Where Should We Fix This Bug? A Two-Phase Recommendation Model

Dongsun Kim, *Member*, *IEEE*, Yida Tao, *Student Member*, *IEEE*, Sunghun Kim, *Member*, *IEEE*, and Andreas Zeller, *Member*, *IEEE*

Abstract—To support developers in debugging and locating bugs, we propose a two-phase prediction model that uses bug reports' contents to *suggest the files likely to be fixed*. In the first phase, our model checks whether the given bug report contains sufficient information for prediction. If so, the model proceeds to predict files to be fixed, based on the content of the bug report. In other words, our two-phase model "speaks up" only if it is confident of making a suggestion for the given bug report; otherwise, it remains silent. In the evaluation on the Mozilla "Firefox" and "Core" packages, the two-phase model was able to make predictions for almost half of all bug reports; on average, 70 percent of these predictions pointed to the correct files. In addition, we compared the two-phase model with three other prediction models: the Usual Suspects, the one-phase model, and BugScout. The two-phase model manifests the best prediction performance.

Index Terms—Bug reports, machine learning, patch file prediction

1 Introduction

In modern software development, bugs are inevitable. Maintainers of software systems thus find themselves faced with a stream of *bug reports*, failure reports stored and managed through *issue tracking systems*. A bug report holds information about a specific software failure, including the failure symptoms, the affected platforms, and a scenario on how to reproduce the failure.

For a developer, all this information is helpful to find and fix the bug—but rarely sufficient: Already the first decision, namely deciding on where to start the investigation, requires expert knowledge about where features are located in the system, and where similar bugs have been fixed in the past. Indeed, developers might even get stuck in this first step, struggling in finding the right location to start debugging [1], [2].

In this paper, we present an approach that automatically *suggests the files where a bug will most likely be fixed* based on its bug report's content. Specifically, our approach extracts features such as the summary, the initial description, product version, and platform descriptors from the given bug report. For a small number of known bugs, we associate these features with actual fix locations and train a *prediction model* that later predicts the fix location for a new bug report.

However, this basic idea, which we refer to as the *one-phase* prediction, does not work well. Many bug reports do not contain enough of the information required for a good prediction. This observation motivates us to propose a *two-phase* model where we first check whether a bug report is "predictable" and only if it is "predictable" do we proceed to predict a fix location; otherwise, we leave everything as it is. This way, if we make a prediction, it can be very precise.

In evaluation on Mozilla "Firefox" and "Core" packages, the two-phase model was able to make predictions for almost half of all bug reports; on average, 70 percent of the predictions pointed to the correct files. The *two-phase* model was further compared with the *Usual Suspects* baseline, the one-phase model, and BugScout [1], which is a state-of-the-art model that locates buggy source files from bug reports. Among these four prediction models, our two-phase model shows the best performance in terms of prediction likelihood, precision, and recall. In addition, the two-phase model ranks correctly predicted files at higher positions compared to the three other models.

In general, this paper makes the following contributions:

1. *A two-phase model for recommendations*. We weed out "unpredictable" inputs to promote precision and avoid possibly misleading recommendations. To the best of our knowledge, this is the first time a prediction model handles the data quality issue during prediction.
2. *A strong baseline*. We introduce the baseline of the "Usual Suspects"—the files most frequently fixed, which would be natural candidates for future fixes.
3. *A comprehensive evaluation*. We evaluate the prediction performance (i.e., likelihood, precision, and recall) of the two-phase model and compare it with the Usual Suspects, the one-phase model, and BugScout. Our evaluation shows the effectiveness of the two-phase predictor over the other models and suggests tangible benefits when deployed in practice.

- *D. Kim, Y. Tao, and S. Kim are with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clear Water Bay Road, Kowloon, Hong Kong. E-mail: {darkrsw, idagoo, humkim}@cse.ust.hk.*
- *A. Zeller is with the Universität des Saarlandes-Informatik, Campus E1 1, Saarbrücken 66123, Germany. E-mail: zeller@acm.org.*

Manuscript received 9 Feb. 2012; revised 11 Nov. 2012; accepted 4 May 2013; published online 20 May 2013.
Recommended for acceptance by T. Menzies.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2012-02-0027.
Digital Object Identifier no. 10.1109/TSE.2013.24.

The remainder of this paper is organized as follows: Section 2 discusses related work. Section 3 describes our approach, detailing the one-phase and two-phase setups. Section 4 introduces our evaluation setup. Section 5 presents the results, which are then discussed in Section 6. After discussing threats to validity in Section 7, Section 8 closes with the conclusion and future work.

2 Related Work

2.1 Bug Localization

Bug localization has been widely studied in the past decades. Various techniques have been proposed which mainly leverage dynamic and static analyses. Recent work has also applied repository-mining techniques to reveal and locate software bugs.

Dynamic analysis. Several studies have analyzed a program's runtime behavior to discover bugs. One line of research is statistical debugging [3], [4], [5]. The basic concept is to collect statistics characterizing a program's runtime behavior over multiple executions. The collected statistics, such as program invariants or predicate evaluations, are analyzed afterward to compute the probability of them being faulty [3].

Some fault localization approaches use test suite execution to locate bugs. For example, Jones et al. [6] visualize each statement's participation in the execution of passing and failing test cases. Such visualization helps developers inspect a program and identify suspicious statements. Cleve and Zeller [7] identified causes of failures by comparing program states of failing and passing runs. However, these approaches heavily rely on the availability and quality of test suites. In addition, together with the above-mentioned statistical debugging, all these approaches require successful execution of the program. In particular, to isolate a failure cause more effectively from passing and failing runs, a larger number of executions and higher execution similarities are required [8].

A program's dynamic properties are also exploited to locate bugs. In the work of Brun and Ernst [9], program properties generated by dynamic program analysis are first marked as fault-revealing or otherwise. These properties are then fed into a machine learner that identifies program properties with latent errors. The Chianti tool proposed by Ren et al. [10] constructs dynamic call-graphs via program test runs. The tool then characterizes a program change as a set of atomic changes and associates each of them with its corresponding call-graph portion. When a test fails after a program change, Chianti can determine the particular part of the change responsible for the test failure. Chianti is later extended to various applications such as Crisp [11] and JUnit/CIA tool [12], which similarly detect the failure-inducing program changes.

Static analysis. While dynamic analysis is in general expensive to apply, static analysis is capable of detecting bugs by only examining program model or source code directly, without any actual run of the program. Techniques such as program slicing have been proposed to facilitate debugging activities by isolating the program location that is likely buggy [13], [14]. However, program slicing is known for its high computation cost and low accuracy [15], [16]. Recent studies have used symbolic execution to explain failures and locate bugs. One of the well-known examples is PSE, a postmortem symbolic evaluation algorithm that helps developers diagnose software failures. PSE requires minimal information about a failure, namely, its type and location in the source code as inputs, and reproduces a set of execution traces that show how the program is driven to the given failure [17]. Specifically, PSE starts with the value of interest at the identified failure point and then applies a novel backward analysis to precisely explore the program's behavior. Experiment results show the scalability and precision of PSE, as well as its usefulness in diagnosing real case failures.

Mining software repositories. Several defect prediction approaches explore the rich information that resides in software repositories. These techniques typically use metrics such as code churn, past fixes, and change-proneness to predict defects [18], [19], [20], [21], [22]. Recently, novel prediction metrics such as interaction patterns of developers' behaviors have been proposed [23].

Ying et al. proposed an approach to mining software cochanges which is particularly close to our work. Their approach applies the association rule mining algorithm to find frequently cochanged files [24]. They used frequent cochange patterns extracted from version control system (VCS) and bug tracking system to predict possible should-be-changed files when given a newly changed file. Another similar study of mining cochanges is introduced by Zimmermann et al., whose work also aimed at predicting likely software cochanges using association rule mining on VCSs [25]. In contrast to Ying's work, they used a particular association rule mining algorithm in which both support and confidence were considered, which allowed a probabilistic representation of the recommendation results. In addition, Zimmermann's tool, ROSE, not only suggested possible cochange files but could also predict other finer-grained cochange entities such as fields or functions.

Nevertheless, the cochange file recommendation techniques require at least one change file be specified at the beginning. In the context of fixing bugs, the developer should at least know one buggy file before adopting any file recommendation tools to identify other cochange files. Similarly, most of the state-of-the-art bug localization approaches described above implicitly assume that buggy source files are known in the first place. However, locating the first buggy file is not always an easy task. Our approach described in Section 3 addresses this issue directly, and therefore is complementary to these file recommendation techniques.

Some of the above-discussed techniques can locate bugs at method level or even at statement level. While our current two-phase prediction model aims at locating bugs at file level, it can be easily extended to achieve finer-grained prediction. We discuss this in Section 8.

2.2 Information Retrieval (IR) and Concept Location

A bug report typically includes information such as the bug's severity, dependencies, textual description, and discussion written in natural language, all of which together record detailed activities along a bug's life cycle.

Researchers have applied various information retrieval approaches on bug reports to identify and locate bugs. Some commonly adopted IR approaches include Latent Semantic Indexing (LSI) [26], [27], [28], Latent Dirichlet Allocation (LDA) [1], [29], Vector Space Model (VSM) [2], [30], and their variations.

Poshyvanyk et al. combined LSI and Scenario-based Probabilistic Ranking (SPR) to locate unwanted features (i.e., bugs) [28]. Their approach located eight real bugs in Eclipse and Mozilla. Nguyen et al. [1] proposed a technique that recommends candidate files to fix based on topics of bug reports and source code files extracted using LDA.

Zhou et al. [2] proposed a combined approach to locate files that need to be fixed in response to a given bug report. This approach basically leverages textual similarity between bug reports and source code. First, it computes the textual similarity of source code files with a given report and calculates the scores of the files based on the computed similarity and the file length. In addition, it also computes the similarity between previously submitted bug reports and the given report and calculates another score for the connected files of the similar reports. Then the approach combines the two scores to recommend top N files to fix for the given bug report. This approach yields promising results—60 percent accuracy on Eclipse. However, performance is comparable only to our one-phase model on Mozilla subjects (Section 5.1). This implies that our two-phase model could improve the performance of this approach.

Other IR-based approaches have also been proposed. For example, DebugAdvisor allows developers to search, using a fat query that contains structured and unstructured data describing the bug. It then recommends assignees, source files, and functions related to the queried bugs [31]. Interactive approaches leverage users' knowledge and improve the localization results according to users' feedback [30], [32]. A recent work by Rao and Kak [33] presented a comparative study of five generic IR models used in the context of bug localization.

This line of work is also closely related to concept location, which aims at associating human conceptual knowledge and their implementation counterparts of a given program [34]. Concept location is later applied to practical software development scenarios such as feature location, bug localization, and traceability recovery. For example, Gay et al. applied their concept location technique in the context of bug localization, where a bug report was used as query to locate the methods to be fixed. McMillan et al. [35] proposed a finer-grained approach to function search. This approach took functional descriptions (queries) from a user and leveraged association models that described relationships between functions to retrieve relevant functions and their usage.

Although sharing the similar goal of locating bugs, we propose a two-phase machine learning model to predict files to fix. Our model does not involve user interaction and does not require users to have sufficient knowledge of the target programs to make appropriate queries. In addition, our model has one crucial filtering step that eliminates inadequate bug reports. Note that insufficient information can result in inaccurate or even misleading recommendation, which is effectively avoided in our model.

2.3 Data Quality in SE Research

Advancements in Software Configuration Management Systems (SCMs) encourage extensive usage of development-related data in software engineering research. However, only a small fraction of such studies explicitly consider the data quality issue [36]. In early 1990s, Balzer [37] proposed a "pollution marker" to tolerate inconsistency in software systems. Basically, the pollution marker can uniquely identify the particular data that violate the consistency criteria, allowing developers to circumvent the inconsistency and continue development [38]. The concept of this work is similar to ours since our predication model also identifies deficient bug reports and filters them out before further recommendation.

Recently, many researchers have recognized the critical role of the dataset used in empirical SE studies. Specifically, the quality and appropriateness of the dataset may significantly impact an approach's effectiveness, as well as the generalizability of the research conclusion. Khoshgoftaar and Seliya [39] stressed the need for assuring quality in software measurement data through a case study on NASA data sets. They pointed out possible reasons, such as presence of noise, improper data collection, and faulty data recording, that may affect the classification performance. Aranda and Venolia [40] performed an extensive field study of bugs' life cycle in Microsoft. They found that electronic repositories often hold incomplete or inaccurate data. In addition, the dataset automatically extracted from the repositories tends to miss personal factors and social interactions during software development.

Several studies have proposed techniques to improve the quality of datasets used in SE research. Mockus [41] reviewed methods handling missing data and applied them to a practical software engineering dataset. Liebchen et al. compared and assessed three noise handling methods—filtering, robust filtering, and polishing—in empirical SE studies [42]. Kim et al. found that the defect prediction performance decreases significantly when the dataset contains 20-35 percent noise [43]. They proposed an algorithm, Closest List Noise Identification (CLNI), to detect and eliminate noise. Bird et al. [44] investigated the analysis bias arising from the missing links between changeset and bug entry. The ReLink algorithm proposed by Wu et al. [45] recovers missing links between changes and bugs. Specifically, ReLink automatically learns criteria of features from explicit links and recovers the unknown link if it satisfies the criteria.

Our work also directly addresses the data quality issue. The first phase in our two-phase prediction model effectively eliminates deficient bug reports, which might otherwise mislead the later buggy-file recommendation. Unlike the above-mentioned studies, our prediction model does not require reliable linkage between bug reports and changes. To the best of our knowledge, our two-phase model is the first of its kind to consider the quality and information sufficiency of bug reports.

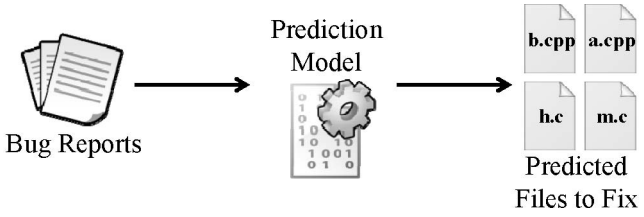


Fig. 1. One-phase prediction model.

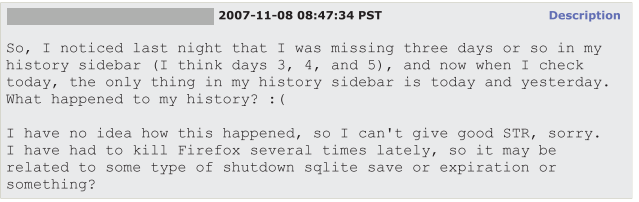


Fig. 2. An uninformative bug report. This is an excerpt from Mozilla Bug #403040, written by the bug submitter. This description is not informative and the bug reviewer indeed had to ask the submitter for further elaboration on his browser's history and bookmark settings.

3 Our Approaches

We propose machine learning-based approaches to predict potential fix locations (i.e., files to fix) for given bug reports. Section 3.1 explains the feature extraction process. Sections 3.2 and 3.3 provide detailed descriptions of the one-phase and two-phase prediction models, respectively.

3.1 Feature Extraction

A bug report is the main source of information for developers to understand a bug: The bug summary briefly describes the bug while the initial description explains it in detail, meta-data provides the bug's basic information such as version and platform, and comment threads record discussions from bug reporters and developers.

Since our approach uses machine learning classification, we transformed a bug report into a feature vector (in the machine learning sense). We extracted feature values from the summary, initial description, and metadata (version, platform, OS, priority, severity, and reporter) of a bug report. We used the bag of words approach [46] to extract word tokens from the summary and initial description since both are natural language text. Then, the word tokens are stemmed into their *ground* form. Finally, stop words such as "I," "are," "he," and "she" were removed (for this process, we used 429 stop words[1]). After these steps, the processed word tokens are used as feature values.

We considered comments added by the same reporter within 24 hours from the bug submission as part of the initial descriptions because these comments are usually a supplementary description of the bug.[2] On the other hand, we excluded information such as assignee, reviewer, and additional comments (except the kind of comments described above) in feature extraction because our goal is to predict potential files to fix *right after* the bug report is submitted.

Bug metadata is directly extracted as feature values without any processing. For example, the platform information of a bug is recorded as nominal values such as "Windows" and "Mac." This information does not need any additional textual processing.

Finally, a bug report's metadata values and word tokens are incorporated into a feature vector. For example, the feature vector for Mozilla bug report #203556 is encoded as: ("client software," "Firefox," "Bookmarks & History," "x86," ..., 4, 0, 4, 1), where each field represents (product classification, product, component, platform, ..., # of "bookmark," # of "history," # of "toolbar," # of "hover"). We used the number of occurrences of each word token (i.e., term-frequency) as features. We built a corpus using features extracted from all bug reports to train and test our prediction models.

1. Adopted from http://www.lextek.com/manuals/onix/stopwords1.html.

2. Please refer to Mozilla bug #264031 as an example. In this bug report, the reporter *pavel.penaz* described a defect briefly first and added a comment with more details just four minutes later.

3.2 One-Phase Prediction Model

To predict files to fix, we first propose the *One-phase Prediction Model* (Fig. 1). This model takes features extracted from collected bug reports as training data. When a new bug report is submitted, features are extracted from it and given to the model, which then recommends files to fix for the new bug report.

We built the prediction model using Naive Bayes [46], [47], which is a simple probabilistic classification algorithm based on Bayes' theorems. Note that our approach is independent of specific machine learning techniques. We chose Naive Bayes because it is well suited to our problem: A bug report could have multiple files to fix, which requires the prediction model to be able to handle multiclass classification problems [48]. In addition, the prediction model should accept nominal values as features.

Once trained, the one-phase model predicts a set of candidate files for a given bug report. The model further computes each file's probability of being a file to fix. Based on this probability, the top k files are recommended to developers as the prediction result.

3.3 Two-Phase Prediction Model

As Hooimeijer and Weimer [49] and Zimmermann et al. [50] noticed, the quality of bug reports can vary considerably. Some bug reports may not have enough information to predict files to fix. Our evaluation of one-phase prediction (Section 5) confirms this conjecture: Bug reports whose files are not successfully predicted usually have insufficient information (e.g., no initial description). In other words, including *uninformative* bug reports might yield poor prediction performance.

Fig. 2 shows an example of an uninformative bug report. In this report, the submitter describes a problem faced when using Firefox. However, this description is very general and contains few informative keywords that indicate the problematic modules. Therefore, it is not helpful for developers to locate the files to fix. Similarly, our one-phase prediction model does not perform well with such uninformative bug reports.

Hence, it is desirable to filter out uninformative bug reports before the actual prediction process. Based on this observation, we propose the *two-phase prediction model* that

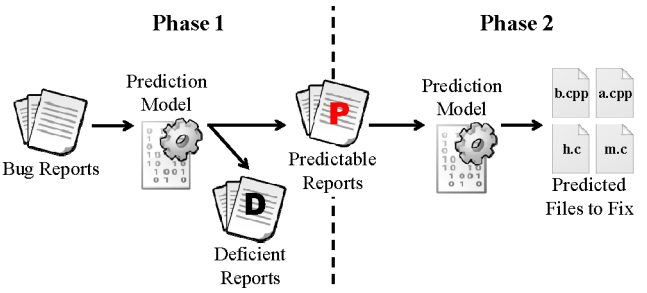


Fig. 3. The two-phase prediction model. This model recommends files to fix only when the given bug report is determined to have sufficient information.

has two classification phases: binary and multiclass classification (Fig. 3). The model first filters out uninformative reports (Section 3.3.1) and then predicts files to fix (Section 3.3.2).

3.3.1 Phase 1

Phase 1 filters out uninformative bug reports before predicting files to fix. Its prediction model classifies a given bug report as "predictable" or "deficient" (binary classification), as shown in Fig. 3. Only bug reports classified as "predictable" are taken up for the Phase 2 prediction.

The prediction model in Phase 1 leverages prediction history. The training dataset of this model uses a set of bug reports that have already been resolved. Let $B = \{b_1, b_2, \ldots, b_n\}$ be a set of n resolved bug reports chronologically sorted by their filing date. $V(b_i)$ is the ith bug report's feature vector, which is extracted as described in Section 3.1. $P(b_i)$ is the set of actual files changed to fix the bug (i.e., the files in the bug's patch), which can be obtained as well from report b_i. For each report, its label ("predictable" or "deficient") is determined by the following process: For an arbitrary report $b_j \in B$, a one-phase prediction model M_j is trained on $\{(V(b_1), P(b_1)), (V(b_2), P(b_2)) \ldots (V(b_{j-1}), P(b_{j-1}))\}$ to predict files to fix for b_j. If the prediction result hits any file in $P(b_j)$, b_j is labeled as "predictable"; otherwise, it is labeled as "deficient." Now, let $L(b)$ be the label of report b. By applying the above process to all reports in $B - \{b_1\}$, we can obtain the training dataset $\{(V(b_2), L(b_2)), (V(b_3), L(b_3)), \ldots, (V(b_n), L(b_n))\}$ for the prediction model of Phase 1. Note that no training dataset is built for b_1 because there is no bug report before b_1 to create $(V(b_1), L(b_1))$.

When a new bug report is submitted, the prediction model classifies it as either "predictable" or "deficient." If the report is classified as "predictable," it is passed on to Phase 2 prediction; otherwise, no further prediction is conducted. In the latter case, developers may ask the report submitter to give more information about the bug.

3.3.2 Phase 2

The Phase 2 model accepts "predictable" bug reports obtained from Phase 1 as the input. It extracts features from these "predictable" bug reports and is trained on $\{(V(b_1), P(b_1)), (V(b_2), P(b_2)) \ldots (V(b_m), P(b_m))\}$, where m is the number of "predictable" bug reports. The model then performs multiclass classification to recommend files to fix.

As a result, our two-phase model produces two different outcomes. For "predictable" reports, the model predicts a set of files to fix; each file is associated with a probability of being the file to fix. After sorting, the top k files are recommended to developers. For "deficient" reports, the model simply produces an empty set because no prediction is actually conducted.

4 Evaluation Setup

We experimentally evaluated our proposed approaches. Specifically, our evaluation addresses the following research questions:

- **RQ1.** What is the predictive power of the two-phase model in recommending files to fix
- **RQ2.** How many bug reports are *predictable* after Phase 1 prediction?
- **RQ3.** Which features are more indicative in fix location prediction?
- **RQ4.** Can recommended files effectively help developers?

In this section, we first present the selected subjects and evaluation setup. We then introduce two prediction models, *Usual Suspects* and *BugScout*, used for comparison, followed by our evaluation measures.

4.1 Subjects

We used the "Firefox" and "Core" projects in the Mozilla Software Foundation as the subjects of evaluation. We selected these two projects because we could reliably collect fixed files (oracle set) for the corresponding bug reports (feature set). For some issue tracking systems of other projects, we needed to use links between bug reports and source code to identify fixed files, but they often have many missing links which lead to noisy data [44], [45]. However, Mozilla developers directly post the patch files in the bug reporting system,[3] which are then reviewed by core developers. Only accepted patches are finally committed to their version control system and all activities related to decision making are recorded in bug reports. For this reason, the oracle set (bug reports and corresponding files in accepted patches) collected from Mozilla projects does not suffer from noise due to missing links [44], [45], [51].

We collected bug reports and the corresponding fix files from eight modules as shown in Table 1. Two modules from the Firefox project, *ff-bookmark* and *ff-general*, have 1,437 and 720 bug reports, respectively. Both have more than 4,600 features and approximately two patch files per report on average. Six modules from the Core project, *core-js*, *core-dom*, *core-layout*, *core-style*, *core-xpcom*, and *core-xul*, have 573-1,906 bug reports. They have approximately 5,500-13,500 features and four patch files per report on average.

The "# of features" column in Table 1 shows the total number of features extracted from each module's bug reports. As described in Section 3.1, there are two main sources of the features: a bug report's textual information (e.g., summary and initial description) and metadata. While the number of features extracted from textual information

3. https://bugzilla.mozilla.org/.

TABLE 1
Dataset Used in Our Evaluation

Module name	Period	# of total reports	# of reports with accepted patches	# of files (# of classes)	# of features	# of files / report		
						avg.	std.	max.
ff-bookmark	2001-07-21 – 2010-04-02	1927	1437	346	7585	2.21	2.54	44
ff-general	2002-06-12 – 2010-05-06	1289	720	559	4631	1.83	1.90	27
core-js	1999-09-16 – 2010-05-06	2391	1906	1114	10982	3.13	5.15	120
core-dom	2000-12-10 – 2010-05-13	1050	838	1496	8763	4.55	11.48	184
core-layout	1998-12-11 – 2010-05-13	2391	1571	1318	13497	4.32	12.51	228
core-style	1999-02-23 – 2010-04-29	1131	573	767	5453	3.71	7.85	150
core-xpcom	1999-04-22 – 2010-04-29	1059	674	2108	8588	4.76	13.81	198
core-xul	1999-06-04 – 2010-04-28	1260	676	1424	7908	3.43	9.23	149

"Period" represents the time period between submission of the first and the last bug report. The "# of total reports" column shows the total number of bug reports with patches. Among these bug reports, the "# of reports with accepted patches" column represents how many of them have accepted patches. "# of files" stands for the number of distinct files fixed during the period. "# of features" shows the total number of features extracted after preprocessing. Three columns in "# of files/report" represent the average, standard deviation, and maximum number of files in a patch per report.

varies, all modules have the same number of metadata features—the six features listed in Section 3.1. For example, *ff-bookmark* has 7,585 features in total which contain 906 features from summary, 6,673 from initial description, and six from metadata; *core-js* has 10,982 features in total with 2,056 from summary, 8,920 from initial description, and six from metadata.

4.2 Training and Test Sets

We divided bug reports into two sets: training and test sets. Bug reports for each module were chronologically sorted with respect to bug IDs. Then, the first 70 percent of bug reports were used as a training set and the remaining 30 percent as a test set. For a fair comparison, we built the prediction models of four different approaches (the one-phase model in Section 3.2, the two-phase model in Section 3.3, and the Usual Suspects and BugScout in Section 4.3) by using the training set and then evaluated them by the test set.

4.3 Models for Comparison

For a comparative study, we used two models: *Usual Suspects* and *BugScout*.

Usual suspects. Unlike the one-phase and two-phase prediction models that use nearly all information (i.e., metadata, summary, and initial description) from a given bug report, the Usual Suspects model takes previously fixed files and their occurrences as the only source of information. The intuition is that previously fixed files are likely to be fixed again soon [52]. In fact, this observation has been widely used in the defect prediction literature. For example, Khoshgoftaar et al. [53] classified a software module as fault-prone if the previous debug code churn (i.e., the number of lines of code added, changed, or deleted due to bug fix) of the module exceeded a given threshold. Hassan and Holt [54] used the number of recently modified and fixed files to predict susceptible or defect-prone subsystems.

We built the Usual Suspects model as follows: First, we use a 70-30 percent chronicle split to obtain training and test sets as described in Section 4.2. The model collects the top k most frequently fixed files from the training set. Then, the Usual Suspects model predicts the collected top-k files as fix candidates for a new bug report. In fact, this simple prediction model performs reasonably well, as shown in Section 5.

BugScout [1]. This model is a state-of-the-art technique that leverages bug-proneness and topic distribution of source code files to predict fix location. When a new bug report is submitted, BugScout first computes the cosine similarity of topic distributions between the new report and each source code file. It also computes the bug-proneness of each source code file as the number of bugs found in its history, which is identical to the Usual Suspects model. Then, the similarity value of each source code file is multiplied by its bug-proneness, which is now its defect-proneness value with respect to the bug report. BugScout then sorts the source code files based on their defect-proneness values. Finally, the top-N files are recommended for fixing the bug.

In our evaluation, the parameters of BugScout were set as suggested in [1]: hyperparameters α and β are set to 0.01. The number of topics was 300. We trained the prediction model of BugScout by using 70 percent of bug reports and tested it using the remaining 30 percent of reports. The bug reports were chronologically sorted, as described in Section 4.2.

4.4 Evaluation Measures

This section describes the performance measures we use for our evaluation:

- *Likelihood* measures the accuracy of prediction results. This is an effective measure to evaluate recommendation techniques [1], [25], [55]. We consider the prediction results to be correct if at least one of the recommended k files matches one of the actual patch files for a given bug report [1]. If none of the recommended files matches, the prediction is incorrect. We denote the number of bug

reports as N_C if the corresponding prediction is correct, N_{IC} if prediction is incorrect. The following formula computes the percentage of bug reports for which the prediction is correct:

$$\text{Likelihood} = \frac{N_C}{N_C + N_{IC}}. \quad (1)$$

- *Precision* characterizes the number of correctly predicted files over the number of files recommended by our approach. We denote the set of actual files fixed for a bug report as F_B and the set of recommended files as F_R:

$$\text{Precision} = \frac{|F_B \cap F_R|}{|F_R|}. \quad (2)$$

- *Recall* characterizes the number of correctly predicted files over the number of actual fixed files:

$$\text{Recall} = \frac{|F_B \cap F_R|}{|F_B|}. \quad (3)$$

- *Average rank* denotes the average rank of all correctly predicted files. We denote the rank of a predicted file f as R_f:

$$\text{Avg. Rank} = \frac{\sum_{f \in F_B \cap F_R} R_f}{|F_B \cap F_R|}. \quad (4)$$

- *Mann-Whitney statistical test*. In Section 4.3, we introduced two models for comparison. To check the significance of the performance differences between the two-phase model and the two models, we conducted the Mann-Whitney statistical test [56] to verify whether the performance differences are statistically significant with 95 percent confidence [57].

  We chose this nonparametric test method instead of any parametric test method such as t-test because the distribution of our evaluation results may not be normal.

In addition, we used *Feedback* [25] to compute the ratio of bug reports classified as *predictable* after Phase 1 prediction. Let N_P denote the number of *predictable* bug reports and N_D denote the number of *deficient* ones. Feedback is computed as follows:

$$\text{Feedback} = \frac{N_P}{N_P + N_D}. \quad (5)$$

5 Results

This section reports the evaluation results. Sections 5.1 and 5.2 report the prediction performance and compare the results of four different models with their statistical significance (RQ1). We discuss the feedback (RQ2) in Section 5.3, and present the sensitivity analysis in Section 5.4 to compare the prediction power of individual features (RQ3). Section 5.5 shows examples of usage to demonstrate how our approach can improve developers' bug-fixing practice (RQ4).

5.1 Performance

We first address *RQ1*: What is the predictive power of the two-phase model in recommending files to fix? We present the *likelihood*, *precision* and *recall* values in Figs. 4, 5, and 6, respectively. Since the model recommends the top-k files, the performance depends on the value of k. The X-axis of the figures represents the k value, which ranges from 1 to 10.

When recommending only the top one file (i.e., $k = 1$), the two-phase model's likelihood ranges from 19 to 57 percent. The likelihood value grows as k increases. When $k = 10$, the two-phase model yields likelihood between 52 and 88 percent. Suppose there are 10 bug reports. In the best scenario, our two-phase prediction model is able to successfully recommend at least one file to fix for 6 to 9 out of 10 reports, which is very promising.

When $k = 1$, the two-phase model's precision ranges from 6 to 47 percent, with average of 23 percent. The precision ranges from 7 to 11 percent when $k = 10$. These values indicate that the two-phase model can make correct prediction even with a small k.

The average recall of the two-phase model increases from 9 to 33 percent as k grows from 1 to 10. This indicates that when recommending the top-10 files, our model can correctly suggest, on average, $1/3$ of files that need to be fixed for a given bug. In addition, the two-phase model achieves a 60 percent recall value for ff-bookmark when $k = 10$.

> *Our two-phase model successfully predicts files to fix for 52% to 88% of all bug reports, with an average of 70%.*

5.2 Comparison

As shown in Fig. 4, the two-phase model outperforms the one-phase model in prediction likelihood. For example, when recommending the top-10 files, the likelihood of the two-phase model for eight modules ranges from 52 to 88 percent, with an average value of 70 percent. The one-phase model, on the other hand, has an average likelihood of only 44 percent when $k = 10$, which is even less than the lowest prediction likelihood of the two-phase model.

To counteract the problem that rare events are likely to be observed in multiple comparisons, we used Bonferroni correction [58] so that a p-value less than $0.05/4 = 0.0125$ indicates a significant difference between the corresponding pair of models. As shown in Table 2, the two-phase model significantly outperforms the one-phase model for half of the modules.

The two-phase model also manifests higher precision and recall than the one-phase model, as shown in Figs. 5 and 6.

> *The two-phase model outperforms the one-phase model in prediction likelihood, precision, and recall.*

The one-phase model, on the other hand, manifests prediction performance comparable to the Usual Suspects model—the last column of Table 2 shows that the p-values between these two models are greater than 0.0125 for all

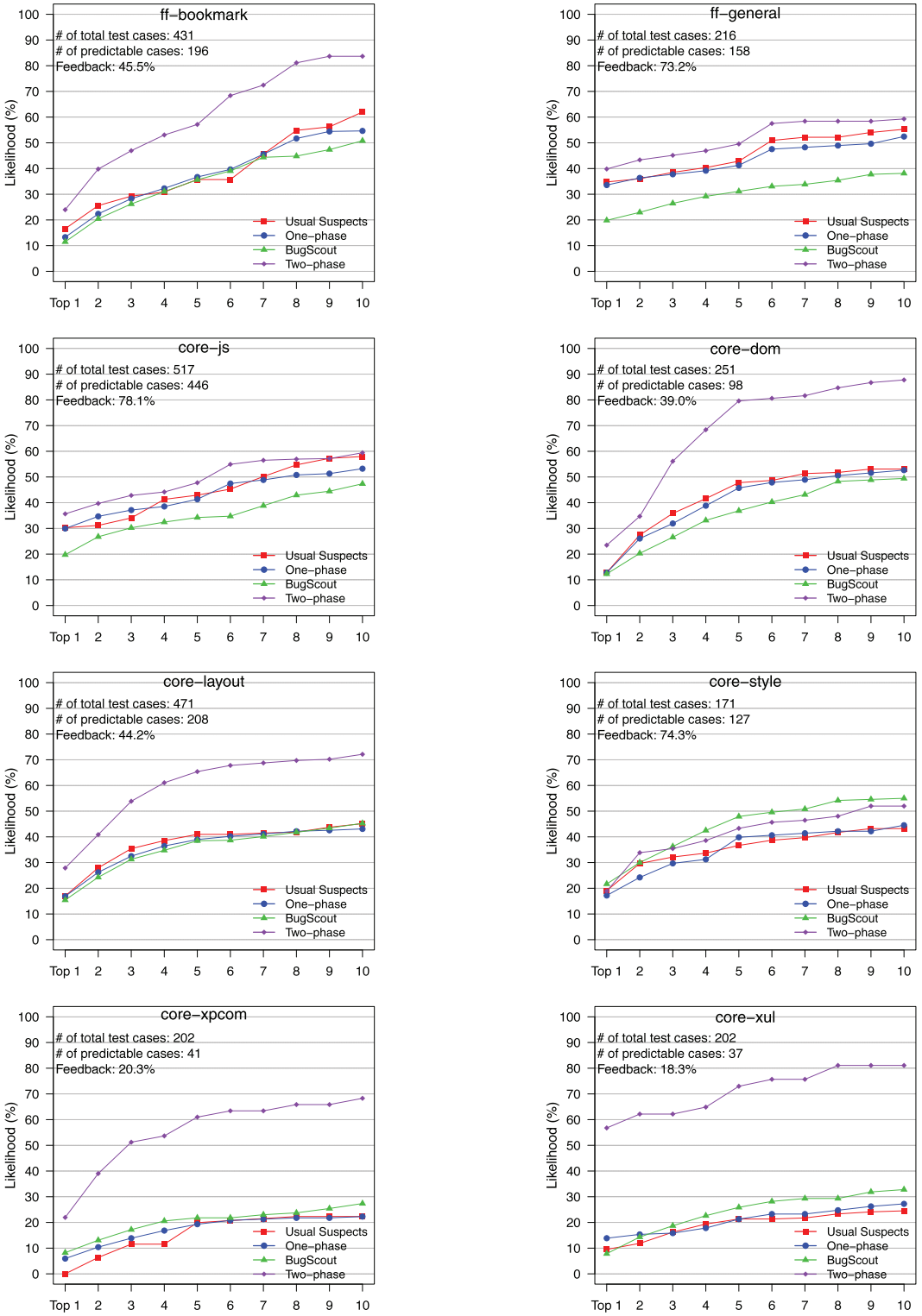


Fig. 4. Prediction likelihood for each module shown in Table 1. The Y-axis represents the *likelihood* values computed by (1). The X-axis represents the k values described in Section 3. In the upper-left corner of each plot, the total number of bug reports in the test set, the number of *predictable* bug reports, and feedback value computed by (5) are shown.

eight modules. BugScout also shows performance similar to the Usual Suspects, as shown in Figs. 4, 5, and 6. One possible reason is that BugScout leverages the defect-proneness information to recommend files to fix, an idea similar to the Usual Suspects model.

Only the two-phase model outperforms the Usual Suspects model, while the one-phase model and BugScout are both on par with the Usual Suspects model.

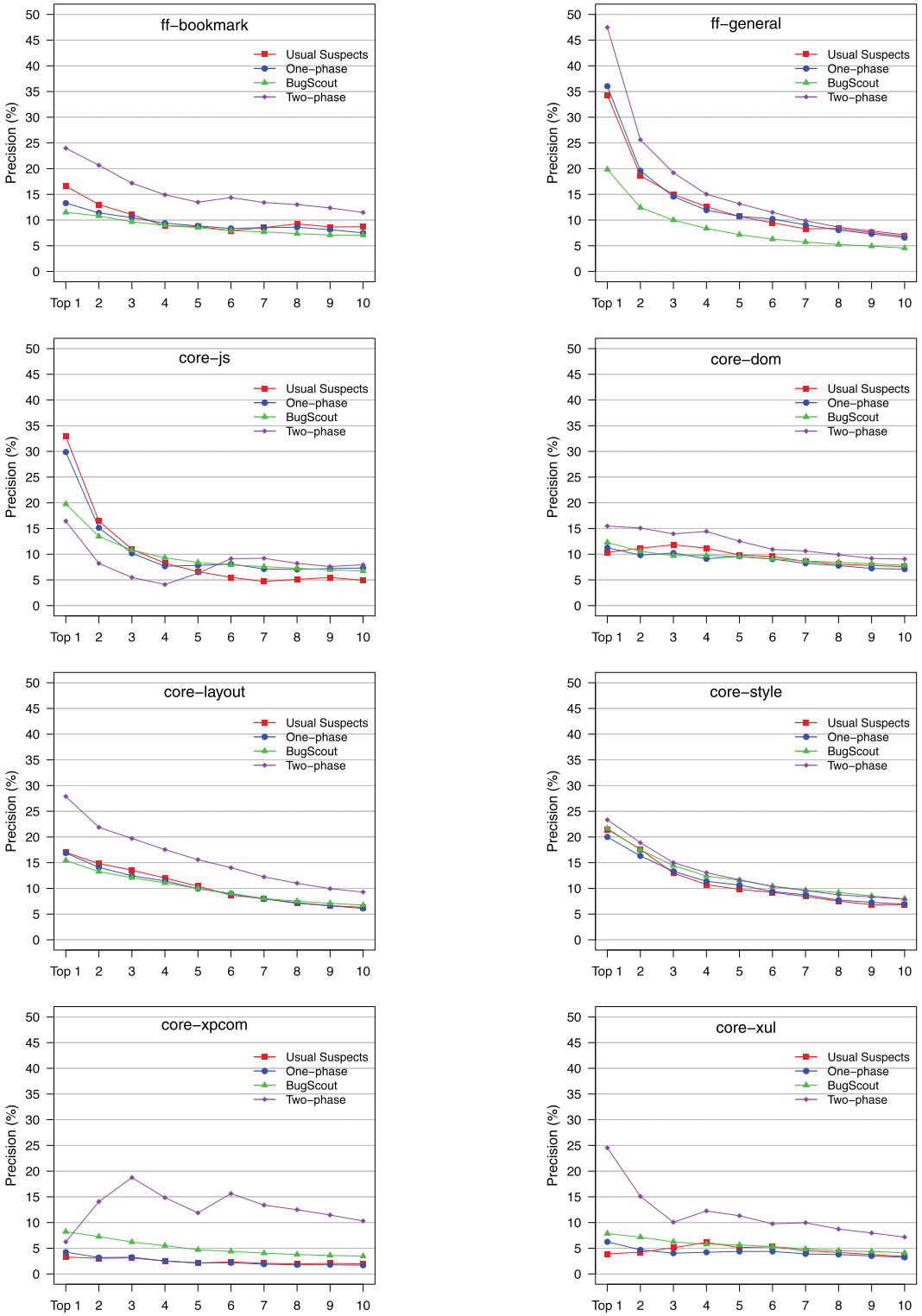


Fig. 5. Prediction precision for each module. The Y-axis represents the average *precision* of all bug reports computed by (2). The X-axis represents the k values described in Section 3.

We also compared the average rank of correctly predicted files for each model (4). As shown in Table 3, the two-phase model has the highest average rank among the four prediction models for six out of eight modules (except for core-js and core-xul). This implies that compared to the other three models, developers might have more confidence in using the two-phase model because it ranks correctly predicted files at a higher position, which could potentially save their inspection time.

The two-phase model ranks correctly predicted files at higher positions than the one-phase, BugScout, and the Usual Suspects models.

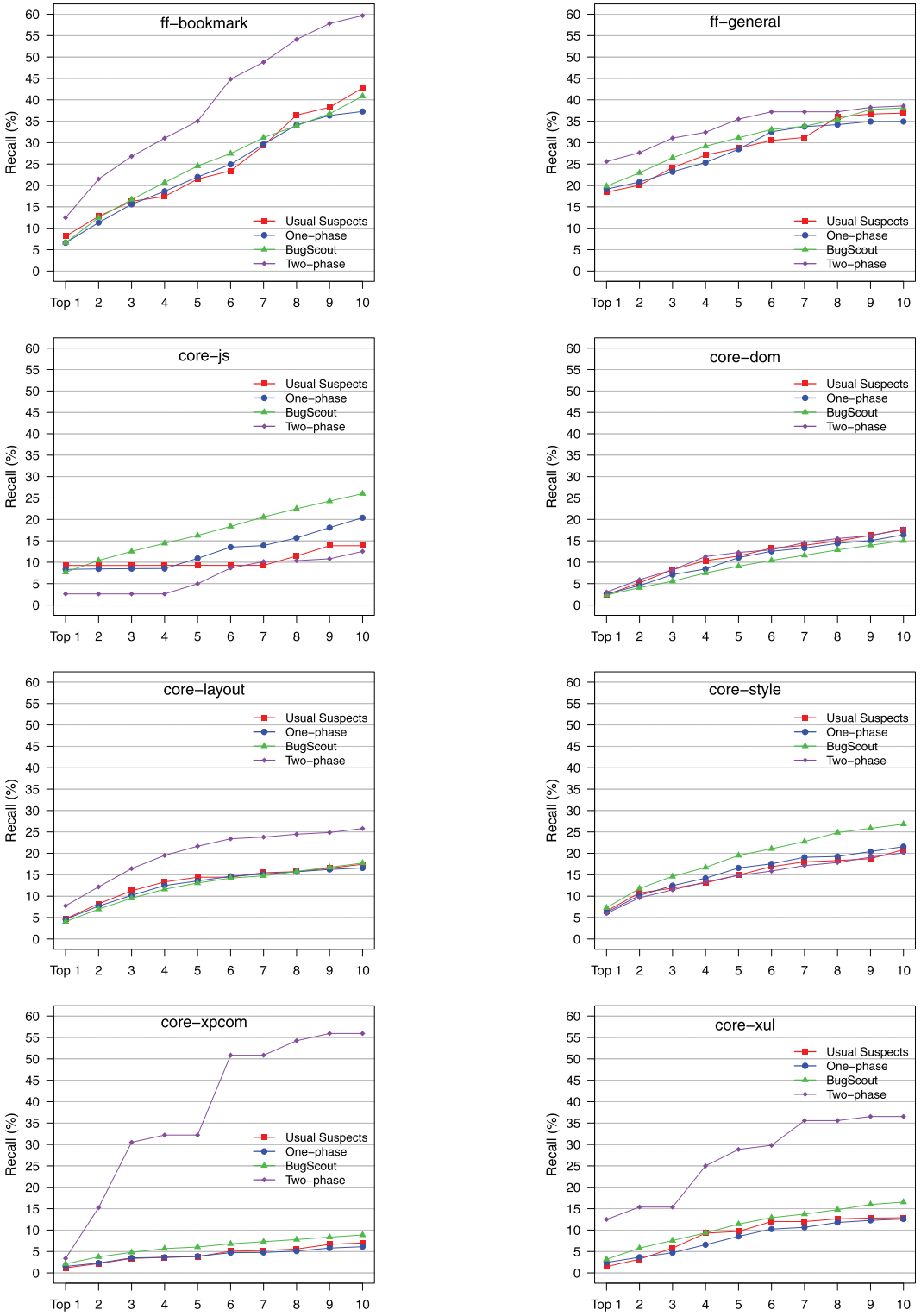


Fig. 6. Prediction recall for each module. The Y-axis represents the average *recall* of all bug reports computed by (3). The X-axis represents the k values.

Overall, the two-phase prediction model can recommend files to fix with high likelihood between 52 and 88 percent and with an average of 70 percent. In addition, the two-phase model has the best performance among the four prediction models: It yields higher prediction likelihood (as well as precision and recall) than the other three models and the difference is statistically significant in most cases. Furthermore, the two-phase model ranks correctly predicted files at higher positions compared to the other models, potentially reducing the time developers spend on the recommended files.

The two-phase model has the best performance among all four prediction models.

TABLE 2
p-Values from the Mann-Whitney Test

Module	Two-phase vs. Usual Suspects	Two-phase vs. One-phase	Two-phase vs. BugScout	One-phase vs. Usual Suspects
ff-bookmark	0.0256	0.0140	**0.0073**	0.7913
ff-general	0.0884	0.0448	**0.0002**	0.4272
core-js	0.3527	0.1051	**0.0029**	0.7960
core-dom	0.0140	**0.0089**	**0.0052**	0.5706
core-layout	**0.0091**	**0.0029**	**0.0021**	0.6775
core-style	0.0887	0.1038	0.3845	0.9097
core-xpcom	**0.0004**	**0.0002**	**0.0006**	0.9095
core-xul	**0.0002**	**0.0002**	**0.0002**	0.6229

Values lower than 0.0125 (shown in bold) indicate that the performance difference between two models is statistically significant.

TABLE 3
The Average Rank of Correctly Predicted Files for Each Module

Module	Usual Suspects	One-phase	BugScout	Two-phase
ff-bookmark	4.24	3.57	3.83	**3.45**
ff-general	2.14	1.80	2.09	**1.10**
core-js	**2.49**	3.81	3.05	5.00
core-dom	3.54	3.42	3.89	**3.27**
core-layout	2.46	2.52	3.09	**2.28**
core-style	2.81	3.05	3.02	**2.75**
core-xpcom	3.75	4.26	3.10	**2.97**
core-xul	**2.88**	3.41	3.47	3.02

Numbers in this table are calculated by (4) when recommending the top 10 files. We highlight the lowest number (i.e., highest rank) across the four prediction models.

5.3 Feedback

We measure *feedback* to address *RQ2*: How many bug reports are predictable? As shown in Fig. 4, the feedback value for the eight modules ranges from 18 to 78 percent, with an average of 49 percent. This indicates that, on average, nearly one-half of bug reports are classified as *predictable* in Phase 1 of our two-phase prediction model.

> *Approximately half of the bug reports are classified as "predictable", i.e., we can predict files to fix for them.*

5.4 Sensitivity Analysis

As described in Section 3.1, we extracted features from the bug report summary, initial description, and metadata to train our prediction models. To better understand which feature or which combination of features is more informative for fix location prediction, we performed a sensitivity analysis on different feature selections to address *RQ3*. We only report results of the sensitivity analysis of the two-phase model's likelihood on the ff-bookmark module here but we observed similar patterns for the other modules.

We tested four possible feature combinations: metadata only, summary only, description only, and summary plus description. Fig. 7 shows the likelihood of each selected feature(s). The two-phase model does not perform so well when trained on the metadata feature only. When we switched to "summary only" and "description only," the prediction likelihood increased. The two-phase model finally reached the best prediction performance when trained on all textual messages ("summary + description").

After comparing the prediction results of these four feature selections, we concluded that textual messages in bug reports are the most important source of information for fix location prediction. However, metadata should not be excluded from training features because when combining all three features together, the two-phase prediction model achieves even better prediction performance (the red "All" curve in Fig. 7, which is indeed the same as reported in Fig. 4).

Note that even when trained only on metadata, the two-phase model still outperforms the one-phase model, which is trained on all three features (see ff-bookmark in Fig. 4). This further illustrates the necessity of filtering out deficient bug reports before the actual prediction.

> *While textual information from bug reports is proved to be the most important source of feature, meta-data serves as complementary feature and should not be excluded.*

5.5 Examples of Use

We now address *RQ4*: Can recommended files effectively help developers? For this purpose, we introduce two cases from ff-bookmark—bug #415757 and #415960—to demonstrate the usefulness of our approach in practice.

To resolve bug #415757, a patch was submitted on 5 February 2008. This patch was reviewed and rejected because it did not successfully resolve the bug. Thereafter, the developer proposed three more patches, all of which were rejected. The main reason was that a file was missing in all the submitted patches. The developer finally realized this and fixed the missing file, `nsNavHistory.h`, in the final patch. This final patch, which was submitted two days after the first submission, was finally accepted.

In the evaluation, our two-phase model correctly recommended `nsNavHistory.h` as a candidate file to fix for bug #415757. In particular, `nsNavHistory.h` ranks

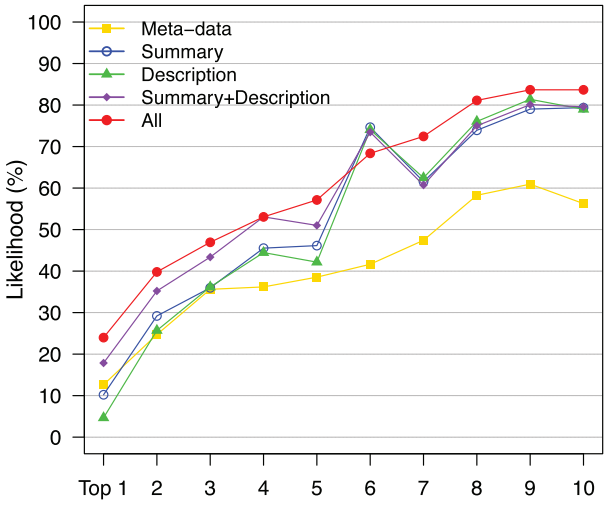


Fig. 7. Feature sensitivity analysis on ff-bookmark. When trained on different feature selections, the likelihood of the two-phase prediction model varies. Better performance implies relatively strong indicative power of the corresponding feature(s).

fifth among the top-10 recommendations, encouraging developers to pay more attention to this file.

The resolution of bug #415960, which experienced an even larger number of patch rejections, further suggests that figuring out the right files to fix is not a trivial task. Between the third and fourth patches, the developer in charge explained why he forgot to include a file in the patch:

> *...moving the last patch to toolkit led me to miss the changes to* `editBookmarkOverlay.xul`*...*

After the fifth patch submission, the reviewers noticed that the patch submitter had chosen the wrong file to capture a key stroke and suggested looking up another file `browser-places.js`.

In the case of the above two missing files in resolving bug #415960, the two-phase model did not recommend `editBookmarkOverlay.xul` but it successfully predicted `browser-places.js` as the fourth candidate file to fix. In addition, we noticed that after the first patch was created on 5 February 2008, it took developers nearly one month to figure out that the file `browser-places.js` was actually missing.

If our two-phase prediction model was deployed in the above cases, it would have recommended files to fix in advance, saving developers' efforts on finding the missing files or reviewing incomplete patches. Moreover, the costly bug fix delay of up to one month could have been avoided.

The investigation of two real-world bug resolution cases suggests that our recommendations are effective and useful.

6 Discussion

The Usual Suspects baseline. The performances of Usual Suspects, the one-phase prediction model, and BugScout [1] are not significantly different from each other for all modules. In some cases, the Usual Suspects model shows better performance (Fig. 4) even though it has a very simple prediction model. This implies that previously fixed files are more likely to be fixed again when a new bug is encountered [52]. Thus, the Usual Suspects model can be a strong defect predictor, comparable to machine learning-based predictors.

Feedback and likelihood. For modules core-style and core-js, whose feedback is 74 and 78 percent, respectively, the likelihood of the two-phase prediction model is not significantly different from the one-phase model. The reason is that when feedback gets closer to 1, nearly all bug reports are classified as "predictable" and no report is filtered out. In such cases, the two-phase prediction actually behaves like the one-phase prediction.

Using stack traces in prediction. Recent observations indicate that crash stack traces facilitate bug resolution activities including fault localization [59], [60]. In particular, crash stack traces might help our fix location prediction because they possibly contain files (represented as file names in stack frames) that are buggy. However, our investigation shows that, on average, only 0.73 percent of bug reports in our subjects contain stack traces in the initial description (Max.: 2.34 percent, Min.: 0.08 percent). Other bug reports may contain stack traces in the comment thread, which is not considered in our feature extraction process.[4] Since we do not have enough stack trace data in the bug reports in our subjects, it is infeasible to confirm whether stack traces contribute to our fix location prediction and evaluate its significance at this point. This remains as our future work.

7 Threats to Validity

- *Subjects are all open source projects*. Since we only use open source projects for evaluation, the results might not be generalizable to closed-source projects. Although recent open source projects have their own quality assurance (QA) teams, the support may not be complete compared to commercial projects. Hence, some *accepted* patches might have missing files, which may lead to deviations in the accuracy of our results.
- *Projects examined might not be representative*. Bug reports of only one open source community, Mozilla, are examined in this paper. Since we intentionally chose this community from which we could extract high-quality bug reports, the absolute prediction results reported in this paper might not be generalizable to other projects. However, the relative improvement of the two-phase prediction over one-phase prediction is less likely to be affected by this threat.
- *Our evaluation method can be biased*. We split the collected bug reports into two sets (70 percent for training and 30 percent for testing) to evaluate our approach. Different splits may yield different accuracies due to concept drift [61]. In addition, we measure the prediction accuracy using likelihood, which considers the prediction to be correct if at least one of the recommended files matches the actual patch file. Other types of measurements might yield different interpretation of the prediction results.

8 Conclusion

Si tacuisses, philosophus mansisses—"If you had been silent, you would have remained a philosopher." This adage, attributed to the Latin philosopher Boethius of the late fifth century, also applies to recommender systems. By staying silent when it does not have enough confidence, our two-phase model avoids misleading recommendations that would otherwise destroy confidence in the prediction model [62]. *If* our approach recommends a location to be fixed, which happens for almost half of the bug reports, 70 percent of the recommendations point to correct files. Since our approach only requires the initial bug report, it can be applied as soon as a problem is reported, providing tangible benefits for debugging.

Our future work will focus on the following topics:

- *Fine-grained defect localization*. Currently, our prediction takes place only at the file level. However, in certain cases, we may be able to predict classes,

4. Please refer to bug reports #242207 and #255027.

methods, and even statements to be fixed; in other cases, we may only be able to predict the package, instead of the file, in which the bug should be fixed. We are currently working on prediction models in which we can adapt prediction granularity while keeping the confidence constant.

- *Performance improvement*. Although our two-phase model outperforms the other models, the precision and recall values for some modules (e.g., core-js and core-dom) are low. We plan to apply advanced techniques such as feature selection [63], [64] to improve the precision and recall values.
- *Bug triaging*. The two-phase model potentially reduces developers' inspection efforts by suggesting possible locations to fix. In addition to the debugging task, our recommendation can also facilitate *bug triaging*, a process of identifying the right developer(s) to address the bug [65]. Given the recommended "suspicious" files for a bug, a project manager could better decide who to assign the bug to and which alternate developers the bug can be tossed to [55], [65], [66].

ACKNOWLEDGMENTS

Rahul Premraj, Sascha Just, and Kim Herzig provided helpful feedback on earlier revisions of this paper. Sunghun Kim is the corresponding author for this paper.

REFERENCES

[1] A.T. Nguyen, T.T. Nguyen, J. Al-Kofahi, H.V. Nguyen, and T. Nguyen, "A Topic-Based Approach for Narrowing the Search Space of Buggy Files from a Bug Report," *Proc. IEEE/ACM 26th Int'l Conf. Automated Software Eng.,* pp. 263-272, Nov. 2011.

[2] J. Zhou, H. Zhang, and D. Lo, "Where Should the Bugs Be Fixed?—More Accurate Information Retrieval-Based Bug Localization Based on Bug Reports," *Proc. 34th Int'l Conf. Software Eng.,* pp. 14-24, June 2012.

[3] C. Liu, X. Yan, L. Fei, J. Han, and S.P. Midkiff, "SOBER: Statistical Model-Based Bug Localization," *Proc. 10th European Software Eng. Conf. Held Jointly with 13th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 286-295, 2005.

[4] B. Liblit, M. Naik, A.X. Zheng, A. Aiken, and M.I. Jordan, "Scalable Statistical Bug Isolation," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 15-26, 2005.

[5] B. Liblit, A. Aiken, A.X. Zheng, and M.I. Jordan, "Bug Isolation via Remote Program Sampling," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 141-154, 2003.

[6] J.A. Jones, M.J. Harrold, and J. Stasko, "Visualization of Test Information to Assist Fault Localization," *Proc. 24th Int'l Conf. Software Eng.,* pp. 467-477, 2002.

[7] H. Cleve and A. Zeller, "Locating Causes of Program Failures," *Proc. 27th Int'l Conf. Software Eng.,* pp. 342-351, 2005.

[8] M. Burger and A. Zeller, "Minimizing Reproduction of Software Failures," *Proc. Int'l Symp. Software Testing and Analysis,* pp. 221-231, 2011.

[9] Y. Brun and M.D. Ernst, "Finding Latent Code Errors via Machine Learning over Program Executions," *Proc. 26th Int'l Conf. Software Eng.,* pp. 480-490, 2004.

[10] X. Ren, B.G. Ryder, M. Stoerzer, and F. Tip, "Chianti: A Change Impact Analysis Tool for Java Programs," *Proc. 27th Int'l Conf. Software Eng.,* pp. 664-665, 2005.

[11] O.C. Chesley, X. Ren, B.G. Ryder, and F. Tip, "Crisp—A Fault Localization Tool for Java Programs," *Proc. 29th Int'l Conf. Software Eng.,* pp. 775-779, May 2007.

[12] M. Stoerzer, B.G. Ryder, X. Ren, and F. Tip, "Finding Failure-Inducing Changes in Java Programs Using Change Classification," *Proc. 14th ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 57-68, 2006.

[13] M. Weiser, "Program Slicing," *Proc. Fifth Int'l Conf. Software Eng.,* pp. 439-449, 1981.

[14] M. Weiser, "Programmers Use Slices When Debugging," *Comm. ACM,* vol. 25, no. 7, pp. 446-452, July 1982.

[15] B. Breech, M. Tegtmeyer, and L. Pollock, "Integrating Influence Mechanisms into Impact Analysis for Increased Precision," *Proc. 22nd IEEE Int'l Conf. Software Maintenance,* pp. 55-65, 2006.

[16] M. Acharya and B. Robinson, "Practical Change Impact Analysis Based on Static Program Slicing for Industrial Software Systems," *Proc. 33rd Int'l Conf. Software Eng.,* pp. 746-755, 2011.

[17] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang, "PSE: Explaining Program Failures via Postmortem Static Analysis," *Proc. 12th ACM SIGSOFT Int'l Symp. Foundations Software Eng.,* pp. 63-72, 2004.

[18] N. Ohlsson and H. Alberg, "Predicting Fault-Prone Software Modules in Telephone Switches," *IEEE Trans. Software Eng.,* vol. 22, no. 12, pp. 886-894, Dec. 1996.

[19] S. Kim, J.E. James Whitehead, and Y. Zhang, "Classifying Software Changes: Clean or Buggy?" *IEEE Trans. Software Eng.,* vol. 34, no. 2, pp. 181-196, Mar./Apr. 2008.

[20] R. Moser, W. Pedrycz, and G. Succi, "A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction," *Proc. 30th Int'l Conf. Software Eng.,* pp. 181-190, 2008.

[21] A.E. Hassan, "Predicting Faults Using the Complexity of Code Changes," *Proc. 31st Int'l Conf. Software Eng.,* pp. 78-88, 2009.

[22] M. D'Ambros, M. Lanza, and R. Robbes, "An Extensive Comparison of Bug Prediction Approaches," *Proc. IEEE Seventh Working Conf. Mining Software Repositories,* pp. 31-41, May 2010.

[23] T. Lee, J. Nam, D. Han, S. Kim, and H.P. In, "Micro Interaction Metrics for Defect Prediction," *Proc. 19th ACM SIGSOFT Symp. and 13th European Conf. Foundations of Software Eng.,* pp. 311-321, 2011.

[24] A.T.T. Ying, G.C. Murphy, R. Ng, and M.C. Chu-Carroll, "Predicting Source Code Changes by Mining Change History," *IEEE Trans. Software Eng.,* vol. 30, no. 9, pp. 574-586, Sept. 2004.

[25] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," *Proc. 26th Int'l Conf. Software Eng.,* pp. 563-572, 2004.

[26] A. Marcus, A. Sergeyev, V. Rajlich, and J.I. Maletic, "An Information Retrieval Approach to Concept Location in Source Code," *Proc. 11th Working Conf. Reverse Eng.,* pp. 214-223, Nov. 2004.

[27] D. Poshyvanyk, A. Marcus, V. Rajlich, Y.-G. Gueheneuc, and G. Antoniol, "Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification," *Proc. 14th IEEE Int'l Conf. Program Comprehension,* pp. 137-148, 2006.

[28] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval," *IEEE Trans. Software Eng.,* vol. 33, no. 6, pp. 420-432, June 2007.

[29] S.K. Lukins, N.A. Kraft, and L.H. Etzkorn, "Bug Localization Using Latent Dirichlet Allocation," *Information and Software Technology,* vol. 52, no. 9, pp. 972-990, Sept. 2010.

[30] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, "On the Use of Relevance Feedback in IR-Based Concept Location," *Proc. 25th IEEE Int'l Conf. Software Maintenance,* pp. 351-360, Sept. 2009.

[31] B. Ashok, J. Joy, H. Liang, S.K. Rajamani, G. Srinivasa, and V. Vangala, "DebugAdvisor: A Recommender System for Debugging," *Proc. Seventh Joint Meeting European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng.,* pp. 373-382, 2009.

[32] D. Shepherd, Z.P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker, "Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns," *Proc. Sixth Int'l Conf. Aspect-Oriented Software Development,* pp. 212-224, 2007.

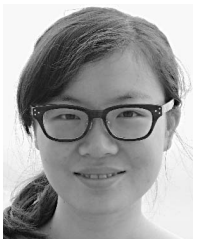
[33] S. Rao and A. Kak, "Retrieval from Software Libraries for Bug Localization: A Comparative Study of Generic and Composite Text Models," *Proc. Eighth Working Conf. Mining Software Repositories,* pp. 43-52, 2011.

[34] T.J. Biggerstaff, B.G. Mitbander, and D. Webster, "The Concept Assignment Problem in Program Understanding," *Proc. 15th Int'l Conf. Software Eng.,* pp. 482-498, 1993.

[35] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: Finding Relevant Functions and Their Usage," *Proc. 33rd Int'l Conf. Software Eng.* pp. 111-120, 2011.

[36] G.A. Liebchen and M. Shepperd, "Data Sets and Data Quality in Software Engineering," *Proc. Fourth Int'l Workshop Predictor Models Software Eng.*, pp. 39-44, 2008.
[37] R. Balzer, "Tolerating Inconsistency," *Proc. 13th Int'l Conf. Software Eng.*, pp. 158-165, May 1991.
[38] E. Kocaguneli, T. Menzies, A. Bener, and J. Keung, "Exploiting the Essential Assumptions of Analogy-Based Effort Estimation," *IEEE Trans. Software Eng.*, vol. 38, no. 2, pp. 425-438, Mar./Apr. 2012.
[39] T.M. Khoshgoftaar and N. Seliya, "The Necessity of Assuring Quality in Software Measurement Data," *Proc. 10th Int'l Symp. Software Metrics*, pp. 119-130, 2004.
[40] J. Aranda and G. Venolia, "The Secret Life of Bugs: Going Past the Errors and Omissions in Software Repositories," *Proc. 31st Int'l Conf. Software Eng.*, pp. 298-308, 2009.
[41] A. Mockus, "Missing Data in Software Engineering," *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D.I.K. Sjberg, eds., pp. 185-200, Springer, 2008.
[42] G. Liebchen, B. Twala, M. Shepperd, M. Cartwright, and M. Stephens, "Filtering, Robust Filtering, Polishing: Techniques for Addressing Quality in Software Data," *Proc. First Int'l Symp. Empirical Software Eng. and Measurement*, pp. 99-106, 2007.
[43] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with Noise in Defect Prediction," *Proc. 33rd Int'l Conf. Software Eng.*, pp. 481-490, 2011.
[44] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and Balanced? Bias in Bug-Fix Data Sets," *Proc. Seventh Joint Meeting European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations Software Eng.*, pp. 121-130, 2009.
[45] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "Relink: Recovering Links between Bugs and Changes," *Proc. 19th ACM SIGSOFT Symp. and 13th European Conf. Foundations of Software Eng.*, pp. 15-25, 2011.
[46] D.D. Lewis, "Naive (Bayes) at Forty: The Independence Assumption in Information Retrieval," *Proc. 10th European Conf. Machine Learning*, C. Nédellec and C. Rouveirol, eds., pp. 4-15, 1998.
[47] E. Alpaydin, *Introduction to Machine Learning*. MIT Press, 2004.
[48] J.D.M. Rennie, "Improving Multi-Class Text Classification with Naive Bayes," master's thesis, Massachusetts Inst. of Technology, 2001.
[49] P. Hooimeijer and W. Weimer, "Modeling Bug Report Quality," *Proc. IEEE/ACM 22nd Int'l Conf. Automated Software Eng.*, pp. 34-43, 2007.
[50] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schröter, and C. Weiss, "What Makes a Good Bug Report?" *IEEE Trans. Software Eng.*, vol. 36, no. 5, pp. 618-643, Sept./Oct. 2010.
[51] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The Missing Links: Bugs and Bug-Fix Commits," *Proc. 16th ACM SIGSOFT Symp. Foundations of Software Eng.*, pp. 97-106, 2010.
[52] S. Kim, T. Zimmermann, E.J. Whitehead Jr., and A. Zeller, "Predicting Faults from Cached History," *Proc. 29th Int'l Conf. Software Eng.*, pp. 489-498, 2007.
[53] T.M. Khoshgoftaar, E.B. Allen, N. Goel, A. Nandi, and J. McMullan, "Detection of Software Modules with High Debug Code Churn in a Very Large Legacy System," *Proc. Seventh Int'l Symp. Software Reliability Eng.*, pp. 364-371, 1996.
[54] A. Hassan and R. Holt, "The Top Ten List: Dynamic Fault Prediction," *Proc. 21st IEEE Int'l Conf. Software Maintenance*, pp. 263-272, 2005.
[55] G. Jeong, S. Kim, and T. Zimmermann, "Improving Bug Triage with Bug Tossing Graphs," *Proc. Seventh Joint Meeting European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng.*, pp. 111-120, 2009.
[56] H.B. Mann, "On a Test of Whether One of Two Random Variables Is Stochastically Larger than the Other," *The Annals of Math. Statistics*, vol. 18, no. 1, pp. 50-60, Mar. 1947.
[57] D.C. Montgomery and G.C. Runger, *Applied Statistics and Probability for Engineers*. John Wiley & Sons, 1994.
[58] O.J. Dunn, "Multiple Comparisons among Means," *J. Am. Statistical Assoc.*, vol. 56, no. 293, pp. 52-64, Mar. 1961.
[59] A. Schröter, N. Bettenburg, and R. Premraj, "Do Stack Traces Help Developers Fix Bugs?" *Proc. Seventh Working Conf. Mining Software Repositories*, pp. 118-121, 2010.
[60] H. Seo and S. Kim, "Predicting Recurring Crash Stacks," *Proc. 27th IEEE/ACM Int'l Conf. Automated Software Eng.*, pp. 180-89, 2012.
[61] G. Widmer and M. Kubat, "Learning in the Presence of Concept Drift and Hidden Contexts," *Machine Learning*, vol. 23, pp. 69-101, 1996.
[62] L. Swartz, "Why People Hate the Paperclip: Labels, Appearance, Behavior and Social Responses to User Interface Agents," master's thesis, Stanford Univ., 2003.
[63] A. Miller, *Subset Selection in Regression*, second ed. Chapman and Hall/CRC, Apr. 2002.
[64] S. Shivaji, E. Whitehead Jr., R. Akella, and S. Kim, "Reducing Features to Improve Code Change Based Bug Prediction," *IEEE Trans. Software Eng.*, vol. 39, no. 4, pp. 552-569, Apr. 2013.
[65] J. Anvik, L. Hiew, and G.C. Murphy, "Who Should Fix This Bug?" *Proc. 28th Int'l Conf. Software Eng.*, pp. 361-370, 2006.
[66] P.J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "'Not My Bug!' and Other Reasons for Software Bug Report Reassignments," *Proc. ACM Conf. Computer Supported Cooperative Work*, pp. 395-404, 2011.

Dongsun Kim received the BEng, MS, and PhD degrees in computer science and engineering from Sogang University, Seoul, Korea, in 2003, 2005, and 2010, respectively. He is currently a postdoctoral fellow at the Hong Kong University of Science and Technology. His research interests include mining software repositories, automatic patch generation, and static analysis. He is a member of the IEEE.



Yida Tao received the BSc degree in computer science from Nanjing University and is working toward the PhD degree in the Department of Computer Science and Engineering at the Hong Kong University of Science and Technology (HKUST). Her research interests include software evolution and program comprehension. She is a student member of the IEEE.

Sunghun Kim received the PhD degree from the Department of Computer Science, University of California, Santa Cruz, in 2006. He is an assistant professor of computer science at the Hong Kong University of Science and Technology. He was a postdoctoral associate at the Massachusetts Institute of Technology and a member of the Program Analysis Group. He was the chief technical officer (CTO) and led a 25-person team for six years at the Nara Vision Co. Ltd., a leading Internet software company in Korea. His core research area is software engineering, focusing on software evolution, program analysis, and empirical studies. He is a member of the IEEE.

Andreas Zeller is a full professor for software engineering at Saarland University in Saarbrücken, Germany. His research concerns the analysis of large software systems and their development process; his students are funded by companies like Google, Microsoft, or SAP. In 2010, he was inducted as a fellow of the ACM for his contributions to automated debugging and mining software archives. In 2011, he received an ERC Advanced Grant, Europe's highest and most prestigious individual research grant, for work on specification mining and test case generation. He is a member of the IEEE.