediction can be viewed as a task of predicting the unknown entity in the incomplete triplet like $(head, relation, ?)$ or $(?, relation, tail)$ when such a triplet does not exist in KBs yet [46]. For each missing entity, the system aims to predict the correct target entity by ranking all entities from the knowledge graph [40]. This prediction is made based on the existing knowledge base extracted from an associated databases [39].

The bug triaging is the process of assigning the unfixed bug reports to appropriate developers for bug fixing. This task is essentially to estimate how good a match a developer is for an unfixed bug report. Therefore, this task can be formalized as a special link prediction task where the *developer* in the unseen triplet $(developer, fix, bug)$ is the target entity to be predicted.

3.5. Continuous Bag-of-Words Encoder

Distributed representation technology (e.g., word2vec [49]) projects words into a low-dimensional semantic space based on the hypothesis that the words with similar meaning tend to appear in similar context [50]. Performing word embedding has been proven to be a useful and efficient way to improve the performance of various natural language processing (NLP) tasks. In addition to the distributed representation of words, the distributed representation learning models for documents (e.g., doc2vec [51]) are also widely studied in NLP. Each word in distributed models (e.g., word2vec) is represented as a dimension-fixed dense vector, but the number of words varies from one document

to another. A common solution to represent documents as fixed-length dense vectors is to employ a continuous bag-of-words encoder (CBOW) model. In CBOW, a document is firstly represented as a matrix, where each row represents a word and the number of columns equals the dimension of word vector. Then, the document representation can be generated simply by averaging all the word vectors contained in the document matrix.

4. The Overall Framework

The overall framework of our proposed bug triaging approach is shown in Figure 1, which consists of five phases: information extraction, knowledge base construction, knowledge base embedding, joint representation learning and fixer recommendation. In this section, we present the overview of all the steps. The key components in PTITransE will be depicted in Section 5.

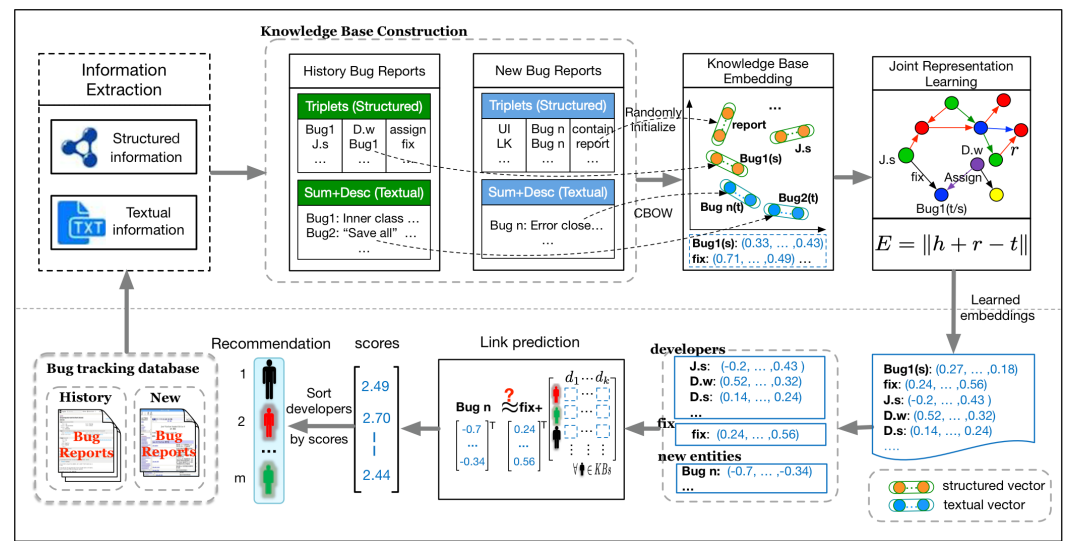


Figure 1. Illustration of our proposed framework, which consists of five phases: information extraction, knowledge base construction, knowledge base embedding, joint representation learning and fixer recommendation using link prediction.

4.1. Information Extraction and Knowledge Base Construction

In this study, we divide the information involved in bug reports into two parts, textual and structured information, which we use to construct the knowledge base for bug triaging. To obtain the structured information, we first parse each bug report to extract the entities and relations within it and then use these entities and relations to construct triplets in the form of (*head entity*, *relation* and *tail entity*) based on the fact involving them. For textual information, we only consider the brief and detailed description of bug reports, i.e., *summary* and *description*, following the previous research studies [10–12,20]. Then, we extract them by the aid of the field tags labeled by the reporters and store them as text [52].

As shown in Figure 1, for a project, we extract the information from historical bug reports and then use them to construct the knowledge base; concurrently, we extract information from new bug reports and store them in the knowledge base constructed from historical bug reports. Note that the information extracted from historical and new bug reports is different since the new bug reports only contain partial information, e.g., *summary*, *description*, *product* and *component*, and do not have *tossed* and *bug-fix* information. Our purpose here is using the partial information to generate the embeddings of new bug entities, which are used for the prediction phase. The details of the specifics of differences will be introduced in Section 5.1.

4.2. Knowledge Base Embedding

After constructing the knowledge base, we initially project the embeddings of both entities and relations onto a distribution space to make them have the same dimensions as the embedding of textual content information. This is designed on purpose to conduct our joint representation learning based on structured information together with textual information. Note that only the bug entities have textual information. The textual embedding of each bug entity is generated via the CBOW model using word embedding that has been trained offline on a large-scale text corpus. The structured embeddings of the entities and their relations are randomly initialized with uniform distribution and then trained via the representation learning model in the next step.

4.3. Joint Representation Learning

To fully utilize both fact triples and bug entity descriptions, we propose combining the textual and structured information to train a more effective representation learning model (i.e., PTITransE) for learning low-dimensional embeddings of entities. Since PTITransE is an improvement over TransE and both are energy-based models, relationships are regarded as *translations between entities in the embedding space*, which can be read as meaning that if the triplet (h, r, t) holds (is true), then the embedding of tail entity t should be close to the embedding of the head entity h plus some vector that depends on the relationship r [14]. Based on this property, we can efficiently assign the unfixed bug reports to the appropriate developers, which is presented in the following step.

4.4. Fixer Recommendation

The goal of this phase is to predict developers for fixing the new bug reports by using the learned embeddings of all entities and relations. Concretely, we first extract all learned embeddings of developers and “fix” relation as well as the new bug entities that need to be assigned. Then, the bug entity (denoted as *bug*) and the “fix” relation (denoted as *fix*) together with the unknown fixer (denoted as *developer*) form an unseen triplet (*developer*, *fix*, *bug*) that can be completed by using link prediction to predict the target entity *developer*. More specifically, under the known embeddings of *bug* and *fix*, the objective of link prediction is to rank all *developer* entities by calculating the matching-degree between “*developer* + *fix*” and “*bug*”, where the bold face (e.g., *developer*, *fix* and *bug*) denotes the corresponding vector embeddings of *developer*, *fix* and *bug*. Finally, the list of recommendations is returned to the system.

5. Our Approach

In this section, we give a detailed description of the following key issues, i.e., knowledge base construction, knowledge base embedding, joint representation learning and link prediction.

5.1. Knowledge Base Construction

The textual information can be easily extracted from bug reports since the “summary” and “description” are manually tagged and submitted by the reporters. Therefore, we can first find the corresponding field tags (i.e., “summary” and “description”) in each bug report and then extract the textual content behind these tags. For the structured information of bug reports, the extraction process is more complex and requires more work than textual information since the structured information needs to be excavated from the bug repository and reorganized in the form of (h, r, t) triplets to facilitate training representation learning model. We take the Eclipse bug report #6447 as a historical bug report to illustrate the process of extracting structured information from historical bug

reports. Firstly, we extract the entities by parsing this bug report and only keep five types of entities (i.e., developer, bug, comment, component and product) that are relevant and conducive to bug triaging. For example, for Eclipse bug report #6447, developer entities include “Darin Wright”, “Darin Swanson” and “Joe Szurszewski”. The component and product entities are “Debug” and “JDT”, respectively. The bug entity is “Bug 6447”, and its comments are marked as “comment 1, 2” and “comment 3”. Note that “comment 1, 2” is a merged symbol representing “comment 1” and “comment 2”, which are both commented by the same developer (i.e., “Darin Swanson”). Other types of entities such as “OS” and “version” are discarded following previous studies [4,10–13,15,19,20,53] because they may rarely contribute to the bug triaging task. We leave investigations of these other entities as future work. Next, we extract the relations between these entities. The “assign to|fix|toss” relations can be extracted by analyzing the activity log of bug report #6447 as shown in Table 2, while the others can be easily inferred according to the fields of bug reports (e.g., the product “JDT” contains the component “Debug”). In the example above, the bug was reported by “Darin Swanson” and first assigned to “Darin Wright” to fix. However, for various reasons, this bug was not fixed properly by “Darin Wright” and then tossed to “Darin Swanson” and “Joe Szurszewski” successively. The seven types of relations as shown in Table 3 are denoted as “report”, “assign to”, “fix”, “write”, “comment”, “contain” and “toss”, respectively.

After extracting the entities and relations from Eclipse bug report #6447, we can combine all the (h, r, t) triplets together to form a knowledge graph (see Figure 2); where the “Class” corresponds to the generic concept of a category related to bug triaging, the “Instance” is an entity or instance of the corresponding class, the predefined relation “type” is used as a predicate in “*Ins type Cla*” to declare that individual *Ins* is an instance of class *Cla*, “Relation” is the fact between *head* and *tail* in triplets and “Data property” indicates the textual information (i.e., *summary* and *description*) of the bug entity is “*Inner class...*”.

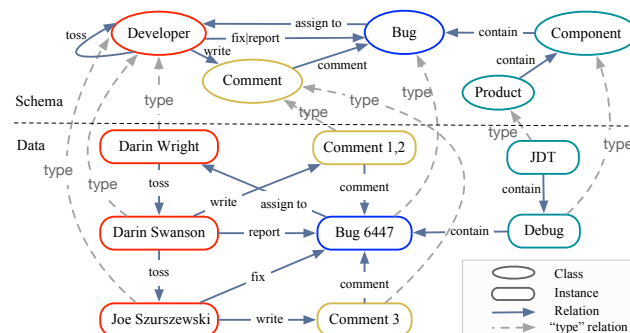


Figure 2. The construction process of knowledge graph taking Eclipse bug report #6447 as a historical bug report.

For a new bug report, we also extract all textual information (e.g., summary and description) and triplets with “report” and “contain” relations from the predefined meta-fields of new bug reports into the knowledge base constructed by historical bug reports. Therefore, if Eclipse bug report #6447 is a new bug report, the knowledge graph of it can be constructed as shown in Figure 3, where the same notation as in Figure 2 are employed for the entities and relations.

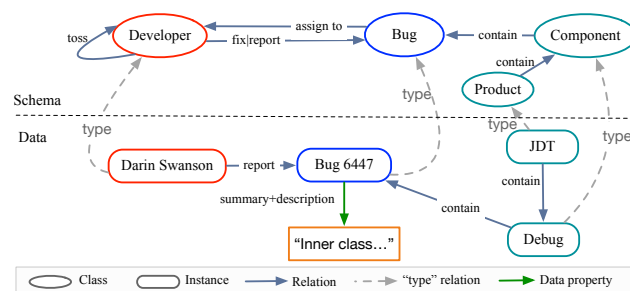


Figure 3. The construction process of knowledge graph taking Eclipse bug report #6447 as a new bug report.

Table 2. The activity log of bug report #6447.

Who	When	What	Removed	Added
darin eclipse	2001/11/29 18:24:32	Severity Priority	normal P3	critical P1
pilf_b	2001/11/29 18:24:32	CC		pilf_b
Darin Swanson	2001/11/30 12:31:36	Assignee	Darin Wright	Darin Swanson
Darin Swanson	2001/11/30 12:31:44	Status	New	Assign
Darin Swanson	2001/11/30 15:58:13	Assignee	Darin Swanson	Joe Szurszewski

Table 3. The seven types of relations in heterogeneous network.

Types of Relation	Description	Cardinality
report	a developer can find more than one (n) bugs and give reports	1:n
assign to	a bug is assigned to a developer	1:1
fix	a developer fixes a bug	1:1
write	a developer can write more than one (n) comments	1:n
comment	one or more comments are written to comment a bug	n:1
contain	a component can contain one or more bugs a product can contain one or more components	1:n 1:n
toss	a developer tosses a bug to another developer	1:1

5.2. Knowledge Base Embedding

As described in Section 4.2, to conduct our proposed joint representation learning procedure, the structured information and the textual information will be represented as the same dimensional vectors. The textual information of each bug report consists of two main components: *summary* and *description*. *summary* is a brief description of the bug report, while *description* provides more details. Thus, *description* generally has a longer sequence of text than *summary*. To obtain the textual embedding, we merge the *summary* and *description* to obtain the textual content of bug reports and then use the CBOW model detailed in Section 3.5 to transform the textual content into a low-dimensional dense

vector. The specific process for the transformation is as follows: Firstly, we preprocess the textual content with the general natural language processing techniques, including sentence splitting, word tokenization, stop-word removal and stemming. Secondly, we use the word2vec model to transform the preprocessed words into low-dimensional dense vectors. Finally, we average the word vectors of all terms contained in the textual content to generate the textual embedding of each bug report. The structured embeddings of the entities and their relations are randomly initialized with uniform distribution [54] and later will be trained in the next step.

5.3. Joint Representation Learning

In this step, we give detailed descriptions on how to learn the embeddings of each entity and relation. First, we briefly review the most representative translation-based model TransE [40] and then give an extension of the improved TransE model for the common knowledge base to the case of the bug triaging that is involved with the textual embedding and structured embedding.

TransE is an energy-based model for learning low-dimensional embeddings of entities [14] that regard the relation r in triplet (h, r, t) as a translation from head entity h to tail entity t . The training goal of TransE is to let the sum of the embeddings of h and r be as close as possible to the embedding of t . These vector embeddings are set in $\mathbb{R}^k$, and we use the same letter in boldface (e.g., $\mathbf{h}, \mathbf{r}, \mathbf{t}$) to denote the corresponding vector embeddings of h, r, t [44]. The energy function of TransE is defined as

$$E = \|\mathbf{h} + \mathbf{r} - \mathbf{t}\| \quad (4)$$

Here, the notation $\|\cdot\|$ denotes ℓ_1 or ℓ_2 norm, measuring how close $\mathbf{h} + \mathbf{r}$ is close to $\mathbf{t}$. By minimizing $E(\mathbf{h}, \mathbf{r}, \mathbf{t})$ for valid triples and maximizing it for negative samples, TransE enforces that true relations correspond to low-energy translations. As shown in the energy function, only the fact triplets are utilized to learn the embeddings of entities and relations. Recent studies have provided evidence suggesting that enhancing the embeddings with entity descriptions can improve the performance of the TransE model [55,56]. However, for bug triaging, not all entities in knowledge base have the textual information. In this study, only bug entities' textual information is considered together with their structured information. Therefore, the issue is how to integrate the textual information of partial entities into the structured information of all entities. To solve this issue, we propose a new method to train a more effective representation learning model for bug triaging. First, we redefine the energy function as follows:

$$E = E_s + E_{pt} \quad (5)$$

$$E_s = \|\mathbf{h} + \mathbf{r} - \mathbf{t}\| \quad (6)$$

$$E_{pt} = \lambda \cdot [\delta_h \cdot \|\text{Align}(\mathbf{h}_t) + \mathbf{r} - \mathbf{t}\| + \delta_t \cdot \|\mathbf{h} + \mathbf{r} - \text{Align}(\mathbf{t}_t)\|] \quad (7)$$

$$\text{Align}(\mathbf{v}_t) = \mathbf{W} \cdot \mathbf{v}_t + \mathbf{b} \quad (8)$$

where E_s denotes the energy function based on structured information, and E_{pt} is a variant of E_s obtained by replacing the structured vector h or t in E_s with the corresponding textual vector. The parameter λ is a trade-off hyperparameter that balances the structural and textual losses. The indicator function $\mathbb{I}(\cdot)$ returns 1 if the condition inside holds, and 0 otherwise. Based on this, $\delta_h = \mathbb{I}(h \in \mathcal{B})$ and $\delta_t = \mathbb{I}(t \in \mathcal{B})$ indicate whether the head or tail entity belongs to the set $\mathcal{B}$ of bug entities. Here, $\text{Align}(\cdot)$ denotes a linear projection aligning textual embeddings to the structural space; $\mathbf{W} \in \mathbb{R}^{k \times k}$ is a trainable projection matrix; and $\mathbf{b} \in \mathbb{R}^k$ is a trainable bias vector. For ease of notation, we omit the subscripts of structured

vector, assuming that all variables without any subscripts (e.g., h , t) are structured vectors and all textual vectors (e.g., h_t and t_t) have subscript t . Note that E_{pt} is not always available because not all the triplets but only a subset of them have the *bug entities* as the *head* or *tail* of triplets. This means we could not perform the element-wise additions of E_s and E_{pt} in a batch sample of triplets unless the corresponding triplets contain at least one *bug* entity. This will be detailed in our next algorithm (i.e., Algorithm 1).

Second, to further enhance the discriminatory power of representation learning model, we use the following margin-based ranking function as the optimization object function for minimization:

$$\mathcal{L} = \sum_{(h,r,t) \in T} \sum_{(h',r,t') \in T'} \max(\gamma + d(\mathbf{h} + \mathbf{r}, \mathbf{t}) - d(\mathbf{h}' + \mathbf{r}, \mathbf{t}'), 0) \quad (9)$$

where (h, r, t) is a positive sample, (h', r, t') is a negative sample constructed by randomly replacing the head, tail or relation in a correct triplet with another entity or relation, and T and T' are the sets of correct and corrupted triplets, respectively. d is the dissimilarity function between $\mathbf{h} + \mathbf{r}$ and $\mathbf{t}$, which generally uses ℓ_1 or ℓ_2 norm. γ is a margin hyperparameter.

Finally, the optimization process of PTITransE is elaborated in Algorithm 1, where E and R stand for the embedding sets of entities and relations, respectively, and X stands for the set of textual embedding vectors of *bug* entities. All embeddings (i.e., X , E and R) are obtained from the step of knowledge base embedding detailed in Section 5.2. For each loop, we first normalize the structured embeddings of each entity and each relation (Lines 3–4) and then sample a minibatch of size b from the training set S as the positive sample set S_{batch} (Line 5). For each triplet (h, r, t) in S_{batch} , we replace either h or t with a random entity to generate a corrupted triplet as a negative sample (Line 8) and then calculate the margin-based loss using the structured embeddings of entities and relations as the structured loss (Line 9). In addition, if the positive or corrupted triplet contains at least one *bug* entity, we obtain the textual embeddings of *bug* entities from set X (Line 13) and then calculate the textual loss (Line 16) as a supplement of the structured loss. In the end of each loop, the parameters are updated with stochastic gradient descent algorithm (Line 18). After model training, the embeddings of entities and relations are obtained.

5.4. Link Prediction

As detailed in Section 4.4, we use link prediction to complete the unseen triplet $(?, fix, bug)$ by predicting the target entity (*developer*). To interpret the process of predicting intuitively, we take Figure 4 as an example in which we assume there are three developers in KBs, including J.s, D.w and D.s. The embeddings of all developers, as well as test *bug* entity and *fix* relation, have been represented as distributional vectors in $\mathbb{R}^3$ through joint representation learning. We first replace the target entity with all *developer* entities and then calculate the vector $\mathbf{s}$ according to the scoring function $\mathbf{s} = \mathbf{t} - \mathbf{h} - \mathbf{r}$. After that, we add up the absolute values of all elements in vector $\mathbf{s}$ to obtain a score for each *head* entity so as to return a ranked list based on these scores sorted in ascending order. Finally, the bug triager can assign the most suitable developers to fix the new bug reports by the aid of the returned ranked list.

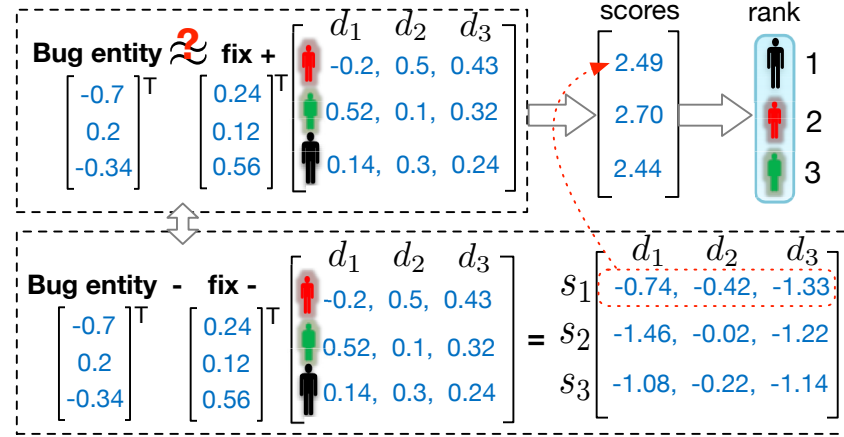


Figure 4. The prediction process of target entity *developer* using link prediction.

Algorithm 1: PTITransE

Input: Triplets set $S = \{(h, r, t)\}$, textual vector set X , entities vector set E , relations vector set R , margin γ , embeddings dim k , training times n , bug entities set $entity_{bug}$

Output: The learned embedding sets E, R .

```

1 Set  $N = 0$ ;
2 while  $N < n$  do
3    $\mathbf{e} \leftarrow \mathbf{e} / \|\mathbf{e}\|$  for each entity  $\mathbf{e} \in E$ ;
4    $\mathbf{r} \leftarrow \mathbf{r} / \|\mathbf{r}\|$  for each relation  $\mathbf{r} \in R$ ;
5    $S_{batch} \leftarrow sample(S, b)$  // sample a minibatch of size  $b$ ;
6    $\mathcal{L} = 0$ ;
7   for  $(h, r, t) \in S_{batch}$  do
8      $(h', r, t') \leftarrow sample(S'_{(h,r,t)})$  // sample a corrupted triplet;
9      $l = \max(\gamma + d(\mathbf{h} + \mathbf{r}, \mathbf{t}) - d(\mathbf{h}' + \mathbf{r}, \mathbf{t}'), 0)$ ;
10    Flag = False;
11    for  $e \in \{h, t, h', t'\}$  do
12      if  $e \in entity_{bug}$  then
13         $\mathbf{e} \leftarrow GetVector(X, e)$  // get the textual embedding of  $e$ ;
14        Flag = True;
15    if Flag then
16       $l += \max(\gamma + d(\mathbf{h} + \mathbf{r}, \mathbf{t}) - d(\mathbf{h}' + \mathbf{r}, \mathbf{t}'), 0)$ 
17     $\mathcal{L} += l$ 
18  Update embedding w.r.t  $\mathcal{L}$ ;
19   $N = N + 1$ ;

```

5.5. Bug Triaging Details Using PTITransE

In order to make clearer how to effectively use PTITransE for bug triaging, we provide a detailed procedure of bug triaging in the following subsections. The algorithm is sketched in Algorithm 2. The entire process contains two phases: offline computation, which accounts for the embedding learning of entities and relations in historical bug reports, and online bug triaging, with a low complexity to learn the embeddings of new bug entities and produce fixer recommendation list using link prediction.

As seen in Figure 1, triplets and textual content are extracted from both the new and historical bug reports together and then taken as input for PTITransE to learn their

embeddings. However, the learning process for all entities and relations from scratch would be extremely expensive and time-consuming because a large number of training epochs are usually needed to tune free parameters in the representation model until an expected performance has been met. To alleviate this problem, we first conduct an offline **Pretrain** task using PTITransE to learn the embedding for each entity and relation extracted from historical bug reports (line 1). Then, we only learn the embeddings of new bug entities from scratch and fine-tuning the existing ones that have been learned in pre-training phase (line 3–11). Finally, we use the *link prediction* to score all developers for each new bug report (line 12–13), and the top K developers who are most likely to fix the new bug reports are returned as the bug triaging results (line 16).

More concretely for the embedding learning (line 3–11) in Algorithm 2, we set the variable $HTrainSet$ to historical bug reports (HBRs) for the next processing (line 2) and through the outermost loop iterates over each batch of new bug reports in turn (line 3). For each loop, we first extract the structured ($Triplets_{Train}$) and textual ($Text_{Train}$) information from $TrainSet$, which is the union of $HTrainSet$ and N_{batch} (line 5–6). Then, for each triplet (h, r, t) in $Triplets_{Train}$, we obtain their embeddings (i.e., the corresponding vectors of h , r and t) from E_{pre} and R_{pre} if h , r and t are in E_{pre} or R_{pre} , which are learned from historical bug reports. Otherwise, the returned embedding are randomly initialized (line 7–9). For textual information $Text_{Train}$, we use CBOW to represent them as fixed-length vectors (line 10). Finally, based on the initializations, PTITransE (Algorithm 1) is applied to learn or fine-tune the embedding for each entity and relation in $TrainSet$ (line 11). Note that most of the embeddings in E^* and R^* are finetuned for a small number of epochs so that the time cost in this phase is much smaller than that of the offline pre-training.

Algorithm 2: Bug Triaging Process

Input: New bug reports NBRs, historical bug reports HBRs
Output: The fixer recommendation list for each new bug report

```

1  $E_{pre}, R_{pre} \leftarrow \text{PreTrain}(HBRs)$  { Return the trained embeddings of entities and
   relations extracted from HBRs};
2  $HTrainSet = HBRs$ ;
3 while  $N_{batch}$  in NBRs do
4   {  $N_{batch}$  represents a minibatch of size  $b$  extracted from NBRs as ordering by
     submitting time } ;
5    $TrainSet = HTrainSet + N_{batch}$ ;
6    $Triplets_{Train}, Text_{Train} \leftarrow \text{ExtractInfo}(TrainSet)$  { Return structured and textual
     information};
7   for  $(h, r, t) \in Triplets_{Train}$  do
8      $E^* \leftarrow \text{GetVector}(E_{pre}, h, t)$  { Return embeddings of  $h$  and  $t$  if present in  $E_{pre}$ 
       and random initialization if not present};
9      $R^* \leftarrow \text{GetVector}(R_{pre}, r)$  { Return embeddings of  $r$  if present in  $R_{pre}$  and
       random initialization if not present};
10   $T^* = \text{CBOW}(Text_{Train})$ 
11   $E^*, R^* \leftarrow \text{PTITransE}(Triplets_{Train}, T^*, E^*, R^*, \dots)$  { Return the trained
     embeddings using PTITransE};
12  for  $br \in N_{batch}$  do
13     $Fixers(br) = \text{linkPrediction}(E^*, R^*, br)$ 
14   $E_{pre}, R_{pre} \leftarrow E^*, R^*$ ;
15   $HTrainSet = TrainSet$  ;
16 return  $\text{Hashmap}(Fixers(br))$  {Returns dictionary of  $br$  mapped to  $Fixers(br)$ }
  
```

6. Cold-Start in Bug Triaging

In this section, we first describe the *cold-start* problem in bug triaging. Then, we explain why our method can alleviate the *cold-start* problem.

6.1. Problem Statement

As already noted, a large number of methods have been proposed for bug triaging. However, most of these methods are suitable to recommend to only developers who have resolved a number of reports for the project and fail to provide effective recommendations for new developers who did not fix any bugs before. We refer to this problem as *cold-start* in bug triaging. The reason behind this lies in the fact that most state-of-the-art works use algorithms with the information of previously fixed bugs to make a model of developers' expertise [57]. This indicates that new developers, recently joining the project, may not meet the prerequisites and thus cannot be recommended by those methods to fix the given bug even though they have the appropriate expertise to resolve specific kinds of reports. Take the deep-learning-based model depicted in Figure 5 as an example to briefly explain the reason. Assume that the training set contains four bug reports, while the testing set only contains one bug report. First, we train a deep learning model using the four bug reports in the training dataset. (Obviously, this is not a practical solution due to the very few samples to train model. We only use the example to explain the *cold-start* problem.) Next, we predict the labels of new bug reports in the testing set and then evaluate the prediction model by comparing the prediction and the ground truth. As shown in Figure 5, since the output of the prediction model is the probability distribution over the sequences of "Dev.A", "Dev.B" and "Dev.C", it is impossible for the model to give the likelihood of the new bug report (i.e., Bug 5) being predicted as "Dev.D". Thus, to avoid affecting the experimental evaluation of model performance, a common practice for deep-learning-based methods in the literature [11,12] is to remove those test bug reports whose developers, involved in fixing bugs, do not appear in the training set or appear fewer than some minimum number of times. Note that even though the example we present above is based on deep learning, the same problems also arise in other methods [10].

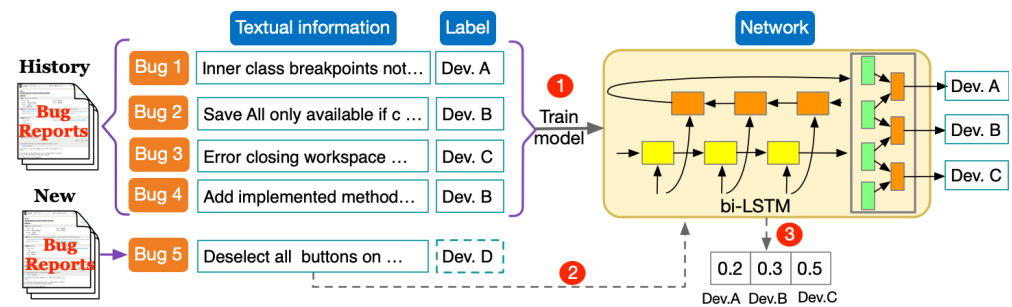


Figure 5. Bug-triaging process based on deep learning methods.

6.2. Alleviating Cold-Start Using PTITransE

Different from previous works, our proposed bug-triaging approach is designed to recommend suitable developers by using link prediction, which can learn the embeddings of all developer entities using PTITransE regardless of whether it is a ground truth of historical bugs or not. For example, if a developer *A* only *reports* a bug or *comments* on bug reports without *fixing* activity, its embedding can also be learned from triplets (*A*, *report*, *bug*) or (*A*, *write*, *comment*). Once the embeddings of developers are obtained, the link prediction can be used to compute the matching degree between each new bug report and each developer, thus achieving accurate developer recommendation for fixing new bugs. Based on the aforementioned analysis, we can see that PTITransE relaxes the

restriction that all developers, who can be recommended by the prediction model, must resolve a number of bug reports in the training dataset. In other words, when evaluating our proposed method with testing dataset, we only need to remove the new bug reports whose ground truth (i.e., developers) does not appear in any report fields of bug reports in the training set. Obviously, the number of bug reports needing to be removed from the testing dataset has been reduced greatly, compared with state-of-the-art approaches for bug triaging. This demonstrates that our proposed approach can alleviate the *cold-start* problem. The comparison between our proposed approach and other state-of-the-art works in terms of alleviating the *cold-start* problem will be discussed in our experiment.

7. Experiments

In this section, we analyze the following research questions (RQs) to demonstrate the effectiveness of our method.

RQ1: Whether PTITransE is superior to other representation learning models for the issue of bug triaging?

RQ2: How does our proposed method perform compared with the state-of-the-art bug-triaging methods under the stringent assumption? And which of the textual and structured information is the more important factor that determines the quality of bug triaging?

RQ3: How does our method perform compared with the existing bug-triaging methods under the weak assumption?

RQ4: How much does the prediction rely on each textual field (e.g., *summary* and *description*) of bug reports, and how much does it rely on other relationships except for the “fix” relationship?

RQ5: How well does the training size affect the prediction performance on bug triaging?

RQ6: Can PTITransE help to alleviate the cold start problem?

Our code and experimental data are publicly available at <https://gitlab.com/jiangyuan448/kbs-and-rl-for-bug-triaging.git> (accessed on 29 May 2025).

7.1. Datasets

We collect 320,520 bug reports with the status “closed” and “fixed” from three open-source software bug repositories (i.e., Eclipse, <http://bugs.eclipse.org/bugs/>, Mozilla, <https://bugzilla.mozilla.org/home> and Apache, <https://bz.apache.org/bugzilla/> Bugzilla repositories, accessed on 29 May 2025) to assess the performance of our proposed approach. For each dataset, we collect the basic information of bug reports, including the number of collected bug reports (# of bug reports), unique developers in the collected bug reports (# of unique developers), unique fixers (# of unique fixers), unique reporters (# of unique reporters), different products (# of products) and different components (# of components). The details of these datasets are shown in Table 4. From the table, we observe the # of unique fixers in Apache is 24, which is much less than that of Eclipse or Mozilla. This phenomenon occurs due to the community-based maintenance process in Apache, i.e., each bug report is assigned to a developer group to fix. We examine different scales of fixers in three projects to investigate if our approach works well for diversity situations.

It is noteworthy that developer aliases may affect our experimental results because one developer may have more than one identifier in a project. Taking the Eclipse bug report #6450 as an example, one of the developers has two identifiers including “Martin Aeschlimann” and “martinae”. Since bug reports are readily available in the form of XML online and developer aliases appear frequently together in an xml element (e.g., “<assigned_to name=“Martin Aeschlimann”>martinae</assigned_to>”, https://bugs.eclipse.org/bugs/show_bug.cgi?ctype=xml&id=6450, (accessed on 29 May 2025)), we thus

Table 4. The details of datasets.

To further prepare the data for our experiments, the bug reports from each project in the datasets are initially sorted in chronological order and then divided into 11 disjoint equal-sized sets $fold_1, fold_2, \dots, fold_{11}$, where $fold_{11}$ contains the newest reports, whereas $fold_1$ is the oldest [58]. Note that the bug reports from Apache are divided into three folds due to the smaller size of the project. We conducted an offline pre-training of our methods on $fold_n$ and online bug triaging on the test dataset $fold_{n+1}$, for all $n \leq 10$, which can guarantee that the embeddings of the *developer* entities and the *fix* relation used in link prediction to predict the suitable fixers for the bug reports in the test dataset are all learned from the most recent bug reports. To ensure there is no “ground truth” of testing bug reports in the constructed knowledge base, we remove a part of the triplets extracted from testing bug reports, whose entities cannot exist in new bug reports, e.g., tossed and bug-fix information. In addition, to adjust the parameters of the bug triaging models, we employ the same split of the oldest fold $fold_1$ between the training and validation sets as employed by previously published research studies [58]. The former 60% of $fold_1$ are used to train models, and the remaining 40% are used to evaluate the optimal parameters that maximize the metric values.

Table 5. The details of statistics of the constructed knowledge base for each CV experiment. “# of train set” represents the total number of historical bug reports used to train model for each CV. “# of test set” represents the total number of new bug reports used to evaluate model built for each CV. “# of entities” and “# of triplets”, respectively, represent the number of entities and triplets in the knowledge base constructed for each CV experiment following the procedure described in Section 5.1.

[illegible]

7.2. Evaluation Metric

Recall@k: the ratio of bug reports whose fixers are successfully detected in the recommended top- k developers ($N_{detected}$) among all bug reports (N_{total}) in testing dataset.

$$Recall@k = \frac{N_{detected}}{N_{total}} \quad (10)$$

MR (Mean Rank): The mean rank of correct entities.

Hits@1: the strict metric that validates the accuracy of picking only the first predicted entity in the ranked list. The higher its value, the more effective the recommendation system [59].

MRR (Mean Reciprocal Rank): The average of the reciprocal ranks of results for a set of queries (new bug reports). If the first returned result is relevant, MRR is 1.0. Otherwise, it is smaller than 1.0 [39]. Here, MRR is defined as follows:

$$MRR = \frac{1}{|N|} \sum_{i=1}^{|N|} \frac{1}{rank_i} \quad (11)$$

where $rank_i$ is the rank of ground truth (i.e., the actual developer) for the i th bug report, and N is the testing set.

MAP (Mean Average Precision): The mean of average precision (Avep) scores for a set of queries (new bug reports). The AveP is defined as follows:

$$AveP = \frac{\sum_{k=1}^{|M|} (P(k) * rel(k))}{number\ of\ relevant\ document} \quad (12)$$

where M is the set of retrieval developer candidates, $|M|$ is the number of retrieval developers, k is the rank in M , $P(k)$ is the precision at cut-off k in the list, and $rel(k)$ is an indicator assigned 1 if the developer is relevant and 0 otherwise.

Among these metrics, MR, Hits@1 and MRR are widely accepted to determine the performance of representation learning models, while Recall@k, MRR and MAP are used to evaluate the ranking performance of bug triaging models, consistent with evaluations from previous research [13,55].

7.3. Experiment Setting

We evaluate the performance of our method from three aspects as described in the RQs.

In RQ1, we evaluate the performance of the proposed representation learning model PTITransE when applied in bug triaging. We choose several commonly used representation learning models including TransE [14], TransD [45] and TransH [43] for comparisons. The evaluation is carried out under the same conditions of the constructed bug-triaging knowledge base and the setting of parameters. To optimize the performance of representation model, we set the uniform dimension k of both word embedding and structured embedding among {100, 200, 300}, the value of minibatch b among {200, 300, 400}, the value of learning rate l among {0.05, 0.01, 0.005, 0.001}, the margin $\gamma = 1$ and the dissimilarity function $d = L_1$. For different combinations of parameters, we repeat the evaluation five times and then calculate the average performance to chose the best hyperparameters.

Furthermore, we extensively evaluate our method compared with the existing automatic bug-triaging methods. In this research, we divide previous studies into two categories based on the different assumptions to estimate the “suitable” developers. In the first category, these studies aim at developing methods to accurately recommend a true or ultimate developer who can completely resolve this bug, which are based on the stringent assumption that a bug is resolved by only one single fixer. However, the second

category of methods adopt the weak hypothesis that a bug is collaboratively resolved by a set of developers, not merely one fixer. We evaluate the performance of our method under different assumptions in RQ2 and RQ3, respectively.

In RQ2, we compare the performance of PTITransE with the state-of-the-art bug-triaging methods under the stringent assumption. These methods often treat bug triaging as a multi-label classification task [12] where the content-based features of bug reports, such as *summary* and *description*, are data points and each developer corresponds to a class label. Following the success of deep learning methods in capturing the semantic and syntactic relationships between words, recent works have focused on using deep neural networks to model the distribution between the bug reports and developers. The experimental results demonstrate the feasibility and effectiveness of the above research methods. Therefore, we compare the proposed method against the deep-learning-based baseline methods including CNN triager [12], LSTM triager and DeepTriage [11]. CNN triager and LSTM triager are based on the convolution neural network (CNN) [60] and the long short-term memory (LSTM) network [61], respectively. The difference between LSTM triager and DeepTriage is whether attention mechanism is applied. The hyperparameters of deep neural networks are determined according to experiments and previous studies. For example, the dropout parameter can simply be set at 0.5, which seems to be close to the optimal value for a wide range of networks and tasks [62]. Similar to the setting of dropout, we employ an adaptive gradient descent optimization (ADAGRAD [63]) of mini-batches using a constant learning rate 0.001 to train the weights of networks, and each mini-batch consists of 50 training items. For the other hyper-parameters (e.g., the shape of the convolution kernel), we conduct a grid search and train all the combinations of the candidate configuration. Then, we compare their performance to find the best hyperparameter values.

In RQ3, we compare our approach with the existing bug-triaging methods under the weak assumption. These methods including KSAP [13], Drex [4], ML-KNN [15,53], DRETOM [19], Bugzie [8], DevRec [15] and developer prioritization (the DP method) [64] are based on heterogeneous network, homogeneous network, multi-label classification, topic model, technical terms analysis, multi-label classification and developer prioritization, respectively. These methods will be detailed in Section 2. To ensure fair comparisons, we take some measures to conduct the experiments: use the same projects as the previous literature [13], use the same metrics and use the reported results in original research publications [13] as comparisons.

It is worth noting that some recent studies have proposed a few improved models based on these technologies to improve the performance of bug triaging, such as MTM [10] and BugFixer [20]. However, the evaluation metrics in these studies are different from our work, and the datasets, as well as source code, are not freely available on the website. We could not reproduce the results reported in these literature works. Therefore, we did not compare with the two methods in this paper.

In RQ4, we want to see how these components (i.e., textual and structured information) combine and contribute to the final performance. To achieve this, we evaluate the importance of different textual fields and structured relationships of knowledge bases on bug triaging with comprehensive ablation experiments using the same experiment scheme for the proposed model as we did in RQ1.

In RQ5, to evaluate the effect of the training set size on the performance of PTITransE, we vary the number of folds N from 4 to 20 to perform N -fold cross-validation. As N increases, the size of the training set gradually decreases.

In RQ6, we also want to see whether our approach can help to alleviate the *cold-start* problem [65]. The one major difficulty for bug triaging, though common, is the *cold-start* problem, where developers, who have recently joined the project, may not be recommended

to fix the given bug since they do not have enough historical bug-fix information even though they have the appropriate expertise to resolve specific kinds of reports.

7.4. RQ1: Whether PTITransE Is Superior to Other Representation Learning Models for the Issue of Bug Triaging?

Approach. To evaluate our proposed method PTITransE at bug triaging, we perform experiments on three real-world datasets and compare with other representation learning models. As described in Section 3.4, we treat bug triaging as a link prediction task. But unlike the previous link prediction, we only treat *head* entity in the unseen triplets as target entity rather than both of *head* and *tail* entities for the bug-triaging task. Thus, the *developer* in the triplet of (*developer*, *fix*, *bug*) is the target entity to be predicted by the representation learning model.

Results. The experiments are performed using cross-validation (i.e., 10-fold CV), and the results are averaged. Table 6 illustrates the averaged experimental results of different representation-learning-based approaches on link prediction metrics (i.e., MR, Hits@1 and MRR) for bug triaging. Additionally, considering varying numbers of developers used for training and testing across the cross-validation (CV) sets, i.e., the bug-triaging model is trained on different folds with different number of developers, taking the averaged performance values (i.e., the results in Table 6) would only provide a statistical estimate of the model performance and is not accurately interpretable [11]. It is, therefore, necessary to report the top-*k* accuracy of each cross-validation set to compare the variance among different approaches introduced in the model training. In this section, the top-*k* accuracy results of each cross-validation set on different projects are bucketed and plotted as box-plots [66,67], as shown in Figure 6, where *k* is in the range of [0, 10] in our experimental setting. From Table 6 and Figure 6, we can draw the following conclusions:

- (1) All typical representation learning models (i.e., TransE, TransD and TransH) use only structured information of bug reports in the bug-triaging task and achieve promising results on all projects. The results indicate that structured information can reflect the process of historical fixing activities of developers and provide valuable information to improve the performance of automatic bug triaging.
- (2) The performance of TransE is superior to that of the improved representation learning models for bug triaging, such as TransD and TransH, when using widely accepted metrics MR and Hits@1 as well as MRR as our evaluation metrics, as shown in Table 6. This is due to the fact that the *fixer* is a specific one-to-one relationship between *developer* and the given *bug* entity, which can be solved more effectively by the TransE model. In addition, more sophisticated representation learning models, e.g., TransH and TransD, need to learn more parameters that may lead to the model overfitting. However, from Figure 6, we found that in some cases, i.e., recommending top-2 ~ top-10 developers for bug reports in Eclipse project, the performances of TransH and TransD are higher than that of TransE, although they are still significantly lower than that of the proposed method PTITransE. This provides supporting evidence that the improved representation learning models (i.e., TransD and TransH) may perform better than TransE for this purpose in recommending multiple (i.e., top-*k*, where $k \geq 2$) developers for bug reports in some projects like Eclipse.
- (3) PTITransE provides further improvement compared with the TransE model because the TransE model only utilizes the structured information and ignores another type of important information, i.e., the textual information of bug reports. It has been proved that the textual information is very useful for bug triaging in traditional methods [10–12], which are mostly based on the fact that a developer can resolve the bug reports with a similar textual description since the bug reports with high

textual similarity often correspond to the same type of bugs. Therefore, PTITransE can improve the performance of the representation learning model by combining this textual information with structured information.

- (4) The superiority of PTITransE's performance is more obvious on the Apache project than that on Eclipse and Mozilla projects. The reason is that the number of developers (i.e., 24) in Apache is much smaller than that of the Eclipse (i.e., 1833) and Mozilla (i.e., 2948) as described in Table 4. The larger the number of developers, the greater the chance to make errors is. This is also consistent with previous works [10,13].

Table 6. Evaluation results of different representation-learning-based approaches on link prediction metrics for bug triaging.

Project	Mozilla			Eclipse			Apache		
Metirc	MRR	MR	Hits@1	MRR	MR	Hits@1	MRR	MR	Hits@1
TransH	0.569	52.314	0.411	0.578	25.64	0.393	0.513	26.272	0.369
TransD	0.571	36.938	0.407	0.577	25.32	0.389	0.610	17.62	0.496
TransE	0.613	16.714	0.440	0.566	11.806	0.394	0.919	1.504	0.883
PTITransE	0.698	8.903	0.505	0.604	8.547	0.431	0.941	1.258	0.909

In a word, the representation learning models have been proved to be effective in bug triaging through our experiments, and the essential reason why PTITransE is superior to other representation learning models is that it enables the learned embeddings of bug entities to encode both structured and textual information. This is the key difference of our model from the existing representation learning models.

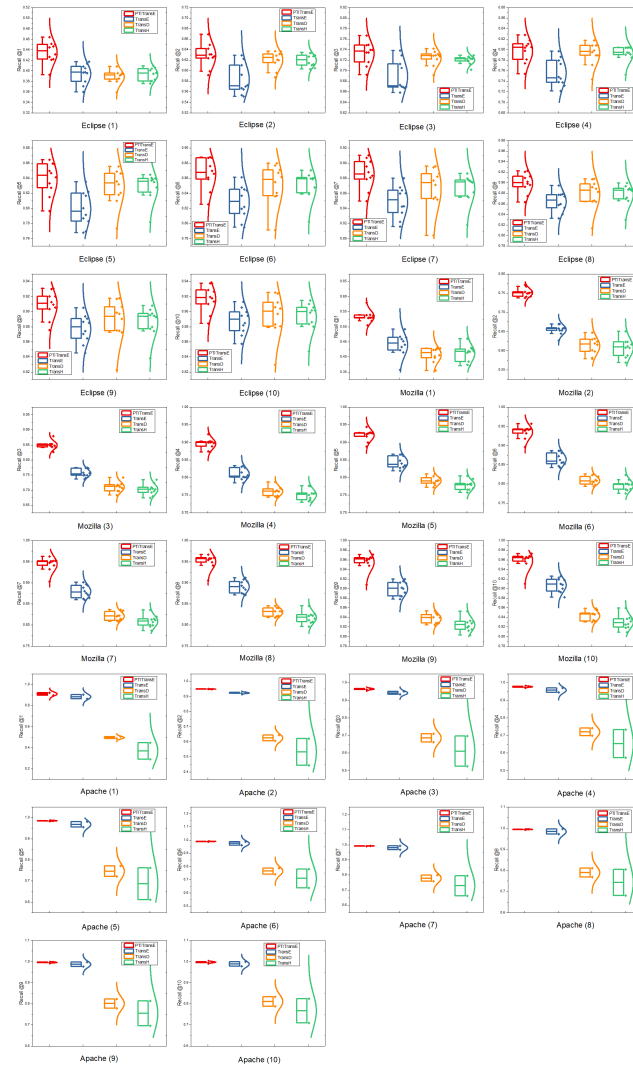


Figure 6. Boxplots showing the performance of different representation-learning-based bug-triaging methods in recommending top-1 ~ top-10 developers (i.e., Recall@1 ~ Recall@10) on N -fold cross-validation on all datasets. For example, “Eclipse (1)” represents the Recall@1 values on 10-fold cross-validation on Eclipse dataset, where each point represents the Recall@1 value on the one testing fold, and each boxplot refers to one of the four (PTITransE, TransE, TransD and TransH) representation-learning-based bug-triaging method alternatives.

7.5. RQ2: How Does Our Proposed Method Perform Compared with the State-of-the-Art Bug-Triaging Methods Under the Stringent Assumption? And Which of the Textual and Structured Information Is the More Important Factor That Determines the Quality of Bug Triaging?

Approach. The deep-learning-based bug-triaging methods, including CNN triager, LSTM triager and DeepTriager, predict suitable developers to fix the given bug by analyzing the content patterns of previously resolved bug reports, while our method recommends the best fixer for the given bug by using the knowledge learned from both the structured and textual features. To better demonstrate the superiority of our method under the assumption of stringent evaluation criterion (i.e., only the developer who really fixed the bug is treated as ground truth), we conduct experiments to compare the proposed with the existing state-of-the-art deep-learning-based bug-triaging methods via 10-fold validation experiments on three datasets. In addition, we would also like to investigate which of the textual and structured properties is the more important factor that determines the quality of bug triaging. Therefore, we compare the TransE method only based on structured features with the textual-classification-based methods. Our experiments recommend different numbers

of developers so as to make a more comprehensive comparison with other methods. The number of developers recommended, denoted as k , ranges from 1 to 10.

Results. Figure 7 shows the averaged performance (i.e., Recall@ k) of different deep-learning-based and representation-learning-based bug-triaging methods in recommending top- k developers on 10-fold cross-validation experiments for three open source projects including Eclipse, Mozilla and Apache. Furthermore, we report the top- k accuracy (i.e., Recall@ k) for each cross-validation set, as shown in Figure 8, to understand the variance between different methods introduced in the training phase considering that it provides more accurate interpretation than the averaged performance values, as detailed in Section 7.4. From Figures 7 and 8, we can obtain the following conclusions:

- (1) PTITransE outperforms the deep-learning-based bug-triaging methods by a great margin on all projects. This could be due to the fact that PTITransE can correctly learn the vector representations for all entities and relations by taking into account both structured and textual information of bug reports. However, deep-learning-based bug-triaging methods make fixer prediction only according to the textual content of bug reports, which may cause great inaccuracy due to the lack of structured information. Additionally, this also demonstrates the effectiveness of applying knowledge bases and representation learning to bug triaging.
- (2) The performance of TransE based on structured features is superior to that of textual-based deep learning methods, which indicates that structured features of bug reports are the more important factor that determines the quality of bug triaging. This could be due to the fact that the structured information is critical as it reflects the developers' behavior in their previous bug-fixing activities, which can help to accurately analyze the developers' experience so that the most appropriate developers are recommended to fix the new bug reports.
- (3) For the Apache project, the deep-learning-based methods also achieved good results due to the lesser number of fixers, which indicates that these methods are more suitable for the projects with a small number of candidate developers. The main reason is that, for deep-learning-based models, the number of nodes of their output layer is determined by the number of developers, i.e., (the # of unique fixers) in Table 4, and the greater the number of output nodes, the more complex the deep-learning-based model. This means that given a fixed amount of data, a greater number of output nodes usually leads to poorer results. However, our proposed method performs well in all projects, which demonstrates the robustness and generalization of our method in terms of the scale of developers. This is because our method relies on a reduced set of parameters as it learns only a dense low-dimensional vector for each entity and each relationship.
- (4) From the obtained results of Figure 8, it can be clearly observed that the deep-learning-based bug-triaging methods produce larger variance than the proposed methods on N -fold cross-validation results, especially for the Apache project. Again, this is due to the fact that the deep-learning-based methods may be sensitive to the text quality of bug reports. Additionally, the bug reporters may have different writing styles and diction for describing bugs [68]; hence, there are possibly notable differences in the text quality across the cross-validation (CV) sets. However, our proposed approach is not restricted to a specific type of information (i.e., the textual information or structured information) of bug reports by jointly learning latent representations of both of them using PTITransE. Therefore, the effects (i.e., high variance) resulted from large differences in text quality across the CV sets can be alleviated by incorporating structured information about the developers and bug reports into the recommendation process.

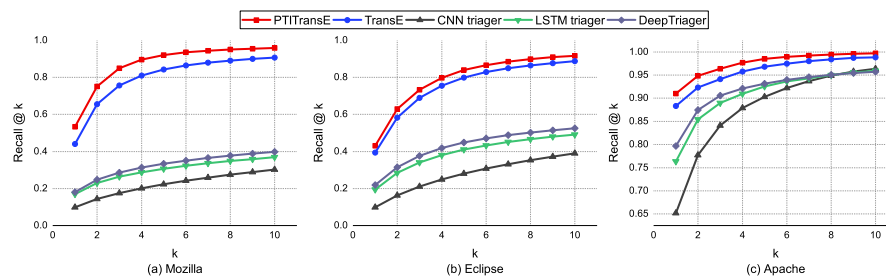


Figure 7. The averaged performance of different deep-learning-based and representation-learning-based bug-triaging methods in recommending top- k developers on 10-fold cross-validation experiments for Eclipse, Mozilla and Apache projects under stringent hypothesis

In addition, the MRR values of PTITransE and other deep-learning-based bug-triaging methods on three projects are shown in Table 7. The results indicate that the performance of PTITransE on the MRR metric are improved from 18% to 286% over the deep-learning-based methods. This means the ranking of the actual fixers (i.e., ground truth) produced by PTITransE is much closer to the top of the ranked list than the baselines.

The results provide evidence to suggest that the deep-learning-based bug-triaging methods are particularly suitable for small-scale developers assignment, and the representation-learning-based methods can scale well to a large number of developer assignments.

Table 7. MRR of PTITransE and other deep-learning-based bug-triaging methods under the stringent hypothesis.

Project	Mozilla	Eclipse	Apache
CNN triager	0.098	0.099	0.652
LSTM triager	0.169	0.195	0.763
DeepTriager	0.181	0.219	0.797
PTITransE	0.698	0.604	0.941

7.6. RQ3: How Does Our Method Perform Compared with the Existing Bug-Triaging Methods Under the Weak Assumption?

Approach. As detailed in Section 7.3, these bug-triaging methods under the weak hypothesis assume that a bug is collaboratively resolved by a set of developers, not merely one fixer. That means the *ground truth* consists of these developers who participated in and contributed to resolving the bug. Taking the bug report #6447 from the Eclipse project as an example, the developers involved in “tossed|fix|assign to” relations include “Joe Szurszewski”, “Darin Wright” and “Darin Swanson”, who are regarded as ground truth for this bug report. Obviously, if the developers who participated in a bug resolution are treated as the ground truth as well, the hit rate of recommendation will increase. Therefore, we conducted further experiments to compare the performance of the proposed method with the existing methods under the weak hypothesis. Similar to RQ2, we set the number of recommended developers (i.e., k) to a value in the range [1, 10] to make a more comprehensive comparison.

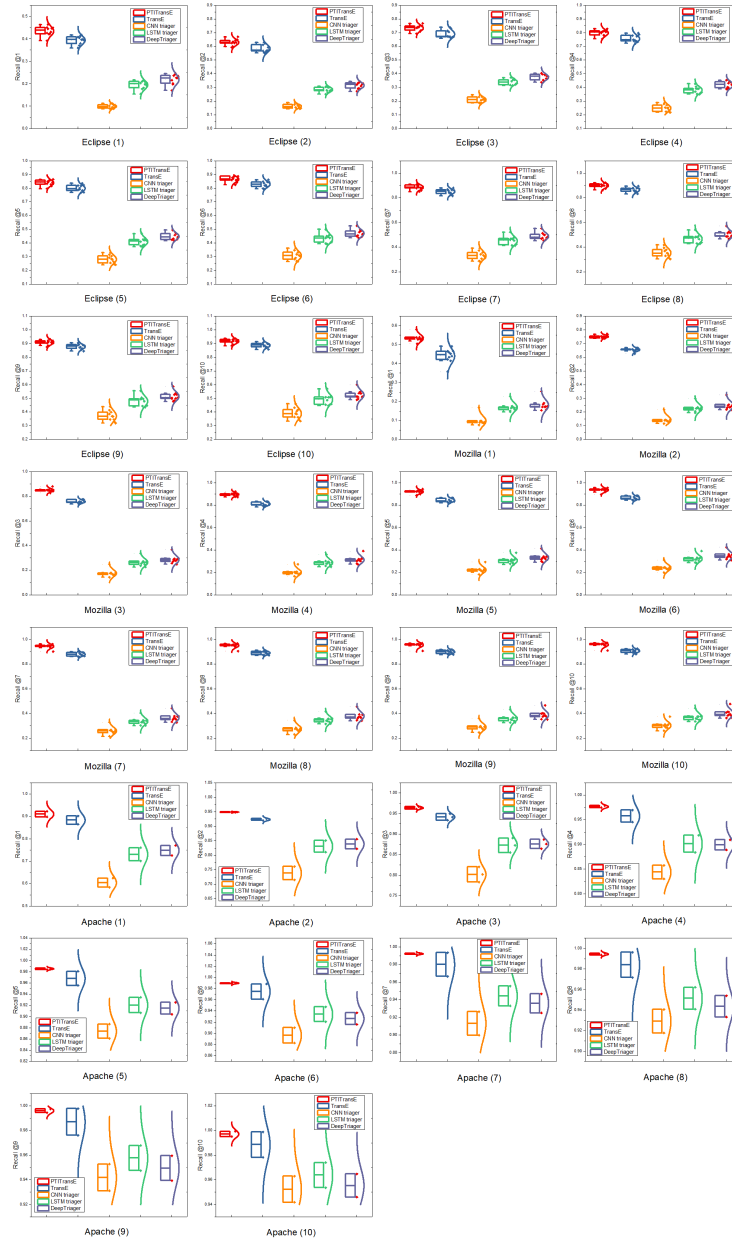


Figure 8. Boxplots showing the performance of different deep-learning-based and representation-learning-based bug-triaging methods in recommending top-1 ~ top-10 developers (i.e., Recall@1 ~ Recall@10) on N -fold cross-validation on all datasets under stringent hypothesis. For example, “Eclipse (1)” represents the Recall@1 values on 10-fold cross-validation on Eclipse dataset, where each point represents the Recall@1 value on the one testing fold, and each boxplot refers to one of the five (PTITransE, TransE, CNN trigger, LSTM trigger and DeepTriager) representation-learning-based or deep-learning-based bug-triaging method alternatives.

Results. The averaged values of *Recall@k* on 10-fold cross-validation experiments for three projects are shown in Figure 9. Additionally, for this RQ, we did not report the top- k accuracy of each cross-validation set to compare the variance between different methods considering the fact that the original experimental results of each cross-validation set for the compared methods are difficult to obtain, which is one of the most challenging problems in the bug-triaging field [11] and will be further discussed in the following analysis and also in the discussion Section (i.e., Section 8). Through Figure 9, we draw three interesting conclusions:

- (1) The highest recall among different methods is derived by PTITransE when top 2~10 developers are recommended for different projects. PTITransE achieves more than 95% in recall metric when top-10 developers are recommended for different projects, which is significantly better than traditional methods.
- (2) For our proposed method PTITransE, the recall curve of Eclipse has the same trend with that of Mozilla. Even if only a small number of developers are recommended, PTITransE still achieves the same desired results for all projects. This indicates that our method has strong robustness.
- (3) For Apache project, PTITransE achieves the best result in recall@k (i.e., 1.0) once k is equal to or greater than 6. This means the top-6 recommended developers in the ranked list contain ground truth for all new bug reports in the testing set.

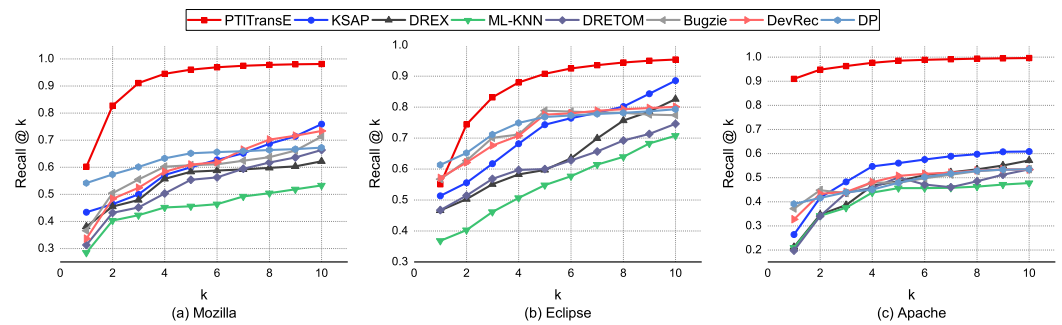


Figure 9. The averaged performance of different information-retrieval-based bug-triaging methods and the proposed method (i.e., PTITransE) in recommending top-k developers on 10-fold cross-validation experiments for Eclipse, Mozilla and Apache projects under weak hypothesis.

The reason behind the higher performance of PTITransE on the Apache project is that the number of fixers in Apache is significantly less than that of other projects. This agrees with the results in Figure 7 and Table 4.

To further evaluate the usefulness of the ranked list, we calculate the MRR and MAP values on different projects for our proposed PTITransE and other traditional methods, including DP, DevRec, Bugzie and KSAP. The experimental results, as shown in Table 8, demonstrate that our approach outperforms the baseline methods with a significant margin.

Note that, as argued in [11], one of the biggest challenges in performance comparisons between different approaches is the lack of a publicly available benchmark dataset and source code implementation of the existing research. Thus, we directly copy the results of different approaches from the literature [13] to compare with our proposed method under the weak assumption. Though both are obtained using the same projects and the same performance metrics, it is obvious that the dataset used in this study is not exactly the same as in [13], which may bring bias to our experiments. Therefore, we should give careful consideration to the impact of different datasets on bug-triaging performance. For Eclipse and Mozilla projects, since the number of developers in these projects used in this study is larger than that in [13], we can cautiously draw a conclusion that our approach outperforms all prior traditional approaches on Eclipse and Mozilla projects, as shown in Figure 9. However, for the Apache project, the number of developers in our experiment is less than that in [13], and the larger the number of developers, the greater the chance to make errors is (as discussed in Section 7.4); hence, conclusions from such data may not be valid. We will further explain the threats to the validity of our experiments in Section 8.

Table 8. MRR of PTITransE and other retrieval-based bug-triaging methods under the weak hypothesis.

Metric	MRR			MAP		
Project	Mozilla	Eclipse	Apache	Mozilla	Eclipse	Apache
DP	0.113	0.085	0.169	0.220	0.207	0.172
DevRec	0.134	0.151	0.170	0.232	0.289	0.173
Bugzie	0.246	0.261	0.187	0.374	0.425	0.211
KSAP	0.278	0.280	0.347	0.445	0.564	0.365
PTITransE	0.758	0.703	0.941	0.672	0.584	0.937

7.7. RQ4: How Much Does the Prediction Rely on Each Textual Field (e.g., Summary and Description) of Bug Reports, and How Much Does It Rely on Other Relationships Except for the “Fix” Relationship?

Approach. In this section, we systematically analyze the influence of each textual information and each structured information of knowledge bases on bug triaging. The main contents of the experimental design are as follows:

(1) Verifying the importance of bug’s *summary* and *description* to bug triaging. Although combining textual and structured information has been shown to improve the performance of bug triaging in RQ1, it requires further refinement and validation about which textual field (e.g., *summary* and *description*) contributes more to the improved performance. To achieve this, we perform additional experiments investigating the effect of each textual field on the performance of bug triaging by replacing the textual information (i.e., the merged text from *summary* and *description*) of each bug entity in the constructed knowledge base with a single textual field (i.e., *summary* or *description*). For this sub-RQ (research question), we use boxplots [66] of the distributions of evaluation measures (e.g., Recall@1~Recall@10) for Eclipse and Mozilla projects to obtain several interesting results. Note that the reason we did not perform experiments on the Apache project is the relatively small size of its dataset. As we detailed in Section 7.1, the Apache dataset is split into three folds, i.e., *fold*₁, *fold*₂ and *fold*₃, in chronological order, where starting from the second fold, every fold (i.e., *fold*₂ and *fold*₃) is used as the testing set with the previous fold for training. For example, *fold*₂ is used to estimate the performance of the model that is trained on *fold*₁. That means there are only two studies with the dataset of the Apache project, which is not enough to analyze the distribution of the performance difference when using boxplot techniques. In addition, due to the page limitation, we mainly focus on the projects with a large number of developers (e.g., Eclipse and Mozilla) since bug triaging using this type of project is more difficult than that using a smaller one (e.g., Apache).

(2) Verifying the importance of each relationship to bug triaging. “Relationship” represents the fact between *head* and *tail* entities in triplets, which provide valuable information such as the historical activities of developers and the process of bug tracking. Although we use the learned representations of “fix” relationship and “developer” entities to predict the suitable fixers, all other relationships (e.g., “comment”, “toss” and “assign to”) are of importance in learning accurate entity representations because they provide more semantic information of relationships among different entities in the knowledge bases. To find how different each relationship in the knowledge base benefits the bug-triaging task, we perform comprehensive ablation experiments to understand the effect of each relationship. More concretely, in the model construction phase, for each training fold, we remove data (i.e., fact triplets) containing one of the relationships at a time from the knowledge base and then build the bug-triaging model with the remaining data containing the other six relationships. In the model testing phase, we evaluate the model using the same testing folds as in RQ1. This process iterates for each relationship and each cross-validation set. For the sake of clarity, we report the top-*k* ($k \in [1, 10]$) accuracy of the

proposed method on different projects to give a comprehensive study on how much each relationship contributes to the final performance.

Results. From Figure 10, we have the following findings for the first sub-RQ:

- The top- k accuracy of each cross-validation set (i.e., Recall@ k where k is from 1 to 10) shows that, by further incorporating textual information regardless of whether it is from singular fields that contain only *summary* or *description*, or the merged text combining both of fields, PTITransE can perform substantially better than that without considering textual information in the bug-triaging task. This fact agrees with the claim made in RQ1, which, yet again, demonstrates the effectiveness of our method by making full use of both structured and textual information of bug reports.
- Different textual fields of bug reports (e.g., *summary* and *description*) have quite different levels of importance for effective training representation-learning-based models and making accurate recommendations. For the Eclipse project, we found that using either only the *summary* or only the *description* performs better than that using both of *summary* and *description*. This could be due to the fact that we employ the CBOW model (see Section 3.5) to encode the semantic information of textual fields of bug reports, which is calculated by averaging the vectors of all words present in textual fields to produce a single feature vector. Although it is an effective approach, when there is a long sequence of words in textual fields, this method may no longer easily memorize all of them and thus results in the loss of valuable information about the contents of the bug. However, for the Mozilla project, the average sequence length of textual information is 41.5, much shorter than that of the Eclipse project, which is 119.6. Therefore, the aforementioned problem has a slightly negative effect on prediction performance. Additionally, using only the *description* produces larger variance than that using either *summary* or the merged text for Mozilla project. This is because the text quality of the *description* of bug reports varies greatly, and the low-quality *description* of bug reports will adversely affect the performance of prediction models.
- Using the merged text can help improve the performance of traditional representation learning methods (e.g., TransE) based only on the structured information and thus serves as a simple and fast strategy of combining textual information with structured information for the bug-triaging task since it does not require additional experiment to judge which of two textual fields of bug reports is more important; however, it may or may not be the best option to achieve the best performance for different projects.

Conclusions: These results suggest that the textual fields are effective at providing useful knowledge and information in the training phase and can be leveraged to further improve the performance of bug-triaging models based only on structured information, but the proposed representation model (i.e., PTITransE) still suffers from drawback, failing to capture and remember a multitude of semantic information embedded in a long sequence of words effectively, all of which may be necessary to understand the topic of the report.

In addition, from Figure 11, we have the following findings for the second sub-RQ:

- In the case that all the relationships in the knowledge bases are considered, the proposed approach achieves better results than the other six cases without considering one of relationships on all projects. This demonstrates the effectiveness of considering all relationships among different entities for the bug-triaging task.
- Different relationships in the knowledge bases have different importance, which affects the performance of the developer recommendation. For the Eclipse project, among six relationships, the *contain* has the most influence on the efforts for recommending suitable developers. This is because, as a result of removing the *contain* relationship from knowledge bases, it is possible that the affinity information of the fixer will be lost since developers often specialized in some products and components. This is

consistent with previous work [10], which finds that using the product–component combination as the input feature can further improve the accuracy of the LDA-based bug-triaging methods. For the Mozilla project, the *report* relationship is more important than the other five relationships in deriving the high performance of bug triaging. This could be due to the fact that it is possible to find the most relevant developers to fix this bug report only by the *reporter* entity of the bug reports.

- Except for the *contain* and *report* relationships, the other four relationships, i.e., *assign to*, *write*, *comment* and *toss*, have shown a similar influence on bug-triaging performance for both projects. These relationships may shadow the similar parts of the developer communities, which represents that the removal of one of them in knowledge bases will reduce the complex sets of the relationships between the developer members of the software systems. Therefore, the learned representation in such four cases captures fewer intrinsic characteristics of the developers than that learned on knowledge bases with all relationships.

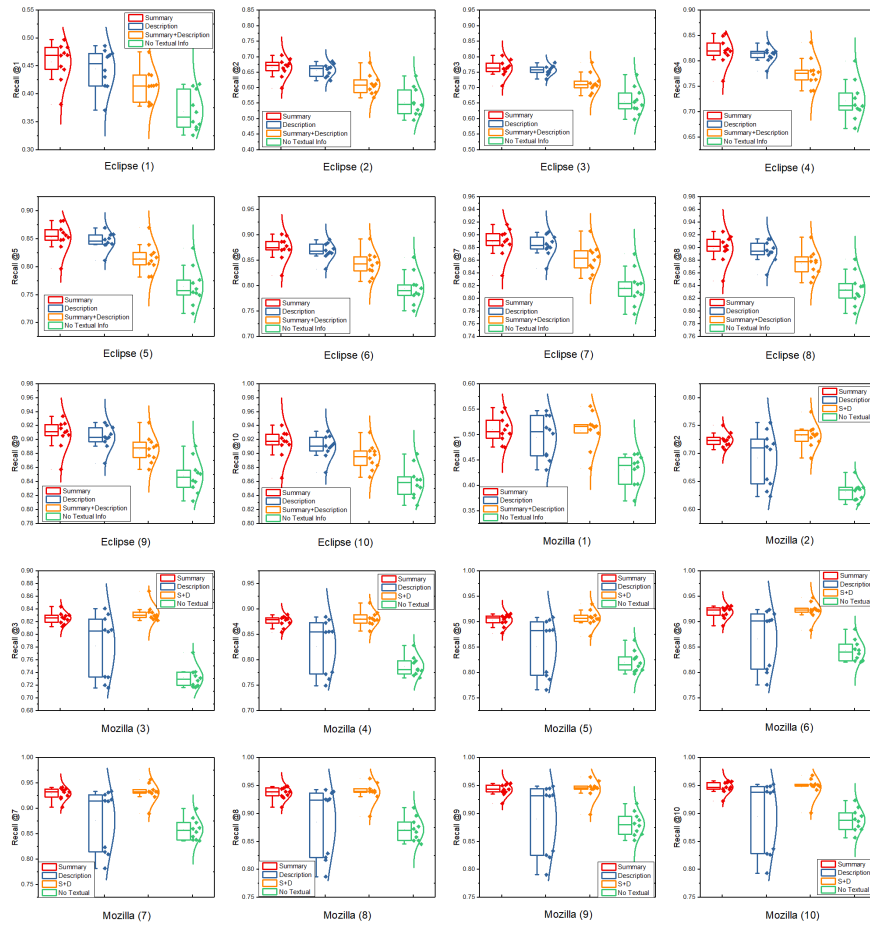


Figure 10. Boxplots illustrating the performance of different textual fields used by the proposed method, *PTITransE*, in recommending top-1 and top-k developers (i.e., Recall@1 and Recall@10) based on 10-fold cross-validation on the Eclipse and Mozilla datasets. The labels *Summary*, *Description* and *Summary+Description* (abbreviated as *S+D*) represent the results of *PTITransE* trained on knowledge bases constructed by combining structured information with the entity summary, entity description and the merged text of both, respectively. Additionally, *No Textual Info* (abbreviated as *No Textual*) indicates the baseline TransE method trained on knowledge bases without incorporating any textual information from bug reports.

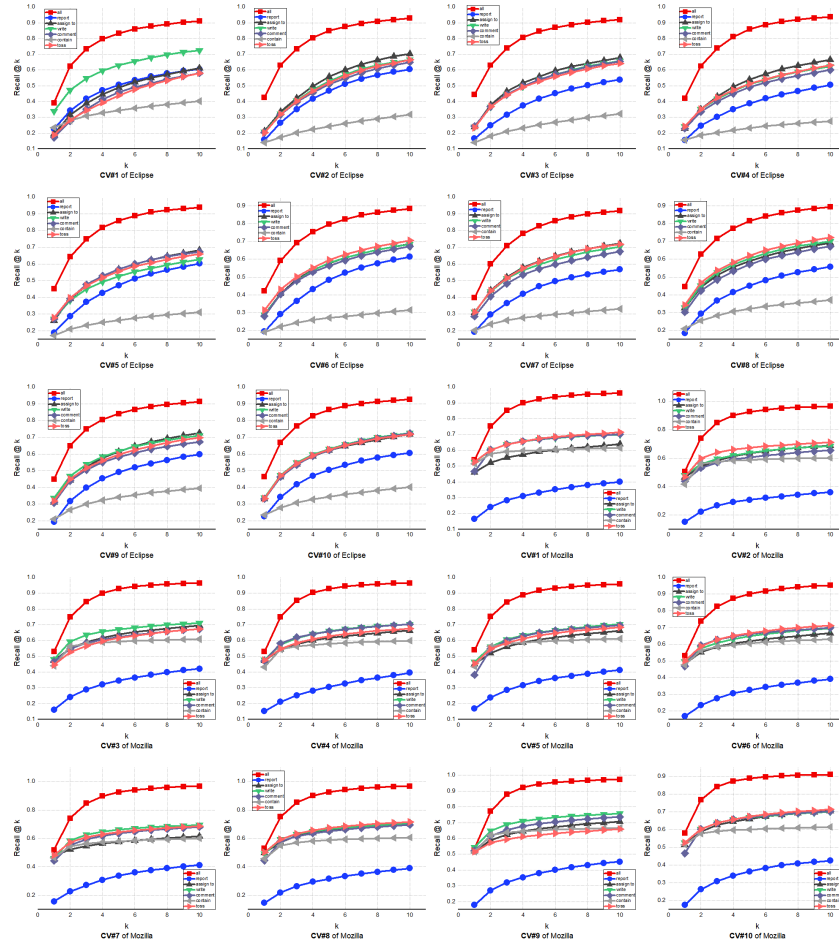


Figure 11. Line charts illustrate the importance of incorporating structured relationships within knowledge bases by conducting a fine-grained, per-relationship ablation analysis. This analysis demonstrates the impact of each relationship on the top- k accuracy ($k \in [1, 10]$) across cross-validation sets using the *Eclipse* and *Mozilla* datasets. For instance, “CV#1 of *Eclipse*” refers to the Recall@ k ($k \in [1, 10]$) results obtained from the first cross-validation fold on the *Eclipse* dataset. In these charts, the “all” curve represents the performance of the proposed model trained on the knowledge base containing all six relationship types, while the other curves (e.g., marked as “report”) represent the model trained on knowledge bases where one specific relationship (e.g., “report”) has been removed, showing how its absence affects performance.

Overall, we have demonstrated that the constructed knowledge bases with all relationships are capable of training a significantly better bug-triaging model than that trained on the knowledge bases lacking of one of the relationships. The reason behind these results is that the representation learning model will generate more and more accurate entity embedding as more relationships are provided for training.

7.8. RQ5: How Well Does the Training Size Affect the Prediction Performance on Bug Triaging?

Approach. To evaluate the performance of our method, we use the longitudinal data setup [8,69] and divide our data into either 11 non-overlapping folds for large-scale datasets (e.g., *Eclipse* and *Mozilla*) or 3 non-overlapping folds for small-scale datasets (e.g., *Apache*); thus, one fold corresponds to different percentages (i.e., 1/11 or 1/3) of the total number of the dataset for each project. In this section, we conduct a comprehensive set of experiments on all datasets to study how well the training sizes affect the prediction performance on bug triaging. To answer this research question, we vary the number of folds (from 4 to 20) of the entire dataset when training the proposed representation model PTITransE. For example, in 4-fold cross-validation, the original dataset of each project is

divided into five equal size subdatasets in the chronological order of the report submission time, where $fold_n$, for $n \leq 4$, is used for training and $fold_{n+1}$ for testing. To compare the performance differences, we use boxplots of the distributions of evaluation measures (i.e., Recall@1 ~ Recall@10) for all projects on N -fold cross-validation to obtain several interesting results, where N is among {2, 4, 6, 8, 10, 12, 14, 16, 18, 20}.

Result. From Figure 12, we have following findings:

- (1) For the Eclipse and Mozilla projects, when the training model using N -fold cross-validation with N is greater than 10, the performance of bug triaging is higher than that with N smaller than 10. That means the performance decreases, although smoothly, as the size of training data increases in most of these results, which is not consistent with the fact that the previous deep-learning-based models with a large number of the learning parameters require a large number of data to adequately train [70]. This is because the proposed representation-learning-based method learns only dense low-dimensional vectors for each entity and each relationship based on their surrounding context, without learning any additional parameters that commonly exist in deep-learning-based models, e.g., network parameters. Therefore, our method does not need a large amount of well-labeled training data compared with deep-learning-based methods. In addition, augmenting the training data by reducing the number of folds will lead to the accumulation of noise and, hence, will show no improvement or even cause a decline in the effectiveness of bug triaging. Furthermore, increasing the number of training instances to cover a longer time period will inevitably introduce more developer candidates, which makes the task more difficult, as we discussed in Section 7.4.
- (2) For the Apache project, the performance of PTITransE decreases as the number of folds increases, which means the top- k accuracy decreases when reducing the sizes of the training dataset. This is because, when applying the N -fold cross-validation setting with a greater N value on the Apache dataset, each fold used for the training model is too small to efficiently improve the discriminability of the learned features for each entity and relationship, which exerts a negative impact on the performance of bug triaging. In addition, as shown in Figure 12 (Apache (1) ~ Apache (10)), we found that, with a smaller number of developers (e.g., 1~4) to recommend, the performance differences caused by the size of the training dataset are larger than that recommending a larger number of developers (e.g., 5~10). This is due to the fact that, although, for various number of folds, the learned embeddings for entities have large differences in capturing the semantic information of these relations, it is more likely that the recommender can make a correct top- k recommendation (e.g., k is from 5 to 10) when the total number of candidates is small.
- (3) Since the number of data instances required in the training phase depends on the project and different projects vary greatly in both complexity and size, especially in the number of developers, it may not help or make sense to give an exact integer number of training instances for different projects. However, across the ten figures for Eclipse and Mozilla projects in Figure 12, PTITransE achieves optimal and stable results on bug triaging when the number of folds is within the specific range of 16~20. We believe it could serve as a useful guide for training the PTITransE model on similar projects to achieve satisfactory results.

Overall, it is concluded that when there are insufficient bug reports (e.g., Apache) to construct the knowledge bases and then train the representation model, the performance of bug triaging may be influenced. Additionally, we believe that the increasing number of training datasets raises the problem of the accumulation of noise when learning entities' representation while making the bug-triaging task more difficult as it introduces more developer candidates.



Figure 12. Boxplots showing the performance differences of PTITransE by varying the number of folds used to split the entire dataset for each project in recommending top-1 ~ top-10 developers (i.e., Recall@1 ~ Recall@10) under stringent hypothesis, where Folds N ($N \in \{2, 4, 6, 8, 10, 12, 14, 16, 18, 20\}$) indicates that the original dataset is partitioned into $N+1$ equal size subdatasets in chronological order of report submission time and $fold_n$, for $n < N$, is used for training with $fold_{n+1}$ for testing. Each point in boxplots represents the Recall value on the one testing fold.

7.9. RQ6: Can PTITransE Help to Alleviate the Cold-Start Problem?

Approach. In this section, we aim to demonstrate whether the proposed approach can effectively alleviate the cold-start problem compared to previous deep-learning-based methods. From the description of *cold-start* problem in Section 6, we can see that, given the same testing dataset, the fewer bug reports that are removed, the better a bug-triaging approach works in terms of alleviating the cold-start problem. Therefore, for each cross-validation set, we compute the number of removed bug reports (i.e., the # of removed reports) that cannot be easily handled by various approaches to verify that the proposed method works for alleviating this problem as intended. Additionally, we are also interested in examining the performance of the proposed method on these bug reports, which are fixed by a minority of developers who have no experience in resolving similar bugs earlier and thus cannot be recommended by deep-learning-based methods.

Results. As evidence, Table 9 shows the statistical results of the number of bug reports removed from each testing fold by a variety of bug-triaging methods (i.e., RL-based and DL-based methods) in three projects, where the bold ones indicate a fewer number of

removed bug reports on each testing dataset. In addition, Figure 13 shows the performance differences of the proposed method PTITransE over the full testing datasets (i.e., “PTITransE tested on all datasets”) or only over the testing bug reports removed by DL-based methods (“PTITransE tested on removed datasets by DL”). From these results, we can observe four notable points.

Table 9. The number of removed bug reports by deep-learning-based methods and representation-learning-based methods. “# of test set” represents the total number of bug reports for each cross-validation set. “# of removed reports by DL” and “# of removed reports by RL” represent the number of testing bug reports removed by deep-learning-based methods and representation-learning-based methods, respectively.

Project	Size	CV#1	CV#2	CV#3	CV#4	CV#5	CV#6	CV#7	CV#8	CV#9	CV#10	Average
Eclipse	# of test set	13,580	13,580	13,580	13,580	13,580	13,580	13,580	13,580	13,580	13,580	13,580
	# of removed reports by DL	1225	858	1505	1285	1698	1379	789	1127	739	625	1123
	# of removed reports by RL	772	459	999	773	1117	918	432	639	241	331	668.1
Mozilla	# of test set	13,628	13,628	13,628	13,628	13,628	13,628	13,628	13,628	13,628	13,628	13,628
	# of removed reports by DL	895	740	722	777	808	655	2730	816	760	1601	1050.4
	# of removed reports by RL	287	261	292	283	337	224	223	402	315	319	294.3
Apache	# of test set	7073	7073	-	-	-	-	-	-	-	-	7073
	# of removed reports by DL	435	602	-	-	-	-	-	-	-	-	518.5
	# of removed reports by RL	10	84	-	-	-	-	-	-	-	-	47

- (1) The fewer number of removed bug reports for different projects are all achieved by representation-learning-based methods as shown in Table 9, which demonstrates that the representation-learning-based method can alleviate the *cold-start* problem.
- (2) Since the removed bug reports from each testing fold cannot be recommended by deep-learning-based methods as their true fixers do not equip with the historical experience in resolving similar bugs earlier (i.e., their fixers did not appear in the training set) as we described in Section 6, the accuracy of the deep-learning-based methods in recommending top-1~top-10 developers (i.e., Recall@1~Recall@10) over those testing bug reports is 0, as shown in Figure 13.
- (3) The performance of the proposed method in recommending top-1~top-10 developers (i.e., Recall@1~Recall@10) over each full testing fold (i.e., “PTITransE tested on all datasets” in Figure 13) is significantly better than that only over the data instances that cannot be recommended by deep-learning-based methods (i.e., “PTITransE tested on removed datasets by DL” in Figure 13). We believe this can be explained by the fact that the removed testing bug reports by deep-learning-based methods are the most difficult to assign among the full testing datasets due to lack of historical experience of their true fixers. Nevertheless, our method still achieves promising results and leads competitors (i.e., deep-learning-based methods) by a large margin. This is because the proposed method can assign the new bug report to a “young” or “new” developer who has no fixing experience in historical bug reports but has participated in the bug resolution activity. This means that our method relaxes the restriction that all developers, who can be recommended by the prediction model, must resolve a number of bug reports in the training dataset. Therefore, we can claim that our approach is effective in alleviating the cold-start problem of bug triaging.
- (4) In particular, the effectiveness of representation-learning-based methods on the Apache project is significantly better than that on other projects in alleviating the *cold-start* problem. That is because the Apache project has less numbers of fixers, one of whom may affect many bug reports in the testing set. Therefore, for deep-learning-based methods, many bug reports in the testing set will be removed in evaluation as long as the ground truth (i.e., developers) of them does not fix any bug reports in the training set. However, as shown in Table 9 and Figure 13, the cold-start problem has less influence on the performance of the proposed method. This may be due

to the fact that most of the developers participated in the bug resolution activity, which also provides evidence that the representation-learning-based methods are more appropriate to organizations with less numbers of developers.

Overall, from Table 9 and Figure 13, it can be verified that representation-learning-based methods can alleviate the *cold-start* problem. Compared with existing deep-learning-based methods, our method outperforms them in alleviating *cold-start* situations.

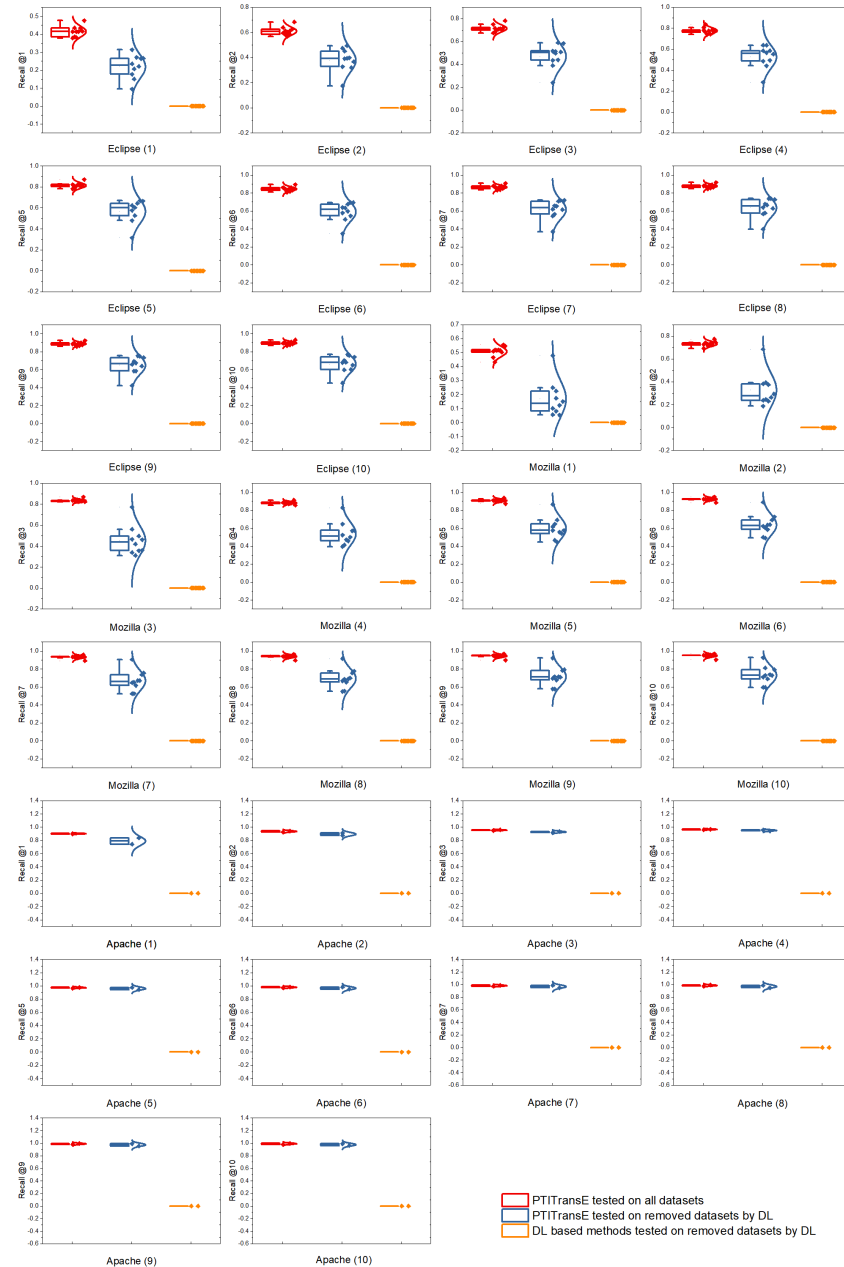


Figure 13. Boxplots showing the performance differences of the proposed method PTITransE tested over different datasets for each project in recommending top-1~top-10 developers (i.e., Recall@1~Recall@10) under stringent hypothesis, where “PTITransE tested on all datasets” indicates that, starting from the second fold, every fold (i.e., $fold_2 \sim fold_{11}$) is used for an assessment of the model with the previous folds for training, and “PTITransE tested on removed datasets by DL” indicates that, on each testing fold, only data removed by DL-based methods are used to estimate the performance of the model while with the previous folds training model. To give a clear comparison with respect to existing DL-based approaches using only textual information, we also present the performance of these methods tested on the datasets that cannot be easily handled by themselves.

8. Discussion

8.1. How to Choose the Best One Between Two Hypotheses for Your Project

In our experiments, we evaluate the performance of our proposed approach under different hypotheses and compare it with other deep-learning-based or rank-based methods. From the experimental results, we can conclude that the proposed approach resulted in better performance under two hypotheses than the traditional ones. Since different development teams may have different numbers of members, here we would like to give some suggestions on which hypotheses are selected in actual practice.

Now, we discuss the following two cases according to the number of fixers who warrant recommendation.

Case 1: The development teams with large numbers of fixers. With the increasing size of software and the increasing number of developers, bug triaging becomes more and more difficult. This is also demonstrated in our experiments by the higher performance of PTITransE on the Apache project than that on the Eclipse and Mozilla projects. As we analyzed in Section 7.6, the most important reason behind this is that the number of fixers in Eclipse and Mozilla are significantly larger than that of the Apache project. Therefore, although this paper proposes an efficient framework to solve this problem using representation learning, it is still challenging to find the most appropriate developers to resolve these bugs in very large bug repositories. Additionally, considering that a bug is collaboratively resolved by a set of developers in large software, we call on the bug triagers to apply our method to real-world applications under weak hypotheses, specifically for large bug repositories.

Case 2: The development teams with smaller numbers of fixers. Apart from large software projects with many developers, it is obvious that there are a lot of projects that are rather small in the size of development teams (e.g., the Apache dataset used in this paper). For such a case, we encourage bug triagers utilizing our proposed method under stringent hypothesis to find the most suitable developers to fix these bugs. The reason for this is twofold: on the one hand, the performance of our approach under stringent hypothesis has a slightly poor but comparative performance with that driving under weak hypothesis, and on the other hand, our method achieves a promising performance at Hits@1 under the stringent hypothesis for projects with smaller numbers of fixers. Hence, it may be more efficient to directly recommend the most likely developers to fix these bugs in small bug repositories.

Note that we do not provide the bound value on the number of team members to distinguish between different scales of projects since this value needs to be estimated by additional empirical research work. We leave it as our future work. But, regardless of which hypothesis is considered by bug triagers, there is no need to change the training process of our proposed method. This is because the training instances in the form of triplets (i.e., (h, r, t)) are the same for both the weak and stringent hypotheses.

8.2. Which Scenarios Are Well Suitable for the Proposed Approach?

In this work, we propose a representation-learning-based method PTITransE for bug triaging to boost the accuracy of recommending suitable developers for fixing given bug reports, which better fits on three projects (i.e., Eclipse, Mozilla and Apache) with different scales as shown in our experimental results. As detailed in Section 7.1, the one aspect that these projects have in common is all of them use Bugzilla (<https://www.bugzilla.org/>) as their bug-tracking system, which could facilitate our data collection work since all other Bugzilla bug databases generally follow the same format, although different projects vary widely [71]. In addition, all bug reports submitted to the Bugzilla-based repository have specific attributes, and each bug report in Bugzilla has a complete bug life cycle [7,72].

Hence, we can easily excavate and make available meaningful the structured (i.e., five entities and seven relationships) and textual (i.e., summary and description) information from the bug repository that the knowledge base construction needs. Therefore, performing bug triaging based on the Bugzilla-based repository is becoming the major target of the proposed method.

Although Bugzilla is one of the most extensive bug databases available, and it has been utilized by many companies [71,72], there are also other emerging forms of reports to record and track bugs happening in the whole life cycle of software products. However, the structured information required in our knowledge base construction process is limited in these bug reports.

For example, we have already noted that the software engineering literature is ripe with bug-triaging methods to directly generate a recommended list of developers for issue reports in github (or called *GitHub* bug reports [73,74]). Taking the *GitHub* bug report #146 from the Angular.js (<https://github.com/angular/angular.js.git>) project as an example, the web page with bug content information in Github is shown in Figure 14, and it can be found in the dataset (<https://github.com/alisajedi/BugTriaging.git>, <https://github.com/TaskAssignment/MSBA>) published by Badashian et al. in [73,74].

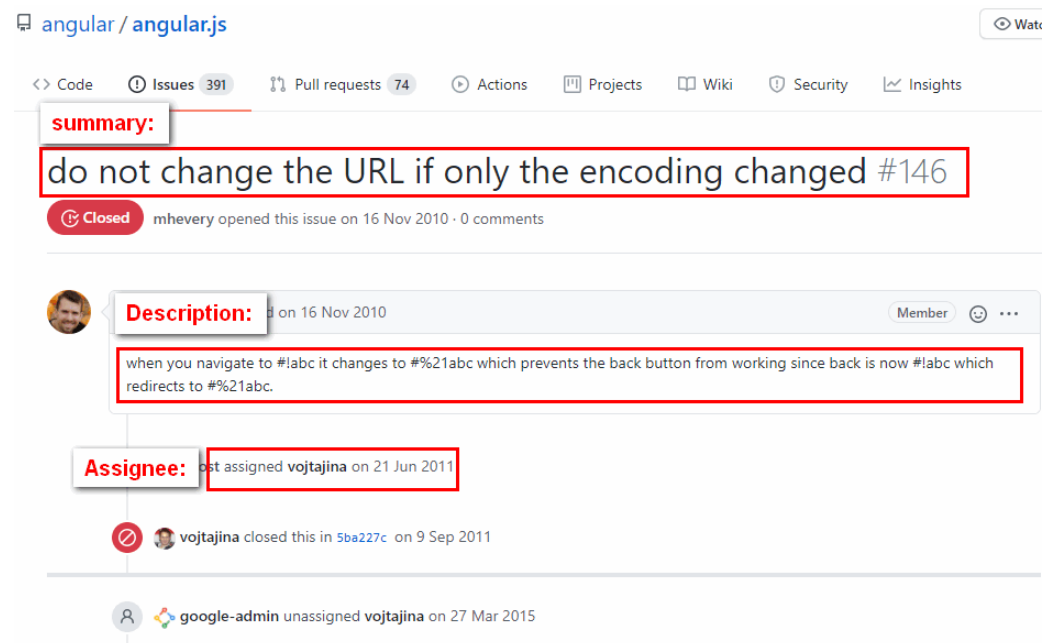


Figure 14. *GitHub* bug report #146 from the Angular.js.

As can be seen from Figure 14, the *GitHub* bug report #146 carries textual information (i.e., summary (or title) and description (or body)) about this bug but contains very little structural information to construct the bug-triaging knowledge base and, therefore, is not suitable for our method to learn the structured information embedded in the interaction among the five entities mentioned in this paper. In the future, we will explore to make full use of this information, e.g., the correlation among developers, by mining StackOverflow or other developer communities and then model developers' experience accurately via the knowledge base so that the most appropriate developer can be recommended to fix the *GitHub* bugs.

In addition, the bug reports for industrial projects are privately owned and cannot easily be obtained since in many cases bug reports contain security-related vulnerabilities that could be exploited by malicious hackers to attack the software-system [52]. Therefore, so far, there are no experiments in this work that apply our proposed model to industrial projects.

8.3. Threats to Validity

Threats to internal validity concern factors that could have influenced our experiments and conclusions. Although we have made efforts to reduce these threats, there may still exist errors that we did not notice. In this study, a comprehensive comparative performance analysis, carried out in Sections 7.4–7.9, demonstrates the high performance of the proposed method over existing approaches. To reduce the experimental biases, we take the following measures to ensure a fair comparison: (1) For RQ1, RQ2, RQ4, RQ5 and RQ6, we reimplement the earlier representation-learning-based and deep-learning-based methods and use the obtained results of our own implementation to expediently compare and analyze the performance differences between our method and these baselines. The main reason for this is that the experimental evaluations of the proposed method in this paper and the existing approaches in the original paper are based on diverse experimental setups, including using different performance metrics and using different projects. Therefore, we cannot directly copy the reported results from their research. Furthermore, to avoid re-implementation errors of existing approaches, we have double-checked our source codes and experiments. (2) For RQ3, we compare the performance of the proposed method with several previously reported results on the same projects used by those methods under the weak assumption. However, due to the fact that the dataset used in this study is not exactly the same as in the previous literature, the conclusion's validity may be impacted as discussed in Section 7.6.

Threats to external validity refer to the generalization of the results [10]. We analyzed 320,520 bug reports from three projects to evaluate the performance of our proposed method. To the best of our knowledge, the scale of bug reports in our experiments is larger than that in previous studies [4,10–13,15,19,20,53]. To further mitigate the threat, we plan to do a more comprehensive evaluation of our method by considering more projects in the future.

9. Conclusions and Future Work

In this study, we proposed an improved representation learning model, PTITransE, that can correctly learn the embeddings for all entities and relations by making full use of both structured and textual information of bug reports. In addition, we built a novel automatic bug-triaging framework based on the knowledge base and representation learning technologies, which can effectively assign bug reports to appropriate developers for bug fixing. The evaluation results on three real-world projects demonstrate not only the feasibility and effectiveness of representation-learning-based bug-triaging methods but also the great performance improvement of our proposed framework compared with the existing approaches. In the future, we will validate PTITransE on more bug repositories for a comprehensive comparison with other bug-triaging methods. More information on bug reports will be investigated and utilized to further improve the performance of bug triaging.

Author Contributions: Methodology, Q.W.; Software, W.Y. and Y.G.; Validation, Y.L. (Yanlong Li); Investigation, S.T.; Data curation, Y.L. (Yiwei Liu); Writing—original draft, Q.W.; Writing—review & editing, P.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: All the data used in this study are available from the authors upon request.

Conflicts of Interest: The authors declare that they have no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

PTITransE	Combining <i>Partial entities' Textual Information</i> with <i>TransE</i>
VSM	Vector Space Model
KBs	Knowledge bases
IR	Information retrieval
RL	Representation learning
NLP	Natural language processing
CBOW	Continuous bag-of-words encoder
HBRs	Historical bug reports
CV	Cross-validation
MR	Mean Rank
MRR	Mean Reciprocal Rank
CNN	Convolution neural network
LSTM	Long short-term memory

References

1. Foundation, M. Bugzilla Bug Tracking System. 1998. Available online: <https://www.bugzilla.org/> (accessed on 1 June 2025).
2. Atlassian. JIRA Issue Tracking System. 2002. Available online: <https://www.atlassian.com/software/jira> (accessed on 1 June 2025).
3. GNATS Project. GNATS Bug Tracking System. 1992. Available online: <https://www.gnu.org/software/gnats/> (accessed on 1 June 2025).
4. Wu, W.; Zhang, W.; Yang, Y.; Wang, Q. Drex: Developer recommendation with k-nearest-neighbor search and expertise ranking. In Proceedings of the 2011 18th Asia-Pacific Software Engineering Conference, Washington, DC, USA, 5–8 December 2011; pp. 389–396.
5. Anvik, J.; Hiew, L.; Murphy, G.C. Who should fix this bug? In Proceedings of the 28th international conference on Software Engineering, Shanghai, China, 20–28 May 2006; pp. 361–370.
6. Murphy, G.; Cubranic, D. Automatic bug triage using text categorization. In Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering, Banff, AB, Canada, 20–24 June 2004.
7. Ahsan, S.N.; Ferzund, J.; Wotawa, F. Automatic software bug triage system (bts) based on latent semantic indexing and support vector machine. In Proceedings of the 2009 Fourth International Conference on Software Engineering Advances, Porto, Portugal, 20–25 September 2009; pp. 216–221.
8. Tamrawi, A.; Nguyen, T.T.; Al-Kofahi, J.M.; Nguyen, T.N. Fuzzy set and cache-based approach for bug triaging. In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, Szeged, Hungary, 5–9 September 2011; pp. 365–375.
9. Salton, G. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*; Addison-Wesley: Reading, UK, 1989; Volume 169.
10. Xia, X.; Lo, D.; Ding, Y.; Al-Kofahi, J.M.; Nguyen, T.N.; Wang, X. Improving automated bug triaging with specialized topic model. *IEEE Trans. Softw. Eng.* **2016**, *43*, 272–297. [CrossRef]
11. Mani, S.; Sankaran, A.; Aralikkatte, R. Deeptriage: Exploring the effectiveness of deep learning for bug triaging. In Proceedings of the ACM India Joint International Conference on Data Science and Management of Data, Kolkata, India, 3–5 January 2019; pp. 171–179.
12. Lee, S.R.; Heo, M.J.; Lee, C.G.; Kim, M.; Jeong, G. Applying deep learning based automatic bug triager to industrial projects. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Paderborn, Germany, 4–8 September 2017; pp. 926–931.
13. Zhang, W.; Wang, S.; Wang, Q. KSAP: An approach to bug report assignment using KNN search and heterogeneous proximity. *Inf. Softw. Technol.* **2016**, *70*, 68–84. [CrossRef]
14. Bordes, A.; Usunier, N.; Garcia-Duran, A.; Weston, J.; Yakhnenko, O. Translating embeddings for modeling multi-relational data. In Proceedings of the Advances in Neural Information Processing systems, Lake Tahoe, NV, USA, 5–8 December 2013; pp. 2787–2795.
15. Xia, X.; Lo, D.; Wang, X.; Zhou, B. Accurate developer recommendation for bug resolution. In Proceedings of the 2013 20th Working Conference on Reverse Engineering (WCRE), Koblenz, Germany, 14–17 October 2013; pp. 72–81.
16. Xuan, J.; Jiang, H.; Ren, Z.; Yan, J.; Luo, Z. Automatic bug triage using semi-supervised text classification. *arXiv* **2017**, arXiv:1704.04769.

17. Naguib, H.; Narayan, N.; Brügge, B.; Helal, D. Bug report assignee recommendation using activity profiles. In Proceedings of the 2013 10th Working Conference on Mining Software Repositories (MSR), San Francisco, CA, USA, 18–19 May 2013; pp. 22–30.
18. Zhang, T.; Lee, B. A hybrid bug triage algorithm for developer recommendation. In Proceedings of the 28th Annual ACM Symposium on Applied Computing, Coimbra, Portugal, 18–22 March 2013; pp. 1088–1094.
19. Xie, X.; Zhang, W.; Yang, Y.; Wang, Q. Dretom: Developer recommendation based on topic models for bug resolution. In Proceedings of the 8th International Conference on Predictive Models in Software Engineering, Lund, Sweden, 21–22 September 2012; pp. 19–28.
20. Hu, H.; Zhang, H.; Xuan, J.; Sun, W. Effective bug triage based on historical bug-fix information. In Proceedings of the 2014 IEEE 25th International Symposium on Software Reliability Engineering, Naples, Italy, 3–6 November 2014; pp. 122–132.
21. Shokripour, R.; Anvik, J.; Kasirun, Z.M.; Zamani, S. Why so complicated? Simple term filtering and weighting for location-based bug report assignment recommendation. In Proceedings of the 2013 10th Working Conference on Mining Software Repositories (MSR), San Francisco, CA, USA, 18–19 May 2013; pp. 2–11.
22. Matter, D.; Kuhn, A.; Nierstrasch, O. Assigning bug reports using a vocabulary-based expertise model of developers. In Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories, Vancouver, BC, Canada, 16–17 May 2009; pp. 131–140.
23. Badashian, A.S.; Hindle, A.; Stroulia, E. Crowdsourced bug triaging. In Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), Bremen, Germany, 29 September–1 October 2015; pp. 506–510.
24. Kagdi, H.; Gethers, M.; Poshyvanyk, D.; Hammad, M. Assigning change requests to software developers. *J. Softw. Evol. Process* **2012**, *24*, 3–33. [CrossRef]
25. Linares-Vásquez, M.; Hossen, K.; Dang, H.; Kagdi, H.; Gethers, M.; Poshyvanyk, D. Triaging incoming change requests: Bug or commit history, or code authorship? In Proceedings of the 2012 28th IEEE International Conference on Software Maintenance (ICSM), Trento, Italy, 23–28 September 2012; pp. 451–460.
26. Jeong, G.; Kim, S.; Zimmermann, T. Improving bug triage with bug tossing graphs. In Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, Amsterdam, The Netherlands, 24–28 August 2009; pp. 111–120.
27. Zhang, J.; Xie, R.; Ye, W.; Zhang, Y.; Zhang, S. Exploiting Code Knowledge Graph for Bug Localization via Bi-directional Attention. In Proceedings of the 28th International Conference on Program Comprehension, Seoul, Republic of Korea, 13–15 July 2020; pp. 219–229.
28. Xie, R.; Chen, L.; Ye, W.; Li, Z.; Hu, T.; Du, D.; Zhang, S. DeepLink: A code knowledge graph based deep learning approach for issue-commit link recovery. In Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, 24–27 February 2019; pp. 434–444.
29. Liu, M.; Peng, X.; Marcus, A.; Xing, Z.; Xie, W.; Xing, S.; Liu, Y. Generating query-specific class API summaries. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 26–30 August 2019; pp. 120–130.
30. Zhou, C. Intelligent bug fixing with software bug knowledge graph. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Lake Buena Vista, FL, USA, 4–9 November 2018; pp. 944–947.
31. Lin, Z.Q.; Xie, B.; Zou, Y.Z.; Zhao, J.F.; Li, X.D.; Wei, J.; Sun, H.L.; Yin, G. Intelligent development environment and software knowledge graph. *J. Comput. Sci. Technol.* **2017**, *32*, 242–249. [CrossRef]
32. Abal, I.; Melo, J.; Stănculescu, Ș.; Brabrand, C.; Ribeiro, M.; Wasowski, A. Variability bugs in highly configurable systems: A qualitative analysis. *Acm Trans. Softw. Eng. Methodol. TOSEM* **2018**, *26*, 1–34. [CrossRef]
33. Szurszewski, J. Bug 6447-Inner Class Breakpoints Are not Triggered. 2001 Available online: https://bugs.eclipse.org/bugs/show_bug.cgi?id=6447 (accessed on 1 June 2025).
34. Szurszewski, J. Bug 7004-Removed Specified Step Filters when Deselecting All Buttons on Preference Page. 2001. Available online: https://bugs.eclipse.org/bugs/show_bug.cgi?id=7004 (accessed on 1 June 2025).
35. Bollacker, K.; Evans, C.; Paritosh, P.; Sturge, T.; Taylor, J. Freebase: A collaboratively created graph database for structuring human knowledge. In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, Vancouver, BC, Canada, 10–12 June 2008; pp. 1247–1250.
36. Auer, S.; Bizer, C.; Kobilarov, G.; Lehmann, J.; Cyganiak, R.; Ives, Z. Dbpedia: A nucleus for a web of open data. In *The Semantic Web*; Springer: Berlin/Heidelberg, Germany, 2007; pp. 722–735.
37. Fabian, M.; Gjergji, K.; Gerhard, W. Yago: A core of semantic knowledge unifying wordnet and wikipedia. In Proceedings of the 16th International World Wide Web Conference, WWW, Banff, AB, Canada, 8–12 May 2007; pp. 697–706.
38. Miller, G.A. WordNet: A lexical database for English. *Commun. ACM* **1995**, *38*, 39–41. [CrossRef]

39. Lao, N.; Mitchell, T.; Cohen, W.W. Random walk inference and learning in a large scale knowledge base. In Proceedings of the Conference on Empirical Methods in Natural Language Processing, Edinburgh, UK, 27–31 July 2011; Association for Computational Linguistics: Stroudsburg, PA, USA, 2011; pp. 529–539.
40. Xie, R.; Liu, Z.; Sun, M. Representation Learning of Knowledge Graphs with Hierarchical Types. In Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, New York, NY, USA, 9–15 July 2016; pp. 2965–2971.
41. Yang, B.; Yih, W.t.; He, X.; Gao, J.; Deng, L. Embedding entities and relations for learning and inference in knowledge bases. *arXiv* **2014**, arXiv:1412.6575.
42. Neelakantan, A.; Roth, B.; McCallum, A. Compositional vector space models for knowledge base completion. *arXiv* **2015**, arXiv:1504.06662.
43. Wang, Z.; Zhang, J.; Feng, J.; Chen, Z. Knowledge graph embedding by translating on hyperplanes. In Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, Québec City, QC, Canada, 27–31 July 2014.
44. Lin, Y.; Liu, Z.; Sun, M.; Liu, Y.; Zhu, X. Learning entity and relation embeddings for knowledge graph completion. In Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, Austin, TX, USA, 25–30 January 2015.
45. Ji, G.; He, S.; Xu, L.; Liu, K.; Zhao, J. Knowledge graph embedding via dynamic mapping matrix. In Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), Beijing, China, 26–31 July 2015; pp. 687–696.
46. García-Durán, A.; Bordes, A.; Usunier, N.; Grandvalet, Y. Combining two and three-way embedding models for link prediction in knowledge bases. *J. Artif. Intell. Res.* **2016**, *55*, 715–742. [[CrossRef](#)]
47. Liben-Nowell, D.; Kleinberg, J. The link-prediction problem for social networks. *J. Am. Soc. Inf. Sci. Technol.* **2007**, *58*, 1019–1031. [[CrossRef](#)]
48. Getoor, L.; Diehl, C.P. Link mining: A survey. *Acm Sigkdd Explor. Newsl.* **2005**, *7*, 3–12. [[CrossRef](#)]
49. Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G.S.; Dean, J. Distributed representations of words and phrases and their compositionality. In Proceedings of the Advances in Neural Information Processing Systems, Lake Tahoe, NV, USA, 5–10 December 2013; pp. 3111–3119.
50. Yang, X.; Lo, D.; Xia, X.; Bao, L.; Sun, J. Combining Word Embedding with Information Retrieval to Recommend Similar Bug Reports. In Proceedings of the 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE), Ottawa, ON, Canada, 23–27 October 2016; IEEE: Piscataway, NJ, USA, 2016; IEEE: Ottawa, ON, Canada, 2016; pp. 127–137.
51. Le, Q.; Mikolov, T. Distributed representations of sentences and documents. In Proceedings of the International Conference on Machine Learning, Beijing, China, 21–26 June 2014; pp. 1188–1196.
52. Jiang, Y.; Lu, P.; Su, X.; Wang, T. LTRWES: A new framework for security bug report detection. *Inf. Softw. Technol.* **2020**, 106314. [[CrossRef](#)]
53. Zhang, M.L.; Zhou, Z.H. ML-KNN: A lazy learning approach to multi-label learning. *Pattern Recognit.* **2007**, *40*, 2038–2048. [[CrossRef](#)]
54. Glorot, X.; Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, Sardinia, Italy, 13–15 May 2010; pp. 249–256.
55. Xie, R.; Liu, Z.; Jia, J.; Luan, H.; Sun, M. Representation learning of knowledge graphs with entity descriptions. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, Phoenix, AZ, USA, 12–17 February 2016.
56. Xu, J.; Chen, K.; Qiu, X.; Huang, X. Knowledge graph representation with jointly structural and textual encoding. *arXiv* **2016**, arXiv:1611.08661.
57. Nasim, S.; Razzaq, S.; Ferzund, J. Automated change request triage using alpha frequency matrix. In Proceedings of the 2011 Frontiers of Information Technology, Islamabad, Pakistan, 19–21 December 2011; pp. 298–302.
58. Ye, X.; Bunesco, R.; Liu, C. Learning to rank relevant files for bug reports using domain knowledge. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 16–21 November 2014; pp. 689–699.
59. Fan, M.; Cao, K.; He, Y.; Grishman, R. Jointly embedding relations and mentions for knowledge population. *arXiv* **2015**, arXiv:1504.01683.
60. LeCun, Y.; Bottou, L.; Bengio, Y.; Haffner, P. Gradient-based learning applied to document recognition. *Proc. IEEE* **1998**, *86*, 2278–2324. [[CrossRef](#)]
61. Hochreiter, S.; Schmidhuber, J. Long short-term memory. *Neural Comput.* **1997**, *9*, 1735–1780. [[CrossRef](#)]
62. Srivastava, N.; Hinton, G.; Krizhevsky, A.; Sutskever, I.; Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.* **2014**, *15*, 1929–1958.
63. Duchi, J.; Hazan, E.; Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.* **2011**, *12*, 2121–2159.
64. Xuan, J.; Jiang, H.; Ren, Z.; Zou, W. Developer prioritization in bug repositories. In Proceedings of the 2012 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2–9 June 2012; pp. 25–35.

65. Schein, A.I.; Popescul, A.; Ungar, L.H.; Pennock, D.M. Methods and metrics for cold-start recommendations. In Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Tampere, Finland, 11–15 August 2002; pp. 253–260.
66. Williamson, D.F.; Parker, R.A.; Kendrick, J.S. The box plot: A simple visual method to interpret data. *Ann. Intern. Med.* **1989**, *110*, 916–921. [[CrossRef](#)] [[PubMed](#)]
67. Kohavi, R. A study of cross-validation and bootstrap for accuracy estimation and model selection. In Proceedings of the 14th International Joint Conference on Artificial Intelligence IJCAI, Montreal, QC, Canada, 20–25 August 1995; Volume 14, pp. 1137–1145.
68. Gegick, M.; Rotella, P.; Xie, T. Identifying security bug reports via text mining: An industrial case study. In Proceedings of the 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), Cape Town, South Africa, 2–3 May 2010; pp. 11–20.
69. Bhattacharya, P.; Neamtiu, I. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In Proceedings of the 2010 IEEE International Conference on Software Maintenance, Timioara, Romania, 12–18 September 2010; pp. 1–10.
70. Lei, C.; Liu, D.; Li, W.; Zha, Z.J.; Li, H. Comparative deep learning of hybrid representations for image recommendations. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 27–30 June 2016; pp. 2545–2553.
71. Wijayasekara, D.; Manic, M.; Wright, J.L.; McQueen, M. Mining bug databases for unidentified software vulnerabilities. In Proceedings of the 2012 5th International Conference on Human System Interactions, Perth, Australia, 6–8 June 2012; pp. 89–96.
72. Banerjee, S.; Syed, Z.; Helmick, J.; Culp, M.; Ryan, K.; Cukic, B. Automated triaging of very large bug repositories. *Inf. Softw. Technol.* **2017**, *89*, 1–13. [[CrossRef](#)]
73. Badashian, A.S.; Hindle, A.; Stroulia, E. Crowdsourced bug triaging: Leveraging q&a platforms for bug assignment. In Proceedings of the International Conference on Fundamental Approaches to Software Engineering, Eindhoven, The Netherlands, 4–7 April 2016; pp. 231–248.
74. Sajedi-Badashian, A.; Stroulia, E. Guidelines for evaluating bug-assignment research. *J. Softw. Evol. Process* **2020**, *32*, e2250. [[CrossRef](#)]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.