2018 IEEE International Conference on Software Maintenance and Evolution

Linking Source Code to Untangled Change Intents

Xiaoyu Liu[1], LiGuo Huang[1], Chuanyi Li[2] and Vincent Ng[3]

[1]Department of Computer Science and Engineering, Southern Methodist University
[2]State Key Laboratory for Novel Software Technology, Nanjing University
[3]Human Language Technology Research Institute, University of Texas at Dallas

***Abstract*—Previous work [13] suggests that tangled changes (i.e., different change intents aggregated in one single commit message) could complicate tracing to different change tasks when developers manage software changes. Identifying links from changed source code to untangled change intents could help developers solve this problem. Manually identifying such links requires lots of experience and review efforts, however. Unfortunately, there is no automatic method that provides this capability. In this paper, we propose *AutoCILink*, which automatically identifies code to untangled change intent links with a *pattern-based link identification system* (*AutoCILink-P*) and a *supervised learning-based link classification system* (*AutoCILink-ML*). Evaluation results demonstrate the effectiveness of both systems: the pattern-based *AutoCILink-P* and the supervised learning-based *AutoCILink-ML* achieve average accuracy of 74.6% and 81.2%, respectively.**

***Index Terms*—Commit, Code change, Machine learning**

I. Introduction

During software development and evolution, source code changes are committed through version control systems (VCS) such as Subversion and Git. When changes are committed, the intents behind these changes are also documented in commit messages by developers. Problems arise, however, when changes are made in the same commit but for different intents (i.e., tangled changes [13])—developers either ignore or forget implementation details behind the changes when different change intents are aggregated in one single commit message [17]. Overlooking the individual change intents (i.e., one segment of change message sentences that is related to one unique task in this commit) in commit messages can lead to confusion, omissions and errors when developers validate changes, locate bug reports, (re)assign bug reports and trace changes to other software artifacts [1]. For example, if one developer committed two bug fix tasks that are tangled together, it could be hard for other developers to verify at the code level if both bugs are fixed without knowing which changed source code is related to which task. Therefore, it is critical and non-trivial to extract untangled change intents by slicing a commit message containing aggregated change intents, and identify/recover links between changed source code files and their untangled change intents—we speak of code to untangled change intent links. Herzig *et al.* [13] highlight the importance of this question by investigating the frequency of tangled changes and how big the impact of tangled changes is. Their investigation results confirm that tangled changes should be avoided. Unfortunately, to the best of our knowledge, no existing research has addressed the task of identifying code to untangled change intent links.

State-of-the-art practices commonly identify links between code changes and untangled change intents manually. However, this process is time-consuming, labor-intensive, and requires a great deal of experience. First, developers may have to review not only commit messages but also other software artifacts such as issue reports and pull requests. Figure 1 shows an example from project *gmail4j*, in which individual change intents described in the segmented commit message of commit *ae66810* need to be linked to the corresponding changed source code files. In this example, to identify the link between the changed source code file and the change intent described in segmented commit message "*Issue 13: quick fix*", the developer starts by locating *issue 13*. The developer notices that the change involves removing the package *maven-source-plugin*. To learn where exactly this change is made, she will need to go through all changed source code files before realizing that the changed file *ImapConnectionHandler.java* is linked to "*issue 13: quick fix*" since one of its imports *org.apache.maven.plugin* is removed.

Although no one has addressed this linking task, one may argue that Information Retrieval (IR)-based approaches such as Latent Semantic Indexing (LSI), Vector Space Model (VSM), Association-based approach (e.g., [3], [7], [10], [11], [22]) could be applied to generate links between commits and software artifacts (e.g., bug reports, design documents, etc.) by comparing the textual similarity between commits and software artifacts. However, these approaches cannot adequately address our task. The reason is that the changed entities extracted from source code (e.g., identifier, comments, string literals) could be very different from what is described in commit messages and other related software documents (e.g., issue reports, pull requests, etc.). As an example, consider Figure 1 again. The terms used to describe the change intent in the commit message (i.e., "quick", "fix", "remove", "plugin", etc.) are mostly different than the identifiers used in the changed source code file (i.e., "org", "apache", "maven", "plugin", "Abstract", "Mojo").

Motivated by this observation, we propose *AutoCILink*, a novel method with a supporting tool for automatically identifying/recovering links between the untangled change intents in segmented commit messages and the changed source code files. To address the issue of one source code file being changed for multiple intents, *AutoCILink* is designed to relate

2576-3148/18/$31.00 ©2018 IEEE
DOI 10.1109/ICSME.2018.00047

IEEE computer society

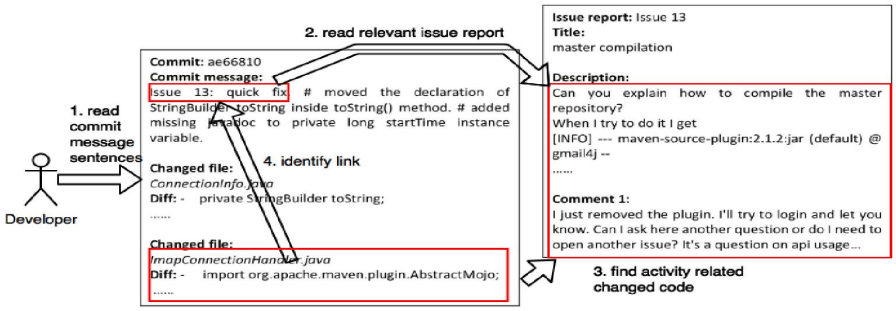

Fig. 1: An example of how software developers link changed source code files to untangled change intents

each changed file to one or more changed intents. Specifically, we have designed two variants of *AutoCILink*. To simulate the human reasoning process, we propose a *pattern-based link identification system* (*AutoCILink-P*) that leverages manually defined patterns to identify links between untangled change intents and changed code files. Motivated by the successful application of machine learning to software engineering tasks, we develop a *supervised learning-based link classification system* (*AutoCILink-ML*) to further understand the intents of source code changes and their associations. *AutoCILink-ML* identifies links between untangled change intents and code changes by encoding the patterns introduced in *AutoCILink-P* as features (*regular expression features* and *vocabulary features*) and employing two novel features: (a) *code import*, which considers changed imports from committed source code files; and (b) *untangled change intent count*, which considers the number of untangled change intents in a commit message.

The main contributions of this paper are:

- *Novel task.* To our knowledge we are the first to examine the task of automatically identifying links between changed source code files and untangled change intents.
- *New resources.* We have annotated a new corpus with links between untangled change intents and changed source code from 19 projects and make this corpus publicly available to stimulate research on this task.
- *Novel pattern-based approach.* We discover that (1) some patterns are recurrently used by developers to trace change intents and further build the links between changed code files and untangled change intents; and (2) related software documents (i.e., issue reports and pull requests) frequently provide root causes of code changes. We leverage these insights to design a *pattern-based link identification system* (a.k.a. *AutoCILink-P*). These patterns are designed with regular expressions that (1) extract changed entities (i.e., classes that encapsulate data and behavior [30]) from changed code files and (2) compute the similarity between these changed entities and the terms that appear in untangled change intents in commit messages as well as in other related software documents (i.e., issue reports and pull requests).
- *Novel learning-based approach.* To further our attempt to automatically generate links from code changes to untangled change intents, we have developed a supervised learning based link classification syst (a.k.a., *AutoCILink-ML*) using novel features specifically designed for this task.

The rest of this paper is organized as follows. Section II presents the motivating examples. Section III describes our approach. Section IV discusses the empirical evaluation setup and results. Section V describes threats to validity. Section VI summarizes related works. Section VII concludes and envisages future work.

II. Motivating Examples

We motivate the development of *AutoCILink* via two examples. Figure 1 shows an example from project *gmail4j* about how a developer reasons about the link between the change intent described in the segmented commit message "*Issue 13: quick fix*" and the changed source code file *ImapConnectionHandler.java* via related issue report *issue 13*. To reduce human efforts, if an IR-based approach (e.g., VSM) is employed to automate the task, this link is unlikely to be recovered due to the differences in the entities used in the source code *ImapConnectionHandler.java* and the terms used in the segmented commit message[1]. *AutoCILink-P* follows a different way to mimic the human reasoning process. First, commit message *ae66810* is automatically segmented into three parts, each of which describes a different intent: (1) "*Issue 13: quick fix*"; (2) "*# moved the declaration of StringBuilder toString inside toString() method*"; and (3) "*# added missing javadoc to private long startTime instance variable*". In addition, *AutoCILink* relates *issue 13* to (1) since (1) clearly indicates that it is relevant to fixing *issue 13*. Then to find which segmented commit message (i.e., (1), (2), or (3)) is linked to from one of the changed files *ImapConnectionHandler.java*, *AutoCILink-P* generates a regular expression using the extracted changed entity "plugin" (i.e., `^.*\s*remove(.*?)plugin(.*)`) from *ImapConnectionHandler.java* and applies it to each segmented commit message and their related software documents.

[1]The terms extracted from "*Issue 13: quick fix*" are: "issue","quick","fix". The entities extracted from *ImapConnectionHandler.java*: "org","apache","maven","plugin","abstract","mojo",etc.

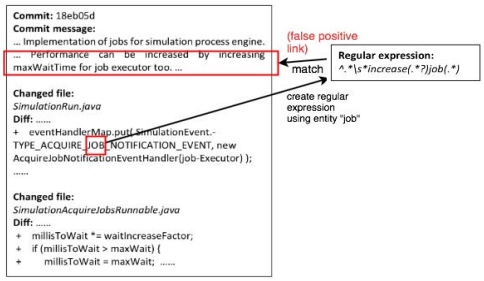

Fig. 2: An example of *AutoCILink-P* "greedy" matching

At last, *AutoCILink-P* finds a link from *ImapConnectionHandler.java* to segmented commit message (1) by matching the mentioned regular expression in *Comment 1* of (1)'s related software document (i.e., *issue 13*): "*I just removed the plugin...*" .

However, since regular expressions identify links based on simple word matching between source code and the text in a change intent, they could introduce errors. For instance, Figure 2 shows an example from commit *18eb05d* in project *activiti-crystalball*, in which *SimulationRun.java* can be erroneously linked to the sentence "*Performance can be increased by increasing maxWaitTime for job executor too.*" (i.e., false positive link in Figure 2) that is not the actual change intent by matching the regular expression `^.*\s*increase(.*?)job(.*)`. This regular expression is generated using the extracted changed entity "job" and it matches with the supporting text in the intent description "...increasing...job executor..." rather than the main topic of the intent (i.e., "...increasing maxWaitTime..."). To overcome this weakness, *AutoCILink-ML* is developed. By learning from known links, each feature is weighted by *AutoCILink-ML*, which can effectively reduce the negative effects imposed by regular expressions.

III. Approach

A. Overview

To enable the automated tracing from changed source code files to untangled change intents, we have designed two variants of *AutoCILink*: *pattern-based link identification system* (*AutoCILink-P*) and *supervised learning-based link classification system* (*AutoCILink-ML*). The workflow of both systems is shown in Figure 3. The subsequent sub-sections elaborate the text preprocessing and methodology employed by the two systems.

B. Text preprocessing

As shown in Figure 3, the text preprocessing component seeks to accomplish three tasks:

1) Untangling change intents: We untangle change intents in two steps. First, we employ the Stanford Tokenizer from the CoreNLP toolkit [28] to split the commit message into individual sentences. Then each sentence is further segmented if multiple change intents are aggregated in one sentence. The

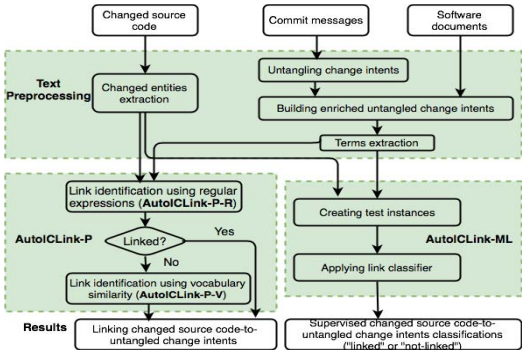

Fig. 3: *AutoCILink* system workflow

second step can be done when an additive transition word is used[2]. For instance, the following sentence in a commit message "*enable proguard by default and correct small mistake in debug log*" can be further split into two untangled change intents: "*enable proguard by default*" and "*correct small mistake in debug log*". We manually verified that 87% of the intents are correctly untangled by this procedure.

2) Building enriched untangled change intents: We leverage additional software documents related to the intent to build an *enriched* untangled change intents. If a software document (i.e., an issue report or a pull request) is related to any untangled change intent in a commit, it will be appended as supplemental description to the corresponding intent. Since this relationship is not always explicitly provided during software development and evolution, it is worthwhile to automatically recover the missing relations by performing the following analysis: **Step 1:** Pairing untangled change intents and software documents. In the example shown in Figure 1, *AutoCILink* groups three untangled change intents in commit *ae66810* with *Issue 13* report into 3 distinct pairs; **Step 2:** Similarity analysis. For each pair generated in Step 1, we apply Vector Space Model (VSM) [37] to calculate the cosine similarity score using Equation (1).

$$sim(i,d) = cos(i,d) = \frac{\sum_{t \in T} \omega(t,i,T) \times \omega(t,d,T)}{\sqrt{\sum_{t \in T} \omega(t,i,T)^2} \times \sqrt{\sum_{t \in T} \omega(t,d,T)^2}} \quad (1)$$

where t is a term, d is a software document, i is an untangled change intent, T is a corpus of terms taken from the commit messages and the software documents, $\omega(t,d)$ is the weight of term t in software document d, and $\omega(t,i)$ is the weight of term t in untangled change intent i. All terms are extracted by tokenizing the corresponding textual content. The weight of each term is calculated using Term Frequency and Inverse Document Frequency (*tf-idf*) (e.g., $\omega(t,d,T) = tf(t,d) \times idf(t,T)$), where $tf(t,d)$ is the term frequency of term t in software document d and Inverse document frequency (i.e., $idf(t,T) = log\frac{|T|}{|\{d:t \in d\}|}$) distinguishes rare terms from those common terms by giving more weight to rare terms in

[2]We apply an additive transition word list pre-defined in: https://msu.edu/user/jdowell/135/transw.html#anchor1671187

the corpus. Based on the similarity scores calculated in Step 2, *AutoCILink* determines that a given pair of software document and untangled change intent are related if $sim(i, d) > 50\%$, as Bacchelli *et al.* [31] indicate that using a 50% threshold yields better results than using other thresholds.

3) Extracting terms and changed entities: Next, *AutoCILink* extracts (1) terms from an enriched untangled change intents and (2) changed entities from changed source code files. Terms are extracted by tokenizing each untangled change intent and related software documents in the enriched untangled change intent. Since entities can be referenced in many different ways [31], *AutoCILink* extracts changed entities from changed source code identifiers, inline comments and string literals. Each entity is defined as a single noun word that represents a class that encapsulate data and behavior [30]. Specifically, since developers compound words through camel casing [31] (e.g., *ObjectContainer* is formed from "object" and "container") and underscore separator (e.g., *TYPE_END_SIMULATION* is formed from "type", "end" and "simulation"), these compounded words must be split into separate entities. In addition, each word in inline comments is tagged with its part-of-speech (POS) tag (a label that is assigned to a word to indicate its syntactic function) using the Stanford Log-linear Part-Of-Speech Tagger from the CoreNLP toolkit [28]. All words tagged with noun POS tags (i.e., NN, NNP, NNS, NNPS) are extracted as entities. Terms and changed entities are not only preprocessed by filtering common English stop words, but also stemmed using Porter Stemmer ([2], [34]) —a widely used algorithm that heuristically converts a word to its stem.

C. Pattern-based Link Identification System (AutoCILink-P)

As shown in Figure 3, the *pattern-based link identification system* (*AutoCILink-P*) operates in two steps;

1) Link identification using regular expression (AutoCILink-P-R): Link identification using regular expression (*AutoCILink-P-R*) is inspired by our observation that entities are usually named by their roles and responsibilities in system design [30].

Generating regular expressions. Given this observation, for each changed entity extracted from changed source code identifiers, inline comments and string literals (mentioned in Section B-3), *AutoCILink-P-R* generates the regular expressions based on the following two regular expression templates:

```
(1). ^.*\s*(<verb>)(.*?)<entity>(.*)

(2). ^.*\s*<entity>(.*?)(<verb>)(.*)
```

The intuition behind both templates is that in commit messages and software document, the terms that indicate changed entities could be separated from the governing verbs (simulating active voice in template (1) and passive voice in template (2)) by other characters (e.g., empty spaces, coma, etc.) or describing words. For example, in "*moved the declaration of StringBuilder toString inside toString() method*", active verb "*moved*" is separated from terms "*String*" and "*Builder*", which represents changed entities, by a sequence of words that describes "*String*" and "*Builder*" (i.e, "*the declaration of*").

As shown in both regular expression templates, $< verb >$ represents an acting verb in an untangled change intent. We define three kinds of verbs: reserved verbs, synonyms and frequent verbs. Reserved verbs and synonyms are defined by the nine change types in [1]. Each change type represents a kind of change in source code entity roles or responsibilities (i.e., *add new features*, *fix bugs*, *improve existing features*, *add code components*, *modify code components*, *delete code components*, *deprecate code components*, *refactor code components*, *modify code inline text*). All reserved verbs are taken directly from the verbs of various change types. As shown above, there are seven reserved verbs for the nine change types: "add" for *add new features* and *add code components*, "modify" for *modify code components* and *modify code inline text*, "delete" for *delete code components*, "deprecate" for *deprecate code components*, "refactor" for *refactor code components*, "fix" for *fix bugs*, and "improve" for *improve existing features*. To improve the likelihood of a match, we additionally employ the synonyms of these reserved verbs. The synonyms of each reserved verb manually identified based on its *synset* (i.e., sets of cognitive synonyms) defined in the WordNet [14] lexical knowledge base. Specifically, for each reserved verb, we manually collect all its *synsets* based on the intended meaning of the verb. For example, for reserved verb "add", we collect all the "synsets" for its intended meaning "*make an addition (to)*". Note that for each changed entity, we generate regular expressions using all seven reserved verbs from all possible change types and their synonyms.

In addition, *AutoCILink-P-R* leverages additional frequent verbs beyond the seven reserved verbs to generate regular expressions, because we observe that some verbs that are related to the nine change types are neither reserved verbs nor their synonyms. For example, the verb "update" is related to *improve existing features*, which is neither a reserved verb nor a synonym of the reserved verb "improve". Based on this observation, we mine verbs that most frequently occur in commit messages. Specifically, we first POS-tag each term in each commit message using the Stanford Log-linear Part-Of-Speech Tagger. Then we extract all the terms tagged with verb POS tags (i.e., VB, VBD, VBG, VBN, VBP, VBZ) and rank them based on their term frequency across all commit messages.

Applying regular expressions. We apply each generated regular expression on the enriched untangled change intents. If any match of the regular expression is identified in the enriched intents, *AutoCILink-P-R* reports a link between the untangled change intent and the changed source code file that contains the changed entity in the regular expression.

2) Link identification using vocabulary similarity (AutoCILink-P-V): Intuitively each changed source code file should be linked to at least one untangled change intent. If *AutoCILink-P-R* described in Section III-C-1 cannot identify any link from the given changed source code file to any untangled change intent, we identify the missing link

based on the vocabulary similarity between each changed source code file and enriched untangled change intents (*AutoCILink-P-V*). The motivation behind *AutoCILink-P-V* is that commit messages and software documents typically elaborate code change details with similar textual information with changed source code entities [2].

AutoCILink-P-V first computes the vocabulary similarity between enriched untangled change intents and changed source code files, which includes the vocabulary similarity between untangled change intent terms and changed source code entities (i.e, $sim(i, c)$) and the vocabulary similarity between terms in related software document (i.e., issue report or pull request) and changed source code entities (i.e, $sim(d, c)$):

$$sim(i,c) = \frac{\sum_{t \in V, e \in V} \omega(t,i,V) \times \omega(e,c,V)}{\sqrt{\sum_{t \in V} \omega(t,i,V)^2} \times \sqrt{\sum_{e \in V} \omega(e,c,V)^2}} \quad (2)$$

$$sim(d,c) = \frac{\sum_{t \in V, e \in V} \omega(t,d,V) \times \omega(e,c,V)}{\sqrt{\sum_{t \in V} \omega(t,d,V)^2} \times \sqrt{\sum_{e \in V} \omega(e,c,V)^2}} \quad (3)$$

in which *t* is a term, *i* is an untangled change intent, *d* is a related software document, *e* is a changed entity, *c* is a changed source code file, *V* is a corpus of extracted terms and entities, $\omega(e, c, V)$ is the weight of changed entity *e* in changed source code *c* in corpus *V*. Similar to $\omega(t, i, V)$ and $\omega(t, d, V)$, $\omega(e, c, V)$ is also calculated using the *tf-idf* approach.

In enriched untangled change intents, one untangled change intent could be related to multiple software documents. As a result, *AutoCILink-P-V* chooses the max value as the vocabulary similarity between the untangled change intent and changed source code file (i.e., $sim(Link_{(i,c)})$), as shown in Equation (4), in which the number of related software documents (d_x) is *n*:

$$sim(Link_{(i,c)}) = max(sim(i,c), \bigcup_{x=1}^{n} sim(d_x, c)) \quad (4)$$

To decide if there is a link or not, we set a threshold. The threshold for each experiment is selected by using a development set (details explained in Section IV-B). That is, when $sim(Link_{(i,c)})$ larger than selected threshold, *AutoCILink-P-V* reports a link between the changed source code file and the untangled change intent.

As we mentioned in Section II, *AutoCILink-P* may erroneously report certain links between changed source code files and untangled change intents as regular expressions are imprecise. To tackle this problem, we propose the *supervised learning-based link classification system*.

D. Supervised learning-based link classification system (AutoCILink-ML)

AutoCILink-ML operates in three steps:

1) Creating training instances: To train a link classifier to identify links, we create training instances for each changed source code and enriched untangled change intent pair in the training dataset. We annotate each training instance as *linked* or *not linked* depending on whether there is a link between the pair or not. Each training instance is represented using the following features:

Regular expression features are inspired by *link identification using regular expressions* described in Section III-C-1, which generates and uses regular expressions to identify code to untangled change intent links. Specifically, we create one binary feature for each such regular expression. Each feature encodes the presence (value=1) or absence (value=0) of the regular expression in the enriched untangled change intent in the training set. In the example shown in Figure 2, the untangle change intent in the commit message contains one regular expression feature (i.e., the one encodes the presence of `^.*\s*increase(.*?)<entity>(.*)`) having the value 1. Unlike in *AutoCILink-P*, in *AutoCILink-ML* the learning algorithm will have the flexibility to determine which of the regular expressions to use. For instance, if a regular expression is deemed useless, the learner can simply assign a low weight to the corresponding feature.

Vocabulary features are motivated by link identification using vocabulary similarity in Section III-C-2. Recall that a link is identified when vocabulary similarity score of the changed source code file and enriched untangled change intent pair is greater than 50%. Specifically, we create three types of vocabulary features: *vocabulary pair features*, *vocabulary similarity features* and *term unmatched features*.

We use (TERM, ENTITY) pairs extracted from change intent and code pairs to create *vocabulary pair features*. As described in Section III-B-3, terms are extracted from enriched untangled change intents and entities are extracted from changed source code files. Each term and entity are paired to create a (TERM, ENTITY) feature, whose value is 1 if the particular (TERM, ENTITY) pair appears in the change intent and code pair under consideration. Otherwise, its value is 0. Returning to the example in Figure 2, one of the vocabulary pair features created for the untangled change intent ("*Implementation of jobs for simulation process engine*") and changed source code file (*SimulationRun.java*) pair is *(process, job)*.

The *vocabulary similarity features* are created to encode the vocabulary similarity scores between an enriched untangled change intent and a changed source code file. We group the similarity scores into ten ranges: [0, 10%), [10%, 20%), [20%, 30%), [30%, 40%), [40%, 50%), [50%, 60%), [60%, 70%), [70%, 80%), [80%, 90%), [90%, 100%] and define ten binary features on these ten ranges. The feature value is 1 if the similarity score falls in the corresponding range. Otherwise, the value is 0.

Finally, for each changed source code and enriched untangled change intent pair, we create *term unmatched features*, which encodes the percentage of *unmatched* noun terms with changed entities in source code (i.e., the percentage of noun terms that do not appear in the set of words derived from the entities). Specifically, *AutoCILink-ML* first obtains all nouns (i.e., the words tagged by the Stanford Log-linear Parts-Of-Speech Tagger as NN, NNP, NNS, NNPS) from the tokenized segmented commit message. Then it calculates the percentage of noun terms that are

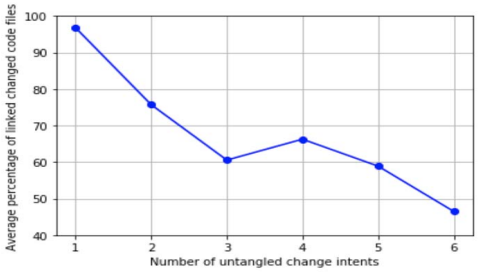

Fig. 4: Average percentage of linked changed source code files per untangled change intent count across all 19 projects

unmatched with changed entities. We define ten binary features, each of which encodes the presence (value=1) or absence (value=0) of the calculated percentage value in the range of: [0%,10%); [10%,20%); [20%,30%); [30%,40%); [40%,50%); [50%,60%); [60%,70%); [70%,80%); [80%,90%); [90%,100%]. In the example in Figure 2, there are 3 out of 7 (42.9%) noun terms ("time", "job", "executor") in the untangled change intent "*Performance can be increased by increasing maxWaitTime for job executor too*"[3], which match with the changed entities extracted from *SimulationRun.java*. Therefore, the feature representing range [50%,60%) will have its value set to 1 since the percentage of noun terms unmatched in changed entities (1-42.9%=57.1%) falls in the range of [50%,60%).

Code import features are motivated by the fact that the intent of a change in source code file may be affected by the changes of its imported code modules (i.e., imported class from the same project). We create 10 *code import features* (by discretizing the [0,1] range into 10 equal-sized intervals, similar to what was done for the *term unmatched features* described above) to encode the percentage of terms in the corresponding enriched untangled change intent that are unmatched in each changed imported code module's changed entities. Similar to *terms unmatched features*, the feature value is 1 if the calculated percentage falls in a specific range defined above.

Untangled change intent count features are inspired by our hypothesis that the less untangled change intents one commit message can be segmented into, the more likely each intent is linked to more changed code files in this commit. For example, when there are ten changed source code files but only one untangled change intent in one commit, most likely all changed source code files are linked to this single intent (i.e., percentage of linked changed code=100%). As another example, if there are ten changed source code files and ten untangled change intents segmented from one commit, the average percentage of linked changed code will be approximately 10% (i.e., 1/10=10%), with each untangled change intent linked with one distinct changed source code file. Figure 4 empirically validates our hypothesis based on data from 19 subject projects collected in the data preparation step (Section IV-A). Specifically, Figure 4 plots the percentage of changed source code files each untangled change intent is linked to on average against the number of untangled change intents in a commit message. As we can see, although there are minor fluctuation when untangled change intent count=3 and 4, the overall trend is obvious: the more the untangled change intents a commit message contains, on average the lower percentage of changed source code files is linked to each untangled change intent. Figure 4 shows that the percentage drops from 96.8% (when untangled change intent count=1) to 38.4% (when untangled change intent count=6).

Given these results, we create the *untangled change intent count features*, which are binary features that encode the number of untangled change intents in each commit message. The value of a feature is 1 if the commit message is segmented into the corresponding number of untangled change intents[4]. Otherwise its value is 0. For example, in Figure 1 the commit message *ae66810* is segmented into four untangled change intents. Therefore, the training instances that *AutoCILink-ML* creates for each changed source code file and enriched untangled change intent pair in this commit set the value of the feature *four untangled change intents* to 1.

2) Training and applying link classifier: We employ the support vector machine (SVM) learning algorithm with Radial Basis Function (RBF) kernel as implemented in the libSVM software package [15] to train the link classifier. This binary classifier determines whether the given untangled change intent and changed source code file are "linked" or "not linked". After training, the resulting classifier can be used to label each test instance. Test instances are created in the same way as the training instances.

IV. Empirical Evaluation

A. Data preparation

We randomly selected 19 open source software projects of different types and domains hosted on GitHub[5]. The data we collected for our experiments include all change commits, issue reports and pull requests from the 19 projects. Table I summarizes the details of these projects.

Given the dataset, we created a binary classification task, where we seek to identify each pair of changed source code file and enriched untangled change intent as "linked" or "not linked". Hence, for system training and evaluation, we manually determined whether there was a link between each pair of enriched untangled change intent and changed source code file, as described below:

[3]The terms extracted from the untangled change intent after tokenization are: "perform","increas","max","wait","time","job","executor"

[4]Based on our study across all 19 projects, the number of untangled change intents falls in the range of [1,7]. Thus 7 binary features are created.

[5]The 19 projects are: activiti-crystalball,androidkickstartr,deprecated-avaje-ebeanorm-api,zoola,SiVi,semaphore_manager,gmail4j,java-color-loggers,java-semver,jdf-stacks-client,kornakapi,picketbox-cdi,sonar-scm-stats,sonar-switch-off-violations,xml2csv,cascading-helpers,dasein-cloud-gogrid, elasticsearch-facet-script,fest-guava-assert

TABLE I: Dataset from 19 open source projects

Total # of projects	19
# of untangled change intents	572
# of changed source code files	2739
# of "linked" code-intent pairs	3025
# of "not linked" code-intent pairs	1288

1) Coding procedure: Two expert analysts, who are co-authors of this paper, were asked to conduct the coding task. First, one of the coders conducted a pilot study on a subset of all the untangled change intents and changed source code file pairs. The pilot study resulted in a list of preliminary coding criteria to identify links between untangled change intent and source code files. After that, this coder trained the other coder with coding criteria in a session that involved open discussion. To minimize subjectivity of coding, both coders coded all code-intent pairs in the dataset. Each coded pair was verified by both coders and disagreements were resolved through open discussion.

2) Coding results: We measured the inter-coder agreement between the two coders with Cohen's Kappa (k) [24]. Coders agreed on 86% (k=0.683, i.e., moderate agreement [25]) of the cases and then resolved their disagreements via discussion. Based on our analysis, the main causes of the disagreements are omissions and misunderstandings in several cases that the terms found in segmented commit messages (untangled change intents) have different meanings in the changed source code entities. For example, the word *initial* in untangled change intent "*initial Simulator implementation*" from project *activiti-crystalball* "*initial*" can be understood as "*the first time*". However, in the changed source code it actually means "*the fundamental*".

B. Experimental setup

1) Evaluation settings: To evaluate *AutoCILink-ML*, we apply leave-one-project-out cross validation. In each experiment, we use 17 projects for training *AutoCILink-ML*, one project for development (i.e., parameter tuning), and the remaining project as our held-out test set. We repeat this experiment 19 times, each time choosing a different project as our held-out test set. This ensures that the entire dataset is used for evaluation. For parameter tuning, we tune SVM's regularization parameter C. Intuitively, the larger the C value is, the higher the penalty on training error is. We tune C that maximizes the average accuracy (as discussed in Section IV-B-2) on the development set. To evaluate *AutoCILink-P*, we apply it to the entire dataset. In other words, results of both variants of *AutoCILink* are obtained on the entire dataset.

2) Baseline Systems: Since this is a new task, there is no existing system that we can employ as a baseline system. Nevertheless, as mentioned in the introduction, one could conceivably employ IR-based approaches as baselines for our task. Consequently, we employ three IR-based baselines, namely VSM [2], [6], LSI [4] and the Association-based approach [3]. Recall that LSI is based on the Vector Space Model that takes words appearing in a context into consideration. VSM calculates the distance between terms from enriched untangled change intents and entities from changed source code files. Association-based approach learns the associations between terms from enriched untangled change intents and the entities from changed source code. Since each IR-based baseline returns a similarity score for each test instance, we need to employ a threshold to determine whether a test instance should be classified as "linked" or "not linked". Specifically, a test instance whose similarity score is at least as large as the threshold will be classified as "linked". Otherwise, it will be classified as "not linked". To avoid giving an unfair advantage to *AutoCILink* whose learning-based variant employs annotated training data, we use the same amount of annotated training data for identifying the "best" threshold for each IR-based baseline. Specifically, when evaluating an IR-based baseline on a particular project, we apply the threshold that achieves the highest accuracy on the remaining 18 projects.

We employ two additional baseline systems: the *majority classifier* and the *untangled intent count classifier*. The *Majority classifier* takes a greedy approach in link identification, simply classifying each test instance into the majority class [23], which for our task means that each test instance will be classified as "linked". The *untangled intent count classifier* is designed based on the assumption that the less untangled change intents one commit message contains, the more changed source code files each untangled change intent in this commit will be linked to. Specifically, it first assigns a random score between 0 and 1 to each test instance (code-intent pairs). Then, a threshold is employed so that a test instance is classified as "linked" if and only if its score is below the threshold. The threshold is dependent on the number of untangled change intents in the commit message the test instance is asssociated with. Specifically, the threshold is set as the reciprocal of the count of untangled change intents in one commit message under consideration. For example, the threshold for a commit containing one untangled change intent commit is set as 1, while the threshold for a commit containing two untangled change intents is set as 1/2.

3) Evaluation metrics: We compute accuracy, as well as recall, precision, and F1-score on both "linked" and "not linked" instances. The recall (**R**) on the "linked"/"not linked" class is defined as the percentage of "linked"/"not linked" instances in the test set that are correctly classified (i.e., Recall = $\frac{TP}{(TP+FN)}$). The precision (**P**) on the "linked"/"not linked" class is the percentage of code-intent pairs classified as "linked"/"not linked" that are indeed correct "linked"/"not linked" classifications (i.e., Precision = $\frac{TP}{(TP+FP)}$). The F1-score (**F**) on the "linked"/"not linked" instances is the harmonic mean of "linked"/"not linked" recall and precision (i.e., F1-score=$\frac{2*R*P}{(R+P)}$)). Finally, we report **Accuracy**, which is the percentage of code-intent pairs correctly classified (i.e., Accuracy = $\frac{TP+TN}{(TP+FP+FN+TN)}$).

TABLE II: Evaluation results of *AutoCILink* and baseline approaches

	Systems	Linked Avg.F1	Not linked Avg.F1	Average Accuracy
1	**AutoCILink-ML**	87.4	62.4	81.2
2	**AutoCILink-P**	83.4	40.2	74.6
3	**LSI**	73.2	32.0	64.3
4	**VSM**	78.4	36.2	70.4
5	**Association-based**	80.4	7.8	66.5
6	**Majority**	82.4	0.0	70.1
7	**Untangled intent count**	77.9	55.9	70.0

TABLE III: Paired *t*-test results on accuracy of *AutoCILink-ML* vs other approaches accuracy

	System 1	System 2	*p* value
1	**AutoCILink-ML**	**LSI**	<0.0001
2	**AutoCILink-ML**	**VSM**	0.0002
3	**AutoCILink-ML**	**Association-based**	<0.0001
4	**AutoCILink-ML**	**Majority**	0.0920
5	**AutoCILink-ML**	**Untangled intent count**	0.0931
6	**AutoCILink-ML**	**AutoCILink-P**	0.0030

C. Evaluation results

This section evaluates the effectiveness of *AutoCILink* by addressing the four research questions.

RQ1. *How effective is AutoCILink in linking changed code to untangled change intents?*

We compare the effectiveness of two variants of *AutoCILink* systems (i.e., *AutoCILink-ML* and *AutoCILink-P*) with the five baselines (i.e., the three IR-based systems, the majority classifier, and the untangled intent count classifier. Table II shows the F1-scores of "linked" code-intent pairs, "not linked" code-intent pairs and the accuracy of each system averaged over the 19 projects using leave-one-project-out cross validation. As we can see, in terms of average accuracy, both *AutoCILink* variants (rows 1–2) outperform the baseline systems (rows 3–7). In particular, *AutoCILink-ML* achieves the best average accuracy (81.2%).

Table II shows several additional interesting results. First, we found *AutoCILink-ML* outperforms other systems on not only the "linked" instances but also the "not linked" instances. When compared with the five baseline systems, *AutoCILink-ML* improves the "linked" F1-score by 5–14.2% and the "not linked" F1-score by 6.5–62.4%.

In addition, we observe that *AutoCILink-P* also achieves better average accuracy than the baseline systems. However, its "not linked" F1-score is lower than that of the Untangled intent count-baseline. We hypothesize that this could be attributed to errors in finding matches via the regular expressions in enriched untangled intents, where many of the "not linked" code-intent pairs are misclassified "linked". We will examine this hypothesis as part of RQ2.

To determine whether the differences in average accuracy between the best performer (i.e., *AutoCILink-ML*) system and baseline systems are statistically significant or not, we employ the two tailed paired *t*-test. To show the soundness of choosing the two tailed paired *t*-test, we perform the Shapiro-Wilk normality test [39] with the null hypothesis that *the performance of corresponding system is normally distributed*. The test result for each system ($p > 0.05$) cannot reject the null hypothesis and the data of each system is normally distributed. For each significance test, our null hypothesis is: *there is no performance difference between the two systems under comparison*. Following [38], the result of significant test is interpreted as: (1) *highly significant* if $p < 0.01$; (2) *significant* if $0.01 \leq p < 0.05$; (3) *moderately significant* if $0.05 \leq p < 0.1$. Otherwise, the difference is *statistically indistinguishable*. The significance test results are shown in Table III (rows 1-5). Each row shows the *p* value of *AutoCILink-ML*'s result compared to one baseline system's result. We conclude from these results that *AutoCILink-ML* is more accurate than its peers with high or moderate statistical significance.

TABLE IV: Evaluation results of *AutoCILink-ML* and *AutoCILink-P*

AutoCILink Model	Linked, Avg			Not Linked, Avg			Average
	R	P	F1	R	P	F1	Accuracy
ML	81.7	94.0	87.4	79.0	51.6	62.4	81.2
P	90.8	78.4	83.4	36.3	68.1	40.2	74.6
P-R	89.8	67.5	75.8	31.0	69.1	37.3	67.9

RQ2. *Which system is more accurate in linking changed code to untangled change intents, AutoCILink-ML or AutoCILink-P?*

The first two rows of Table IV shows the results between the two variants of *AutoCILink*, namely *AutoCILink-ML* and *AutoCILink-P*. As we can see, *AutoCILink-ML* is considerably more accurate than *AutoCILink-P*, with average accuracies of 81.2% vs 74.6%. To determine whether this improvement is significant or not, we again employ a two tailed paired *t*-test on these systems. Row 6 in Table III shows that the improvement is highly significant. In addition, Table II shows that *AutoCILink-ML* outperforms *AutoCILink-P* by 4% in "linked" F1-score and by 22.2% in "not linked" F1-score.

To gain additional insights, we investigate the performance of *AutoCILink-ML* and *AutoCILink-P* in terms of their recall and precision in predicting "linked" and "not linked" instances, respectively. Results are shown in Table IV. It is interesting to see that although *AutoCILink-P* achieves better recall in predicting "linked" instances (90.8%) and better precision in predicting "not linked" instances (68.1%) than *AutoCILink-ML*, it results in much lower precision in predicting "linked" instances (78.4%) per project and recall (36.3%) in predicting "not linked" instances per project. Again, we hypothesize that this can be attributed to the "imprecision" of the regular expression based system (*AutoCILink-P-R*), which tends to identify code-intent links by searching for a match in the enriched untangled change intents, compromising its preciseness in pinpointing the actual links. To test this hypothesis, we evaluate the contribution of the regular expressions to *AutoCILink-P*'s performance. Specifically, we obtained results on the test set by running only "*AutoCILink-P-R*" (i.e., Step

TABLE V: Feature ablation results in average accuracy

Iter 1	**-Type 4**	**-Type 6**	**-Type 3**	**-Type 2**	**-Type 1**	**-Type 5**
	72.3	79.4	79.5	80.4	80.4	80.9
Iter 2	**-Type 4**	**-Type 6**	**-Type 3**	**-Type 2**	**-Type 1**	
	75.8	78.2	78.9	80.0	80.4	
Iter 3	**-Type 4**	**-Type 6**	**-Type 3**	**-Type 2**		
	74.1	77.1	78.9	79.7		
Iter 4	**-Type 4**	**-Type 6**	**-Type 3**			
	75.9	79.6	79.7			
Iter 5	**-Type 4**	**-Type 6**				
	63.1	69.0				

TABLE VI: Paired *t*-test results on *AutoCILink* feature selection

System 1	System 2	*p* value
AutoCILink-Type 4	**AutoCILink-ML**	$<$0.0001
AutoCILink-Type 6	**AutoCILink-ML**	0.0397

1 of *AutoCILink-P*). These results are shown in the last row of Table IV. As we can see, in comparison to *AutoCILink-ML*, *AutoCILink-P-R*'s higher recall on the "linked" instances and its higher precision on the "not linked" instances provide suggestive evidence for our hypothesis.

RQ3. *Which feature types have the largest impact on the performance of AutoCILink-ML?*

Recall that *AutoCILink-ML* employs six types of features: **Type 1**: regular expression features; **Type 2**: vocabulary pair features; **Type 3**: vocabulary similarity features; **Type 4**: untangled change intent count features; **Type 5**: code import features; and **Type 6**: terms unmatched features. To understand which feature type(s) have the largest impact on the performance of *AutoCILink-ML*, we perform feature ablation experiments in which we remove the feature types from the system one-by-one.

We show the results of the ablation experiments in Table V, where results are expressed in terms of average accuracy. The top line of the table shows what the system that uses all available features' score would be if we removed just one of the six feature types. So to see how our system performs if we remove only the untangled change intent count features (Type 4), we would look at the first row of results under the column headed by **-Type 4**. The number here tells us that the resulting system's average accuracy is 72.3%. Since *AutoCILink-ML* (when all feature types are used) achieves an accuracy of 81.2% (see Table IV), the removal of the untangled change intent count features costs the complete system 8.9% points in accuracy.

From row 1 of Table V, we can see that removing Type 5 (code import features) yields a system with the best average accuracy in the presence of the remaining feature types in this row. For this reason, we permanently remove the Type 5 features from the system before we generate the results in row 2. We iteratively remove the feature type that yields a system with the best performance in this way until we get to the last line, where only one feature type is used to generate each result.

TABLE VII: Distribution of misclassified pairs of changed source code and enriched untangled change intents

	Percentage of misclassification	Misclassified as	
		Linked	Not linked
AutoCILink-ML	18.8%	617	175

Since the feature type whose removal yields the best system is always the rightmost entry in a line, the order of column headings indicates the relative importance of the feature types, with the leftmost feature types being the most important to performance and the rightmost feature types being least important in the presence of other feature types. As we can see, the most important features are Type 4 and Type 6, as their removal results in a 18.1% and 12.2% drop in accuracy, respectively.

We conduct the paired *t*-test to determine whether the removal of either of these two types of features yields a system that performs significantly worse than the system that employs all six types of features. Similar to RQ1, we conduct the Shapiro-Wilk test to show that the performance data yielded by the system with certain types of features is normally distributed. Results are shown in Table VI. Specifically, row 1 compares the system using all six types of features (System 2) with the system that uses all but the Type 4 features (System 1), whereas row 2 compares the system using all six types of features (System 2) with the system that uses all but the Type 6 features (System 1). As we can see from the results, the difference between the two systems in each of these two experiments is statistically significant.

RQ4. *What is the root cause of mistakes made by AutoCILink-ML?*

To shed light on how to improve the performance of *AutoCILink-ML* in future work, we performed a comprehensive error analysis on misclassified pairs of changed source code and untangled change intents by the two variants of *AutoCILink* system. Table VII lists the percentage of misclassified instances and the number of test instances misclassified as "linked" and "not linked" by *AutoCILink-ML*. We notice that one typical error is the misclassification of a "linked" code-intent pair as "not linked" by *AutoCILink-ML*, which leads to relatively lower precision in predicting "not linked" pairs. Based on our analysis, the main reason for this type of misclassification is can be attributed to the inconsistent definitions/ambiguity of certain terms used in commit messages and their related documents and the same ones used in source code. For example, in the project *androidkickstartr*, one change intent untangled from the commit message says: "*remove the canonical name constants*", where "name" means "variable name". However, this intent is erroneously linked to changed source code file that contains an entity "name"[6] which means "by which a thing is known". To reduce this kind of errors, word sense disambiguation would be helpful. In WordNet [14], English words are grouped into sets of

[6]Extracted from changed source code identifier *packageName*

synonyms called "synsets". As mentioned in Section III-C-1, every meaning of a word is represented in a unique "synset". In this case, "synsets" features can be added to ensure the unambiguous interpretation of the term and/or entity under its own linguistic context.

V. Threats to Validity

One main threat to *internal validity* was introduced in the process of the manual coding of changed code to untangled change intent links (Section IV-A-1). To minimize subjectivity, we ensured that each changed code to untangled change intent link was coded by two coders independently. We also defined coding criteria and trained the coders via open discussion. To minimize the subjectivity, inter-coder agreement is measured to ensure the coding reliability.

External validity threats come from the generalization of our results. To strengthen this validity, our dataset covers all the changed source code, commit messages and their related software document from 19 open source projects across different types and domains. Nevertheless, the approaches proposed in the paper can be easily generalized to other projects.

VI. Related Works

This section summarizes two categories of research that *AutoCILink* is closely related to.

A. Analysis of commits and changed source code

Moreno *et al.* [5] examine the changed source code in commits and use them to automatically generate release notes with patterns. Hinton *et al.* [4] discover software release patterns from documents in commits. Several approaches [1], [7]–[9] are proposed to automatically generate commit messages from changed source code. Corts-Coy *et al.* [1] and Linares-Vasquez *et al.* [9] present automated tools to generate natural language commit messages by extracting entities and commit stereotypes from changed source code. Different from the code to untangled change intent linking task, their research are built upon the assumption that commit messages can't reflect all intents of source code changes. Buse *et al.* [7] present an automatic approach to describe source code modifications using symbolic execution and summarizations. Rastkar *et al.* [8] propose an approach to describe the motivation of source code changes with multi-document summarization technique. D'Ambros *et al.* [17] present a tool to augment commits with visual contexts of changes. Herzig *et al.* [13] analyze the impact of source code changes and find that up to 15% of bug fixes consist of multiple tangled changes. Dias *et al.* [35] propose a novel approach to group related changed source code together. Similarly, Kreutzer *et al.* [36] propose an approach to automatically cluster changed source code. Murphy-Hill *et al.* [18], [19] find that 30% of commits contain code refactorings mixed with other code changes. Barnet *et al.* [40] introduce an automatic technique for decomposing changesets according to code reviews. Kawrykow *et al.* [41] and Kirinuki *et al.* [42] present approaches to detect non-essential code differences and tangled changes by mining historical changes. Some related works [20], [21] analyze large commits and small source code changes. Different from the perspective of views of these approaches, AutoCILink links changed source code to untangled change intents. We propose some patterns based on domain knowledge to detect code to untangled change intent links. In addition, a *supervised learning-based approach* is proposed based on not only these patterns but also new features extracted from untangled change intents, software documents and changed source code files.

B. Linking commits to software artifacts

Nguyen *et al.* [3] propose an approach based on text matching and association patterns to find links between bug reports and changed source code. They not only extract text features from bug reports but also from changed source code. Wu *et al.* [10] present *ReLink* to automatically recover missing links between bug reports and changed files by text similarity, time intervals, bug owners and change committers. Sun *et al.* [2] utilize non-source documents in commits to find missing commit-issue links with a VSM model. Le *et al.* [7] generate links between issue and commits with enriched commit messages that summarize the intents and detailed modifications in commits. Sliwerski *et al.* [11] propose an approach to identify links between issues and commits and then analyze if commits induce further fix. Bird *et al.* [22] present *LINKSTER* to provide query interfaces to locate possible links between issues and commits. As mentioned, the difference between other commit to software artifact link tasks and our task is that our task focuses on the intents of source code changes instead of software function or description of bugs. In this case, an IR-based approach can't resolve this task. So *AutoCILink* does not rely on any IR-based approach. It instead proposes a *pattern-based link identification system* and a *supervised learning-based classification system*. Evaluation results show that *AutoCILink* outperforms IR-based approaches on code to untangled change intent links identification.

VII. Conclusions and Future Work

We proposed the *AutoCILink* system, which comprises a pattern-based approach and a learning-based approach, to address the novel task of automatically link changed source code files to untangled change intents. In experiments on a newly annotated corpus from the repositories of 19 open source projects with links between changed source code files and untangled change intents, we showed that *AutoCILink* outperformed all five baseline systems under comparison. *AutoCILink-P* achieves 83.4% F1-score on "linked" code-intent data, 40.2% F1-score on "not linked" data and an average accuracy of 74.6%. *AutoCILink-ML* further improves the performance and achieves 87.4% F1-score on "linked" data and 62.4% F1-score on "not linked" data, and an average accuracy of 81.2%. Future work will further experiment our systems on other software projects as well as explore additional features for code to untangled change intent link identification.

REFERENCES

[1] Corts-Coy, L. F., Linares-Vsquez, M., Aponte, J., Poshyvanyk, D. (2014, September). On automatically generating commit messages via summarization of source code changes. In Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on (pp. 275-284). IEEE.

[2] Sun, Y., Wang, Q., Yang, Y. (2017). Frlink: Improving the recovery of missing issue-commit links by revisiting file relevance. Information and Software Technology, 84, 33-47.

[3] Nguyen, A. T., Nguyen, T. T., Nguyen, H. A., Nguyen, T. N. (2012, November). Multi-layered approach for recovering links between bug reports and fixes. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (p. 63). ACM.

[4] A. Hindle, M.W. Godfrey, R.C. Holt, Release pattern discovery via partitioning: methodology and case study, Mining Software Repositories, 2007. ICSE Work- shops MSR07. Fourth International Workshop on, IEEE, 2007. 1919

[5] Moreno, L., Bavota, G., Di Penta, M., Oliveto, R., Marcus, A., Canfora, G. (2017). ARENA: An Approach for the Automated Generation of Release Notes. IEEE Transactions on Software Engineering, 43(2), 106-127.

[6] Le, T. D. B., Linares-Vsquez, M., Lo, D., Poshyvanyk, D. (2015, May). Rclinker: Automated linking of issue reports and commits leveraging rich contextual information. In Program Comprehension (ICPC), 2015 IEEE 23rd International Conference on (pp. 36-47). IEEE.

[7] R.P. Buse, W.R. Weimer, Automatically documenting program changes, in: Proceedings of the IEEE/ACM International conference on Automated software engineering, ACM, 2010, pp. 3342.

[8] Rastkar, S., Murphy, G. C. (2013, May). Why did this code change?. In Proceedings of the 2013 International Conference on Software Engineering (pp. 1193-1196). IEEE Press.

[9] M. Linares-Vasquez, L.F. Cortes-Coy, J. Aponte, D. Poshyvanyk, Changescribe: a tool for automatically generating commit messages, in: 37th IEEE/ACM International Conference on Software Engineering (ICSE15), Formal Research Tool Demonstration, 2015.

[10] Wu, R., Zhang, H., Kim, S., Cheung, S. C. (2011, September). Relink: recovering links between bugs and changes. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (pp. 15-25). ACM.

[11] J. Sliwerski, T. Zimmermann, A. Zeller, When do changes induce fixes? in: ACM Sigsoft Software Engineering Notes, 30, ACM, 2005, pp. 15.

[12] Wen, M., Wu, R., Cheung, S. C. (2016, September). Locus: Locating bugs from software changes. In Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on (pp. 262-273). IEEE.

[13] K. Herzig, A. Zeller, The impact of tangled code changes, in: Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on, IEEE, 2013, pp. 121130.

[14] Fellbaum, C.: WordNet: An Electronic Lexical Database. MIT Press, Cambridge (1998)

[15] Chih-Chung Chang and Chih-Jen Lin. 2011. LIB- SVM: a library for support vector machines. ACM Transactions on Intelligent Systems and Technology, 2(3):27.

[16] Vapnik, V.: The Nature of Statistical Learning. Springer, Berlin (1995)

[17] M. DAmbros, M. Lanza, and R. Robbes. Commit 2.0. In Workshop on Web 2.0 for Software Engineering (Web2SE 10), pages 1419, 2010.

[18] E. Murphy-Hill, A.P. Black, Refactoring tools: fitness for purpose, Software, IEEE 25 (5) (2008) 3844.

[19] E. Murphy-Hill, C. Parnin, A.P. Black, How we refactor, and how we know it, Softw. Eng., IEEE Trans. 38 (1) (2012) 518.

[20] A. Hindle, D.M. German, R. Holt, What do large commits tell us?: a taxonomical study of large commits, in: Proceedings of the 2008 International Working Conference on Mining Software Repositories, ACM, 2008, pp. 99108.

[21] R. Purushothaman, D.E. Perry, Toward understanding the rhetoric of small source code changes, Software Eng., IEEE Trans. 31 (6) (2005) 511526.

[22] Bird, C., Bachmann, A., Rahman, F., Bernstein, A. (2010, November). Linkster: enabling efficient manual inspection and annotation of mined data. In Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering (pp. 369-370). ACM.

[23] Thuraisingham, B., Kantarcioglu, M., Khan, L., Chen, H., Yang, C. C., Chau, M., Li, S. H. (2009). Intelligence and Security Informatics: Pacific Asia Workshop, PAISI 2009, Bangkok, Thailand, April 27, 2009. Proceedings.

[24] Jacob Cohen. 1960. A Coe cient of Agreement for Nominal Scales. Educational and psychological measurement 20, 1 (1960), 3746.

[25] Anthony J. Viera, Joanne M. Garrett, et al. 2005. Understanding Interobserver Agreement: The Kappa Statistic. Family medicine 37, 5 (2005), 360363.

[26] Marcus, A., Maletic, J. I. (2003, May). Recovering documentation-to-source-code traceability links using latent semantic indexing. In Software Engineering, 2003. Proceedings. 25th International Conference on (pp. 125-135). IEEE.

[27] J. S liwerski, T. Zimmermann, A. Zeller, When do changes induce fixes? in: ACM Sigsoft Software Engineering Notes, 30, ACM, 2005, pp. 15.

[28] Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J. R., Bethard, S., McClosky, D. (2014, June). The stanford corenlp natural language processing toolkit. In ACL (System Demonstrations) (pp. 55-60).

[29] Freitag, D. (2017, January). Greedy attribute selection. In Machine Learning Proceedings 1994: Proceedings of the Eighth International Conference (p. 28). Morgan Kaufmann.

[30] Dragan, N., Collard, M. L., Maletic, J. I. (2010, September). Automatic identification of class stereotypes. In Software Maintenance (ICSM), 2010 IEEE International Conference on (pp. 1-10). IEEE.

[31] Bacchelli, A., Lanza, M., Robbes, R. (2010, May). Linking e-mails and source code artifacts. In Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1 (pp. 375-384). ACM.

[32] Livia Polanyi. 2003. The Handbook of Discourse Analysis. Vol. 18. Wiley-Blackwell, Chapter The Linguistic Structure of Discourse, 265.

[33] Matthew B. Miles, A. Michael Huberman, and Johnny Saldana. 2013. Qualitative Data Analysis: A Methods Sourcebook (3rd ed.). SAGE Publications, Inc.

[34] M.F. Porter, 1980, An algorithm for suffix stripping, Program, 14(3) pp 130137.

[35] Dias, M., Bacchelli, A., Gousios, G., Cassou, D., Ducasse, S. (2015, March). Untangling fine-grained code changes. In Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on (pp. 341-350). IEEE.

[36] Kreutzer, P., Dotzler, G., Ring, M., Eskofier, B. M., Philippsen, M. (2016, May). Automatic clustering of code changes. In Proceedings of the 13th International Conference on Mining Software Repositories (pp. 61-72). ACM.

[37] C.D. Manning, P. Raghavan, H. Schutze, et al., Introduction to information re- trieval, vol. 1, Cambridge university press Cambridge, 2008.

[38] Miller, D. A. (1966). Significant and highly significant. Nature, 210(5041), 1190.

[39] Shapiro, S. S.; Wilk, M. B. (1965). An analysis of variance test for normality (complete samples). Biometrika. 52 (34): 591611.

[40] Barnett, M., Bird, C., Brunet, J., and Lahiri, S. K. (2015, May). Helping developers help themselves: Automatic decomposition of code review changesets. In Proceedings of the 37th International Conference on Software Engineering-Volume 1 (pp. 134-144). IEEE Press.

[41] D. Kawrykow and M. P. Robillard, Non-essential changes in version histories, in Proceedings of the 33rd International Conference on Software Engineering. ACM, 2011, pp. 351360.

[42] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto, Hey! are you committing tangled changes? in Proceedings of the 22Nd International Conference on Program Comprehension, 2014, pp. 262265.