Clustering Commits for Understanding the Intents of Implementation

Kenji Yamauchi, Jiachen Yang, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University, Japan
1-5, Yamadaoka, Suita, Osaka, 565-0871, Japan
{y-kenji, jc-yang, k-hotta, higo, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—This paper proposes a novel technique for clustering commits for understanding the intents of implementation. Such a classification of commits should be able to assist developers to understand commits related to particular requirements, for example, how and why has this function been implemented, or has this function suffered from any bugs? Our technique adopts a clustering algorithm on identifier names that are related to changes in each commit. Such an approch allows us to take the semantics of each commit into account without commit messages, and so our approach is robust for the situation where some commits lack accurate descriptions. We conducted a pilot study to confirm that our idea answers to our objective. The pilot study found some good examples that showed the usefulness of our approach, and there were some undesirable results that gave some ideas to improve it.**

***Keywords*—Commit Classification, Version Control System, Mining Software Repositories**

I. Introduction

Version control systems (VCS), such as git or Subversion, are very helpful for managing software artifacts during software evolution. They store changes on software artifacts when developers commit the changes. In other words, they record the history of the software systems. Hence, developers can understand how each function was implemented by reviewing past changes on its related artifacts, or even can revert past changes that seem incorrect or inappropriate.

To reap such benefits from VCSs, it is necessary to understand the intents of implementation in past commits. It should be beneficial for this objective to cluster commits based on requirements to which each commit is related. For instance, suppose a case where a feature was implemented by multiple commits, and the developer wants to revisit the evolution of the function. In this case, the developer can know which commits she/he should review if the commits related to the requirement "*implementing the feature*" are classified into the same cluster. Furthermore, when a developer wants to understand a commit in the category, the information in the other commits including the source code changes in them should be helpful.

There are some techniques for clustering commits to understand them. A typical approach is the one based on metadata of commits including commit messages or committer names. Commit messages are used because they directly represent what the commits do. This approach classifies commits via natural language processing or information retrieval techniques on the data [1], [2]. These techniques work well if commits were annotated well by its committers. Hence, it should be best suited to the objective under the ideal situation where developers carefully annotate every commit. However, developers often commit without any messages, or with incomplete and/or inaccurate messages. If a commit does not have a commit message, or has a commit message with incomplete or inaccurate description, the approach will fail to classify the commit into a proper category.

Another approach is based on syntax differences of source code in each commit. This approach detects structural differences from every commit, and classifies commits based on them [3]. It does not use commit messages, and so it works well even though some commits lack good commit messages. However, this approach uses only syntactic information, and so it might miss semantic information that is valuable for understanding commits. Hence, this approach will not be suitable for our objective, which is clustering commits to understand the intents of implementation in each commit.

This paper proposes a novel approach for clustering commits. Our approach detects identifier names related to changes in every commit, and classifies every commit via the bag-of-words model with the identifier names. Our approach does not use commit messages, which allows it to cover the weak point of the commit message based approach. Furthermore, it can consider semantics of commits because identifier names should provide us with an insight into semantics of changes occurring in each commit [4].

We implemented our approach as a prototype tool, and conducted a pilot study with it to discuss the pros and cons of our approach. The pilot study found some desirable examples of clustering, which supports the usefulness of our approach. On the other hand, the pilot study also found some undesirable examples, but they give us insights to improve our approach.

The remainder of this paper is organized as follows. First of all, we introduce related work in Section II. Section III describes our proposed technique. Section IV shows the result of the pilot study. In addition, we discuss the validity of the result in Section V. Finally, Section VI concludes this paper.

II. Related Work

Hindle et al. proposed an automated technique to classify commits into maintenance categories, namely Corrective, Adaptive, Perfective, Feature Addition, and Non Functional [1]. Their technique trains the machine learning models with word distribution of commit messages, names of committers, and the number of changed files per directory.

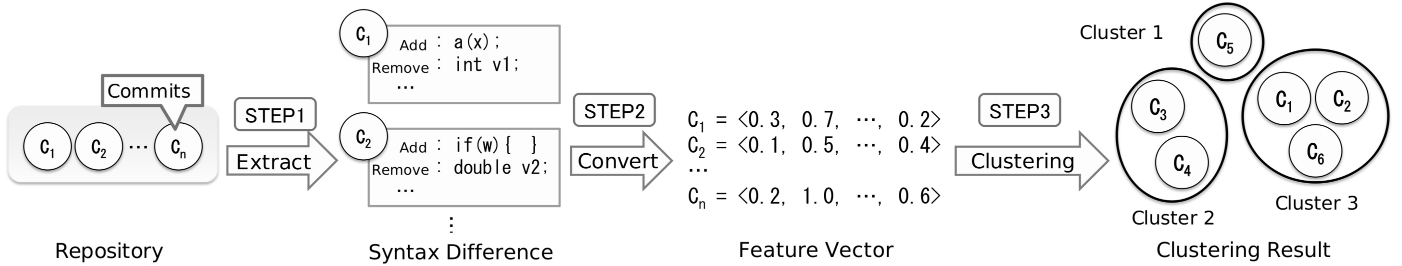

Fig. 1. The process flow of clustering commits

Hattori and Lanza conducted a study on the nature of commits by classifying them with their size and commit messages [2]. They reported that the majority of commits change very few files, and the majority of tiny commits are not related to development activities including adding a new feature or testing.

Dragan et al. proposed a technique to classify commits to understand design changes of software systems [3]. They were interested in design changes, and so they focused on method and class level changes. Their technique is based on method stereotypes [5], which generalize intrinsic or atomic behavior of the method. They classify commits into some categories including *Structure Modifier* and *Behavior Modifier*, and they stated that these classifications assist developers in understanding the extent and impact of changes in the commits.

Although the objectives of the existing research differ from each study, all of these techniques share the common avenue, classifying commits, for their objectives. Our objective also differs from the ones of the above studies, which is to cluster commits based on their related requirements. To achieve our objective, we should consider the semantics of commits. In addition, we should not use commit messages for practical use in a non-ideal situation where some commits lack commit messages. Hence, we propose a novel approach which considers the semantics of source code changed in each commit by analyzing identifier names affected by the changes.

There exists a large body of work that applies information retrieval techniques to the field of software engineering, which includes the work of Maletic and Marcus [6]. They proposed a technique to cluster software components with latent semantic indexing. Although their work and ours aim at different goals, but they are similar in that both of them use information retrieval technique on software artifacts.

III. Proposed Method

In this section, we will discuss our proposed method to classify the commits into clusters that correspond to the requirements.

The inputs of our method are the software repositories organized by the VCSs. The outputs are the commits in the repositories that have been classified by the different intents correspond to the requirements. We classify the commits based on the modifications on the source files, i.e. the occurrence of identifiers in the source code between the revisions. We do not take other information such as commit messages or committers into account currently.

The flow of processing of our proposed method consists of the following three steps, as shown in the Figure 1.

Step 1 Obtains the *syntax differences* of each commits from the source code repository. The result has finer granularity than the line-based differences which can be easily obtained from the VCSs.

Step 2 Extracts the identifier names from the syntax differences which are obtained from the Step 1. Further, we divide the identifier names into English words and count the occurrence of each word, to generate a *feature vector* of the syntax difference.

Step 3 Applies a clustering algorithm on the commits using the feature vectors to form the clusters that correspond to the requirements. We consider each resulted cluster as a group of commits to implement a single intent of a requirement.

We treat all the operations in the commit as the modifications to the contents of the files. In other words, we consider the addition of a new file as the modification of adding all the contents to an empty file, and the removal of an existing file as the modification of deleting all the contents from the file. Further, we consider the rename operation on a file as no modifications to the contents of the file. As a result, instead of tracking each operation recorded in the commit, we only process one kind of the operation, i.e. the modification of the file contents.

In the following subsections, we will discuss each step in detail.

A. (Step 1) The Extraction of the Syntax Differences

Firstly, we obtain a list of modified source code files in each commit. For each modified file, we utilize the approach of Change Distilling [7] to get the syntax difference in the revisions of the commit. The output of the Change Distilling is a list of the modifications, of each corresponding to either a statement or an expression fragment from the source code.

We illustrate the extraction using an example with the source code modifications, which are shown in the Figure 2. The extracted modifications will be three syntax differences, which are `private int logCounts`, `!IS LOGGED` and `log(messages)`.

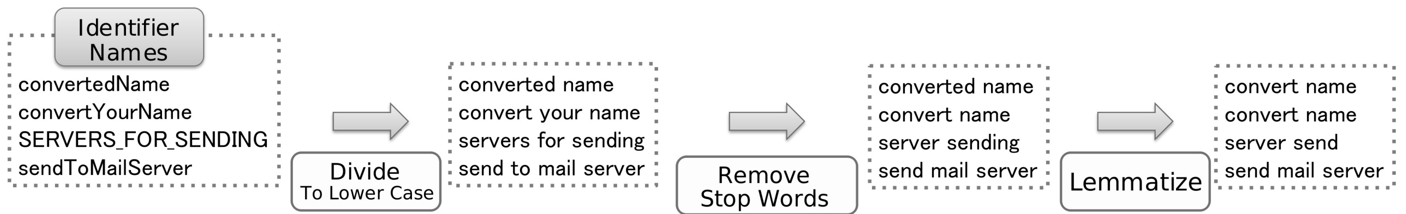

Fig. 3. An example of modifications to the source code

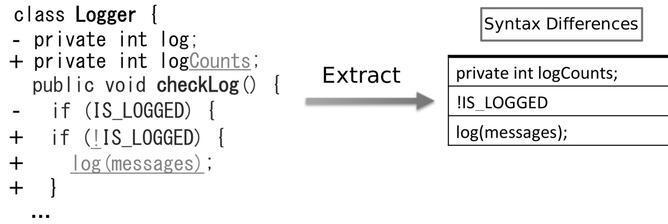

Fig. 2. An example of syntax differences extracted from a commit.

*B. (*STEP 2*)The Extraction of the Identifier Names and the Generation of Feature Vector*

Secondly, we extract the identifier names from each syntax difference. Previous study [8] suggested that the structural information will aid to the information retrieval, therefore we also extracted the method name and class name that contains the syntax difference, in addition to the identifier names. If the syntax difference appeared inside a method body, the names of the methods and the classes that contain the syntax difference are extracted as well. For the syntax difference appeared outside a method body, the class name is extracted, in addition to the modified identifier names. These syntax differences outside a method body often represent the modifications to the declarations of the methods or fields. The additional method and class names will be treated as appearing once in the syntax differences.

In the previous example of Figure 2, we extract the identifier names from the three syntax differences. For the syntax difference `private int logCounts`, we extract the variable name `logCounts` together with its class name `Logger`. Finally, we will get a set of identifier names: `logCounts`, `Logger`, `IS_LOGGED`, `Logger`, `checkLog`, `log`, `messages`, `Logger` and `checkLog`.

Next, we generate the feature vectors for clustering. This step involves the following 4 sub-steps:

STEP 2A Divide the identifier names into English words.

STEP 2B Lemmatize the English words into their root words. The extracted root words will be referred as *feature word*s in our following discussion.

STEP 2C Count the occurrences of each feature word.

STEP 2D Convert the occurrences of the feature words into a feature vector.

*1) (*STEP 2A*) Divide Words:* We divide the identifier names extracted in STEP 1 into English words, assuming they are named in the camel case or snake case. The camel case is a naming convention that each word begins with a capital letter, such as `getYourName`. The snake case is another naming convention that words are separated with underscore character, such as `MAX_COUNTS`. Furthermore, we convert the divided words into lower case to help the further process. For example, given the identifier names `convertedName` and `MAX_COUNTS`, we divide them into `converted name` and `max counts`, respectively.

*2) (*STEP 2B*) Lemmatization:* Next, for each divided words, we remove its prefix and suffix and lemmatize them into their root words in English[1]. For the previous example, we will convert `converted` and `counts` to their root words `convert` and `count`, respectively.

An example combining STEP 2A and 2B is shown in Figure 3. In this example, we extract from 4 identifier names, namely `convertedName, convertYourName`, `SERVERS_FOR_SENDING` and `sendToMailServer`, and get a list of words `convert, name, your, server, for, send, to, mail` from the lemmatization, to be used as the *feature words*.

*3) (*STEP 2C*) Count the Occurrence of Each Feature Word:* In this step, we count the occurrence of each word in the extracted feature words. We ignore the occurrence of the stop words of English[2] during this step. The stop words are those words that frequently appear, but bring little semantic meaning. We ignore them to avoid negative influences on the classification. As far as the example in the Figure 3 is concerned, we have `convert` twice, `name` once, `server` twice, `send` twice and `mail` once.

*4) (*STEP 2D*) Convert to Feature Vector:* Lastly, we convert the counted occurrences into a *feature vector*. For each commit, we define the feature vector as a Bag-Of-Words (referred as BOW) vector that takes the normalized occurrence of feature words as its features.

As a result, the number dimensions of the total BOW vector will be the number of unique words that appeared in the identifier names from all the syntax differences of all the commits. A value of 0 in the vector indicates that the corresponding word does not appear in the syntax difference of the commit.

For example, in commit A the feature words `a, b` appear once for each, and in commit B the feature words `b, c` appear

[1]We used the `WordNetLemmatizer` defined in python library http://www.nltk.org/

[2]We used the list of English stop words defined by python library http://scikit-learn.org/

twice for each. If we arrange the BOW vector in the order of `a, b, c`, then the BOW feature vector for commit A and B will be (1, 1, 0) and (0, 2, 2) respectively.

C. STEP 3 *Apply the Clustering Algorithm*

Lastly, we cluster the commits using the feature vectors obtained from STEP 2D. Among all available clustering algorithms, we considered the following 2 requirements, and then choose the Repeated Bisection [9]:

- No need to specify the expected number of clusters. In our approach, it is difficult to approximate the number of different requirements, i.e. the number of expected clusters. Therefore, we can not use the algorithms that need to specify the number of expected clusters.
- High scalability. For those long lasting projects, it is quite possible that there is a huge amount of commits recorded in their repositories. Therefore, we need a clustering algorithm with high scalability.

Besides Repeated Bisection, there are several algorithms that do not need to specify the number of clusters. Some of the algorithms are costly in terms of computing complexity, which require a long time for processing a large amount of data. Repeated Bisection fulfilled both requirements as it is faster than other algorithms that do not require the number of clusters [9]. Finding and experimenting with other clustering algorithms will be a future work of this study.

IV. PILOT STUDY

In this section, we will describe the experiments that we have done as our pilot study. We applied our proposed method on several software repositories and clustered the commits to confirm whether there exists clusters that correspond to the requirements.

A. *Confirm the Clustering Result*

In this experiment, we applied four open source software (referred as OSS) repositories, and observed the resulted number of clusters.

We selected four targets that are written in Java and organized by Git, with relatively large number of commits (more than 5000) in their repositories.

In Table I, we show the clustering result for each target repository. These repositories contain the commits that have no modifications to the source code files, therefore we filtered these commits and count the target commits that have

[1] https://lucene.apache.org/solr/
[2] http://jenkins-ci.org/
[3] http://www.wildfly.org/
[4] http://jruby.org/

TABLE I. CLUSTERING RESULT

Projects	# of Total Commits	# of Target Commits	# of Clusters
Lucene/Solr[3]	11,159	7,341	919
Jenkins CI[4]	17,640	8,452	1,102
WildFly[5]	14,280	10,247	1,204
JRuby[6]	20,531	13,142	1,487

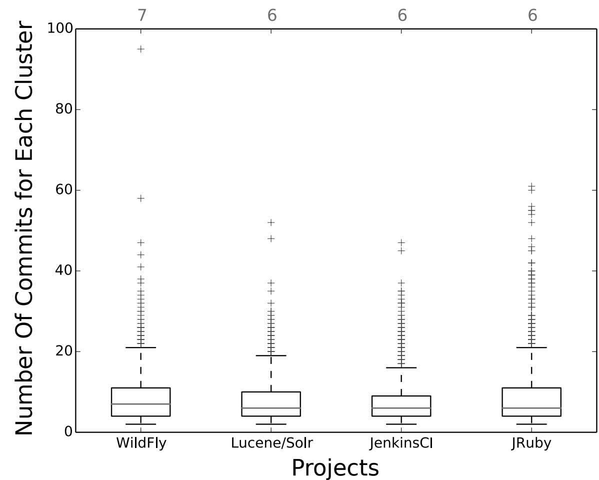

Fig. 4. The Distribution of the Number of Commits for Each Cluster

modifications in the source code. In Figure 4, we show the distribution of the number of commits for each resulted cluster in a boxplot. The X-axis of this figure is the four software projects. We can see from this figure that the median number of the commits are around 6 or 7 for every project.

1) Example of Desirable Clustering Result: Firstly we will show an example from the properly clustered commits. We show one cluster from the analysis of commits of Lucene/Solr in the Table II. This cluster consists of four commits. For the three commits with ID [7] e4a64f5, 5adc910 and c5a985e, we found the commit messages of them mentioned the same issue ID SOLR-4275 [8] that corresponds to a single implementation requirement. We show the top 5 weighted words of this cluster in Table III. The weight of the words are calculated by the Repeated Bisection algorithm. By viewing the syntax difference and the commit message of the commit a3e95d0, we confirmed that it is related to the implementation of the offset used in class `TokenTokenizer`. Further, a bug that introduced by this commit was fixed by the commit c5a985e. From the timestamp of the commit a3e95d0, we can confirm that it was committed 2 years before the other 3 commits. Therefore, we can confirm the intent of the commit a3e95d0 was implementing the requirement while the intents of the

[7] The source code and other information of these commits can be accessed from https://github.com/apache/lucene-solr/commit/*ID*
[8] http://issues.apache.org/jira/browse/SOLR-4275

TABLE II. DESIRABLE CLUSTERING RESULT (LUCENE/SOLR)

ID	Modified Files	Timestamp	Associated Issue ID
e4a64f5	TestTrie.java	6 Jan 2013	SOLR-4275
5adc910	TestTrie.java	6 Jan 2013	SOLR-4275
c5a985e	TrieTokenizerFactory.java	6 Jan 2013	SOLR-4275
a3e95d0	TrieTokenizerFactory.java	20 Jan 2011	*None*

TABLE III. TOP 5 WEIGHTED WORDS OF THE CLUSTER IN TABLE II

Word	Weight from Repeated Bisection
trie	0.75257
tokenizer	0.471848
ofs	0.235517
step	0.228368
prec	0.17303

commits e4a64f5, 5adc910 and c5a985e are fixing the issues of the same requirement. To summarize, these four commits in the same cluster have the same intent to implement the same requirement.

2) Example of Undesirable Clustering Result: Next we will show an example from the undesirable clustered commits. We show one cluster from the analysis of commits of Jenkins CI in the Table IV, which consists of three commits.

Firstly we discuss the two commits with ID b3553d6 and 7d0bac1 that modifies `IOUtils`. The purpose of this class was to provide the utilities about the input and output operations in the software. Therefore, it consists of a group of static member functions that are independent in features. Therefore, we considered that the implementations of these static methods related to different intents. By viewing the syntax differences in commit b3553d6 and 7d0bac1, we confirmed that they are implementing different static methods. Therefore, we considered that they are implemented for different requirements.

Secondly we viewed the syntax difference of commit a545a39 and found that it is modifying the access level of the inner classes of the class `Launcher`, which can also be confirmed from the commit message. Because the implementations of these inner classes are performing IO operations, they are grouped with the other 2 commits.

As the discussion showed, the commits in Table IV were implementing different requirements. Therefore, this cluster was undesirable for our purpose, which is to group the commits with the same implementation intent.

V. Validity of the Result

We discuss the validity of our result in this section.

A. Validity of the Proposed Method

Firstly, we assumed that the identifier names in the commits can reflect the implementation intent. Therefore, our method can generate undesirable result when the identifier names are inadequate. Secondly, we assumed that the commits are the minimum units to divide the implementation. The existing study suggested that a commit can contain multiple implementation contents with different intents. Previous studies [10], [11] in our research group have focused on identifying and dividing these tangled commits. They are the enabler of this research.

B. Validity of the Discussion

In the discussion we considered the requirements that are distributed among multiple source files by showing the percentage of the *simultaneous modification cluster*s in the software repositories, in order to check the effectiveness of our approach. However, we did not manually check the resulted clusters to verify whether they are really corresponding to a single implementation intent. Therefore, it is necessary to study the precision of the identified *simultaneous modification cluster*s in our further research.

TABLE IV. Undesirable Clustering Result (Jenkins CI)

ID	Modified File	Commit Message
b3553d6	IOUtils.java	added a convenience method
7d0bac1	IOUtils.java	doh
a545a39	Launcher.java	for serialization work these interfaces need to be public

VI. Conclusion

This paper proposed a technique to classify commits into requirements to understand the intents of implementation. It used a clustering algorithm with identifier names that are affected by changes in each commit. This approach allows us to consider the semantics of commits without commit messages, and so the proposed technique will work well even though some commits do not have enough descriptions.

We conducted a small pilot study on open source projects with a prototype tool. The pilot study confirmed the usefulness of our approach, and gave us some insights to improve the proposed technique.

As future work, we are going to improve the accuracy of classification of the proposed technique by other clustering algorithms or by improving word extraction from identifiers. In addition, we are going to seek other information that can be extracted from source code and can contribute to improve the accuracy of classification.

Acknowledgement

This work was supported by MEXT/JSPS KAKENHI 25220003, 24650011, and 24680002.

References

[1] A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt, "Automatic Classification of Large Changes into Maintenance Categories," in *Proceedings of the 17th International Conference on Program Comprehension*, 2009, pp. 30–39.

[2] L. P. Hattori and M. Lanza, "On the Nature of Commits," in *Proceedings of the 4th ERCIM Workshop on Software Evolution and Evolvability*, 2008, pp. 63–71.

[3] N. Dragan, M. L. Collard, M. Hammad, and J. I. Maletic, "Using Stereotypes to Help Characterize Commits," in *Proceedings of the 27th International Conference on Software Maintenance*, 2011, pp. 520–523.

[4] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic Clustering: Identifying Topics in Source Code," *Information and Software Technology*, vol. 49, no. 3, pp. 230–243, 2007.

[5] N. Dragan, M. L. Collard, and J. I. Maletic, "Reverse Engineering Method Stereotypes," in *Proceedings of the 22nd International Conference on Software Maintenance*, 2006, pp. 24–34.

[6] J. I. Maletic and A. Marcus, "Supporting Program Comprehension Using Semantic and Structual Information," in *Proceedings of the 23rd International Conference on Software Engineering*, 2001, pp. 103–112.

[7] B. Fluri, M. Wursch, M. PInzger, and H. C. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[8] B. Bassett and N. A. Kraft, "Structural information based term weighting in text retrieval for feature location," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 2013, pp. 133–141.

[9] G. Karypis, "Cluto-a clustering toolkit," DTIC Document, Tech. Rep., 2002.

[10] N. Kusunoki, K. Hotta, Y. Higo, and S. Kusumoto, "How much do code repositories include peripheral modifications?" in *Software Engineering Conference (APSEC, 2013 20th Asia-Pacific*. IEEE, 2013, pp. 19–24.

[11] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto, "Hey! are you committing tangled changes?" in *ICPC*, 2014, pp. 262–265.