A Study of Repetitiveness of Code Changes in Software Evolution

Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, and Hridesh Rajan
Iowa State University
Email: {hoan,anhnt,tung,tien,hridesh}@iastate.edu

***Abstract*—In this paper, we present a study of repetitiveness of code changes in software evolution. Repetitiveness is defined as the ratio of repeated changes over total changes. Focusing on fine-grained code changes, we model a change as a pair of old and new AST sub-trees within a method. A change is considered repeated within or cross-project if it matches another change having occurred in the history of the project or another project, respectively. We report the following important findings. First, repetitiveness of changes could be as high as 70-100% at small sizes and decreases exponentially as size increases. Second, repetitiveness is higher and more stable in cross-project setting than in within-project one. Third, fixing changes repeat similarly to general changes. Importantly, learning code changes and recommending them in software evolution is beneficial with accuracy for top-1 recommendation of over 30% and top-3 of nearly 35%. Repeated fixing changes could also be useful for automatic program repair.**

I. Introduction

In a project, software artifacts are written and maintained by human beings. "To err is human", thus, software is also defect-prone. Developers could repeat their own mistakes or unknowingly repeat the errors from others. A reason for that is the nature of software reuse and its practice by software engineers to save development effort. Common programming tasks expressed in programming languages may lead to similarity in source code. Software reuse could be at different levels of abstraction. Multiple software projects could share common specifications, designs, or algorithms. They may reuse the same libraries and frameworks, resulting in API usage patterns or common programming idioms in source code. Such similar code may lead to the *similar software changes* and *repeated defects and fixes* within or across different projects.

Several approaches in mining software repositories (MSR) have taken that observation and advanced its applications in automating several software evolution and maintenance tasks. An example application is *automatic program repairing* [11], [17] based on previously seen fixing patterns in the same or different projects. PAR [17] is an automatic pattern-based program repair method that learns common patterns from prior human-written patches. FixWizard [29] recommends fixes based on the code peers/clones and code with similar API usages. Weimer *et al.* [11] proposed GenProg, a patch generation method that is based on genetic programming. Other types of application are *automated library update*, *language/library migration*, etc. SemDiff [5] is a method to learn from previous updating changes to a framework in order to update its client code. LibSync [28] learns adaptation change patterns from client code to update a given program to use the new library version. Zhong *et al.* [34] mine common code transformation to support language migration.

While those approaches have gained much success in MSR, they focus on respective application domains and are often studied on small-scale settings with small sets of subject projects. There is still no large-scale, systematic study on how repetitive software changes are across the histories of software projects, what are the repetitiveness characteristics of software changes, or whether fixing changes exhibit different repetitiveness than general ones. To address them, we conducted a large-scale study with the following key research questions: R1) how code changes repeat in software evolution, and R2) how useful those repeated and previously seen changes/fixes within or across different projects are. The answers for those questions not only provide the empirical evidences but also could enhance those aforementioned MSR approaches. For example, a genetic programming-based automatic program repair could avoid unnecessary mutations by considering the information on the popular types and sizes of program elements that have been used in fixes for certain program contexts, thus, reducing their search space for possible fixes. Language migration or library update methods could benefit in similar manners when the repetitive characteristics of changes are studied.

In our study, we collected a large-scale data set consisting of 2,841 Java projects, with 1.7 billion lines of code (LOCs) at the latest revisions, 1.8 million code change revisions (0.4 million fixing ones), 6.2 million changed files, and 2.5 billion changed LOCs. We extracted consecutive revisions and compared their abstract syntax trees (ASTs). A change is modeled as a pair of subtrees in the ASTs. A change (s, t) is considered as matching another change (s', t') if s and s', and t and t' structurally match with the abstracting on the literal and local variables' nodes. The size of a change (s, t) is measured as the height of the sub-tree s in the source AST. Its type is defined as the AST node type of s. We perform the analysis in two settings: within and cross-project. In the within-project setting, a change in a project is considered as repeated if it matches another change previously occurred in the project's history. In the cross-project setting, it is considered as repeated if it matches another change occurring in another project. Repetitiveness is computed via the number of repeated changes over the total number of changes. We studied repetitiveness of changes in three dimensions: size, type, and general/fixing changes.

The key findings in our study include the following:

1) Repetitiveness is very high for changes of small sizes (up to 60-100% for the changes of sizes 1 and 2), however, it decreases exponentially as size increases. Repetitiveness for changes with sizes larger than 6 is very small. Thus, the above automatic tools should consider change fragments with the sizes from 2-6 (changes of size 1 are on literals, identifiers, modifiers, etc).
2) Repetitiveness also varies by syntactic types of changes. Changes involving simple structures (e.g. array accesses, method calls) are highly repetitive, while those with compound structure (e.g. control/loop statements) are less. In addition, most popular types of fixing changes include method calls, infix expressions, condition (e.g. if) and loop statements (e.g. for, enhance for). Thus, program repair tools could focus on those types with small sizes in their search space, and then combine them.
3) *Cross-project* repetitiveness is generally higher and more stable than *within-project* one. In addition, while cross-project repetitiveness of *fixing* changes is as high as that of general changes and even higher in small change sizes, within-project repetitiveness of fixing changes is low. This implies that program repair tools should not rely solely on the changes in a single project, but rather make use of repeated bug fixes across different projects.
4) To learn the recommending capability of repetitive changes/fixes, we conducted an experiment in which we wrote a simple tool to recommend different options of changes/fixes for a given code fragment based on the collected repetitive changes/fixes. We found that accuracy for top-1 fixing recommendation is over 20%, top-3 is nearly 25%. The corresponding numbers for general changes are 30% and 35%. This result shows a promising future for more sophisticated learning approaches to the aforementioned software maintenance problems.

Sections II-IV present our data collection and experimental procedures. Sections V-VII present the results and our analysis. Section VIII is the related work. Conclusions appear last.

II. Concepts

A. Illustration Example

Let us start with an illustration example on code change and repetitiveness. Figure 1 shows two changes on two if statements. They are considered as fine-grained changes because they occur within individual methods. Both of them include a replacement of a literal (1 or 10) by a variable (b or y) and an addition of an else branch. The variables and literals in the pairs a and x, b and y, 1 and 10 have the same roles. That is, if we replace a, b, and 1 with x, y, and 10 respectively, we can derive the second change from the first. Therefore, we consider the second change repeat the first (or vice versa).

In this paper, we aim to study the characteristics of such repeated changes, for example, how often they occur, how large they are, what are the popular types, etc. In the next section, we will formally define the concepts such as code changes and repeated code changes.

	Source fragment	Target fragment

Change 1	**if** (a >= b)
	  a = a − 1;
		if (a > b)
		  a = a − b;
		else
		  break;

Change 2	**if** (x >= y)
	  x = x − 10;
		if (x > y)
		  x = x − y;
		else
		  break;

Fig. 1. An example of code change

B. Code and Code Change Representation

As writing and modifying code, developers would think of code in terms of programming constructs such as functions, statements, or expressions rather than lines of code or sequences of lexical tokens. For example, in the above illustration example, one would think about the code (before change) as an *if statement*, and modify it by replacing an *operand* in an *infix expression* by another, and adding an *else branch*.

To address this phenomenon, in this study, we model source code and code change in terms of program constructs rather than the lower levels of representation such as code tokens or lines of code. In a program language, a programming construct is often defined as a syntactic unit and represented as a subtree in an Abstract Syntax Tree (AST). For example, an if statement is represented as an AST's subtree, in which the root node specifies its type (i.e. if statement), and the children nodes represent its sub-constructs, i.e. an expression for the predicate, and two code blocks for two branches.

Definition 1 (Code Fragment): A code fragment in a source file is a (syntactically correct) programming construct and is represented as a subtree in the abstract syntax tree of the file.

We consider a code change as a replacement of a code fragment by a different code fragment. Since a code fragment is modeled via an AST, we formulate code change as follows:

Definition 2 (Code Change): A code change is represented as a pair of ASTs (s, t) *where* s *and* t *are not label-isomorphic.*

Since AST are labeled trees, the condition of *not being label-isomorphic* is needed to specify that the code fragments before and after change are different. In this definition, s or t is called *source* or *target* tree, respectively. Either of them (but not both) could be a null tree. s or t is a null tree when the change is an addition or deletion of code, respectively.

To check two code changes for repetitiveness, we could match their source and target trees correspondingly. However, as seen in the illustration example, repeated changes might have different variable names or literal values. Therefore, we need to perform normalization to remove those differences before matching. An AST tree t is normalized by re-labeling the nodes for local variables and literals. For a node for a local variable, its new label is the node type (i.e. ID) concatenating with the name for that variable via alpha-renaming. For a node for a literal, its new label is the node type (i.e. LIT) concatenating with its data type.

Figure 2 shows the AST trees for the code changes in the illustration example after normalization. As seen, nodes for

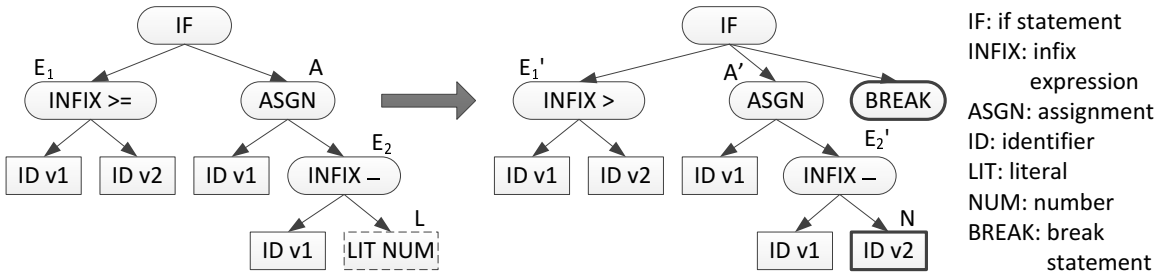

Fig. 2. Tree-based Representation of Code Changes

variables a and x are re-labeled as ID v1 while the ones for b and y have the label of ID v2, since v1 and v2 are the respective names for them after alpha renaming. The nodes for literals 1 and 10 have the same label LIT NUM. Thus, after normalization, two changes have the same tree-based representation. Using normalization, we define repeated code changes as follows:

Definition 3 (Repeat Code Change): A code change (s, t) is a repeated change of another one (s', t') when s' and s, and t' and t are label-isomorphic after normalization.

We want to study the repetitiveness of changes in a project in both scenarios: within its history, and across the histories of different projects. Therefore, we define:

Definition 4: A change in a project P is a repeated change within a project *if it is a repeated change of another one occurring in an earlier revision of P. It is a* cross-project repeated change *if it is a repeated change of another in other project(s).*

Since we want to study the repetitiveness of code changes on types and sizes, we need to define them. We use the AST type of the source tree as the type of the code changes since we want to learn what types of code fragments that are frequently changed. For size, in literature, size of a tree is often defined as its number of nodes. However, for source code, the number of nodes of ASTs highly vary. For example, a method call might have no children (e.g. no parameter) or many children (e.g. many parameters). (In our experiment, some trees might have thousands of nodes). In contrast, tree height (i.e. the number of nodes along the longest path from the root node to a leaf node) varies less (often from 1 to 10). Thus, we choose tree height as a measurement of change size. We define type and size as the following.

Definition 5 (Change Type and Size): Type and size of a code change (s, t) are AST type and the height of s (or of t if s is a null tree), respectively.

For example, in the illustration example, two code changes have type of if (i.e. changes to if statements). Their size is 4.

III. Research Questions and Methodology

A. Research Questions

In this study, we are interested in studying the popularity and potential usefulness of repeated code changes and fixes. Therefore, the first research question we want to answer is

R1. How repetitive code changes and bug fixes are in software evolution?

TABLE I
Collected projects and changes

Projects	2,841
Total source files	16 millions
Total LOCs	1.7 billions
Total revisions	3.6 millions
Revisions having code changes	1.8 millions
Revisions having fixing changes	0.4 millions
Total changed files	6.2 millions
Total LOCs of changed files	2.5 billions
Total changed methods	8.6 millions
Total AST nodes of changed methods	1.3 billions
Total changed AST nodes	89 millions
Total detected changes	213 millions

We are interested in repeated code changes in different dimensions. First, we want to know how large they are (i.e. size of change) and what kind of program constructs that they often occur on (i.e. type of change). Such information will help designers of development tools use repeated changes to focus more on the sizes and types of changes that most likely repeat. In addition, whether changes repeat in within and cross-project settings is also important. If they repeat frequently in the within-project setting, then historical changes/fixes of a project will be a useful source for predict and recommend future changes/fixes of that project. If they repeat frequently in the cross-project setting, then we can learn changes/fixes from other projects to use for a project, especially when it is newly developed. Lastly, we want to study whether the respectiveness of fixing changes, an important type of changes, is different from that of general changes.

R2. How useful repeated and previously seen changes and bug fixes are?

We are interested in the potential use of repeated changes and fixes to recommend changes and fixes for a project in its development, maintenance, and evolution. We expect that, if repeated changes and fixes are popular, a tool could learn from frequently repeated changes and fixes for recommendation.

B. Data Collection

To answer those questions, we have collected a large dataset of code changes. First, we downloaded from sourceforge.net, a hosting service for open-source projects, the development history of all projects written in Java and using SVN for

version control. We focused on only Java and SVN to reduce engineering effort and simplify the classification of change types (e.g. we do not have to define common AST representation for different languages). Future research could include other languages and version control systems. We filtered out the projects with very short development histories, i.e. projects with less than 100 revisions are discarded.

Table I summaries our final dataset. It contains 2,841 projects which at their last snapshots have in total 16 millions of source files and 1.7 billion non-blank, non-commented lines of code. The projects cover variety of domains and topics, and have been written by thousands of developers. We downloaded their repositories to our server for faster processing.

In term of changes, the projects in our dataset have in total 3.6 million revisions, among them, 1.8 million revisions having code changes and 0.4 millions having fixing changes. To detect fixing changes, we used the popular key-word based approaches [35], in which if the commit log message of a revision has the keywords indicating fixing activities, the code changes in that revision are considered as fixing changes.

We processed all 3.6 million revisions and parsed in total 6.2 million changed source files with the total size of 2.5 billion lines of code. Our change detection algorithm detected 8.6 million changed methods with the total size of 1.3 billion AST nodes. From those methods, it detected 213 million fine-grained code changes made from 89 million changed AST nodes. The processing time was 90 hours.

C. Methodology Overview

In this section, we describe our process to collect code changes/fixes from the corpus to build our change database, search for repeated changes/fixes, and compute their repetitiveness. This process composes of three steps and is applied to each revision of every project in the corpus.

1) Detecting all code changes for each revision. Since we focus on the fine-grained changes, we collect only changes within the bodies of changed methods.
2) Updating detected changes to our database. The database is globally accessed for all projects to improve the performance in the study of cross-project repeated changes.
3) Computing the repetitiveness for all changes in both within- and cross-project settings for different dimensions: size, type, and fix/non-fixing.

Let us explain in detail these steps in the next sections.

D. Detecting Code Changes

1) Coarse-grained Differencing: The purpose of this step is to map methods before and after a commit. We use our origin analysis tool (OAT) [28] for this step. For each revision, given as set of changed files provided by the version control system, OAT identifies the mapping for each class/method before and after the change. We extend OAT to support also mapping of classes' instance/static initializers, and treat them similarly as methods. The un-mapped methods and initializers are discarded. All mapped ones are used for fine-grained differencing in the next step.

2) Fine-grained Differencing: To derive those fine-grained changes within the body of each changed method, we use our prior AST differencing algorithm [27]. Given a pair of methods before and after the change, the algorithm parses them into ASTs and finds the mapping between all the nodes between two trees. The key idea of this algorithm is that it maps two nodes based on their node types and the structural similarity between the two sub-trees rooted at them. The unmapped nodes are considered as deleted in the old tree or added in the new tree. Along with the mapping, the algorithm also provides the information if the mapped nodes have change in their labels or in their descendants.

For example, in Figure 2, the algorithm detects that the literal node L is deleted under the infix expression node E_2 and the identifier node N is added under E_2'. The node E_2, in turn, is mapped with E_2' with the same label and has change in its children nodes. Similarly, the top if statements and the assignments A are mapped with changes in descendent nodes. It can also identify that E_1's operator is modified and a break statement is added as the else branch of the if statement.

3) Collecting Code Changes: For each pair of trees T and T' of a changed method, we aim to collect all changes with different heights (sizes). Our tool traverses them in pre-order from the roots to get the changes. If a node n in T is mapped to a node n' in T' and they change in either labels or children nodes, a code change represented by a pair of trees $(T(n), T'(n'))$ is extracted, where $T(n)$ and $T'(n')$ are the trees rooted at n and n', respectively. If a node n in T does not have any mapped nodes in T', a change of $(T(n), null)$ is extracted. Similarly, if a node n' in T' is un-mapped, a change of $(null, T'(n'))$ is extracted. Note that, if a tree is deleted or added, all of its sub-trees will also be collected into the change database because the changes of the sub-trees constitute to the changes of that root tree. During collecting the changes, the parent-child relation between trees are also recorded. This information will be used in recommending changes.

Figure 3 shows all collected changes with different heights (sizes) from 1-3 for the illustration example in Figure 1. The change of height 4 is shown in Figure 2. Note that, one change of small size can be included in a larger one. We analyze change repetitiveness when the code fragment size increases.

E. Building Change Database and Computing Repetitiveness

1) Design Strategies: We design our data structure and algorithm with the key idea that a change and its repeated one have the same type and size, and the same pair of source and target ASTs after normalization. If we create a hashcode for each change by concatenating the hashcodes of its normalized source and target trees, repeated changes will have the same hashcode. Thus, if the changes are grouped based on hashcodes computed via that scheme, repeated changes will be hashed to the same group, which have the same size and type. We used those groups to compute the number of repeated changes by size and by type.

Based on that idea, we extracted the changes in our dataset into a change database. This database is a dictionary of change

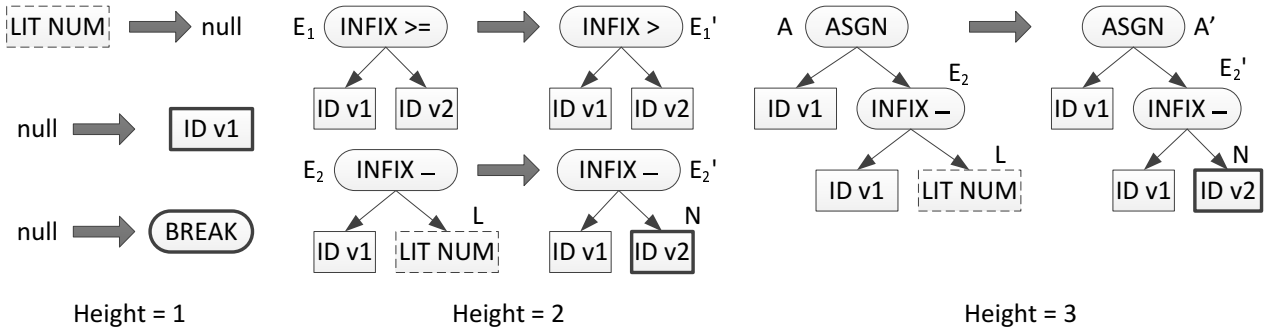

Fig. 3. Extracted Code Changes for Different Heights for the Example in Figure 2

```
1  function BuildDatabase(ProjectList L, ChangeDatabase D)
2    foreach project p in L
3      foreach revision r in RevisionList(p)
4        foreach change c ∈ ChangeList(r)
5          h = HashCode(c)
6          if D not contain h
7            D[h] = new ChangeGroup(c)
8          D[h].Count[p]++
9  end
10
11 function Compute(ChangeDatabase D)
12   foreach group c in D
13     h = HashCode(c), s = Size(c)
14     foreach project p in D[h].Count
15       N[p, s]+ = D[h].Count[p]
16       Nw[p, s]+ = D[h].Count[p] − 1
17       if (D[h].Count.size > 1)
18         Nc[p, s]+ = D[h].Count[p]
19   foreach project p and size s
20     Rw[p, s] = Nw[p, s]/N[p, s]
21     Rc[p, s] = Nc[p, s]/N[p, s]
22 end
```

Fig. 4. Algorithm for Extracting and Computing Repetitiveness

groups indexed by hashcodes computed by aforementioned method. Each change group contains a hash table to map a project's id to the number of changes having the same hashcode in that project. This hash table is used to compute the within and cross-project repetitiveness. That is, if a project p has a count n_p, then p will have $n_p - 1$ changes repeated within p. If the hash table has another project, then all n_p changes of p are counted toward cross-project repetitiveness.

2) Detailed Algorithm: Figure 4 lists the algorithm for building the change database (function BuildDatabase, lines 1-9) and computing the repetitiveness (function Compute, lines 11-22). To build the change database, the algorithm processes each change c in each project p. First, it computes the hashcode for c (line 5). If the database does not have a change group for that hash code, a new change group is created for it (lines 6-7). Then, the count value for p is updated (line 8).

Function Compute (line 11) computes repetitiveness in size. N[p,s] is the total number of changes of size s in project p. Nw[p,s] and Nc[p,s] are the numbers of changes repeated within and across projects, respectively. They are updated using the above idea (lines 15-18). After they are computed, within and cross-project repetitiveness for p at size s, Rw[p,s] and Rc[p,s], are computed as the ratios of Nw[p,s] and Nc[p,s] over N[p,s], respectively. Computation for type is similar (not shown).

IV. Analysis Results

A. Boxplot Representation of Change and Fix Repetitiveness

Figure 5 shows repetitiveness results of general and fixing changes in both within- and cross-project settings. For each change size s from 1-10, we computed the repetitiveness $R(s)$ for all corresponding changes of every project. Thus, for each size s, we have a distribution of 2,841 projects as data points. This distribution is plotted as a box plot, with five quartiles: 5% (the lower whisker), 25% (the lower edge of the box), 50% (the middle line), 75% (the upper edge of the box), and 95% (the upper whisker). There are 10 box plots for 10 sizes. Let us explain the leftmost boxplot in Figure 5 for the within-project repetitiveness of general changes of size 1.

1) The 50% quartile, i.e. median, is at 72%. Since the median could be seen as the center of the distribution, one could say that on average, the projects in our dataset have 72% of their size-1 changes repeated within individual project.

2) The 25% quartile is at 62%, implying that more than 75% of the projects have at least 62% those changes repeated within a project.

3) The 75% quartile is at 80%, meaning that at least 25% of projects have those changes repeated more than 80%.

4) The 95% quartile is 94%, suggesting that, at least 5% of total projects have 94% of their size-1 changes repeated within a project.

5) The inter-quartile (difference between 25% and 75% lines) is 18%, referring to the spread of the distribution.

B. Exponential Relationship of Repetitiveness and Size

Comparing the box plots for different change sizes in both within and cross-project settings, we see that repetitiveness is very high for small changes, but it significantly decreases when the change size increases, as expected. For example, in the cross-project setting, size-2 changes have median repetitiveness of more than 60%, but that for size-6 changes drops below 10%. The repetitiveness of larger changes (size of 7-10) is very small (less than 2% on average).

We modeled $R(s)$ and s with several classes of simple curves, and found that the exponential curve $R(s) = \alpha e^{\beta s}$ represents their relationship the best. We used the least square method to compute two parameters α and β for every project.

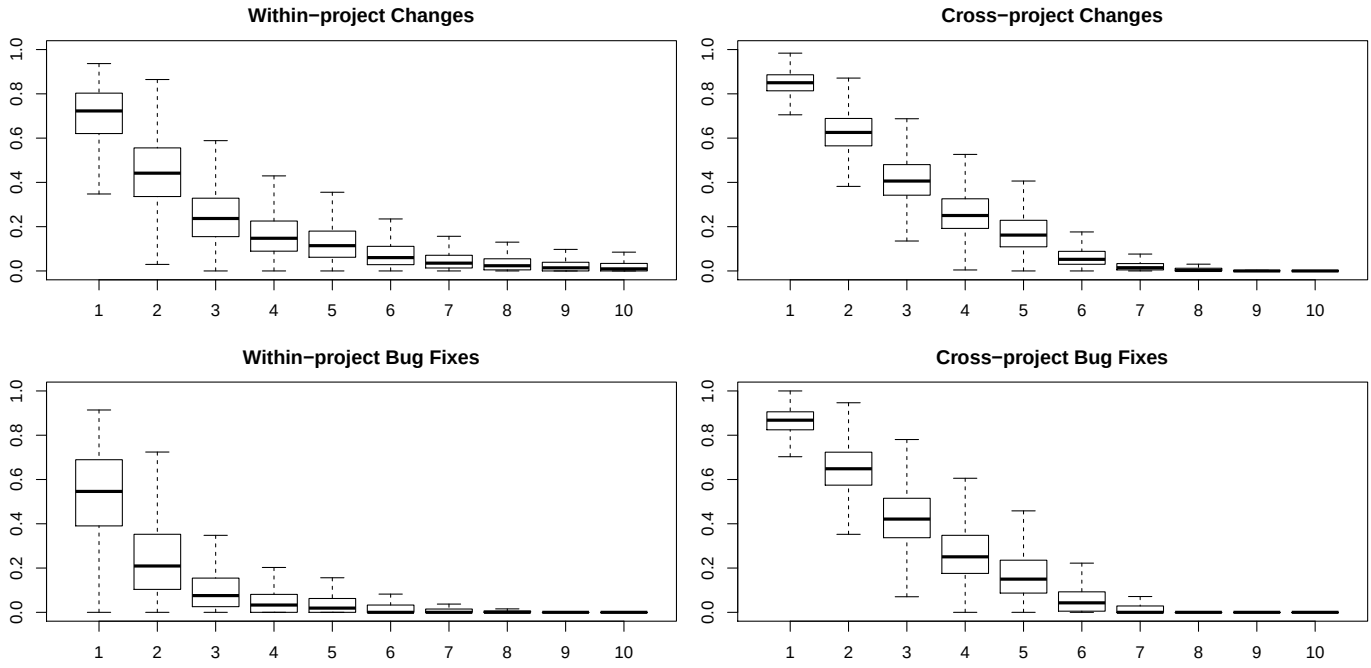

Fig. 5. Repetitiveness of Code Changes and Fixes over Change Size for all 0,000 Projects in the Corpus

TABLE II
R^2 OF FITTED EXPONENTIAL CURVE TO REPETITIVENESS

Setting	Change	Median	≥ 0.90
Within-Project	General	0.99	93%
	Fixing	0.99	90%
Cross-Project	General	0.98	96%
	Fixing	0.97	88%

The goodness of fit is measured by the coefficient of determination R^2. The closer R^2 is to 1, the better the fit is.

Table II summarizes the goodness of fit. As seen, it is very high. For example, for general changes in the within-project setting, median R^2 is 0.99, and 93% of projects have R^2 of at least 0.90. Results in the cross-project setting are similar. The median R^2 is 0.99, and 96% of projects have R^2 of at least 0.90. This high level of fit for most of projects in the dataset implies that $R(s)$ has a strong exponential relationship to s. That is, **repetitiveness of code changes decreases exponentially when change size increases**.

As an implication, the automatic program repair or library update tools should focus on the change fragments with the syntactic units of the height from 2-6 to reduce the search spaces of solutions (size-1 changes are on literals/variables).

C. Within and Cross-project Repetitiveness Comparison

As seen in Figure 5, the box plots for the sizes from 1-5 in the cross-project setting are higher than those in the within-project setting. For example, for size-1 changes, the median cross-project repetitiveness is 85%, while the within-project one is 72%. For size-2 changes, the corresponding numbers are 63% and 44%.

To statistically verify this observation, we use the paired Wilcoxon test to compare the distributions of $R(s)$ in the within-project and cross-project settings. All the tests for sizes from 1 to 5 infer that **cross-project repetitiveness is statistically higher than within-project repetitiveness**. For large sizes, changes repeat about the same or slightly less frequently in the cross-project setting.

For all sizes, the inter-quartiles of box plots in the cross-project setting is always shorter than those in the within-project one. For example, the inter-quartile for cross-project repetitiveness with size-1 changes is 7%, while that in the within-project setting is 18%. However, at the size 5, the difference is insignificant, with the corresponding numbers of 11.97% and 11.82%. Nevertheless, that result implies that **repetitiveness in cross-project setting is more stable**. Thus, repeated changes are more likely to be found across projects.

D. Repetitiveness of Bug Fixes

As seen in Figures 5 and Table II, repetitiveness of fixing changes is similarly to that of general changes. That is, at smaller sizes (s from 1 to 2), bug fixes repeat frequently, with repetitiveness usually higher than 60% in the cross-project setting. At larger sizes (s from 6-10), fixing changes repeat less frequently, with repetitiveness often less than 10%. Thus, *the automatic program repair methods should focus on the change fragments with the small sizes from 2-5*.

Importantly, we conducted a paired Wilcoxon test and found

that at smaller sizes (from 1 to 5), **cross-project repetitiveness of fixing changes is statistically higher than that in the within-project setting**. As an example, the median of cross-project repetitiveness for size-2 fixing changes is 65% in comparison with 21% in the within-project setting. The corresponding numbers for size-3 fixing changes are 42% and 8%. As seen, within-project repetitiveness of fixing changes is low. Those results suggest that *automatic patching and program repairing tools should not rely solely on the changes in an individual project, but rather make use of repeated bug fixes across different projects*.

In Figure 5, repetitiveness for cross-project changes is comparable to that for cross-project bug fixes. However, our paired Wilcoxon test results showed that at the **small sizes (1-3), repetitiveness of fixing changes is statistically higher than that of general changes**. This suggests that the bug fixes tend to be at small sizes. Thus, automatic patching tools could start with small changes and gradually compose them.

E. Repetitiveness on Representative Projects

While previous sections present the results on the analysis on all 0,000 projects in our dataset, this section presents the results for a small set of the representative projects for further detailed analysis. Figure 6 plots the cross-project repetitiveness values of general changes (in solid lines) and fixing changes (in dashed lines) for those projects. As seen, although following the same trends, the curve for one project might look different from another. For example, the curve for general changes in jedit is higher than that of jitterbit (they have similar α parameters, however, β for jedit is larger than that for jitterbit). Figure 6 also illustrates that at smaller sizes, some projects have the repetitiveness of fixing changes higher than that of general changes, such as jitterbit or springframework.

Figure 7 plots for the same projects in the within-project setting. As seen, the repetitiveness of fixing changes is lower than that of general changes. In some projects such as jquant or pulse-java, the difference is quite significant.

F. Repetitiveness and Change Type

1) Change Type: We perform another analysis for the repetitiveness of changes classified based on the types of the corresponding code structures. Given a change as a pair of AST sub-tree (s, t), its type is defined as the AST node type of s. For example, if s is a sub-tree for an if statement, that change is classified as a change to an if statement.

The repetitiveness of a change type is computed as the ratio of the number of repeated changes of that type over the total number of changes of that type in all projects (we did not compute separately for each project). From the previous results, we focused on the repetitiveness in the cross-project setting. In addition to general changes, we also computed the repetitiveness of fixing changes.

We choose 30 most popular AST node types and divide them in 4 groups. The Array group contains nodes representing program elements related to arrays, such as an array access or array declaration. The Call group contains nodes representing the elements related method/constructor calls and field accesses. The Expression group is for expressions. The Statement group contains all statements such as if, while, try, throw, etc.

TABLE III
CROSS-PROJECT REPETITIVENESS AND CHANGE TYPE

Group	Type	General changes Total	General changes Repeat	Fixing changes Total	Fixing changes Repeat
Array	array declaration	321670	0.82	51522	0.82
	array access	888224	0.65	145001	0.66
	array initializer	201187	0.61	31181	0.61
	array creation	232462	0.56	37669	0.58
Call	super constructor	80615	0.72	10775	0.71
	constructor	36504	0.56	5537	0.58
	super method	64037	0.45	11170	0.47
	class instantiation	3425828	0.43	547945	0.43
	field access	1099426	0.42	176047	0.42
	method	23088645	0.40	4117854	0.42
	super field access	5430	0.18	969	0.21
Expression	postfix	333142	0.92	59119	0.90
	prefix	766036	0.61	159937	0.60
	infix	5875878	0.53	1201842	0.54
	instance of	223488	0.34	49456	0.37
	cast	1072131	0.33	195376	0.35
	conditional	202497	0.22	41347	0.23
Statement	case	357582	0.62	51953	0.60
	throw	377629	0.55	95576	0.52
	assert	38546	0.38	8144	0.39
	catch	535793	0.37	131942	0.32
	if	4454870	0.10	927589	0.11
	while	198858	0.06	39759	0.06
	try	786262	0.05	172699	0.06
	for	417898	0.05	74852	0.05
	synchronized	51432	0.05	12087	0.04
	enhanced for	373466	0.04	68926	0.05
	initializer block	11082	0.03	1899	0.02
	switch	123181	0.02	21391	0.03
	do while	11735	0.01	2460	0.01

2) Repetitiveness: Table III lists the total number and repetitiveness of changes computed for those types. At a first glance, the number of changes is different for those types. For example, **method calls, infix expressions, and if statements have the most changes**, while **changes to constructor calls, super field accesses, and do statements are less**.

The repetitiveness for changes also vary according to change types. It is very high for changes related to arrays, expressions, and calls (often 40-80%), while it is very low for common statements such as if or while (often no more than 10%). It is interesting that **changes to method calls are the most popular and frequently repeated (40% repetitiveness)**, while **changes to if statements are also popular but repeat much less frequently (just 10% repetitiveness)**. Change size is a possible explanation for this observation. Array accesses and method calls (especially super calls) are structurally simpler than the compound statements (e.g. if or while), thus they could repeat more. For example, 92% of changes to array accesses have sizes of 1-3, while only 3% of changes to if statements have such small sizes. In addition, among statements, the small ones such as case, throw, and catch statements also repeat more frequently than the larger ones (37-62%).

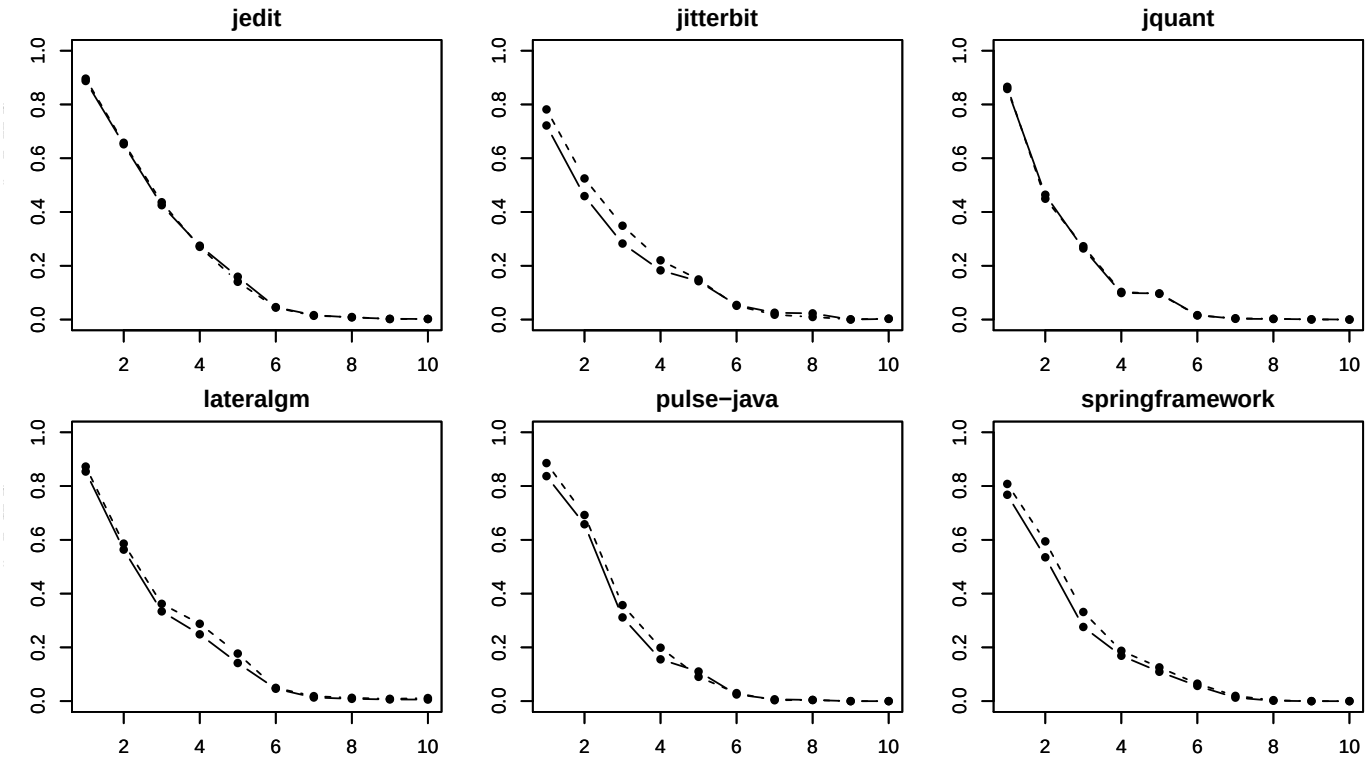

Fig. 6. Cross-project Repetitiveness of General (solid line) and Fixing Changes (dashed line)

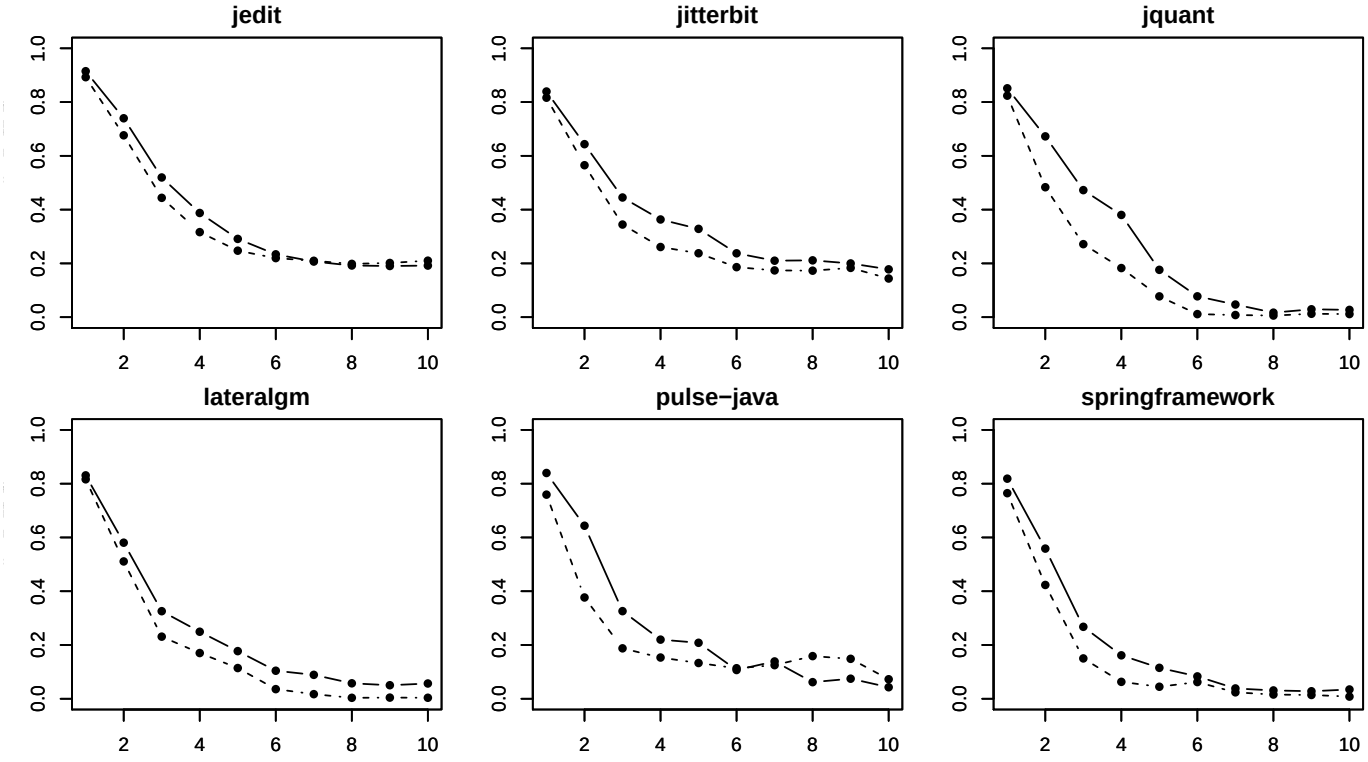

Fig. 7. Within-project Repetitiveness of General (solid line) and Fixing Changes (dashed line)

```
1  function Recommend(Tree s, ChangeDatabase D)
2    List T
3    Changes C = D.GetChangesWithSourceTree(s)
4    foreach c = (s, r) ∈ C
5      bestScore = ComputeScore(s, r, D)
6      A_s = GetKLevelAncestors(s)
7      A_r = GetKLevelAncestors(r)
8      foreach (p, q) ∈ A_s × A_t
9        score = ComputeScore(p, q, D)
10       if score > bestScore bestScore = score
11     T.AddAndSortByScore((r, bestScore))
12     return T
13 end
14
15 function ComputeScore(Tree s, Tree r, ChangeDatabase D)
16   N_s = number of occurences of changes having s as source tree
17   N_{s,r} = number of occurences of change (s, r)
18   return N_{s,r}/(1 + N_s)
19 end
```

Fig. 8. Algorithm for Recommending Changes

Importantly, as seen in Table III, cross-project repetitiveness of bug fixes is high, especially for small program constructs. It is as high as in general changes and much higher than the fixing changes in the within-project setting. This result on change repetitiveness over change size and type suggests that the aforementioned automated program repair should focus on the fixes with small sizes and of highly repetitive types such as array access, method calls, and if/case statements.

V. Change Recommendation

A. Experiment Setting

To learn the recommending capability of repetitive changes and fixes, we conducted an experiment in which we wrote a simple tool to recommend different options of changes/fixes for a given code fragment based on the collected repetitive changes/fixes. For each project, we ran the tool on all the changes in a chronological order. Such a change is represented as a pair of trees (s, t), where s is for the original code fragment (source) and t is for the replacing one (target). The tool takes s and returns a ranked list $T(s)$ of k recommendations. If t matches any result in that list, we count this as a hit, i.e. a correct recommendation. The overall accuracy is defined as the ratio between the number of hits over the total number of recommended cases. The process is repeated for different values of k. The process was first run for within-project mode. That is, the recommendation list $T(s)$ is collected only from the previous changes of the project under processing. Then, we ran it in hybrid mode, where the recommendation list $T(s)$ is also collected from the changes of all other projects. In addition to running for general changes, we also performed the same procedure for fixing changes.

Figure 8 shows the pseudo-code of the algorithm for recommending the target tree. Given a source tree s, it looks for all existing changes (s, r) (line 3). The score of a target r is computed as the ratio between the number of occurrences of change from s to r and the number of occurrences of all changes having s as the source tree (lines 16-18). This score means the confidence of choosing r as the target of source s among all other seen targets. We use 1 as the smoothing factor for the cases where the numbers of occurrences are too small. For better results, the algorithm considers the surrounding code of s and r when ranking candidate targets. The idea is that if the enclosing fragment of r is also the target of an enclosing fragment of s with high confidence, r might have a better chance to be the right target. Thus, it also considers the scores among their ancestors. The recommendation score for r is defined as the maximum score among the score between s and r and the scores between any pair of their ancestors. This score is used to rank the candidate targets in the list. Note that the counting of N_s and $N_{s,r}$ in function ComputeScore can be adjusted to fit with the settings. In the within-project setting, only the occurrences seen in the same project and before the revision of the change (s, t) are counted. In the hybrid setting, in addition to the occurrences as in the within-project setting, all the ones seen in other projects are also counted.

B. Recommendation Result in Within-project Setting

Figure 9 shows the accuracy of recommending changes and fixes in the within-project setting. We executed the recommendation tool for 100 largest projects with different values of k (the number of recommendations): 1, 2, 3, 4, 5, 10, 15, and 20. The accuracy for each k is shown as a box plot. As seen, the top-1 accuracy is around 10% to 30%, with the median value of around 20%. At $k = 3$, the accuracy can be up to 40% (and the median is around 25%). After that, accuracy is stable, i.e. does not improve much with more recommendations. The accuracy for recommending fixing changes is lower than that for general changes. The median accuracy is around 10%, even though with more recommendations. Given the lower within-project repetitiveness of fixing changes, this result is expected.

C. Recommendation Result with Hybrid Approach

Hybrid approach combines both within- and cross-project repeated changes. As seen in Figure 9, for general changes, the median top-1 accuracy is now around 30%. At $k = 3$, the accuracy can be up to 55% (and the median is around 35%). The median accuracy for fixing changes also increases to 25%. This result shows that fix recommendation tools could benefit much from cross-project fix repetitiveness.

Threats to validity. Although, our dataset contains a large number of projects, all of them are developed on Java. Thus, the observations on the repetitiveness of changes over change size and type might not be generalizable for projects developed in other languages or paradigms. In addition, all subjects are open-source software, thus, their repetitiveness characteristics, especially in the cross-project setting, might not be the same for commercial software.

Another threat is on the accuracy of recommended changes and fixes. We currently compare the recommended changes/-fixes and actual ones by their trees *after* normalization for literals and local variables. In other words, that accuracy result is for template recommendation, rather than that of concrete variable names and literal values. However, we expect that in the concrete application of change/fix recommendation, a tool must concretize the literal values and local variables' names.

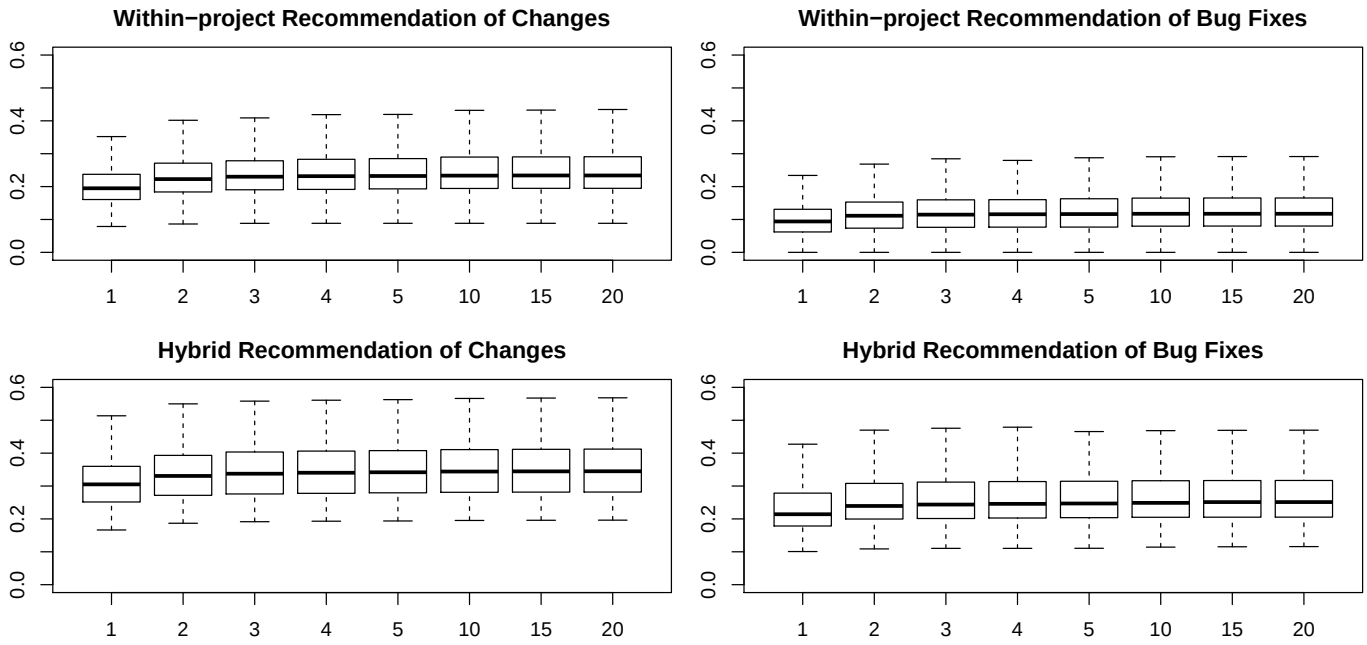

Fig. 9. Accuracy of changes and fixes recommendation

VI. Related Work

Our study is related to the large-scale study by Gabel and Su [8] on the uniqueness of source code on more than 420 million LOCs in 6,000 software projects. They consider a source file as a sequence of syntactical tokens with the abstraction on variables' names. They reported *syntactic redundancy* at levels of granularity from 6-40 tokens. At the level of granularity with 6 tokens, 50-100% of each project is redundant. Later, in a study about 20 projects, Hindle *et al.* [12] have used n-gram model to show that source code has high repetitiveness, and n-gram model has good predictability and could be useful in code suggestion. Another large-scale study on code reuse at the *file* level was from Mockus [24], [25] on 13.2 millions source files in continually-growing 38.7 thousand unique projects. They reported that more than 50% of the files were used in multiple projects. Jiang and Su [14] locate functionally equivalent code fragments based on testing. The method could be used to study source code reuse at the functional level.

There are advanced approaches in automatically generating/synthesizing the program fixes based on the previously seen fixes in the projects' histories [17]. Weimer *et al.* [11] proposed GenProg, a patch generation method that is based on genetic programming. Kim *et al.* [17] introduced PAR, an automatic pattern-based program repair method, that learns common patterns from prior human-written patches. Our study provides empirical evidences for such automatic patch generation approaches. Our prediction model could serve as the baseline to enhance those approaches. Our prior study in FixWizard [29] and a study by Kim *et al.* [19] have confirmed the recurring nature of fixes. However, they were conducted in a much smaller scale with less than ten projects.

There are a large body of research and tools on clone detection, which is concerned with the detection of copy-and-paste fragments of code [4], [30]. Generally, they can be classified based on their code representations. The typical categories are text-based [6], [22], token-based [2], [15], [21], [23], tree-based [3], [13], [7], and graph-based [20]. Most clone detection tools focus on individual projects, rather than across projects. There have been several studies on software changes [32], non-essential changes [16], change-based bug prediction [31], [9], code clone changes [18], cloning across projects [1], patch identification [33], threats when using version histories to study software evolution [26], etc. Giger *et al.* [10] proposed an approach to predict type of changes such as condition changes, interface modifications, inserts or deletions of methods and attributes, or other kinds of statement changes. They use the types and code churn for bug prediction [9]. Our prediction study does not have different types of changes, but focuses more exact fine-grained changes.

VII. Conclusions

In this paper, we present a study of repetitiveness of code changes in software evolution. Repetitiveness is defined as the ratio of repeated changes over total changes. We model a change as a pair of old and new AST sub-trees within a method. First, we found that repetitiveness of changes could be very high at small sizes and decreases exponentially as size increases. Second, repetitiveness is higher and more stable in cross-project setting than in within-project one. Third, fixing changes repeat similarly to general changes. Importantly, learning code changes and recommending them in software evolution is beneficial with accuracy for top-1 recommendation of over 30% and top-3 of nearly 35%.

REFERENCES

[1] R. Al-Ekram, C. Kapser, R. C. Holt, and M. W. Godfrey. Cloning by accident: an empirical study of source code cloning across software systems. In *ISESE*, pages 376–385, 2005.

[2] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. Comput.*, 26(5):1343–1362, 1997.

[3] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 368. IEEE Computer Society, 1998.

[4] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE TSE*, 33(9):577–591, 2007.

[5] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 481–490. ACM, 2008.

[6] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99. IEEE CS, 1999.

[7] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, Nov. 2007.

[8] M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 147–156. ACM, 2010.

[9] E. Giger, M. Pinzger, and H. Gall. Comparing fine-grained source code changes and code churn for bug prediction. In *8th working conference on Mining software repositories*, pages 83–92. ACM, 2011.

[10] E. Giger, M. Pinzger, and H. C. Gall. Can we predict types of code changes? an empirical analysis. In *MSR*, pages 217–226. IEEE CS, 2012.

[11] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.

[12] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 837–847. IEEE Press, 2012.

[13] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 96–105. IEEE CS, 2007.

[14] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 81–92. ACM, 2009.

[15] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, July 2002.

[16] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *ICSE*, pages 351–360, 2011.

[17] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE 2013. IEEE Press (To appear), 2013.

[18] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. *SIGSOFT Softw. Eng. Notes*, 30(5):187–196, 2005.

[19] S. Kim, K. Pan, and E. E. J. Whitehead, Jr. Memories of bug fixes. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 35–45. ACM, 2006.

[20] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS '01: Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56. Springer-Verlag, 2001.

[21] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Softw. Eng.*, 32(3):176–192, Mar. 2006.

[22] A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01. IEEE CS, 2001.

[23] T. Mende, R. Koschke, and F. Beckwermert. An evaluation of code similarity identification for the grow-and-prune model. *J. Softw. Maint. Evol.*, 21(2):143–169, Mar. 2009.

[24] A. Mockus. Large-scale code reuse in open source software. In *Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*, FLOSS '07. IEEE CS, 2007.

[25] A. Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pages 11–20. IEEE CS, 2009.

[26] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig. Is it dangerous to use version control histories to study source code evolution? In *ECOOP*, pages 79–103, 2012.

[27] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Clone management for evolving software. *IEEE Trans. Softw. Eng.*, 38(5):1008–1026, Sept. 2012.

[28] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen. A graph-based approach to api usage adaptation. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 302–321. ACM, 2010.

[29] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 315–324. ACM, 2010.

[30] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, May 2009.

[31] S. Shivaji, E. J. W. Jr., R. Akella, and S. Kim. Reducing features to improve code change-based bug prediction. *IEEE Trans. Software Eng.*, 39(4):552–569, 2013.

[32] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim. How do software engineers understand code changes?: an exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 51:1–51:11. ACM, 2012.

[33] Y. Tian, J. L. Lawall, and D. Lo. Identifying linux bug fixing patches. In *ICSE*, pages 386–396, 2012.

[34] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE '10, pages 195–204. ACM, 2010.

[35] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07. IEEE CS, 2007.