



Article

Combining Machine Learning and Logical Reasoning to Improve Requirements Traceability Recovery

Tong Li ^{1,*} , Shiheng Wang ¹, David Lillis ²  and Zhen Yang ¹¹ Faculty of Information Technology, Beijing University of Technology, Beijing 100124, China; yeweimian21@163.com (S.W.); yangzhen@bjut.edu.cn (Z.Y.)² School of Computer Science, University College Dublin, Dublin 4, Ireland; david.lillis@ucd.ie

* Correspondence: litong@bjut.edu.cn

Received: 12 September 2020; Accepted: 5 October 2020; Published: 16 October 2020



Abstract: Maintaining traceability links of software systems is a crucial task for software management and development. Unfortunately, dealing with traceability links are typically taken as afterthought due to time pressure. Some studies attempt to use information retrieval-based methods to automate this task, but they only concentrate on calculating the textual similarity between various software artifacts and do not take into account the properties of such artifacts. In this paper, we propose a novel traceability link recovery approach, which comprehensively measures the similarity between use cases and source code by exploring their particular properties. To this end, we leverage and combine machine learning and logical reasoning techniques. On the one hand, our method extracts features by considering the semantics of the use cases and source code, and uses a classification algorithm to train the classifier. On the other hand, we utilize the relationships between artifacts and define a series of rules to recover traceability links. In particular, we not only leverage source code's structural information, but also take into account the interrelationships between use cases. We have conducted a series of experiments on multiple datasets to evaluate our approach against existing approaches, the results of which show that our approach is substantially better than other methods.

Keywords: requirements traceability recovery; artificial intelligence; machine learning; rule-based reasoning; feature engineering

1. Introduction

Software systems generate a large number of software artifacts during iterative development, such as requirements documents, source code, test cases, and bug reports [1]. Software traceability is the ability to trace software artifacts that are produced in different phases of software development, which is an important quality of software systems [2]. Maintaining requirements traceability links between requirements artifacts and other software artifacts (e.g., source code) is essential for program understanding, impact analysis, and software development management. In particular, the satisfaction of requirements can be effectively evaluated based on such requirements traceability links [1]. Therefore, this topic has attracted more and more attention in the field of software engineering.

With the continuous iterative development of software systems, software artifacts are constantly modified and the corresponding traceability links have to be continuously updated and maintained, which is an expensive and non-trivial task [3,4]. In the development process of real software projects, due to the time pressure, developers typically do not deal with the traceability links during their development as it does not bring short-term benefits. Therefore, traceability links are often not updated and maintained in time and thus typically outdated or even missing [2,5]. It is essential to recover the traceability links between software artifacts, contributing to the successful software maintenance and evolution [4].

Creating and maintaining traceability links manually is simple and straightforward, but time-consuming, and so it is only suitable for small projects [1]. Several studies in the field of software engineering are dedicated to solving this problem through automation [4,6]. The IR (Information Retrieval)-based methods are a classic method for traceability recovery [3]. Such methods focus on calculating text similarity between software artifacts via typical IR methods, e.g., Vector Space Model (VSM) [7] and Latent Semantic Indexing (LSI) [8], creating traceability links between artifacts that have high similarity. However, the IR-based approaches typically only consider the text of software artifacts, which include a large proportion irrelevant and redundant information, resulting in lower accuracy [3,5]. Some studies attempt to use structural information (relationships between source code, such as class inheritance) to improve IR-based methods [3,4,9]. However, the existing approaches do not systematically and comprehensively explore the structural information of related artifacts [3].

In this paper, we propose a hybrid framework to systematically and effectively recover traceability links between requirements use cases and source code classes, which is shown in Figure 1. Specifically, we propose to concentrate on only essential information for traceability link recovery based on the corresponding domain knowledge. All such essential information is used to engineer features for training traceability link classifiers, i.e., phase 1 in Figure 1. In the second phase, we investigate and explore the structural information between use cases and source code classes, respectively, and define corresponding inference rules in order to discover more traceability links. This paper is an extended version of our previous work [10]. In particular, we have significantly extended and improved our previous work in the following aspects.

- We further investigate the semantics of code constructs and improve our framework by proposing an additional collection of meaningful features.
- We conduct in-depth investigation on the relationships between use cases, based on which we have defined five additional logical reasoning rules to enhance the effectiveness of the second analysis phase.
- We have extensively enriched the evaluation by considering three additional data sets (previously only one data set was used). Correspondingly, we have designed and conducted many more experiments to comprehensively evaluate our proposal.

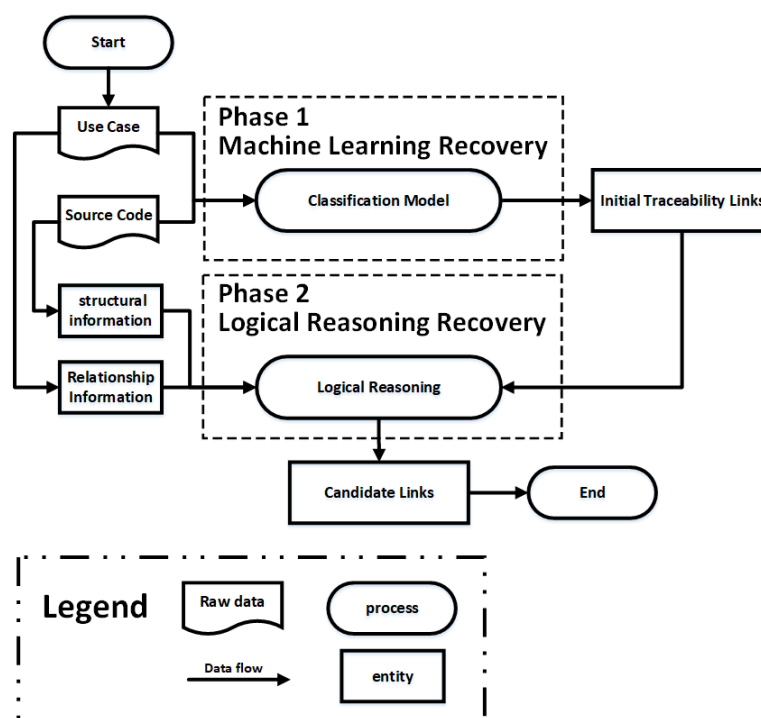


Figure 1. The overall process of our study.

The reminder of the paper is structured as follows. In Section 2, we survey existing related works and discuss their advantages and disadvantages. We then present our proposal in a step-by-step fashion in Section 3. Section 4 presents the details of our evaluation, in which we have systematically designed and conducted a series of experiments. After that we review the threats to validity of our experiments in Section 5. Eventually, we conclude the paper and discuss future work in Section 6.

The remaining part of this paper is organized as follows. We first review and discuss related work in Section 2. We detail our main proposal in Section 3, while reporting its evaluation in terms of experiments in Section 4. Discussions regarding threats to the validity of our proposal are described in Section 5. Finally, Section 6 concludes the paper and points out future work.

2. Related Work

2.1. IR-Based Methods

Software artifacts contain a lot of textual information. The classic method is using IR techniques for traceability link recovery. This kind of method considers that software artifacts with high text similarity probably have traceability links. Hayes et al. propose using VSM, LSI and other IR methods to calculate the textual similarity between artifacts directly, obtaining candidate links by setting a threshold or selecting the first k links [11–14]. De Lucia et al. improve traceability recovery by using a smoothing filter [13].

Oliveto et al. apply a variety of different IR methods, namely JS (Jenson-Shannon) [15], VSM [7], LSI [8] and LDA (Latent Dirichlet Allocation) [16] for traceability recovery to improve the accuracy of independent methods [17]. The results show that VSM, JS and LSI are equivalent, while the precision of LDA is lower than other IR methods. However, LDA can capture unique dimensions and provide orthogonal similarity measures. The combination of LDA and other IR technologies can improve the precision of traceability recovery.

Gethers et al. propose a relational topic model (RTM) [18] based on LDA, which can extract and analyze topics and the relationships between them from software artifacts [2]. Case studies show that when RTM is combined with VSM and JS, the results of traceability recovery are improved. They proposed the RTM is orthogonal to VSM and JS [2].

Capobianco et al. propose a simple method based on IR which only considers nouns [5,19]. The proposed method extracts the nouns from software artifacts and indexes them to define the semantics of the software artifacts. They propose that the basic principle of the method is that the language used in the software documents can be classified as sectoral language, in which nouns provide more indication of the semantics of the document [5]. The proposed method is applied to JS and LSI models, improving the precision.

Asuncion et al. propose an automatic technology that combines traceability recovery with LDA topic modeling [1]. It automatically records traceability links during software development and learns probabilistic topic models on artifacts. The learned model allows semantic classification of artifacts and topic visualization of software systems. At the same time, they propose a prospective method which creates links between artifacts based on the operation of developers. They implement several tools to validate their method.

This type of method can create and maintain the traceability links automatically and reduce the cost of traceability recovery. However, it treats the contents of software artifacts as simple text, which will be interfered with by a lot of irrelevant redundant information, and returns a large number of false-positive links, resulting in low precision.

2.2. Hybrid Methods

Some studies propose combining textual information and structural information of software artifacts for traceability recovery. The combination method obtains a set of candidate links based on IR method initially, and then updates the similarity of the traceability links based on the structural information, thereby extending or filtering the traceability links.

McMillan et al. propose a technique to recover traceability links by combining textual analysis and structural analysis [9]. They speculate that related requirements share related source code [9]. In textual analysis, they get the textual similarity between software artifacts using LSI. In terms of structural analysis, they use the JRipples [20] structural analysis tools to recover traceability links between source code artifacts and build an Evolving Interoperation Graph (EIG) [21]. They propose the Traceability Link Graph (TLG), which encodes all recovered traceability links as edges between nodes. Taking the textual similarity matrix calculated by LSI, the EIG constructed by JRipples and the threshold of similarity as input, a TLG is created and traceability links are generated according to the relationship between the nodes. They name this method as Indirect Traceability Link Recovery Engine (ILRE) [9]. A preliminary case study shows that the combined approach improves precision and recall compared to independent methods based on textual similarity.

Panichella et al. believe that the combination method is susceptible to the IR method [3]. If the candidate links obtained by IR are correct, then the use of structural information can help to find more correct links. Otherwise, using structural information is not helpful and may even lead to more unrelated links. Therefore, they propose that software engineers should verify the traceability links recovered by the IR method before extending traceability links with structural information [3]. The structural information should only be used in the situation where the traceability links recovered by the IR method are identified as correct links. They propose a method named User-Driven Combination of Structural and Textual Information (UD-CSTI) [3]. Experiments show that the UD-CSTI outperforms pure IR-based methods and methods that combine textual information with structural information simply.

Zhang et al. implement a tool named R2C which utilizes synonyms, verb-object phrases, and structural information to recover traceability links between requirements and source code [4]. R2C uses WordNet [22] to locate synonyms for a given term and defines a method for calculating the similarity between two terms. R2C uses Stanford Parser [23] to analyze the syntactic structure of the sentence and extract verb-object phrases. Because requirements and code comments are different from programming languages, they use different methods to extract verb-object phrases from them. They build a code graph according to the method call and inherit the relationship of the code, increase the similarity of requirements, and code according to the code graph with an adaptive reward. An empirical study shows that verb-object phrases are the most effective feature to recover traceability links. R2C can achieve better results when compared with IR-based techniques using a single feature.

Kuang et al. suggest using closeness analysis for call dependencies to improve existing methods [24–27]. Specifically, they quantify the closeness of each call dependency between the two methods, refining the quality of keywords extracted from source code changes, thereby improving the accuracy of outdated requirements identification. The proposed method is named SPRUCE (SuPporting Requirements Update by ClosEness analysis). Although these combination methods have improved the performance of the traceability recovery to some extent, they are very dependent on the results of the IR method.

2.3. Other Studies

There are several other approaches in the literature to recovering traceability links. Ali et al. inspired by the web trust model, propose a method called Trustrace to improve the precision and recall of traceability recovery [28]. Trustrace can get a set of traceability links and re-evaluate rankings. They also propose a method named Histrace to identify traceability links between requirements and source code by establishing VSM in CVS/SVN changelogs. Diaz et al. propose a method called

TYRION (Traceability link Recovery using Information retrieval and code Ownership) [29]. It uses code ownership information to capture the relationship between source code artifacts to improve traceability recovery between documents and source code. Specifically, it extracts the author of each source code component and determines the context of each author. It then computes the similarity between the document and the context of the author.

Ali et al. speculate that understanding how developers verify traceability links can help improve IR-based technology [30,31]. Firstly, they use an eye-tracking system to capture the eye movements of developers while verifying traceability links. They analyze the obtained data to identify and rank types of source code entities according to developers' preferences. They propose a term weighting scheme SE/IDF (source code entity/inverse document frequency) and DOI/IDF (domain or implementation/inverse document frequency). They integrate the weighting scheme with the LSI technique and apply it to traceability recovery. Lucassen et al. propose Behavior Driven Traceability (BDT) to establish traceability by using automatic acceptance testing [32]. These tests detail user story requirements into steps that reflect user interaction with the system. Unlike unit tests, these steps do not directly execute the source code itself. Instead, it initiates a full instantiation of the software system and then simulates how the user interacts with the interface. It uses a runtime tracker to identify all the source code without the need for IR techniques.

Guo et al. use deep learning to incorporate the semantics and domain knowledge of requirements artifacts into a traceability solution [33]. They use word embedding and recurrent neural network (RNN) models to generate traceability links. Word embedding learns word vectors representing domain corpus knowledge, and RNN uses these word vectors to learn sentence semantics of artifacts. They identify the Bidirectional Gated Recurrent Unit (BI-GRU) as the best model for traceability. The experiment result shows that BI-GRU significantly surpasses VSM and LSI.

Sultanov et al. make use of reinforcement learning (RL) for traceability [34]. The RL method demonstrates targeted candidate link generation between textual requirement artifacts. It does so by identifying common text segments between documents and suggesting links between these documents. The RL method surpasses TF-IDF in terms of recall. Ana et al. propose a method called Evolutionary Learning to Rank for Traceability Link Recovery (TLR-ELtoR) [35]. They recover the traceability links between requirements and models by combining evolutionary computing and machine learning to generate rankings of model fragments that can fulfill requirements. Although these methods can improve traceability recovery to a certain extent, they usually require preconditions, are often accompanied by high computing complexity, and the improvements are limited.

3. A Hybrid Traceability Recovery Approach

Our entire approach is systematically designed with three phases (including preprocessing), which is presented in Figure 2. Specifically, the traceability links we investigate in this paper are the ones that connecting use cases and source code classes. As shown in the figure, our approach starts with a preprocessing phase, which is designed to extract data that is relevant and meaningful to our approach. In phase 1, (i.e., machine learning recovery), according to the domain knowledge, the information that is critical to the traceability links recovery of software artifacts is extracted to engineer features. Following this, the traceability links between software artifacts are represented as vectors based on these features, which are then used to train traceability link classifiers. In phase 2 (i.e., logical reasoning recovery), the structural information between the source code classes and the interrelationships between the use cases are explored in depth to complement the trained classifiers obtained in the first phase. The details of our proposal are presented in the remainder of this section.

3.1. Preprocessing

The software artifacts that are derived from real projects typically contain noise and cannot be used directly by our proposed workflow. Thus, as the first step of our approach, we follow a classic process to pre-process the contents of the software artifacts, as follows:

- Remove punctuation, numbers, and special characters in software artifacts.
- Extract words in software artifacts by word segmentation. Not that use cases and source code comments are typically written in natural language, while and source code itself is developed with specific programming languages. Since natural language and programming languages are different, we need to deal with them separately. For natural language texts that appear in the content of use cases and comments in source code files, all words are separated by spaces. For source code identifiers, such as class names and method names, we perform word segmentation according to the naming convention adopted in the project, such as the camel-case style and underscore style. It is worth noting that we here assume that the source code should comply with certain naming conventions, otherwise, it cannot be appropriately processed by our segmentation algorithm.
- Since many project repositories are developed and documented in natural languages other than English, it is necessary to translate the content of the software artifacts. To this end, we leverage the advanced translation engine to translate all the documents into English, if they were originally not specified in English. In particular, we used Google Translate APIs in this paper, which would be evolved according to the recent advances in language translation.
- We remove the stop words from the software artifacts (We refer to the stop word list at <https://gist.github.com/sebleier/554280>). In addition, we also remove Java keywords in source code comments (e.g., static and private).
- We perform lemmatization for all words in order to facilitate natural language analysis, i.e., transforming the verb, noun, adjectives and adverbs into their bases.

3.2. Phase 1 Machine Learning Recovery

In the first analysis phase, we propose a machine learning-based method to recover a preliminary set of traceability links. Specifically, we select the essential features of software artifacts and vectorize the traceability links based on such features. The detailed analysis process (as shown in Figure 2) is presented and explained below.

3.2.1. Feature Extraction

As use cases and source code are written in natural languages and programming languages, respectively, they involve different types of information that are not all useful for identifying traceability links. For example, the specific programming statements typically do not correspond to requirements, while class names, method names, and method comments are very likely to reflect the functional requirements. Similarly, use cases are documented with various fields, only some of which are meaningful for traceability link analysis. In this paper, we exclusively investigate the following features which are meaningful for recovering traceability links (Table 1).

Table 1. The vital features of a traceability link vector in Phase 1.

Artifact	Feature	Description
Use Case	Title	The title of use case
	Description	The description of use case
Source Code	Class Name	The class name
	Class Comment	The comment associated with a class
	Method Name	The method name in a class
	Method Comment	The comment associated with a method
	Class Attribute	The class attribute (Type, name)
	Method Parameter	The method parameter (type, name, Javadoc)
	Method Return	The method return (type, name, Javadoc)

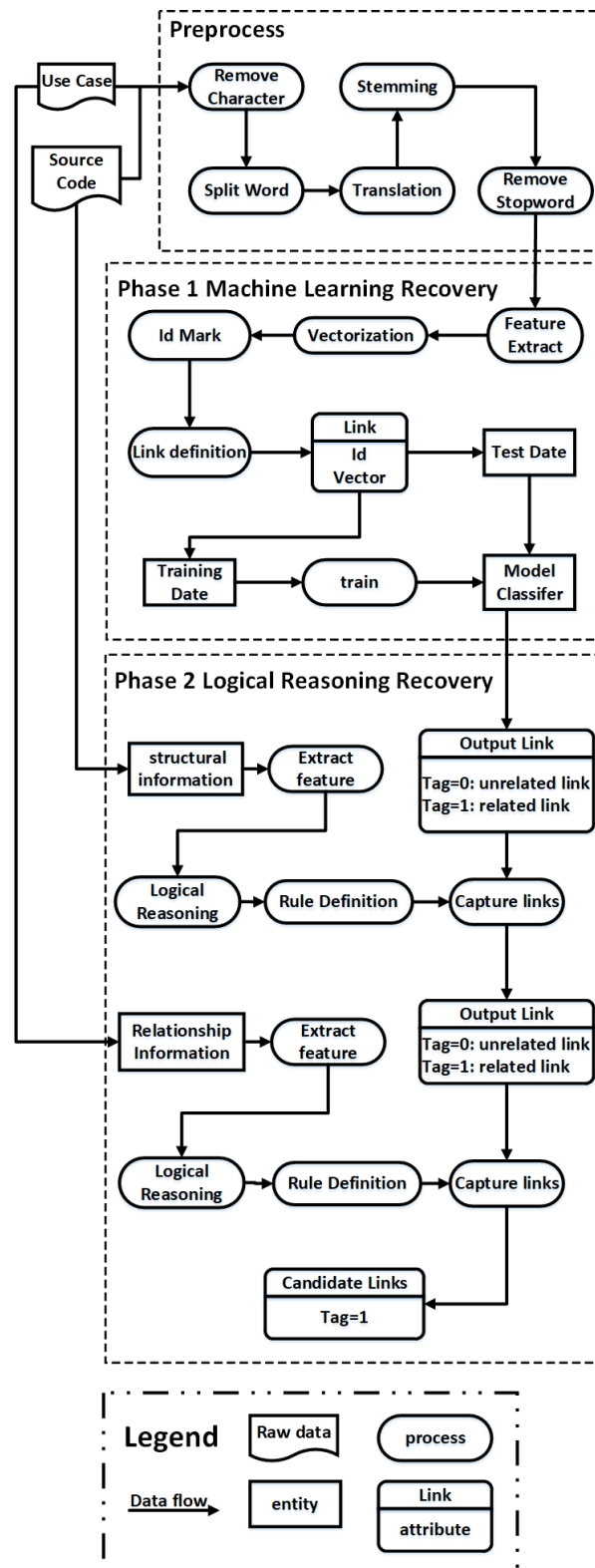


Figure 2. The analysis process of our proposed hybrid approach for recovering traceability links.

Building on our previous research [10], we further explore the class attributes, method parameters, and method return information of the source code as the features in the traceability link vectors. Also, we find that the type, name and Javadoc of the class attributes, method parameters and method return are the most influential information for identifying traceability links. Here, we illustrate the importance of such information in detail.

Type. Figure 3 shows a code segment of parameter type information as an example. In the parameter list (*IdentityCard c*) of the method *registerIdentityCard*, the parameter name *c* is abbreviated and contains no information, but the parameter type *IdentityCard* is valuable information. This suggests that the type information of the parameter should be considered to be relevant and useful.

```
public class DbIdentityCard {
    public boolean registerIdentityCard (IdentityCard c) {
        ...
    }
}
```

Figure 3. An example of parameter type information of a code feature.

Name. An example of parameter name information of a code feature is shown in Figure 4. In the parameter list (*int citizenId, String newEmail*) of the method *modifyCitizenEmail*, the parameter types *int* and *String* are common Java keywords and contain no useful information, whereas the parameter names *citizenId*, *newEmail* are valuable information. Thus the name information of the parameter should also be considered.

```
public class DbCitizen {
    public boolean modifyCitizenEmail(int citizenId,
        String newEmail){
    }
}
```

Figure 4. An example of parameter name information of a code feature.

Javadoc. An example of Javadoc information of a code feature is shown in Figure 5. Javadoc is a standardized and widely accepted way to document Java programs, from which we can extract classes and methods from the program source code. For example:

@param The description of a method parameter

@return The description of the method return value

```
/**
 * @param The Id of the searched classroom
 */
public class ManagerClassroom {
    public Classroom getClassroomById(int pId) {
        ...
    }
}
```

Figure 5. An example of parameter Javadoc information of a code feature.

Therefore, Javadoc comments are also useful information. The parameter type and parameter name of the parameter list (*int pId*) do not contain much information, but the *@param* annotation contains a lot of valuable information. Therefore the Javadoc comments for the parameters also include important information that is considered in our approach.

Specifically, for the method parameter and method return features, we focus on their type, name and Javadoc information. Since the Class Attribute does not have corresponding Javadoc comments, its type and name information are considered. In the experimental evaluation, we will evaluate the experimental results of adding these three features.

3.2.2. Vectorization

In order to embed the semantics of all the essential features we have mentioned previously, we propose a vectorization approach to comprehensively characterize the interrelationships between use cases and source code classes. Vectors are computed based on the features of use cases and source

code. Specifically, only features identified in Table 1 are considered. If one feature involves multiple contents, for example, a class has multiple method names, we will merge these into a single feature. The vector of the link is defined as follows:

$$\begin{aligned} vector &= (d_1, d_2, \dots, d_{14}) \\ d_i &= \text{sim}(\text{feature}_{\text{usecase}}, \text{feature}_{\text{sourcecode}}) \\ i &= 1, \dots, 14 \end{aligned} \quad (1)$$

As is shown in Figure 6, we compute the similarity between each use case feature and each source code feature, so we can obtain a vector with 14 dimensions. Each dimension of the vector represents a particular aspect of similarity between features. Therefore, we convert the corresponding contents of each feature into a sentence vector by using the Doc2vec algorithm [33]. Then, we compute the cosine similarity between two vectors, the results of which are used to characterize the particular dimension of the vector. Eventually, we can obtain the traceability link vectors, comprehensively representing the relationship between the use cases and source code.

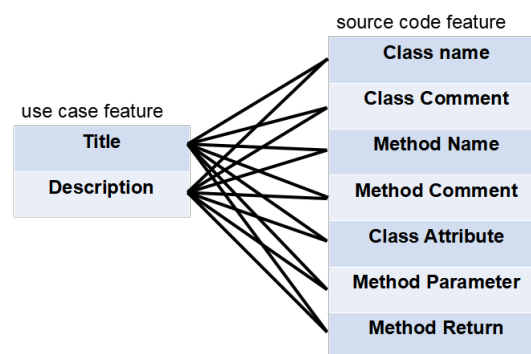


Figure 6. Computation of the similarity between each feature.

To formally and clearly represent the links between use cases and source code classes, we propose formal definitions, which are shown in Table 2. Specifically, we enumerate all possible links between each use case and each source code class. The *id* indicates the source artifact and target artifact of the link. The *vector* presents the vectorized link interrelationships that were derived previously. The *tag* is a boolean variable, which indicates whether the link is a valid traceability link or not.

Table 2. The definition of a link between a use case and a source code component.

Attribute Value		Description
<i>id</i>	(s_i, c_j)	s_i : Use Case, c_j : Source Code
<i>vector</i>	(d_1, d_2, \dots, d_n)	The vector that represents link relationship
<i>tag</i>	0/1	0: unrelated link; 1: related link

3.2.3. Machine Learning Classification

In this step, we input all the tagged training data into specific classification models in order to obtain the corresponding classifiers. Specifically, we have combined multiple classification algorithms in order to yield an inclusive and comprehensive classifier, including decision tree, k-nearest neighbors (KNN), random forest, and gradient boosted decision trees (GBDT). All links that are classified by any of these classifiers as true will be included in the set of candidate links. The design rationale of this is to emphasize the recall of the final recovery model.

3.3. Phase 2 Logical Reasoning Recovery

There are a large number of relationships in software artifacts, which are of great significance for traceability recovery. Therefore, we make use of such relationships between software artifacts to improve traceability recovery. Specifically, we have identified a set of important relationships among source code classes and use cases based on related domain knowledge, respectively. According to such relationships, we have defined corresponding inference rules to effectively identify traceability links.

3.3.1. Structural information between Source Code Components

There many types structural information among source code components such as *implementation*, *inheritance*, and *method call*. The structural information reflects the closeness between source code components and is of great value for traceability link recovery. Based on our previous research [10], we have added the use of *class attribute* and *method call* relationships for traceability recovery. Such features are selected based on both our domain knowledge and empirical studies. Specifically, the *implements* relationship between a class and an interface indicate the class has similar functions with the interface, and they should be analyzed together. *inherits* relations represent the interrelationships between a class and its super-class, which are also closely associated. Moreover, if methods defined in one class are used by another class, it is also an important indicator of associated classes.

Based on previous research [10], we found that *class attribute* and *method call* relationships between source code components also indicate potential traceability links. The *attributes* of a class describe the properties of the class. If a class contains another class as its attribute, it usually means that there is an inclusion relationship between the two classes. According to empirical research, if a use case has a traceability link with a class A, then the use case usually also has a traceability link with other classes that have *attribute* relationships with the class A. A method of a class often calls methods of another class. If there is a *method call* relationship between two classes, it often means that there is a close relationship between them. According to empirical research, if there is a traceability link between a use case and class A, there is usually a traceability link between the use case and other classes that call methods of class A.

Therefore, we propose to focus on the features of the source code structural information that are shown in Table 3. In previous research, we found that the method *parameter* structural information is not helpful for traceability recovery, and even negatively impacts the results sometimes. Therefore, in this study, Phase 2 did not use method *parameter* structural information for traceability recovery.

Table 3. The structural information features of source code in Phase 2.

Artifact	Feature	Description
Source Code	Class Implementation	The class implementation relationship
	Class Inheritance	The class inheritance relationship
	Method Return	The method return relationship
	Class Attribute	The class attribute relationship
	Method Call	The method call relationship

According to the domain knowledge and the structural information features between source code components, we have proposed the following inference rules in order to discover more traceability links. Specifically, we traverse all the traceability links output from phase 1, and identify new links as long as they match our defined inference rules.

$$\begin{aligned}
 & \text{Rule 1 : } \text{traceability_link}(\text{use_case}(s_i), \text{source_code}(c_k)) \\
 & \leftarrow \text{traceability_link}(\text{use_case}(s_i), \text{source_code}(c_j)) \\
 & \wedge \text{implement}(\text{source_code}(c_k), \text{source_code}(c_j))
 \end{aligned} \tag{2}$$

$$\begin{aligned}
 & \text{Rule 2 : } \text{traceability_link}(\text{use_case}(s_i), \text{source_code}(c_k)) \\
 & \leftarrow \text{traceability_link}(\text{use_case}(s_i), \text{source_code}(c_j)) \\
 & \wedge \text{inherit}(\text{source_code}(c_k), \text{source_code}(c_j))
 \end{aligned} \tag{3}$$

$$\begin{aligned}
 & \text{Rule 3 : } \text{traceability_link}(\text{use_case}(s_i), \text{source_code}(c_k)) \\
 & \leftarrow \text{traceability_link}(\text{use_case}(s_i), \text{source_code}(c_j)) \\
 & \wedge \text{return}(\text{source_code}(c_k), \text{source_code}(c_j))
 \end{aligned} \tag{4}$$

$$\begin{aligned}
 & \text{Rule 4 : } \text{traceability_link}(\text{use_case}(s_i), \text{source_code}(c_k)) \\
 & \leftarrow \text{traceability_link}(\text{use_case}(s_i), \text{source_code}(c_j)) \\
 & \wedge \text{attribute}(\text{source_code}(c_k), \text{source_code}(c_j))
 \end{aligned} \tag{5}$$

$$\begin{aligned}
 & \text{Rule 5 : } \text{traceability_link}(\text{use_case}(s_i), \text{source_code}(c_k)) \\
 & \leftarrow \text{traceability_link}(\text{use_case}(s_i), \text{source_code}(c_j)) \\
 & \wedge \text{method_call}(\text{source_code}(c_k), \text{source_code}(c_j))
 \end{aligned} \tag{6}$$

Rule 1 means if there is a traceability link between use case s_i and source code c_j , and source code c_k implements source code c_j , then there is a traceability link between use case s_i and source code c_k .

Rule 2 means if there is a traceability link between use case s_i and source code c_j , and source code c_k inherits source code c_j , then there is a traceability link between use case s_i and source code c_k .

Rule 3 means if there is a traceability link between use case s_i and source code c_j , and the method in source code c_j use source code c_k as its return type, then there is a traceability link between use case s_i and source code c_k .

Rule 4 means if there is a traceability link between use case s_i and source code c_j , and source code c_k contain source code c_j as its attribute, then there is a traceability link between use case s_i and source code c_k .

Rule 5 means if there is a traceability link between use case s_i and source code c_j , and the method in source code c_k call the method in source code c_j , then there is a traceability link between use case s_i and source code c_k .

3.3.2. Interrelationships between Use Cases

In addition to exploring the interrelationships between source code classes, we argue it is equally important to investigate the interrelationships between use cases. A use case describes the system response to participant requests in certain conditions. The interrelationships between use cases contain *include*, *extend*, and *generalize*. All of these are explored in our approach. We here first briefly introduce the three types of relationships, and then describe how they are used in our approach.

Include: The include relationship means that a use case can include behaviors that other use cases have and contain them as a part of its own behavior. Use of the include relationship can avoid inconsistencies and duplicate work in multiple use cases. Figure 7 shows an example of an include relationship. Use cases with an include relationship are usually closely related. According to our empirical investigation, if there is a traceability link between the use case U and a class C , then other use cases which have an include relationship with the use case U usually also have traceability links with the class C .

Extend: The extend relationship means adding a new behavior to an existing use case under specified conditions. In this type of relationship, the base use case provides one or more insertion points, and the extending use case provides the behaviors that need to be inserted for these insertion points. An example of an extend relationship is shown in Figure 8. Use cases with extend relationship are usually closely related. According to our empirical research, if there is a traceability link between the a case U and a class C , then other use cases that have an extend relationship with the use case U are very likely to have traceability links with the class C .

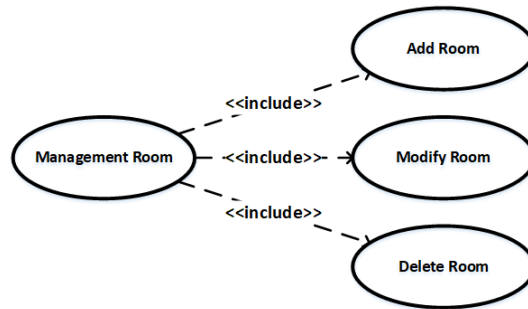


Figure 7. The include relationship between use cases.

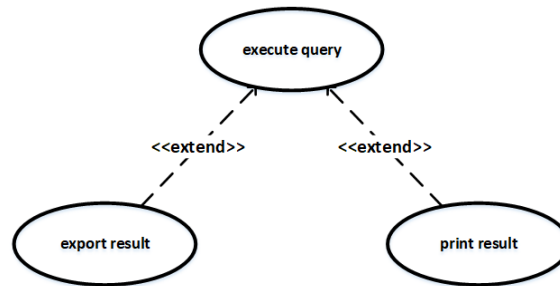


Figure 8. The extend relationship between use cases.

Generalize: The generalize relationship of use cases means that a basic use case can be generalized into multiple sub use cases. If multiple use cases have the same structure and behavior, we can abstract their commonality as the basic use case. An example of a generalize relationship is shown in Figure 9. Use cases with generalization relationships are usually closely related. According to our empirical research, if there is a traceability link between a use case U and a class C , then other use cases that have generalization relationship with use case U usually also have traceability links with the class C .

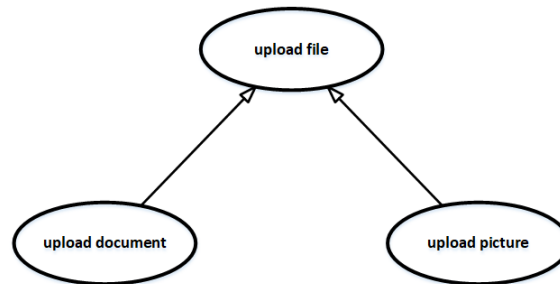


Figure 9. The generalize relationship between use cases.

On the whole, we propose the following use case relationships as features, which are shown in Table 4. Considering such relationships, we have defined the following inference rules based on the semantics of these relationships: Rule 6 means if there is a traceability link between use case s_i and source code c_j , and use case s_k includes use case s_i , then there is a traceability link between use case s_k and source code c_j . Rule 7 means if there is a traceability link between use case s_i and source code c_j , and use case s_k extends use case s_i , then there is a traceability link between use case s_k and source code c_j . Rule 8 means if there is a traceability link between use case s_i and source code c_j , and use case s_i generalizes use case s_k , then there is a traceability link between use case s_k and source code c_j .

$$\begin{aligned}
 & \text{Rule6 : } \text{traceability_link}(\text{use_case}(s_k), \text{source_code}(c_j)) \\
 & \leftarrow \text{traceability_link}(\text{use_case}(s_i), \text{source_code}(c_j)) \\
 & \wedge \text{include}(\text{use_case}(s_k), \text{use_case}(s_i))
 \end{aligned} \tag{7}$$

$$\begin{aligned} \text{Rule7} : & \text{traceability_link}((\text{use_case}(s_k), \text{source_code}(c_j)) \\ & \leftarrow \text{traceability_link}(\text{use_case}(s_i), \text{source_code}(c_j)) \end{aligned} \quad (8)$$

$$\begin{aligned} & \wedge \text{extend}(\text{use_case}(s_k), \text{use_case}(s_i)) \\ \text{Rule8} : & \text{traceability_link}((\text{use_case}(s_k), \text{source_code}(c_j)) \\ & \leftarrow \text{traceability_link}(\text{use_case}(s_i), \text{source_code}(c_j)) \end{aligned} \quad (9)$$

$$\wedge \text{generalize}(\text{use_case}(s_i), \text{use_case}(s_k))$$

We apply these rules to capture more links based on the traceability links output by the inference rules that are defined in Section 3.3.1. In other words, the two types of interrelationships (source code interrelationships and use case interrelationships) are sequentially explored in our approach. Specifically, we traverse the traceability links output from the first step and set the tag of the corresponding link to 1 where it matches any of rules 6–8.

Table 4. The relationship feature of use case in phase 2.

Artifact	Feature	Description
Use Case	Include	The include relationship between use cases.
	Extend	The extend relationship between use cases.
	Generalize	The generalization relationship between use cases.

4. Evaluation

In this section, we detail the design and analysis of our evaluation experiments.

4.1. Datasets

To evaluate the approach proposed in this paper more effectively, we apply it on multiple datasets. We selected eTour, SMOS, Albergate, and eANCI as project repositories that are widely used in traceability recovery evaluation [2–4,17]. eTour is an electronic touristic guide system that we have used in our previous study. It contains 58 use cases, 116 source code classes and 336 correct links. SMOS is a high school student monitoring system that contains 67 use cases, 100 source code classes and 1045 correct links. Albergate is a hotel management system containing 17 use cases, 55 source code classes and 54 correct links. eANCI is a municipalities management system with 140 use cases, 55 source code classes and 567 correct links. The artifact language for the project systems are all Italian, so we need to translate it into English. The correct links are verified, which are called the oracle to analyze the experimental results of the proposed approach. The details of the dataset are shown in Table 5.

Table 5. Details of the datasets.

Project	Source Artifact(#)	Target Artifact(#)	Correct Links	Description
eTour	Use cases (58)	Source Code Classes (116)	336	An electronic touristic guide system
SMOS	Use cases (67)	Source Code Classes (100)	1045	High school student monitoring system
Albergate	Use cases (17)	Source Code Classes (55)	54	Hotel management system
eANCI	Use cases (140)	Source Code Classes (55)	567	Municipalities management system

4.2. Metric

We here adopt two widely used metrics to evaluate our proposed approach, i.e., *precision* and *recall*.

$$precision = \frac{|correct \cap retrieved|}{|retrieved|} \% \quad (10)$$

$$recall = \frac{|correct \cap retrieved|}{|correct|} \% \quad (11)$$

where *correct* represents the set of correct links, and *retrieved* is the set of all links retrieved by the traceability recovery technique. In addition, we use *F1-score* to complement the above two metrics, which is the harmonic mean of those two.

$$F1\text{-score} = \frac{2 \times precision \times recall}{precision + recall} \quad (12)$$

4.3. Research Questions

To fully evaluate our study, we propose a series of research questions in this section.

- **RQ1:** In Phase 1 machine learning recovery, does the addition of the class attribute, method parameter, and method return features to the traceability link vector definition improve traceability recovery?
- **RQ2:** Which is the most appropriate machine learning algorithm for our approach?
 - **RQ2.1:** For the traceability recovery task, which classification algorithm is most effective? Which classification algorithm is most suitable for traceability recovery?
 - **RQ2.2:** What is the most suitable value of parameter *k* when using the KNN classification algorithm for traceability recovery?
 - **RQ2.3:** When using SVM classification algorithm for traceability recovery, which kernel can obtain better experimental results?
- **RQ3:** Which structural information features between source code classes are effective for predicting traceability links based on logical reasoning?
- **RQ4:** Can using logical reasoning rules based on relationship features between use cases further improve traceability recovery?
- **RQ5:** What is the interaction between structural information features of source code components and relationship features of use cases?
 - **RQ5.1:** Is the structural information feature of source code components more important than the relationship features of use cases for traceability recovery?
 - **RQ5.2:** Can the combination of source code and use case relationship features further improve traceability recovery?
- **RQ6:** Can our proposal outperform existing traceability link recovery approaches?

4.4. Experiment Design

In this section, we design a series of experiments in response to the research questions we have proposed. We implement our approach on four data sets that are eTour, SMOS, Albergate and eANCI, as discussed in Section 4.1. We will calculate and present the average results obtained from all the four datasets.

Experiment 1: For the research question **RQ1**, in phase 1 machine learning recovery, we respectively input the traceability link vector with and without the feature dimensions of class attribute, method parameter, and method return into the classification model for traceability recovery.

Experiment 2: For the research questions *RQ2*, *RQ2.1*, *RQ2.2*, and *RQ2.3*, we use different classification algorithms and set different parameters to observe which classification algorithm with specific parameters can achieve better results. We will use the results of Experiment 2 to choose the suitable algorithms with proper parameters for later experiments.

Experiment 3: For the research question *RQ3*, in Phase 2 Logical Reasoning Recovery, for each structural information feature between source code classes, we use the corresponding logical reasoning rule separately to capture traceability links.

Experiment 4: For the research question *RQ4*, we have designed a series of experiments to measure the effectiveness of capturing relationships among use cases when defining the inference rules. Specifically, we first compare the following two cases to see whether use case relationships contribute to the original approach: (1) applying only the machine learning approach; (2) applying the machine learning approach and inference rules derived from use case interrelationships. Then, we compare the following two cases to see whether use case relationships can render additional values on the top of the initial hybrid approach: (1) applying the machine learning approach and inference rules derived from source code classes; (2) applying the machine learning approach and inference rules derived from both source code classes and use cases.

Experiment 5.1: For the research question *RQ5.1*, we observed the experimental results in the following two cases: (1) After phase 1 machine learning recovery, additionally using logical reasoning rules based on source code structural information features to perform traceability recovery. (2) After phase 1 machine learning recovery, additionally using logical reasoning rules based on use case relationship features to perform traceability recovery.

Experiment 5.2: For the research question *RQ5.2*, we observe the experimental results of the following cases: (1) Using logical reasoning rules based on source code structural information features to perform traceability recovery. (2) Using logical reasoning rules based on use case relationship features to perform traceability recovery. (3) Combine using logical reasoning rules based on source code structural information features and use case relationship features.

In order to answer the research question *RQ6*, we compared our approach with other research methods on multiple datasets.

Experiment 6.1: On the eTour data set, we compare our approach with Combine IR [17], Relational Topic Modeling [2], UD-CSTI [3], and R2C [4].

Experiment 6.2: On the Albergate dataset, we compare the proposed method with the VSM-based method [14].

Experiment 6.3: On the SMOS dataset, we compare the proposed in this paper with the UD-CSTI method [3].

4.5. Results and Analysis

In this section, we present and analyze the results of our experiments. Figure 10 shows the result of *Experiment 1*. In our previous research [10], we utilized the class name, class comment, method name and method comment source code features. In this paper, we further utilize the class attribute, method parameter and method return features. As can be seen in Figure 10, after adding the class attribute, method parameter and method return features for the traceability link vector definition, the precision, recall, and F1-score are all slightly improved.

Therefore, we can answer the *RQ1*. We propose that adding class attribute, method parameter, and method return features to the traceability link vector definition can improve traceability recovery. Class attribute, method parameter, and method return are meaningful features for traceability recovery.

Next we conduct *Experiment 2*. Figure 11 shows the classification result on eTour. It can be seen from Figure 11 that Decision Tree, KNN, Random Forest, and GBDT can achieve good performance, and their performance is similar. In this paper, we combine using the decision tree, KNN, random forest and GBDT classification models for traceability recovery, merging the results of these classifier

model. The SVM algorithm can achieve high precision, but its recall and F1-score are low. The results obtained by logistic regression and multinomial native bayes are low.

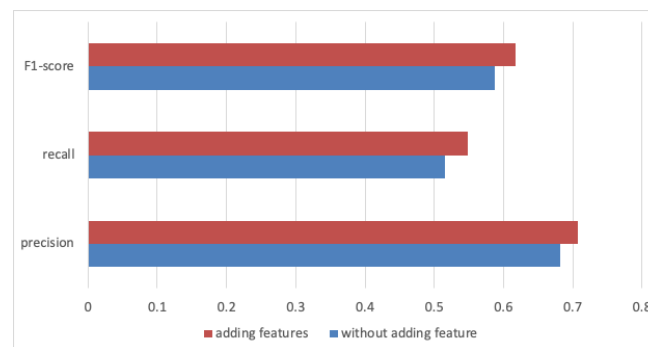


Figure 10. Comparison of before and after adding features (class attribute, method parameter and method return) to the traceability link vector definition.

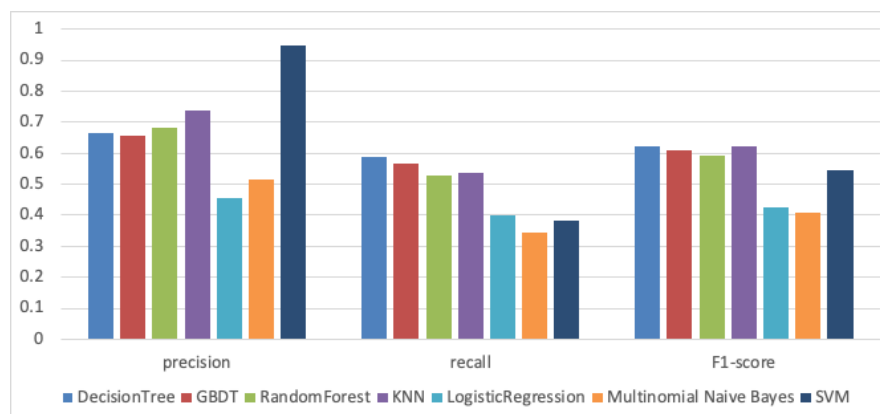


Figure 11. Results of various classification models that are applied in the machine learning recovery phase.

Therefore, we can answer the *RQ2* and *RQ2.1*. We find that decision tree, KNN, random forest and GBDT are all suitable classification algorithms for traceability recovery. Logistic regression, multinomial native bayes are not suitable for the traceability recovery task. If the traceability recovery task needs to achieve high precision without considering recall, an SVM classification model is a good choice. In our study, we combine the decision tree, KNN, random forest and GBDT classification models for traceability recovery, merging the output links of these classifier models.

Next, we explore the suitable value of the k parameter for KNN. Generally, if the value of k is too small, noise will have a greater impact on the prediction, which will cause deviations. A decrease in k value means that the overall model becomes more complex and prone to overfitting. If the value of k is too large, the approximate error of learning will increase, and instances far from the input target point will have a detrimental effect on the prediction.

It can be seen from Figure 12 that as the value of k increases, precision improves slightly, and recall and F1-score decrease. When the parameter k is set to a smaller value, a better classification effect can be achieved. Therefore, when the value of k is smaller, the classification effect is better. In response to *RQ2.2*, the value of k is 5 in our study.

As can be seen from the Figure 13, the experimental results of the SVM classification algorithm are highest when using an rbf kernel, and lowest when using a sigmoid kernel. As the answer to the *RQ2.3*, we propose using rbf as the kernel when using SVM.

Although the recall and F1-score obtained by using SVM for traceability recovery are low, it can achieve a high precision. When it is necessary to obtain a high precision without considering recall, SVM is a suitable choice.

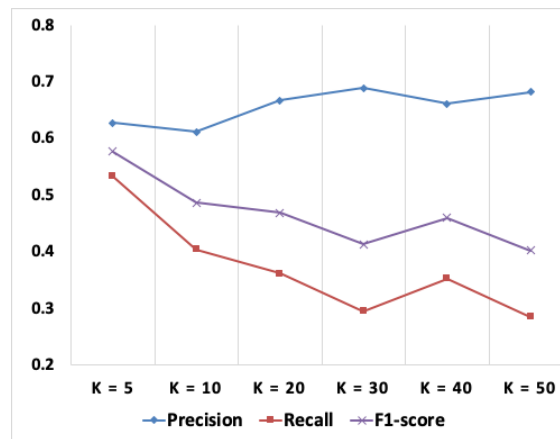


Figure 12. Results of using the k-nearest neighbors (KNN) classification model for traceability recovery with different k values on the eANCI dataset.

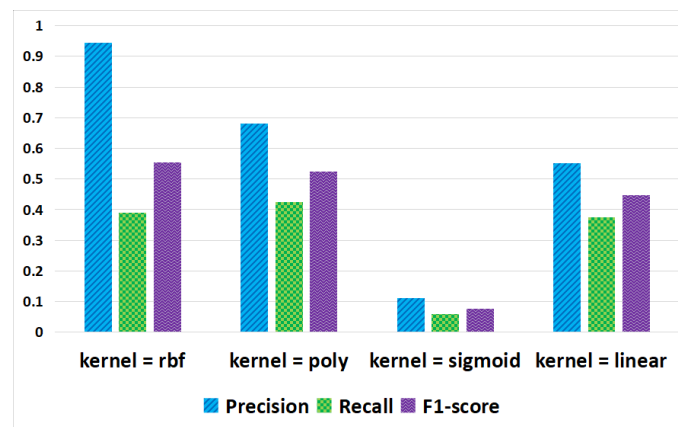


Figure 13. Results of using the support vector machine (SVM) classification model for traceability link recovery with different kernel functions on the SMOS dataset.

The results of *Experiment 3* can be seen in Figure 14. By applying the implement, inherit, return, attribute and method call rules to capture traceability links after machine learning recovery, our approach can achieve higher results, especially for the implement, inherit and method call features. Using these source code structural information feature rules in combination can further improve the results.

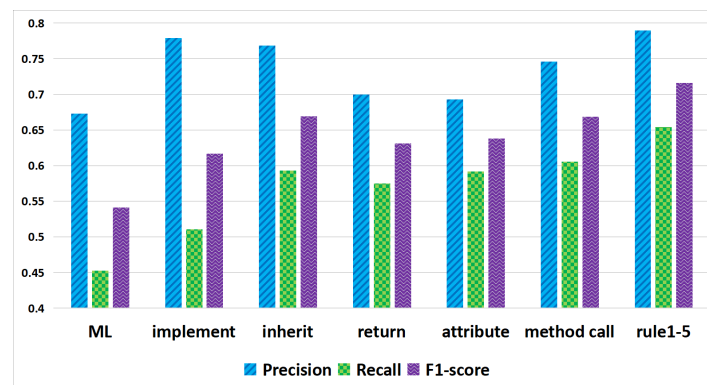


Figure 14. Results of using various structural information features of source code to perform traceability link recovery.

Therefore, we can answer the *RQ3*. We propose that using logical reasoning rules based on the features of implement, inherit, return, attribute, method call can effectively capture the traceability links. Implement, inherit, return, attribute, and method call are all important features for traceability recovery.

According to the *Experiment 4* results in Figure 15, it can be seen that after applying logical reasoning rules based on use case relationship features to capture more traceability links, the precision, recall, and F1-score of traceability recovery are further improved.

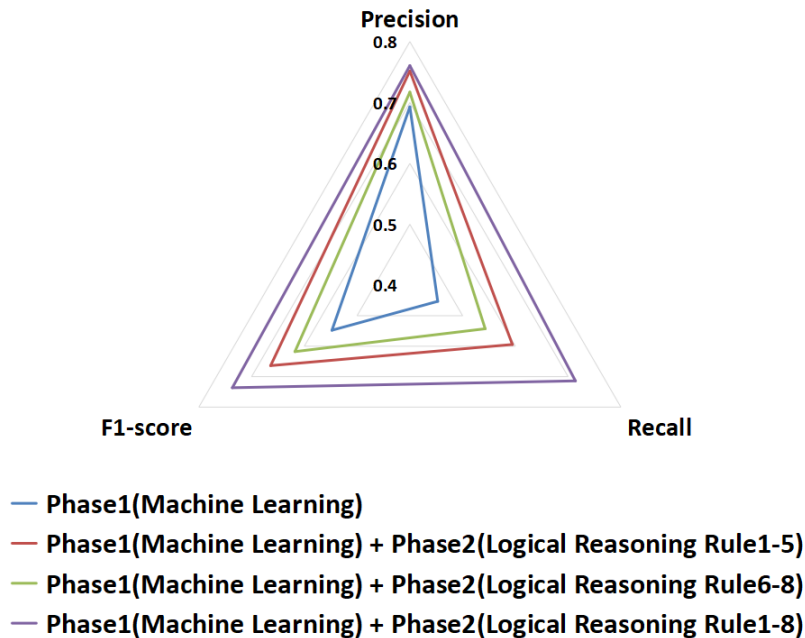


Figure 15. Comparison of different combinations of our proposal on the SMOS dataset. Logical reasoning rule 1–5 are designed based on source code interrelationships, while logical reasoning 6–8 are designed based on use case interrelationships.

In response to *RQ4*, we propose that the relationship features of use cases can help to improve traceability recovery. Using the rules based on the use case relationship features can further capture more traceability links. The use case relationship features are significant for traceability recovery.

The result of *Experiment 5.1* can be seen from the Figure 15. Using logical reasoning rules based on source code structural information features to perform traceability recovery, the improvement of the traceability recovery is more obvious, when compared to using logical reasoning rules based on use case relationship features.

Therefore, as the answer to *RQ5.1*, we believe that the features based on structural information between source code classes are more important than the relationship features between use cases, and can play a greater role. We speculate that this is because in the project system, the source code structural information is larger and more complex than for the use case relationships.

The result of *Experiment 5.2* can be seen from Figure 15. The experimental results shows that using the source code structural information features and use case relationship features in combination can further improve traceability recovery, particularly the recall.

In response to *RQ5.2*, we propose that the source code structural information features and use case relationship features have gain effect on each other, and the combination of using the source code structural information features and use case relationship features can capture more traceability links.

In conclusion for *RQ5*, the structural information is more important than the relationship between use cases, but they can improve each other.

The result of *Experiment 6.1* can be seen from Figure 16 that the precision, recall, and F1-score obtained by our approach are higher than other research methods [2–4,17] on the eTour data set.

According to the result of *Experiment 6.2* shown in Figure 17, the recall obtained by our method is lower than the VSM-based method [14], but the precision, F1-score is higher than the VSM-based method on the Albergate dataset.

According to the result of *Experiment 6.3* shown in Figure 18, on the SMOS dataset it can be seen that the precision, recall, and F1-score obtained by our method are higher than the UD-CSTI method [3], especially in precision and F1-score.

Therefore, as the answer of *RQ6*, the method proposed in this paper explore features of artifacts effectively which can achieve very high evaluation results. Our approach is an effective method for traceability recovery, and surpasses the methods proposed in related research.

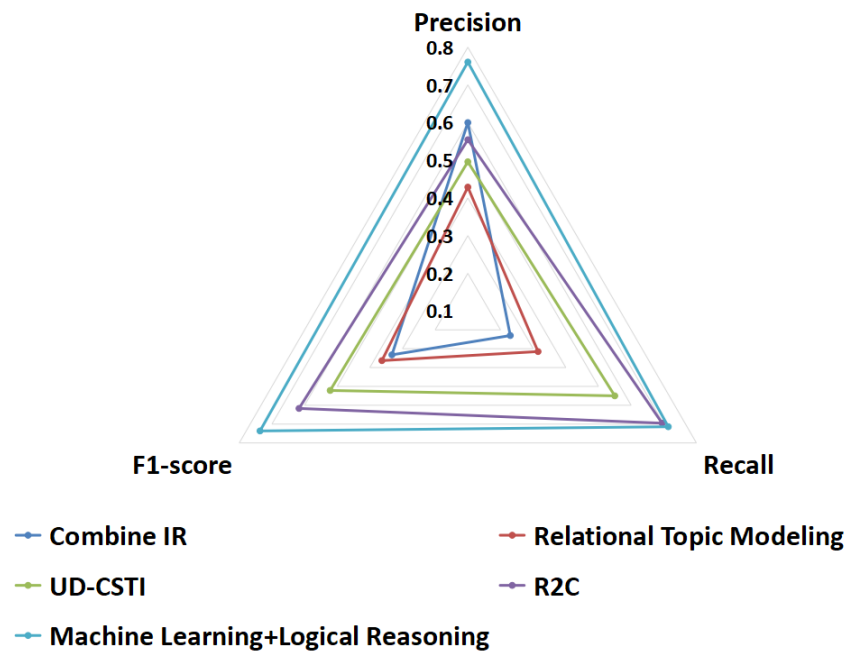


Figure 16. The result of comparing our proposed approach with methods proposed by related researchers on the eTour dataset.

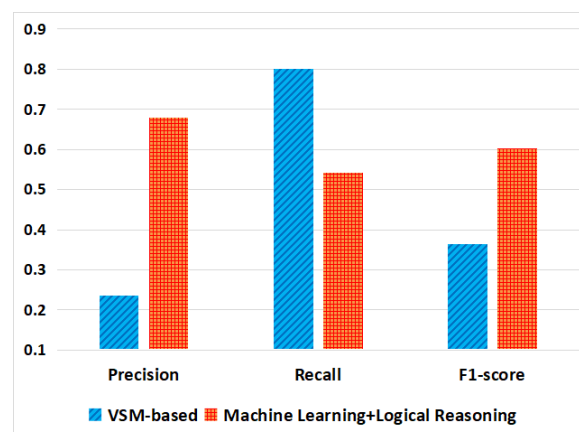


Figure 17. The result of comparing our proposed approach with methods proposed by related researchers on the Albergate dataset.

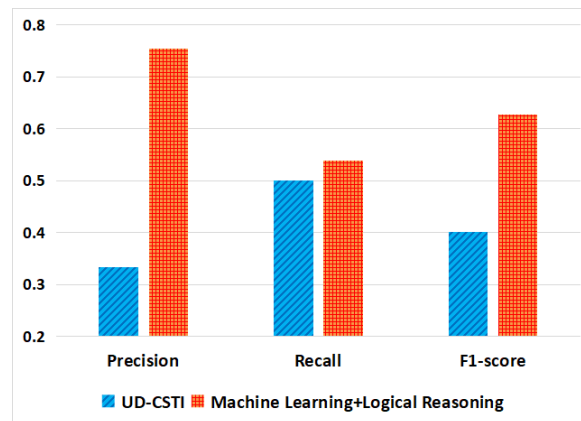


Figure 18. The result of comparing our proposed approach with methods proposed by related researchers on the SMOS dataset.

5. Threats to Validity

We discuss the validity of our experiments by following the classification of threats used in [36]. In particular, we discuss internal validity, construct validity, external validity, and conclusion validity, respectively.

Internal validity takes into account the relationship between approach and result. The dataset projects used in our study are developed in Italian, and we translate the content of the software artifacts into English. Some semantics of the software artifacts may be lost during translation, so the translation process may affect the results of our approach. The classification algorithm used by the study may have an impact on traceability link recovery. Supervised learning contains a variety of classification algorithms. This paper adopts the decision tree, KNN, random forest and GBDT as the classification models.

Construct validity describes whether the theoretical construction of our proposed method is correct. The approach that extracting features based on domain knowledge and defining rules to capture links directly are valid, which has been demonstrated by empirical research. Whether all traceability links conform to the rules requires further investigation.

External validity discusses whether our approach can be extended from the experimental environment to the development of industrial systems. The software project repositories used in this paper are implemented by students. The business logic of industrial projects is often very complex. So the software artifacts of industrial projects are much more complicated than they are in our study. However, these software projects are comparable to the repositories used by other studies.

Conclusion validity usually involves the ability of our approach to reach the correct conclusion. Due to the scale of the data set, it may affect the effectiveness of machine learning classification algorithms. To this end, we use k-fold cross-validation to overcome this problem.

6. Conclusions and Future Work

Software requirements traceability recovery creates links between software artifacts, which is important for software comprehension. Given a large amount of redundant irrelevant information in the software artifacts, we propose not to treat them as a whole but investigate and focus only on essential features for recovering traceability links. Specifically, we select meaningful features of software artifacts and propose an effective method to embed such meaning into training vectors. A set of useful machine learning algorithms are adopted in order to discover a wide spectrum of traceability links.

In addition, we also propose using logical reasoning to further discover traceability links in addition to the results derived from the machine learning analysis. To this end, we investigate the semantics of corresponding artifacts in detail and propose a set of meaningful inference rules.

The experimental results show that our proposed method can outperform other approaches by achieving higher precision and recall, substantially improving F1-score.

In the future, we plan to find more significant features of artifacts to improve traceability recovery. Note that we explore artificial intelligence techniques by combining machine learning and logical reasoning, while the machine learning part is not as explainable as the logical reasoning part. In order to iteratively improve the performance of our approach, it is essential to open the black box of the machine learning part, making it explainable [37]. Specifically, the explainable evidence derived from the machine learning part will shed light on the design of the logical reasoning part, i.e., disclosing meaningful inference rules to identify traceability recovery rules.

As we clarified before, our proposal is not supposed to fully automate the traceability link recovery task due to its inherent difficulty. Instead, our approach would serve as an auxiliary tool that help people to recover traceability links in a more efficient manner than manual analysis. As such, another branch of our future is to investigate the specific way of combining manual and automatic efforts, pragmatically helping people to efficiently use our approach to recover traceability links with the least human efforts. Finally, we are seeking for practical scenarios from industry to pragmatically examine the utility of our approach.

Author Contributions: methodology, T.L.; software, S.W.; investigation, T.L. and S.W.; writing—original draft preparation, S.W.; writing—review and editing, T.L. and D.L.; supervision, T.L. and Z.Y.; project administration, Z.Y.; funding acquisition, Z.Y. and T.L. All authors have read and agreed to the published version of the manuscript.

Funding: National Natural Science of Foundation of China (No.61902010, 61671030), Beijing Excellent Talent Funding-Youth Project (No.2018000020124G039)

Acknowledgments: This work is partially supported by the National Natural Science of Foundation of China (No.61902010, 61671030), Beijing Excellent Talent Funding-Youth Project (No.2018000020124G039), and Engineering Research Center of Intelligent Perception and Autonomous Control, Ministry of Education.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Asuncion, H.U.; Asuncion, A.U.; Taylor, R.N. Software traceability with topic modeling. In Proceedings of the 2010 ACM/IEEE 32nd International Conference on Software Engineering, Cape Town, South Africa, 1–8 May 2010; Volume 1, pp. 95–104.
2. Gethers, M.; Oliveto, R.; Poshyvanyk, D.; De Lucia, A. On integrating orthogonal information retrieval methods to improve traceability recovery. In Proceedings of the 2011 27th IEEE International Conference on Software Maintenance (ICSM), Williamsburg, VA, USA, 25–30 September 2011; pp. 133–142.
3. Panichella, A.; McMillan, C.; Moritz, E.; Palmieri, D.; Oliveto, R.; Poshyvanyk, D.; De Lucia, A. When and how using structural information to improve ir-based traceability recovery. In Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering, Genova, Italy, 5–8 March 2013; pp. 199–208.
4. Zhang, Y.; Wan, C.; Jin, B. An empirical study on recovering requirement-to-code links. In Proceedings of the 2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), Shanghai, China, 30 May–1 June 2016; pp. 121–126.
5. Capobianco, G.; De Lucia, A.; Oliveto, R.; Panichella, A.; Panichella, S. On the role of the nouns in IR-based traceability recovery. In Proceedings of the 2009 IEEE 17th International Conference on Program Comprehension, Vancouver, BC, Canada, 17–19 May 2009; pp. 148–157.
6. Asuncion, H.U.; Taylor, R.N. Capturing custom link semantics among heterogeneous artifacts and tools. In Proceedings of the 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering, Vancouver, BC, Canada, 18 May 2009; pp. 1–5.
7. Raghavan, V.V.; Wong, S.M. A critical analysis of vector space model for information retrieval. *J. Am. Soc. Inf. Sci.* **1986**, *37*, 279–287.
8. Hofmann, T. Probabilistic latent semantic indexing. In Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Berkeley, CA, USA, 15–19 August 1999; pp. 50–57.

9. McMillan, C.; Poshyvanyk, D.; Reville, M. 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, Vancouver, BC, Canada, 18 May 2009; pp. 41–48.
10. Wang, S.; Li, T.; Yang, Z. Exploring Semantics of Software Artifacts to Improve Requirements Traceability Recovery: A Hybrid Approach. In Proceedings of the 2019 26th Asia-Pacific Software Engineering Conference (APSEC), Putrajaya, Malaysia, 2–5 December 2019; pp. 39–46.
11. Hayes, J.H.; Dekhtyar, A.; Osborne, J. Improving requirements tracing via information retrieval. In Proceedings of the 11th IEEE International Requirements Engineering Conference, Monterey Bay, CA, USA, 12 September 2003; pp. 138–147.
12. Lucia, A.D.; Fasano, F.; Oliveto, R.; Tortora, G. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Trans. Softw. Eng. Methodol.* **2007**, *16*, 13-es, doi:10.1145/1276933.1276934.
13. De Lucia, A.; Di Penta, M.; Oliveto, R.; Panichella, A.; Panichella, S. Improving ir-based traceability recovery using smoothing filters. In Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension, Kingston, ON, Canada, 22–24 June 2011; pp. 21–30.
14. Lucia, D. Information retrieval models for recovering traceability links between code and documentation. In Proceedings of the Proceedings 2000 International Conference on Software Maintenance, San Jose, CA, USA, 11–14 October 2000; pp. 40–49.
15. Topsøe, F. Jensen-shannon divergence and norm-based measures of discrimination and variation. *preprint* **2003**.
16. Blei, D.M.; Ng, A.Y.; Jordan, M.I. Latent dirichlet allocation. *J. Mach. Learn. Res.* **2003**, *3*, 993–1022.
17. Oliveto, R.; Gethers, M.; Poshyvanyk, D.; De Lucia, A. On the equivalence of information retrieval methods for automated traceability link recovery. In Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension, Braga, Portugal, 30 June–2 July 2010; pp. 68–71.
18. Chang, J.; Blei, D.M. Hierarchical relational models for document networks. *Ann. Appl. Stat.* **2010**, *4*, 124–150.
19. Capobianco, G.; Lucia, A.D.; Oliveto, R.; Panichella, A.; Panichella, S. Improving IR-based traceability recovery via noun-based indexing of software artifacts. *J. Softw. Evol. Process* **2013**, *25*, 743–762.
20. Buckner, J.; Buchta, J.; Petrenko, M.; Rajlich, V. JRipples: A tool for program comprehension during incremental change. In Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05), St. Louis, MO, USA, 15–16 May 2005; pp. 149–152.
21. Rajlich, V.; Gosavi, P. Incremental change in object-oriented programming. *IEEE Softw.* **2004**, *21*, 62–69.
22. Miller, G.A. WordNet: A lexical database for English. *Commun. ACM* **1995**, *38*, 39–41.
23. De Marneffe, M.C.; MacCartney, B.; Manning, C.D. Generating typed dependency parses from phrase structure parses. *Lrec* **2006**, *6*, 449–454.
24. Kuang, H.; Nie, J.; Hu, H.; Lü, J. Improving automatic identification of outdated requirements by using closeness analysis based on source code changes. In *National Software Application Conference*; Springer: Berlin, Germany 2016; pp. 52–67.
25. Kuang, H.; Nie, J.; Hu, H.; Rempel, P.; Lü, J.; Egyed, A.; Mäder, P. Analyzing closeness of code dependencies for improving IR-based Traceability Recovery. In Proceedings of the 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), Klagenfurt, Austria, 20–24 February 2017; pp. 68–78.
26. Kuang, H.; Mäder, P.; Hu, H.; Ghabi, A.; Huang, L.; Lü, J.; Egyed, A. Can method data dependencies support the assessment of traceability between requirements and source code? *J. Softw. Evol. Process* **2015**, *27*, 838–866.
27. Kuang, H.; Mäder, P.; Hu, H.; Ghabi, A.; Huang, L.; Jian, L.; Egyed, A. Do data dependencies in source code complement call dependencies for understanding requirements traceability? In Proceedings of the 2012 28th IEEE International Conference on Software Maintenance (ICSM), Trento, Italy, 23–28 September 2012; pp. 181–190.
28. Ali, N.; Gueheneuc, Y.G.; Antoniol, G. Trust-based requirements traceability. In Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension, Kingston, ON, Canada, 22–24 June 2011; pp. 111–120.

29. Diaz, D.; Bavota, G.; Marcus, A.; Oliveto, R.; Takahashi, S.; De Lucia, A. Using code ownership to improve ir-based traceability link recovery. In Proceedings of the 2013 21st International Conference on Program Comprehension (ICPC), San Francisco, CA, USA, 20–21 May 2013; pp. 123–132.
30. Ali, N.; Sharafi, Z.; Guéhéneuc, Y.G.; Antoniol, G. An empirical study on requirements traceability using eye-tracking. In Proceedings of the 2012 28th IEEE International Conference on Software Maintenance (ICSM), Trento, Italy, 23–28 September 2012; pp. 191–200.
31. Ali, N.; Sharafi, Z.; Guéhéneuc, Y.G.; Antoniol, G. An empirical study on the importance of source code entities for requirements traceability. *Empir. Softw. Eng.* **2015**, *20*, 442–478.
32. Lucassen, G.; Dalpiaz, F.; van der Werf, J.M.E.; Brinkkemper, S.; Zowghi, D. Behavior-driven requirements traceability via automated acceptance tests. In Proceedings of the 2017 IEEE 25th International Requirements Engineering Conference Workshops (REW), Lisbon, Portugal, 4–8 September 2017; pp. 431–434.
33. Guo, J.; Cheng, J.; Cleland-Huang, J. Semantically enhanced software traceability using deep learning techniques. In Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), Buenos Aires, Argentina, 20–28 May 2017; pp. 3–14.
34. Sultanov, H.; Hayes, J.H. Application of reinforcement learning to requirements engineering: requirements tracing. In Proceedings of the 2013 21st IEEE International Requirements Engineering Conference (RE), Rio de Janeiro, Brazil, 15–19 July 2013; pp. 52–61.
35. Marcén, A.C.; Lapeña, R.; Pastor, Ó.; Cetina, C. Traceability Link Recovery between Requirements and Models using an Evolutionary Algorithm Guided by a Learning to Rank Algorithm: Train Control and Management Case. *J. Syst. Softw.* **2020**, *163*, 110519.
36. Opdahl, A.L.; Sindre, G. Experimental comparison of attack trees and misuse cases for security threat identification. *Inf. Softw. Technol.* **2009**, *51*, 916–932.
37. Holzinger, A. From machine learning to explainable AI. In Proceedings of the 2018 World Symposium on Digital Intelligence for Systems and Machines (DISA), Košice, Slovakia, 23–25 August 2018; pp. 55–66.

Publisher’s Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).