2016 IEEE International Conference on Software Maintenance and Evolution

Using Topic Model to Suggest Fine-grained Source Code Changes

Hoan Anh Nguyen
Iowa State University
Email: hoan@iastate.edu

Anh Tuan Nguyen
Iowa State University
Email: anhnt@iastate.edu

Tien N. Nguyen
Iowa State University
Email: tien@iastate.edu

***Abstract*—Prior research has shown that source code and its changes are repetitive. Several approaches have leveraged that phenomenon to detect and recommend change/fix patterns. In this paper, we propose TasC, a model that leverages the context of change tasks in development history to suggest fine-grained code change/fix at the program statement level. We use Latent Dirichlet Allocation (LDA) to capture the change task context via co-occurring program elements in the changes in a context. We also propose a novel technique for measuring the similarity of code fragments and code changes using the task context.**

We conducted an empirical evaluation on a large dataset of 88 open-source Java projects containing more than 200 thousand source files and 3.5 million source lines of code in their last revisions with 423 thousand changed methods. Our result shows that TasC relatively improves recommendation accuracy up to 130%–250% in comparison with the base models that do not use task context. Compared with other types of contexts, TasC outperforms the models using structural and co-change contexts.

I. Introduction

Due to the practice of software reuse, source code in software projects often contains code fragments with a certain degree of similarity. That could lead to the repeated/similar changes and bug fixes to source code. Moreover, the repeated/similar programming tasks often require the repeated/similar changes as well. Prior studies have also confirmed that *code changes are repetitive* [1], [16], [17]. The research relevant to repeated changes can be broadly classified into two groups: 1) *detecting change/fix patterns*, and 2) *leveraging change patterns to support software engineering (SE) tasks*. For example, the frequent adapting changes to the changes to a framework or a library's APIs are used to support framework/API adaptation [3], [18]. With detected fine-grained change patterns [16], IDE designers can build automated support for such frequent editing changes. Migration patterns are learned to support code migration between languages [25]. An important application is to suggest the relevant changes/fixes [9], [19] by leveraging the fix patterns detected in the same or other projects.

The approaches to detect change/fix patterns have examined the changes/fixes in their context with relation to the other changes/fixes in the same tasks, commits, or in the change history of a project [10], [16]. However, the existing approaches to leverage change/fix patterns for change and bug-fix suggestion are limited to using such context of changes only at the coarse-grained level. For example, Ying *et al.* [24] and Zimmmermann *et al.* [27] use change history to suggest co-changes only at the *coarse-grained* level of *methods and files*. Change recommendation in an IDE, for example, requires a model to work at the level finer than a method.

In this work, we conjecture that *the changes belonging to the same task/purpose are related and might need to occur together, thus, such changes would be useful in predicting the next change or fix as they likely belong to the same change task*. Let us call such information the **change task context** or **task context** for short. We develop TasC that learns the task contexts from a project's history and leverages them to find a match for the current context in order to suggest the next change/fix at the fine-grained level for a given statement in a program. The task context is modeled via Latent Dirichlet Allocation (LDA) [2] in which *the topics are discovered to model the change tasks*. TasC can be used in an IDE, in which it processes the current changes/fixes with the history of task contexts to recommend a change/fix to the current code at the requested point.

We conducted a large-scale empirical evaluation with a large data set of 88 open-source Java projects, with 3.6 million source lines of code (SLOC) at the latest revisions, 88 thousand code change revisions (20 thousand fixing revisions), 300 thousand changed files, and 116 million changed SLOCs. We extracted consecutive revisions from the code repositories of those projects and built the changes at the abstract syntax tree (AST) level. A change is modeled as a pair of subtrees (s,t) in the ASTs for all statements. The (sub)trees are normalized via alpha-renaming local variables and abstracting literals.

Our empirical result shows that with the task context, TasC is able to make good improvements over the state-of-the-art approaches. Specifically, TasC relatively improves up to 250% top-1 accuracy over an existing model [17] that is based solely on the repeated changes and does not consider any context. We also observed that using tasks derived from recent transactions, i.e., within certain window of commits, achieves better accuracy than using tasks from the entire history. This shows the **temporal locality of tasks/topics for code changes**. We also observed the **spatial locality** of tasks/topics for code changes where using task context within a project works better than using tasks across projects. The suggestion accuracy for bug fixes, a special type of changes, is lower than that for general changes in within-project setting, and higher in the cross-project setting. We also compared TasC with the existing models using the *co-change context*, which consists of fine-grained changes that occur together in the same transactions. Such state-of-

```
1 a)
2   void process (Element e) {
3      e.run();
4   }

1 b)
2   void process ( Set <Element> s ) {
3      foreach (Element e : s)
4        e.run();
5   }
```

Fig. 1. A Change Task: Converting Element to Collection [16]

the-art models include Rolfsnes *et al.*'s association [22], Ying *et al.* [24], and Zimmermann *et al.* [27]. Our result shows that TasC relatively improves up to 130% top-1 accuracy over those models with the co-change context. It also outperforms FixWizard [19] that uses the structural context of the given code fragment that needs to be changed. In FixWizard [19], if two code fragments with the same structural context in term of containing structural units, the change to one fragment is used to suggest the other. Our key contributions include

1. A novel technique for using topic modeling to measure the similarity between code fragments and code changes,
2. TasC, a model to recommend the change/fix for the current code in an IDE, considering the task context, and
3. A large-scale empirical evaluation on TasC's accuracy.

II. Motivating Example

Figure 1 shows an example of a change task in which a developer modifies his/her code from processing a single element to processing a set of elements. This task was reported by Negara *et al.* [16] as *"converting element to collection"*. For this task, the code fragments displayed in boxes are newly edited. Due to the conversion of the type for a single element Element to the Set type, the code for a loop with either for, foreach, or while is likely to occur at line 3 because it could be part of that task. Therefore, if the user requests for a suggestion at line 3, a recommender should give the list of candidates including adding a for, foreach, or while statement.

The elements in the change task (e.g., Set, variable s, foreach, Element, variable e) are said to belong to a **task context**. The idea is that *if the purpose(s)/task(s) of the current edits and those of the recent changes can be discovered, we can leverage such knowledge to predict the next change since a task might require changes that occur together as part of the task*.

We use topic modeling to capture those likely co-occurring changes and recover this hidden information (Section IV-B).

III. Code Change Representation

In this work, we develop TasC, a model to recommend the most likely change to the current source code in an IDE. With TasC, we can build a code change recommender to suggest a list of candidate changes when a user requests for a change suggestion. We could also build an automated bug-fixing engine by requesting TasC for a fixing change to a candidate buggy code fragment. Let us first explain our change representation.

```
1 a)
2   while (tokenSc.hasNext() && n++ < MAX) {
3      ...
4      if (n%10==0)
5         fw.append("\r\n");
6   }

1 b)
2  while (tokenSc.hasNext() &&n++ <= MAX){
3     ...
4     if (n%10==0)
5       fw.append( System.lineSeparator() );
6     else
7        fw.append(" ");
8  }
```

Fig. 2. A Change Task: Processing Tokens with Text Scanner

A. Coarse-grained Change Detection

In this work, we are interested in the changes committed to a repository in the same transaction. Thus, we define a **transaction** *as a collection of the code changes that belong to a commit in a version control repository.*

For each revision of a project, given the code before and after the changes that were checked out from a version control repository, we first need to perform program differencing to identify the changes at the method and class level, i.e., to identify what classes and methods have been changed or not.

To do that, we use our origin analysis tool (OAT) [18]. For each revision, OAT takes as input the set of all changed (added/deleted/modified) files provided by the version control system and computes the mappings between the classes and methods before and after the change.

B. Fine-grained Change Representation

After identifying the methods that are changed, TasC identifies the fine-grained changes within each method.

In TasC, we represent code fragments as subtrees of the Abstract Syntax Tree (AST) of a program. A change is modeled as a pair of subtrees (s,t) in the ASTs for all statements. The (sub)trees are normalized via alpha-renaming the local variables and abstracting the literals.

1) Code Fragment: A **code fragment** in a file is defined as a (syntactically correct) program unit and is represented as a subtree in the AST of the file.

2) Code Change: When a fragment is changed, its AST is changed to another AST representing the new fragment. In TasC, we are interested in changes at the statement level. Thus, we could use a pair of ASTs corresponding to the two statements before and after a code change to reflect the change.

Figure 2 illustrates a change task consisting of multiple fine-grained changes. Comparing the source and target fragments, we can see that 1) the operator '<' is replaced with '<=', 2) the literal string "\r \n" is changed into System.lineSeparator() to support different OSs (since Linux does not use "\r \n"), and 3) the else part is newly added to insert a whitespace after each token. Figure 3 shows the two ASTs representing the source

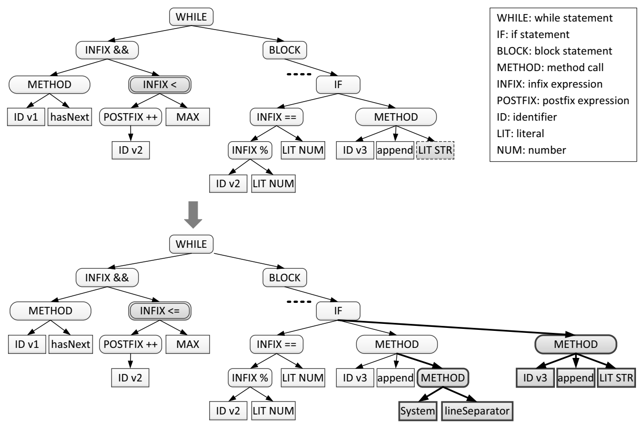


Fig. 3. Tree-based Representation for the Code Change in Fig. 2

and target fragments of the change. For simplicity, in Figure 3, we do not draw the nodes of type ExpressionStatement which are the parents of the method call nodes under the if statements.

Collapsing process. Since some statements can be compound statements, i.e., having other statement(s) in their bodies, when a statement is changed, all containing statements could be automatically considered as changed. For example, a single change to a literal in the code can cause the whole method to be considered as changed. This would lead to a very large number of changes. We avoid this effect by replacing the body statement(s), if any, of compound statements with empty blocks. We call this process *collapsing*. For example, an *if* statement will be represented as an AST which roots at an *if* node, and contains a child sub-tree for its condition expression, a *block* node for its *then branch* and possibly another *block* node for its *else branch*. The tree (b) in Figure 4 shows such an example for the *if* statement represented by the lower tree in Figure 3.

*Definition 1 (***Code Change***): A code change at the statement level is represented as a pair of ASTs (s,t) where s and t are not label-isomorphic. The trees s or t can be a null tree or a tree representing a statement obtained from the original statement by replacing all sub-statements with empty blocks.*

In this definition, s and t are called *source* and *target* trees, 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. Since AST is a labeled tree, the condition of *not being label-isomorphic* is needed to specify that the code fragments before and after change are different.

Alpha-renaming process. Due to naming convention and coding style, the same code fragment when written by different developers and/or in different projects could have different lexical tokens. As a result, changes to them would be considered different. In order to remove those differences, we need to perform normalization. An AST tree t is normalized by re-labeling the nodes for local variables and literals. For a node of a local variable, its new label is the node type (i.e., ID) concatenated with the name for that variable via alpha-renaming. For a literal node, its new label is the node type (i.e., LIT) concatenated with its data type.

Figure 3 shows the subtrees for the code changes in the illustrating example in Figure 2 after normalization. The node for the variable tokenSc is labeled as ID v1 while the one for n is labeled as ID v2 since they are local variables and, thus, alpha-renamed into v1 and v2, respectively. The node for the literal value 10 is labeled as LIT NUM.

C. Fine-grained Code Change Extraction

This step derives the fine-grained changes within the body of each changed method. We use our prior AST differencing algorithm [18]. Given a pair of methods before and after the change, the algorithm parses them into ASTs and finds the mapping between all the nodes of the two trees.

We process all the mapped methods and initializers to extract fine-grained code change as follows. For each pair of trees T and T' of a changed method or initializer before and after the change, we extract all code changes at the statement level as defined earlier. We traverse all statement nodes in the two trees in the pre-order from their root nodes. If we encounter a node marked as unchanged after fine-grained differencing, we skip the whole sub-tree rooted at that node because there will be no change to collect. If we see a changed node, we will first *collapse* the corresponding statement. If a node n in T does not have a mapped node in T', a code change of pair $(S, null)$ is extracted, where S is a collapsed tree of the statement rooted at n. Similarly, if a node n' in T' does not have a mapped node in T, a code change of pair $(null, S')$ is extracted, where S' is a collapsed tree of the statement rooted at n'. If the node n in T is mapped to the node n' in T' and either the collapsed tree S or S' has a change node, a code change of pair (S,S') is extracted. During this process of collecting changes, we also normalize the source and target fragments, with alpha-renaming and literal abstraction, and store their sequences of tokens after normalization. The parent-child relations between code fragments are also recorded. This information will be used in suggesting changes.

Figure 4 shows all collected changes for the illustrating example in Figure 3. The first pair (a) is for the change in the operator '$<$'. The second one (b) is for the modification to the if statement. The statements in its body (then and else branches) are replaced with the empty block statements after collapsing. The third one (c) is for the change from a string literal to a method call. The last one (d) is the addition of the method call append.

Next, let us explain how we model the task context of changes in the change history with topic modeling.

IV. Modeling Task Context with LDA

A. Mapping to Concepts in Topic Modeling

We model task context via LDA topic modeling as follows.

1) A code **change** is considered as a sentence with multiple words involved in the changed fragments. In the context of our change recommendation problem, let us use the term **token**, instead of "word".

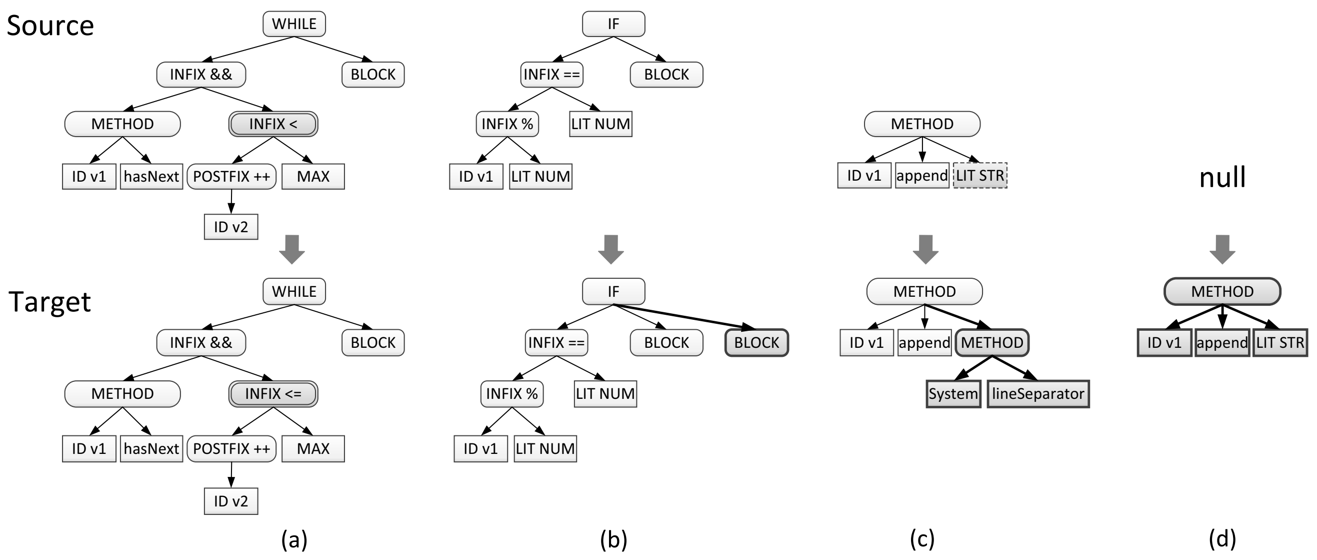


Fig. 4. Extracted Code Changes for the Example in Fig. 3

2) A **transaction** or commit is a collection of changes (sentences), thus, also a collection of tokens, and can be viewed as a *document* in LDA. All tokens are collected in a vocabulary V.

3) A *topic* in LDA is used to model a **task**, which can be seen as the purpose of a change or a set of changes. A task is represented by a set of changes with associated probability for each change. For example, for the task of fixing bug #01, the probability for the change #1 to occur is 25%, while that for the change #2 to occur is 20%, and so on. Since each change is viewed as a sentence with multiple tokens involved in the changed fragments, a task can be represented by a set of such tokens with associated probabilities (see the Tasks in Figure 5).

4) A transaction (*document*) of changes can be for multiple tasks (*topics*). A transaction t has a **task proportion** θ_t to represent the significance of each task in t. Assume that in the entire history, we have K tasks. Then, $\theta_t[k]$ with k in $[1,K]$ represents the proportion of task k in t.

Based on this, if we use topic modeling on the set of transactions in a project, we have the task proportion of all transactions ts, i.e., the proportion of each task in any transaction t.

B. Details on Modeling Task Context

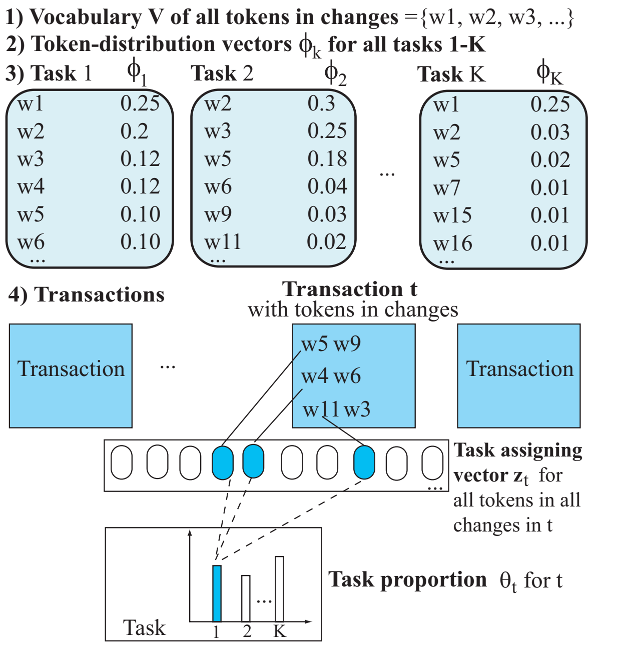
Figure 5 illustrates our modeling.

1) **Vocabulary** V: for each change, we collected all syntactic code tokens in the AST after normalization of the source fragment of the change. If the source is null, i.e., the change is an addition, the target fragment will be used. In our example, we would collect while, ID v1, hasNext, &&, ID v2, ++, $<$, MAX, etc. All of the tokens w_i's collected for all the changes in the recent history up to the current transaction are placed into the vocabulary V.

2) To perform a task k among all K tasks, one might make different changes with different tokens from V. Moreover, a change c in V might contribute to multiple tasks. Thus, each token w in a change c has a probability to achieve a task k.

We use a **token-distribution vector** ϕ_k of size V for the task k, i.e., *each element of* ϕ_k *represents the probability that a token* w *in a change* c *achieves the task* k. For example, in Figure 5, to achieve task 1, the probability that the change includes token w_1 is 25%. Putting together all of those vectors for all K tasks, we have a matrix called **per-task token distribution** ϕ.

3) A **task** k is represented by a set of changes with the corresponding probabilities of the tokens in those changes.


1) Vocabulary V of all tokens in changes ={w1, w2, w3, ...}
2) Token-distribution vectors ϕ_k for all tasks 1-K
3) Task 1 ϕ_1
Task 2 ϕ_2
Task K ϕ_K
4) Transactions
Transaction t with tokens in changes
Transaction
Transaction
w5 w9
w4 w6
w11 w3
Task assigning vector z_t for all tokens in all changes in t
Task proportion θ_t for t
Task
1
2
K

Task 1	ϕ_1	Task 2	ϕ_2	Task K	ϕ_K
w1	0.25	w2	0.3	w1	0.25
w2	0.2	w3	0.25	w2	0.03
w3	0.12	w5	0.18	w5	0.02
w4	0.12	w6	0.04	w7	0.01
w5	0.10	w9	0.03	w15	0.01
w6	0.10	w11	0.02	w16	0.01
...		...		...	

Fig. 5. LDA-based Task Context Modeling

Those changes contribute to achieve that task. A change that does not contribute to achieve a task will have its probability of zero. Vocabulary, tasks, and per-task token-distribution matrix are meaningful for all transactions in the history.

4) A **transaction** t has several changes with N_t tokens. Each transaction has two associated parameters:

a. **task proportion** θ_t: A transaction t can be for multiple tasks. Thus, we associate each transaction t with a proportion/distribution to model the contribution of the transaction t to each task k. The higher the value $\theta_t[k]$, the higher the changes in the transaction t contribute toward the task k. The total of all values $\theta_t[k]$ for all tasks $k = 1...K$ is 100%. For example, if $\theta_t = [0.2, 0.3, 0.4, ...]$, 20% of the changes in transaction t contribute toward task 1, 30% is toward task 2, etc.

b. **task assignment vector** z_t: This vector for the transaction t represents the assignment of the tokens in all changes in t to the tasks.

To find the tasks of a transaction t, as in LDA, we assume that the transaction t is an "instance" generated by a "machine" with 3 variables θ_t, z_t, and ϕ. Given a transaction t, the machine generates the vector z_t assigning each position in t a task k based on the task proportion θ_t. For each position, it generates a token w for a change c based on the task k assigned to that position in t and the token-selection vector ϕ_k of that task k.

The changes in all transactions in the history are observed from data. This LDA-based model can be trained to derive those 3 variables. For a new transaction t', we can derive the task assignment $z_{t'}$ and the proportion $\theta_{t'}$ of the tasks in t'. Thus, we can derive the tasks for all transactions.

V. Change Recommendation Algorithm

Based on our modeling of task context via LDA, we develop a change recommendation algorithm for the current code. Our algorithm is developed with two key design ideas:

1) *Source fragments that contribute similarly to the tasks in the change transactions would be changed in the similar manner*. Thus, given a source fragment s for suggestion, the likely (candidate) target fragment could be found in the *candidate changes in the past having similar source fragments with s in term of their tasks*.
2) The more frequently a target has been seen, the more likely it is the actual target of a given source fragment.

A. Task-based Similarity Measurement for Code Fragments

This measurement is used to measure the similarity between code fragments in term of their levels of contributions to the change tasks. The task contributions of a fragment can be computed by combining the task contributions from the tokens in the fragment (which are computed by topic modeling).

We realize that idea by using the per-task token distribution ϕ computed by topic modeling. Note that in Figure 5, ϕ is the matrix formed by putting together all vectors ϕ_k for $k = 1..K$. We first build **a task vector for each token** via ϕ. The size of the vector for a given token is the number of topics/tasks, each index corresponds to a topic/task and the value of an index k is the probability of that token being contributed toward

```
1 function Suggest(Fragment $s$, ChangeDatabase $C$)
2   PerTaskTokenDistribution $\phi$ = LDA($C$)
3   Initialize a map $T$
4   for $c = (u, v)$ in $C$
5     $sim$ = Sim($u$, $s$, $\phi$)
6     if $sim \geq threshold$
7       $score = sim \times c$.frequency
8       $T(v) = max(T(v), score)$
9   return Sort($T$)
```

Fig. 6. Change Recommendation Algorithm

the task k. For example, in Figure 5, if the number of tasks K=3, the task vector for token $w1$ is $v1 = [0.25, 0.0, 0.25]$ and that for token $w2$ is $v2 = [0.2, 0.3, 0.03]$. Since the tasks/topics in LDA [2] are assumed to be uniformly distributed over all documents in the corpus, such a task vector represents the contributions of that token to the tasks. For example, among those two tokens, $w1$ contributes to task 1 more than $w2$ does.

The summation of the task vectors for all tokens in a code fragment will represent the contributions of the corresponding fragment to the tasks. For example, if a fragment is composed by the two tokens $w1$ and $w2$, its combined task vector is $v = [0.45, 0.3, 0.28]$, which means that it contributes the most to task 1. We normalize the combined task vector from all tokens so that the sum of all values is 1. The normalized version of the above vector v is $\bar{v} = [0.43, 0.30, 0.27]$. Then, we use the normalized vector as the task vector for that fragment. *Such task vector represents the probability of the fragment contributing to a task. The task similarity between two code fragments f_1 and f_2 is measured by their shared contributions to the tasks normalized by the maximum of their contributions.*

$$Sim(f_1, f_2, \phi) = Sim(v_1, v_2) = \frac{\sum_{t=1}^{K} min(v_1[t], v_2[t])}{\sum_{t=1}^{K} max(v_1[t], v_2[t])} \quad (1)$$

B. Detailed Algorithm

Figure 6 shows the pseudo-code of our algorithm. The input of the algorithm is a source fragment s to be changed at the requested point in an IDE and the database of all past changes. The algorithm will output a ranked list of likely target fragments for s. To do that, the algorithm first builds the task model for the past changes by running LDA on the change transactions (line 2). The output of this step is the distributions ϕ_ks of tokens for all the tasks k in the past. Then, we use those distributions to find the source fragments with similar tasks. The algorithm looks for all prior changes (u, v) whose source fragment u is similar to the given source s with respect to their tasks (lines 4–6). The similarity measurement is shown in formula (1). If it finds such a change c, it will update the target of c in the store T of all candidate target fragments. The algorithm gives higher scores to the targets that both have occurred *more frequently* in the past and belong to the changes whose sources are *more similar* to the given source s (line 7). Since a candidate target can belong to multiple changes (with similar sources), we use the best score from all those changes when updating the store T of the candidate targets (line 8). Finally, all candidate targets in T are ranked based on their scores.

TABLE I
COLLECTED PROJECTS AND CODE CHANGES

Projects	88
Total source files	204,468
Total SLOCs at last snapshots	3,564,2147
Java code change revisions	88,000
Java fixing change revisions	19,947
Total changed files	290,688
Total SLOCs of changed files	116,481,205
Total changed methods	423,229
Total AST nodes of changed methods	54,878,550
Total detected changes	491,771
Total detected fixes	97,018

VI. EMPIRICAL EVALUATION

We conducted several experiments to evaluate TasC's quality. We aim to answer two research questions:

1. How accurate TasC is in recommending changes and fixes?

2. Does TasC improve the quality of change recommendation over the models using other types of context such as structure [19] and co-change associations [22], [24], [27] and over the base models with only repeated changes [17]?

We evaluated the recommendation quality for both general changes and bug-fixing changes (fixes). We also studied several characteristics of task context in code change recommendation.

A. Data Collection

We collected code change data from open-source projects in SourceForge.net [23]. To filter out the toy projects among them, we kept only projects that satisfy two criteria: 1) having standard trunks (i.e., the main line of development) in their SVN repositories, and 2) having at least 1,000 revisions with source code changes. Since the numbers of revisions greatly vary among these projects (from some thousands to some ten thousands), we collected only the first 1,000 revisions with Java code changes for each project. We used the keyword-based approach [26] to detect fixing revisions (i.e., those that have log messages containing keywords indicating fixing activities).

B. Experimental Setup and Metrics

We used a longitudinal setup. For each project, we divided equally the 1,000 revisions into 10 folds, each of which has 100 consecutive revisions. Folds are ordered by the committing time of their revisions. A testing change is picked from a testing fold i ($i = 2...10$). The changes in the previous folds (0 to $i-1$) are used to compute the task context via LDA.

We measure the quality of change recommendation via top-ranked accuracy. Given a source fragment of a testing change, our tool produces a ranked list of candidate target fragments. If the actual target matches one fragment in the top-k candidate list, we count it as a hit for top-k recommendation. The accuracy of top-k recommendation is computed as the ratio between the number of top-k hits over the number of tests. We recorded both the accuracy for each project and that for all the projects.

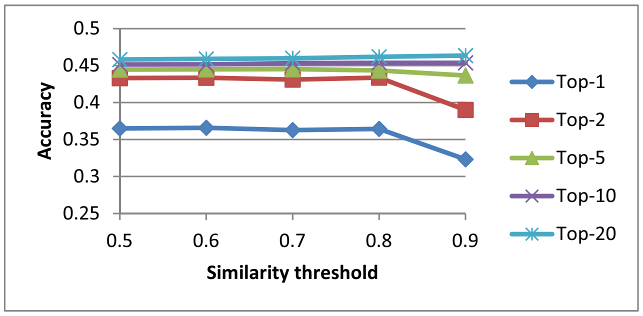


Fig. 7. Impact of Similarity Threshold on Accuracy in Project ONDEX

To evaluate the recommendation quality in the cross-project setting, given a testing change in a project, we use the changes from all previous folds of that project along with the changes from all folds of the other projects as the training data. For topic modeling implementation, we built our model on top of the LDA library in MALLET [11]. For the parameters of LDA, we used different values for the number of tasks K to see its impact on the accuracy (Section VI-C). For other parameters, we used the suggested values from MALLET, i.e., α=0.01, β=0.01 and the number of iterations is 1,000.

C. Parameter Sensitivity Analysis

In this experiment, we analyzed the impact of the similarity threshold (Figure 6) and the number of tasks K on the recommendation accuracy. We randomly chose the project ONDEX. To analyze the threshold, we fixed the number of task $K = 10$ and ran TasC with different values of the similarity threshold from 0.5 to 0.9. Figure 7 shows the accuracy results for different top-k recommendations. When the threshold is small, the number of candidates is large, thus, one would expect that the accuracy is low. However, from the results, we can see that when the threshold is less than or equal to 0.8, accuracy is stable. This occurs due to two reasons. First, we compute the ranking score by multiplying the similarity with the frequency (line 7 of Figure 6). Second, the frequencies of candidate changes are usually small. Therefore, the candidates with low similarity have low chance to be ranked high in the recommendation list. When the threshold is increased from 0.8 to 0.9, the number of candidates drops leading to decreasing in accuracy. We use threshold of 0.8 in the next experiments since it gives the best accuracy with a minimum set of candidates.

The accuracy results are shown in Figure 8 when we varied the number of tasks K. From top-5 to top-50, the model is not sensitive to K because the numbers of candidates in the ranked lists are usually small. The best accuracy can be achieved at $K = 10$. When K is small, many code fragments are considered similar because the size of a topic vector is small and many fragments are grouped into the same LDA topics/tasks even though they are for different tasks. When K is too large, the task vectors of source fragments become distinct. Nuance tasks become dominant. Thus, many actual targets are not collected into the ranked list, leading to decreasing in accuracy.

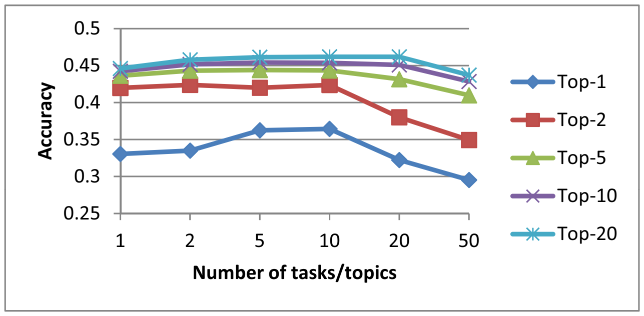


Fig. 8. Impact of the Number of Tasks/Topics on Accuracy in Project ONDEX

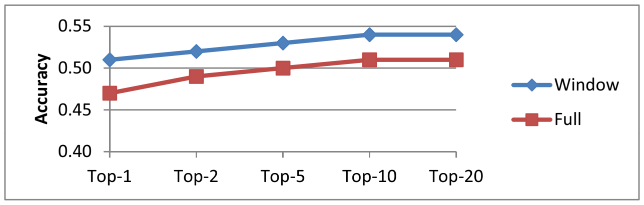


Fig. 9. Temporal Locality of Task Context

D. Locality of Task Context

In this experiment, we would like to study how the locality of training data for topic modeling affects accuracy. We study two aspects of locality: time and space. For temporal locality, we investigated whether using recent transactions and entire change history would produce different accuracy values. For spatial locality, we performed an experiment to compare the accuracy in two settings: 1) the training data from *within* the histories of individual projects, and 2) the training data from the current project as well as from the histories of other projects.

1) Temporal locality of task context: For each testing change, we ran our tool with two different training datasets for LDA. The first one simulates the use of recent transactions by using only the *most recent* fold before the revision of the testing change. The second training dataset uses the *full* history prior to the revision of the testing change. The comparison result is shown in Figure 9. As seen, for all the top-k accuracy, the accuracy when using a small window of prior revisions is higher than the accuracy when using the full change history. Examining the results for all individual projects, we observed the same trend consistently. We used a paired Wilcoxon test to compare the distributions of the accuracy over all projects between using a small window of history and entire history settings. The test result shows that the accuracy for the former is significantly higher than that for the latter.

This result shows that **using a window of recent changes is more beneficial than using the full history** in capturing the task context in change recommendation. Using recent data not only increases accuracy but also reduces the running time. The intuition is that task context is local in time, i.e., a task is usually realized within a certain window of transactions, rather than spanning over many transactions in the development history.

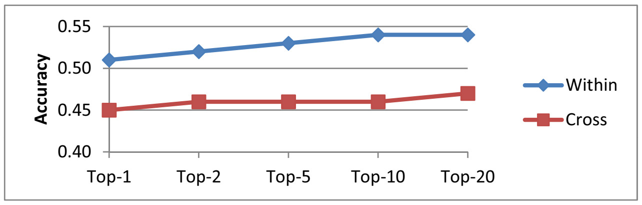


Fig. 10. Spatial Locality of Task Context

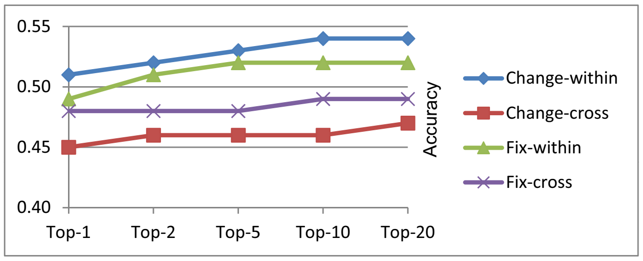


Fig. 11. Accuracy Comparison between Fixing and General Changes

2) Spatial locality of task context: We studied spatial locality by comparing the accuracy between within-project and cross-project settings. In this experiment, we used the training data from the windows of change histories. The process is similar to that of the experiment for temporal locality. The result is shown in Figure 10. As seen, using training data from individual projects gives better accuracy for all top ranks than using data from other projects. We also observed this result consistently in all projects in our dataset. A paired Wilcoxon test to compare the distributions of accuracy over all projects between two settings shows that the difference is statistically significant.

This result implies that **the task context captured by topic modeling with LDA is local in space**: tasks/topics are not shared much among different projects. Adding data from different projects might not improve the recommendation quality. In contrast, it increases complexity and yet could add noise to the task inference, thus, reducing accuracy.

E. Fix Recommendation Using Task Context

We also performed experiments on bug fixing changes. As seen in Figure 11, similarly to the general changes, the accuracy is higher in within-project setting than in cross-project setting. Comparing between fixes and general changes, fix recommendation accuracy is lower than change recommendation accuracy in within-project setting. However, fix recommendation accuracy is higher in cross-project setting. This result implies that **the fixing tasks are more likely to be repeated across projects than within a project, while general change tasks are more likely to be repeated within a project than across projects**.

F. Accuracy Comparison with Base Models

In this experiment, we aim to answer the question if our model using task context improves over the base models that use only repeated changes and do not use context information [17]. Those base models also use the recommendation algorithm in

TABLE II
ACCURACY COMPARISON BETWEEN TASC AND BASE MODELS

(a) Within-project Accuracy Comparison

	Top-1	Top-2	Top-5	Top-10	Top-20
Exact	0.20	0.21	0.21	0.21	0.21
Similar	0.32	0.34	0.35	0.35	0.35
TasC	0.51	0.52	0.53	0.54	0.54

(b) Cross-project Accuracy Comparison

	Top-1	Top-2	Top-5	Top-10	Top-20
Exact	0.22	0.23	0.24	0.24	0.24
Similar	0.35	0.37	0.38	0.38	0.39
TasC	0.45	0.46	0.46	0.46	0.47

Figure 6. However, they do not use topic modeling result to compute similarity in finding the candidate changes. Instead, the first base model, named Exact, uses all the changes whose source fragments are exactly matched to the given source s (i.e., their normalized ASTs are isomorphic). In the second base model, named Similar, the similarity of two fragments is measured via the similarity between their respective syntactic code tokens (after normalization). Specifically, the similarity is measured as the ratio between the length of the longest common sub-sequence of the two code sequences and the maximum length of their sequences. The similarity threshold is set to be 0.8, which is the same as that for task similarity.

The result is shown in Table II. The first base model misses many cases and achieves no more than 22% for top-1 recommendation. The reason is that exact matching in finding candidate changes would be too strict. That is why when we use the similar matching in the second base model, the accuracy increases more than 150% relatively.

Importantly, TasC with task context relatively improves much over both the base models: more than 250% over Exact model and almost 130% over Similar model. The large improvement is observed consistently for all top-k accuracy in both within-project and cross-project settings. This improvement could be attributed to the use of topic modeling to capture a higher level of abstraction in the tasks of the code changes. We will show some examples to demonstrate this in Section VI-H.

Comparing between within- and cross-project settings, TasC achieves better accuracy in the former than in the latter. In contrast, the base models achieve better accuracy in the latter. While adding more change data from other projects introduces noise to task inference and reduces the accuracy in TasC, using more changes in the base models increases the chance that a test change has been seen in the past, thus, reduces the number of missing cases and increases the accuracy.

G. Comparison with Structural and Co-Change Contexts

In this experiment, we compare TasC with task context and the state-of-the-art models using two types of contexts: 1) structural context (e.g., used in FixWizard [19]), and 2) co-change context (e.g., used in Rolfsnes *et al.*'s association [22], Ying *et al.* [24], and Zimmmermann *et al.* [27]). Let us briefly explain those contexts and show comparison results.

The structural context captures the surrounding code of a change. This context is a set of code fragments containing the source fragment of the change (it is a set because a fragment can be nested in more than one fragments). The statements in the context are also normalized and collapsed in the same manner as in code change extraction.

Note that those existing models recommend at the file or entire method levels. In this work, we aim to compare TasC with the models using co-change context at a finer granularity. Thus, we re-define the co-change context at the fine-grained level as the set of fine-grained changes that occur in the same transaction with a change. The idea of using this context is that given a change co in the same transaction with the test change, candidate changes that have frequently co-occurred with co in the past will more likely be the actual change.

1) Using other contexts: We explore the following contexts.
Structural context. We added structural context to the base model Similar to build model Structure as follows. If among the candidate changes $\{c = (u,v), Sim(u,s) \geq threshold\}$, there exist changes that share structural context with the given source s, we will keep only those changes. That is, we will skip all the changes that do not share structural context with s. A change $c = (u,v)$ shares structural context with s if the set of code fragments as the structural context of u overlaps with that of s. That is, at least one ancestor code fragment of u is exactly matched with some ancestor fragment of s. The scoring and ranking schemes are the same as in the model Similar.
Co-change context. To build the model Co-change with co-change context, we ran an association rule mining algorithm on all transactions. Then, if we find the candidate changes that have co-occurred with the change under investigation in the same transaction, i.e., sharing the co-change context with that change, we keep only those changes as the candidates. Otherwise, the candidate changes will be the same as in the model Similar. The scoring and ranking schemes are the same as in the model Similar.
Combinations. We also investigated the combination of those two contexts and the task context. We combined the task and structural contexts to create the model named Task+Struct, and combined the task and co-change contexts to create the model named Task+Co. The method to add each context to our original task model is the same as the method to add each context to model Similar that was described above. Finally, we combined all three contexts to create the model named All. If we find the candidate changes that share either structural or co-change context with the change to be suggested, we will keep only those changes as the candidates. Otherwise, the candidate changes will be the same as in the model TasC.

2) Comparison results: The results are shown in Table III for general changes and in Table IV for fixes. For both types of changes and in both settings, TasC outperforms the structural and co-change models. Figure 12 shows the differences at the top-1 accuracy in which TasC relatively improves the accuracy almost 130%. This trend is consistent for all top-k accuracy.

TABLE III
GENERAL CHANGE RECOMMENDATION ACCURACY COMPARISON BETWEEN USING TASK CONTEXT AND USING OTHER CONTEXTS

(a) Within-project Comparison for General Change Recommendation

	Top-1	Top-2	Top-5	Top-10	Top-20
Single context					
TasC	0.51	0.52	0.53	0.54	0.54
Structure	0.32	0.34	0.35	0.351	0.35
Co-change	0.33	0.34	0.35	0.351	0.35
Combined context					
Task+Struct	0.51	0.52	0.53	0.54	0.54
Task+Co	0.50	0.52	0.53	0.53	0.54
All	0.50	0.52	0.53	0.53	0.54

(b) Cross-project Comparison for General Change Recommendation

	Top-1	Top-2	Top-5	Top-10	Top-20
Single context					
TasC	0.45	0.46	0.46	0.46	0.47
Structure	0.35	0.37	0.38	0.38	0.39
Co-change	0.36	0.37	0.38	0.38	0.39
Combined context					
Task+Struct	0.45	0.46	0.46	0.46	0.47
Task+Co	0.44	0.45	0.46	0.46	0.47
All	0.44	0.45	0.46	0.46	0.47

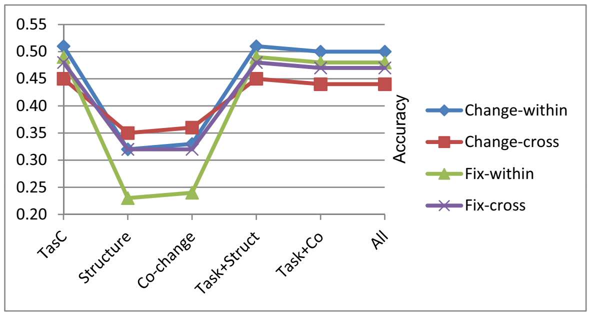


Fig. 12. Top-1 Accuracy Comparison between using Task Context and Others

Comparing the models with combined contexts and our model TasC, we see that adding other contexts does not improve the accuracy. We investigated the reason for this by examining the sets of candidate changes from different models. We observed that the number of candidates that share the structural or co-change context is much smaller than the number of those that do not. It means that most of the time, those models behave the same as the TasC model (without adding other contexts). Among the candidates that share other contexts, the number of choices for target fragments is small, mostly one, which means that most of them have been seen only once in the past. This makes most of the suggestions from those candidates are close to the results from TasC. In brief, the results for combined models are specific for this data.

H. Case Studies

Let us explain some cases where TasC correctly suggests at the top of its ranked list while the other models did not.

TABLE IV
FIX RECOMMENDATION ACCURACY COMPARISON BETWEEN CONTEXTS

(a) Within-project Comparison for Fixing Change Recommendation

	Top-1	Top-2	Top-5	Top-10	Top-20
Single context					
TasC	0.49	0.51	0.52	0.52	0.52
Structure	0.23	0.24	0.26	0.26	0.27
Co-change	0.24	0.24	0.27	0.27	0.27
Combined context					
Task+Struct	0.49	0.51	0.52	0.52	0.52
Task+Co	0.48	0.50	0.51	0.51	0.52
All	0.48	0.50	0.51	0.51	0.52

(b) Cross-project Comparison for Fixing Change Recommendation

	Top-1	Top-2	Top-5	Top-10	Top-20
Single context					
TasC	0.48	0.48	0.48	0.49	0.49
Structure	0.32	0.33	0.34	0.35	0.35
Co-change	0.32	0.33	0.34	0.35	0.35
Combined context					
Task+Struct	0.48	0.48	0.48	0.49	0.49
Task+Co	0.47	0.48	0.48	0.48	0.49
All	0.47	0.48	0.48	0.48	0.49

	Source	Target
Test	return v1.isEmpty () ? SWGResourceSet.EMPTY : v1;	return v1;
Candidate	return v1.size () > 0 ? v1 : SWGResourceSet.EMPTY ;	return v1;

Fig. 13. Case 1 in the project SWGAide

Figure 13 shows the first one which is in the project SWGAide, a utility for SWG's players. The test is a change at revision 802. In this case, TasC found a candidate change at revision 728 that contains the correct target. The base model Exact did not suggest any target because this source fragment had never appeared before. The other base model Similar could find some candidates in the past changes but none of them contains the correct target. It missed the candidate in Figure 13 because the two source fragments are too much different in terms of code token sequences (one calls isEmpty() and the other calls size()). However, those two checking conditions are actually two alternatives for checking if the set (SWGResourceSet) is empty or not. Both of them are identified by LDA as contributing very similarly to the tasks in the past changes. In this example, TasC gave isEmpty the concrete vector with $(task3 = 0.014; task7 = 0.007)$ and gave size the vector with $(task3 = 0.015; task7 = 0.011)$. In each pair of numbers, the left number is the task id and the right number is the probability/contribution of the token in that task. Thus, even two code fragments look quite different, they are still considered similar in terms of tasks.

The second case is a test in project ONDEX, an open source framework for text mining (Figure 14). The base models could not find this candidate because the code of two source fragments is different: one uses modifier final primitive type int and one uses class Integer with additional keyword new for class

	Source	Target
Test	final int v1 = v2.readInt ();	int v1 = v2.readInt ();
Candidate	Integer v1 = new Integer (v2.readInt ());	int v1 = v2.readInt ();

Fig. 14. Case 2 in the project ONDEX

instantiation. However, the tokens final, int and Integer appear in many places for all the tasks, thus, their contributions to tasks are very low. Thus, they do not affect the task similarity between two source fragments. TasC was able to match two sources and suggested the correct target.

I. Threats to Validity

We conducted our empirical evaluation with open-source Java projects repositories. Thus, the results could not be generalized for closed-source projects or the projects written in other languages. There are also many datasets using other version control systems and/or hosted on other hosting services that we have not covered. We plan to extend our evaluation to include projects hosted on GitHub and written in C/C++ in the future work. Our comparison suffers from the threat that the methods we used to integrate the context might not be the most suitable ones. Because we aimed to evaluate change recommendation at the fine-grained level, we re-implemented the existing approaches [22], [24], [27] for comparison.

VII. Related Work

Ying *et al.* [24] and Zimmermann *et al.* [27] propose approaches to *suggest a co-change* at the file and function levels. Ying *et al.* examine the co-changing files in the history and use association rule mining algorithm to find frequently co-changed files. From such information, they predict possible files for changing when given a newly changed file. Zimmermann *et al.* support not only co-changes at the file level, but also at the function and field level. They also use association rule mining algorithm with support and confidence. Rolfsnes *et al.* [22] improve change recommendation at the file level using aggregated association rules. There are two key differences between TasC and those approaches. First, we aim to suggest at the finer-grained level of AST. Second, we use a statistical approach, rather than deterministic pattern mining algorithms in those approaches. Giger *et al.* [6] predict type of changes, e.g., 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 [5].

There are automated approaches to suggest a fix for a given code fragment. Such recommendation is in the context of automated program repairing. GenProg [7] is a patch generation method that is based on genetic programming. To evolve a program variant through transformation, it reuses the existing program statements in the current program and creates the combinations. SearchRepair [8], a repair technique that searches through a database of code fragments encoded as SMT constraints on input-output behavior. However, those tools do not consider the change/fixing history in the process. PAR [9], another program auto-repair technique, derives a patch by mining fixing patterns from prior human-written patches. To derive a patch, it does not use the task context in a fixing history. FixWizard [19] suggests a fix for a given fragment based on its similarity with other previously fixed code. The similarity is defined based on similar code structures and/or similar API usages.

Ray *et al.*'s Repertoire [20], [21] is a tool to identify ported edits between patches in forked projects. They compare the content of individual patches. LASE [12] is a tool to automate similar changes from examples. It creates context-aware edit script, identifies the locations and transforms the code. Similar to FixWizard, Repertoire and LASE are based on comparing the code and apply the changes from one place to another. In our prior work [17], we leverage repeated changes and fixes to suggest a change/fix for a given code fragment. The tool rely only on repeated changes. None of those existing approaches consider the task context in recent history.

Other approaches detect the *patterns of changes* to support software maintenance. SemDiff [3] mines the updating patterns to a framework to support automated updating for its client code. MAM [25] mines common graph transformations representing the code after migration to learn migration rules. Similarly, LibSync [18] learns adaptation patterns from client code and uses them to update a program to use a new version of a library. Negara *et al.* [15] detect frequent change patterns by using closed frequent itemset mining on a sequence of changes expressed as tree editing operations.

Barr *et al.* [1] reported a high degree of graftability, independent of size and type of commits. They also found that changes are 43% graftable from the exact version of the software being changed. Gabel and Su [4] found code also contains much syntactic redundancy: at the level of granularity with 6 tokens, 50–100% of each project is redundant. Mockus [13], [14] found the repeated files in multiple projects.

VIII. Conclusion

In this paper, we propose TasC, a novel statistical model that uses the task context of changes in software development history to suggest fine-grained code change/fix at the program statement level. We use Latent Dirichlet Allocation to capture the task context where topics are used to model change tasks. We conducted an empirical evaluation on a large dataset of 88 open-source Java projects containing more than 200 thousand source files and 3.5 million source lines of code (SLOCs) in their last revisions. Our result shows that TasC improves the suggestion accuracy relatively up to 130%–250% in comparison with the base models that do not use contextual information. Compared with other types of contexts, it outperforms the state-of-the-art models using structural and co-change contexts.

Acknowledgment

The authors would like to thank the anonymous reviewers for their constructive comments. This work was supported in part by the US NSF grants CCF-1518897, CNS-1513263, CCF-1413927, CCF-1320578, CCF-1349153, TWC-1223828, CCF-1018600, and CCLI-0737029.

References

[1] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE'14, pages 306–317. ACM, 2014.

[2] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, 2003.

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

[4] 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 Press, 2010.

[5] 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 (MSR'11)*, pages 83–92. ACM, 2011.

[6] E. Giger, M. Pinzger, and H. Gall. Can we predict types of code changes? an empirical analysis. In *MSR'12*, pages 217–226. IEEE CS, 2012.

[7] 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.

[8] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun. Repairing programs with semantic code search. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 295–306. IEEE Computer Society, 2015.

[9] 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'13, pages 802–811. IEEE Press, 2013.

[10] B. Livshits and T. Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 296–305. ACM, 2005.

[11] A. K. McCallum. Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu, 2002.

[12] N. Meng, M. Kim, and K. S. McKinley. LASE: locating and applying systematic edits by learning from examples. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE '13, pages 502–511. IEEE Press, 2013.

[13] 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.

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

[15] S. Negara, M. Codoban, D. Dig, and R. Johnson. Mining continuous code changes to detect frequent program transformations. Technical report, University of Illinois - Urbana Champaign, 2013.

[16] S. Negara, M. Codoban, D. Dig, and R. E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE'14, pages 803–813. ACM, 2014.

[17] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. In *Proceedings of the 28th International Conference on Automated Software Engineering*, ASE'13, pages 180–190. IEEE CS, 2013.

[18] 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.

[19] 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.

[20] B. Ray and M. Kim. A case study of cross-system porting in forked projects. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 53:1–53:11. ACM, 2012.

[21] B. Ray, C. Wiley, and M. Kim. REPERTOIRE: a cross-system porting analysis tool for forked software projects. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 8:1–8:4. ACM, 2012.

[22] T. Rolfsnes, L. Moonen, S. Di Alesio, R. Behjati, and D. Binkley. Improving change recommendation using aggregated association rules. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 73–84. ACM, 2016.

[23] SourceForge. http://sourceforge.net/.

[24] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30(9):574–586, Sept. 2004.

[25] 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.

[26] 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.

[27] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572. IEEE, 2004.