Automating Change-Level Self-Admitted Technical Debt Determination

Meng Yan, Xin Xia, Emad Shihab, *Member, IEEE*, David Lo, Jianwei Yin, and Xiaohu Yang

Abstract—Technical debt (TD) is a metaphor to describe the situation where developers introduce suboptimal solutions during software development to achieve short-term goals that may affect the long-term software quality. Prior studies proposed different techniques to identify TD, such as identifying TD through code smells or by analyzing source code comments. Technical debt identified using comments is known as Self-Admitted Technical Debt (SATD) and refers to TD that is introduced intentionally. Compared with TD identified by code metrics or code smells, SATD is more reliable since it is admitted by developers using comments. Thus far, all of the state-of-the-art approaches identify SATD at the file-level. In essence, they identify whether a file has SATD or not. However, all of the SATD is introduced through software changes. Previous studies that identify SATD at the file-level in isolation cannot describe the TD context related to multiple files. Therefore, it is beneficial to identify the SATD once a change is being made. We refer to this type of TD identification as "Change-level SATD Determination", which determines whether or not a change introduces SATD. Identifying SATD at the change-level can help to manage and control TD by understanding the TD context through tracing the introducing changes. To build a change-level SATD Determination model, we first identify TD from source code comments in source code files of all versions. Second, we label the changes that first introduce the SATD comments as TD-introducing changes. Third, we build the determination model by extracting 25 features from software changes that are divided into three dimensions, namely diffusion, history and message, respectively. To evaluate the effectiveness of our proposed model, we perform an empirical study on 7 open source projects containing a total of 100,011 software changes. The experimental results show that our model achieves a promising and better performance than four baselines in terms of AUC and cost-effectiveness (i.e., percentage of TD-introducing changes identified when inspecting 20 percent of changed LOC). On average across the 7 experimental projects, our model achieves AUC of 0.82, cost-effectiveness of 0.80, which is a significant improvement over the comparison baselines used. In addition, we found that "Diffusion" is the most discriminative dimension among the three dimensions of features for determining TD-introducing changes.

Index Terms—Technical debt, software change, change-level determination, self-admitted technical debt

1 Introduction

Software companies and organizations always expect to deliver high quality software. However, in most practical settings, there are many constraints during the software development lifecycle that impact software quality. For example, constraints related to budget, scheduling and resources. Due to these constraints, developers may introduce suboptimal solutions to achieve various short-term goals, such as cost reduction, satisfying customers' needs and market pressure from competition [1]. Although the short-term goals are achieved, the suboptimal solutions may hurt the software in the long-term.

- M. Yan, J. Yin, and X. Yang are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China. E-mail: {mengy, yangxh}@zju.edu.cn, zjuyjw@cs.zju.edu.cn.
- X. Xia is with the Faculty of Information Technology, Monash University, Melbourne, VIC 3800, Australia. E-mail: xin.xia@monash.edu.
- E. Shihab is with Data-driven Analysis of Software (DAS) Lab, Department of Computer Science and Software Engineering, Concordia University, Montreal, QC H3G 1M8, Canada. E-mail: eshihab@encs.concordia.ca.
- D. Lo is with the School of Information Systems, Singapore Management University, Singapore 188065. E-mail: davidlo@smu.edu.sg.

Manuscript received 13 Dec. 2017; revised 22 Apr. 2018; accepted 24 Apr. 2018. Date of publication 0 . 0000; date of current version 0 . 0000.
(Corresponding author: Xin Xia.)
Recommended for acceptance by F. Shull.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2018.2831232

Technical debt (TD) is a metaphor that describes the aforementioned situation [2], [3]. Like financial debt, this metaphor regards taking suboptimal solutions as a type of "debt", which brings a short-term benefit (e.g., higher productivity or shorter release time) that needs to be paid back (i.e., using maintenance effort) even with "interest" in future. The "interest" is the potential penalty (i.e., increased effort) that will have to be paid in the future as a result of not performing the task optimally in the first place [4]. Prior work has shown that TD is common and can have a negative impact on the quality of the software [5], [6].

Due to the importance of TD, a number of studies proposed methods to help identify it. Generally, there are two main method used to identify TD [7]. One approach is to identify TD through source code analysis (e.g., by code metrics or code smells) [8], [9], [10], [11]. The other approach is to identify TD through source code comments, which is referred to as Self-Admitted Technical Debt (SATD) [12], [13], [14], [15], [16], [17]. SATD refers to the TD that is introduced by a developer intentionally and is documented by the developer using source code comments [16]. For example, the comment *"FIXME handle EVT_GET_ALL_SESSIONS later"* in the Tomcat project indicates that the corresponding code needs to be "fixed" in the future to handle the session problem.

Identifying TD through source code comments (i.e., SATD) has the following advantages compared with the identification of TD through source code analysis. First, SATD when compared with TD identified by code metrics or code smells, is more reliable since it is 'admitted' by developers [16]. The TD identified by code metrics or code smells might suffer from high false positive rates [7], [18]. Second, SATD identification is more lightweight compared with TD identified through source code analysis [7]. It does not require the construction of abstract syntax trees or other computationally expensive representations. For example, some code smell detectors may provide refactoring recommendations [19], [20] based on computationally expensive source code representations to match structural patterns or compute metrics, such as program dependence graphs [21] and method call graphs [22]. On the other hand, SATD identification only relies on source code comments that can be easily and efficiently extracted from source code files using regular expressions.

Thus far, all of the state-of-the-art approaches identify SATD at the file-level (e.g., [16], [17]). In essence, they identify whether a file has SATD or not. However, all of the SATD is introduced through software change (i.e., a commit to source control system) [17]. Therefore, it is beneficial to identify the SATD once a change is being made. We refer to this type of TD identification as "Change-level SATD Determination". Change-level determination aims to determine whether or not a change introduces SATD. Identifying SATD at the change-level can yield many benefits, such as:

- It can help to understand the TD context by tracing the introducing change. Previous studies that identify SATD on file-level in isolation cannot describe the TD context related to multiple files. Understanding the context of SATD-introducing changes can help to address the TD and possibly mitigate introducing TD.

- It can help to identify TD. Changes that are determined as SATD-introducing ones can be used to identify TD. In change-level SATD determination, only change features are used. In this way, the determination can be performed when a change is submitted. Once a SATD-introducing change is determined, we can provide a warning to the whole development team in a timely manner. Thus, it can help the team to identify and address TD.

Therefore, in this paper, we propose an automated change-level TD[1] determination model that can determine TD-introducing changes. To the best of our knowledge, this is the first work to focus on change-level TD determination. Particularly, we first identify TD from source code comments in source code files of all versions. Then, we label the changes that first introduce the TD comments as TD-introducing changes by analyzing source code comments. After labelling the TD-introducing changes, we extract 25 features that are grouped into three dimensions, i.e., diffusion, history, and message. The diffusion dimension, contains change features that depend on the comparison of two neighbouring revisions (e.g., number of modified subsystems and files) [23]. The history dimension aims to measure change features that depend on historical activities related to changed files and the developer that submitted the change (e.g., the number of developers that changed the modified files, and the number of historical changes made by the developer). The message dimension contains features by analyzing the change message (e.g., whether or not the change is a bug fix). Then, we build our determination model using a random forest classifier. In our model, it is noted that we use comments analysis in the data labelling step for identifying TD-introducing changes. And we use source code analysis and change history statistics for extracting change features.

To evaluate the effectiveness of our model, we conduct an empirical study on 7 open source projects containing a total of 100,011 changes, namely, Ant, Camel, Hadoop, Jmeter, Log4j, Tomcat and Gerrit, which is an enhanced version of the dataset provided by Maldonado et al. [17]. We adopt two performance measures (i.e., AUC and cost-effectiveness) using 10 times 10-fold cross-validation setting; AUC is the area under the receiver operator characteristic curve [24] and cost effectiveness evaluates the model performance given a certain cost threshold, e.g., a certain percentage of changed code to inspect (e.g., 20 percent). In practice, when a team has limited resources to inspect lines of code that potentially contain TD, it is crucial that the manual inspection of the top percentages of lines that are likely to contain TD can help developers discover as many TD as possible. In our study, by default, we define cost-effectiveness as the recall of TD-introducing changes when using 20 percent of the entire effort required to inspect all changes in test set to inspect the top ranked changes based on the confidence levels that our model outputs [23], [25]. And the total number of lines modified by a change as a measure of the effort required to inspect a change. We set the default threshold as 20 percent by following many prior studies on change-level determination [23], [26], [27], [28], [29], [30], [31] and we also analyze the cost-effectiveness by varying the threshold from 1 to 100 percent.

The experimental results show that our proposed model achieves AUC of 0.82, cost-effectiveness of 0.80 on average across 7 projects, which significantly improves the baseline approach by a substantial margin. Additionally, in order to understand what change factors impact TD-introducing most, we perform feature importance analysis. It consists of four steps: correlation analysis, redundancy analysis, importance feature identification and effect size calculation.

In summary, the main contributions of this paper are:

(1) We propose the problem of change-level TD determination, and we propose a change-level TD determination model, which utilizes 25 change features. To the best of our knowledge, this paper is the first work to perform change-level TD determination.

(2) We evaluate our proposed model on 7 projects with totally 100,011 changes. The experimental results show that out approach achieves AUC of 0.82, cost-effectiveness of 0.80 on average across 7 projects, which significantly improves the baseline approach in a substantial margin.

(3) We investigate the most important features that impact TD determination. The experimental results show that "diffusion" features is the most important dimension among the three dimensions of features.

1. In the remaining part of this paper, we use "TD" and "SATD" exchangeably.

TABLE 1
Summary of Studied Projects

Project	# All changes	# TD-introducing changes	Ratio
Hadoop	13,183	328	2.49%
Log4j	3,274	74	2.26%
Tomcat	16,876	487	2.89%
Camel	23,188	831	3.58%
Gerrit	17,973	135	0.75%
Ant	13,252	317	2.39%
Jmeter	12,265	517	4.22%
Total	*100,011*	*2,689*	*2.69%*

Paper Organization. The rest of the paper is structured as follows. Section 2 presents the data of our study. Section 3 presents our empirical study setup, including research questions, studied features, classifiers, validation setting and performance measures. In Section 4, we provide the experimental results and their analysis. In Section 5, we discuss three more findings that impact the determination model. Section 6 describes the threats to validity. Section 7 presents the related work of our study, including TD, self-admitted technical debt and change-level determination. At last, in Section 8, we conclude and present future plans.

2 Empirical Study Data

To conduct our study, we use an enhanced version of the publicly available dataset provided by Maldonado et al. [17]. The manually classified dataset of software changes from 7 open source projects, containing 100,011 changes from Ant, Camel, Hadoop, Jmeter, Log4j, Tomcat and Gerrit. This section details the dataset used in this study. First, we describe the summary of studied open source projects. Second, we describe the method of identifying TD-introducing changes.

2.1 Dataset

Table 1 lists the statistics of the studied projects. The studied projects cover different application domains, are of different sizes and have a varying numbers of contributors. Hadoop[2] is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. Log4j[3] is a logging library for Java. Tomcat[4] is an open source implementation of the Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies. Camel[5] is a versatile open-source integration framework based on known Enterprise Integration Patterns. Gerrit[6] is a free, web-based team code collaboration tool. Ant[7] is a Java library and command-line tool whose mission is to drive processes described in build files as targets and extension points dependent upon each other. Jmeter[8] is a Java application designed to load test functional behavior and measure performance. The analysis of the selected projects started on March 15, 2016. In total, there are 100,011 changes in the studied projects.

2. http://hadoop.apache.org/
3. https://logging.apache.org/log4j/2.x/
4. http://tomcat.apache.org/
5. http://camel.apache.org/
6. https://www.gerritcodereview.com/
7. http://ant.apache.org/
8. http://jmeter.apache.org/

2.2 TD-Introducing Change Identification

In the dataset, all the changes have been labeled as TD-introducing or not. We refer the reader to the paper by Maldonado et al. [17] for full details, however, to make this paper self-sufficient, we explain the key points related to the identification of TD-Introducing changes. The labelling process of identifying TD-introducing change are as follows:

Step 1: Checkout All File Versions. Since the TD-introducing change identification needs to track file change history, it is required that all versions of files in the studied projects be checked out from their version control systems. First, all Java source code files in the latest version of the project are identified. Then, all changes done to each file are tracked by analyzing the source code repository. Each change to a file will produce a new version of that file. The objective of checking out all file versions is to analyze the source code comments in each version file that indicate SATD. Once the SATD is identified in all file versions, we consider the first file version that contains the SATD as the file version that introduces the SATD.

Step 2: Extracting Source Code Comments. The open source library ScrML [32] is used to parse the source code and extract the comments and the related information, such as the line that each comment starts, finishes and the type of comment (i.e., Javadoc, Line, or Block). Since not all comments can contain SATD [6], [12], we exclude irrelevant comments by applying the heuristic rules mentioned in prior work [13], [16]: (1) remove automatically generated comments with fixed format (i.e., auto-generated constructor stubs, auto-generated methods stubs and auto-generated catch blocks), which are inserted as part of code snippets by the IDE to generate constructors, methods and try catch blocks are removed; (2) remove commented source code fragments since they do not contain SATD. (3) Multiple single line comments that are related to each other are grouped into a block comment. (4) Javadoc and licence comments are removed unless they contain at least one task annotation (i.e., “TODO:”,“FIXME”, or “XXX:”) [16], [33].

Step 3: TD-Introducing Change Identification. In this step, we need to identify SATD comments extracted from step 2 first. To do so, the first author manually examined each comment and classified the comments based on whether the comment is a SATD comment or not. To mitigate personal bias, we take a stratified sample of the full dataset, which is a sample that achieves a confidence level of 99 percent and a confidence interval of 5 percent. Then we invited another independent Ph.D student at Zhejiang University to classify the stratified sample of the comments and measured the level of agreement between the two manual classifiers. We find a high level of agreement with a Cohen’s Kappa coefficient [34] of +0.75, which shows substantial agreement among different labellers. Thus, we are confident in the classification of the provided dataset.

Once the SATD comment has been identified, we can identify their corresponding TD-introducing changes by tracking the comment in the change history. In particular, each change made to a file produces a different version of that file, and by extracting them we can analyze each file version looking for comments that indicate SATD. Once we identify all file versions, we consider the first available file version that contains the self-admitted technical debt as the file version that introduced the self-admitted technical debt [17]. Table 1 lists the number of TD-introducing changes. From the table, we can notice that all projects have a high imbalance class distribution [35], i.e., the total ratio of TD-introducing changes is 2.69 percent on average across 7 projects. In practice, it is difficult to determine TD-introducing changes through manual analysis due to the class imbalance phenomenon, because we may need to check many changes until we capture a TD-introducing one. This will cost a lot of inspection effort.

3 Empirical Study Setup

In this section, we present the setup of our empirical study. First, we mention the research questions that we are interested to answer. Then, we present the 25 studied features, which are grouped into three dimensions. Next, we present the approach used in this study. Finally, we present the validations settings and evaluation measures used to evaluate our technique.

3.1 Research Questions

We formalize our study with the following three research questions:

RQ1: Can we effectively determine the changes that introduce TD? In this RQ, we evaluate the effectiveness of the proposed model in determining changes that introduce technical debt. To answer this RQ, we will conduct an empirical study to evaluate proposed model on 7 open source projects. Additionally, we compare our performance with several baselines, i.e., Random Guess (RG) determination model and determination based on commit messages.

RQ2: Which dimension of features are most important in determining TD-introducing changes? In the second RQ, we would like to know which dimension of features are most discriminative and whether all the three dimensions are necessary. To answer this RQ, we build three different models based on three dimensions of features, i.e., diffusion, history and message, respectively. Except for the features used, the other configurations of the model are kept the same.

RQ3: How effective are our models when varying levels of inspection effort are allocated/inspected? Prior effort-aware studies showed that the effectiveness of their prediction models may vary under different effort allocations. Therefore, we also examine the effectiveness of our models under varying effort allocations as well. By default, we set the percentage of changed LOC to inspect as 20 percent to compute cost-effectiveness. Then, we compute the cost-effectiveness of our model and baselines while varying the percentages of changed LOC to inspect.

3.2 Features Studied

In total, we extract 25 change features divided into three unique dimensions: diffusion, history and message. All these features are derived from the source code control system's repository (e.g., Git). Table 2 presents a summary of the extracted features. We decided to focus on these features since:

(1) Prior work shows that they perform well for predicting defective changes [23], [31], [37], [38], [39]. These features describes the complexity of a change or the history of changed files, such as diffusion and history dimensions. We conjecture that these features also have an impact on introducing technical debt.

(2) Prior work on SATD shows that these features have an impact on TD-introducing, such as message dimension which can indicate the activity of change [17].

In the following sections, we will present the details of the features in each dimension.

Diffusion. The diffusion of a change is one of the most important dimension for predicting defective changes [23]. We conjecture that the diffusion dimension can also be leveraged to determine the likelihood of technical debt introduction. Totally, we extract 16 change features in this dimension as listed in Table 2.

In detail, *NS* represents the number of modified subsystems and *ND* represents the number of modified directories. We use the root directory name as the subsystem name and the directory name to identify directories. For example, if a change modifies a file with the path "camel-core/src/main/java/org/apache/camel/Body.java", then the subsystem is "camel-core", the directory name is "camel-core/src/main/java/org/apache/camel". *Entropy* aims to measure the distribution of the change across the different files [40]. We compute entropy of a change as: $H(P) = -\sum_{k=1}^{n}(p_k * log_2 p_k)$, where probability $p_k \geq 0$ and it indicates the proportion that $file_k$ is modified in a change (i.e., modified lines in $file_k$ respects to total modified lines of a change), thus, $(\sum_{k=1}^{n} p_k) = 1$. For example, a change modifies three files, A, B, and C with modified lines 30, 20, and 10, respectively, The Entropy is measured as 1.46 by using the formula: $(= -\frac{30}{60} log_2 \frac{30}{60} - \frac{20}{60} log_2 \frac{20}{60} - \frac{10}{60} log_2 \frac{10}{60})$. Changes with higher entropy have a larger spread which may have larger likelihood for introducing technical debt. Lines of code added (i.e., *LA*) and lines of code deleted (i.e., *LD*) that describes the size of a change. They can be directly measured from source control repository. Prior study may also divide them into size group. Note that we measure *NS, ND, Entroyp, LA and LD* by following Kamei et al.'s work [23].

Number of files added, modified, deleted, renamed or copyied (i.e., *FA, FM, FD, FR* and *FC*) aim to measure different activities on files that are also used in prior studies [41], [42]. We conjecture that changes touching more files are more likely to introduce technical debt. Number of low, medium, high, and crucial significance level code changes (i.e., *LCC, MCC, HCC* and *CCC*) aim to measure a fine-grained activities for each source code change (one software change may contain many source code changes). The significance level expresses how strongly a code change may impact other source code entities and whether a code change may be functionality modifying or functionality preserving [36]. We adopt the significance level for each code change proposed by Fluri et al. [36], [43]. They presented a taxonomy of code changes according to tree edit operations in the abstract syntax tree [36], [43]. As a result, they classify each code change type with a significance level (i.e., low, medium, high and crucial) that expresses how strong a code change may impact other source code entities (i.e., how likely other source code entities have to be changed). For example, code changes in a method

TABLE 2
Studied Features

Dimension	Name	Definition	Rationale
	NS	Number of modified subsystems	We conjecture that changes touching more subsystems are more likely to introduce technical debt.
	ND	Number of modified directories	We conjecture that changes touching more directories are more likely to introduce technical debt.
	Entropy	Distribution of modified code across each file	We conjecture that changes with high entropy are more likely to introduce technical debt, since a developer will have to recall and track more scattered changes across each file.
Diffusion	LA	Lines of code added	We conjecture that changes touching more lines of code are more likely to introduce technical debt.
	LD	Lines of code deleted	
	FA	Number of files added	
	FM	Number of files modified	We conjecture that changes touching more files are more likely to introduce technical debt.
	FD	Number of files deleted	
	FR	Number of files renamed	
	FC	Number of files copied	
	LCC	Number of low significance level code changes	The significance level expresses how strongly a change may impact other source code entities and whether a change may be functionality modifying or functionality preserving [36]. We conjecture that changes with more high and crucial significant code changes are more likely to introduce technical debt
	MCC	Number of medium significance level code changes	
	HCC	Number of high significance level code changes	
	CCC	Number of crucial significance level code changes	
	language_num	number of modified programming languages in this change	We conjecture that changes with more languages are more likely to introduce technical debt.
	file_type_num	number of modified file types in this change	We conjecture that changes touching more file types are more likely to introduce technical debt.
History	NDEV	Number of developers that changed the modified files	We conjecture that changed files touched by more developers before are more likely to introduce technical debt, since different developers have different design thoughts and code styles.
	NUC	Number of unique changes to the modified files before	We conjecture that lager NUC changs are more likely to introduce technical debt, since a developer will have to recall and track many previous changes.
	EXP	Developer experience	The experience of developers has an impact on introducing TD [12].
Message	msg_length	Message length: number of words in the message	Message contains purpose of this change. Past study has found that TD removal has a correlation with change purposes [17]. We conjecture changes with different purposes have an impact on TD-introducing.
	has_bug	Whether message of this change contains word “bug”	
	has_feature	Whether message of this change contains word “feature”	
	has_improve	Whether message of this change contains word “improve”	
	has_document	Whether message of this change contains word “document”	
	has_refactor	Whether message of this change contains word “refactor”	

body are considered to have a low or medium significance level, whereas code changes on the interface of a class have a high or crucial significance level [43]. We conjecture that the significance level may have a impact on introducing technical debt. We measure *FA, FM, FD, FR, FC, LCC, MCC, HCC* and *CCC* by adopting ChangeDistiller [36].

Additionally, one change may modify different types of files, (i.e., they have different extensions) and written in different programming languages. We use *file_type_num* and *language_num* to measure the unique number of file types and language types. In terms of *file_type_num*, we count the number of file extensions in a change. In terms of *language_num*, we match the file extensions with specific languages' file extensions. In detail, we consider "Java", "C/C++", "Python", "Javascript", "ruby", "bash", "php", and "html". Note that number of file extensions is not equal to number of languages, since a change may modify documents, images, or configuration files which is not related to programming languages. Also, a programming language such as C++ might use multiple types of file extensions such as "c", "h" and "cpp".

History. Features in the history dimension aim to measure historical information related to changed files and the developer that submit the change. For example, *NDEV* indicates number of developers that changed the modified files before. Higher *NDEV* means that the changed files are touched by more developers before. *NUC* indicates number of unique changes to the modified files before. *EXP* indicates number of previous submitted changes of the developer who make the current change. Higher *EXP* means the developer have higher experience. Prior study may also divide it into experience dimension. Note that we measure *NDEV*, *NUC* and *EXP* by following Kamei et al.'s work [23].

Message. Features in the message dimension aim to extract useful information from change logs (i.e., message written by developer that submits the change). Prior studies found that message can indicate the purpose of a change [44], [45]. We group them into six categories: fixing bug, adding feature, improvement, documentation, refactoring and other [46]. We use *has_bug*, *has_feature*, *has_improve*, *has_document* and *has_refactor* to represent the purpose of a change. We measure them by simply checking whether the message contains the related words as Table 2 shows. Additionally, we use *msg_length* to represent the length of message by counting the number of words.

3.3 Approach

We use the extracted change features to characterize a software change. Then, we train a classifier on our extracted features to determine whether a newly submitted change introduces TD. By default, we adopt Random Forest (RF) to construct the determination model and use the implementation in Weka [47]. Random Forest is an ensemble approach that is specifically designed for decision tree classifier [48]. The basic idea behind random forest is to combine multiple decision trees for classification. Each decision tree is built by using a random subset of the extracted features. The advantages of random forest are: 1) it is generally highly accurate and feature importance can be generated automatically; 2) Since random forest unifies many trees that are learned differently, it can mitigate the overfitting problem and is not sensitive to outliers.

3.4 Validation Settings

To validate we use the widely used method 10-fold cross-validation. We perform 10 times stratified 10-fold cross-validation. In each stratified 10-fold cross validation, we randomly divide the dataset into ten folds by using stratified random sampling. The objective of the stratified random sampling is to keep the class distribution of each fold the same as the original dataset. Then, nine folds are used to train the classifier, while the remaining one fold is used to evaluate the performance. This process is repeated 10 times, so that each fold is used exactly once as the testing set. We perform 10-fold cross validation 10 times to reduce the bias due to random training data selection [31]. As a result, there are 100 effectiveness values for each project, and we present the average of the 100 values.

3.5 Performance Measures

In this study, we use AUC and Cost-effectiveness to evaluate the effectiveness of the proposed model.

AUC. AUC represents the area under the receiver operating characteristic (ROC) curve. In the ROC curve, the true positive rate (TPR) is plotted as a function of the false positive rate (FPR) across all thresholds. The value of AUC ranges from 0 to 1, and higher AUC values means better performance. AUC is also widely used in many software engineering studies [42], [49], [50]. The AUC of 0.7 is considered as promising performance [49], [50]. In summary, we choose AUC as our performance measure for the following reasons:

(1) *Threshold Independent.* AUC is a threshold independent measure [51]. A threshold represents the likelihood threshold for deciding an instance is classified as positive or negative. Usually, the threshold is set as 0.5 and other performance measures for a classifier such as precision, recall and F1-score rely on the determination of threshold. However, we may need to change the threshold in some cases, such as the class imbalance case. We use AUC to avoid the threshold setting problem since AUC measures the classification performance across all thresholds (i.e., from 0 to 1).

(2) *Robust Towards Class Distribution.* AUC is robust towards class distribution [49]. Other performance measures such as precision, recall, and F1-score are highly affected by class distribution, it might make it difficult to fairly compare two models [50], [52]. Unlike them, AUC is insensitive to class distribution [49]. Thus, it is recommended as the primary indicator for comparative studies [49].

(3) *Statistical Interpretable.* AUC has a statistical interpretation [49]. In our context, it evaluates the possibility that a classifier ranks a randomly chosen TD-introducing change higher than a randomly chosen not TD-introducing change. Since our motivation is to determine TD-introducing change and prioritize the inspecting tasks, AUC is an appropriate measure to evaluate the performance of our approach and the baseline method.

Cost-Effectiveness. Cost-effectiveness aims to measure the performance considering the limited inspecting resources. It has been widely used for evaluating software defect prediction models [23], [25], [26], [31], [53]. The main idea is to simulates the practical usage of the proposed model. In practice, it is important to take into account the cost-effectiveness of using our determination model to focus on verification and validation activities. Cost effectiveness is an appropriate measure that can evaluate how effective a model is to prioritize changes that are assigned to inspect. In our context, we consider the effort required to inspect those changes determined as TD-introducing. Due to the limited resources, developers can only inspect a limited number of software changes, and they would expect to identify as many TD-introducing changes as possible. Thus, following prior studies [23], [26], [31], the cost-effectiveness in our study denotes the recall of TD-introducing changes when using 20 percent of the entire effort required to inspect all changes to inspect the top ranked changes. And the total number of lines modified by a change ($LA + LD$) as a measure of the effort required to inspect a change.

4 Results

In this section, we aim to answer the aforementioned three research questions. In RQ1, we evaluate the performance of our proposed determination model on seven open source projects and compare it with a baseline model. In RQ2, we present the results of three determination models built using three dimensions of features, i.e., diffusion, history, and message dimensions. In RQ3, we present the cost-effectiveness analysis when different percentages of LOC are inspected.

4.1 RQ1: Can We Effectively Determine the Changes That Introduce TD?

Motivation. As shown in previous work [7], [16], technical debt can be determined from source code comments in source code files. However, there is no automatic way to determine whether a *change* introduces TD. The differences between TD determination at the file-level and that at the change-level lie in two aspects : first, the research object is different. TD determination on file-level aims to investigate source code comments, while TD determination on change-level aims to investigate software changes. Understanding and tracing the TD-introducing changes can help to address TD and mitigate introducing TD. Second, the employing time on the development phase is different. TD determination at the file-level can be employed before a product releases. It can serve as a technical debt management step before a release. TD determination at the change-level can be employed when each change is submitted. It can serve as a continuous activity of technical debt management.

Approach. To answer this research question, we conduct an empirical study on seven open source projects. We implement our proposed model on top of the Weka tool [47]. By default, we built our model using a Random Forest classifier and adopt 10 times stratified 10-fold cross validation to estimate the accuracy of our model as default. In this way, there will be 100 effectiveness values for each project. We report the average value and perform statistical test on the 100 effectiveness values.

In order to compare the effectiveness of the proposed model with other methods, we implement four baselines:

Baseline 1, Random Guess (RG). RG is usually adopted as a baseline when there is no previous method for addressing the same research question [54]. In random determination, the model randomly determines TD-introducing changes. In terms of the performance measures, the AUC of RG is 0.5 [54]. In terms of cost-effectiveness, it only relates to the order of instances in the testing set. Thus, we sort the changes in testing set randomly and we repeat the random sorting 10 times to get the average cost-effectiveness in RG.

In addition, since change messages may also indicate that a change introduces technical debt or not, we design the following text classification baselines based on commit messages using Naive Bayes (NB), Naive Bayes Multinomial (NBM) and Random Forest. Therefore, the other baselines are:

Baseline 2, Naive Bayes classification based on Change Messages (NBCM). Naive Bayes is a simple probabilistic classifier based on applying Bayes' theorem with strong independence assumption between features. It assigns class labels to problem instances, represented as vectors of feature values, where the class labels are drawn from some finite set. An advantage of Naive Bayes is that it only requires a small number of training data to estimate the parameters necessary for classification [55].

Baseline 3, Naive Bayes Multinomial classification based on change Messages (NBMCM). Naive Bayes multinomial is one of the variants of the Naive Bayes algorithm, which builds a classifier based on multinomially distributed data [56]. We adopt NB and NBM as baseline classifiers since they are simple text classification techniques that have been used in many software text analysis studies [16], [54], [57].

Baseline 4, Random Forest classification based on Change Messages (RFCM). We adopt RF as a baseline classifier since it is used as the default classifier in our determination model.

Baselines 2, 3 and 4 (i.e., NBCM, NBMCM and RFCM) are built in the following steps. First, we preprocess all the change messages by tokenization, stop-word removal and stemming [16]. Tokenization aims to break a stream of text up into words, phrases, symbols, or other meaningful elements called tokens. In our experiment, we only keep tokens that contain English letters and convert all words to lowercase. Stop-word removal aims to remove words that are used often and carry little meaning, such as "I", "to", "the", "of". Stemming aims to reduce inflected (or sometimes derived) words to their word stem, base or root form. We employ the well-known Porter stemmer[9] to reduce a word to its representative root form. Second, we use the resulting textual tokens and count the number of times each token appears to represent each change message. Third, we constructs a classifier based on the textual representation using NB, NBM, and RF respectively.

To investigate whether the improvement of proposed model over the baseline model is statistically significant, we employ the Wilcoxon signed-rank test [58] with a Bonferroni correction [59] at 95 percent significance level. Wilcoxon signed-rank test is a non-parametric hypothesis test used when comparing two related samples, matched samples, or repeated measurements on a single sample to assess whether their population mean ranks differ. Bonferroni correction is used to counteract the problem of multiple comparisons. In addition, we compute Cliff's delta to measure the effect size. Cliff's delta is a non-parametric effect size measure that can evaluate the amount of difference between two approaches. It defines a delta of less than 0.147,

9. http://tartarus.org/martin/PorterStemmer/

TABLE 3
AUC and Cost-Effectiveness for our Model (Ours) Compared with the Baselines

Project	AUC					Cost-effectiveness				
	RG	NBCM	NBMCM	RFCM	Ours	RG	NBCM	NBMCM	RFCM	Ours
Hadoop	0.5	0.75	0.76	0.73	**0.87**	0.19	0.52	0.56	0.49	**0.87**
Log4j	0.5	0.70	0.69	0.73	**0.81**	0.25	0.69	0.66	0.73	**0.89**
Tomcat	0.5	0.70	0.74	0.74	**0.81**	0.22	0.33	0.36	0.42	**0.71**
Camel	0.5	0.72	0.72	0.72	**0.81**	0.31	0.65	0.46	0.54	**0.88**
Gerrit	0.5	0.76	0.60	0.76	**0.76**	0.22	0.03	0.43	0.54	**0.72**
Ant	0.5	0.73	0.71	0.73	**0.85**	0.22	0.40	0.37	0.42	**0.64**
Jmeter	0.5	0.65	0.67	0.67	**0.81**	0.30	0.53	0.55	0.55	**0.93**
Average	*0.5*	*0.72*	*0.70*	*0.73*	*0.82*	*0.24*	*0.45*	*0.48*	*0.53*	*0.80*
Improved	*64%*	*14%*	*17%*	*12%*	–	*232%*	*78%*	*66%*	*53%*	–
W/T/L	*0/0/7*	*0/1/6*	*0/0/7*	*0/1/6*	–	*0/0/7*	*0/0/7*	*0/0/7*	*0/0/7*	–

The best performance values are highlighted in bold. The row "W/T/L" reports the number of projects for which the corresponding model obtains a statistically significantly better, equal, and worse performance than our model.

between 0.147 and 0.33, between 0.33 and 0.474 and above 0.474 as "Negligible", "Small", "Medium", "Large" effect size, respectively [60].

Results. Table 3 presents the AUC and cost-effectiveness values of our determination model (Ours) and four baselines. On average across the seven projects, our model achieves an AUC of 0.82 and cost-effectiveness of 0.80. As shown in the table, our model consistently shows an improvement over the baseline across the 7 projects in terms of AUC and cost-effectiveness. We use "Improved" to represents the improvement ratio, it is computed as $\frac{Ours-baseline}{baseline} * 100\%$. On average, our model improves RG, NBCM, NBMCM and RFCM by 64, 14, 17 and 12 percent in terms of AUC, by 232, 78, 66 and 53 percent in terms of cost-effectiveness, respectively.

The row "W/T/L" in Table 3 reports the number of projects for which the corresponding determination model obtains a significantly better, equal, and worse performance than our model. And Table 4 presents the adjusted p-values and effect sizes according to Cliff's delta. From the table, in terms of AUC, we notice that our approach significantly improves the RG, NBMCM in all the seven projects, and all the effect size are large; our approach significantly improves NBCM and RFCM in six projects, there is no significant difference in Gerrit. In terms of cost-effectiveness, our approach significantly improves the baselines in all the seven projects, and all the effect size are large. Thus, our approach shows statistically significant improvement over the baseline in most of the datasets, and the improvements are substantial.

For each project, our model achieves a promising and better performance than the baseline in terms of AUC and cost-effectiveness. On average across the seven projects, our model achieves AUC of 0.82, cost-effectiveness of 0.80, which significantly improves the baseline approach in a substantial margin in most cases.

4.2 RQ2: Which Dimension of Features Are Most Important in Determining TD-Introducing Changes?

Motivation. In addition to determining TD-introducing changes with high accuracy, we are interested in investigating which features are the best contributors to our determination model. By default, our determination model combines three dimensions of features: diffusion, history and message. They characterize changes from different aspects. Some aspects may be more discriminative for determining TD-introducing changes. For answering this RQ, we can investigate two questions: first, whether our model benefits from all features; second, which dimension is the most discriminative for determining TD-introducing changes.

Approach. We build three determination models by learning on features in each dimension and denote them as the dimension name (i.e., diffusion, history and message). In each model, we keep the classifier (i.e., Random Forest) as the same. We compare their performance by experimenting on the 7 projects and using 10 times stratified 10-fold cross validation setting. In each cross validation, in order to

TABLE 4
Adjusted P-Values and Cliff's Delta Comparing AUC and Cost-Effectiveness Scores for Our Approach with Baselines

Project	AUC				Cost-effectiveness			
	RG	NBCM	NBMCM	RFCM	RG	NBCM	NBMCM	RFCM
Hadoop	1.00 (L)***	0.99 (L)***	0.98 (L)***	1.00 (L)***	1.00 (L)***	1.00 (L)***	1.00 (L)***	1.00 (L)***
Log4j	1.00 (L)***	0.62 (L)***	0.62 (L)***	0.50 (L)***	1.00 (L)***	0.65 (L)***	0.70 (L)***	0.53 (L)***
Tomcat	1.00 (L)***	0.98 (L)***	0.84 (L)***	0.86 (L)***	1.00 (L)***	1.00 (L)***	0.99 (L)***	0.96 (L)***
Camel	1.00 (L)***	0.99 (L)***	0.99 (L)***	0.98 (L)***	0.99 (L)***	0.67 (L)***	0.88 (L)***	0.86 (L)***
Gerrit	1.00 (L)***	–	0.87 (L)***	–	1.00 (L)***	0.96 (L)***	0.88 (L)***	0.71 (L)***
Ant	1.00 (L)***	0.97 (L)***	0.98 (L)***	0.96 (L)***	1.00 (L)***	0.88 (L)***	0.91 (L)***	0.80 (L)***
Jmeter	1.00 (L)***	1.00 (L)***	1.00 (L)***	1.00 (L)***	1.00 (L)***	0.91 (L)***	0.97 (L)***	0.97 (L)***

****p < 0.001, **p < 0.01, *p < 0.05, –p > 0.05; L: Large effect size according to Cliff's delta.*

TABLE 5
Performance for Difference Dimension of Features

Project	AUC				Cost-effectiveness			
	Diffusion	History	Message	All features	Diffusion	History	Message	All features
Hadoop	0.83	0.63	0.56	**0.87**	0.81	0.56	0.85	**0.87**
Log4j	0.77	0.56	0.58	**0.81**	0.78	0.43	0.72	**0.89**
Tomcat	0.79	0.65	0.59	**0.81**	0.65	0.56	0.64	**0.71**
Camel	0.79	0.69	0.63	**0.81**	0.84	0.69	**0.92**	0.88
Gerrit	0.70	0.60	0.52	**0.76**	0.62	0.36	0.48	**0.72**
Ant	0.83	0.64	0.59	**0.85**	0.57	0.43	0.52	**0.64**
Jmeter	0.76	0.60	0.52	**0.81**	0.85	0.65	0.88	**0.93**
Average	*0.78*	*0.62*	*0.57*	*0.82*	*0.73*	*0.53*	*0.72*	*0.80*

The best performance between three models built on three dimensions is underlined. The column "All features" represents the performance of our model that using all dimensions of features. And the performance highlighted in bold is the best among the four columns.

confirm a fair comparison, we keep the training and testing sets the same as in the comparison.

Additionally, in order to investigate whether the difference between our determination model (i.e., learning on all features) and the three determination models that learn on each dimension of features is statistically significant, we adopt the Wilcoxon signed-rank test [58] with a Bonferroni correction [59] at 95 percent significance level and compute the Cliff's delta to measure the effect size. *Results.* Table 5 presents the results of AUC and cost-effectiveness values. We list the performance of three models built on each dimension of features (in column "Diffusion","History","Message", respectively) and the model built on all features (in column "All features"). The best performance between three dimensions is underlined. And the performance highlighted in bold is the best among the four columns.

From Table 5, we see that the most discriminative dimension is "Diffusion" in terms of AUC. On average across 7 projects, the model "Diffusion" achieves AUC of 0.78 that is the best among the three dimensions. Additionally, our model (using all features) achieves the best AUC in all the projects compared with the other three models. In terms of cost-effectiveness, the best dimension varies in different projects. In projects Log4j, Tomcat, Gerrit and Ant the best dimension is "Diffusion' among the three dimensions. In projects Hadoop, Camel, and Jmeter, the best dimension is "Message". Our model (using all features) achieves the best cost-effectiveness in six projects, and it shows worse than "Message" in Camel. On average across 7 projects, our model achieves the best in terms of both AUC and cost-effectiveness.

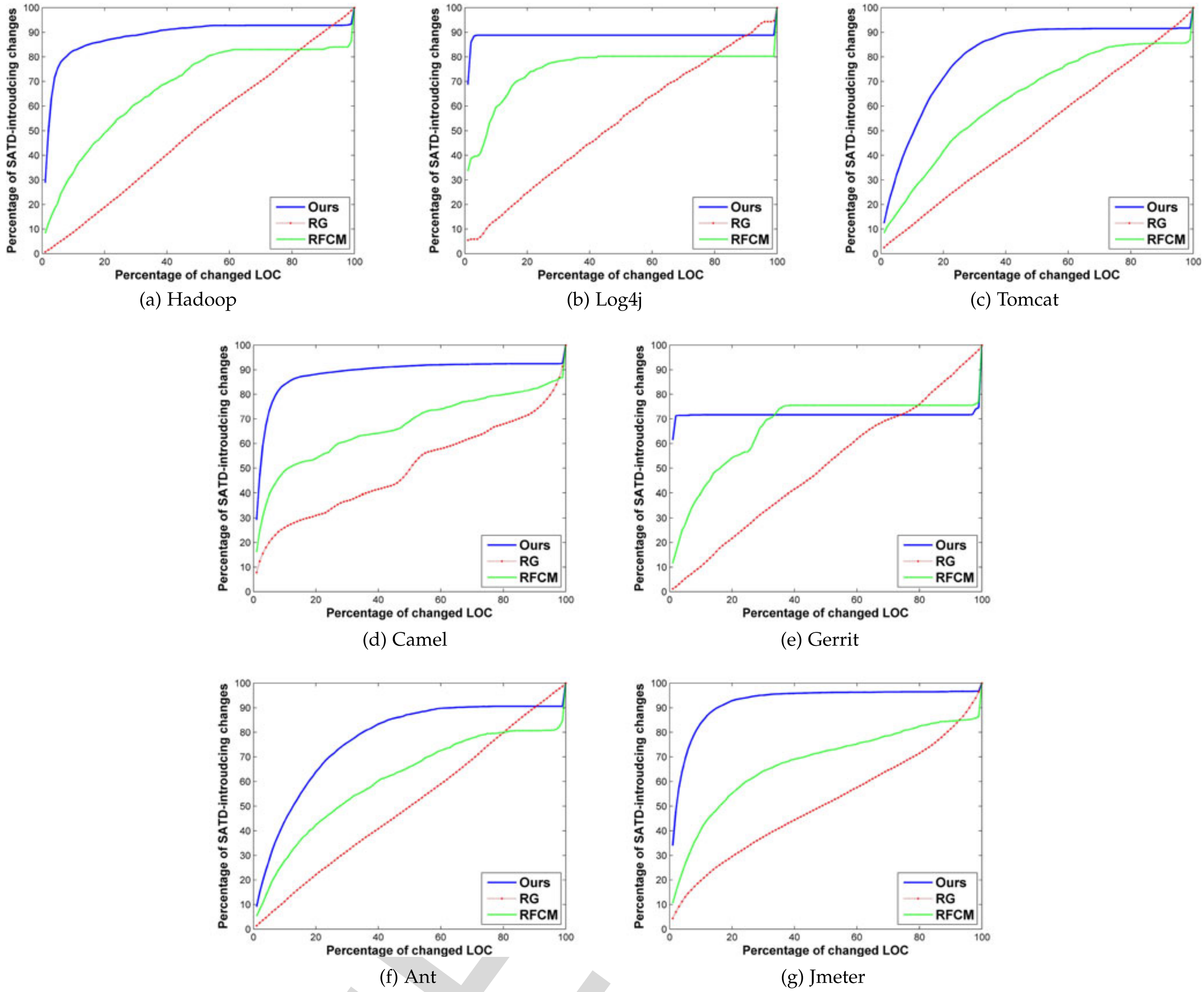
In order to test whether the differences between our model and the models built on subset features are statistically significant, Table 6 presents the results of Wilcoxon signed-rank test with Bonferroni correction. We also present Cliff's delta to measure the effect size and highlight the non-negligible effect size in bold. The row "W/T/L" reports the number of projects for which the corresponding model obtains a significantly better, equal, and worse performance than our model (using all features).

From the table, we can find that the improvements of our model over all the three models built on three dimensions are statistically significant (p-value < 0.05) in each project in terms of AUC. And all the effect sizes are non-negligible. In terms of cost-effectiveness, the results show that our model significantly outperforms "Diffusion", "History" and "Message" in 7, 7, 6 projects with non-negligible effect size. There is no significant difference (p-value > 0.05) in Camel compared with "Message" dimension. Thus, the statistical test results indicate that our model shows a significant improvement in terms of AUC, and shows comparable or significant better performance in terms of cost-effectiveness over the three models built on three dimension features. This also suggests that we should use all the three dimensions of change features when applying our model.

TABLE 6
Adjusted P-Values and Cliff's Delta Comparing AUC and Cost-Effectiveness Scores for Our Approach with the Models Built Using a Dimension of Features

Project	AUC			Cost-effectiveness		
	Diffusion	History	Message	Diffusion	History	Message
Hadoop	**0.61 (L)*****	**1.00 (L)*****	**1.00 (L)*****	**0.44 (M)*****	**1.00 (L)*****	**0.15 (S)***
Log4j	**0.28 (S)*****	**0.96 (L)*****	**0.88 (L)*****	**0.48 (L)*****	**0.96 (L)*****	**0.57 (L)*****
Tomcat	**0.40 (M)*****	**1.00 (L)*****	**1.00 (L)*****	**0.40 (M)*****	**0.85 (L)*****	**0.47 (M)*****
Camel	**0.58 (L)*****	**1.00 (L)*****	**1.00 (L)*****	**0.54 (L)*****	**0.82 (L)*****	–
Gerrit	**0.47 (M)*****	**0.90 (L)*****	**0.97 (L)*****	**0.41 (M)*****	**0.95 (L)*****	**0.81 (L)*****
Ant	**0.27 (S)*****	**1.00 (L)*****	**1.00 (L)*****	**0.34 (M) *****	**0.83 (L) *****	**0.51 (L)*****
Jmeter	**0.82 (L)*****	**1.00 (L)*****	**1.00 (L)*****	**0.73 (L) *****	**0.98 (M) *****	**0.42 (M) *****
W/T/L	*0/0/7*	*0/0/7*	*0/0/7*	*0/0/7*	*0/0/7*	*0/1/6*

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$, $-p > 0.05$.
"L", "M" and "S" represent "Large", "Medium" and "Small" effect size according to Cliff's delta respectively.



(a) Hadoop (b) Log4j (c) Tomcat

(d) Camel (e) Gerrit

(f) Ant (g) Jmeter

Fig. 1. Cost-effectiveness graphs of 7 projects.

> *"Diffusion" is the most discriminative dimension among the three dimensions of features for determining TD-introducing changes. However, using all the three dimensions of change features is better when applying our determination model.*

4.3 RQ3: How Effective Are Our Models When Varying Levels of Inspection Effort Are Allocated/Inspected?

Motivation. Although one would like to capture all of the TD-introducing changes, there is always a conflicting interest between the amount of effort (changed LOC to inspect) one allocates and the amount of TD-introducing changes they can capture. Therefore, it is important to investigate the cost-effectiveness of our models. By default, we set the percentage of changed LOC to inspect as 20 percent to compute model cost-effectiveness, i.e., how many TD-introducing changes can be captured when inspecting 20 percent of the changed LOC in testing set according to the output of our model. Additionally, we are also interested to investigate the cost-effectiveness of our model and the comparison with baselines when different percentages (from 1 to 100) of changed LOC are inspected. The experiment is conducted using the same dataset, features and settings with RQ1, the only difference is that we vary the percentage of changed LOC to inspect in this RQ.

Approach. To answer this RQ, we plot cost-effectiveness graphs that show the percentages of TD-introducing changes that can be detected by inspecting different percentages of changed LOC. In detail, we set the percentages from 1 to 100. As a result, there are 100 cost-effectiveness values in each plot. Note that for each percentage, we use the average effectiveness value of 100 values produced by 10 times 10-fold cross validation. In addition, we also plot the cost-effectiveness graphs of two baselines, i.e., RG and RFCM. Since the result of RQ1 shows that RFCM outperforms other two baselines based on change messages (i.e., NBCM and NBMCM), we do not show the cost-effectiveness graphs of NBCM and NBMCM.

Results. Fig. 1 presents cost-effectiveness graphs for 7 projects of our model (Ours), RG and RFCM. We can notice that our model is better than the baselines for a wide range of percentages of changed LOC to inspect. For Hadoop and Tomcat, the percentage range for which our model achieves better performance is from 1 to 93 percent. For Log4j and Ant, the percentage range for which our model achieves better performance is from 1 to 90 percent. For Camel and Jmeter, the percentage range for which our model achieves better performance is from 1 to 99 percent. For Gerrit, the percentage range for which our model achieves better performance is from 1 to 33 percent. We notice that our model performs worse than RFCM in the higher percentages of LOC to inspect in Gerrit, i.e., greater than 33 percent. However, in practice, developers would not inspect such a high number of LOC due to limited project budget and tight project schedule.

> *Our model can detect more TD-introducing changes than baseline for a wide range of percentages of changed LOC to inspect, hence it is cost effective.*

5 Discussion

As shown in previous sections, our model can achieve a promising performance in determining TD-introducing changes. However, there are some other observations worth for further investigation. In this section, we will report other observations including four aspects: (1) what change features are more important that impact TD introduction? (Section 5.1) (2) what is the impact of using other underlying classifiers? (Section 5.2) (3) how effective of our model when using time-wise validation setting? (Section 5.3) (4) how effective of our model when using cross-project setting? (Section 5.4) (5) other general discussions. (Section 5.5).

5.1 Investigating the Importance of Features

Motivation. In addition to determining TD-introducing changes with high accuracy, we are also interested in understanding what change features impact TD-introducing the most. There are 25 software change features in our determination model. Being aware of what features impact TD-introducing the most can help to gain a deeper understanding of why developers introduce TD. In addition, for developers, they can know what features they should carefully consider when determining whether or not they submit a TD-introducing change. For researchers, feature importance analysis can encourage them to propose more discriminative features for determining TD-introducing changes.

Approach & Results. Following previous studies [61], [62], [63], we use a four step process:

Step 1: Correlation Analysis. This step aims to reduce collinearity among the features. For each project, this step will compute the correlations among the features by using variable clustering analysis implemented in the R package *Hmisc*.[10] As a result, it will produce a hierarchical overview of the correlations among all the features. The correlated features are grouped into sub-hierarchies. To remove correlated features, we use the same method in the previous study [61]. If the correlations of features in the sub-hierarchy are above 0.7, we randomly select one feature and remove the other features from a sub-hierarchy. And during the random selection, we will following a guideline, i.e., trying to drop the same feature set for all the studied projects [64].

After step 1, we remove 8, 4, 2, 3, 12, 4 and 3 features in Hadoop, Log4j, Tomcat, Camel, Gerrit, Ant and Jmeter, respectively.

TABLE 7
Importance of Features in Hadoop as Ranked According to the Scott-Knott ESD Test

Group	Features	p-value	Cliff's delta
1	NUC	<0.001	-0.130
2	**Entropy**	**<0.001**	**0.409 (Medium)**
3	EXP	>0.05	0.047
4	**FA**	**<0.001**	**0.561 (Large)**
5	**msg_length**	**<0.001**	**0.184 (Small)**
6	**HCC**	**<0.001**	**0.297 (Small)**
7	NS	<0.01	0.078
8	FD	<0.001	0.069
9	**language_num**	**<0.001**	**0.195 (Small)**
10	FR	<0.001	0.052

The second and third columns show P-values, Cliff's Delta for the features. The features with non-negligible effect sizes are in bold.

Step 2: Redundancy Analysis. After reducing the collinearity among the features, this step aims to remove redundant features that do not have a unique signal relative to the other features. In this step, we use the redun function in the rms[11] R package.

After step 2, none of the remaining features are redundant in Tomcat, Camel, Gerrit, Ant and Jmeter. In Hadoop, there is one redundant feature, i.e., "ND". In Log4j, there are two redundant features, i.e., "FD" and "FA". Therefore, we remove one more feature in Hadoop and remove two more features in Log4j. After this step, there are 16, 19, 23, 22, 13, 21 and 22 features remaining in Hadoop, Log4j, Tomcat, Camel, Gerrit, Ant and Jmeter, respectively.

Step 3: Important Feature Identification. This step aims to determine the importance of each feature. We use the bigrf[12] R package to implement. It leverages a random forest model with 10-times stratified 10-fold cross-validation to investigate the most important features. The feature importance evaluation is based on an internal error estimate of a random forest classifier, which is called "Out Of the Bag" (OOB) estimate [65]. The key idea behind it is to check whether the OOB estimate will be reduced significantly or not when features are randomly permuted one by one.

In each run of 10-fold cross-validation, we have 10 importance values for each feature. To determine which of the features are the most important, we apply Scott-Knott Effect Size Difference (ESD) test for the importance values taken from all 10 runs of 10-fold cross-validation [64], [66], [67]. Note that Scott-Knott ESD test is different from Scott-Knott test [68]. Scott-Knott test assumes that the data is normally distributed. This might cause that the created groups are trivially different from one another. Scott-Knott ESD test can correct the non-normal distribution of an input dataset and merge any two statistically distinct groups (i.e., the groups have a negligible effect size) into one group.

After step 3, Tables 7, 8, 9, 10, 11, 12 and 13 present top 10 features as ranked according to Scott-Knott ESD test results in Hadoop, Log4j, Tomcat, Camel, Gerrit, Ant and Jmeter, respectively.[13]

10. http://cran.r-project.org/web/packages/Hmisc/index.html

11. https://cran.r-project.org/web/packages/rms/rms.pdf
12. http://cran.r-project.org/web/packages/bigrf/bigrf.pdf
13. Full list of important features can be found in Appendices, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2018.2831232.

TABLE 8
Importance of Features in Log4j

Group	Features	p-value	Cliff's delta
1	**LA**	**< 0.001**	**0.570 (Large)**
2	EXP	> 0.05	−0.057
3	Entropy	> 0.05	0.105
	msg_length	**< 0.001**	**0.315 (Small)**
4	**NUC**	**< 0.001**	**−0.265 (Small)**
5	LD	> 0.05	0.028
6	**MCC**	**< 0.01**	**0.168 (Small)**
7	file_type_num	> 0.05	0.008
8	CCC	< 0.01	0.113
9	HCC	< 0.01	0.128
10	NS	> 0.05	0.071

TABLE 9
Importance of Features in Tomcat

Group	Features	p-value	Cliff's delta
1	**LA**	**< 0.001**	**0.585 (Large)**
2	EXP	> 0.05	−0.033
3	NUC	< 0.001	0.113
4	**Entropy**	**< 0.001**	**0.337 (Medium)**
5	**msg_length**	**< 0.001**	**0.273 (Small)**
6	**LD**	**< 0.001**	**0.313 (Small)**
7	**LCC**	**< 0.001**	**0.282 (Small)**
8	**MCC**	**< 0.001**	**0.210 (Small)**
9	**ND**	**< 0.001**	**0.299 (Small)**
10	**FA**	**< 0.001**	**0.216 (Small)**

TABLE 10
Importance of Features in Camel

Group	Features	p-value	Cliff's delta
1	**Entropy**	**< 0.001**	**0.493 (Large)**
2	EXP	> 0.05	−0.027
3	NUC	< 0.001	0.113
4	**msg_length**	**< 0.001**	**0.335 (Medium)**
5	**LCC**	**< 0.001**	**0.457 (Medium)**
6	**LD**	**< 0.001**	**0.351 (Medium)**
7	**FA**	**< 0.001**	**0.427 (Medium)**
8	**MCC**	**< 0.001**	**0.380 (Medium)**
9	**ND**	**< 0.001**	**0.457 (Medium)**
10	**CCC**	**< 0.001**	**0.284 (Small)**

TABLE 11
Importance of Features in Gerrit

Group	Features	p-value	Cliff's delta
1	NUC	> 0.05	0.034
2	EXP	< 0.05	−0.122
	LA	**< 0.001**	**0.415 (Medium)**
3	**msg_length**	**< 0.001**	**0.531 (Large)**
4	**CCC**	**< 0.001**	**0.156 (Small)**
5	FR	> 0.05	−0.011
6	language_num	< 0.05	0.097
7	has_bug	> 0.05	0.029
8	FC	> 0.05	−0.005
9	has_refactor	< 0.001	0.029
10	has_feature	< 0.01	0.031

TABLE 12
Importance of Features in Ant

Group	Features	p-value	Cliff's delta
1	**LA**	**< 0.001**	**0.701 (Large)**
2	**EXP**	**< 0.001**	**−0.292 (Small)**
3	**msg_length**	**< 0.001**	**0.436 (Medium)**
4	**Entropy**	**< 0.001**	**0.401 (Medium)**
5	**NUC**	**< 0.001**	**0.231 (Small)**
6	**LD**	**< 0.001**	**0.404 (Medium)**
7	**MCC**	**< 0.001**	**0.428 (Medium)**
8	**FA**	**< 0.001**	**0.364 (Medium)**
9	**ND**	**< 0.001**	**0.392 (Medium)**
10	**HCC**	**< 0.001**	**0.308 (Small)**

TABLE 13
Importance of Features in Jmeter

Group	Features	p-value	Cliff's delta
1	EXP	> 0.05	0.026
2	**LA**	**< 0.001**	**0.472 (Medium)**
3	NUC	> 0.05	0.025
4	**Entropy**	**< 0.001**	**0.158 (Small)**
5	**LCC**	**< 0.001**	**0.404 (Medium)**
6	msg_length	< 0.001	0.124
7	**LD**	**< 0.001**	**0.278 (Small)**
8	**MCC**	**< 0.001**	**0.323 (Small)**
9	**ND**	**< 0.001**	**0.184 (Small)**
10	**HCC**	**< 0.001**	**0.188 (Small)**

Step 4: Effect of Important Features. This step aims to determine the effect of important features. To understand the impact of each feature, we compare the feature values of the remaining features between TD-introducing and not TD-introducing changes. We apply the Wilcoxon rank-sum test [69] at 95 percent significance level to analyze the statistical significance of the difference between TD-introducing and not TD-introducing changes. Then, to show the effect size of the difference between the two groups, we calculate the Cliff's Delta. The effect sizes can be positive or negative. A higher level of a feature with a positive effect can increase the likelihood of a change being TD-introducing change, while a higher level of a feature with a negative effect can decrease that likelihood.

After step 4, we compute the p-value and effect size to compare the impact of each feature. Tables 7, 8, 9, 10, 11, 12, and 13 preset the p-value and effect size values.

Based on the results shown in these tables, we have the following findings:

(1) "Entropy" are ranked in the top 10 important groups for 6 projects. Most of them have a non-negligible positive effect. This indicates that changes with higher entropy are more likely to introduce TD. The reason is that in a high entropy change, a developer will have to recall and track more scattered changes across each file.

(2) "Msg_length" are ranked in the top 10 important groups for 7 projects. Most of them have a non-negligible positive effect. This indicates that changes with longer message length are more likely to introduce TD. This may be resulted from that developers need more detailed message for describing an TD-introducing change.

TABLE 14
Performance of Different Classifiers

Project	AUC					Cost-effectiveness				
	NB	NBM	DT	KNN	RF	NB	NBM	DT	KNN	RF
Hadoop	0.79	0.80	0.60	0.57	**0.87**	0.40	0.53	0.86	0.84	**0.87**
Log4j	0.70	0.81	0.67	0.52	**0.81**	**1.00**	1.00	0.97	1.00	0.89
Tomcat	0.78	0.76	0.69	0.56	**0.81**	0.39	0.57	0.69	0.42	**0.71**
Camel	0.81	0.79	0.66	0.55	**0.81**	0.85	**0.93**	0.90	0.81	0.88
Gerrit	**0.81**	0.74	0.50	0.52	0.76	0.96	0.96	**0.97**	0.97	0.72
Ant	0.85	0.84	0.64	0.59	**0.85**	0.44	0.52	0.50	0.49	**0.64**
Jmeter	0.77	0.75	0.53	0.56	**0.81**	0.78	0.92	0.92	0.92	**0.93**
Average	*0.79*	*0.78*	*0.61*	*0.55*	*0.82*	*0.69*	*0.78*	*0.83*	*0.78*	*0.80*

(3) "LA" and "LD" are ranked in the top 10 important groups across 5 projects. "FA" and "ND" are ranked in the top 10 important groups across 4 projects. And most of them have a non-negligible positive effect. This indicates that larger size changes are more likely to introduce TD. The reason is that larger size changes (i.e., change more number of LOC, more files, or more directories) have a higher chance to introduce TD.

(4) Among code change significant features "LCC", "MCC", "HCC" and "CCC", "MCC" are the most important feature and ranked in the top 10 important groups across 5 projects. And most of them have a non-negligible positive effect. This indicates that changes with more medium significant code changes are more likely to introduce TD. Medium significant code changes mainly consist of condition changes (e.g., loop and if-else changes) [70].

(5) We notice that "EXP" and "NUC" have a non-negligible negative effect in Ant and Log4j, respectively. This indicates that changes with higher value of "EXP" and "NUC" may have less chance for introducing TD. In terms of "EXP", our finding suggests that lower experience developers may tend to introduce TD. In terms of "NUC", our finding suggests that files which are not frequently modified previously may tend to introduce TD.

In summary, the features "Entropy", "msg_length", "LA", "LD", "FA", "ND", and "MCC" are the important features for determining TD-introducing changes.

5.2 Investigating the Impact of Different Classifiers

By default, we use Random Forest as the classifier in our determination model. However, the model can use other classifiers too. In order to investigate the impact of other underlying classifiers, we investigate four more classifiers, namely Naive Bayes, Naive Bayes Multinomial, Decision Tree (DT) and K-Nearest Neighbor (KNN). NB and NBM have been briefly introduced in RQ1. We describe DT and KNN briefly in this section.

Decision Tree (DT). C4.5 is one of the most popular decision tree algorithms [55]. It builds decision trees from a set of training instances using information entropy. Instances are classified by comparing their factors with various conditions captured in the nodes and branches of the tree.

K-Nearest Neighbor (KNN). K-nearest neighbors algorithm (KNN) is a non-parametric method used for classification or regression. In KNN classification, the output is a class membership. An instance is classified by a majority vote of its neighbors. Namely, the instance is assigned to the class most common among its k nearest neighbors [55].

Table 14 presents the performance of different models built on our extracted features using different classifiers. We implement these classifiers on top of Weka and use the default parameter settings [47]. In terms of AUC, the best classifier is random forest on average. We notice that random forest model achieves the best performance compared with other classifiers across six projects. In terms of cost-effectiveness, we notice that the random forest model achieves the best performance compared with other classifiers across four projects, and perform worse than other models in three projects. On average, the best classifier is DT that slightly better than random forest in terms of cost-effectiveness. However, it achieves a very low AUC value as the table shows. Thus, in practice, we suggest to use random forest as the underlying classifier to build our model.

5.3 Investigating the Effectiveness of Time-Wise Evaluation Setting

In the experimental setting of RQ1, we use the 10 times 10-fold cross validation setting to evaluate the effectiveness of our model. However, in practice, developers may use a time-wise (i.e., based on the chronological order of the changes) validation setting. Thus, we want to investigate the effectiveness of our model when using time-wise evaluation setting.

In time-wise validation, for each project, we first rank all changes in chronological order according to the commit date and time. Then, all the changes are divided into n approximately equal parts according to the total number of changes. Each part will have approximately $n/6$ changes. After that, we will build a classification model on part i, and apply it to determine changes in part $i+1$. In this way, there will be $n-1$ effectiveness values for each project. This kind of validation is used to simulate real-life usage of the determination model.

In our evaluation, we first rank all changes in chronological order according to the commit date. Then, all the changes are divided into 6 approximately equal parts. After that, we will build a model on part i, and apply it to predict changes in part i + 1. In this way, there will be 5 effectiveness values for each project. Since we only have 5 AUC and cost effectiveness scores, we do not apply any statistical testing due to the small sample.

Table 15 presents the results of time-wise validation. We also implement the baselines Random Guess and Random Forest classification based on Change Messages (i.e., RG and RFCM as described in RQ1) to compare with our model under the same time-wise evaluation setting. From the table, we notice that our model can also achieves promising performance in terms of AUC and cost-effectiveness. It performs better than baselines across 6 projects in terms of AUC, and 7 projects in terms of cost-effectiveness. Considering the average across 7 projects, our model achieves AUC of 0.75 that improves RG and RFCM by 50 percent and 15 percent respectively. And it achieves cost-effectiveness of 0.65 that improves RG and RFCM by 195 and 63 percent respectively.

TABLE 15
Performance of Time-Wise Validation for Our Model (Ours) Compared with Baselines

Project	AUC			Cost-effectiveness		
	RG	RFCM	Ours	RG	RFCM	Ours
Hadoop	0.5	0.65	**0.83**	0.21	0.44	**0.80**
Log4j	0.5	0.60	**0.74**	0.22	0.37	**0.76**
Tomcat	0.5	0.67	**0.74**	0.21	0.32	**0.43**
Camel	0.5	0.66	**0.78**	0.22	0.34	**0.61**
Gerrit	0.5	**0.69**	0.66	0.21	0.47	**0.59**
Ant	0.5	0.63	**0.78**	0.22	0.38	**0.55**
Jmeter	0.5	0.61	**0.75**	0.25	0.46	**0.85**
Average	*0.5*	*0.65*	*0.75*	*0.22*	*0.40*	*0.65*
Improved	*50%*	*15%*	–	*195%*	*63%*	–

The better performance values are highlighted in bold.

TABLE 16
AUC of Cross-Project Determination

	Hadoop	Log4j	Tomcat	Camel	Gerrit	Ant	Jmeter
Hadoop	–	0.77	0.76	0.80	0.77	0.82	0.68
Log4j	0.80	–	0.68	0.73	0.72	0.77	0.69
Tomcat	0.85	0.77	–	0.80	0.76	0.84	0.74
Camel	0.82	0.79	0.77	–	0.80	0.83	0.72
Gerrit	0.75	0.81	0.71	0.75	–	0.80	0.64
Ant	0.77	0.78	0.75	0.77	0.72	–	0.69
Jmeter	0.80	0.73	0.77	0.78	0.72	0.80	–
Average	*0.80*	*0.77*	*0.74*	*0.77*	*0.75*	*0.81*	*0.69*

5.4 Investigating the Effectiveness of Cross-Project Determination

In the experimental setting of RQs 1 and 2, we train our determination model by learning from the historical labeled dataset within the project. However, for new projects or projects with limited development history, there is often not enough labeled data for building a model. An alternative solution is to learn from other projects that have enough labeled data, i.e., cross-project determination. In this section, we would like to investigate how effective of our model for cross-project determination.

For each *target project*, we built the determination model by learning from other alternative projects (refer to as *source project*. In this way, there are 6 source projects for each *target project*. In the evaluation, we use the *source project* as training data, use the *target project* as testing data. In this way, there are 6 effective values for each *target project*.

Tables 16 and 17 present the results of cross-project determination. The results show that our determination model can also achieves a reasonable result in terms of AUC and cost-effectiveness. There are 6 projects which achieve greater than AUC of 0.7 considering average across

TABLE 17
Cost-Effectiveness of Cross-Project Determination

	Hadoop	Log4j	Tomcat	Camel	Gerrit	Ant	Jmeter
Hadoop	–	0.89	0.54	0.90	0.89	0.55	0.67
Log4j	0.80	–	0.60	0.90	0.90	0.55	0.83
Tomcat	0.88	0.96	–	0.97	0.97	0.56	0.87
Camel	0.85	0.96	0.66	–	0.96	0.63	0.82
Gerrit	0.75	0.85	0.49	0.74	–	0.58	0.44
Ant	0.86	0.96	0.59	0.89	0.96	–	0.67
Jmeter	0.86	0.99	0.70	0.99	0.96	0.55	–
Average	*0.83*	*0.93*	*0.60*	*0.90*	*0.94*	*0.57*	*0.72*

6 alternative source projects. In terms of cost-effective, the average performance among six source projects ranges from 0.60 to 0.94. One issue is that the performance of cross-project determination may be not stable, it depends on the source project selection. For example, for target project Jmeter, the cost-effectiveness is promising when learning on Tomcat, but it is poor when learning on Gerrit. Thus, the results indicate that our determination model can also be effective for cross-project determination, but should be careful about the source project selection.

5.5 Other General Discussion

Benefit of Change-Level Approach for Identifying TD. Compared with file-level approach, one benefit of change-level approach is that it can help to understand the context of TD related to multiple files. Understanding the TD context can help to address the TD. For example, Fig. 2 presents a TD-introducing change and the corresponding TD-removing change in Hadoop project. Fig. 2a is the TD-introducing change, its ID is *1e346aa829519f8a2aa830e76d9856f914861805*. We present two code fragments that are introduced by this TD-introducing change. This change added the function *verifyAndSetNamespaceInfo()* in *BPOfferService.java* and added the function *sendHeartBeat()* in *BPServiceActor.java*. For simplification, we use "DoSomething;" to represent the remaining of the function. We note that the function *sendHeartBeat ()* contains a SATD comment, namely *"TODO: saw an NPE here - maybe if the two BPOS register at the same time, this one won't block on the other one?"*. Fig. 2b is the corresponding TD-removing change, its ID is *b3f28dbb3d1ab6b2-f686efdd7bdb064426177f21*. We also present two code fragments that were changed by this TD-removing change. We notice that this change removed the SATD comment and changed the function *verifyAndSetNamespaceInfo()* to *"synchronized"* to address this TD. From this example, if we identified TD at the file-level in *BPServiceActor.java*, it might take more effort to understand why this TD is introduced and which file should be modified to remove this TD.

```
BPOfferService.java
+  void verifyAndSetNamespaceInfo(NamespaceInfo nsInfo) throws IOException {
+     DoSomething;
+  }

BPServiceActor.java
+ DatanodeCommand [] sendHeartBeat() throws IOException {
+  LOG.info("heartbeat: " + this);
+  // TODO: saw an NPE here - maybe if the two BPOS register at
+  // same time, this one won't block on the other one?
+  DoSomething;
+ }
```

(a) TD-introducing change

```
BPOfferService.java
- void verifyAndSetNamespaceInfo(NamespaceInfo nsInfo) throws IOException {
+ synchronized void verifyAndSetNamespaceInfo(NamespaceInfo nsInfo) throws IOException {
   DoSomething;
}

BPServiceActor.java
  DatanodeCommand [] sendHeartBeat() throws IOException {
    LOG.info("heartbeat: " + this);
- // TODO: saw an NPE here - maybe if the two BPOS register at
- // same time, this one won't block on the other one?
  DoSomething;
}
```

(b) TD-removing change

Fig. 2. TD-introducing and TD-removing change example.

However, if we identify this TD at the change-level, we can observe that this TD is introduced by change *1e346aa829519f8a2aa830e76d9856f914861805*. Additionally, we can observe the objective of this change is *"Send block report from datanode to both active and standby namenodes"* by retrieving the change log, and we can find the files that are modified or added in this change, such as *BPOfferService.java* and *BPServiceActor.java*. As shown in Fig. 2b, the TD is addressed by modifying *BPServiceActor.java*, therefore we can find that the context can help to understand and address TD.

Change-Level versus File-Level Approach for Identifying TD. Although we state that our proposed change-level approach can yield many benefits, we do not aim to use the change-level approach to supersede file-level approach. Actually, there are two main difference of these two kinds of approach. First, the development phase when they are employed is different. The change-level approach is conducted when the change is submitted. It aims to be a continuous activity for identifying TD. The file-level approach is usually conducted at a particular timing, such as before a product release. It is impractical to frequently use the file-level approach. Second, the change-level approach is a more fine-grained approach, it aims to identify changes that introduce TD. Moreover, it characterizes TD-introducing changes and helps to understand the context and address the TD. In terms of the file-level approach, the main objective is to identify which files contain TD by detecting all files. These two kinds of approach can complement each other to improve the software quality.

Combining Code Metrics/Smells and Code Comments Analysis for Identifying TD. From the above-mentioned research questions, we conclude that our proposed change-level TD determination model is effective and it can help to identify TD on change-level. In particular, our model is built by learning from changes labeled as TD-introducing. When identifying TD-introducing changes, we analyze source code comments to identify SATD first, and then label the change which introduced SATD comments as a SATD-introducing change. Compared with traditional TD identifying approaches using code metrics or code smells, our approach is more reliable, since SATDs has been admitted by developers using comments. However, not all TDs are self-admitted using comments. Thus, one potential weakness of our approach is that we may not cover all TD-introducing changes in our model. Based on this, our approach can be served as a complementary approach to existing source code analysis based approaches for identifying TD. We encourage the investigation on hybrid approach for identifying TD-introducing changes. A hybrid approach refers to that an approach combines identifying TD through code metrics/smells and comments (i.e., SATD). In this way, it can build a more comprehensive and accurate determination model.

6 Threats to Validity

Threats to internal validity relate to potential errors in our implementation. First, one potential threat to validity is the potential errors in change features extraction. To mitigate the threat, we compute the same features by following the method in previous studies [23], and adopt the third-party library, such as ChangeDisttler that has been used in past studies for change feature extraction [36], [43]. Second, one potential threat is that many comments in source code might be stale. This may impact the accuracy of our labelling step. To mitigate this issue, we performed some preprocessing steps (e.g., removing automatically generated comments and commented source code) and introduced manual analysis in the labelling step. We also conducted a study in the ICSME 2014 (in discussion section) to show that most comments are changed with code [12]. This shows that staleness is not a major issue. Third, in RQ2, we conclude the diffusion dimension is the most important dimension. One potential threat to its validity is the impact of the number of features. In diffusion dimension, we have 16 features which have more features than other dimensions. To reduce this threat, extracting more features in other dimensions and creating a more balanced division of features are needed in the future to evaluate the importance of dimensions.

Threats to external validity relate to generalizability of our results. We have analyzed 100,011 software changes from 7 different open-source Java software projects. When applying our approach to projects written by other programming languages, some features (e.g., extracted by ChangeDisttler) and the code comments preprocessing steps should be carefully adapted. In the future, further investigation by analyzing even more projects including commercial projects and projects written by other programming languages is needed to mitigate this threat.

Threats to construct validity relate to the suitability of our evaluation. One potential threat is that we use AUC and cost-effectiveness as the performance measure, and use Wilcoxon signed-rank test to investigate whether the improvement of our proposed model over baseline is significant. One or all of them have been used in past studies [23], [25], [31], [49], [50]. Thus, we believe we have little threats to construct validity.

7 Related Work

This paper aims to propose a change-level self-admitted technical debt determination model. Therefore, we divide our related work into three parts: technical debt, self-admitted technical debt and change-level determination.

7.1 Technical Debt

Due to the importance of technical debt, a number of studies have proposed different techniques for technical debt determination and management.

Zazworka et al. [8] use code smells for determining technical debt. In particular, they focus on how design debt (i.e., god classes) impacts the product quality. They found that the technical debt has a negative impact on software quality (i.e., maintainability and correctness). Thus, they suggests that technical debt should be identified and managed early. Fontana et al. [10] also use code smells for detecting design debt (i.e., god classes, data class and duplicate code). They perform an empirical study to investigate which design debt should be paid first. As a result, they found that duplicate code debt is more critical respect to others.

In their following work, Zazworka et al. [11] use code smells and issues raised by Automatic Static Analysis (ASA) tools to detect technical debt. In detail, they select CodeVizard [71] and FindBugs [72]. They compare the technical debts found by tools and human to investigate the effectiveness of automatic technical debt determination. As a result, they found that there is a small overlap between technical debts reported by tools and human. In addition, they found that automatic tools are more efficient for detecting defect debt but cannot help detecting other types of debt. Thus, it is better to combine the tools with other detection techniques.

Guo et al. [73] perform a case study to explore the effect of technical debt by tracking a single delayed maintenance task in a real software project throughout its lifecycle. They simulate how explicit technical debt management might have changed project outcomes. Ernst et al. [74] perform a survey among 1,831 participants to investigate the definition and tools on technical debt in practice. As a result, they found that software practitioners agree on the usefulness of the technical debt metaphor. In terms of the source of technical debt, they found that architectural choices are the most important source of technical debt. Additionally, they found that developer desire standard practices and tools to manage technical debt. More attention should be paid on moving technical debt from metaphor to practice.

Li et al. [5] perform a systematic mapping study on technical debt and its management. They found that some types of technical debts can be detected by using automatically code analysis tools. However, some other types of technical debt cannot be automatically detected. They suggest that additional detection techniques should be developed for their documentation and further management.

Our work is motivated by these prior works. The difference is that our work focuses on determining technical debt at the change-level.

7.2 Self-Admitted Technical Debt

Recently, Potdar and Shihab [12] first defined the concept of self-admitted technical debt. It refers to the technical debt that is introduced by a developer intentionally and is documented by developer using source code comments. They developed 62 patterns that can indicate SATD by manual inspecting 100k comments. Their 62 patterns are commonly used in the following studies.

Maldonado and Shihab [13] manually examine 33K code comments to determine different types of technical debt patterns, including design debt, defect debt, document debt, requirement debt and test debt. By performing an empirical study, they found that the most common type of SATD is design debt.

Bavota and Russo [15] perform a large-scale empirical study by presenting a differentiated replication of the work of Potdar and Shihab [12]. In detail, they investigate the diffusion and evolution of SATD and its relationship with software quality across 159 projects. As a result, they found that SATD is diffused and can survive long time. In addition, the number of SATD increases over time due to the introduction of new instances that are not fixed by developers.

Wehaibi et al. [6] examine the relationship between SATD and software quality by conducting an empirical study. They found that the impact of SATD is not related to defects, rather making the system more difficult to change in the future.

Kamei et al. [14] propose to measure the “interest” of SATD and the how much of the technical debt incurs positive interest. In detail, they use LOC and Fan-In to measure interest. As a result, they found that approximately 42-44 percent of the SATD incurs positive interest.

Farias et al. [75], [76] propose a contextualized vocabulary model (namely CVM-TD) to identify SATD from source code comments. This model focuses on using word classes and code tags to provide a vocabulary, aiming to support the detection of different types of debt through code comment analysis.

Maldonado et al. [17] perform an empirical study on the removal of SATD. They found that the majority of SATD is removed and the majority of SATD is removed by the same developer who introduces it. Fixing bugs or adding new features are the most frequently activities for removing SATD.

Most recently, different from the previous manual detection of SATD, Maldonado et al. [7] and Huang et al. [16] propose to automatically determining SATD by using natural language processing and text mining techniques. In Maldonado et al.’s work, they classify each comment into design SATD, requirement SATD and non-SATD by using natural language processing technique. The results show that it outperforms the detection method based on fixed keywords and phrases [12]. In Huang et al.’s work, they propose a more general SATD determination method, which consider all kinds of SATD. In detail, they classify each comment into SATD or non-SATD by using text mining approach. The results show that it outperforms the natural language processing method in Maldonado et al.’s work.

Our work is inspired by these prior work that also used SATD by analyzing source code comments. However, our work differs from above-mentioned works in that our work performs the determination on change-level.

7.3 Change-Level Determination

Change-level determination refers to determining if a particular characteristic of a software change, such as defective change determination and build co-change determination. Defective change determination aims to determine whether or not a change is a defect inducing change [23], [31], [37], [38]. Build co-change determination aims to determine whether or not a change requires build co-change [41], [42], [77].

For example, in terms of defective change determination, Mockus and Weiss [37] assess the risk of software changes (i.e., the probability that changes are defect inducing) in 5ESS network switch project. Kim et al. [38] classify each software change as buggy or clean by using the identifiers in added and deleted source code and textual features in change logs. Kamei et al. [23] a large-scale empirical study of change-level quality assurance on a variety of open source and commercial projects from multiple domains. They first apply effort-aware evaluation (i.e., considering review effort for inspecting defective changes) on defective change determination. Following on their work, Yang et al. [31] propose to use simple unsupervised models for defective change prediction. They found that simple unsupervised models can perform better than supervised models on defective change prediction.

In terms of build co-change determination model, McIntosh et al. [41] build a classifier that determine whether or not a software change will be build co-changing. By the following, Xia et al. [42] propose cross-project build co-change determination model to improve the performance of build co-change determination in projects in the initial development phases. Subsequently, Macho et al.[77] improve the existing model performance by taking into account detailed information on source code changes and commit categories.

The similarity between our work and these aforementioned work is the used change metrics. Many of change metrics used in our work are inspired by them, such as diffusion metrics [23], [41], [42], message metrics [38], and history metrics [23], [31]. The difference between our work and these aforementioned work is that we aim to determine whether or not a change introduces TD. To the best of our knowledge, this is the first work for determining TD at change-level.

8 Conclusion and Future Work

In this paper, we propose a change-level self-admitted technical debt determination model by extracting 25 change features that divided into three dimensions, namely diffusion, history and message. The model can determine whether or not a software change introduces TD when it is submitted. To the best of our knowledge, this paper is the first work to perform change-level TD determination. To evaluate the effectiveness of our determination model, we perform an empirical study on 7 open source projects with totally 100,011 software changes.

In summary, the experimental results show that: (1) Our model achieves a promising and better performance than the baselines in terms of AUC and cost-effectiveness. On average across the 7 experimental projects, our model achieves AUC of 0.82, cost-effectiveness of 0.80, which significantly improves the baselines in a substantial margin. (2) "Diffusion" is the most discriminative dimension among the three dimensions of features for determining TD-introducing changes. Our model (using all features) achieves the best performance compared with the three models considering the average across the 7 projects. (3) The features "Entropy", "msg_length", "LA", "LD", "FA", "ND", and "MCC" are the important features for determining TD-introducing change.

In the future, we plan to evaluate our determination model with more software projects, including both open source and commercial projects. And we also plan to study more change features that can impact TD introduction, and design a better model to improve the performance further.

Acknowledgments

The authors would like to thank the editor and the anonymous reviewers for their constructive comments and recommendations to improve this paper. This research was supported by NSFC Program (No. 61602403 and 61572426) and China Postdoctoral Science Foundation (No. 2017M621931).

References

[1] E. Lim, N. Taksande, and C. Seaman, "A balancing act: What software practitioners have to say about technical debt," *IEEE Softw.*, vol. 29, no. 6, pp. 22–27, Nov./Dec. 2012.
[2] W. Cunningham, "The WyCash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993.
[3] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing technical debt in software engineering (Dagstuhl seminar 16162)," *Dagstuhl Rep.*, vol. 6, no. 4, pp. 1–29, 2016.
[4] C. Seaman and Y. Guo, "Measuring and monitoring technical debt," *Advances Comput.*, vol. 82, no. 25–46, 2011, Art. no. 44.
[5] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *J. Syst. Softw.*, vol. 101, pp. 193–220, 2015.
[6] S. Wehaibi, E. Shihab, and L. Guerrouj, "Examining the impact of self-admitted technical debt on software quality," in *Proc. IEEE 23rd Int. Conf. Softw. Anal. Evol. Reeng.*, 2016, pp. 179–188.
[7] E. Maldonado, E. Shihab, and N. Tsantalis, "Using natural language processing to automatically detect self-admitted technical debt," *IEEE Trans. Softw. Eng.*, vol. 43, no. 11, pp. 1044–1062, Nov. 2017.
[8] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality," in *Proc. 2nd Workshop Manag. Tech. Debt*, 2011, pp. 17–23.
[9] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, "In search of a metric for managing architectural technical debt," in *Proc. Joint Work. IEEE/IFIP Conf. Softw. Archit. Eur. Conf. Softw. Archit.*, 2012, pp. 91–100.
[10] F. A. Fontana, V. Ferme, and S. Spinelli, "Investigating the impact of code smells debt on quality code evaluation," in *Proc. 3rd Int. Workshop Manag. Tech. Debt*, 2012, pp. 15–22.
[11] N. Zazworka, R. O. Spínola, A. Vetro, F. Shull, and C. Seaman, "A case study on effectively identifying technical debt," in *Proc. 17th Int. Conf. Eval. Assessment Softw. Eng.*, 2013, pp. 42–47.
[12] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2014, pp. 91–100.
[13] E. D. S. Maldonado and E. Shihab, "Detecting and quantifying different types of self-admitted technical debt," in *Proc. IEEE 7th Int. Workshop Manag. Tech. Debt*, 2015, pp. 9–15.
[14] Y. Kamei, E. D. S. Maldonado, E. Shihab, and N. Ubayashi, "Using analytics to quantify interest of self-admitted technical debt," in *Proc. Joint Int. Workshop Quantitative Approaches Softw. Quality Int. Workshop Tech. Debt Anal.*, 2016, pp. 68–71.
[15] G. Bavota and B. Russo, "A large-scale empirical study on self-admitted technical debt," in *Proc. 13th Int. Conf. Mining Softw. Repositories*, 2016, pp. 315–326.
[16] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li, "Identifying self-admitted technical debt in open source projects using text mining," *Empirical Softw. Eng.*, vol. 23, pp. 418–451, 2018.
[17] E. D. S. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik, "An empirical study on the removal of self-admitted technical debt," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2017, pp. 238–248.
[18] F. A. Fontana, J. Dietrich, B. Walter, A. Yamashita, and M. Zanoni, "Antipattern and code smell false positives: Preliminary conceptualization and classification," in *Proc. IEEE 23rd Int. Conf. Softw. Anal. Evol. Reeng.*, 2016, pp. 609–613.
[19] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," *J. Syst. Softw.*, vol. 84, no. 10, pp. 1757–1782, 2011.
[20] N. Tsantalis, D. Mazinanian, and G. P. Krishnan, "Assessing the refactorability of software clones," *IEEE Trans. Softw. Eng.*, vol. 41, no. 11, pp. 1055–1090, Nov. 2015.
[21] J. Graf, "Speeding up context-, object-and field-sensitive SDG generation," in *Proc. 10th IEEE Work. Conf. Source Code Anal. Manipulation*, 2010, pp. 105–114.
[22] K. Ali and O. Lhoták, "Application-only call graph construction," in *Proc. Eur. Conf. Object-Oriented Program.*, 2012, pp. 688–712.
[23] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 757–773, Jun. 2013.
[24] J. Huang and C. X. Ling, "Using AUC and accuracy in evaluating learning algorithms," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 3, pp. 299–310, Mar. 2005.
[25] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang, "HYDRA: Massively compositional model for cross-project defect prediction," *IEEE Trans. Softw. Eng.*, vol. 42, no. 10, pp. 977–998, Oct. 2016.
[26] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2013, pp. 279–289.

[27] X. Xia, D. Lo, X. Wang, and X. Yang, "Collective personalized change classification with multiobjective search," *IEEE Trans. Rel.*, vol. 65, no. 4, pp. 1810–1829, Dec. 2016.

[28] X. Yang, D. Lo, X. Xia, and J. Sun, "TLEL: A two-layer ensemble learning approach for just-in-time defect prediction," *Inf. Softw. Technol.*, vol. 87, pp. 206–220, 2017.

[29] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep learning for just-in-time defect prediction," in *Proc. IEEE Int. Conf. Softw. Quality Rel. Secur.*, 2015, pp. 17–26.

[30] Q. Huang, X. Xia, and D. Lo, "Supervised versus unsupervised models: A holistic look at effort-aware just-in-time defect prediction," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2017, pp. 159–170.

[31] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, "Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 157–168.

[32] M. L. Collard, M. J. Decker, and J. I. Maletic, "srcML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration," in *Proc. 29th IEEE Int. Conf. Softw. Maintenance)*, 2013, pp. 516–519.

[33] M.-A. Storey, J. Ryall, R. I. Bull, D. Myers, and J. Singer, "Todo or to bug," in *Proc. ACM/IEEE 30th Int. Conf. Softw. Eng.*, 2008, pp. 251–260.

[34] J. Cohen, "Weighted kappa: Nominal scale agreement provision for scaled disagreement or partial credit," *Psychological Bulletin*, vol. 70, no. 4, 1968, Art. no. 213.

[35] N. Japkowicz and S. Stephen, "The class imbalance problem: A systematic study," *Intell. Data Anal.*, vol. 6, no. 5, pp. 429–449, 2002.

[36] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Softw. Eng.*, vol. 33, no. 11, pp. 725–743, Nov. 2007.

[37] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Tech. J.*, vol. 5, no. 2, pp. 169–180, 2000.

[38] S. Kim, E. J. Whitehead Jr , and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 181–196, Mar./Apr. 2008.

[39] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: Analytics and risk prediction of software commits," in *Proc. 10th Joint Meet. Found. Softw. Eng.*, 2015, pp. 966–969. [Online]. Available: http://doi.acm.org/10.1145/2786805.2803183

[40] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 78–88.

[41] S. McIntosh, B. Adams, M. Nagappan, and A. E. Hassan, "Mining co-change information to understand when build changes are necessary," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2014, pp. 241–250.

[42] X. Xia, D. Lo, S. McIntosh, E. Shihab, and A. E. Hassan, "Cross-project build co-change prediction," in *Proc. IEEE 22nd Int. Conf. Softw. Anal. Evol. Reeng.*, 2015, pp. 311–320.

[43] B. Fluri and H. C. Gall, "Classifying change types for qualifying change couplings," in *Proc. 14th IEEE Int. Conf. Program Comprehension*, 2006, pp. 35–45.

[44] Y. Fu, M. Yan, X. Zhang, L. Xu, D. Yang, and J. D. Kymer, "Automated classification of software change messages by semi-supervised latent dirichlet allocation," *Inf. Softw. Technol.*, vol. 57, pp. 369–377, 2015.

[45] M. Yan, Y. Fu, X. Zhang, D. Yang, L. Xu, and J. D. Kymer, "Automatically classifying software changes via discriminative topic model: Supporting multi-category and cross-project," *J. Syst. Softw.*, vol. 113, pp. 296–308, 2016.

[46] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: How misclassification impacts bug prediction," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 392–401.

[47] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: An update," *ACM SIGKDD Explorations Newslett.*, vol. 11, no. 1, pp. 10–18, 2009.

[48] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[49] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, Jul./Aug. 2008.

[50] J. Nam and S. Kim, "CLAMI: Defect prediction on unlabeled datasets (T)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2015, pp. 452–463.

[51] A. P. Bradley, "The use of the area under the ROC curve in the evaluation of machine learning algorithms," *Pattern Recognit.*, vol. 30, no. 7, pp. 1145–1159, 1997.

[52] F. Rahman, D. Posnett, and P. Devanbu, "Recalling the imprecision of cross-project defect prediction," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, Art. no. 61.

[53] M. Yan, Y. Fang, D. Lo, X. Xia, and X. Zhang, "File-level defect prediction: Unsupervised versus supervised models," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.*, 2017, pp. 344–353.

[54] X. Xia, E. Shihab, Y. Kamei, D. Lo, and X. Wang, "Predicting crashing releases of mobile applications," in *Proc. 10th ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.*, 2016, Art. no. 29.

[55] J. Han, J. Pei, and M. Kamber, *Data Mining: Concepts and Techniques*. Amsterdam, The Netherlands: Elsevier, 2011.

[56] A. McCallum, K. Nigam, et al., "A comparison of event models for naive bayes text classification," in *Proc. AAAI Workshop Learn. Text Categorization*, 1998, pp. 41–48.

[57] X. Xia, D. Lo, W. Qiu, X. Wang, and B. Zhou, "Automated configuration bug report prediction using text mining," in *Proc. IEEE 38th Annu. Comput. Softw. Appl. Conf.*, 2014, pp. 107–116.

[58] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in Statistics*. Berlin, Germany: Springer, 1992, pp. 196–202.

[59] H. Abdi, "Bonferroni and Šidák corrections for multiple comparisons," *Encyclopedia Meas. Statist.*, vol. 3, pp. 103–107, 2007.

[60] N. Cliff, *Ordinal Methods for Behavioral Data Analysis*. London, U.K.: Psychology Press, 2014.

[61] Y. Tian, M. Nagappan, D. Lo, and A. E. Hassan, "What are the characteristics of high-rated apps? A case study on free android applications," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2015, pp. 301–310.

[62] Y. Fan, X. Xia, D. Lo, and S. Li, "Early prediction of merged code changes to prioritize reviewing tasks," *Empirical Softw. Eng.*, pp. 1–48, 2018, https://doi.org/10.1007/s10664-018-9602-0

[63] L. Bao, Z. Xing, X. Xia, D. Lo, and S. Li, "Who will leave the company?: A large-scale industry study of developer turnover by mining monthly work report," in *Proc. IEEE/ACM 14th Int. Conf. Mining Softw. Repositories*, 2017, pp. 170–181.

[64] H. Li, W. Shang, Y. Zou, and A. E. Hassan, "Towards just-in-time suggestions for log changes," *Empirical Softw. Eng.*, vol. 22, no. 4, pp. 1831–1865, 2017.

[65] D. H. Wolpert and W. G. Macready, "An efficient method to estimate bagging's generalization error," *Mach. Learn.*, vol. 35, no. 1, pp. 41–55, 1999.

[66] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 1–18, Jan. 2017.

[67] X. Xia, L. Bao, D. Lo, P. S. Kochhar, A. E. Hassan, and Z. Xing, "What do developers search for on the web?" *Empirical Softw. Eng.*, vol. 22, pp. 3149–3185, 2017.

[68] A. J. Scott and M. Knott, "A cluster analysis method for grouping means in the analysis of variance," *Biometrics*, vol. 30, pp. 507–512, 1974.

[69] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *Ann. Math. Statist.*, vol. 18, pp. 50–60, 1947.

[70] H. C. Gall, B. Fluri, and M. Pinzger, "Change analysis with evolizer and changedistiller," *IEEE Softw.*, vol. 26, no. 1, pp. 26–33, Jan./Feb. 2009.

[71] N. Zazworka and C. Ackermann, "CodeVizard: A tool to aid the analysis of software evolution," in *Proc. ACM-IEEE Int. Symp. Empirical Softw. Eng. Meas.*, 2010, Art. no. 63.

[72] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, 2004.

[73] Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. Q. Da Silva, A. L. Santos, and C. Siebra, "Tracking technical debt—An exploratory case study," in *Proc. 27th IEEE Int. Conf. Softw. Maintenance*, 2011, pp. 528–531.

[74] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure it? manage it? ignore it? software practitioners and technical debt," in *Proc. 10th Joint Meet. Found. Softw. Eng.*, 2015, pp. 50–60.

[75] M. A. de Freitas Farias, M. G. de Mendonça Neto, A. B. da Silva, and R. O. Spínola, "A contextualized vocabulary model for identifying technical debt on code comments," in *Proc. IEEE 7th Int. Workshop Manag. Tech. Debt*, 2015, pp. 25–32.

[76] M. A. de Freitas Farias, J. A. M. Santos, A. B. da Silva, M. Kalinowski, M. G. Mendonça, and R. O. Spínola, "Investigating the use of a contextualized vocabulary in the identification of technical debt: A controlled experiment," in *Proc. 18th Int. Conf. Enterprise Inf. Syst.*, 2016, pp. 369–378.
[77] C. Macho, S. McIntosh, and M. Pinzger, "Predicting build co-changes with source code change and commit categories," in *Proc. IEEE 23rd Int. Conf. Softw. Anal. Evol. Reeng.*, 2016, pp. 541–551.

Meng Yan received the PhD degree from the School of Software Engineering, Chongqing University, in June 2017. He is a post-doctoral research fellow with the College of Computer Science and Technology, Zhejiang University. His currently research focuses on how to improve developers productivity, how to improve software quality and how to reduce the effort during software development by analyzing rich software repository data.

Xin Xia received the bachelor's and PhD degrees in computer science and software engineering from Zhejiang University, in 2009 and 2014, respectively. He is a lecturer with the Faculty of Information Technology, Monash University, Australia. Prior to joining Monash University, he was a post-doctoral research fellow in the Software Practices Lab, University of British Columbia in Canada, and a research assistant professor with Zhejiang University in China. To help developers and testers improve their productivity, his current research focuses on mining and analyzing rich data in software repositories to uncover interesting and actionable information.

Emad Shihab received the PhD degree from Queens University. He is an associate professor with the Department of Computer Science and Software Engineering, Concordia University. His research interests include software quality assurance, mining software repositories, technical debt, mobile applications, and software architecture. He worked as a software research intern at Research in Motion in Waterloo, Ontario and Microsoft Research in Redmond, Washington. He is a member of ACM, and a senior member of IEEE. More information can be found at http://das.encs.concordia.ca.

David Lo received the PhD degree from the School of Computing, National University of Singapore, in 2008. He is currently an associate professor with the School of Information Systems, Singapore Management University. He has close to 10 years of experience in software engineering and data mining research and has more than 200 publications in these areas. He received the Lee Foundation Fellow for Research Excellence from the Singapore Management University in 2009, and a number of international research awards including several ACM distinguished paper awards for his work on software analytics. He has served as general and program co-chair of several prestigious international conferences (e.g., IEEE/ACM International Conference on Automated Software Engineering), and editorial board member of a number of high-quality journals (e.g., the *Empirical Software Engineering*).

Jianwei Yin received the PhD degree in computer science from Zhejiang University, in 2001. He is currently a professor with the College of Computer Science, Zhejiang University. He is the visiting scholar of Georgia Institute of Technology, in 2008. His research interests include service computing, software engineering, and distributed computing.

Xiaohu Yang received the PhD degree from the College of Computer Science and Technology, Zhejiang University, in 1993. He is currently a professor with the College of Computer Science and Technology, Zhejiang University. His research interests include software engineering, cloud computing, and FinTech.