Multi-layered Approach for Recovering Links between Bug Reports and Fixes

Anh Tuan Nguyen
anhnt@iastate.edu

Tung Thanh Nguyen
tung@iastate.edu

Hoan Anh Nguyen
hoan@iastate.edu

Tien N. Nguyen
tien@iastate.edu

Electrical and Computer Engineering Department
Iowa State University
Ames, IA 50011, USA

ABSTRACT

The links between the bug reports in an issue-tracking system and the corresponding fixing changes in a version repository are not often recorded by developers. Such linking information is crucial for research in mining software repositories in measuring software defects and maintenance efforts. However, the state-of-the-art bug-to-fix link recovery approaches still rely much on textual matching between bug reports and commit/change logs and cannot handle well the cases where their contents are not textually similar.

This paper introduces MLink, a multi-layered approach that takes into account not only textual features but also *source code features* of the changed code corresponding to the commit logs. It is also capable of learning the *association* relations between the *terms* in bug reports and the *names* of entities/components in the changed source code of the commits from the established bug-to-fix links, and uses them for link recovery between the reports and commits that do not share much similar texts. Our empirical evaluation on real-world projects shows that MLink can improve the state-of-the-art bug-to-fix link recovery methods by 11-18%, 13-17%, and 8-17% in F-score, recall, and precision, respectively.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Algorithms, Documentation, Experimentation, Measurement

Keywords

Bug-to-Fix Links, Bugs, Fixes, Mining Software Repository

1. INTRODUCTION

During the development of a software system, software engineers produce and manage their source code in a version repository such as CVS [25] or SVN [38]. When erroneous behaviors are found in the system, the users/testers/developers reported the issues, which are stored in a different repository called a bug-tracking system (e.g. Bugzilla [39]). The *links* between *bug reports* in a bug tracking system and *corresponding committed fixing changes* in a source code repository (called *bug-to-fix* links) are not often recorded [11]. This type of links is crucial for many research approaches in mining software repositories (MSR) because the information on software quality and maintenance efforts such as the measurements on software defects and bug-fixing changes can be derived from those links [10, 34].

Recognizing the importance of bug-to-fix links, several approaches have been proposed to recover such links from bug reports in a bug tracking system and commit logs in a version archive. The widely used, traditional link recovery approach [31, 35, 36] is based on the mining of key phrases and common text patterns that the fixers have noted in the change logs such as "Fixed the issue XYZ", "Fix bug ID ...", etc. Unfortunately, recent studies have found that those traditional pattern-based heuristics likely result in biased data with many false negatives, i.e. missing links [9, 10, 28]. The authors reported that developers may not always document bug references in commit logs. They found that only a portion of bug fixes are actually linked to change logs in version archives. Thus, the software quality and maintenance measurements derived from the biased resulting link data are not accurate and affect much MSR research results [10, 20].

To address those issues, Wu *et al.* [34] recently proposed ReLink, a bug-to-fix link recovery technique based on three key heuristics: 1) time interval, i.e., a bug should be fixed after the creation and before the closure of a bug report; 2) text similarity between bug reports and change logs; and 3) the developers responsible for a bug are typically the committers of the bug-fixing change. However, the authors reported that accuracy of ReLink is affected much by the reports and logs that are not textually similar [34]. This is common because the bug reporter describes the issue itself from the *users' perspective* (e.g. the error scenarios, interactions, re-producing steps, etc) while the fixer records how (s)he has fixed the bug from the *developers' point of view.*

Aiming to improve further bug-to-fix link recovery accuracy, we introduce MLink approach that explores not only textual features and traditional heuristics, but also *source code features* of the changed source files that correspond to the commit/change logs. In other words, in addition to the commit logs, which are just the textual information associated with the fixes, MLink takes advantage of the actual

Bug Report #86
(Reported by whoister...@gmail.com, Oct 3, 2008
Type: Defect, Status: Fixed, Closed: Oct 6, 2008)
Summary: NoSuchMethodError: No such method getHeight().
Description: ...
Expect decoded output, but see Error:
NoSuchMethodError: No such method getHeight().
...
Got the following trace:
com/google/zxing/common/BaseMonochromeBitmapSource.estimateBlackPoint. ALERT: java/lang/NoSuchMethodError: No such method getWidth()

Comment 1 by project member srowen, Oct 4, 2008
Exactly, sounds like the same thing. BaseMonochromeBitmapSource is an abstract class which implements an interface defining methods like getHeight() and getWidth(). The abstract class doesn't have to define an implementation for all the interface methods, and it doesn't define these ...But the Nokia JVMs don't seem to accept that...The workaround is that we can just add
public abstract int getHeight();
public abstract int getWidth();
...
Comment 4 by project member srowen, Oct 6, 2008
I will add those other two methods...

Figure 1: Bug Report #86 in the ZXing Project

Commit Log #599
(Author: srowen, Date: Oct 4, 2008)
Log message: Added redundant abstract method declarations to maybe work around problems on Nokias.

Figure 2: Commit Log #599 to Fix Bug #86

Commit Change #599
Affected files: Modify /trunk/core/src/com/google/zxing/-common/BaseMonochromeBitmapSource.java
Diff:

```
public abstract class BaseMonochromeBitmapSource implements
      MonochromeBitmapSource {
+ //These two methods should not need to exist because they are
       defined in the interface that
+ //this abstract class implements. However this seems to cause
      problems on some Nokias.
+ //So we write these redundant declarations.
+ public abstract int getHeight();
+ public abstract int getWidth();
}
```

Figure 3: Committed Change #599 to Fix Bug #86

source code changes for the fixes themselves. MLink uses different cascading layers to retrieve additional links after each layer. After applying traditional pattern-based heuristics and time interval filtering, it extracts the following code features to detect links: 1) the *names* of program entities and system components that appear frequently and are significant in both *bug records* (including descriptions, summaries and bug comments) and *commits* (including commit logs, changed source code and its textual inline comments), and 2) the *code fragments* appearing in both bug reports (and their bug comments), and the commits (note: this type of code in a bug report is likely to be involved in the corresponding fix, e.g., a recommended patch from bug commenters).

After detecting with code features, MLink uses textual features for further link recovery. In addition to text matching between a bug report and a commit log as in ReLink, MLink also considers the textual comments *within* the changed source files of a commit (We found several cases in which the fixers used parts of the descriptions of the bugs being fixed to explain about the reasons for their fixes in inline comments). Importantly, to recover the links where the texts in both bug records and commits are *not* quite similar, we provide an association method that learns the *association relations* between the *terms* in the bug records and the *names* of entities/components in the changed code of the commits from the established bug-to-fix links detected by the traditional heuristics. From the association frequencies, MLink derives the association relation between a bug record and a commit to determine if the link exists between them.

Our empirical evaluation on the real-world projects showed that MLink achieves a very high level of accuracy with 87-93% in F-score, 85-90% in recall, and 82-97% in precision. It is able to improve the state-of-the-art approach ReLink [34] by 6-11% in F-score, 4-13% in recall, and 5-8% in precision. We also showed that MLink can give better estimation values for the software quality and maintenance metrics including the percentage of the buggy files in a project and the percentage of bug-fixing changes over the total number of code changes. The contributions of this paper include

1. MLink, a multi-layered bug-to-fix link recovery approach that takes advantage of both textual and code features,
2. An empirical evaluation to show its accuracy in link recovery and estimating code quality and maintenance metrics.

Next section presents motivating examples. Details of MLink are described in Sections 4-7. Section 8 is for our evaluation. Related work is in Section 9. Conclusions appear last.

2. MOTIVATING EXAMPLES

Let us explain a few examples motivating MLink approach.

2.1 Linking via Code Features

A. Important Concepts

Figure 1 shows a bug record in the bug tracking repository of the ZXing project [37], an open-source barcode image processing software. A **bug record/report** usually consists of

1. A short **summary** of the issue(s), e.g. "NoSuchMethodError: No such method getHeight()" in Figure 1,
2. A textual **description** on the issue(s) (Both the summary and description fields are provided by a bug reporter),
3. A list of **bug comments**. The bug tracking system allows the developers in the project or users to discuss the reported issue(s), e.g. the comment 1 in Figure 1. Sometimes, those commenters even express their thoughts on potential fixes for the bug, e.g. in comments 1 and 4,
4. Associated **meta-data** such as type (defect), status (fixed, closed, reopen, duplicate, invalid), priority, bug reporter, fixer, commenters, report time, closed time, etc.

We also had checked in the version control repository of the ZXing project and found the corresponding fix for the issue #86. Figure 2 shows its **commit log**, which is a textual description about that set of **code changes** which are committed to the version control repository. Because that set of code changes (called a **change set**) was for fixing the issue, we call it a **fixing change set** or a **fix**. In addition to fixing changes, developers might make other types of code changes for enhancement, improvement, etc. A change set or a fix can be involved with multiple changes to different source files. Figure 3 shows the change set at the revision #599 to fix the issue #86 in Figure 1.

In general, the issue(s) in a bug report can be fixed by multiple fixing change sets (i.e. fixes) committed at different transaction/time. On the other hand, each change set committed at a certain time can also fix one or multiple issues in different bug reports. That is, a bug report can be linked to one or multiple fixes, and a fix can be linked to one or multiple bug reports. In the cases which involve multiple bug reports or multiple fixing change sets, we will separate them and consider a **link** as a connection between a bug report and a change set. We use the word 'change set' and '**commit**' interchangeably. In addition to the commit log, a commit also has its associated **meta-data** such as the committer, the committing date, and a list of changed files.

B. Observations

1. The traditional pattern-based approach [35, 36] does not work in this case because the commit log does not contain common patterns. That approach relies on the hints from developers about bug fixing in the commit logs via common phrases, e.g. 'fix the issue #XYZ', 'bug #123', etc.

2. In this case, the bug report and its corresponding commit log are not *textually similar*. The bug reporter describes the issue from the *users' perspective* (e.g. the output message). In contrast, the bug fixer describes the fixing changes from the *developers' point of view* (e.g. adding abstract method declarations) because (s)he writes more about *how* (s)he fixes the issue, than *what* bug (s)he has fixed. That is, the bug reporter describes the issue itself while the fixer records how (s)he has fixed it. This is reasonable because the commit log is designed as part of the version control repository to help developers to record their notes on any change set. In their study, Wu *et al.* [34] also reported many similar cases where the texts of the bug records are not similar to those of the commit logs, leading to inaccuracy in their tool. Therefore, the existing link recovery techniques [34, 36] that rely much on *textual similarity between bug reports and commit logs* do not work in this case. This implies that automatic link recovery for bug reports and fixes should not solely rely on textual patterns or textual similarity between the reports and commit logs of the fixes.

Code Features. Aiming to find an additional mechanism for such link recovery, we explored further the corresponding commit #599 for the bug report #86. Instead of examining only the commit log (Figure 2) of the fix as in the state-of-the-art bug-to-fix link recovery techniques [34, 35, 36], we also investigated **the fix itself** (Figure 3), i.e. the fixing changes that developers made to the source files in order to fix the issue #86. From that fixing code, we observe that

1. In addition to a commit log, the fixing changes and corresponding changed source files are a crucial part of a fix and an important source of features to be used for link recovery.

2. The fixing changes (Figure 3) contain the program entity getHeight, which is mentioned in the bug report. The program entities (e.g. getWidth, getHeight) in the changed file (BaseMonochromeBitmapSource) were discussed in the comments of a bug record (see comment 1, Figure 1), as well as in the summary and the description of the bug report. Thus, the names of program entities are an important type of features from the fix that could help in link recovery.

3. The changed source file implements certain components/functions of the system. One of those is erroneously implemented, leading to the bug report about the issue(s) on that component. Thus, the terms describing those components (e.g. Nokia, MonochromeBitmapSource) can appear in both textual comments of the bug report and changed source code. Matching them can help in automatic link recovery.

4. In the comment 1 of Figure 1, a commenter suggests a potential patch (public abstract in getHeight();...) and that code fragment was actually used in the fix (Figure 3). Using such patch, we could recover the link between them.

Bug Report #631
(Reported by giovanni...@gmail.com, Nov 20, 2010)...
Summary: Wrong characters in Data Matrix decoding
Description:
The attached 2-D barcode should give:
`http://www.prismaindustriale.com`
but instead it gives:
`http:/awww.prismaindustriale9com` ...
Comment 1 by project member srowen, Nov 25, 2010
I got lucky and spotted the problem pretty fast...

Commit #1670
(Author: srowen, Date: Nov 25, 2010)
Log message: Minor fix to carry shift value across input triads
Affected files:
Modify /trunk/core/src/com/google/zxing/datamatrix/decoder/DecodedBitStreamParser.java

```
  final class DecodedBitStreamParser {
    private static void decodeTextSegment(...) {...
      boolean upperShift = false;
      int[] cValues = new int[3];
+ int shift = 0;
      do { ...
        parseTwoBytes(firstByte, bits.readBits(8), cValues);
– int shift = 0;
        for (int i = 0; i < 3; i++) {...
```

Figure 4: Bug Report #631 and corresponding Fix #1670

Table 1: Term Occurrences in Reports and Fixes

Word	#BRs	Code token	Fix	CoOccur
decoding	46	DecodedBitStreamParser	10	9
data	22	DecodedBitStreamParser	10	5
character	65	DecodedBitStreamParser	10	9
matrix	10	DecodedBitStreamParser	6	3
datamatrix	8	upperShift	3	3

2.2 Linking with Different Terms in Both Sides

Figure 4 shows another bug record (#631) in the ZXing project and its corresponding commit (#1670). The text in the bug record is not similar to that in the commit log or changed source code. Moreover, unlike in the previous example, the entities and components' names in the changed source files do not appear in the summary, description, or comments of the bug record. In this case, the matching via common patterns, texts, names of program entities and components, or recommended patch code does not work well.

We further performed a simple text analysis on the entire collection of bug reports and commits for the terms appearing in the bug record #631 and in its corresponding fixing commit #1670 such as decoding, data, DecodedBitStreamParser, upperShift (see Table 1). Each row represents a pair of terms in the bug report and a commit, respectively. For example, we found in the history, there are 46 bug reports containing the word decoding and 10 fixes requiring the modifications containing DecodedBitStreamParser. Among those 10 fixes, nine of them have the corresponding bug reports containing the term decoding. Similar explanation is for other terms in Table 1. Thus, if the fixing commits contain the terms DecodedBitStreamParser or upperShift (i.e. the fixes involve those classes/methods), then the corresponding bug records likely have the terms on the left (e.g. decoding, datamatrix). Therefore, if we can learn such association between two terms in two sides from the established/detected bug-to-fix links in the past history, we could infer the links in which the commit involving with some entity names is likely to be the fix of a bug report(s) having their associated terms or vice versa.

2.3 Linking via Notes in Bug Comments

We also found that four ZXing's bug records in which bug commenters recorded the corresponding fixing revisions. For example, in the comment of bug report #472, commenter 3 left a note as follows: *"Fixed by r1480"*. We were able to confirm this link at that revision. This type of information is useful for link recovery as in the pattern-based approach [35]. The difference is that such method explores the notes left by fixers in the commit logs. No existing approach makes use of the notes on the fixing revisions recorded in the bug reports.

3. MLINK APPROACH

This paper introduces MLink, a multi-layered approach to automatically recover bug-to-fix links. Given the history of bug records in a bug-tracking repository and that of commits (commit logs and changed source files) in a version repository, it will recover the links between the already-fixed bug reports and the corresponding fixing commits (i.e. fixes).

MLink extracts and makes use of not only *textual features* in bug records (summary, description, and bug comments) and in commit logs, and meta-data as in existing bug-to-fix link recovery approaches [34, 36], but also *code features* in the associated information of the bug records and commits. MLink recovers links in cascading layers in which each layer is a detector with its own set of textual and code features (Figure 5). The input of each layer is the remaining candidate links that the previous layers could not confirm/detect. Its remaining candidate links are passed into the next layer, with the expectation that some additional links will be revealed via features used in the next layer. The detected links are combined into the final link set from all layers. The detectors/layers having features with higher levels of confidence on accurate detection are applied at earlier stages.

MLink Architectural Overview. Figure 5 displays the process in MLink to recover bug-to-fix links. Bug records in the bug-tracking database and the commits with their associated logs and changed code are first analyzed by the feature extractor. The features will be fed into the appropriate detectors at different layers, e.g., the time features are used in the filtering layer, while recommended patch code features, names of program entities and system components, text features, and term features are provided into the *patch-based*, *name-based*, *text-based*, and term/code *association-based* link detectors, respectively. Let us explain them.

1. Pattern-based detector. This module is adapted and extended from the pattern-based approach for bug-to-fix link recovery [36]. It works based on the following:

a) the notes/hints that the fixers provided in the *commit logs* about the issues/bugs for which their fixing changes were intended. The typical patterns/phrases include 'fix the issue #...', 'fix the bug ID...', etc.

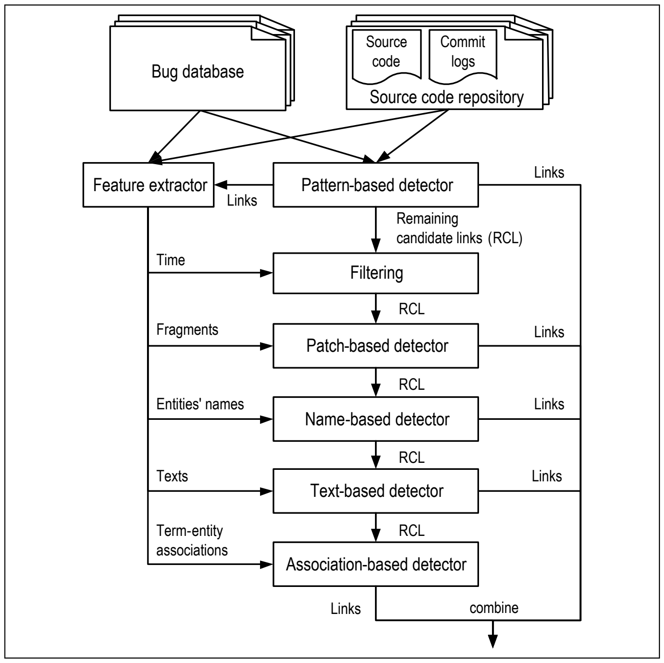

Figure 5: MLink Approach

b) the notes the fixers/commenters left in a *bug record* to refer to the fixing revision(s) for that bug report. Common patterns/phrases are 'fixed by r123', 'fixed in r123', etc.

The key difference from the traditional approach [36] is that in addition to commit logs, MLink also mines the common patterns in the *comments of the changed code* as well as in the comments of bug records. Moreover, although the notes in the above a) and b) are similar in spirit, no existing approach makes use of the notes in b). Importantly, because the notes from developers are correct indication of true bug-to-fix links, we use the resulting links from this detector as the training sets to learn the thresholds, parameters, and criteria for the measurements in the next detectors. This is a form of un-supervised learning [34] (will be described later).

2. Filtering layer. The remaining candidate links that were not detected by the pattern-based detector are analyzed and those violating a time constraint will be removed. The constraint is that the committing time for a fix must be between the open and close time of the corresponding resolved bug record. This step reduces the number of unnecessary candidate pairs of bug records and irrelevant fixes.

3. Patch-based detector. This layer first extracts the patch code recommended by the bug reporters or commenters (if any) that is embedded within bug descriptions/-comments. It then matches them against the *changed source code* (i.e. changed code portions) of the commit under testing. We develop a method to isolate code fragments within the texts of a bug report and use a token-based clone detection method for code matching. Details are in Section 4.

4. Name-based detector. In some cases, the recommended patches are not available. However, the names of entities and system components would be mentioned in both bug records (descriptions, summaries, and comments) and fixes (commit logs, changed code, and inline comments). We also develop an algorithm to detect such names in both sides and match them against each other. Details are in Section 5.

5. Text-based detector. After detecting links via the layers with code-based features, MLink compares bug records with fixes via textual features extracted from bug descriptions/summaries/comments and from commits (with commit logs and inline comments in the changed code). MLink also uses the terms in commit logs as in ReLink [34]. However, ReLink did not use inline comments within code.

6. Association-based detector. This detector is used where the texts or entity names in bug records and commits are not similar. MLink computes the association strengths between the terms in bug reports and the entity names in commits from the link set detected by the pattern-based detector, and then infers the links. That is, the commit with some entity names is likely the fix for a bug record(s) having their associated terms or vice versa (see Section 6).

4. PATCH-BASED LINK DETECTION

This section describes our technique to extract patch code embedded within the texts of a bug record, and then to match it against the changed code for link recovery.

4.1 Patch Extraction

Embedded patch code in the texts of a bug record has the following characteristics. First, it is a sequence of contiguous code tokens surrounded by the descriptive texts written in a natural language. Second, those code tokens include conventional program tokens in programming languages, e.g. keywords, operators, delimiters, identifiers, literals, etc. Third, among those tokens, there are code tokens similar to the tokens in a natural language such as *textual comments*, *string literals*, *identifiers*, and *keywords*. Those four types of code tokens pose a challenge in patch extraction since it is not trivial to distinguish them from (natural-language) regular texts. Finally, a patch must have a sufficiently large size.

Based on those characteristics, MLink extracts patches from a bug report B (Figure 6). It processes B in three passes. In the first pass, it tokenizes all the contents of a bug report and identifies the clearly recognizable program tokens (e.g. operators, delimiters, etc) from regular texts. Among the aforementioned four types of code tokens, *textual comments* within a patch and *string literals* can be recognized in this pass because those comments must begin with special characters such as "/*" and "//", and strings must appear within quotes. Thus, MLink replaces them with special code tokens comments and lits, respectively. Moreover, if identifiers follow a naming convention (e.g. CamelCase), they can be matched in this pass via lexical patterns according to the programming language (see Section 5.1. for this matching).

MLink's second pass over all tokens is designed to handle the cases of *keywords* and *identifiers* that are not clearly distinguishable from regular texts of a bug record. The idea is that if such a *not-yet-determined code token* (keyword or identifier) is surrounded by several clearly recognizable code tokens, (i.e. the density of code tokens around it is high), then it is also marked as a code token (i.e. the code area is expanded). That condition is measured by C/N, where C and N are the numbers of code tokens and the surrounding tokens, respectively. The expansion process is repeated for next not-yet-code tokens until all of them are consumed. Finally, in the last pass over all tokens, MLink detects all the patched code by collecting all sequences of contiguous code tokens whose sizes are sufficient large (i.e. $\geq$ a threshold).

```
function PatchExtractor (B: bug record)
  1. Tokenizing
       Recognize code tokens
       Replace comments/literals by special code tokens
  2. Expanding the areas of code tokens
       Repeat
         Find next not−yet−determined code token T
         Count C: number of code tokens in a
                  window of size N centered at T
         If C/N> 0.5, mark T as code token //Not used for
              finding other code tokens
       Until no more not−yet−determined code token
  3. Finding patches
       Foreach sequence of contiguous code tokens P_i:
         If Len(P_i)>=Threshold, add P_i to the patch list
```

Figure 6: Patch Extraction Algorithm

4.2 Recovering Links via Patch Matching

A patch, after extracted from a bug record, is used to trace the corresponding fixing code. MLink compares the patch against the changed code fragments in each candidate commit, and ranks them based on their similarity levels. For comparison, it computes the longest common subsequence between two sequences of code tokens derived from a patch in a bug record and from changed code. Code tokens can be easily extracted from the changed code after the inline comments and literals are replaced with comment and lit tokens.

Assume that MLink extracts m patches $P_1, P_2, .., P_m$ from a bug record B. The candidate commit C has n changed code fragments $S_1, S_2, .., S_n$. Let us also use P_i and S_j to denote the sequences of tokens derived from them, respectively. The matching level $m(P_i, S_j)$ between the patch P_i from B and code fragment S_j in C is computed as follows:

$$m(P_i, S_j) = \frac{|lcs(P_i, S_j)|}{1 + \min(|P_i|, |S_j|)}$$

where $|lcs(P_i, S_j)|$ is the length of the longest common subsequence of P_i and S_j. $|P_i|$ and $|S_j|$ are the numbers of tokens in P_i and S_j, respectively. We use the min function (instead of $|P_i|+|S_j|-|P_i \cap S_j|$) to avoid the case where either P_i or S_j is much larger than the other, and $m(P_i, S_j)$ would be very small. This can occur since the recommended patch is often smaller than the actual fixed code. We also add 1 into the denominator to regularize the value of $m(P_i, S_j)$ in the case where $\min(|P_i|, |S_j|)$ and $lcs(P_i, S_j)$ are small (e.g. equal 1). The matching degree between B and C is as follows:

$$m(B, C) = \max_{P_i \in B, S_j \in C}(m(P_i, S_j))$$

We use the max function (instead of avg) because the commenter might suggest multiple patches and the fixer chooses only one. Thus, MLink aims to find the best matching pair.

The link between the bug record B and a commit C is established if their matching degree is over a threshold θ_p.

5. NAME-BASED LINK DETECTION

Let us describe our technique to trace links via the names of entities extracted from bug records and changed code.

5.1 Entity Name Extraction

During programming, developers often follow some naming convention/style. For example, the identifiers are written in CamelCase style, C notations; the fully qualifier names

Table 2: Entity Name Detection Patterns

Type	Pattern	Convention	Example
CamelCase	[A-Za-z]+.*[A-Z]+.*	Name in Java	getHeight
C_notation	[A-Za-z]+[0-9]*_.*	Name in C	dns_look_up
Qualified.Name	[A-Za-z]+[0-9]*[\.].+	Name in Java	obj.toString
UPPERCASE	[A-Z0-9]+	Literal	DEFAULT

must follow the lexical syntax with separators '.'; constants are often written in uppercase letters; etc. Since MLink focuses on clearly recognizable entities' names, it detects them via lexical patterns as described in Table 2.

For extracting entity names in a bug record B, MLink first tokenizes B's contents. It removes all delimiters and operators, and then uses the patterns in Table 2 to determine if an extracted token is an entity name. To avoid insignificant entity names, it filters the names that do not appear in all changed code and the ones that occur in most of bug records.

5.2 Recovering Links via Entity Names

When the name of an entity e is mentioned in a bug record B, that entity has potential relevance to the reported technical issue(s) in that report. If e belongs to the *changed source code* or the *log* of a commit C, it is related to the technical function(s) that was changed in that commit. Therefore, the link between B and C can be decided via the correlation between 1) e is relevant to the reported issue in B, and 2) e, mentioned in C, is involved in fixing the *same* function/issue in B. The correlation via a common entity e is computed as

$$\eta_e(B,C) = \frac{N_e(B)/N(B) + N_e(C)/N(C)}{|\{B_i : e \in B_i\}| + |\{C_j : e \in C_j\}|}$$

$N_e(B)$ and $N_e(C)$ are the numbers of occurrences of e in B and C, respectively. $N(B)$ and $N(C)$ are the total numbers of occurrences of all entities' names in B and C. $|\{B_i : e \in B_i\}|$ and $|\{C_j : e \in C_j\}|$ are the numbers of bug records and commits having e, respectively.

The correlation $\eta_e(B,C)$ is higher if entity e is significant with respect to B and C. That is, e occurs more frequently in both bug record B and the changed code or commit log of C (see the numerator), however, it appears much less frequently in other reports and commits (see the denominator).

Because there might exist multiple such entities e, the link between B and C is decided via the aforementioned correlation over all extracted common entities as follows:

$$\eta(B,C) = \sum_{e \in (B \cap C)} \eta_e(B,C)$$

We use the summation in the formula to emphasize on all significant entities that appear in both sides.

When matching B against all candidate commits C, MLink will select C if the correlation $\eta_e(B,C)$ is over a threshold θ_n.

6. TEXT-BASED LINK DETECTION

In a system, one or more technical functions were erroneously implemented in some source file(s). The terms describing that issue(s) in a bug report could be similar to the terms in the comments of those fixed files since a developer wants to record the reason why (s)he made that fixing change. MLink extracts terms from those sources and pre-processes them with tokenizing, stop-word removal, and stemming. Identifier names in the changed files are also separated in accordance with common coding styles, e.g. getHeight is split into get and height. Next, MLink computes the significance of a term w_i in a document d via Term Frequency - Inverse Document Frequency (Tf-Idf) [30]: $s_i = tf_i \times idf_i$, where tf_i is the occurrence frequency of w_i in document d, and idf_i, the inverse document frequency, is equal to $log\frac{|D|}{|\{d : w_i \in d\}|}$. $|D|$ is the total number of documents. $|\{d : w_i \in d\}|$ is the number of documents containing w_i. Then, MLink calculates the cosine similarity between a bug record and all remaining candidate fixes via their vectors of the Tf-Idf values of all the terms. It selects the top-ranked candidate as the detected corresponding fix of the bug record if the respective similarity is larger than a threshold θ_t. This threshold can be learned from the set of links returned from the pattern-based detector (will be detailed in Section 8).

7. ASSOCIATION-BASED DETECTION

This section describes our technique to recovery links where the texts, patch code, and the names of entities or components in both bug records and commits are *not* quite similar.

7.1 Term and Program Entity Association

In a program, program entities/components are used to realize some system functionality. If the changed source code of a commit involves with some program entities/components, that commit is considered as a change made to their corresponding functionality. In a bug record, the terms describe the reported technical issue(s) in the system. Thus, in MLink, the association level between an *entity/component e* in the changed code of a commit and a technical *term w* in a bug record indicates/represents *how likely that functionality realized via entity e has the issue(s) described by w.*

The association $\mu_e(w)$ between a program entity e and a term w in bug records is calculated via the correlation between the number of bug records containing w (i.e. using w in describing some technical issues) and the number of corresponding commits involving e in fixing those issues:

$$\mu_e(w) = \frac{n_{w,e}}{1 + \min(n_w, n_e)} \qquad (1)$$

- $n_{w,e}$ is the number of commits whose changed source code contains e for fixing the bug records containing w,
- n_w is the number of all bug records containing w, and
- n_e is the number of commits whose changed code contains e.

If w appears many times in a bug report, MLink counts only once for n_w. If e appears many times in changed code, it is also counted once. In the above formula, we use *min* function (instead of $n_w + n_e - n_{w,e}$) to avoid the cases in which either the set S_w of bug records having w or the set S_e of commits having e is much larger than the other, and $\mu_e(w)$ would be very small. We add 1 in the denominator to regularize the value of $\mu_e(w)$ as in Section 4.2.

Let us use N to denote $min(n_w, n_e)$. If every time w is used in a bug record to describe an issue, the corresponding fix always contains the entity e, then $\mu_e(w)$ gets the highest value $N/(1+N)$. This value approaches 1 if N is sufficiently large. If entity e is always involved in a fix for a technical issue described via term w, then $\mu_e(w)$ also gets the highest value $N/(1+N)$. If e has never been used to fix a technical issue described by w, $\mu_e(w)$ is zero. In general, it is within $[0,\ N/(N+1)]$. The more frequently w and e co-appear in its respective document, the higher the value of $\mu_e(w)$ is.

7.2 Bug Record and Entity Association

Because a bug record contains multiple terms, MLink computes the association between a program entity in a commit and that bug record via its terms. The association between the entity e and the bug record B is calculated as the maximum association value between e and all the terms w in B:

$$\mu_e(B) = \max_{w \in B}(\mu_e(w)) \qquad (2)$$

where $\mu_e(w)$ is the association value between w and e.

The higher $\mu_e(B)$ is, the more likely the functionality realized via entity e has the issue(s) reported in B. Since a bug record might contain several terms not much related to reported technical issue(s), MLink uses the max function (instead of the sum function) in order to select the most significant term(s) with the highest correlation with the entity e.

7.3 Bug Record and Commit Association

In a bug record B, the technical issue(s) is expressed via terms, while the changed code in a commit C is written in a programming language via program entities. The association between B and C indicates the correlation between 1) whether B reports about some technical issue(s), and 2) whether C is aimed to change the same reported issue/function(s). Thus, that association value represents how likely a bug report B was fixed by a particular commit C.

Assume that the commit C has the program entities e's which are involved in the fixed code of C. The association between B and C is computed as the maximum value among all association values between B and all involved entities in the changed code of the commit C:

$$\mu_C(B) = \max_{e \in C}(\mu_e(B)) \qquad (3)$$

where $\mu_e(B)$ is the association between an entity e in C and bug record B. MLink uses the max function (instead of the sum function) in order to select the most significant entities with the highest correlation with the bug report B.

Link Recovery with Association. MLink's recovery process has both training and detecting phases:

1. In the training phase, the training set consists of the detected bug-to-fix links returned from the pattern-based detection layer. MLink first pre-processes the bug records and the changed code fragments in the corresponding commits including tokenizing, stop-word removal, stemming, parsing the code, collecting tokens/entities, etc. In the bug records, only the significant terms with high Tf-Idf values are kept. From the changed code, entities are extracted as described in Section 5.1. Next, MLink calculates the values of n_w, n_e and $n_{w,e}$ for each word w in all bug records and each entity e in all commits in the training set. It then computes the association values $\mu_e(w)$ according to the formula (1).

2. In the detection phase, for a given bug record B and each candidate commit C, MLink calculates the association values $\mu_e(B)$ and $\mu_C(B)$ with the formulas (2)-(3). The link between B and C is decided if the association value is the highest among all candidates and is over a threshold θ_a.

8. EMPIRICAL EVALUATION

8.1 Data Collection and Evaluation Metrics

Let us present our empirical studies to evaluate and compare MLink with ReLink [34] and BugScout [27] approaches.

Table 3: Subject Systems

Project	Period	Revisions	FBugs	Links
ZXing [37] (ZX)	11/07-12/10	1-1,694	135	143
OpenIntents [29] (OI)	12/07-12/10	1-2,890	101	129
Apache [3] (AP)	11/04-4/08	76,294-899,841	686	1090

For comparison, we used the same data set as in ReLink [34] in which the ground truth for bug-to-fix links of three projects are publicly provided at [40]. Table 3 summaries their information with time and revision ranges, the number of fixed bugs, and the number of links. We also downloaded the respective bug records (summaries, descriptions, comments, and meta-data) as well as the commit logs, changed source code, and meta-data from the projects' Web sites.

The recovered links from MLink were compared to the ground truth. As in Relink [34], we also use three evaluation metrics. *Precision* is defined as the ratio between the total number of correctly detected links and the total number of detected links. *Recall* is the ratio between the number of correctly detected links and the total number of links in the ground truth. *F-score* is balanced metrics for precision and recall and is defined as F-score = $\frac{2 \times Precision \times Recall}{Precision + Recall}$.

8.2 Module-based Sensitivity Analysis

8.2.1 Sensitivity Analysis

In this experiment, we analyzed the sensitivity of the recovery accuracy of each module/layer in MLink with respect to its threshold. The base module does not have threshold, thus, we will present its accuracy in the next section. Let us call the four other modules patch, name, text, and assoc. We ran those four modules on ZXing and measured its accuracy for various threshold values. For all experiments, thresholds are within [0..1] with the varying step of 0.01.

Figure 7 shows the sensitivity analysis result for the patch module. As seen, as the threshold θ_p is smaller (< 0.1), i.e. the matching criteria for patched code is not strict, recall is higher since more links are detected. However, precision is lower due to more false positive cases. When θ_p is increased, precision increases and recall decreases. As seen, the patch module can get stable at highest precision (100%) when $\theta_p >= 0.6$ as it detected 9 correct links (out of 143). However, patch must not be used individually due to its low recall and F-score. It must be used in a combination at an *earlier stage* to take advantage of its *high precision* with high threshold value (i.e. strict matching criteria of patched code).

Figure 8 shows the sensitivity analysis result for name. As θ_n is increased (i.e. the requirement of a report and a commit sharing entity names is stricter: $\eta > \theta_n$), precision increases and recall decreases. However, F-score also decreases, indicating that the name module should not be used individually for link recovery. However, precision quickly achieves highest at 100% as θ_n is increased in the range [0.09-0.3]. At θ_n=0.09 and θ_n=0.3, it detected 9 and 1 true links, respectively. As θ_n>0.3, the stricter requirement of many shared entity names does not allow it to detect any more link. Thus, the name layer must be used in *an earlier stage* to achieve high precision within that range of θ_n. Note: small η values are due to the normalization formula in Section 5.2.

Figure 9 shows the sensitivity analysis result for text. As seen, there is a range of θ_t between [0-0.1] in which precision,

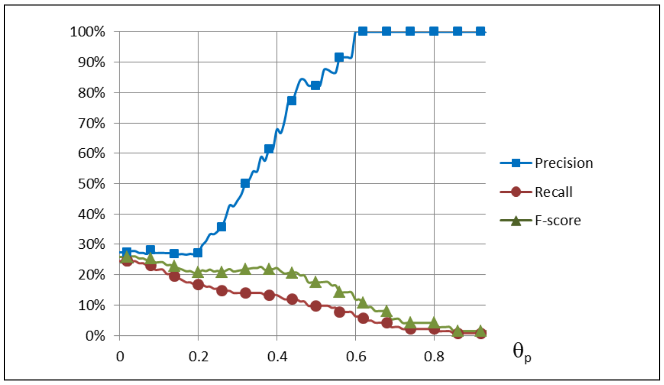

Figure 7: Accuracy Analysis of patch on θ_p

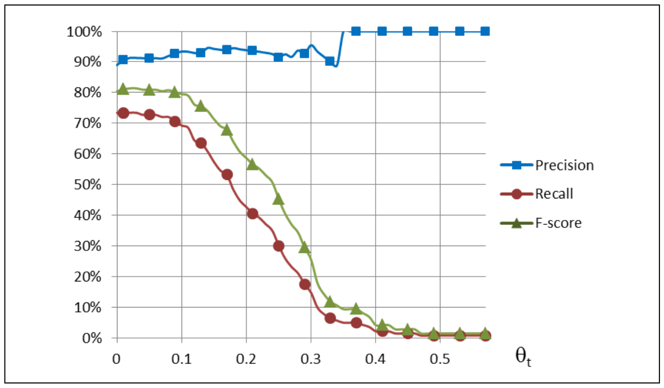

Figure 9: Accuracy of text on θ_t

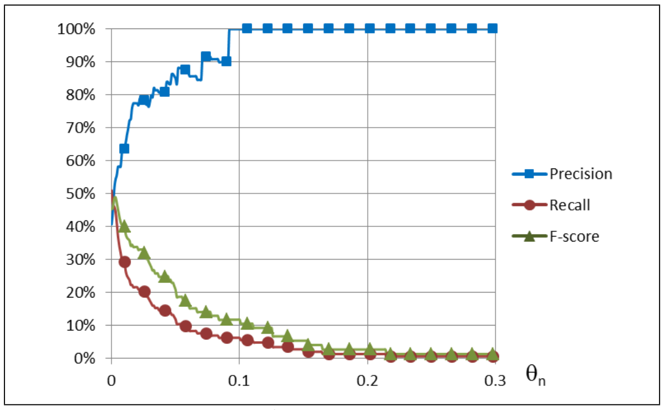

Figure 8: Accuracy of name on θ_n

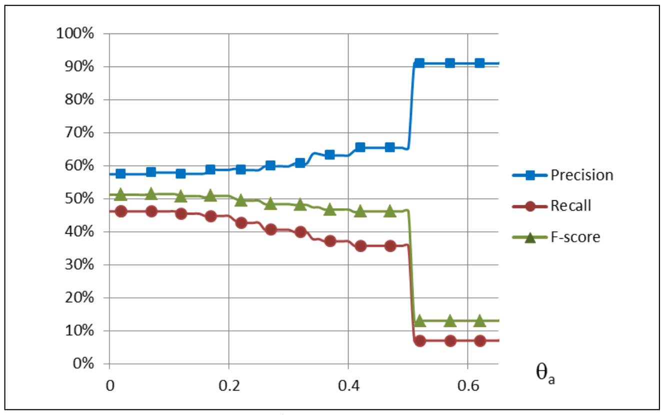

Figure 10: Accuracy of assoc on θ_a

recall, and F-score values are stable, and F-score reaches its highest (81-82%). The peak F-score value is higher than those in patch and name. As θ_t is increased over 0.1 (i.e. text similarity between a report and a commit must be higher), precision increases a bit, however, recall and F-score decrease much. Precision reaches its highest value of 100% at θ_t=0.35 when it detected 7 correct links out of 143 true ones. As θ_t is over 0.4, it cannot detect any more link. This result suggests that link recovery should set a low threshold θ_t for textual similarity between a report and a commit because higher θ_t would lead more missing links. Thus, aiming for high precision (100%) for patch and name, one should *connect them in earlier stages than the text module* because text with low θ_t has lower precision than them. Note that the text module differs from ReLink in which it considers also inline comments in changed code. Moreover, it takes only the top-ranked candidate, rather than multiple ones as in ReLink. This enables MLink to pick up high-quality links and to rely on the assoc module to detect more links later.

Figure 10 shows the sensitivity analysis result of assoc. There is a range when θ_a=[0-0.2] in which accuracy is stable. As θ_a is increased (i.e. stricter criteria on high associations of terms in a bug report and entities in the fixes), precision increases gradually, while recall and F-score decrease slowly. Precision reaches its highest of 91% as θ_a is in [0.5-0.63]. However, both recall and F-score decrease much. The result shows that one should not use high threshold for the assoc layer. Moreover, this module has lower F-score than text module, thus, it should be connected *after* text. In summary, this confirms the order of layers: patch, name, text, and assoc.

8.2.2 *Threshold Learning*

Because all modules in MLink are cascading in layers as shown in Section 3, the thresholds explained in the previous experiment might not be exactly the optimal ones when the modules are used together. This section explains our experiment to automatically learn those thresholds. Note that the pattern-based detector, base, does not need any threshold and always returns highly accurate links [34]. Thus, we design a unsupervised hill-climbing algorithm to learn the values for the thresholds θ_p, θ_n, θ_t, and θ_a for other detectors from the resulting links of the base detector.

The algorithm aims to find the thresholds such that the highest F-score is achieved for the set of links recovered by the base module. Let us use $f_t = f(\theta_p, \theta_n, \theta_t, \theta_a)$ to denote the F-score goal function. The idea of our algorithm is that to estimate the local optimum value of a function with multiple variables $t_1, t_2, ..., t_n$, it first sets the initial values for all variables. At each iteration, it fixes n-1 values and varies the remaining variables to find the optimum values. It varies the value for each variable until it cannot get better F-score.

Figure 11 shows MLink's hill-climbing algorithm. At first, the set of all links P detected by the pattern-based layer is divided into the training and testing sets for 3-fold cross validation, in which two folds are used for training and one for testing. The training folds are used to determine the association values between bug records' words and program entities. The algorithm then repeatedly fixes three of four thresholds and varies the remaining threshold in its range. It uses MLink to detect links on the testing fold and F-score is computed. When the highest f_t at a value x of the threshold is

```
function HillClimbingOptimization ($P$: Set of links detected by
        the pattern−based detector)
  Initializing $\theta_p = 0$, $\theta_n = 0$, $\theta_t = 0$, $\theta_a = 0$
  1. Dividing $P$ into
     Training set $P_{train}$ with $size = \frac{2}{3} size$ of $P$
     Testing set $P_{test}$: the remaining subset of $P$
  2. Training:
     Computing the association values $\mu_e(w)$s on $P_{train}$
     Repeat
       foreach $\theta_i$ in $\{\theta_p,\theta_n,\theta_t,\theta_a\}$
         Varying $\theta_i$ in its range
         Detecting links in $P_{test}$
         if $f_t$ reaches the highest value at $x$ then
           set $\theta_i$ to $x$ and move to next $\theta$
     Until $f_t$ stop increasing
```

Figure 11: Hill-climbing Algorithm for Threshold Learning

reached, it sets the value of the threshold to x and continues to vary other thresholds until f_t cannot get better values.

Our algorithm returned the following values for the thresholds: θ_p=0.49, θ_n=0.10, θ_t=0.08, and θ_a=0.42. We used these values for the remaining experiments.

8.2.3 Accuracy of Combinations of Modules

In this experiment, we measured the accuracy of different combinations of the modules. Table 4 shows the result.

Our base module is similar to the traditional pattern-based approach (T) [35, 36] except that we also consider the fixers' notes *within bug records* (Section 2.3). In comparison, MLink, via mining such notes, was able to detect additional 4, 13, and 118 links in ZXing, OpenIntents, and Apache, respectively. As seen, base achieved high precision (83-100%), however, it missed many true links, leading to low F-score.

As connecting to base, the patch module helps improve both recall and F-score while maintaining high precision. In ZXing, base+patch detected 5 additional correct links without any false detection, and improves recall by 4% and F-score by 3%. In Apache, it detected 11 additional true links with 5 false ones, and improved both recall and F-score. No patched code in bug comments was detected in OpenIntents, thus, accuracy stays the same as that of the base module.

When connecting to base+patch, the name module with its matching criteria on entities' names has much improved over base+patch with many additional correctly detected links and a very few incorrect ones. It detected 14 and 12 more true links with 1 and 5 incorrect ones in ZXing and Apache, respectively. In OpenIntents, it got 3 more correct links without an incorrect one. Generally, high precision of base+patch is maintained in base+patch+name, while recall is improved from 2-9%, and F-score from 1-7%. This result is consistent with that of the previous experiment because patch and name modules should contribute to MLink when they give high precision with strict matching criteria.

Compared with base+patch+name, base+patch+name+text detected many more additional true links: 33 and 41 more links in ZXing and Apache, respectively. The numbers of new incorrect links are small (4 and 14 in ZXing and Apache). In OpenIntents, it detected 2 more correct ones with no incorrect link. For ZXing, the text module helps improve significantly recall (+23%) and F-score (+14%), while precision decreases only 3%. For other two systems, recall increases 1-3%, while precision stays the same as base+patch+name.

Table 4: Accuracy Result for Cascading Layers

Sys.		T	base	base+ patch	base+ patch+ name	base+ patch+ name+ text	base+ patch+ name+ text+ assoc (MLink)	Re-Link	base+ Bug-Scout
ZX	Det	69	73	78	93	130	131	118	130
	Corr	69	73	78	92	125	126	107	102
	Incor	0	0	0	1	5	5	11	28
	Miss	74	70	65	51	18	17	36	41
	Prec	100	100	100	99	96	96	91	79
	Rec	48	51	55	64	87	88	75	71
	Fs	65	68	71	78	92	93	82	75
OI	Det	87	100	100	103	105	113	95	112
	Corr	87	100	100	103	105	110	95	105
	Incor	0	0	0	0	0	3	0	7
	Miss	42	29	29	26	24	19	34	24
	Prec	100	100	100	100	100	97	100	94
	Rec	67	78	78	80	81	85	74	81
	Fs	81	87	87	89	90	91	85	87
AP	Det	1,060	1,096	1,112	1,129	1,184	1,193	1,261	1,142
	Corr	791	909	920	932	973	978	934	921
	Incor	269	187	192	197	211	215	327	221
	Miss	299	181	170	158	117	112	156	169
	Prec	75	83	83	83	82	82	74	81
	Rec	73	83	84	86	89	90	86	84
	Fs	74	83	84	85	86	87	79	83

We also connected all modules in MLink. As seen, for the cases that textual similarity is not high, the assoc module was able to detect additional true links (1, 5, and 5 for ZXing, OpenIntents, and Apache, respectively) that were not detected by any previous modules. It improved recall from 1-4% over base+patch+name+text with equally high precision.

In this study, we observed that when a new module was added to the previous modules, it was always able to detect additional true links that were not found by previous modules. The modules base, patch, and name contributed additional correct links, increase recall, while maintaining high precision. The modules text and assoc detected more additional true links than the others, with significantly higher recall and slightly lower precision. Thus, F-score increases much.

8.3 Comparisons with ReLink & Topic Model

Table 4 shows the results of two state-of-the-art methods, ReLink [34] and BugScout [27]. We used the result for ReLink as reported in its paper [34] because we used the same data sets and oracle. BugScout is a technique to recover links between bug reports and source files using topic modeling. We used BugScout to derive the links between bug reports and commits via the links to changed files. We trained it via the link set recovered by the base module.

As shown, MLink achieves a very high accuracy level: 87-93% in F-score, 85-90% in recall, and 82-97% in precision. MLink is able to improve over ReLink by 6-11% in F-score, 4-13% in recall, and 5-8% in precision. It detected 19, 15, and 44 more true links than ReLink in ZXing, OpenIntents, and Apache, respectively. The numbers of incorrect links are reduced by 6-112 links as compared to ReLink (except 3 more incorrect ones in OpenIntents). The numbers of missing links are smaller from 15-44 links in comparison with ReLink. Moreover, MLink also outperformed BugScout by 4-18% in F-score, 4-17% in recall, and 1-17% in precision. It detected 24, 5, and 57 more true links than BugScout in ZXing, OpenIntents, and Apache, respectively.

Table 5: Comparison of Measurement Data

	ZX			OI			AP		
	Gold	Re-Link	M-Link	Gold	Re-Link	M-Link	Gold	Re-Link	M-Link
Changes	1,694			2,890			43,167		
Fixes	138	107	120	121	94	103	976	866	921
%Fixes	8.1%	6.3%	7.1%	4.2%	3.3%	3.6%	2.3%	2.0%	2.1%
Files	399			742			194		
Buggy	118	83	102	36	30	30	98	101	100
%Buggy	29.6%	20.8%	25.6%	4.9%	4.0%	4.0%	50.5%	52.1%	51.5%

Table 6: Time Efficiency

Sys.	BRs	Commits	Train(s)	Detect(s)	Detect/BR(s)
ZX	135	1,694	205.7	187.3	1.4
OI	101	2,890	203.2	177.0	1.8
AP	686	43,867	3,250.3	3,172.7	4.6

We also conducted another experiment to evaluate how other maintenance metrics would be more precise when they are computed from the links recovered by MLink in comparison with the links detected by ReLink. Two metrics used in this experiment include

1) The percentage of bug fixes among all other types of changes in a project. This value reflects developers' bug-fixing efforts in the total efforts of code change activities;

2) The percentage of buggy files in a project.

Those two metrics can be easily derived from the recovered link set. Table 5 shows the result. Column Gold displays the ground truth values, which are also available with the data sets provided by ReLink's authors. As shown, the estimated values using the result from MLink are closer to the ground truth than those from ReLink. For example, the percentage of bug-fixing changes in ZXing detected by MLink is 7.1%, i.e. only 1.0% different from the golden value, while the difference between ReLink's result and the golden one is 1.8%. Similarly, the percentage of buggy files in ZXing detected by MLink is 25.6%, i.e. 4% different from the golden value, while the difference between ReLink's and the golden one is 8.8%. Thus, MLink gives better measurement values.

Time Efficiency. Table 6 shows MLink's time efficiency result. As seen, the training time including time to train thresholds and association values is reasonable ranging from 203s to less than 1 hour (for the large number of Apache's bug reports). Detection time per bug report is very small from 1.4-4.6s. Thus, MLink is very time efficient.

Threats to Validity. The oracle provided by ReLink's authors might contain errors due to human checking. However, our goal was to compare MLink with other approaches. Thus, they affected all approaches. Subject systems are open-source with varied quality of bug records and commits.

9. RELATED WORK

Several approaches have been proposed to recover bug-to-fix links. In Fischer *et al.* [14, 15]'s approach, if file names exist in both bug reports and change logs, the links will be recovered. Bachmann *et al.* [8] enhanced that method by checking the bug-closing date against the commit date. Zimmermann *et al.* [31, 35, 36] use the traditional heuristics such as common phrases in commit logs. These traditional heuristics have been widely used in MSR research, e.g. bug prediction models [19, 32, 35, 36], bug-introducing/fixing classification/identification [16, 17, 18, 24], and fixing efforts [23]. Śliwerski *et al.* [32] check semantic relationships, e.g. whether a change log contains terms in a bug report, etc. None of them use source code and comments as in MLink.

Bachmann *et al.* [9] found strong evidences that only a fraction of bug fixes are actually labeled in source code repositories. Thus, the recovered links by traditional heuristics can be a biased sample of the entire population of fixed bugs [9, 10]. LINKSTER [11] is a tool to facilitate the manual identification of such links. Nguyen *et al.* [28] reported a similar bias in a bug-fix dataset in IBM Jazz. Other researchers also observed the negative impacts of low data quality on the results of empirical studies [1, 4, 21, 24, 26, 33].

To improve link recovery accuracy over the traditional approach, Wu *et al.* [34] developed ReLink, an algorithm based on three key features: 1) time interval, 2) text similarity, and 3) mapping of bug commenters and change committers (see Section 1). ReLink first uses the traditional heuristics to recover the initial set of links. That set is considered as the correct set from which ReLink then learns the thresholds for time interval, text similarity, and the mapping between bug commenters and change committers. In comparison, first, MLink uses both texts and code features in source code and changes, and algorithms to extract/compare them. Entities' names and code fragments in both sides are extracted/compared. Second, our association algorithm is able to associate terms and entities based on the established links from the base module for the cases of dissimilar texts. Moreover, we also use version IDs recorded in bug reports' comments.

A related line of work is traceability link recovery (TLR). Bacchelli *et al.* [6, 7] use entities' names to recover the links between emails and source code. Corley *et al.* [12] use patch code for bug-to-fix link recovery. However, both methods do not support where bug reports do not contain the same entities' names as in source code. This is common since bug reporters describe the bugs with more general terms, while developers use more compact names for program entities. MLink can handle those cases via its association layer.

Other popular TLR approaches are based on information retrieval (IR) including vector space model and probabilistic models [2], Latent Semantics Indexing (LSI) [22], topic modeling [5, 27], etc. In comparison, those IR methods are based on textual similarity between bug reports and source code (and its comments). They do not handle well the cases where the texts in reports and commits are quite different. Cleland-Huang *et al.* [13] used machine learning to recover the links from regulatory code to product requirements.

10. CONCLUSIONS

This paper introduces MLink, a bug-to-fix link recovery method that considers both textual and source code features in artifacts. MLink is also capable of learning the *association relations* between the *terms* in the bug records and the *names* of entities in changed code to recover links for reports/commits without much similar texts. Our empirical evaluation on real-world projects shows that MLink can improve the state-of-the-art methods by up to 18% in accuracy.

11. ACKNOWLEDGMENTS

This project is funded in part by US National Science Foundation (NSF) CCF-1018600 grant.

12. REFERENCES

[1] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *Proceedings of the conference of the center for advanced studies research*, CASCON'08. ACM, 2008.

[2] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28:970–983, October 2002.

[3] Apache. http://httpd.apache.org/.

[4] J. Aranda and G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In ICSE '09, pp. 298–308. IEEE CS, 2009.

[5] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor. Software traceability with topic modeling. In ICSE'10, pages 95–104. ACM, 2010.

[6] A. Bacchelli, M. D'Ambros, M. Lanza, and R. Robbes. Benchmarking lightweight techniques to link e-mails and source code. In *Working Conference on Reverse Engineering*, WCRE'09, pp. 205–214. IEEE CS, 2009.

[7] A. Bacchelli, M. Lanza, and R. Robbes. Linking e-mails and source code artifacts. In ICSE'10, pages 375–384. ACM, 2010.

[8] A. Bachmann and A. Bernstein. Software process data quality and characteristics: a historical view on open and closed source projects. In IWPSE-Evol'09, pages 119–128. ACM, 2009.

[9] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein. The missing links: bugs and bug-fix commits. In FSE'10, pages 97–106. ACM, 2010.

[10] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: bias in bug-fix datasets. In ESEC/FSE '09, pages 121–130. ACM, 2009.

[11] C. Bird, A. Bachmann, F. Rahman, and A. Bernstein. Linkster: enabling efficient manual inspection and annotation of mined data. In FSE'10, ACM. 2010.

[12] C. Corley, N. Kraft, L. Etzkorn, S. Lukins. Recovering traceability links between source code and fixed bugs via patch analysis. In TEFSE'11, IEEE CS, 2011.

[13] J. Cleland-Huang, A. Czauderna, M. Gibiec, and J. Emenecker. A machine learning approach for tracing regulatory codes to product specific requirements. In ICSE'10, pages 155–164. ACM, 2010.

[14] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *Working Conference on Reverse Engineering*, WCRE'03, pages 90–99. IEEE CS, 2003.

[15] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In ICSM'03, pp. 23–32. IEEE, 2003.

[16] A. Hindle, D. M. German, and R. Holt. What do large commits tell us?: a taxonomical study of large commits. *Int. working conference on Mining software repositories*, MSR '08, pages 99–108. ACM, 2008.

[17] S. Kim, J. Whitehead, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. on Software Engineering*, 34(2):181–196. 2008.

[18] S. Kim, T. Zimmermann, K. Pan, and J. Whitehead. Automatic identification of bug-introducing changes. In ASE'06, pages 81–90. IEEE CS, 2006.

[19] S. Kim, T. Zimmermann, J. Whitehead, and A. Zeller. Predicting faults from cached history. In ICSE'07, pages 489–498. IEEE CS, 2007.

[20] S. Kim, H. Zhang, R. Wu and L. Gong. Dealing with Noise in Defect Prediction. In ICSE'11, pages 481-490. IEEE CS, 2011.

[21] G. A. Liebchen and M. Shepperd. Data sets and data quality in software engineering. In *international workshop on Predictor models in software engineering*, PROMISE '08, pp. 39-44. ACM, 2008.

[22] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In ICSE'03, pages 125–135. IEEE CS, 2003.

[23] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11:309–346, July 2002.

[24] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In ICSM'00, pages 120–130. IEEE CS, 2000.

[25] T. Morse. Concurrent Versions System. Linux Journal. Vol no 21es. 1996.

[26] I. Myrtveit, E. Stensrud, and U. H. Olsson. Analyzing data sets with missing data: An empirical evaluation of imputation methods and likelihood-based methods. *IEEE Trans. Softw. Eng.*, 27:999–1013, Nov 2001.

[27] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen. A Topic-based Approach for Narrowing the Search Space of Buggy Files from a Bug Report. In ASE'11. IEEE CS, 2011.

[28] T. H. D. Nguyen, B. Adams, and A. E. Hassan. A case study of bias in bug-fix datasets. In WCRE'10, pages 259–268. IEEE CS, 2010.

[29] Openintents. http://www.openintents.org/.

[30] G. Salton and C. Yang. On the specification of term values in automatic indexing. *Journal of Documentation*, 29(4):351–372, 1973.

[31] A. Schröter, T. Zimmermann, R. Premraj, and A. Zeller. If your bug database could talk... In *Proceedings of the 5th International Symposium on Empirical Software Engineering*, pages 18–20, 2006.

[32] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *Int. workshop on Mining software repositories*, MSR'05, pages 1–5. ACM, 2005.

[33] K. Strike, K. El Emam, and N. Madhavji. Software cost estimation with incomplete data. *IEEE Trans. Softw. Eng.*, 27:890–908, October 2001.

[34] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In ESEC/FSE '11, pages 15–25. ACM, 2011.

[35] T. Zimmermann. Preprocessing cvs data for fine-grained analysis. In *MSR'04*, pp. 2-6. IEEE, 2004.

[36] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In PROMISE'07, pages 9-19. IEEE CS, 2007.

[37] Zxing. http://code.google.com/p/zxing/.

[38] Subversion SVN. http://subversion.tigris.org/.

[39] Bugzilla. http://www.bugzilla.org/.

[40] ReLink Project. http://www.cse.ust.hk/˜scc/Relink.htm.