DeepLink: A Code Knowledge Graph Based Deep Learning Approach for Issue-Commit Link Recovery

Rui Xie[1,2], Long Chen[1,2], Wei Ye[1]*, Zhiyu Li[1,2], Tianxiang Hu[1,2], Dongdong Du[1,3], and Shikun Zhang[1]
[1]National Engineering Research Center for Software Engineering, Peking University, Beijing 100871, China
[2]School of Software and Microelectronics, Peking University, Beijing 100871, China
[3]School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China
{ruixie, clcmlxl, wye, lizhiyu, hutianyang, dudongdong, zhangsk}@pku.edu.cn

***Abstract*—Links between issue reports and corresponding code commits to fix them can greatly reduce the maintenance costs of a software project. More often than not, however, these links are missing and thus cannot be fully utilized by developers. Current practices in issue–commit link recovery extract text features and code features in terms of textual similarity from issue reports and commit logs to train their models. These approaches are limited since semantic information could be lost. Furthermore, few of them consider the effect of source code files related to a commit on issue–commit link recovery, let alone the semantics of code context. To tackle these problems, we propose to construct code knowledge graph of a code repository and generate embeddings of source code files to capture the semantics of code context. We also use embeddings to capture the semantics of issue- or commit-related text. Then we use these embeddings to calculate semantic similarity and code similarity using a deep learning approach before training a SVM binary classification model with additional features. Evaluations on real-world projects show that our approach DeepLink can outperform the state-of-the-art method.**

***Index Terms*—Issue–Commit Link, Semantic Similarity, Code Knowledge Graph, Code Context, Code Embeddings, Deep Learning**

I. Introduction

Issue tracking and code version control are two important activities in the software development life cycle and are often inseparable in practice. For example, an issue is commonly fixed by one or more commits. However, in a traditional software development environment, the issue tracking system and the code version control system are usually two independent systems, which separates these two activities. While developers can link issues to code commits by adding issue IDs to the commit logs on a code version control platform such as Github, it is hard to enforce such practice in an online open-source development community. Studies [1], [2] have shown that most fixed issues on Github are not linked to any commit. Therefore, there is a large number of links missing between issues and commits.

The links between issues and commits, often shortened as issue–commit links, play an important role in software maintenance and are widely used in many maintenance activities. For example, in commit analysis [2], corresponding issues can help better understand the logic and purpose of a commit. Moreover, in bug prediction [3], issue–commit links can help gather more training data and thus increase prediction accuracy. However, manual maintenance of issue–commit links is inefficient and prone to link missing or link error due to human errors. Therefore, developers would like to construct and maintain issue–commit links automatically.

Current practices in issue–commit link recovery calculate the textual similarity between issue and commit and train a classifier to predict whether there is a link between a pair of issue and commit or not. For example, ReLink [3] extracts textual similarity features between issue descriptions and commit logs; MLink [4] extracts textual similarity features between issue descriptions and commit-related source code files; FRLink [5] and PULink [6] extract textual similarity features from both source code files and non-source files. But unlike FRLink, PULink constructs the link recovery model with positive and unlabeled links rather than positive and negative links to increase data efficiency. In comparison, PULink is the state-of-the-art method with the best performance.

However, there are at least two limitations in existing literatures: 1. The semantic information cannot be fully captured via textual similarity; 2. Code context—the related code information in a commit—has not been fully exploited.

Go beyond textual similarity. Semantic information contains key information that can aid the learning process of automatic issue–commit link recovery, yet textual similarity features are commonly used in issue–commit link recovery. Semantic information may be lost using such approaches. Only a few researchers try to go beyond textual similarity. For example, [7] infers links based on a semantic level of confidence; FRLink [5] extracts textual similarity features using Vector Space Model (VSM) [8], which contain some semantic information. We argue that semantic information can be better captured using a deep learning approach.

The number of researchers working on natural language processing (NLP)—especially those who use deep learning approaches—has greatly increased over the past five years.

*Corresponding author.

Researchers using deep neural networks have claimed state-of-the-art performance in many NLP tasks [9], and semantic information from text plays an important role in these algorithms. For example, CNN [10] [11], LSTM [12] [13], recursive neural network [14] and highway network [15] are widely used to extract semantic information from text and have achieved great performance in many NLP tasks. It is conceivable that semantic information from text can also improve issue–commit link recovery and deep learning is currently the best way to capture such semantics.

Take code context into consideration. Most existing approaches extract textual similarity features between issue reports and commit logs and thus fail to consider code context, which is the related code information for a given commit. For example, although the tendency that "user register" is related to "RegisterController" may be learned by existing approaches, a large amount of related classes referenced by "RegisterController" are ignored such as "PasswordUtils", which will be invoked in "RegisterController" when a password is changed. From this perspective, the semantics of code context can be a valuable source of information in issue–commit link recovery.

To capture the information contained in source code, [29] uses NP-CNN to encode both lexical and program structure information from source code. While NP-CNN can effectively learn source code embeddings, it is not sufficient to capture the semantics of code context since it does not consider the interrelationships among code entities. We propose to use knowledge graph to represent code context and learn code context embedding based on code knowledge graph.

Knowledge graph is a graph made up of nodes and directed edges that is used to represent domain knowledge [16]. Based on this framework, [17] proposed software knowledge graph, which is defined as a graph representing relevant knowledge in software domains, projects, and systems. In a software knowledge graph, nodes represent software knowledge entities (e.g., classes, issue reports and business concepts), and directed edges represent various relationships between these entities (e.g., method invocation and traceability link). We limit the software knowledge to code knowledge in which nodes represent code entities (e.g., classes, methods and properties) and edges represent relationships between code entities (e.g., method invocation and class declaration). We believe code knowledge graph is an appropriate way to represent code context.

Similar to words, code entities should also be represented as embeddings before they can be used in deep learning based algorithms. TransE [18] is a classic approach to encode nodes and edges in an knowledge graph into corresponding embeddings. And based on TransE, a lot of approaches such as TransH [19], TransR [20] and TransD [21] have been proposed to improve the performance of knowledge graph embedding. Using these knowledge graph embedding methods, we can incorporate code embeddings, which encode the semantic information of code context, into our model to improve the performance of automatic issue–commit link recovery.

Based on the above observations, we propose DeepLink, a code knowledge graph based deep learning approach for issue–commit link recovery. It uses deep learning to learn distributed representations or embeddings of issue- or commit-related texts as an enhancement to textual similarities. It also builds a code knowledge graph for each code repository and uses knowledge graph embedding method to learn embeddings of source code files, which are then used alongside those text embeddings in predicting a potential link between a pair of issue and commit. In this way, not only the information of code context pertaining to a commit is utilized, but also the semantics rather than mere textual information of the source code files are captured.

The major contributions of our work include:

- We propose a novel approach called DeepLink to solve the issue–commit link recovery problem, which uses advanced technologies including deep learning and knowledge graph and thus goes beyond textual similarities to semantic similarities. Experiments show that DeepLink can outperform the state-of-the-art approach.
- To the best of our knowledge, DeepLink is the first deep learning approach to this problem.
- We are also the first to propose using code knowledge graph to incorporate the semantics of code context into the issue–commit link recovery model and thus improve its performance.

The rest of this paper is organized as follows. We explain our motivation in Section II. Section III describes the details of our approach. Section IV presents experiments that evaluate our approach against the state-of-the-art. Section V summarizes related works on issue–commit link recovery and knowledge graph embedding. Section VI concludes the paper.

II. Motivation

Oftentimes, we can reconstruct the link between an issue and a commit via the logs of the commit and the text related to the issue such as its title, description and comments. For example, in Table I, "ZXingMIDlet" and "ZXingMidlet" appear in both the commit log and the issue's comment (Comment 3). By analyzing the similarity of the two, it's safe to say that there may be a link between them. Traditional issue–commit link recovery models usually extract text features in terms of textual similarities. FRlink [5] uses VSM to extract text features of either an issue or a commit, and then determines their similarity by calculating the cosine similarity of their vectors. The corresponding formula is as follows, where t is a word in the document, q and d are two kinds of document, and D is the collection of all documents:

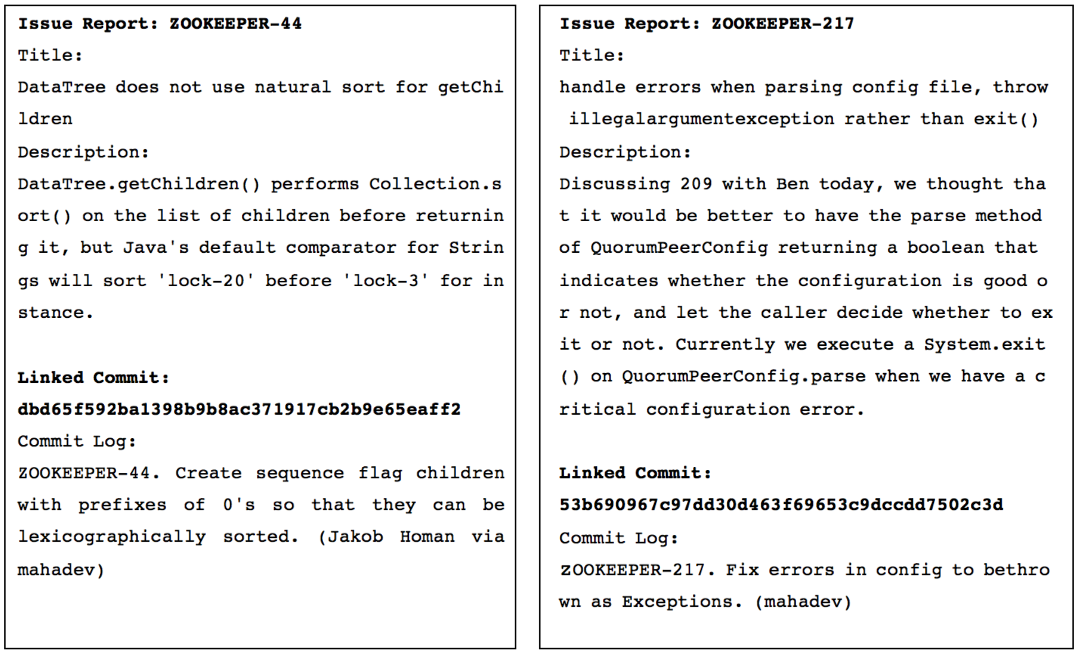
$$\omega(t,d,D) = tf(t,d) \times idf(t,d) \tag{1}$$

$$idf(t,d) = log\frac{|D|}{|\{d \in D : t \in d\}|} \tag{2}$$

$$cos(q,d) = \frac{\sum_{t \in D} \omega(t,q) \times \omega(t,d)}{\sqrt{\sum_{t \in q} \omega(t,q)^2} \times \sqrt{\sum_{t \in d} \omega(t,d)^2}} \tag{3}$$

TABLE I
AN EXAMPLE OF COMMIT LOG AND ITS CORRESPONDING ISSUE REPORT

Source	Created Time	Content
Commit log	2008.01.22	Name of midlet is “ZXingMIDlet”, not “ZXingMidlet”!
Issue Title	2008.01.11	Issue 19: does not work on nokia 5300
Comment 1	2008.01.11	Which version, regular or basic?
Comment 2	2008.01.12	I tested both of them (regular and basic) but in both case it just make an exception
Comment 3	2008.01.12	...The class is called “ZXingMIDlet” but the exception mentions “ZXingMidlet” (note different capitalization). It looks like the manifest file I wrote gets this wrong. ...



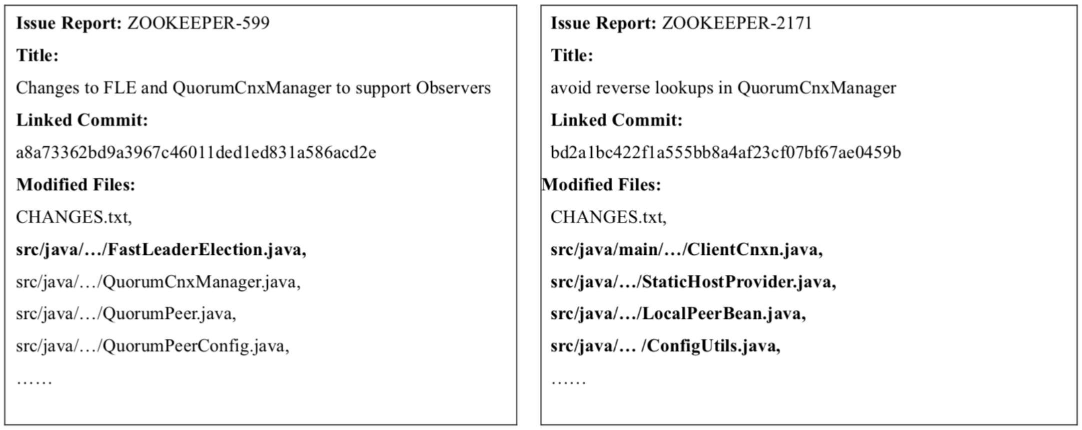
Issue Report: ZOOKEEPER-44
Title:
DataTree does not use natural sort for getChi
ldren
Description:
DataTree.getChildren() performs Collection.s
ort() on the list of children before returnin
g it, but Java's default comparator for Strin
gs will sort 'lock-20' before 'lock-3' for in
stance.

Linked Commit:
dbd65f592ba1398b9b8ac371917cb2b9e65eaff2
Commit Log:
ZOOKEEPER-44. Create sequence flag children
with prefixes of 0's so that they can be
lexicographically sorted. (Jakob Homan via
mahadev)

Issue Report: ZOOKEEPER-217
Title:
handle errors when parsing config file, throw
illegalargumentexception rather than exit()
Description:
Discussing 209 with Ben today, we thought tha
t it would be better to have the parse method
of QuorumPeerConfig returning a boolean that
indicates whether the configuration is good o
r not, and let the caller decide whether to ex
it or not. Currently we execute a System.exit
() on QuorumPeerConfig.parse when we have a c
ritical configuration error.

Linked Commit:
53b690967c97dd30d463f69653c9dccdd7502c3d
Commit Log:
ZOOKEEPER-217. Fix errors in config to bethro
wn as Exceptions. (mahadev)

Fig. 1. Two true issue–commit pairs that have low textural similarity.



Issue Report: ZOOKEEPER-599
Title:
Changes to FLE and QuorumCnxManager to support Observers
Linked Commit:
a8a73362bd9a3967c46011ded1ed831a586acd2e
Modified Files:
CHANGES.txt,
src/java/…/FastLeaderElection.java,
src/java/…/QuorumCnxManager.java,
src/java/…/QuorumPeer.java,
src/java/…/QuorumPeerConfig.java,
……

Issue Report: ZOOKEEPER-2171
Title:
avoid reverse lookups in QuorumCnxManager
Linked Commit:
bd2a1bc422f1a555bb8a4af23cf07bf67ae0459b
Modified Files:
CHANGES.txt,
src/java/main/…/ClientCnxn.java,
src/java/…/StaticHostProvider.java,
src/java/…/LocalPeerBean.java,
src/java/… /ConfigUtils.java,
……

Fig. 2. Two issue reports about QuorumCnxManager with modified files in their corresponding commits.

Issue–commit link recovery via features of textual similarity can achieve decent performance in evaluations. However, in the process of similarity calculation, a lot of information would be lost, leading to link missing or link error. As shown in Figure 1, issue report 44 and its corresponding commit both have the same meaning in semantics, but their textual similarity is low, which is 0.24. It is the same for issue report 217 and its corresponding commit, whose textual similarity is only 0.27 in spite of shared semantics. In order to preserve the semantic information of issues and commits to the maximum extent, we propose to represent text as a real-value vector or embedding and use deep neural networks to capture semantic information.

Apart from text features, code features are also a key feature source in issue–commit link recovery. For example, in the left box of Figure 2, the code element "QuorumCnxManager" appears in the issue description. Judging from experience, it is easy to see that the issue is rooted in the class file related to "QuorumCnxManager," in this case, "QuorumCnxManager.java." Therefore, if a commit modifies this file, chances are that there is a link between the commit and the issue. Based on this observation, FRLink uses regular expressions to find all the code elements in the text, generates two new documents for the issue and the commit respectively by putting corresponding code elements together, and use Equation 1–3 to calculate their code similarity. However, this calculation of code similarity does not take into account the code context of the commit. For example, as shown in Figure 2, although only "QuorumCnxManager" exists in the descriptions of the issues, the modified files of the commits include not only "QuorumCnxManager" but also other classes like "FastLeaderElection" and "StaticHostProvider," which are not similar to "QuorumCnxManager" at all. These classes may reference "QuorumCnxManager" or be referenced by "QuorumCnxManager" directly or indirectly. Thus, for the issue–commit pair in the right box of Figure 2, the link cannot be recovered by FRLink because their code similarity is low. To tackle this problem, we propose to incorporate the semantics of code context into the link recovery model.

III. Proposed Approach

In this section, we present our link recovery approach DeepLink in detail. DeepLink solves the issue–commit link recovery problem by constructing a code knowledge graph and learning embeddings using knowledge graph embedding method. Figure 3 shows the overall framework of DeepLink. The main process of DeepLink includes:

- **Data Construction** This process compares each issue in the issue tracking system with each commit in the code repository, and constructs a potential link for a pair of issue and commit if certain conditions are satisfied.
- **Text to Embedding** This process converts each issue to "issue text embedding," and each commit as "commit text embedding," which encodes the semantic information of the issue or commit.
- **Code to Embedding** This process converts the code context of each commit to "code context embedding," which encodes the semantic information of the corresponding source code files.
- **Similarity Calculation** This process calculates the semantic similarity and code similarity so that the semantic information and code context information can be used in the link prediction model easily.
- **Feature Extraction** This process extracts features from potential links as a feature vector, which is one of the inputs of the link prediction model.
- **Link Prediction** This process takes semantic similarity, code similarity and the feature vector as inputs and trains a model for link prediction.
- **Missing Link Recovery** This process will recover the missing links from unlabeled links.

A. Data Construction

There are usually thousands of issue reports in an issue tracking system and thousands of commits in a code repository. As a result, there are millions of links between all issues and all commits. Thus it is necessary to filter the links before feeding them into DeepLink. In the data construction process, each issue is compared with all commits in the code repository. If the commit time is seven days before or after the issue resolved time, we construct a potential link between the pair. This is a common practice as in [23]. In addition, if the issue ID is recorded in the commit log, the corresponding potential link is labeled as positive link, and other potential link which contains the same commit is labeled as negative link. The positive links and negative links compose the training data, while the unlabeled links are used for missing link recovery.

B. Text to Embedding

All text information related to an issue including title, description and comments are grouped together to form a new "issue document." Similarly, "commit document" consists of commit logs and all non-source files the commit modifies. We train a CBOW model on all documents to get word embeddings using Word2Vec [25] so that each document can be represented as a sequence of embeddings:

$$d_{issue} = \{w_1, w_2, ..., w_{|d_{issue}|}\} \tag{4}$$

$$d_{commit} = \{w_1, w_2, ..., w_{|d_{commit}|}\} \tag{5}$$

We use GRU [24] to encode a sequence of word embeddings into a fixed-length embedding. Recently, it has been shown that GRU performs equally well across a range of tasks compared to other RNN architectures, such as LSTM [26]. The hidden state of the GRU at each time step t is computed as:

$$z_t = \sigma(W_z[h_{t-1}, x_t]) \tag{6}$$

$$r_t = \sigma(W_r[h_{t-1}, x_t]) \tag{7}$$

$$\hat{h}_t = tanh(W[r_t * h_{t-1}, x_t]) \tag{8}$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \hat{h}_t \tag{9}$$

where σ is the sigmoid function $\sigma(x) \in [0, 1]$, h_{t-1} is the previous hidden state, x_t is the current word embedding, and variables W_z, W_r, W are the parameters of GRU. We feed the sequence of word embeddings of issue document and commit document into GRU respectively to get issue text embedding and commit text embedding as part of the inputs for similarity calculation.

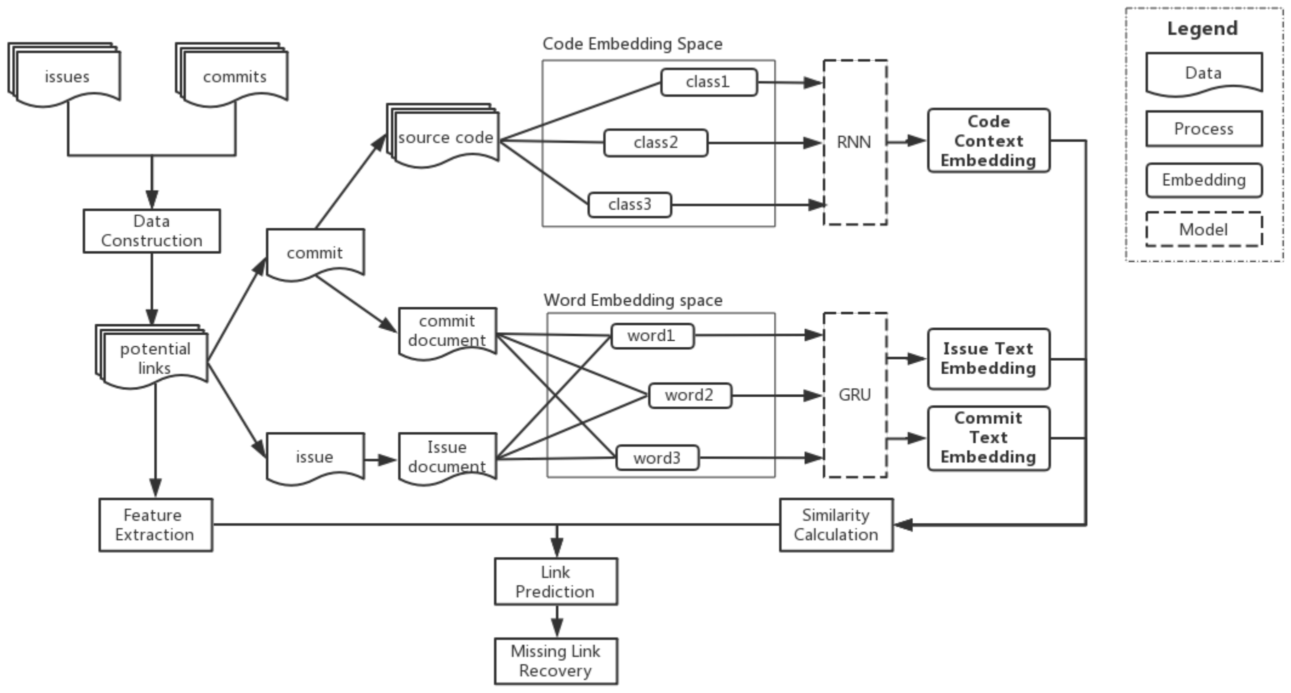


Fig. 3. Overall framework of DeepLink.

C. Code to Embedding

Firstly we learn embeddings of each code entity (class, property, method, parameter and variable) in the code repository based on code knowledge graph. Code knowledge graph is a graph representation of source code which can be constructed via abstract syntax tree (AST). In a code knowledge graph $G = (E, V)$, each vertice $v_i \in V$ corresponds to a code entity in the code repository and each edge $e_k \in E$ corresponds to the relation between two vertices, including “inheritance” (between class and class), “has”(between class and property, class and method, method and parameter, method and variable), “instance_of” (between property and class, variable and class), “return_type” (between method and class) and “call” (between method and method). Figure 4 shows the model of code knowledge graph that we have defined.

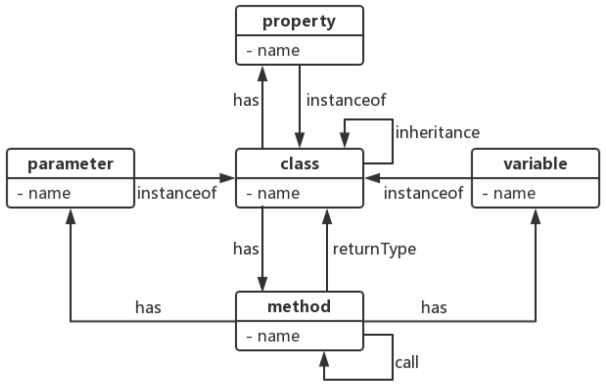


Fig. 4. Model of Code Knowledge Graph

A knowledge graph embedding method is used to build code entity embeddings, which regards a relation as a translation from head entity to tail entity. In a code knowledge graph, a relation pair can be represented as a triplet (h, r, t), where h is the head entity, t is the tail entity, and r is the relation between the head entity and tail entity. The embeddings H, R, T are used to represent the triplet (h, r, t). To learn such embeddings, we minimize a margin-based ranking criterion over the training set:

$$\zeta = \sum_{(h,r,t)\in S} \sum_{(h',r,t')\in S'} [\gamma + f_r(h,t) - f_r(h',t')]_+ \tag{10}$$

where S is all triplets in the code knowledge graph, S' is the negative triplet sets generated from (h, r, t), and $[x]_+$ is the positive part of x, i.e. $max(0, x)$. $f_r(h, t)$ is the score function for h and t under relation r. $f_r(h, t)$ is different for different knowledge graph embedding methods, and we have tried TransE [18], TransH [19] and TransR [20] as the knowledge graph embedding method to build code entity embeddings and the comparison of their performances will be discussed in Section IV.

The code information in each commit can be considered as a sequence of corresponding code embeddings, i.e., $commit = \{E_{code1}, E_{code2}, ..., E_{coden}\}$, which will be used to generate the code context embedding via the basic RNN model. As there are a large nunmber of code entities in commits, we only use class embeddings to represent the commit to simplify the computation. The hidden state at each time step t of RNN is computed as:

$$h_t = tanh(W[h_{t-1}, x_t]) \tag{11}$$

where h_{t-1} is the previous state, x_t is the current code embedding, and variables W are the parameters of RNN. The code context embedding of the commit is another part of the inputs for similarity calculation.

D. Similarity Calculation

Once issue text embedding, commit text embedding and code context embedding are all ready, we can calculate the semantic similarity between issue text embedding and commit text embedding, and the code similarity between issue text embedding and code context embedding. Traditional similarity functions like cosine similarity do not apply in this case as each of these three embeddings belongs to independent embedding spaces. We consider this problem as a binary classification problem and use MLP to build our similarity calculation model. The corresponding label is 1 if there is a link between the issue and commit pair and 0 otherwise. We regard the probability of having label 1 as the corresponding similarity. Semantic similarity and code similarity are used as inputs in the link prediction process.

E. Feature Extraction

The metadata pertaining to a pair of issue and commit are first put into a feature vector, including issue type, issue priority, issue reporter, issue assignee, issue created time, issue resolved time, commit time and committer. We then calculate the similarity values in terms of textual similarity for potential links, which are added to the feature vector. Detailed steps to calculate textual similarity can be found in FRLink [5]. The feature vectors are also used as inputs in the link prediction process.

F. Link Prediction

Once the inputs are all ready, we can use the positive links and negative links as training data to train a SVM binary classification model for link prediction. The score function of the SVM classifier can be written as:

$$g(issue_i, commit_j) = W[s_{i,j}; c_{i,j}; v_{i,j}] \tag{12}$$

where $s_{i,j}$ stands for the semantic similarity, $c_{i,j}$ stands for the code similarity, $v_{i,j}$ stands for the feature vector between $issue_i$ and $commit_j$, and W is the parameters of the classifier. $g(issue_i, commit_j)$ should be positive if the potential link is a true link, and negative if otherwise. To learn the parameters, we minimize the objective function ζ over the whole training set:

$$\zeta = ||W||^2 + \sum_{(issue_i, commit_j) \in L} [1 - y_{i,j} g(issue_i, commit_j)]_+ \tag{13}$$

in which L represents all potential links, $y_{i,j}$ is the ground truth label, and $[x]_+$ means $max(0, x)$.

G. Missing Link Recovery

As mentioned in the data construction process, we have a large number of unlabeled links and many missing issue–commit links are hidden in them. With the model we have learned in the link prediction process, missing issue–commit links in all unlabeled links can be reconstructed.

Note that our approach is not an end-to-end deep learning approach: we first use labeled data to train MLPs for similarity calculation and then use the outputted semantic similarity score and code similarity score alongside other features as inputs to the subsequent SVM model for link prediction. We believe such an approach is more interpretable and manageable than the end-to-end deep learning approach. We did try using a deep neural network in place of SVM but it only generated less satisfactory results.

IV. Experiments & Results

A. Subject Projects

We investigated six active Apache Software Foundation projects, namely ZOOKEEPER [1], MAHOUT [2], CHUKWA [3], AVRO [4], LANG [5] and TEZ [6] as shown in Table II. Column "# Fixed Issues" represents the number of issues whose status is "RESOLVED" in the issue tracking system. Column "# Commits" represents the number of commits in the master branch of the code repository. Column "# Links" represents the number of positive links that can be found in the issue tracking system and code repository. Column "Studied period" represents the time span of the information we have collected from the project. The link information in these six projects are well maintained. For example, in project TEZ, there are 2503 positive links among 2901 issues and 2574 commits, meaning that almost 97% commits have one link to an issue at least.

B. Experiment Settings

The issue IDs in commit logs and the commit IDs in issue-related texts were removed first for fear that our model may mainly learn from such information, which makes the link recovery problem much easier, and thus perform poorly when in lack of such information.

To evaluate the performance of our model more accurately, we used 10-fold cross-validation when training the model for

[1]https://zookeeper.apache.org/
[2]http://mahout.apache.org/
[3]http://chukwa.apache.org/
[4]http://avro.apache.org/
[5]https://commons.apache.org/
[6]http://tez.apache.org/

TABLE II
SUBJECT PROJECT STATISTICS

Project	# Fixed Issues	# Commit	# Links	Studied period
ZOOKEEPER	1594	1719	1513	2008.6–2018.4
MAHOUT	1386	3925	1921	2008.6–2018.4
CHUKWA	819	847	718	2007.9–2018.4
AVRO	1511	1607	1398	2009.4–2018.4
LANG	1767	5114	1178	2003.8–2018.4
TEZ	2901	2574	2503	2013.3–2018.4

each project. Positive and negative links are divided into 10 equally sized subsets randomly. To avoid the negative effect of having too many negative links, we removed negative links in each subset until the positive links and negative links in the subset had the same amount. Each time we used nine subsets as the training set and the other one as the validation set. The validation process was repeated 10 times and the average results were reported for evaluation.

As the vocabulary of issue documents and commit documents of a project is more restricted than that of open domains, we chose 100 as the dimension of text embedding. Besides, the scale of our training data is also relatively small. Experiment results show that the chosen embedding size is sufficient to capture the semantic information for issue-link recovery.

As TransR is more effective to encode code entities into code embeddings than TransH and TransE according to the experiment results, we chose TransR as our default code knowledge graph embedding method. We pretrained the code embeddings and used them directly in the subsequent model. We chose 50 as the dimension of the code embeddings.

C. Evaluation Metrics

To verify the experiment results, precision, recall and F1-measure are calculated in the validation process. These metrics are calculated as follows:

$$Precision = \frac{TP}{TP + FP} \tag{14}$$

$$Recall = \frac{TP}{TP + FN} \tag{15}$$

$$F1 - Measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{16}$$

The variable definitions in the formula are as follows:

- TP: the Number of positive links correctly predicted.
- TN: the Number of negative links correctly predicted.
- FP: the Number of negative links incorrectly predicted.
- FN: the Number of positive links incorrectly predicted.

D. Research Questions

1) RQ_1: How effective is DeepLink in recovering missing links between issue reports and commits in all six projects?

To show the effectiveness of our approach, we evaluated the performance of DeepLink in each project using dataset, settings and metrics described earlier.

2) RQ_2: How does the semantic information affect the performance of DeepLink?

We have contended that the semantic information could improve the performance of link recovery. To prove this, we removed the semantic similarity from the input features of DeepLink and implemented a simplified model DeepLink NoText. We compare these two models based on the aforementioned metrics.

3) RQ_3: How does the code context affect the performance of DeepLink?

To demonstrate the effectiveness of code context embedding generated from code knowledge graph, we removed the code similarity from the input features of DeepLink and implement a simplified model DeepLink NoCode. We compare these two models based on the aforementioned metrics.

4) RQ_4: Which of the knowledge graph embedding methods best discriminates true links from the other ones?

We implemented three models DeepLink TransE, DeepLink TransR, DeepLink TransH which used TransE, TransR, TransH respectively. We compare these three models based on the aforementioned metrics.

5) RQ_5: How effective is our DeepLink approach compared to the state-of-the-art?

There are several approaches for recovering missing issuecommit links, among which PULink [6] is the state-of-the-art. We compare DeepLink to PULink based on the aforementioned metrics.

E. Experiment Results

1) RQ_1: How effective is DeepLink in recovering missing links between issue reports and commits in all six projects?

We use the average precision, recall, and F1-Measure of 10-fold cross-validation to evaluate the effectiveness of DeepLink in recovering missing links of the six subject projects. As shown in Table III, the precision, recall and F1-Measure of DeepLink are all higher than 75% in all six projects. Particularly, in Project ZOOKEEPER, the precision, recall and F1-Measure of DeepLink are higher than 90%.

To see the performance trend more clearly, we further draw the Precision-Recall curve (PR curve) of DeepLink in each subject project, as shown in Figure 5. To get the PR curve, we first normalize the output of the SVM binary classifier (using l2 normalization) and vary the threshold of the classifier from -1 to 1. Note that the normalization step does not affect the performance measurement of the original model which we can say has a threshold of 0. As shown in Figure 5, the recall of DeepLink is higher than 70% in all six projects, and its precision is even 100% in the project ZOOKEEPER when the corresponding recall does not exceed 75%.

According to the experiment results, DeepLink can effectively accomplish the missing link recovery task.

2) RQ_2: How does the semantic information affect the performance of DeepLink?

We evaluate the effect of semantic information by comparing the performance of DeepLink with that of DeepLink NoText, which is implemented by excluding the semantic

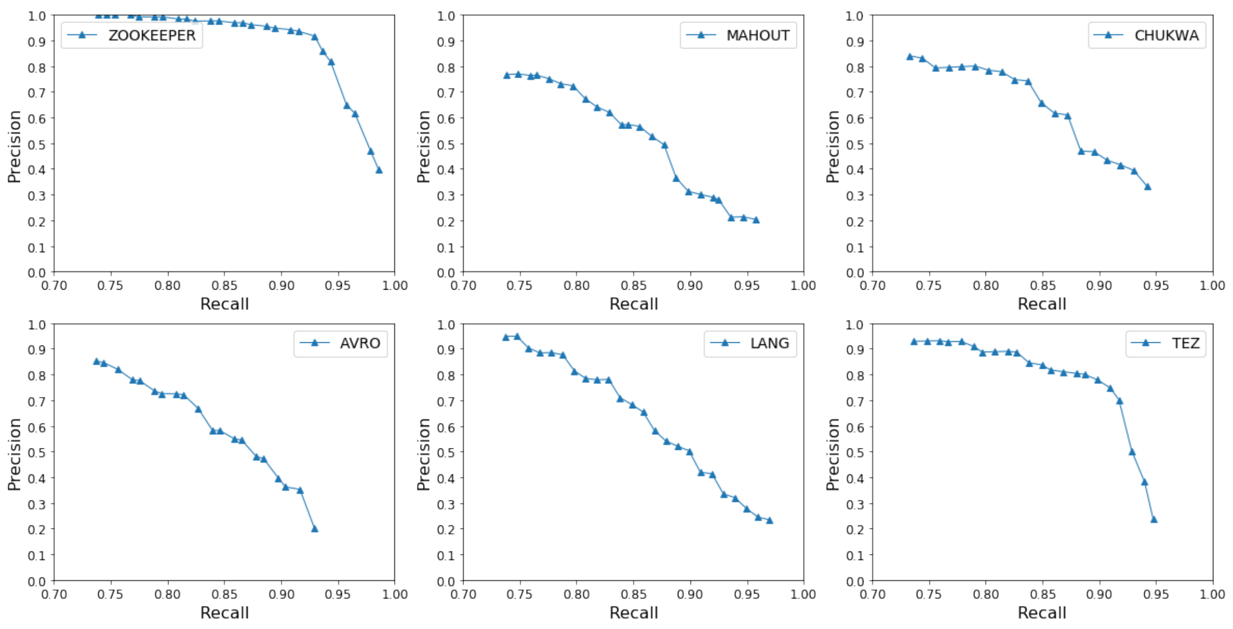


Fig. 5. Performance of DeepLink in each subject project.

TABLE III
PERFORMANCE COMPARISON BETWEEN DEEPLINK, DEEPLINK NOTEXT AND DEEPLINK NOCODE

Projects	**DeepLink**			**DeepLink NoText**			**DeepLink NoCode**		
	Precision	Recall	F1-measure	Precision	Recall	F1-measure	Precision	Recall	F1-measure
ZOOKEEPER	93.53%	91.55%	92.53%	92.81%	88.36%	90.53%	87.41%	85.51%	86.45%
MAHOUT	76.47%	76.47%	76.47%	76.82%	65.54%	70.73%	72.56%	69.59%	71.04%
CHUKWA	77.78%	81.4%	79.55%	90.74%	62.03%	73.68%	71.25%	72.15%	71.7%
AVRO	84.67%	74.36%	79.18%	75.37%	71.63%	73.45%	66.67%	70.83%	68.69%
LANG	94.87%	74.75%	83.62%	91.0%	73.98%	81.61%	82.29%	73.83%	77.83%
TEZ	88.71%	82.71%	85.6%	91.27%	78.77%	84.56%	90.74%	78.71%	84.3%

TABLE IV
PERFORMANCE COMPARISON BETWEEN DEEPLINK TRANSR, DEEPLINK TRANSH AND DEEPLINK TRANSE

Projects	**DeepLink TransR (DeepLink)**			**DeepLink TransH**			**DeepLink TransE**		
	Precision	Recall	F1-measure	Precision	Recall	F1-measure	Precision	Recall	F1-measure
ZOOKEEPER	93.53%	91.55%	92.53%	93.2%	90.73%	91.95%	93.02%	83.33%	87.91%
MAHOUT	76.47%	76.47%	76.47%	72.68%	74.5%	73.58%	81.44%	69.74%	75.14%
CHUKWA	77.78%	81.4%	79.55%	88.89%	62.5%	73.39%	75.34%	72.37%	73.83%
AVRO	84.67%	74.36%	79.18%	76.74%	72.79%	74.72%	79.41%	61.36%	69.23%
LANG	94.87%	74.75%	83.62%	86.41%	75.42%	80.54%	94.12%	68.97%	79.6%
TEZ	88.71%	82.71%	85.6%	91.75%	78.75%	84.75%	90.23%	79.84%	84.72%

similarity between issue text embedding and commit text embedding from the input features of the link prediction classifier with the other processes remaining the same. The detailed evaluation results in all six subject projects are summarized in Table III, and it shows that the F1-Measure decreases in all six projects from DeepLink to DeepLink NoText. And in three out of six projects, the F1-Measure has declined more than 5%.

We can further compare the PR curve of DeepLink and DeepLink in terms of AUC (areas under the curve). The larger the AUC, the better the classifier. As shown in Figure 6, in which the PR curves are derived from the project AVRO, there is a distinct performance decline in DeepLink NoText.

The experiment results indicate that the semantic information is very important for automatic link recovery.

3) *RQ_3: How does the code context affect the performance of DeepLink?*

We have shown that there are some files that are related to "QuorumCnxManager" but it is not possible to recognize them via textual similarity in Section II. After training code embeddings, we searched in the code embedding space and tried to locate the top 10 code entities that are nearest to "QuorumCnxManager." The result is summarized in TABLE

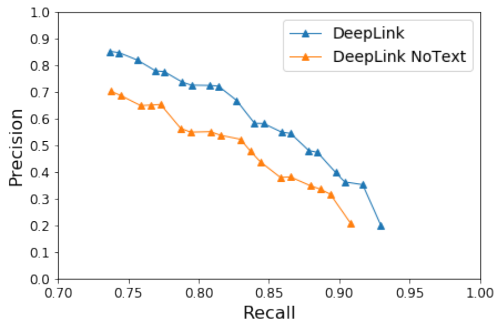


Fig. 6. Performance comparison between DeepLink and DeepLink NoText in AVRO.

TABLE V
TOP 10 ENTITIES NEAREST TO QuorumCnxManager

Index	Entity	Distance
1	**ClientCnxn**	0.3011
2	ZooInspectorN	0.3072
3	**StaticHostProvider**	0.3349
4	ZooKeeperAdmin	0.3383
5	**QuorumPeerConfig**	0.3400
6	Login	0.3412
7	**QuorumPeer**	0.3491
8	FileTxnLog	0.3544
9	RestCfg	0.3714
10	AtomicFileOutputStream	0.3759

V. Via code embeddings, we are able to find related source code files such as "ClientCnxn," "StaticHostProvider," "QuorumPeerConfig" and "QuorumPeer," all of which cannot be found using existing approaches. This indicates that the code embeddings learned from code knowledge graph can capture the code context information effectively and thus are important for performance improvement in automatic link recovery.

To further evaluate the effect of code context embedding, we compare the performance of DeepLink with that of DeepLink NoCode, which, as the name suggests, does not have the code similarity fed into the link prediction model with the other processes remaining the same. As shown in TABLE III, DeepLink outperforms DeepLink NoCode in all three metrics across all six projects. What's more, the performance of DeepLink NoText is better than that of DeepLink NoCode in terms of F1-Measure in five out of six projects. The experiment results indicate that the code context embedding plays an important role in issue-commit link recovery. Therefore, our approach to encode the semantics of code context via code knowledge graph is very effective.

4) RQ_4: Which of the knowledge graph embedding methods best discriminates true links from the other ones?

We evaluate which of the knowledge graph embedding methods is the most effective in missing link recovery by comparing the performance among DeepLink TransR, DeepLink TransH and DeepLink TransE. These three models are implemented via the corresponding knowledge graph embedding method with the other processes remaining the same. As shown in the TABLE IV, the performance of DeepLink TransR is better than those of DeepLink TransR and DeepLink TransE in terms of F1-Measure across all six projects. As TransE has issues modeling N-to-1, 1-to-N and N-to-N relations, DeepLink TransE has the worst performance among the three methods. The experiment results indicate that TransR is the most effective one and thus we choose TransR as our default knowledge graph embedding method in our DeepLink approach.

TABLE VI
Performance of PULink

Project	Precision	Recall	F1-measure
ZOOKEEPER	84.78%	87.07%	85.9%
MAHOUT	67.67%	74.97%	71.01%
CHUKWA	60.52%	74.65%	66.76%
AVRO	73.39%	74.6%	73.89%
LANG	90.59%	75.92%	81.68%
TEZ	88.8%	80.6%	84.46%

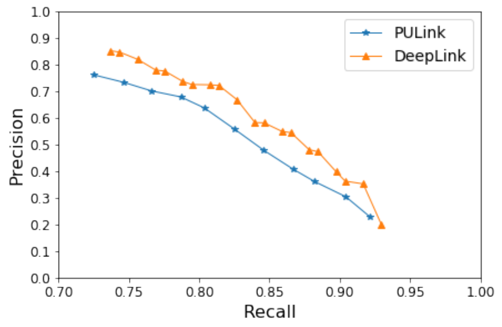


Fig. 7. Performance comparison between DeepLink and PULink in AVRO.

5) RQ_5: How effective is our DeepLink approach compared to the state-of-the-art?

We evaluate and compare our DeepLink approach against the baseline approach PULink [6], which is the state-of-the-art issue-commit link recovery method. TABLE VI reports the performance of PULink across six subject projects, which uses all positive links plus unlabeled links. Compare TABLE VI with TABLE III, and one can see that DeepLink performs better than PULink across all six projects in terms of F1-measure. As for recall and precision, DeepLink has a higher recall in four out of six projects and a higher precision in five out of six projects. Figure 7 shows the PR curves of DeepLink and PULink using the project AVRO as a demonstration, and DeepLink outperforms PULink in terms of AUC. Thus we conclude that DeepLink can outperform PULink, which is the state-of-the-art method.

F. Threats to Validity

There are some potential threats to the validity of our work. Only six projects are used in our experiments and all of the six projects are fairly large and well maintained. And our code knowledge graph currently only supports Java programming language. In the future, we plan to minimize these threats by evaluating DeepLink on more projects of different sizes and written in various programming languages.

V. Related Work

A. Issue–Commit Link Recovery

Links between issue reports and corresponding code commits to fix them can greatly reduce the maintenance costs of a software project and many methods have been proposed to automatic link recovery.

Early works on Issue–Commit link recovery mainly reconstruct the link from the perspective of textual similarity. ReLink [3] analyses how the link between an issue and a code commit is affected by the time interval between the issues completion time and the code commits commit time, the correspondence between who completed the issue and who committed the code, and the textural similarity between the description of the issue and the commits log messages. However, since it is not the main task for the developer to write commit log messages, most of the messages are very short or even empty, making it difficult for ReLink to recover the missing links.

To tackle the problem of poor-quality commit log messages, RCLink [27] uses code summary technique to automatically generate commit log messages from corresponding code changes. The log messages contain a wealth of contextual information, including the committer's intention, code change information and code change impact etc. In total, RCLink extracts 20 features—including 9 text features and 11 metadata features—from issue descriptions and commit log messages. Furthermore, it considers the link recovery problem as a binary classification problem for the first time, and uses a random forest to predict whether there is a link between an issue and a commit.

Besides source code, other non-source files also contain important information. FRLink [5] extracts text information contained in non-source files, and experiment results show that there is more issue-related information in non-source files than in source code files 58% of the time.

Aforementioned studies use positive links to train a model. A positive link is an existing link between an issue and its solution commit. To make better use the large amount of unlabeled data, Yan Sun et. al. pointed out that the link recovery problem should be considered as a model learning problem with both positive links and unlabeled links and proposed PULink [6], which, compared to previous studies, can achieve better results with only 70% positive links.

B. Knowledge Graph Embedding

The concept of knowledge graph was put forward in 2012 by Google [28]. Since then, a lot of studies have been carried out on knowledge graph and it has played a pivotal role in social network analysis, relation extraction, question answering etc. Knowledge graph usually contains structured data as the form of triplets (head entity, relation, tail entity), where the relation models the relationship between the two entities [21]. For example, triplet (RandomUtils, has_method, useTestSeed) means that class RandomUtils has method useTestSeed, where RandomUtils and useTestSeed are entities and has_method is an edge in the knowledge graph.

However, traditional knowledge bases are symbolic and logic, thus numerical machine learning methods can not be leveraged to support the computation over the knowledge bases [22]. To this end, knowledge graph embedding has been proposed to project entities and relations into embeddings. We list some typical studies as follows.

As the first model of translation-based embedding methods, TransE [18] is simple and effective. The basic idea behind TransE is that, relation between two entities corresponds to a translation between the embeddings of entities, i.e. $h + r \approx t$ when (h, r, t) holds. This indicates that t should be the nearest neighbor of $h+r$. The score function of TransE can be written as:

$$f_r(h,t) = ||h + r - t|| \tag{17}$$

TransE applies well to 1-to-1 relations but has issues when applied to N-to-1, 1-to-N and N-to-N relations. For example, for triplets (Matrix, has_property, defaultMatrix) and (Matrix, instance, defaultMatrix), TransE tends to assume that $embedding_{has\_property} = embedding_{instance}$, which does not conform with the fact. To tackle this problem, TransH [19] is proposed to enable an entity to have distinct distributed representations when involved in different relations. The score function of TransH can be written as:

$$f_r(h,t) = ||h_\perp + r - t_\perp|| \tag{18}$$

$$h_\perp = h - w_r^\top h w_r \tag{19}$$

$$t_\perp = t - w_r^\top t w_r \tag{20}$$

Both TransE and TransH assume embeddings of entities and relations being in the same space $\mathbb{R}^k$. However, an entity may have multiple aspects, and various relations focus on different aspects of the entities. Hence, TransR [20] models entities and relations in distinct spaces, i.e., entity space and multiple relation spaces (i.e., relation-specific entity spaces), and performs translation in the corresponding relation space. The score function of TransR can be written as:

$$f_r(h,t) = ||h_r + v_r - t_r|| \tag{21}$$

$$h_r = M_r h \tag{22}$$

$$t_r = M_r t \tag{23}$$

VI. Conclusion

To tackle the problem of Issue–Commit recovery, we propose a novel approach DeepLink based on knowledge graph and deep learning. On top of textual similarities between issue reports and commit logs, DeepLink learns the embeddings of source code files related to a commit via code knowledge graph and thus adds to the model the semantics of code context. Experiments on six real-world projects show that DeepLink has better performance than that of the state-of-the-art method in terms of F1-measure.

REFERENCES

[1] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig, *How do centralized and distributed version control systems impact software changes?*, in ICSE14, pp. 322-333.

[2] A. Bachmann and A. Bernstein, *Software process data quality and characteristics: a historical view on open and closed source projects*, IWPSE/Evol09, pp. 119-128.

[3] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, *Relink: recovering links between bugs and changes*, in FSE11, pp. 15-25.

[4] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, *Multi-layered approach for recovering links between bug reports and fixes*, in FSE12, p. 63.

[5] Y. Sun, Q. Wang, and Y. Yang, *Frlink: Improving the recovery of missing issue–commit links by revisiting file relevance*, in Information and Software Technology, vol. 84, pp. 33-47, 2016.

[6] Y. Sun, C. Chen, Q. Wang, B. W. Boehm, *Improving missing issue–commit link recovery using positive and unlabeled data*, in ASE17, p. 147–152.

[7] J. liwerski, T. Zimmermann, A. Zeller, *When do changes induce fixes?*, in ACM sigsoft software engineering notes (Vol. 30, No. 4, pp. 1-5).

[8] G. Salton, A. Wong, C. Yang, *A Vector Space Model for Automatic Indexing*, in Commun. ACM 18(11): 613-620 (1975).

[9] W. Lan, W. Xu, *Neural Network Models for Paraphrase Identification, Semantic Textual Similarity, Natural Language Inference, and Question Answering*, in COLING 2018: 3890-3902.

[10] H. He, K. Gimpel, J. J. Lin, *Multi-Perspective Sentence Similarity Modeling with Convolutional Neural Networks*, in EMNLP 2015: 1576-1586.

[11] W. Yin, H. Schtze, B. Xiang, B. Zhou, *ABCNN: Attention-Based Convolutional Neural Network for Modeling Sentence Pairs*, in TACL 4: 259-272 (2016).

[12] Q. Chen, X. Zhu, Z. Ling, S. Wei, H. Jiang, D. Inkpen, *Enhanced LSTM for Natural Language Inference*, in ACL (1) 2017: 1657-1668.

[13] R. Ghaeini, S. A. Hasan, V. V. Datla, J. Liu, K. Lee, A. Qadir, Y. Ling, A. Prakash, X. Z. Fern, O. Farri, *DR-BiLSTM: Dependent Reading Bidirectional LSTM for Natural Language Inference*, in NAACL-HLT 2018: 1460-1469.

[14] R. Socher, C. Lin, A. Y. Ng, C. D. Manning, *Parsing Natural Scenes and Natural Language with Recursive Neural Networks*, in ICML 2011: 129-136.

[15] Y. Gong, H. Luo, J. Zhang, *Natural language inference over interaction space*, in arXiv:1709.04348 2017.

[16] Q. Liu, Y. Li, H. Duan, Y. Liu, Z. Qin, *Knowledge graph construction techniques*, in Journal of Computer Research and Development, 2016, 53(3): 582-600. (in Chinese)

[17] Z. Lin, B. Xie, Y. Zou, J. Zhao, X. Li, J. Wei, H. Sun, G. Yin, *Intelligent Development Environment and Software Knowledge Graph*, in J. Comput. Sci. Technol. 32(2): 242-249 (2017).

[18] A. Bordes, N. Usunier, A. Garca-Durn, J. Weston, O. Yakhnenko, *Translating Embeddings for Modeling Multi-relational Data*, In NIPS 2013: 2787-2795.

[19] Z. Wang, J. Zhang, J. Feng, Z. Chen, *Knowledge Graph Embedding by Translating on Hyperplanes*, In AAAI 2014: 1112-1119.

[20] Y. Lin, Z. Liu, M. Sun, Y. Liu, X. Zhu, *Learning Entity and Relation Embeddings for Knowledge Graph Completion*, In AAAI 2015: 2181-2187.

[21] G. Ji, S. He, L. Xu, K. Liu, J. Zhao, *Knowledge Graph Embedding via Dynamic Mapping Matrix*, In ACL 2015: 687-696.

[22] H. Xiao, M. Huang, X. Zhu, *TransG : A Generative Model for Knowledge Graph Embedding*, In ACL 2016.

[23] C. Bird , A. Bachmann , F. Rahman , A. Bernstein, *Linkster: enabling ecient manual inspection and annotation of mined data*, in: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2010, pp. 369370.

[24] K. Cho, B. v. Merrienboer, C. Gulcehre, F. Bougares, H. Schwenk, and Y. Bengio, *Learning phrase representations using RNN encoder-decoder for statistical machine translation*, In Proceedings of the Conference on Empirical Methods in Natural Language Processing, pages 1724-1734, 2014.

[25] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, J. Dean, *Distributed Representations of Words and Phrases and their Compositionality*, in NIPS 2013: 3111-3119.

[26] K. Greff, R. K. Srivastava, J. Koutnk, B. R. Steunebrink, J. Schmidhuber, *LSTM: A Search Space Odyssey*, IEEE Trans. Neural Netw. Learning Syst. 28(10): 2222–2232, 2017.

[27] T. B. Le, M. Vsquez, D. Lo, D. Poshyvanyk, *RCLinker: automated linking of issue reports and commits leveraging rich contextual information*, In ICPC 2015: 36-47.

[28] Singhal, Amit, *Introducing the knowledge graph: things, not strings*, In Official google blog 5 (2012).

[29] X. Huo, M. Li, Z.H. Zhou, *Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code*, In IJCAI 2016: 1606-1612.