A Self-enhanced Automatic Traceability Link Recovery via Structure Knowledge Mining for Small-scale Labeled Data

Lei Chen[13], Dandan Wang*[1], Lin Shi[1], Qing Wang[123]
[1]*Laboratory for Internet Software Technologies*, [2]*State Key Laboratory of Computer Science*, *Institute of Software Chinese Academy of Sciences*, Beijing, China
[3]*University of Chinese Academy of Sciences*, Beijing, China
{chenlei2016, dandan, shilin, wq}@iscas.ac.cn

***Abstract*—Traceability links between requirements and source code are beneficial to the maintenance and evolution activities. Compared with the proposed unsupervised solutions, supervised solutions are more effective to generate trace links automatically and gaining more attention. However, supervised solutions often need to spend a lot of effort on labeling data. To overcome this limitation, we propose a self-enhanced automatic traceability link recovery approach based on structure knowledge mining for small-scale labeled data, named K2Trace, which not only enhances the semantic representations of artifacts by mining context information but also self-enhances the size of training set by exploring transitive relationships. Evaluation results show that K2Trace can outperform the state-of-the-art baseline approach. K2Trace proves the usefulness of mining knowledge from the structure information of software artifacts, as well as provides a new way to substantially reduce the amount of training data needed for training efficient classification models, which may pave the way for generating accurate trace links.**

Keywords—traceability link recovery, context information, context embedding, transitive relationship, inference rule

I. INTRODUCTION

During the processes of software development, writing a piece of code usually contributes to implementing a certain requirement, and generating traceability links between them (abbreviated as R2C links) is meaningful for software maintenance and evolution [1]. For example, the R2C links can help developers better understand the logic and purpose of source code [2], and also help to locate the source code affected by the requirements changes, which may greatly reduce the maintenance costs of a software project [3]. Additionally, the R2C links can be used to automatically select the relevant test case execution, which makes the automated test project more efficient [4]. However, R2C links are often lost, mis-linked, or coarsely-linked in practices. Furthermore, most of the R2C links are many-to-many types. Manually maintaining these links is inefficient and error-prone. Therefore, it is important and meaningful to promote automated methods to recover R2C links.

Various automated approaches have been proposed in previous studies, which can be classified into unsupervised and supervised learning approaches. For unsupervised learning approaches, they mainly choose bag-of-words and probabilistic topic models based IR/ML methods to calculate the textual similarity for recovering R2C links. However, these methods have been inadequate to deal with the term mismatch and partial topic overlap since they do not consider the semantics of software artifacts [5]. For supervised learning approaches, which can learn the underlying correlative information from training data. Current practices have shown that supervised learning approaches [5-10] are more effective than the existing unsupervised learning approaches. However, these approaches have an obstacle that they often require large amounts of existing labeled data for training such as up to 90% of all potential trace links. What's worse, in practice software projects often lack pre-existing trace links [11].

To create accurate classification models that require as little training data as possible, the newest research is proposed by Mills et al. [7] in 2019. They proposed a supervised method ALCATRAL, which adopts active learning to reduce the amounts of training data needed and achieves comparable performance. However, it still exists glaring limitations. It only extracts features from the textual information of artifacts, which has difficulty bridging the logical abstraction gaps that exist between requirements written in natural language and source code written in programming languages. Due to ignoring the implicit knowledge in the structure information of software artifacts, the performance of ALCATRAL is hindered. Furthermore, it is not a fully automated process since it needs human experts to label data, the quality of labeled data is likely to be affected by human error in practice.

Based on the above analysis, we propose a novel supervised learning approach K2Trace (a self-enhanced automatic traceability link recovery approach based on structure knowledge mining for small-scale labeled data), which can create accurate classification models with less training data and without human involvement. On one hand, K2Trace mines the context information to enhance the semantic representations of artifacts. It first learns the context embedding of each artifact from the requirement-code knowledge graph by the complex relation-aware knowledge representation learning, and then combines the context embedding to obtain more comprehensive and precise semantic representations of artifacts. On the other hand, K2Trace explores the transitive relationships of trace links to expand the size of training set without human involvement. It adopts quantitative reasoning to extract inference rules for reducing the amounts of training data needed automatically.

In our study, we evaluate our K2Trace approach on three public datasets to address the following three research questions.

RQ1: How effective is our K2Trace approach compared with the state-of-the-art approach ALCATRAL?

RQ2: How does K2Trace benefit from the context information?

RQ3: Whether the inference rules can generate extra effective training data for improving the classification model?

* Corresponding author

Results show that K2Trace outperforms the state-of-the-art approach ALCATRAL in terms of F1 on average, which provides an improvement of 23.7% on average than ALCATRAL in the smallest training sets and outperforms the best results of ALCATRAL when only using 30% of the data set for training. Both learned context embeddings of artifacts and extracted inference rules contribute to creating accurate classification models when the training data is insufficient. To mine the implicit knowledge, the modeling of context information and the exploring of transitive relationships provide new insights into the supervised generate R2C links in this paper. The improvement in the traceability performance when the training data is insufficient further proves the usefulness of the K2Trace approach.

This paper makes the following contributions:

- We propose the K2Trace approach to create accurate classification models with less training data, which outperforms the state-of-the-art approach.
- We propose a complex relation-aware knowledge representation learning to effectively embed the context information for the more comprehensive and precise understandings of artifacts.
- We extract a set of inference rules by quantitative reasoning to explore the transitive relationships of trace links for self-enhancing the size of training set.

The remainder of the paper is organized as follows. We first summarize related work and explain our motivation in Section II and Section III respectively. The details of our method are described in Section IV. Section V describes our experimental setup, and the results and discussion are shown in Section VI. Finally, in Sections VII to VIII, we discuss threats to validity and conclusions.

II. Related Work

In this section, we focus on prior work related to different aspects of our work.

Early work on R2C link recovery mainly reconstruct the link from the perspective of textual similarity by the IR/ML methods. Antoniol et al. [12] used identifiers and comments of code as queries to recover trace links by VSM and PM. Oliveto et al. [13] studied the effectiveness of several IR-based methods for trace link recovery. Gethers et al. [14] further researched relational topic modeling to complement IR-based methods and combined the similarity measures for improving trace link recovery. However, these researches improve the IR/ML method and their precisions and recalls are often below expectations since these methods only leverage the text information of requirements and source code.

The following work considers source code as an important knowledge source, which contains dependency relationships of code to augment IR/ML-based methods. Mcmillan et al. [15] combined text information and structure information for recovering trace links by constructing the traceability link graph (TLG). Kuang et al. [16] combined IR technology with closeness measure on a method call and data type dependencies in source code to improve IR-based traceability recovery. Panichella et al. [17] utilized the call and inheritance relationships to extend a set of initial trace links by the IR-based method. Although their approach considers dependency relationships of code to recover potential trace links, the performance of these approaches is over-reliance on the results of IR/ML methods. If the initial results of IR/ML methods are not ideal, the effect of the improvement is limited.

To solve the problem of insufficient performance of unsupervised IR/ML methods. Some researchers proposed supervised machine learning approaches. Zhao et al. [8] proposed a domain word embedding-based method, which adopted the learning to rank (LTR) model to recover trace links based on the similarity of the pre-trained domain word embeddings. Wang et al. [9] proposed a hybrid approach, which adopted doc2vec to extract the semantics features, then train the classification model and use logical reasoning to recover trace links. Zhao et al. [10] considered the structure information of software artifacts and used DKRL [18] to learn the contextual representation for recovering the traceability links among artifacts. Mills et al. [6] proposed TRAIL, which leverages query quality and document statistics features in addition to bidirectional IR rankings as feature representations for training predict models to predict trace links. However, these approaches rely on an enormous availability of labels to train effective classification models. This restricts their application in real-world practice since often lack pre-existing trace links in practice projects and collecting labeled data has long been considered time-consuming and costly.

To tackle the limitation of the supervised learning approach that required large amounts of labeled data for creating accurate classification models, ALCATRAL improves upon TRAIL by applying active learning to directly address the prohibitive amounts of training data required to generate those models [7]. Our method is different from ALCATRAL in several ways. First, we consider different knowledge sources such as the structure information of software artifacts except for the traditional text information. We model the context information from the structural relationships of artifacts to obtain more comprehensive and precise semantic representations of artifacts. Second, we explore the transitive relationships of trace links to expand the size of training set automatically, which alleviates the negative influence of human error in practice.

III. Motivation

In this section, we introduce two motivations of our research based on the above analysis of related work.

A. *Mining the context information*

Current practices in R2C link recovery extract text features in terms of various textual mathematic statistics and semantic representation from artifacts to train their models. However, these approaches are limited since some semantic information of software artifacts may be ignored. Furthermore, few of them consider the effect of the context information related to an R2C link recovery, let alone the contextual semantics of each artifact. For example, in Fig. 1, although the existing approaches can recover a trace link between use case “Login” and code class “ServletLogin” based on the text information, another trace link also can be recovered between use case “Login” and code class “ManagerUser” based on the direct code dependency (e.g., call, inheritance). Some indirectly related classes referenced by “ServletLogin” are ignored such as code class “Role”, which will not be directly invoked in “ServletLogin”. However, use case “Login” and code class “Role” contain similar context that role information is required when user login, which provides important clues for R2C link recovery. From this perspective, the context can be a valuable source of information in R2C link recovery.

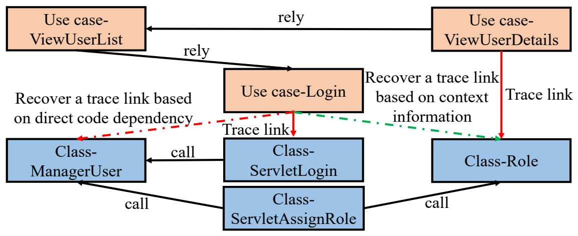

Fig. 1. An example of context information to recover a trace link.

Zhao et al. [10] first mined the context information of software artifacts and their research provided a solid foundation for our work. They modeled the use cases and code elements as a graph through a variety of structure relationships, and then adopted DKRL to encode the context information, contributing to better understand the semantics of artifacts. However, we noted that large amounts of complex relation types are existing in their graph defined. For example, the trace link is a many-to-many relation type. DKRL is insufficient to effectively capture the more distinctive context information of each artifact since DKRL adopts TransE [19] to embed the structure relationships, which can only handle the simple 1-to-1 relation type. The issue leads to learning the context embeddings of lower distinctiveness when handling complex relations. To learn higher distinctive context embeddings of artifacts, we combined relation projection with text representation learning by using TransR [20] instead of TransE.

B. *Explore the transitive relationship*

The transitive relationship is an inherent relationship between two software artifacts (A1, A2) that may influence the existence of a trace link between either A1 and any other artifact or A2 and any other artifact. The use of transitive relationships can simply and directly recover the trace links. The transitive relationships were first explored by Nishikawa et al. [21], which recovered the trace links between two artifacts using the third artifact by textual similarity. Wang et al. [9] directly recovered a trace link between a use case (UC1) and a code class (CC2) based on the pre-existing trace link between UC1 and another code class (CC1), and the extracted explicit code dependency relationships (e.g., implements, inheritance, parameter type, and return type) between CC1 and CC2. These methods utilize the transitive relationships to improve the accuracy of their inferred trace link.

However, the transitive relationships are often implicit among software artifacts [11]. These existing approaches only considered the explicit transitive relationships from the textual similarity metrics or code dependencies. To explore the implicit transitive relationships of trace links, we employ the closeness measure to quantify the functional similarity among code classes [16]. According to our assumption of higher closeness indicates the more likely there is a transitive trace link between two code classes. Therefore, we leverage the closeness measure to explore the transitive relationships for inferring the transitive trace links. For example, Fig. 2 accounts for the transitive relationships of the given trace link among artifacts. Code class "ManagerVotes" has the call relationship with two other classes: "Teaching" and "Votes". The transitive trace links can be generated by judging whether the closeness satisfies the given threshold. Therefore, the transitive trace links can help to generate new trace links, which automatically increases the amount of training data to prevent considerable manual effort.

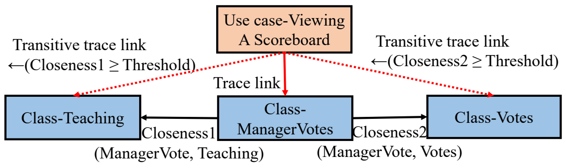

Fig. 2. An example of transitive trace links by closeness measure.

IV. Proposed Approach

In this section, we introduce the K2Trace approach in detail and Fig. 3 presents its overall framework. The detailed processes of K2Trace include eight steps:

1) *Preprocess data:* preprocess the text information and extract the structure information from software artifacts (detail in Section IV. A).

2) *Build requirement-code knowledge graph:* build the requirement-code knowledge graph, which contains the context information of each software artifact based on the preprocessed text information and extracted structure information (detail in Section IV. B).

3) *Learn entity embedding:* design the complex relation-aware knowledge representation learning to capture the context information for obtaining the context embedding of each requirement and code element entity in the requirement-code knowledge graph (detail in Section IV. C).

4) *Build code dependency graph:* calculate the closeness among code classes based on the extracted structure relationships from source code for building a code dependency graph, which contains the closeness measure among code classes (detail in Section IV. D).

5) *Extract inference rule:* extract inference rules based on exploring the transitive relationships of trace links from the code dependency graph by quantitative reasoning (detail in Section IV. E).

6) *Build feature representation:* based on the inference rules extracted in step 5, we generate the new trace links to add into the training set, then based on entities embeddings learned from step 3, we build the feature representation matrix of the training set and the test set respectively (detail in Section IV. F).

7) *Train classification model:* train a classification model for R2C link recovery (detail in Section IV. G).

8) *Recover trace link:* leverage the trained classification model to predict R2C links from unlabeled test data (detail in Section IV. H).

A. *Preprocess Data*

We preprocess the text information and extract the structure information from requirements and source code.

1) *Preprocess text information*

a) Remove character: perform the tokenization operation on all text, remove non-alpha-numeric characters, punctuation, stop words of the sentences in artifacts through a string pattern matching process.

b) Split word: extract meaningful words from identifiers in artifacts. The identifier (e.g., requirement name, class name, method name) is a compound word by concatenating two or more words (or abbreviations) or any other separators such as the underscore character. We employ the upper camel

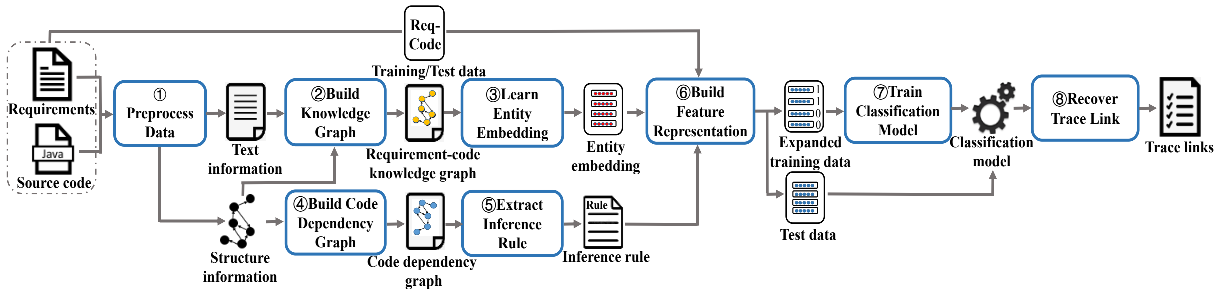

Fig. 3. Overview framework of K2Trace.

case rule to split the compound words.

c) Translation: when requirements and source code contain non-English words, we adopt Google Translate to translate these textual content into English.

2) Extract structure information

We extract the structure information from both the requirements and source code, which include the relationships between code elements (e.g., package, class, method, or field), the relationships between requirements, and the relationships between requirements and code classes. All of these structure information of artifacts are defined in Table I.

a) Relationships between code elements: we adopt the static analysis of the source code by the Java development tool (JDT) to extract an abstract syntax tree (AST), including all the code elements and their relations. Hence, the relationships among code elements can be obtained by parsing the XML format files, which are generated by the JDT.

b) Relationships between requirements: like parsing the relationships between code elements, it is equally important to explore the relationships between requirements. We employ text analysis to parse these relationships through pattern matching. For example, the descriptions of a requirement contain the name of another requirement, namely the requirement relies on another requirement.

c) Relationships between requirements and code classes: there is an R2C link between a requirement and a code class, we extract the relationship from the labeled data.

TABLE I. RELATIONSHIP FROM REQUIREMENTS AND SOURCE CODE

No.	Relationship	Source	Description
1	contain	source code	A class to belong its package
2	extend		Inheritance of a child class to its parent
3	implement		A class to interface it implements
4	hasMethod		A class/interface to its member methods
5	hasField		A class to its fields
6	fieldType		A field to its type
7	parameterType		A method to types of its parameters
8	returnType		A method to types of its return objects
9	methodCall		A method to methods invoked in it
10	throw		A method types of exceptions it throws
11	includes	requirements	A requirement belongs to another requirement
12	extends		A requirement extends another requirement
13	implies		A requirement relies on another requirement
14	traceLink	requirements& source code	A trace link between a requirement and a class

B. Build Knowledge Graph

To mine the context information, we need to build a graph structure, which contains the context information of software artifacts. Therefore, we build the requirement-code knowledge graph based on preprocessed text information and the extracted structure information of artifacts, which is a directed graph $G = (V, E)$. Each node $v_i \in V$ represents an entity, which is a requirement or code element. Each edge $e_j \in E$ is a relation type among these nodes. We employ the 14 relationship type as the relations of entities in Table I.

In each relation above, a relation pair can be represented as one triple (head_entity, relation, tail_entity). Besides, we extract corresponding the preprocessed text information (e.g., the description of requirement, the comment of class and method) from each requirement and code element entity, which can provide the textual description of each corresponding entity. Fig. 4 shows we build the requirement-code knowledge graph and the direction of the arrows is from the node corresponding to the head entity to the node corresponding to the tail entity in the knowledge graph.

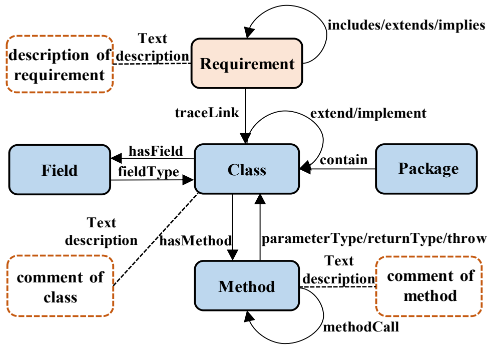

Fig. 4. Shows the requirement-code knowledge graph.

C. Learn Entity Embedding

There is the context information of each artifact in the requirement-code knowledge graph defined. For example, for each requirement or code element, other requirements and code elements with it have various relationships that can be seen as its context. To capture the context information from the requirement-code knowledge graph, we propose a complex relation-aware knowledge representation learning, which combines the complex relation projection with text representation learning. We aim to learn the context embedding for enhancing the semantic representation of each requirement and code element entity. Each entity embedding that we have defined:

$$e = e_c \oplus e_t \tag{1}$$

where e_c is the context embedding learned of each entity, which contains the context information. e_t is the text embedding learned of each entity by the text encoder of our model, which contains the textual semantics. $\oplus$ represents the concatenate operation of vectors for obtaining a more comprehensive and precise embedding of each entity.

1) Complex relation-aware knowledge representation learning

The relation information can be used efficiently, which can be beneficial for knowledge representation learning. Therefore, we adopt TransR to encode the complex structure relationships of the requirement-code knowledge graph into the representation space. The head entity and the tail entity are mapped into a vector space oriented to the relation type by a linear transformation. TransR implements the linear constraints on the embeddings corresponding to the head entity and tail entity in the relation-specific space. The score function of TransR is defined as:

$$E_S(e_i, r_k, e_j) = ||e_i M_r + r_k - e_j M_r||_{L1/L2} \tag{2}$$

where e_i, r_k, and e_j corresponding to the embedding of the head entity, relation, and tail entity respectively. M_r is a matrix projecting entities in the entity space into a relation-specific space. $L1/L2$ is the L1-norm or L2-norm. If a triple exists in the requirement-code knowledge graph, the E_S should be as small as possible, otherwise, E_S should be as large as possible.

Fig. 5 provides a brief explanation of TransR. We consider the code class "Role" and "ManagerUser" are invoked by code class "ServletAssignRole" in Section III. A. In the relation space, TransR binds the embeddings of "Role" and "ManagerUser" to a small neighborhood centered on ServletAssignRoleM$_r$ + r, namely RoleM$_r$ ≈ ManagerUserM$_r$ ≈ ServletAssignRoleM$_r$ + r. In the entity space, the embeddings corresponding to "Role" and "ManagerUser" are constrained around the same relation type. Therefore, if two entities have a more similar neighborhood and relation type are used to constraint them, they are more close to each other.

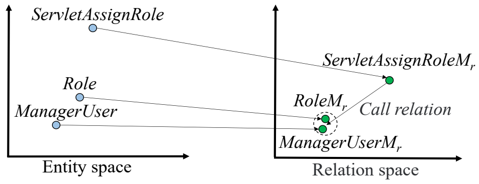

Fig. 5. Shows the brief illustration of TransR

The text information can help to model knowledge graphs such as DKRL [18], which can effectively capture the semantics of entity descriptions. Therefore, we also employ the convolutional neural network (CNN) leveraged in DKRL as the text encoder of our model for a better understanding and promotes the expressive ability of textual semantics by preserving the word order information [5, 22]. We combine the structure representations with the text representations. Finally, we define the energy function as:

$$E(e_i, r_k, e_j) = E_S(e_i, r_k, e_j) + E_D(e_i, r_k, e_j) \tag{3}$$

where E_S shares the same function as (2), to make the two representations of entities compatible, we define $E_D(e_i, r_k, e_j)$:

$$E_D(e_i, r_k, e_j) = E_{DD}(e_i, r_k, e_j) + E_{DS}(e_i, r_k, e_j) + E_{SD}(e_i, r_k, e_j) \tag{4}$$

where $E_{DD}(e_i, r_k, e_j)$ is fully based on the text representation. $E_{DS}(e_i, r_k, e_j)$ and $E_{SD}(e_i, r_k, e_j)$ are compatible energy functions. e_i is the text representation and e_j is the structure representation. The relation representation r_k is shared by all the energy functions, which will contribute to the mutual promotion of the two different representations. Our model will finally project the two vector representations of each entity into the same vector space.

To train our model, we employ stochastic gradient descent (SGD) to minimize the margin-based loss function which is defined as:

$$L = \sum_{(e_i, r_k, e_j)\epsilon T} \sum_{(e_i', r_k', e_j')\epsilon T'} max(0, E(e_i + r_k, e_j) - E(e_i' + r_k', e_j') + \gamma) \tag{5}$$

where T is the correct triples, T' is the corrupted triples and generated from the training triples by randomly replacing an entity or relation. For example, the relation type is traceLink, the requirement entity was replaced by a random other requirement entity, or the code class entity was replaced by a random other code class entity. $\gamma > 0$ is a margin used to separate the correct triples and the corrupted triples.

2) Construct the entity embedding

After model training, we utilize the complex relation-aware knowledge representation learning to generate the context embedding and text embedding of each entity simultaneously. According to (1), we can obtain the final embedding of each requirement and code element entity.

D. Build Code Dependency Graph

To explore the transitive relationships of trace links, we adopt Kuang's core idea based on the closeness measure to build the code dependency graph. Different from their work, which needs executable versions of systems or projects to capture method-level calls and data of code dependencies by dynamic analysis. However, the high required requirement limits the application of their method in practice. Therefore, we consider to eases the difficulties faced by dynamic analysis and employ the result of static analysis to build the code dependency graph, which is defined as $G' = (V', E')$, V' is a set of code classes, and these are two kinds of edges E' in the graph, E'_{DC} represents direct code dependencies and E'_{CD} represents class data type dependencies. The weighted edges are the value of closeness corresponding between code classes, which is in the range [0, 1].

1) Calculating closeness for direct code dependency

For a direct code dependency, if the source class calls the methods of sink class and two code classes share more distinct method calls, they will interact more closely. Besides, the in-degree of the sink class and out-degree of source class also are important factors, the in-degree represents the number of provided method calls from the sink class to the related source classes, and the out-degree represents the number of method calls from the related sink classes to the source class. Smaller in-degree of sink class and smaller out-degree of source class mean that the two code classes also closely interact with each other. The $Closeness_{DC}$ of direct code dependency is calculated as:

$$Closeness_{DC} = \frac{2 \times N}{In\text{-}Degree_{Sink} + Out\text{-}Degree_{Source}} \quad (6)$$

where N represents the number of distinct method calls from a direct code dependency between two code classes. *In-Degree*$_{Sink}$ refers to the number of provided method calls from the sink class to the related source classes. *Out-Degree*$_{Source}$ refers to the number of method calls from the related sink classes to the source class.

2) Calculating closeness for class data type dependency

For an undirect code dependency, the class data type dependency is a weighed schema based on the inverse data type frequency (idtf). The idtf adopts the same concept with inverse document frequency (IDF), which means the higher idtf of a class data type is more possible uniquely shared among code classes. The $Closeness_{CD}$ of class data type dependency can indicate the degree of interaction between the two code classes sharing the data type, which is calculated as:

$$idtf = \log \frac{N}{n_{dt}},\ Closeness_{CD} = \frac{\sum_{x \in \{DT_i \cap DT_j\}} idtf(x)}{\sum_{y \in \{DT_i \cup DT_j\}} idtf(y)} \quad (7)$$

where N refers to the number of all captured class data type dependencies and n_{dt} refers to the occurrence of a given data type in all class data type dependencies. $DT_i \cap DT_j$ is the data types shared between code classes, and $DT_i \cup DT_j$ is all data types between code classes.

Fig. 6 shows a part of the code dependency graph, which contains direct code dependencies (solid lines with arrow) and class data type dependencies (dashed lines).

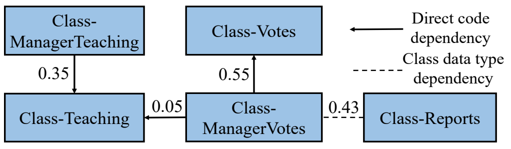

Fig. 6. Shows a part of the code dependency graph.

E. Extract Inference Rule

After the code dependency graph is built, we extract inference rules based on quantitative reasoning to generate the transitive trace links for increasing the amounts of training data by satisfies the given threshold. The training set provides a solid foundation for generating new "oracles". First, we adopt the scale threshold method to define the adaptive thresholds of direct code dependencies and class data type dependencies respectively, which are calculated as follows:

$$Threshold_{DC} = weighted_{DC} \times maxCloseness_{DC} \quad (8)$$

$$Threshold_{CD} = weighted_{CD} \times maxCloseness_{CD} \quad (9)$$

where $weighted_{DC}$ and $weighted_{CD}$ are in the range [0, 1]. $maxCloseness_{DC}$ and $maxCloseness_{CD}$ are the maximum of $Closeness_{DC}$ *and* $Closeness_{CD}$ in all code class pairs respectively. $Threshold_{DC}$ and $Threshold_{CD}$ are also different in different datasets since each dataset has different characteristics.

Then, according to the above thresholds, we extract two inference rules by quantitative reasoning as follow:

Definition 1. Code dependency-based inference rule: Given a trace link between requirement r_i and code class c_j, and a direct code dependency between code class c_j and code class c_k with $Closeness_{DC}$. If $Closeness_{DC} \geqslant Threshold_{DC}$, then there is generated a trace link between requirement r_i and code class c_k.

Rule1: trace link(requirement(r_i), code_class(c_k))

$$\Leftarrow trace\ link(requirement(r_i),\ code\_class(c_j)) \quad (10)$$

$$\wedge\ Threshold_{DC} \leqslant Closeness_{DC}(code\_class_{source}(c_j),\ code\_class_{sink}(c_k)) \vee ((code\_class_{source}(c_k),\ code\_class_{sink}(c_j))$$

Definition 2. Data dependency-based inference rule: Given a trace link between requirement r_i and code class c_j, and a class data type dependency between code class c_j and code class c_k with $Closeness_{CD}$. If $Closeness_{CD} \geqslant Threshold_{CD}$, then there is generated a trace link between requirement r_i and code class c_k.

Rule2: trace link(requirement(r_i), code_class(c_k))

$$\Leftarrow trace\ link(requirement(r_i),\ code\_class(c_j)) \quad (11)$$

$$\wedge\ Threshold_{CD} \leqslant Closeness_{CD}((code\_class(c_j),\ code\_class(c_k)) \vee ((code\_class(c_k),\ code\_class(c_j))$$

F. Build Feature Representation

K2Trace aims to predict the R2C links from the unlabeled test data. First, we can generate the new trace links to add into the training set based on the inference rules automatically, and then build the feature representation matrix based on the entity embeddings of requirements and code classes in the training set and test set respectively.

1) Expand training set

According to the inference rules and the training set, we can generate the new trace links to increase the amounts of training data. For example, in Fig. 7, given a trace link between "Viewing A Scoreboard" and "ManagerVotes" in the training set, and "ManagerVotes" has the calling relationships with two other code classes: "Teaching" and "Votes" in the code dependency graph. The closeness of "ManagerVotes" and "Teaching" equals 0.05 and the closeness of "ManagerVotes" and "Votes" equals 0.55 respectively. If the $Threshold_{DC}$ is 0.5, we can generate a new trace link between "Viewing A Scoreboard" and "Votes" into the training set based on the code dependency-based inference rule. In the same way, the class data dependency-based inference rule is also considered to increase the amounts of training data.

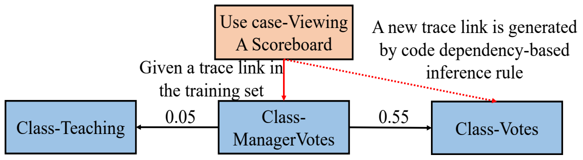
Use case-Viewing A Scoreboard
A new trace link is generated by code dependency-based inference rule
Given a trace link in the training set
Class-Teaching
0.05
Class-ManagerVotes
0.55
Class-Votes

Fig. 7. An example of generating a new trace link by code dependency-based inference rule.

2) Build feature representation with label

The complex relation-aware knowledge representation learning that we have proposed is also a translation-based knowledge representation learning model, which is assumed that the embedding vectors of a valid triple (e_h, r_k, e_t) should satisfy the condition:

$$e_h - e_t \approx r_k \quad (12)$$

where e_h is the embedding of the head entity, e_t is the embedding of the tail entity, *and* r_k is the embedding of the relation.

According to the above assumption, requirement r_i is the head entity and its embedding is e_i, code class c_j is the tail entity and its embedding is e_j. Therefore, we build the feature representation $f(r_i, c_j)$ of the R2C link, which can be calculated:

$$f(r_i, c_j) = e_i - e_j \tag{13}$$

If there is a trace link between a requirement and a code class in the training set and test set, we mark the label of feature representation, which is 1, and 0 otherwise.

G. Train Classification Model

We use the feature representation matrix of R2C links with labels as the training data to train a classification model of support vector machines (SVM). The score function of the classification model can be defined as:

$$g(r_i, c_j) = w\,[f(r_i, c_j)] \tag{14}$$

where $g(r_i, c_j)$ should be 1 if the candidate relation is a trace link, and 0 otherwise. The w is the parameters of the classification model. We minimize the objective function L over the training set to learn the parameters.

$$\mathrm{L} = \|w\|^2 + \sum_{(r_i,\, c_j) \in X} max\left(1 - y_{i,j}\, g\left(r_i, c_j\right)\right) \tag{15}$$

where X represents all potential trace links, $y_{i,j}$ is the ground truth label.

H. Recover Trace Link

By the classification model trained, we can recover the R2C links based on the predicted results. If there are R2C links, the labels are 1 between requirements and code classes, and 0 otherwise.

V. Experiment Setup

In this section, we introduce our experiment setup to evaluate the proposed approach. In Section V. A, we introduce the three evaluated datasets and relevant details. In Section V. B, we select the baseline from existing approaches. In Section V. C, we define metrics for evaluating the performance of the proposed approach. In Section V. D, we introduce experiment settings for the proposed approach.

A. Datasets

Our evaluation is based on three real-world software systems: eAnci [6, 7, 11, 14], eTour [6-10, 14, 22], and SMOS [6-8, 14]. These datasets are public benchmark datasets and can be accessed from the website of CoEST [23]. We choose these datasets because of the availability of requirements specifications and "oracles" at the same trace granularity (e.g., class-level). More significantly, in our study, we focus on Java projects in which source code is written in Java. Table II lists the detailed information of the selected datasets.

TABLE II. Detail of datasets

Item	eAcni	eTour	SMOS
Programming language	Java	Java	Java
KLoC	8	25	15
Number of classes	55	116	100
Number of requirements	41	58	67
Invalid Links	7091	6363	5656
Valid Links	554	365	1044
Number of entity	313	1756	422
Number of relation	14	14	14
Number of word	792	1514	998
Size of Knowledge graph	1794	8303	2742
Direct code dependencies	110	102	233
Class data type dependencies	40	526	12

B. Baselines

Previous studies can be divided into unsupervised and supervised learning approaches. For unsupervised learning approaches, the newest studies are proposed by Chen et al. [22] and the best F1 performance is 37.4% on average which is significantly worse than supervised learning approaches [6, 7, 9, 10]. Therefore, we choose supervised learning approaches as the baseline. Additionally, our study focuses on less labeled data, and ALCARAL [7] is the newest study focusing less data in the traceability link recovery domain currently available and achieves good performance. Therefore, we only choose ALCARAL as the baseline.

C. Evaluation Metrics

To evaluate the performance of K2Trace, precision (P), recall (R), and F-measure (F1) are used to analyze the experiment results, which are the most basic and widely measures in the requirements traceability research field. These metrics are calculated as follows:

$$\text{precision} = \frac{|\text{correct} \cap \text{retrieved}|}{|\text{retrieved}|}\,\% \tag{16}$$

$$\text{recall} = \frac{|\text{correct} \cap \text{retrieved}|}{|\text{correct}|}\,\% \tag{17}$$

$$\text{F1} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{18}$$

where correct represents the set of correct trace links and retrieved is the set of all trace links retrieved by the traceability recovery approaches.

D. Experiment Settings

1) Performance evaluate

To evaluate the performance of K2Trace more accurately, we employ the 10-fold cross-validation when training the classification model for each dataset. Related and unrelated links are divided into 10 equally sized subsets randomly. We begin with a randomly selected initial training set from subsets. For each subsequent experiment, we add a subset (10%), and measure its performance with the other subset as the validation set, until the final experiment where nine subsets (90%) are used as the training set. As with ALCARAL, we perform the same 50 trials of this experiment and present the average results [7].

2) Parameter initialize

K2Trace approach has some parameters needing to be set in advance, the detail of these parameters are as follows:

a) Learn entity embedding: we refer to the optimal parameters of DKRL [18]: learning rate $\lambda = 0.001$, margin $\gamma = 1.0$, window size $k = 2$, dimension of word embedding $n_w = 100$, dimension of feature map $n_f = 100$, number of epochs $n_e = 1000$. Besides, we use the word embeddings trained on Wikipedia by word2vec as inputs for the CNN encoder of complex relation-aware knowledge representation learning.

b) Extract inference rule: we adopt the $weighted_{DC}$ of 0.7 and the $weighted_{CD}$ of 0.9, which corresponding to $Threshold_{DC}$ and $Threshold_{CD}$ respectively [16]. Besides, we define a fixed $Threshold_{idtf} = 1.0$ to filter the class data types by idtf value, since there is difficulty in setting accurate and fixed $Threshold_{idtf}$ by the manual or automatic way [24].

TABLE III. COMPARE THE RESULTS OF ALCATRAL

Dataset	Size of training set																	
	10%		20%		30%		40%		50%		60%		70%		80%		90%	
	KT	ACL	KT	ACL	KT	ACL	KT	ACL	KT	ACL	KT	ACL	KT	ACL	KT	ACL	KT	ACL
eAnci	61.4	52.2	72.5	72.8	**80.2**	78.7	82.6	78.8	83.2	78.8	84.3	78.6	84.2	78.3	84.3	78.2	<u>85.4</u>	78.1
eTour	51.2	42.5	58.8	56.1	**64.4**	58.6	66.4	59.7	67.8	60.1	68.5	60.5	70.4	60.7	72.2	61.1	<u>73.6</u>	60.6
SMOS	64.3	47.6	75.2	62.9	**82.4**	75.3	83.6	79.9	84.4	80.0	85.1	79.4	85.0	79.3	85.3	79.6	<u>86.2</u>	79.7
Avgage	59.0	47.4	68.8	64.0	**75.7**	70.9	77.5	72.8	78.5	73.0	79.3	72.8	79.9	72.8	80.6	73.0	<u>81.7</u>	72.8

The bolded values show the lowest percentage of training data with which K2Trace achieves results that are comparable to the ALCATRAL baseline. The underlined values indicate the best result overall for a dataset.

VI. RESULTS AND DISCUSSION

A. *RQ1: How effective is our K2Trace approach compared with the state-of-the-art approach ALCATRAL?*

For answering this question, we measure the performance of K2Trace using the same evaluation index (F1) with ALCATRAL, since high F1 ensures that our models with both high precision and recall simultaneously, which are pragmatic for automatically performing trace link recovery in practice. In Table III, we can see our K2Trace (abbreviated as KT) achieves better F1 performance on all evaluated datasets than ALCATRAL (abbreviated as ACL). K2Trace outperforms the best result of ALCATRAL overall datasets and outperforms ALCATRAL by 6.7% on average when only using 30% of the data is labeled for training and outperforms ALCATRAL by 12.2% on average when using 90% of the data is labeled for training. Especially, K2Trace provides an improvement of 23.7% on average for the 10% training sets, with a minimum and maximum improvement across systems of 17.6% and 35.1% respectively.

K2Trace captures more semantics by model the context information of software artifacts. ALCATRAL only considers the feature vectors from the various mathematic statistics of text information based on the term co-occurrence. Besides, K2Trace extracts inference rules to explore the transitive relationships of trace links to generate more training data. More importantly, the K2Trace does not require human involvement while ALCATRAL needs to train an initial model for active learning to support with expert annotation.

Summary for RQ1: these results show that K2Trace provides superior F1 performance compared to ALCATRAL particularly for small training sets, which is the most common scenario for software projects in the wild. K2Trace provides additional assistance to create accurate classification models through mining the implicit knowledge from the structure information of software artifacts, which is suitable for the situation of insufficient labeled data and without human involvement.

B. *RQ2: How does K2Trace benefit from the context information?*

For answering this question, we compared K2Trace with two sub-methods by ignoring the part of features. (1) K2Trace-Context (abbreviated as KT-C). It denotes the sub-method only learns context embedding of each artifact, namely, according to (1), $e = e_c$, when $e_t = 0$; (2) K2Trace-Text (abbreviated as KT-T). It denotes the sub-method only learns text embedding of each artifact, namely, according to (1), $e = e_t$, when $e_c = 0$. For each sub-method, it will set the same default parameters.

As shown in Fig. 8, K2Trace as the fusion features model performs significantly better than the sub-methods that features learned separately from context information and text information. It implies that context embedding and text embedding are essential to obtain more comprehensive and precise semantic representations of artifacts than other sub-methods. Besides, these two-part embeddings are both useful in our proposed approach and combine context embeddings and text embeddings can achieve a complementary effect on capturing more precise embeddings to improve R2C link recovery.

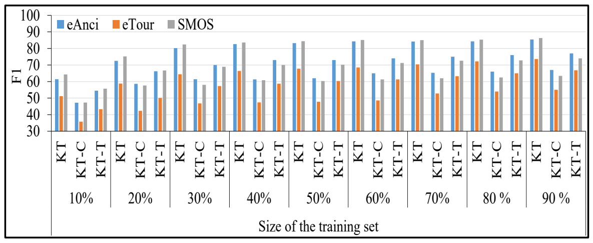

Fig. 8. Performance comparison between K2Trace, K2Trace-Context, and K2Trace-Text.

To further explain the effect of context embedding learned, we regard the example in Section III. A as an example. "Login" and "Role" cannot be linked based on text information or direct code dependency. If we used the context information, we can found that there is a link. For instance, we used 50% of the data for training to obtain context embeddings and searched the top 10 code classes that are nearest to "Login" in the context embedding space shown in Table IV. We can find that "Role" is contained in the top 10 code classes. Additionally, "Login" is actually linked with seven code classes, in the top 10 code classes, we can find 6 code classes (in bold), which nearly 90%. This shows that the context embeddings learned can capture the code context information effectively for R2C link recovery.

TABLE IV. TOP 10 CODE CLASSES NEAREST TO USE CASE "LOGIN"

No.	Code class	Similarity
1	**ServletAlterPersonalDate**	0.277
2	**ManagerUser**	0.243
3	**ServletLogin**	0.217
4	**ServletLogout**	0.189
5	Report	0.182
6	Environment	0.177
7	DBConnection	0.177
8	ServletLoadClassByAccademicYear	0.167
9	**Role**	0.162
10	**UserListItem**	0.143

The bolded code classes show the related code classes with "Login".

Summary for RQ2: K2Trace combined the context embedding and text embedding to obtain more comprehensive and precise semantic representations of artifacts, which are important for F1 performance improvement in the R2C link recovery. Besides, the context embedding compensates the

TABLE V. EFFECT OF INFERENCE RULES ON K2TRACE PERFORMANCE

Dataset	**Size of training set**								
	10%	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**
eAnci	6.5%	1.2%	0.6%	0.4%	0.3%	0.4%	-0.4%	-0.7%	-0.7%
eTour	13.8%	11.4%	9.2%	1.3%	0.2%	-0.6%	-2.3%	-4.6%	-5.4%
SMOS	10.3%	9.6%	1.3%	0.6%	0.3%	0.2%	0.2%	0.1%	0.0%
Average	10.2%	7.4%	3.7%	0.8%	0.3%	0.0%	-0.8%	-1.7%	-2.0%

TABLE VI. RATIO OF CORRECT TRAINING DATA GENERATED BY INFERENCE RULES USING DIFFERENT AMOUNTS OF DATA

Dataset	**Size of training set**								
	10%	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**
eAnci	92.9%	94.7%	95.9%	94.3%	95.7%	95.4%	93.5%	92.6%	91.9%
	13/14	36/38	47/49	66/70	89/93	104/109	115/123	126/136	136/148
eTour	75.8%	70.1%	71.2%	68.8%	66.8%	67.0%	66.7%	66.7%	66.8%
	25/33	61/87	89/125	108/157	125/187	134/200	136/204	138/207	139/208
SMOS	100%	100%	100%	100%	100%	100%	100%	100%	100%
	10/10	26/26	38/38	50/50	66/66	78/78	84/84	96/96	106/106
Average	89.6%	88.2%	89.0%	87.7%	87.5%	87.4%	86.7%	86.4%	86.2%
	48/57	123/151	174/212	222/277	280/346	316/387	335/411	360/439	381/462

lost context information of each artifact by the complex relation-aware knowledge representation learning.

C. RQ3: Whether the inference rules can generate extra effective training data for improving the classification model?

For answering this question, we compared K2Trace with a sub-method, which is named K2Trace-NFR. It trains the classification model without inference rules and will set the same default parameters.

Table V presents the F1 performance improvement on K2Trace by inference rules expanding. It shows that the improvement is obvious when the amounts of data are smaller. For example, for the 10% training set, the inference rules provide an improvement of 10.2% on average. However, we note the F1 performance degradation when 70-90% of the data used for training in eAnci and eTour datasets. For SMOS dataset, the F1 performance of R2C link recovery is not significantly improved, when exceeding 50% of the data used for training.

To further evaluate the effect of generating extra training data, we have analyzed the correct links generated by inference rules when increasing the amounts of training data in Table VI. We can find that the improvement of performance is related to the ratio of the generated correct links. When the ratio of the generated correct links reduces, more noise data are introduced, which will have negative efforts on performance. For instance, the performance fluctuation is relatively large for eTour dataset. For the 10% training set in eTour dataset, the ratio of extra correct links is 75.8%, while for the 90% training set, the ratio is only 66.8%. More noise data are introduced when the training set is increasing, and the improvement of performances drops from 13.8% to -5.4%.

We noted that ALCATRAL has an ideal assumption that developers make no mistakes at providing labels to the training set. However, these perfect results are difficult to achieve in practice. ALCATRAL is similar to the Rocchio method proposed by Hayes et al. [25], which relies on the correctness of each manual decision since mislabeling of links is a possible scenario that can impact the performance of trace link recovery. However, the manual decision was found that about 25% on average of the provided decisions of the developers were incorrect [26]. What's worse, the higher the traceability quality of a system, the worse decisions the human makes [27]. Our inference rules as an automated method achieve a lower error ratio compared with the manual decision, about 12.4% on average.

Summary for RQ3: these results clearly show that the inference rules can effectively improve the F1 performance of K2Trace when the smaller data (10-50%) used for training. In particular, inference rules provide the biggest improvement of 10.2% on average when using the smallest data labeled for training. This is extremely important because we aim to create accurate classification models that require as little training data as possible, which can greatly reduce the time and effort costs manually.

VII. THREATS TO VALIDITY

We have identified the following threats to validity:

1) Internal Validity

The first threat to internal validity concerns the selection of the studied datasets. To ensure the data selected are active and suitable, we choose the evaluated datasets that come from a public repository and have been used in other publications [6-10, 14, 22]. The second threat is on the baselines. We select the state-of-the-art approach as the baseline which is from the latest methods, to reduce this threat.

2) External Validity

The threats concern the possibility of generalizing our results. First, our K2Trace approach currently only supports the Java programming language. Hence, more studies on various programming languages are needed. In the future, we plan to extend K2Trace on more projects of different sizes and written in various programming languages. Second, the size of selected datasets is small. Hence, they might not be appropriate representatives of all the datasets in the requirements traceability domain. More studies on various datasets are needed. Therefore, in the future, we will verify our results with more datasets.

VIII. CONCLUSIONS

In this paper, we proposed the K2Trace approach, a self-enhanced automatic traceability link recovery via structure knowledge mining for small-scale labeled data, which can create accurate classification models with less labeled data. On one hand, we designed the complex relation-aware knowledge representation learning to generate the context embeddings for obtaining the more comprehensive and precise semantic representations of artifacts. On the other hand, we extracted inference rules based on quantitative reasoning to increase the amounts of training data. Our experiment indicated that K2Trace can achieve better F1 performance than ALCATRAL on three real-world projects. Especially, K2Trace can outperform the best result of ALCATRAL overall datasets when only using 30% of the data set for training. Moreover, we evaluated the enhanced strategies of K2Trace, and the results showed that our two enhanced strategies were necessary and effective. K2Trace strives to bridge a logical abstraction gap that exists between requirements written in “high-level” descriptions of natural language and source code written in “low-level” programming languages. It provides a new idea into the R2C link recovery, which reduces the limitation leads due to insufficient labeled data in the supervised solutions.

ACKNOWLEDGMENT

This work is supported by the National Key R&D Program of China No. 2018YFB1403400, National Natural Science Foundation of China No. 62002348, No.62072442, No.61602450, No.61432001, No. 61802374, and Youth Innovation Promotion Association CAS.

REFERENCES

[1] M. Goodrum, J. Cleland-Huang, R. R. Lutz, J. Cheng, and R. A. Metoyer, “What requirements knowledge do developers need to manage change in safety-critical systems?” in Proceedings of the 25th IEEE International Requirements Engineering Conference (RE), 2017, pp. 90-99.

[2] A. De Lucia, R. Oliveto, F. Zurolo, and M. D. Penta, “Improving comprehensibility of source code via traceability information: a controlled experiment,” in Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC), 2006, pp. 317-326.

[3] C. Arora, M. Sabetzadeh, A. Goknil, L. C. Briand, and F. Zimmer, “Change impact analysis for natural language requirements: an NLP approach,” in Proceedings of the 23rd IEEE International Requirements Engineering Conference (RE), 2015, pp. 6-15.

[4] V. Csuvik, A. Kicsi, and L. Vidács, “Source code level word embeddings in aiding semantic test-to-code traceability,” in Proceedings of the 10th International Symposium on Software and Systems Traceability (SST), 2019, pp. 29-36.

[5] J. Guo, J. Cheng, and J. Cleland-Huang, “Semantically enhanced software traceability using deep learning techniques,” in Proceedings of the 39th International Conference on Software Engineering (ICSE), 2017, pp. 3-14.

[6] C. Mills, J. Escobar-Avila, and S. Haiduc, “Automatic traceability maintenance via machine learning classification,” in Proceedings of the 2018 International Conference on Software Maintenance and Evolution (ICSME), 2018, pp. 369-380.

[7] C. Mills, J. Escobar-Avila, A. Bhattacharya, G. Kondyukov, S. Chakraborty, and S. Haiduc, “Tracing with less data: active learning for classification-based traceability link recovery,” in Proceedings of the 2019 International Conference on Software Maintenance and Evolution (ICSME), 2019, pp. 103-113.

[8] T. Zhao, Q. Cao, and Q. Sun, “An improved approach to traceability recovery based on word embeddings,” in Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC), 2017, pp. 81-89.

[9] S. Wang, T. Li, and Z. Yang, “Exploring semantics of software artifacts to Improve requirements traceability recovery: a hybrid approach,” in Proceedings of the 26th Asia-Pacific Software Engineering Conference (APSEC), 2019, pp. 39-46.

[10] G. Zhao, T. Li, and Z. Yang, “An extended knowledge representation learning approach for context-based traceability link recovery,” in Proceedings of the 32nd International Conference on Software Engineering & Knowledge Engineering (SEKE), 2020, pp. 77-82.

[11] K Moran, DN Palacio, C Bernal-Cárdenas, D. McCrystal, “Improving the effectiveness of traceability link recovery using hierarchical bayesian networks,” in Proceedings of the 42nd International Conference on Software Engineering (ICSE), 2020, pp. 873-885.

[12] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, “Information retrieval models for recovering traceability links between code and documentation,” in Proceedings of the 2000 International Conference on Software Maintenance (ICSM), 2000, pp. 40-49.

[13] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, “On the equivalence of information retrieval methods for automated traceability link recovery,” in Proceedings of the 18th International Conference on Program Comprehension (ICPC), 2010, pp. 68-71.

[14] M. Gethers, R. Oliveto, D. Poshyvanyk, and A. De Lucia, “On integrating orthogonal information retrieval methods to improve traceability recovery,” in Proceedings of the 27th International Conference on Software Maintenance (ICSM), 2011, pp. 133-142.

[15] C. McMillan, D. Poshyvanyk, and M. Revelle, “Combining textual and structural analysis of software artifacts for traceability link recovery,” in Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering, 2009, pp. 41-48.

[16] H. Kuang, J. Nie, H. Hu, P. Rempel, J. Lü, A. Egyed, and P. Mäder, “Analyzing closeness of code dependencies for improving IR-based traceability recovery,” in Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017, pp. 68-78.

[17] A. Panichella, C. McMillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshyvanyk, and A. De Lucia, “When and how using structural information to improve IR-based traceability recovery”, in Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR), 2013, pp. 199-208.

[18] R Xie, Z Liu, J Jia, H Luan, and M.Sun, “Representation learning of knowledge graphs with entity descriptions,” in Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI), 2016, pp. 2659-2665.

[19] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko, “Translating embeddings for modeling multi-relational data,” in Proceedings of the 2013 Advances in neural information processing systems (NIPS), 2013, pp. 2787-2795.

[20] Y. Lin, Z. Liu, M. Sun, Y. Liu, and X. Zhu, “Learning entity and relation embeddings for knowledge graph completion,” in Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI), 2015, pp. 2181-2187.

[21] K. Nishikawa, H. Washizaki, Y. Fukazawa, K. Oshima and R. Mibe, “Recovering transitive traceability links among software artifacts,” in Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution (ICSME), 2015, pp. 576-580.

[22] L. Chen, D. Wang, J. Wang, and Q.Wang, “Enhancing unsupervised requirements traceability with sequential semantics,” in Proceedings of the 26th Asia-Pacific Software Engineering Conference (APSEC), 2019, pp. 23-30.

[23] CoEST traceability datasets: http://sarec.nd.edu/coest/datasets.html.

[24] H. Kuang, P. Mäder, H. Hu, A. Ghabi, L. Huang, J. Lü, and A. Egyed, “Can method data dependencies support the assessment of traceability between requirements and source code?” Journal of Software: Evolution and Process (J. Softw. Evol. and Proc.), 2015, 27(11): 838-866.

[25] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, “Advancing candidate link generation for requirements tracing: The study of methods[J],” IEEE Transactions on Software Engineering (TSE), 2006, 32(1): 4-19.

[26] W. -K. Kong, J. H. Hayes, A. Dekhtyar, and O. Dekhtyar, “Process improvement for traceability: a study of human fallibility,” in Proceedings of the 20th IEEE International Requirements Engineering Conference (RE), 2012, pp. 31-40.

[27] D. Cuddeback, A. Dekhtyar, J. Huffman Hayes, J. Holden, and W.-K. Kong, “Towards overcoming human analyst fallibility in the requirements tracing process: Nier track,” in Proceedings of the 33rd International Conference on Software Engineering (ICSE), 2011, pp. 860-863.