Predicting Code Context Models for Software Development Tasks

Zhiyuan Wan*
Zhejiang University
Hangzhou, China
wanzhiyuan@zju.edu.cn

Gail C. Murphy
University of British Columbia
Vancouver, Canada
murphy@cs.ubc.ca

Xin Xia
Monash University
Melbourne, Australia
xin.xia@monash.edu

ABSTRACT

Code context models consist of source code elements and their relations relevant to a development task. Prior research showed that making code context models *explicit* in software tools can benefit software development practices, e.g., code navigation and searching. However, little focus has been put on how to proactively form code context models. In this paper, we explore the proactive formation of code context models based on the topological patterns of code elements from interaction histories for a project. Specifically, we first learn abstract topological patterns based on the stereotype roles of code elements, rather than on specific code elements; we then leverage the learned patterns to predict the code context models for a given task by graph pattern matching. To determine the effectiveness of this approach, we applied the approach to interaction histories stored for the Eclipse Mylyn open source project. We found that our approach achieves maximum F-measures of 0.67, 0.33 and 0.21 for *1-step*, *2-step* and *3-step* predictions, respectively. The most similar approach to ours is *Suade*, which supports *1-step* prediction only. In comparison to this existing work, our approach predicts code context models with significantly higher F-measure (0.57 over 0.23 on average). The results demonstrate the value of integrating historical and structural approaches to form more accurate code context models.

CCS CONCEPTS

• **Software and its engineering** → **Development frameworks and environments**; • **Human-centered computing** → *Human computer interaction (HCI).*

KEYWORDS

Context Models, Task, Developer, Interaction, Context Prediction

1 INTRODUCTION

As a software developer performs a development task, she spends substantial time searching and navigating through code to understand relevant parts in a software system for that task. Meanwhile, she forms, in her mind, an implicit *code context model* consisting of source code elements and relations between those elements relevant to the task [12]. When even a portion of such a model can be made explicit, the information in the model can be used in software tools to benefit software developers and the software development project. For example, approaches to make aspects of code context models explicit have shown promise to support searching activities [19], to improve code recommendations [13, 29] and to improve the quality of changes made to a system [7].

Despite the promise of improving software tools using explicit code context models, there has been little focus on how to proactively form these models to benefit a software developer as he works. At present, there are three approaches that have been used to capture and represent code context models.

First, a code context model for a task can be explicitly created by the developer performing the task. For example, the CodeBasket tool enables a developer to capture explicitly her mental model as she works with code; this representation can support subsequent navigation of the code [4]. This approach requires the developer to expend extra effort to create the model. In an exploratory study conducted about the CodeBasket tool, participants requested support to automatically create or complete the representation.

Second, a code context model can be formed by utilizing information about the structure of the code. For example, Robillard describes an algorithm, called Suade, that leverages topological features of the code structure to suggest a fuzzy set of potential code elements of interest for a code context model given elements already identified as interesting (seed code elements) [30]. This class of approaches decreases the effort required to create a code context model, but still requires either additional effort on the part of a developer, or access to other information, such as documentation, which can be difficult to obtain.

Third, a code context model can be formed based on the history of the software development project. The history may describe which files were changed as part of a task as found in a source code repository or may contain information about both viewed and changed files as found in interaction histories [13] or change history data [29]. Techniques like association rule mining can be used against historical information to suggest what other code elements have been associated with given seed code element(s) in the past (e.g., [39]). These approaches tend to treat code elements involved in past changes as isolated with no relations, rather than ones that have structural, or other relations, to other elements that comprise the system. This treatment of elements limits the ability to form sufficiently complete code context models for tasks.

We believe the benefits of code context models can be increased by improving the proactive automatic formation of code context models as a developer works. In effect, the proactive formation would represent a collection of other code elements and relationships the developer is likely to need to draw on to complete the task, beyond a recommendation of what is the next code element for the developer to consider (e.g., [10]). The proactive formation of code context models can enable tools to draw upon the context of

*The work was done when the first author was affiliated with the University of British Columbia.

the work being performed to improve the accuracy of recommendations, whether for navigation, searching or other reasons, and to better speculate on, and prefetch answers to, questions a developer is likely to pose (e.g., [33]), amongst other potential applications. By automatically forming code context models, tools can provide these benefits at lower cost to the developer.

In this paper, we explore how we can improve the proactive automatic formation of code context models by investigating whether we can: 1) learn abstract patterns of how developers typically investigate structurally connected code elements when performing tasks on a system, and then 2) use the learned patterns to predict code context models based on a developer's new interaction with the code for a system. Specifically, we use interaction histories collected as a developer works to form code context models at different points of time in the development of a system. We assign stereotypes to code elements in these code context models that represent the behavioral aspects and design intents of the elements [22], such as whether a method is a *getter* or a *setter*. The stereotypes allow us to abstract away from specific code elements. We then mine patterns across these abstracted code context models; as each code context model is a graph, we use a graph pattern matching approach to support prediction. Our approach enables a d-step prediction where the known code context model is expanded to include likely code elements of interest up to d steps away in the structure from known code elements. When presented with code element seeds as a developer works on a new task on the system, we can apply the learned patterns to predict the future code context model. The approach applies equally well at the start or middle of a task.

We applied this novel approach to interaction histories created and stored as part of the Eclipse Mylyn open source project. We found that our approach achieves maximum F-measures of 0.67, 0.33 and 0.21 for *1-step*, *2-step* and *3-step* predictions, respectively. The most similar approach to ours is *Suade*, which supports *1-step* prediction only. In comparison to this existing work, our approach predicts code context models with significantly higher F-measure (0.57 over 0.23 on average). The results demonstrate the value of integrating historical and structural approaches to forming more accurate code context models.

This paper makes three contributions:

- We introduce a novel approach to forming code context models that learns abstract patterns of how developers work with code as part of performing change tasks to a system.
- We demonstrate that our approach can predict code context models effectively, achieving maximum F-measures of 0.67, 0.33 and 0.21 for *1-step*, *2-step* and *3-step* predictions, respectively.
- We provide a dataset that includes 1,887 code context models to enable future investigations by others[1].

We begin by describing existing work in the area of code context models (Section 2). Next, we describe our approach for forming a dataset using interaction histories (Section 3) and helping to form code context models from interaction histories (Section 4). We then evaluate the ability of the approach to predict the evolution of code context models (Section 5). We discuss the implications of results (Section 6) and limitations of the approach (Section 7), before concluding (Section 8).

[1]https://github.com/zhiyuan-wan/ASE_2020_predicting_code_context

2 RELATED WORK

Many empirical methods in a variety of settings have been used to explore how developers understand code, from investigating the comprehension approaches taken by developers (e.g., [36]) to studying how they interact with code and tools (e.g., [20]) as they perform change tasks to a system. A number of the studies performed about these phenomena find that developers spend a substantial amount of time searching and navigating source code to understand and locate the relevant parts for a task. To help developers perform these activities, researchers have investigated several ways to capture and represent the code context models developers mentally form as they perform their work.

Some of these efforts focus on saving code context models after the relevant code elements have been identified or navigated for work being performed. Concern graphs [32] help a developer manually capture and represent the relevant elements and relationships between them. Code Bubbles [5] propose a novel IDE editor interface that allows a developer to create views of code fragments relative to work being performed. CodeBasket [4] enables developers to externalize their mental models by providing a canvas on which developers can arrange code elements. These approaches specialize in saving the code context models after the relevant code elements have been identified or navigated for work being performed. We are interested in this paper in proactively forming a code context model so that the model is available for a developer and tools to make use of the information. More general benefits possible from increasing the capture and use of *context* in software development are provided elsewhere [24].

Other efforts focus on automating the creation, or otherwise lessening, the manual burden on a developer to capture code context models. Mylyn [13] automatically creates a task context, which contains information relevant to a code context model, from a developer's interactions with code. Suade [31] analyzes the structural dependencies of code elements that have already been navigated, and identifies additional relevant code elements as the context for a task. Our approach is most similar to Suade in the proactive generation of code context information. In contrast to Suade, our approach uses information learned from historical information about how a developer has worked to predict potential code context elements.

Closely related to the formation of code context models are various recommendation approaches. These approaches build on a variety of information to try to predict code that might be relevant to a developer's work. For example, DeLine et al. use information about how developers navigate a code base to recommend where a developer should navigate next [10]. Other approaches use program structural information [3, 14], textual similarity between code elements and task descriptions [15] version histories [16, 38, 39], or a combination of multiple sources [28]. Our approach differs in two ways. First, instead of aiming to produce a direct recommendation to the developer, we predict what code matters for a task to inform and improve other tools, such as the display of code or filtering of search results. Second, our approach can leverage additional context when expanding what a code context model may be by

basing predictions on the graph structure of previous code context models.

Our approach relies on the use of stereotype roles [22] to generalize from specific code context models formed as a developer works to abstract forms that facilitate pattern detection. We use these patterns as a basis for completing code context models based on developer's partial work on a task. Prior studies have utilized stereotype roles for other purposes, including generating natural-language summaries for code in Java [21, 23] and C++ [1] programming languages, feature location [2], detecting code smells [9], categorizing source code identifiers [27], generating commit messages [6, 18], categorizing methods in unit tests [17], and serving as an indicator of system design [11]. We are the first to use stereotype roles to summarize the behaviors of methods and classes in past code contexts and to then use the abstracted code context models to support prediction.

3 CODE CONTEXT MODEL DATASET

To experiment with the proactive formation of code context models, we need a dataset of such models. We form such a dataset using *interaction histories* captured as developers work with the Eclipse Mylyn open source project[2]. We describe the data we extract from the project (Section 3.1) and how we transform this data into code context models ready for experimentation (Section 3.2). Figure 1 provides an overview of the process used to create the dataset. The resulting dataset formed from this process is available online[3].

3.1 Data Extraction

The top part of Figure 1 describes the extraction of data from the Mylyn system development. The Eclipse Mylyn tool records interaction histories as a developer works on a code base. Each interaction history includes a record of the code elements that are viewed and edited by the developer. Mylyn enables one or more interaction histories to be associated with each task performed by developers on a system. For the development of the Mylyn tool, interaction histories are stored with the tasks recorded in the Eclipse Bugzilla system. We chose to use the Mylyn project as the data source for our investigations because the project has collected interaction histories for over 15 years, and these interaction histories represent the work of over 117 developers. We consider the threat to validity of our work from this choice in Section 7.

Bug Report Filtering. To gather interaction histories, we considered the 5,208 *FIXED* and *CLOSED* bug reports of the Mylyn project from the Eclipse Bugzilla bug tracker between April 2004 and December 2019. From this set of fixed bug reports, we filtered all bug reports that did not have one or more interaction histories associated with the report, leaving 1,246 bug reports to consider. The 1,246 bug reports have an average of 1.72 interaction histories attached (Min: 1, Max: 11, Median: 1, SD: 1.3).

Interaction Trace Extraction. We extracted and used the last interaction history associated with each of these bug reports. We only considered interaction histories with events directly recording interaction with code elements ("selection" and "edit" events about class, method, and field code elements). The final dataset consists of 1,219 valid interaction traces with an average of 145.19 interaction events (Min: 1, Max: 4,179, Median: 65, SD: 376.37).

3.2 Code Context Model Formation

The bottom part of Figure 1 describes the formation from the extracted data into code context models.

Breaking Interaction Histories. For code context models, we are interested in representing the models that developers usually keep in their minds as they work with code for a task. As a result, we need to break interaction histories into units that more likely represent a period of time in which a developer is working with the code and for which they may have formed a working mental code context model. To capture such units, we define the concept of a *working period*, consisting of the portion of the interaction history consisting of events within two hour time periods. We chose two hours because an analysis of all of the interaction histories showed two hours was the mean time between two consecutive interaction events in the histories. By applying this step, we formed 2,815 working periods.

Extracting Code Elements. We are only interested in interaction histories[4] recording work with code elements, as opposed to documentation or xml files. Thus, we filtered for interaction histories accessing or editing Java code elements as Mylyn is predominantly written in Java. Of the 2,815 working periods we identified, developers considered Java code elements in 2,726 working periods (96.8%).

Structural dependencies between code elements are not available in interaction histories. To capture structural information, we need to be able to relate each interaction history to version(s) of the code active when the interaction history was collected. Thus, for each working period, we 1) resolved the git repository for extracted code elements, 2) extracted event timestamps from the interaction history, and 3) associated each working period with code snapshot(s).

Resolving Git Repository. We resolved git repository (repositories) of accessed code elements for each working period as the Mylyn code is stored across several git repositories. This step excluded 839 working periods that access only coarse-grained code elements (*directory* or *file*), which lack structural relations, or involve code elements from unavailable code repositories (e.g., dependency libraries), or access only code elements that were not committed to the repository when the interaction history was collected. For example, for the working period in Figure 1, we resolved two related git repositories: `mylyn.tasks` and `mylyn.commons`. After this step, we are left with 1,887 working periods from which to form code context models.

Extracting Event Timestamps. We extracted the *StartDate* attribute of each interaction event from an interaction history as the timestamp of the event, and identify the timestamps of the first and last events. The timestamps help to locate the commits before and during the working period. In terms of the example working period

[2]https://www.eclipse.org/mylyn
[3]https://github.com/zhiyuan-wan/ASE_2020_predicting_code_context

[4]In the remaining of this section, unless otherwise mentioned, we use "interaction history" to refer to a portion of an interaction history corresponding to a working period.

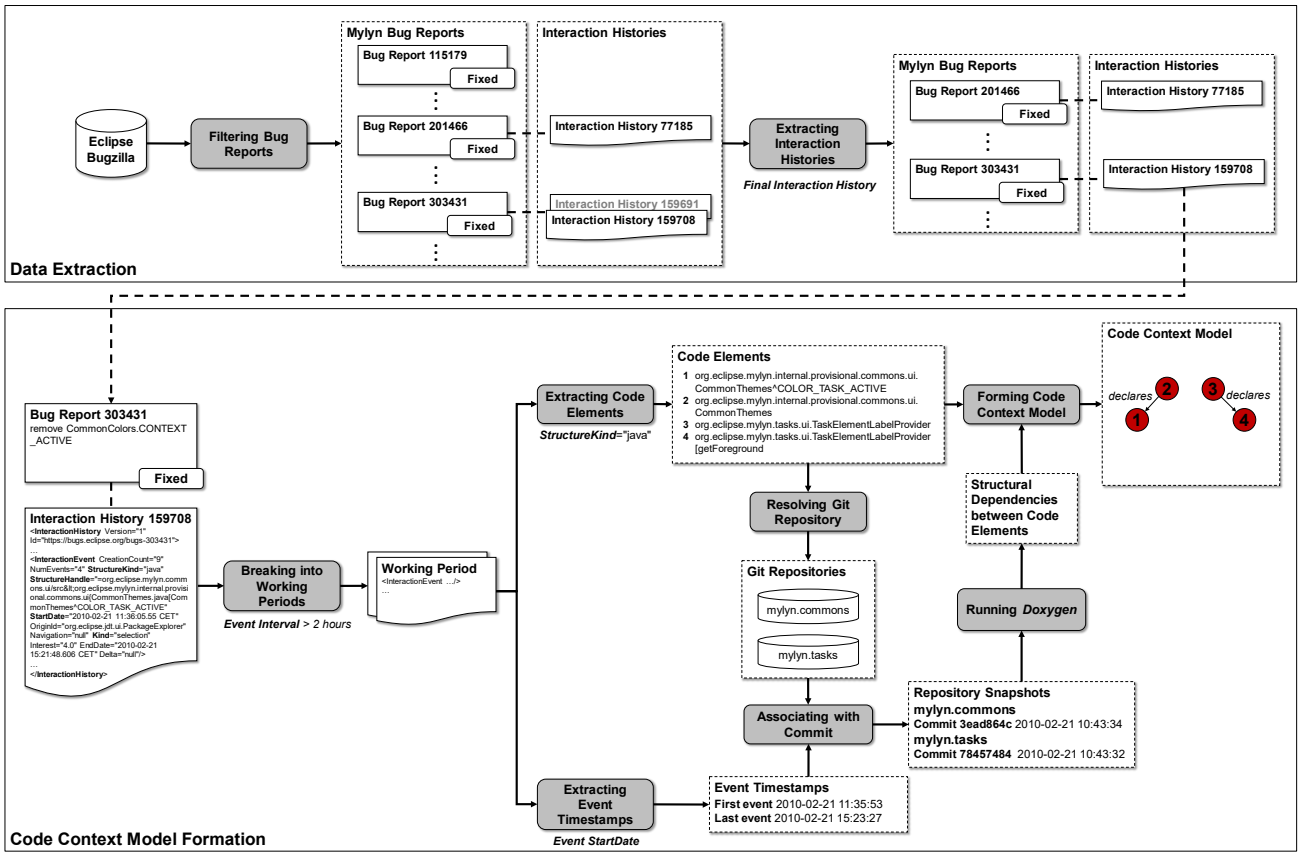


Figure 1: Process of collecting dataset.

Table 1: Characteristics of 1,887 code context models in dataset. In terms of node number, 51 code context models lie outside the interval $[Q1 - 3IQR, Q3 + 3IQR]$ where $Q1 = 3$, $Q3 = 14$, $IQR = 11$; 4 code context models lie above 200.

	Code Context Model			Connected Component (CC)		
	Node #	Edge #	CC #	Node #	Edge #	Diameter
Min	1	0	1	1	0	0
Max	944	421	572	152	210	16
Median	7	3	3	1	0	0
Mean	12.32	8.48	4.61	2.67	1.84	0.90
SD	30.34	18.38	17.58	5.02	6.51	1.56

in Figure 1, the timestamps of first and last events are "2010-02-21 11:35:53" and "2010-02-21 15:23:27", respectively.

Associating with Commits. By using the timestamps of first and last events in the interaction history, we associated each interaction history with one or multiple commits in the related git repositories. Specifically, we retrieved the most recent commit in the git repository prior to the timestamp of the first event in the interaction history. In addition, to capture the code changes during a working period, we accessed any other commits that occur before the timestamp of the last event. With regard to the example working period in Figure 1, we associated the working period with the commit `78457484` in `mylyn.tasks` and the commit `3ead864c` in `mylyn.commons`.

Running Doxygen. We used Doxygen [35] to identify structural relations between code elements. Specifically, we run Doxygen for each code snapshot of each commit associated with the working period. In this paper, we consider four structural relations: *declares*, *calls*, *inherits*, and *implements*. Figure 1 illustrates that using Doxygen we identify two *declares* relations between the code elements.

Forming Code Context Model. For each working period, we formed one code context model. The extracted code elements form the nodes of the code context model for a working period, while

the identified structural dependencies form the edges of the code context model. Figure 1 presents the code context model for the example working period, with four nodes and two directional edges labeled by structural relations. This code context model, which consists of two connected components, is the sole working period associated with the Bug Report 303431.

Our final dataset of code context models consists of 1,887 models. To give a sense of these models, Table 1 reports on statistics about these models. This data shows that the size of code context models varies, with an average of just over 12 nodes. The code context models are typically comprised of multiple connected components, with an average of 4.61, indicating that developers' worked with multiple clusters of structurally connected code elements during a working period. The right side of Table 1 reports statistics about the range of size of the 8,696 connected components comprising the code context models. This data shows that the average diameter of connected components is 0.90, indicating that the developers did not navigate code elements by following structural dependencies in depth during a working period. The maximum number of nodes in a code context model is 944, leading us to question the number of outliers. We found that 51 models lie outside the interval $[Q1 - 3IQR, Q3 + 3IQ]$ where the upper bound is 47 ($Q1 = 3$, $Q3 = 14$, $IQR = 11$)[5].

4 CODE CONTEXT MODEL PREDICTION

Our goal is to predict the code context model for a task initiated by a developer. We assume the developer has initiated the task by identifying some code elements of interest for the task. For example, a developer may have just started work on a task and could benefit from a prediction of the code that will need to be consulted to support the remaining work to be performed on the task. By framing the problem as a prediction opportunity, we aim to provide the code context model to tools that may help a developer search or otherwise work with the code rather than simply recommend the next step to a developer.

Specifically, in terms of a task on a system m, our approach takes as input a set of seeds, S, for each connected component cc in the complete code context model r for the task. Each seed $s \in S$ contains a portion of a connected component. Our approach supports a d-step prediction where the code elements predicted are structurally connected with, and d steps away, from the code elements in s. Our approach relies on abstract topological patterns of historical code context models to make an effective prediction. These code context models are extracted from previous interaction histories captured and stored for tasks on the system m. The extracted topological patterns are based on *stereotype roles* assigned automatically to code elements in s.

We describe the assignment of stereotype roles to code elements (Section 4.1) before explaining our prediction approach (Section 4.2).

[5]Q1 is the 25th quartile; Q3 is the 75th quartile; IQR (Interquartile Range) is defined as the difference between the 25th and 75th quartile and served as a measure of statistical dispersion.

4.1 Stereotype Role Assignment

Developers perform tasks on a software system to add new and fix existing functionality. As a result, developers often work on different parts of the code base. An analysis of the code context models in our dataset, which include Java code, indicates that developers accessed each *class* element an average of 4.76 times (Min: 1, Max 118, Median: 3, SD: 7.53), each *method* element 1.91 times (Min: 1, Max 60, Median: 1, SD: 2.11), and each *field* element 1.21 times (Min: 1, Max 9, Median: 1, SD: 0.56). These relatively low rates of access to code indicate that if we wish to build on patterns of access to predict code context models, we must abstract from the specific code elements accessed. We hypothesize that the roles the code elements play in the system are a good basis for this abstraction.

We use the method and class stereotype taxonomy proposed by Moreno and Marcus to assign roles [22]. The taxonomy provides 17 stereotype roles for method elements divided across four categories: structural accessor, structural mutator, creational and collaborational. An example of a specific stereotype within these categories is a structural mutator called *Command* that indicates a method performing a complex change to an object's state. The taxonomy also provides 13 stereotype roles for class elements, including *Data Provider*, which encapsulates data and consists mainly of accessor methods, and *Pure Controller* that consists entirely of controller and factory methods.

We use Moreno et al.'s *JStereoCode* tool [22] to assign stereotypes to each code element on the fly as needed during the prediction process. Specifically, we run the tool on the snapshots of code repositories associated with the context models. During the prediction, we search for each code element in the output of *JStereoCode* for the related snapshot to assign a stereotype. For instance, in terms of the example bug fixing task in Figure 1, we searched for `org.eclipse.mylyn.internal.provisional.commons.ui.CommonThemes` (node 2) in the *JStereoCode* output for the snapshot right after the commit `3ead864c`. As a result, we assigned *Pool* Class as the stereotype of node 2. Theoretically, it is possible that a code element could have multiple stereotypes across different snapshots in a repository due to software evolution.

4.2 Prediction Approach

The prediction approach consists of two stages. The first stage mines abstract topological patterns from historical code context models. Based on the abstract topological patterns, the second stage predicts the final code context model of a task, with a set of seeds S for each connected component in the code context model as input.

Stage 1: Topological Pattern Mining. The stage takes as input a system, m, a threshold of pattern support, $MinSupp$, and a repository of code context models, R. R is formed from interaction histories for previous tasks completed on m. Each code element of $r \in R$, where possible, has been assigned a stereotype. A set of topological patterns P is populated by mining frequent graph patterns in R. Specifically, we run *gSpan* [37] with R as input and $MinSupp$ as the parameter.

gSpan is an efficient algorithm for graph-based substructure pattern mining. Given a dataset of graphs, $D = \{G_0, G_1, ..., G_n\}$, $support(g)$ denotes the number of graphs (in D) in which g is a subgraph. *gSpan* explores depth-first search to find any connected

1: **Input**
2: S a seed set for a connected component cc
3: d prediction step
4: P a set of topological patterns
5: **Output**
6: $\hat{G}$ a set of matched subgraphs
7: s_d expanded seed
8: $\hat{s}'$ a predicted context model for the connected component cc
9: **for** all $s \in S$ **do**
10: $g = expand(s, d)$
11: $s_d = graphMerge(g, s_d)$
12: **end for**
13: $assignStereotypeRole(s_d)$
14: **for** all $p \in P$ **do**
15: $\hat{G} = \hat{G} \cup patternMatch(p, s_d)$
16: **end for**
17: **for** all $g \in \hat{G}$ **do**
18: $\hat{s}' = graphMerge(g, \hat{s}')$
19: **end for**
20: **for** all $v \in \hat{s}'$ **do**
21: $o_v = 0$
22: **for** all $g \in \hat{G}$ **do**
23: **if** $v \in g$ **then**
24: $o_v = o_v + 1$
25: **end if**
26: **end for**
27: $confidence_v = o_v / |\hat{G}|$
28: **end for**

Figure 2: Context model prediction algorithm.

subgraph g s.t. $support(g) \geq minSup$ (a minimum support threshold). We use a Python implementation[6] of *gSpan* and calculate the threshold *minSup* for *gSpan* by taking into account the size of graph dataset, i.e., $minSup = MinSupp * |D|$.

Stage 2: Context Model Prediction. Figure 2 illustrates the algorithm for this stage. The stage takes as input a set of seeds S for each connected component cc in the code context model r for a task on m, the prediction step, d, and a set of topological patterns that are derived at stage 1, P. The first step expands each $s \in S$ to g by performing a depth-first search along the structural relations. The resulting code elements are structurally connected with the nodes in s to a depth of d (line 10). After the expansion, all of the expanded graphs g are merged to form s_d (line 11); all code elements in s_d have been assigned a stereotype (line 13). Then, for each pattern p in P, we locate similar structures in s_d by applying the *patternMatching* function (line 15).

The *patternMatching* function searches for the pattern p among particular subgraphs in s_d. Note that d represents the prediction step and is limited by the diameter of a connected component. Thus, the *patternMatching* function extracts a subset of subgraphs in s_d for pattern matching. These subgraphs contain at least $(d+1)$ nodes from s_d, and are of two types: 1) subgraphs whose nodes exist in any $s \in S$; or 2) subgraphs that have d nodes that do not exist in any $s \in S$. Finally, the *patternMatching* function returns the subgraphs that are matched with the pattern p.

Finally, all matched subgraphs with $p \in P$ are merged to form the predicted context model $\hat{s}'$ for a connected component (line 18). The *confidence* value of each node in $\hat{s}'$ is calculated by evaluating the frequency of occurrence across matched subgraphs (line 20 - 28).

Figure 3 illustrates the process of *1-step* context model prediction with the example bug fixing task (bug ID: 303431) as shown in Figure 1. The code context model has two connected components. We take the connected component cc with node 1 and node 2 as the example. cc has a *Class* node (labeled 2) and a *Field* node (labeled 1). Each node in cc can serve as a seed for *1-step* prediction and belongs to the seed set S (labeled seed 1 and seed 2). The *1-step* prediction approach then expands each seed to a depth of $d = 1$ and generates an expanded graph g for each seed. After expansion, the two expanded graphs are merged to form s_1, where all nodes are assigned a stereotype. After running pattern matching with each $p \in P$, we find that the pattern `Pool class declares Field` matches 12 subgraphs in s_1, which form the set of subgraphs $\hat{G}$. All of the subgraphs in $\hat{G}$ are merged to form a predicted context model $\hat{s}'$. The *confidence* value for each node in $\hat{s}'$ is calculated based on its frequency of occurrence across subgraphs in $\hat{G}$. For instance, the *confidence* value of node 2 equals to $12/12 = 1$ because node 2 occurs in every subgraph. Meanwhile, node 1 occurs in just one subgraph thus its *confidence* value equals to $1/12$.

5 EVALUATION

We explain the results of applying the prediction approach to the dataset described in Section 3. Specifically, we explore three research questions to investigate the effectiveness of our approach for code context model prediction:

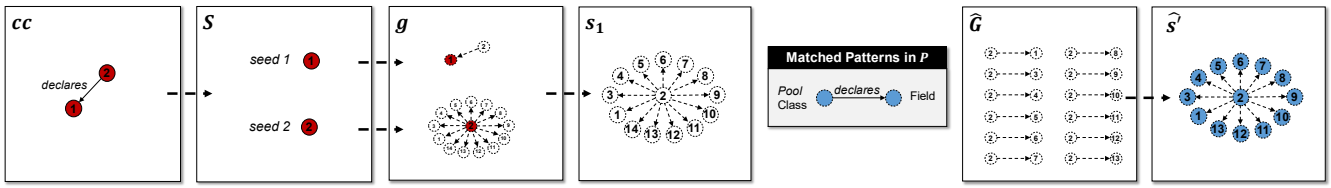
- **RQ1.** What kinds of patterns can be learned from interaction histories?
- **RQ2.** How does the performance of our approach for code context model prediction differ over various values of d, the number of steps of prediction?
- **RQ3.** How does our approach compare to the state-of-the-art?

5.1 Experiment Design

To answer the research questions, the experiment simulates code context model prediction and applies our prediction approach to the dataset.

Experimental Setup. We choose a simulation-based method to experiment with our approach and evaluate our approach in a usage scenario where a developer starts code search and navigation from the middle of a task. The experimental method involves a training set, a test set, and a simulator. The training set, R_{Train}, is a subset of our dataset, to mine the abstract topological patterns P. The test set, R_{Test}, is the remaining code context models from our dataset that are not in R_{Train}. Given the sequential nature of our dataset, we use the 1,254 code context models from the year 2007 to 2009 (84%) as the training set, and the 231 code context models from the year 2010 to 2011 (16%) as the test set.

[6] gspan-mining v0.2.2, https://pypi.org/project/gspan-mining



	Node	Stereotype	Confidence
1	org.eclipse.mylyn.internal.provisional.commons.ui.CommonThemes^COLOR_TASK_ACTIVE	Field	1/12
2	org.eclipse.mylyn.internal.provisional.commons.ui.CommonThemes	Pool Class	1
3	org.eclipse.mylyn.internal.provisional.commons.ui.CommonThemes^COLOR_CATEGORY	Field	1/12
4	org.eclipse.mylyn.internal.provisional.commons.ui.CommonThemes^COLOR_CATEGORY_GRADIENT_END	Field	1/12
5	org.eclipse.mylyn.internal.provisional.commons.ui.CommonThemes^COLOR_CATEGORY_GRADIENT_START	Field	1/12
6	org.eclipse.mylyn.internal.provisional.commons.ui.CommonThemes^COLOR_COMPLETED	Field	1/12
7	org.eclipse.mylyn.internal.provisional.commons.ui.CommonThemes^COLOR_COMPLETED_TODAY	Field	1/12
8	org.eclipse.mylyn.internal.provisional.commons.ui.CommonThemes^COLOR_INCOMING_BACKGROUND	Field	1/12
9	org.eclipse.mylyn.internal.provisional.commons.ui.CommonThemes^COLOR_OVERDUE	Field	1/12
10	org.eclipse.mylyn.internal.provisional.commons.ui.CommonThemes^COLOR_SCHEDULED_PAST	Field	1/12
11	org.eclipse.mylyn.internal.provisional.commons.ui.CommonThemes^COLOR_SCHEDULED_THIS_WEEK	Field	1/12
12	org.eclipse.mylyn.internal.provisional.commons.ui.CommonThemes^COLOR_SCHEDULED_TODAY	Field	1/12
13	org.eclipse.mylyn.internal.provisional.commons.ui.CommonThemes^FONT_EDITOR_COMMENT	Field	1/12
14	org.eclipse.mylyn.internal.provisional.commons.ui.CommonThemes[isCommonTheme	Collaborator Method	-

Figure 3: Sample task to illustrate how our approach does *1-step* prediction.

1: **Input**
2: R_{Test} test set
3: d predition step
4: **Output**
5: $\hat{S}$ seed sets
6: **for** all $r \in R_{Test}$ **do**
7: **for** all $cc \in r$ **do**
8: $size = nodeNumber(cc)$
9: $S = extractSubgraphs(cc, size - d)$
10: $\hat{S} = \hat{S} \cup \{(r, cc, S)\}$
11: **end for**
12: **end for**

Figure 4: Seed generation algorithm.

The simulator first creates multiple seed sets from the test set R_{Test}, as illustrated in Figure 4. For each code context model $r \in R_{Test}$, the simulator creates a seed set S for each connected component $cc \in r$ with prediction step d as input. Once the seed set S is created from R_{Test}, the simulator mines abstract topological patterns through R_{Train}. To select topological patterns, the simulator uses the minimum support $MinSupp$ as a threshold. Then, the simulator iterates over each seed set $S \in \hat{S}$ and bootstraps a prediction approach (i.e., our approach and the state-of-the-art) with S as input. Finally, the simulator aggregates prediction results of all connected components for each code context model $r \in R_{Test}$.

Experiment for RQ1. We investigate the numbers and kinds of patterns extracted by considering the effect of $MinSupp$ on the mined topological patterns. To make a tradeoff between the generalizability and specificity of mined patterns, we experiment with a range of 0.02 to 0.2 for $MinSupp$. We compare the topological patterns with various $MinSupp$ values in the range of 0.02 to 0.2 in increments of 0.02. We use the best result for $MinSupp$ in subsequent experiments.

Experiment for RQ2. To understand the effect of prediction step d, we evaluate the performance of our prediction approach with d to be 1, 2 and 3, representing *1-step*, *2-step*, and *3-step* predictions, respectively. Note that prediction step d is limited by the diameter of a connected component. For instance, a connected component of diameter 2 can support *1-step* and *2-step* predictions. The average diameter of the connected components in the test set is 2.1 (Min: 1, Max: 16, Median: 2, SD: 1.7). To consider adequate instances in the test set for each prediction step, we choose d to be 1, 2 and 3. We further investigate the impact of confidence threshold $MinConf$ on the performance of prediction. The *confidence* value of a code element in prediction results varies between 0 and 1. Thus, we set $MinConf$ to be in the range of 0.1 to 1.0 in increments of 0.1.

Experiment for RQ3. *Suade* is the state-of-the-art approach for code context model completion [31]. *Suade* leverages heuristic characteristics of the structural dependencies (i.e., specificity and reinforcement) to rank code elements connected with the seed. Given that the original implementation of *Suade* is not accessible, we implement *Suade* in Python and make it available online[7].

As *Suade* supports only *1-step* prediction, we compare only for this prediction step. We have the simulator iterate over $S \in \hat{S}$ for the prediction step $d = 1$, and bootstrap the experimental setting of *Suade* as described in Figure 5. We use the simulator to run *Suade* for each seed $s \in S$ and merge predicted results to form the predicted context model $\hat{s}'$ (line 7 - 10).

We simulate *Suade* with 1, 3 and 5 as the selection window w, respectively, where *Suade* makes top-w recommendation for each connected component. Whenever there exists a tie for a top position, we break the tie by randomly choosing the top element(s) among the equal-valued suggestions.

To ensure a fair comparison, we use *declares*, *calls* and *inherits* as the relation types L for *Suade*. For each selection window w when applying *Suade*, we calculate the actual size of $\hat{s}'$ as k for each

[7] https://github.com/zhiyuan-wan/ASE_2020_predicting_code_context

Input
  L relation types
  w selection window
  S a seed set for a connected component cc
Output
  $\hat{s}'$ predicted context model for the connected component cc
for all $s \in S$ **do**
  $F = suade(s, L, w)$
  $\hat{s}' = graphMerge(F, \hat{s}')$
end for

Figure 5: Experimental setting of *Suade*.

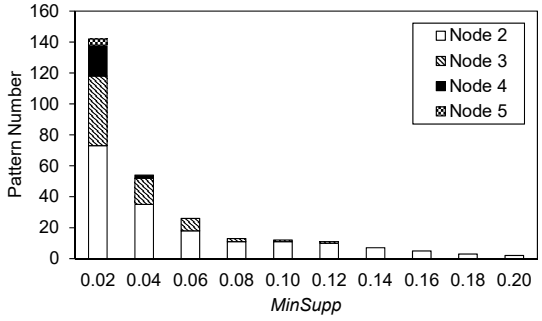


Figure 6: Number of patterns with *MinSupp* in the range of 0.02 to 0.20.

connected component in a code context model. For comparison, we choose top-k code elements with greatest *confidence* values for each connected component from the prediction results of our approach. We used the same tie breaker as *Suade*.

5.2 Measurement

To measure the effectiveness of prediction, we use several commonly used metrics. For each context model $r \in R_{Test}$, we use actual code elements V and predicted code elements V' to compute the metrics. V are the code elements in the actual context model of r. V' are the code elements in the context model predicted based on a portion of V. We calculate precision P and recall R metrics as:

$$\textbf{Precision } P = \frac{|V \cap V'|}{|V'|} \quad (1)$$

$$\textbf{Recall } R = \frac{|V \cap V'|}{|V|} \quad (2)$$

To capture the trade-off between precision and recall, we compute the harmonic mean *F-measure* from the averaged values of precision and recall across code context models:

$$\textbf{F-measure } F = \frac{2 * \overline{P} * \overline{R}}{\overline{P} + \overline{R}} \quad (3)$$

5.3 Results

We describe the results for each experimental question in turn.

5.3.1 RQ1. Abstract Topological Patterns. The first research question considers the kinds of patterns that can be learned from interaction histories. Figure 6 summarizes the number of patterns mined from R_{Train} with various numbers of nodes in the pattern. As the value of *MinSupp* increases, the number of topological patterns decreases sharply from 142 to 2. This result indicates that topological patterns, even in terms of stereotype roles, are not frequently occurring in the dataset.

Patterns with 2 nodes (node-2 pattern) account for more than 50% of the patterns across various *MinSupp* values. Node-5 patterns disappear when *MinSupp* >= 0.04, while Node-4 patterns disappear when *MinSupp* >= 0.06. Note that our proposed prediction approach leverages node-n patterns to make d-*step* prediction where $n > d$. To capture adequate topological patterns for prediction, we use the topological patterns where *MinSupp* = 0.02 in the experiments hereafter. The topological patterns with *MinSupp* = 0.02 are distributed across 94% of the code context models in the training set.

Table 2: Stereotype roles in the topological patterns with *MinSupp* = 0.02.

Stereotype	Count	Percentage
Method		
Collaborator	94	46.8%
Command-Collaborator	37	18.4%
Set-Collaborator	22	10.9%
Factory-Collaborator	13	6.5%
Set	10	5.0%
Constructor	7	3.5%
Non Void Command-Collaborator	5	2.5%
Get	5	2.5%
Abstract	3	1.5%
Local Controller	3	1.5%
Command	1	0.5%
Factory	1	0.5%
Class		
Boundary-Commander	45	34.4%
Boundary	34	26.0%
Other	31	23.7%
Entity	12	9.2%
Commander	3	2.3%
Interface	2	1.5%
Minimal Entity	2	1.5%
Factory	1	0.8%
Boundary-Data Provider	1	0.8%
Field	49	-

To give a sense of the kinds of patterns mined, Table 2 shows the distribution of stereotype roles of the nodes in topological patterns where *MinSupp* = 0.02. The *Method* and *Class* stereotypes involved in the patterns account for 63% of *Method* stereotypes and 38% of *Class* stereotypes that occur in the training set, respectively. The distribution of stereotype roles involved in the patterns are consistent with that in the training set.

Table 3: Test set sizes for *step-1, step-2* and *step-3* predictions.

	Code Context Model #	Connected Component #
Step-1	231	527
Step-2	165	291
Step-3	92	112

5.3.2 RQ2. Prediction Performance. The second research question considers the performance of our approach for code context model prediction. Table 3 reports the numbers of code context models (column 2) and connected components (column 3) in the test set that can support 1-step, 2-step and 3-step predictions.

F-measure. Figure 7 presents the resulting *F-measure* for *step-1*, *step-2* and *step-3* predictions. Each point represents the *F-measure* of all the predictions across code context models with *MinConf* ranging from 0.1 to 1.0.

The *F-measure* values of *step-1* start from 0.34 and end with 0.26, achieving the highest value 0.67 at *MinConf* = 0.3. The *F-measure* values of *step-2* start from 0.25 and end with 0.23, achieving the highest value 0.33 at *MinConf* = 0.6. The *F-measure* values of *step-3* start from 0.21 and end with 0.08, achieving the highest value 0.21 at *MinConf* = 0.1.

The average *F-measure* of *1-step* prediction is 1.7 times higher than that of *2-step* prediction (0.48 vs. 0.28), and 3.4 times higher than that of *3-step* prediction (0.48 vs. 0.14). The *F-measure* values of *2-step* prediction are slightly higher than those of *3-step* predictions. Thus, *1-step* prediction significantly outperforms *2-step* and *3-step* predictions.

Precision and Recall. Figure 8 presents the resulting precision and recall graphs for *1-step*, *2-step* and *3-step* predictions. Each point in each curve represents the average precision and recall of prediction results based on patterns with *MinConf* ranging from 0.1 to 1.0. The label for each point indicates the corresponding *MinConf*.

1-step prediction achieves a maximum *precision* average of 0.91 where *recall* = 0.38 (*MinConf* = 0.6). *2-step* prediction achieves a maximum *precision* average of 0.82 where *recall* = 0.14 (*MinConf* = 0.9). *3-step* prediction achieves a maximum *precision* average of 0.58 where *recall* = 0.05 (*MinConf* = 0.9).

Figure 8 shows that the *precision* averages for *step-d* predictions consistently increase and achieve the maximum values at the beginning, and drops sharply as the *recall* averages increase. The *recall* averages increase as *MinConf* values increase. Overall, we observed that *1-step* prediction significantly outperforms *2-step* and *3-step* predictions; *MinConf* can be used to make a tradeoff between precision and recall for the predictions.

5.3.3 RQ3. Comparison with Suade. The third research question asks how our approach compares to state-of-the-art, represented by the *Suade* algorithm. Table 4 compares the *precision*, *recall*, and *F-measure* values of our approach and *Suade*. Our approach shows *F-measure* values of 0.65, 0.58. and 0.49 when $k = 1$, $k = 3$ and $k = 5$, respectively, significantly outperforming *Suade* (0.20, 0.25 and 0.25). In addition, the performance metrics of our approach and *Suade* show similar tendency: as the window size k increases, *precision* values decrease, but *recall* values increase.

Table 4: Comparison of performance with *Suade*.

	Our Approach			Suade		
	P	R	**F**	P	R	**F**
$k = 1$	0.86	0.53	**0.65**	0.23	0.17	**0.20**
$k = 3$	0.50	0.69	**0.58**	0.20	0.32	**0.25**
$k = 5$	0.36	0.74	**0.49**	0.18	0.42	**0.25**

6 DISCUSSION

We reflect on the performance of our approach, delving into why the approach performs as it does and opportunities for improvement. We also consider the limitations of roles as a generalization mechanism and discuss how our approach differs from developer-oriented recommendation tools.

6.1 Analysis of Approach

We consider how our approach performs by delving into the types of patterns we find in the system and how those patterns affect the performance of the approach.

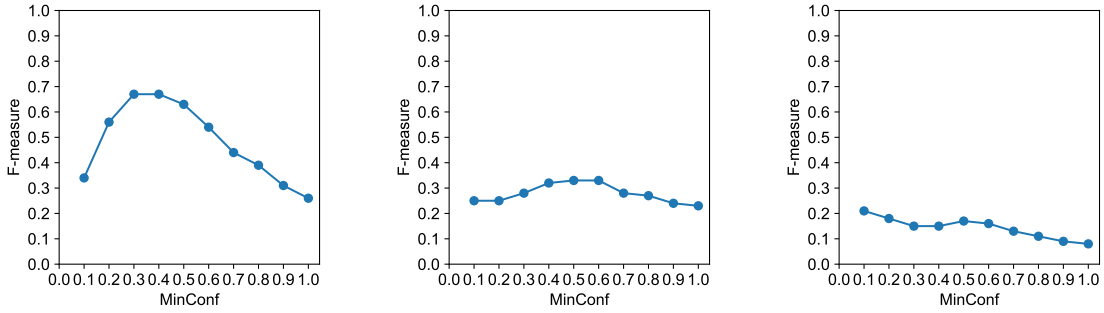
Prediction Accuracy vs. Confidence. In Figure 7(a), the *1-step* prediction yields significant higher prediction accuracy when *MinConf* = 0.4. To explore the reason, we analyzed the *confidence* values of *true positive* and *false positive* code elements in the prediction results. The *true positive* elements achieve an average *confidence* of 0.44 (Min: 0.01, Max: 1, Median: 0.38, SD: 0.31), while the *false positive* elements achieve an average *confidence* of 0.16 (Min: 0.01, Max: 1, Median: 0.12, SD: 0.16). A Mann-Whitney test showed that the distributions of *confidence* values are significantly different between *true positive* and *false positive* code elements in the prediction results with $p < 0.001$ ($U = 9289927.5$). *MinConf* = 0.4 helps to exclude 52% code elements from the true positives, and 92% from the false positives in prediction results.

To achieve an acceptable level of precision, the prediction approach should choose a *MinConf* value that helps to discriminate between relevant and irrelevant code elements in a code context model.

Prediction Accuracy vs. Prediction Step. As found in our study, step-1 prediction outperforms step-2 and step-3 predictions in terms of both recall and precision. We further investigate the results for step-2 and step-3 predictions.

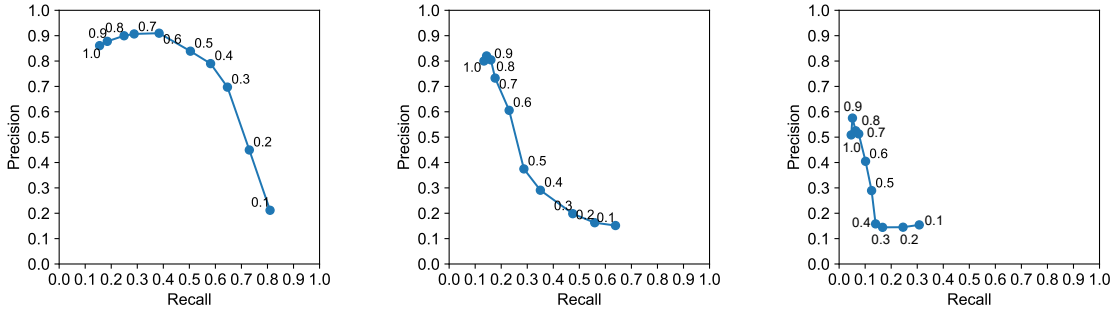
On the one hand, the inadequacy of topological patterns in step-2 and step-3 predictions leads to low recall. The number of topological patterns decreases sharply as the number of nodes in topological patterns increases. Amongst the 142 discovered patterns, we observed 73 2-node patterns, 45 3-node patterns, 20 4-node patterns, and four 5-node patterns. As prediction step d increases, the number of applicable patterns decreases (*1-step*: 142, *2-step*: 69, *3-step*: 24). The 69 applicable patterns for *2-step* prediction covers 59% of 22 stereotype roles. The coverage of stereotype roles for 24 applicable patterns for *3-step* prediction is 32%.

On the other hand, longer step prediction involves fewer code elements in the seed as for the identical topological pattern, which leads to low precision. For instance, a 3-node pattern (as shown in Figure 9) matches the subgraphs with two code elements as seed for



(a) *1-step* prediction (b) *2-step* prediction (c) *3-step* prediction

Figure 7: *F-measure* for *1-step, 2-step* and *3-step* predictions.



(a) *1-step* prediction (b) *2-step* prediction (c) *3-step* prediction

Figure 8: *Precision* and *recall* graphs for *1-step, 2-step* and *3-step* predictions. Labels represent *MinConf* values.

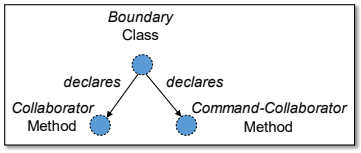


Figure 9: 3-node pattern applicable for 1-step and 2-step predictions.

1-step prediction but matches the subgraph with one code element as a seed for 2-step prediction. The precision of the pattern differs significantly between 1-step and 2-step predictions (0.46 vs. 0.20).

To support the formation of code context models with *diameter* > 2, future studies could explore whether applying multiple 1-step predictions can achieve an applicable level of accuracy.

6.2 Roles as a Generalization Mechanism

Our approach relies on the assignment of roles to code context models formed from previous tasks on the system. A limitation of our experimentation is its application to one system, which we discuss further in Section 7. Future work needs to consider whether the topological patterns we identify and their frequency are unique to a system or occur in similar distributions to other system developments. If the patterns frequently occur across systems, it may be possible to mine patterns on one system that can then be used even in new developments that do not yet have a history to learn from. Given that the stereotypes were developed for a paradigm of programming—object-oriented programming—and the patterns mined are relatively small (e.g., with 2.7 nodes on average), there is reason to be optimistic that the use of patterns across system developments is possible. Future experimentation is needed to test this hypothesis. In terms of *1-step* prediction, our approach captures "association rules" between structurally connected stereotype roles.

6.3 Developer-oriented Recommendations

Researchers have considered a variety of ways to make recommendations related to navigation for developers. These recommendation approaches suggest code often navigated to next from a given location [10], pre-fetch information related to likely developer queries from a given point in the code [8], present other code often changed with code currently being considered [39], amongst other aids. There are many challenges with providing recommendations directly to developers, including capturing the developer's attention to provide a recommendation, gaining the trust of the developer by providing good recommendations, and explaining why a recommendation is being made [25]. Muslu et al. considered some of these issues, suggesting that speculative analysis, which projects a recommended action, could be used to help with developer acceptance of recommendations by explaining the consequence of recommendations [26]. In trying to predict forward, our approach is similar to speculative analysis. Similar to speculative analysis,

we focus on predicting information about actions yet to be taken to help inform tools that might aid development, rather than trying to provide a recommendation directly to a developer. By predicting forward, we can suggest several possible alternatives, enabling tools that might help a developer fast forward across a number of steps at once that would otherwise be chosen one-by-one and potentially eliminate paths that are not useful. In addition, the background of developers (e.g., experience) may affect the interaction histories, and thus future work could investigate this aspect. Future work could also consider temporal information in the prediction.

7 THREATS TO VALIDITY

To implement our approach, we used *Doxygen*[8] to statically derive structural dependencies between code elements. Doxygen's static analysis may overestimate these dependencies, In particular, calling relationships may be overestimated because static analysis overestimates the number of target methods for each call site. The overestimation of dependencies could lead to the discovery of potentially non-existent patterns in actual code contexts, and further reduce the precision of prediction approach. We also relied on *JStereoCode* [22] to assign stereotypes to code elements across code context models. However, 2% of the code elements we processed could not be assigned a stereotype due to compilation errors in the source code. The missing stereotype roles would prevent us from finding potential patterns and predicting code elements that are with no stereotype roles assigned. This would reduce the recall of prediction approach. The use of a more precise static analysis tool and the ability to extract from git compliable code would have reduced these errors.

A significant limitation to our exploration is the reliance on one system development, Mylyn. The structure and interaction histories stored for Mylyn may not be representative of other projects. For instance, the distribution of stereotype roles relies on the system design of a project, the topological patterns may vary across different software projects. Mylyn is unique in having a repository of stored interaction histories from many developers over many years. However, no information is directly available about the experience level or other background of the developers contributing to Mylyn. To help understand the generalizability of the results, it would be helpful to explore the interaction traces from other systems stored by other tools, such as Blaze [34].

8 CONCLUSIONS

In this work, we have explored how developer interaction histories can improve the proactive formation of code context models. Specifically, we first learned abstract topological patterns from the code context models in interaction histories. Based on the patterns, we proposed an approach to predict the code context model for a new task. The accessed code elements of the task are used as the seed of the prediction. To evaluate our approach, we used a simulation-based method to create seeds, bootstrap predictions with various configurations, and collect prediction results. In this experiment, we found that our approach can predict code context model effectively, achieving maximum F-measures of 0.67, 0.33 and 0.21 for *1-step*, *2-step* and *3-step* predictions, respectively. In comparison with *Suade*, our approach achieves significantly higher F-measure (0.57 over 0.23 on average). The results demonstrate that integrating interaction histories and structural information can benefit the proactive formation of more accurate code context models.

[8]https://www.doxygen.nl

9 ACKNOWLEDGEMENTS

This research was supported by NSERC (RGPIN-2106-03758) and the Australian Research Council's Discovery Early Career Researcher Award (DECRA) (DE200100021).

REFERENCES

[1] Nahla J Abid, Natalia Dragan, Michael L Collard, and Jonathan I Maletic. 2015. Using stereotypes in the automatic generation of natural language summaries for C++ methods. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 561–565.

[2] Nouh Alhindawi, Natalia Dragan, Michael L Collard, and Jonathan I Maletic. 2013. Improving feature location by enhancing source code with stereotypes. In *2013 IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 300–309.

[3] Vinay Augustine, Patrick Francis, Xiao Qu, David Shepherd, Will Snipes, Christoph Braunlich, and Thomas Fritz. 2015. A field study on fostering structural navigation with prodet. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 229–238.

[4] Benjamin Biegel, Sebastian Baltes, Ivan Scarpellini, and Stephan Diehl. 2015. Code Basket: Making Developers' Mental Model Visible and Explorable. In *2015 IEEE/ACM 2nd International Workshop on Context for Software Development*. IEEE, 20–24.

[5] Andrew Bragdon, Robert Zeleznik, Steven P Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J LaViola Jr. 2010. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2503–2512.

[6] Luis Fernando Cortés-Coy, Mario Linares-Vásquez, Jairo Aponte, and Denys Poshyvanyk. 2014. On automatically generating commit messages via summarization of source code changes. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 275–284.

[7] Davor Cubranic and Gail C Murphy. 2003. Hipikat: Recommending pertinent software development artifacts. In *25th International Conference on Software Engineering*. IEEE, 408–418.

[8] Brian de Alwis and Gail C. Murphy. 2008. Answering conceptual queries with Ferret. In *30th International Conference on Software Engineering (ICSE 2008)*. ACM, 21–30.

[9] Michael J Decker, Christian D Newman, Natalia Dragan, Michael L Collard, Jonathan I Maletic, and Nicholas A Kraft. 2018. Which Method-Stereotype Changes are Indicators of Code Smells?. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 82–91.

[10] Robert DeLine, Amir Khella, Mary Czerwinski, and George Robertson. 2005. Towards understanding programs through wear-based filtering. In *Proceedings of the 2005 ACM symposium on Software visualization*. 183–192.

[11] Natalia Dragan, Michael L Collard, and Jonathan I Maletic. 2009. Using method stereotype distribution as a signature descriptor for software systems. In *2009 IEEE International Conference on Software Maintenance*. IEEE, 567–570.

[12] Thomas Fritz, David C. Shepherd, Katja Kevic, Will Snipes, and Christoph Bräunlich. 2014. Developers' Code Context Models for Change Tasks. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 7–18. https://doi.org/10.1145/2635868.2635905

[13] Mik Kersten and Gail C. Murphy. 2006. Using Task Context to Improve Programmer Productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/1181775.1181777

[14] Thomas D LaToza and Brad A Myers. 2011. Visualizing call graphs. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 117–124.

[15] Joseph Lawrance, Rachel Bellamy, and Margaret Burnett. 2007. Scents in programs: Does information foraging theory apply to program maintenance?. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*. IEEE, 15–22.

[16] S. Lee, S. Kang, S. Kim, and M. Staats. 2015. The Impact of View Histories on Edit Recommendations. *IEEE Transactions on Software Engineering* 41, 3 (March 2015), 314–330. https://doi.org/10.1109/TSE.2014.2362138

[17] Boyang Li, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. 2018. Aiding comprehension of unit test cases and test suites with

stereotype-based tagging. In *Proceedings of the 26th Conference on Program Comprehension*. 52–63.

[18] Mario Linares-Vásquez, Luis Fernando Cortés-Coy, Jairo Aponte, and Denys Poshyvanyk. 2015. Changescribe: A tool for automatically generating commit messages. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 709–712.

[19] Collin Mcmillan, Denys Poshyvanyk, Mark Grechanik, Qing Xie, and Chen Fu. 2013. Portfolio: Searching for relevant functions and their usages in millions of lines of code. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, 4 (2013), 1–30.

[20] Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. I know what you did last summer-an investigation of how developers spend their time. In *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 25–35.

[21] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Automatic generation of natural language summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 23–32.

[22] Laura Moreno and Andrian Marcus. 2012. JStereoCode: Automatically Identifying Method and Class Stereotypes in Java Code. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*. ACM, New York, NY, USA, 358–361. https://doi.org/10.1145/2351676.2351747

[23] Laura Moreno, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Jsummarizer: An automatic generator of natural language summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 230–232.

[24] Gail C Murphy. 2019. Beyond integrated development environments: adding context to software development. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 73–76.

[25] Emerson Murphy-Hill and Gail C. Murphy. 2014. *Recommendation delivery: Getting the user interface just right.* Springer, 223–242.

[26] Kivanç Muslu, Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2012. Speculative analysis of integrated development environment recommendations. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012*. ACM, 669–682.

[27] Christian D Newman, Reem S AlSuhaibani, Michael L Collard, and Jonathan I Maletic. 2017. Lexical categories for source code identifiers. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 228–239.

[28] David Piorkowski, Scott Fleming, Christopher Scaffidi, Christopher Bogart, Margaret Burnett, Bonnie John, Rachel Bellamy, and Calvin Swart. 2012. Reactive information foraging: An empirical investigation of theory-based recommender systems for programmers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1471–1480.

[29] Romain Robbes and Michele Lanza. 2010. Improving code completion with program history. *Automated Software Engineering* 17, 2 (2010), 181–212.

[30] Martin P Robillard. 2008. Topology analysis of software dependencies. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 17, 4 (2008), 1–36.

[31] Martin P. Robillard. 2008. Topology Analysis of Software Dependencies. *ACM Trans. Softw. Eng. Methodol.* 17, 4, Article Article 18 (Aug. 2008), 36 pages. https://doi.org/10.1145/13487689.13487691

[32] Martin P Robillard and Gail C Murphy. 2002. Concern graphs: finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*. 406–416.

[33] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. 2006. Questions Programmers Ask during Software Evolution Tasks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14)*. Association for Computing Machinery, New York, NY, USA, 23–34. https://doi.org/10.1145/1181775.1181779

[34] Will Snipes, Anil R Nair, and Emerson Murphy-Hill. 2014. Experiences gamifying developer adoption of practices and tools. In *Companion Proceedings of the 36th International Conference on Software Engineering*. 105–114.

[35] Dimitri Van Heesch. 2008. Doxygen: Source code documentation generator tool. http://www.doxygen.org. Online; accessed April 2020.

[36] Anneliese Von Mayrhauser and A Marie Vans. 1995. Program comprehension during software maintenance and evolution. *Computer* 28, 8 (1995), 44–55.

[37] Xifeng Yan and Jiawei Han. 2002. gSpan: graph-based substructure pattern mining. In *2002 IEEE International Conference on Data Mining, 2002. Proceedings.* 721–724. https://doi.org/10.1109/ICDM.2002.1184038

[38] Annie TT Ying, Gail C Murphy, Raymond Ng, and Mark C Chu-Carroll. 2004. Predicting source code changes by mining change history. *IEEE transactions on Software Engineering* 30, 9 (2004), 574–586.

[39] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. 2005. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering* 31, 6 (2005), 429–445.