Continuous Software Bug Prediction

Song Wang
York University, Toronto, Canada
wangsong@yorku.ca

Jaechang Nam
Handong Global University, Pohang, South Korea
jcnam@handong.edu

Junjie Wang
Chinese Academy of Sciences, Beijing, China
junjie@iscas.ac.cn

Nachiappan Nagappan
Facebook, Seattle, USA
nachiappan.nagappan@gmail.com

ABSTRACT

Background: Software bug prediction plays a fundamental role in software quality assurance. Many software bug prediction models have been proposed and evaluated on a set of well-known benchmark datasets. We conducted pilot studies on the widely used benchmark datasets and observed common issues among them. Specifically, most of existing benchmark datasets consist of randomly selected history versions of software projects, which poses non-trivial threats to the validity of existing bug prediction studies since the real-world software projects often evolve continuously. Yet how to conduct software bug prediction in the real-world continuous software development scenarios is not well studied.

Aims: In this paper, to bridge the gap between current software bug prediction practice and real-world continuous software development, we propose new approaches to conducting bug prediction in real-world continuous software development regarding model building, updating, and evaluation.

Method: For model building, we propose ConBuild, which leverages distributional characteristics of bug prediction data to guide the training version selection. For model updating, we propose ConUpdate, which leverages the evolution of distributional characteristics of bug prediction data between versions to guide the reuse or update of bug prediction models in continuous software development. For model evaluation, we propose ConEA, which leverages the evolution of buggy probability of files between versions to conduct effort-aware evaluation.

Results: Experiments on 120 continuously release versions that span across six large-scale open-source software systems show the practical value of our approaches.

Conclusions: This paper provides new insights and guidelines for conducting software bug prediction in the context of continuous software development.

CCS CONCEPTS
• Software and its engineering → Software testing and debugging; Empirical software validation.

KEYWORDS
Empirical software engineering, software quality, software defect prediction, continuous software development

ACM Reference Format:

1 INTRODUCTION

Software bug prediction techniques have been proposed to guide bug detection and reduce software development costs [47, 66, 86]. Over the past decades, we have seen many successful cases of applying bug prediction models to help improve software quality from many software organizations, e.g., Samsung Electronics [30], AT&T [53], Microsoft Research [48, 49, 54, 83, 84], Google [55], and Cisco [43, 58, 65]. To evaluate the performance of bug prediction models, most of the existing studies adopt the widely used and publicly available datasets, e.g., PROMISE dataset that contains 71 release versions from 33 open source projects [26], NASA dataset that contains 12 release versions from four projects [60], AEEEM dataset that contains five release versions from five projects [8], SOFTLAB dataset that contains three release versions from one project [68], or Relink dataset that contains three release versions from three open source projects [50, 75], etc. Most of the above datasets were created based on the widely-used heuristic approach, i.e., identifying post-release defects (by using specific keywords/issue IDs) within a specific post-release window period (e.g., six months) [9, 13, 28, 63, 67, 80].

Recently, Yatish et al. [79] proposed to use the affected releases recorded in issue tracking systems to collect defect data more accurately for a given software bug. We refer to their dataset as YatishData, which contains 32 release versions from nine projects. In our two pilot studies (Section 2.2), we evaluated these defect datasets and found some common issues regarding building, updating, and evaluating software bug prediction. These issues pose non-trivial threats to the validity of many defect prediction studies particularly under real-world continuous software development.

First, from our pilot study I (Section 2.2.1), we observed that most defect datasets used in existing bug prediction studies do not fit in with real-world continuous software development. Specifically, these datasets contain randomly selected release versions from software projects. The time span between two release versions (in chronological order) from a bug prediction dataset could be up years, e.g., the two selected versions from project ActiveMQ (in YatishData dataset [79]) have a four-year interval and skipped 11
in-between history release versions. However, an active software project often evolves continuously and releases a large number of continuous versions chronologically [12]. In addition, evaluating a bug prediction model on randomly selected versions introduces non-trivial bias in the its performance. For example, bug prediction models built on different training datasets (i.e., release versions) of project Camel (in PROMISE dataset [26]) have significant different AUC values and the difference can be up to 0.359 and on average is 0.221 (see Figure 1). The randomness of version selection in current widely used defect datasets opens a huge gap between existing software bug prediction studies and real-world software practice, which motivates us to explore bug prediction in the context of continuous software development. Yet, little is known about how to select defect datasets for building bug prediction models in the continuous software development.

Second, our pilot study II (Section 2.2.2) shows that the performance of a bug prediction model often evolves over time. It is commonly known that the data distributions between datasets from different release versions could be significantly different and prediction models built on datasets of earlier release versions could have performance declines on datasets from later release versions because of the data drift problem during software evolution [6, 10]. However, we found that a bug prediction model built on datasets of early release versions does not lose the predictive power sharply. This sheds light on reusing prediction models built on early release versions for saving model-building cost, since building bug prediction models for a newly released version requires extra time and space effort to relabel data, collect features, and tune the parameters of classifiers [65, 67]. Yet, little is known about when to update bug prediction models before they deliver unacceptable performance in the real-world continuous software development.

Third, due to limited resources and human effort, developers often can only inspect a limited number of lines of code given the prediction results of a bug prediction model. Thus, effort-aware evaluation is widely adopted by existing bug prediction models [23, 24, 65, 78]. Most of existing effort-aware evaluation approaches prioritize the files in a version to be inspected by using their buggy probabilities, i.e., checking files with a higher buggy probability first. However, such approach cannot take the evolution of the fault proneness of source files into consideration. For example, suppose that a file f1 was predicted as buggy with a buggy probability of 0.8 on an early release version v1. Then it was predicted as buggy with a probability of 0.51 on a later release version v2. Such a dramatic change on the buggy probability may indicate a potential bug appeared in file f1 between the two versions. Suppose that another file f2 had buggy probability evolved from 0.52 to 0.521 on the same two versions (i.e., from v1 to v2). Existing effort-aware evaluation approaches give higher priority to f2 since f2 has a higher buggy probability in v2, although f1 has a higher chance to be a real bug giving the dramatic change on its buggy probability (from 0 to 0.51) between versions.

To bridge the gap between existing software bug prediction studies and real-world continuous software practice, in this paper, we explore software bug prediction in real-world continuous software development scenarios regarding the building, updating, and evaluating of bug prediction models. Specifically, for model building, we propose ConBuild, which leverages distributional characteristics of data from different versions to guide the training data selection for building bug prediction models in continuous software development. For model updating, we propose ConUpdate, which leverages the evolution of distributional characteristics of data from different versions to guide the reuse or update of bug prediction models in continuous software development. For model evaluation, we propose ConEA, which leverages the difference of the buggy probabilities of files between different versions to conduct effort-aware evaluation of bug prediction results in continuous software development. Experiments on 120 continuously released versions from six software systems show the practical value of our approaches.

This paper makes the following contributions:

- We conduct the first empirical study to explore challenges of current software bug prediction practice in the context of continuous software development regarding the process of bug prediction model building, updating, and evaluation.
- We propose ConBuild to guide the training data selection for building bug prediction models in continuous software development based on the distributional characteristics of bug prediction datasets.
- We propose ConUpdate to guide the reuse or update of bug prediction models in continuous software development based on the evolution of distributional characteristics of bug prediction datasets.
- We propose ConEA to conduct the effort-aware evaluation of bug prediction models in the context of continuous software development by leveraging the difference of the buggy probabilities of files between versions.
- We provide a new benchmark bug prediction dataset of 120 continuously released versions that span across six open-source software systems for facilitating future continuous software bug prediction research.

The rest of this paper are organized as follows. Section 2 describes the background and motivations of this study. Section 3 presents our approaches. Section 4 shows the setup of our experiments. Section 5 presents the results of our study. Section 6 discusses open questions and the threats to the validity of this work. Section 7 surveys the related work. Finally, we summarize this paper in Section 8.

2 BACKGROUND AND MOTIVATION

This section introduces the background of bug prediction and two pilot studies that motivate us to explore bug prediction models in continuous software development.

2.1 Process of File-level Bug Prediction Models

The objective of a file-level bug prediction model is to determine risky files for further software quality assurance activities. [19, 33, 45, 48, 56, 86]. A typical release-based file-level bug prediction model mainly has three steps. The first step is to label the files in an early version as buggy or clean based on post-release defects for each file. Post-release defects are defined as defects that are revealed within a post-release window period (e.g., six months) [56, 72]. One could collect these post-release defects from a Bug Tracking System (BTS) via linking bug reports to its bug-fixing changes.

---

1. https://zenodo.org/record/3830114
Files related to these bug-fixing changes are considered as buggy. Otherwise, the files are labeled as clean. The second step is to collect the corresponding defect features to represent these files. Instances with features and labels are used to train machine learning classifiers. Finally, trained models are used to predict files in a later version as buggy or clean. In this work, we refer to the release version used for building models as a training version, whereas the release version used to evaluate the trained models is referred to as a test version.

2.2 Problems of Existing Bug Prediction Process

2.2.1 Pilot Study I: Nature of Existing Benchmark Data and its Influence. As most of the existing bug prediction models are evaluated on the widely used benchmark datasets, i.e., PROMISE [26], AEEEM [8], SOFTLAB [68], ReLink [50, 75], and YatishData [79], we start from manually analyzing the nature of these datasets to check whether the datasets are in line with real-world continuous software development process, i.e., whether the selected versions from a project are continuous. During the above manual analysis, we exclude three datasets either because the datasets only contain one version from each project (i.e., AEEEM [8] and ReLink [50, 75]) or the dataset did not release version numbers (i.e., SOFTLAB [68]).

Overall, in half of PROMISE projects [26] and all the YatishData projects [79], the selected release versions for bug prediction studies are continuous. We also find that the time span between two selected release versions (in chronological order) from a dataset could be up to years, e.g., the selected latest two release versions from project ActiveMQ in YatishData [79] have a four-year interval with skipping 11 in-between history release versions. However, during the lifetime of a software project, a large number of continuous versions is released chronologically. Thus, the widely used PROMISE [26] and YatishData [79] datasets do not fit in with real-world continuous software development.

We further investigate the potential performance bias of bug prediction models on current widely used bug prediction benchmark datasets. Specifically, we select two projects that were used in both PROMISE [26] and YatishData [79], namely, Camel and Lucene as our experimental subjects to examine how much performance difference a bug prediction model could have with training data from different release versions. For each project, we select 20 continuous release versions (details are in Table 2) and build the ADTree based bug prediction models. We further select recent five release versions (from v16 to v20) as the test versions. For each test version, we build bug prediction models with defect data from each of its previous versions to train prediction models. In total, we have 2 (projects) * 85 (training-test pairs) = 170 prediction experiments. Given a test version (from v16 to v20), we record the AUC values of the models trained on different training versions and obtain the difference between each model and the best model.

Figure 1 shows the performance difference. As we can see, bug prediction models with different training datasets significantly differ in their performance (Wilcoxon signed-rank test, $p < 0.05$). Specifically, on project Camel, the difference of AUC values can be up to 0.359 and on average is 0.221. On the project, Lucene, the difference of AUC values can be up to 0.227 and on average is 0.151.
versions that have similar distributional characteristics with a given training version could generate better performance [52, 74]. In order to verify such assumption, we use the above two models built on v1 from Lucene and Camel as the experiment datasets and check whether there exists correlation between the performance of the models and the similarity of distributional characteristics between the training version (v1) and a test version (from v2 to v20). We use the standard deviation of each bug prediction metric to represent its distributional characteristic in the training and test datasets. We then use the standard deviation values of bug prediction metrics to calculate the cosine similarity between a training dataset and a test dataset to measure their distance, which is widely used in prior studies to measure the similarity [70]. Note that, the goal of this pilot study is not to find the best metrics of measuring the similarity between two release versions, thus we only use standard deviation to measure the distributional characteristic of a dataset. More metrics to capture the distributional characteristics of data are discussed in Section 3.1.

Following the existing studies [39, 85], we use the Spearman rank correlation [46] to measure the correlation between the performance of a bug prediction model on test versions (i.e., AUC) and the similarity between the training version (v1) of the bug prediction model and the test versions. The closer the value of a correlation is to +1 (or -1), the higher two measures are positively (or negatively) correlated. The Spearman correlation values are 0.84 and 0.91 on Lucene and Camel respectively, which confirms that the performance of a bug prediction model on a given test version is correlated to the similarity between the training version and the test version.

Due to the data drift issue in the bug prediction datasets, the performance of a bug prediction model evolves over time. However, a model does not lose the predictive power sharply, which sheds light on model reuse to save model rebuilding effort.

Results of our pilot studies motivate us to look deeper in practicing bug prediction in continuous software development, and inspire us to propose solutions for effectively building, updating, and evaluating bug prediction models in continuous software development by using the evolution information of projects’ distributional characteristics.

3 METHODOLOGY

In this section, we present our solutions to guide building, updating, and evaluating bug prediction models in continuous software development. Specifically, to build effective bug prediction models for a given release version, we propose ConBuild, which leverages the distributional characteristics of bug prediction data to select the appropriate training versions (Section 3.1). To acknowledge the reuse or update of bug prediction models, we propose ConUpdate, which leverages the evolution of the distributional characteristics of bug prediction data to guide the model reuse or update (Section 3.2). In addition, we also propose ConEA, which leverages the differences of buggy probabilities of files between versions to conduct effort-aware evaluation of bug prediction models (Section 3.3). The details of these three approaches are shown in Figure 3.

3.1 ConBuild: Model Building in Continuous Software Development

Motivated by our pilot study II (Section 2.2.2), given a test version \( v_n \), ConBuild leverages the similarity of distributional characteristics between a candidate training version and the test version \( v_n \) to identify an appropriate training version from history to build a bug prediction model for the test version \( v_n \).

To capture the characteristics of a bug prediction dataset properly, we use 11 widely used indicators in existing studies to describe the distributional characteristics [21, 34, 38], which are shown in Table 1. In this work, we consider the characteristics of a dataset as the collection of distributional characteristics of each bug prediction metric (as listed in Table 3). Combining these indicators together, we generate a vector of 594 indicators (i.e., 54 bug prediction metrics multiplying 11 indicators) to describe the distributional characteristics of a given training dataset or test dataset. Following our pilot study II (Section 2.2.2), we use cosine similarity to measure their similarity.

Given a test version \( v_n \) and each of its previous versions (from \( v_1 \) to \( v_{n-1} \)), ConBuild first collects the value for each bug prediction metric and obtains the characteristic indicators as described in Table 1. Then, ConBuild applies cosine similarity to calculate the similarities between \( v_n \) and each of its previous versions by using the indicator vectors. Finally, ConBuild ranks the previous versions based on the similarities and recommends the top version as the training version to build a bug prediction model. The details of ConBuild are described in Algorithm 1.

3.2 ConUpdate: Model Reuse/Update in Continuous Software Development

Given a newly released version, we aim to acknowledge developers whether to reuse previous models for saving efforts or to update previous models, e.g., relabeling data with additional bug information and finding an appropriate training version to rebuild a bug prediction model. Specifically, we propose ConUpdate, which leverages the evolution of distributional characteristics of datasets to guide bug prediction model reuse or update for a newly released version in continuous software development.

For a newly released version \( v_n \), ConUpdate recommends either Reuse or Update operations. If both the previous version \( v_{n-1} \) and the training version of \( v_{n-1} \)’s applied model (i.e., the bug
Algorithm 1 ConBuild: training version selection

Require:
version to be predicted \( v_n \);
history version list \( V_{\text{trainings}} \);

Ensure:
recommended version \( v_r \);
1: initialize a Map \( S \);
2: for each version \( v \) in \( V_{\text{trainings}} \) do
3: extract characteristic indicators listed in Table 1;
4: compute the similarity \( s \) between \( v \) and \( v_n \);
5: put \( v \) and \( s \) into \( S \);
6: end for
7: sort \( S \) by \( s \);
8: return top one from \( S \);

Table 1: Descriptions of indicators used to describe the distributional characteristics of a dataset used in this work.

<table>
<thead>
<tr>
<th>Indicator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode</td>
<td>The most frequent value</td>
</tr>
<tr>
<td>Median</td>
<td>The middle value</td>
</tr>
<tr>
<td>Mean</td>
<td>The average value</td>
</tr>
<tr>
<td>Minimum</td>
<td>The smallest value</td>
</tr>
<tr>
<td>Maximum</td>
<td>The largest value</td>
</tr>
<tr>
<td>First Quartile</td>
<td>The value cutting off 25% lowest cases</td>
</tr>
<tr>
<td>Third Quartile</td>
<td>The value cutting off 75% lowest cases</td>
</tr>
<tr>
<td>Variance</td>
<td>The arithmetic mean of the squared deviation of the Mean to values in a dataset</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>The square root of the variance</td>
</tr>
<tr>
<td>Skewness</td>
<td>A measure of the asymmetry of a dataset</td>
</tr>
<tr>
<td>Kurtosis</td>
<td>A measure of the peakedness of a dataset</td>
</tr>
</tbody>
</table>

Algorithm 2 ConUpdate: bug prediction model reuse/update

Require:
newly released version to be predicted \( v_n \);
previous version \( v_{n-1} \);
training version of \( v_{n-1} \)'s applied model \( A_{n-1} \);
all existing version list \( V_{\text{all}} \);
threshold \( k \);

Ensure:
recommended operation;
1: initialize a Map \( S \);
2: for each version \( v \) in \( V_{\text{all}} \) do
3: extract characteristics indicators listed in Table 1;
4: compute the similarity \( s \) between \( v \) and \( v_n \);
5: put \( v \) and \( s \) into \( S \);
6: end for
7: sort \( S \) by \( s \);
8: if \( v_{n-1} \) and \( A_{n-1} \) are in top \( k \) of \( S \) then
9: return Reuse;
10: else
11: return Update;
12: end if

However, such approaches do not take the evolution of fault proneness of source files into consideration. We assume files that have a dramatic change in the fault proneness between release versions may indicate a potential risk and should be given higher priority. In this work, we propose ConEA, to evaluate bug prediction models in the continuous software development, which gives files that have a dramatic change in the fault proneness (i.e., buggy probability) between the two release versions a higher priority.

Given the predicted buggy files in the version \( v_n \), ConEA first calculates the probability differences between version \( v_n \) and its previous version \( v_{n-1} \). Second, ConEA ranks these files based on the probability differences and then prioritizes their orders to be inspected with the ranking. Note that, version \( v_n \) would have new files, e.g., introduced by new features, which are not in the previous versions. For the newly added files in version \( v_n \), ConEA uses their buggy probabilities in \( v_n \) as the probability differences to rank them together with other files to be inspected. Given the example in Section 1, i.e., file f1 (whose probability evolves from 0 to 0.51) and file f2 (whose probability evolves from 0.52 to 0.521), ConEA gives file f1 a higher priority than file f2 since f1 has a higher probability difference value (i.e., 0.51) than that of f2 (i.e., 0.01).

Algorithm 3 describes the details of ConEA.

3.3 ConEA: Effort-aware Evaluation in Continuous Software Development

As described in Section 1, most of current effort-aware evaluation approaches only focus on files with high buggy probabilities, i.e., rank the probabilities of the predicted buggy files and prioritize the files to be inspected based on the ranking [23, 24, 42, 65, 78].
Algorithm 3 ConEA: effort-aware evaluation

Require:
- predicted result of version \( v_n; R_n \)
- predicted result of version \( v_{n-1}; R_{n-1} \)

Ensure:
- result map \( F \); // maintaining inspection priority of files
1: for each file \( f \) in \( R_n \) do
2:     if \( f \) in \( R_{n-1} \) and \( f \) is true buggy in \( R_{n-1} \) then
3:         calculate the probability \( \text{diff} \) in \( R_n \) and \( R_{n-1} \);
4:         put \( \text{diff} \) and its probability from \( R_n \) into \( F \);
5:     end if
6:     if \( f \) not in \( R_{n-1} \) and \( f \) is predicted as buggy in \( R_n \) then
7:         put \( f \) and its probability from \( R_n \) into \( F \);
8:     end if
9: end for
10: sort map \( F \) by \( \text{diff} \);
11: return \( F \);

\( f(x[i]; \text{defects?}) \) and issue/report IDs in the commit messages. Then we label files modified in these bug-fixing commits as buggy. To make our dataset more representative, we select two unique projects from PROMISE [26] (i.e., Ant and Poi), two unique projects from Ya\( t\)i\( s\)h\( D\)a\( t\)a [79] (i.e., ActiveMQ and HBase), and two shared projects of them (i.e., Lucene and Camel). For each project, we collect recent 20 continuous release versions before 2018-12-31 (i.e., the time when we collect data). Table 2 shows the details of the six projects and the 20 continuous release versions selected from each project.

### 4.2 Bug Prediction Metrics

Prior studies introduced many different bug prediction metrics to build accurate bug prediction models. Since the goal of this study is not to find the best bug prediction metrics, we adopt the widely used code metrics, e.g., lines of code [36], code complexity metrics (e.g., McCabe Cyclomatic) [40, 44], and object-oriented metrics (e.g., coupling between object classes) [18], etc., to build bug prediction models. We use the Understand\(^2\) to collect code metrics, the detailed explanation of each code metric is available on SciTools website. Since some software metrics are computed at the method level, following existing study [79], we use three aggregation schemes (i.e., min, max, and average) to aggregate these metrics to the file level. Table 3 describes the details of the studied code metrics.

### 4.3 Evaluation Metrics

We use AUC to evaluate the performance of a bug prediction model. It has been widely used to evaluate classification algorithms in prediction tasks [14, 50, 51, 55, 69, 71, 81]. We use accuracy to evaluate the performance of ConBuild and ConUpdate. The accuracy is defined as the ratio of the correctly recommended training versions (for ConBuild) or actions (for ConUpdate) among all the recommendation results. For evaluate ConEA, we employ PofB20 [24] to measure the percentage of bugs that a developer can identify by inspecting the top 20 percent lines of code. A higher PofB20 score indicates that a developer can detect more bugs when inspecting a limited number of LOC.

#### 4.4 Benchmark Approaches

To explore the performance of the proposed ConBuild, we compare it to the following baselines.

- **FSSBagging** [20]: This is the state-of-the-art training version selection approach for cross project bug prediction (CPBP), which adopts a different data similarity measure from ConBuild. Specifically, it first generates a synthetic dataset by combining data from a candidate training version and the given test version. It then trains a Logistic Regression based classifier on the synthetic dataset and uses its accuracy to measure the similarity between the candidate training version and the given test version. Based on the similarity, FSSBagging selects top\( N \) training versions from different projects and filters out unstable features and then uses the bagging ensemble method to combine defect prediction results generated by models built with different training sets.

- **RandomBuild**: It does not have any training version selection strategy, which randomly selects a release version as the training data and builds a bug prediction model.

To evaluate ConEA, we compare it to the existing effort-aware bug prediction evaluation approaches.

- **EA** [24, 72]: It first sorts files in the test dataset based on the probabilities of being predicted as buggy by a bug prediction model. With the list of sorted files, EA then accumulates the lines of code (LOC) and the number of buggy files identified until 20 percent of the LOC in the test data have been inspected and the percentage of buggy files that are identified is referred to as the PofB20 score. The calculation of PofB20 for ConEA is slightly different from existing effort-aware evaluation approaches, instead of sorting files based on the probabilities, ConEA sorts files based on the difference of buggy probability between versions.

### 4.5 Research Questions

We answer the following research questions to evaluate the performance of our proposed approaches.

**RQ1**: To what degree ConBuild can find the appropriate training version for model building in continuous software development?

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
<th>#Files</th>
<th>#KLOC</th>
<th>Bug rate</th>
<th>Studied Releases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant</td>
<td>Java project building framework</td>
<td>0.71K-0.86K</td>
<td>94-107</td>
<td>9.7%-16.4%</td>
<td>from v1.7.1 to v1.9.13</td>
</tr>
<tr>
<td>Poi</td>
<td>Java library for handling Microsoft Office docs</td>
<td>1.8K-2.3K</td>
<td>199-244</td>
<td>7.2%-13.1%</td>
<td>from v3.10.1 to v4.0.1</td>
</tr>
<tr>
<td>ActiveMQ</td>
<td>Java based open source message broker</td>
<td>19K-36K</td>
<td>142-299</td>
<td>6.2%-15.5%</td>
<td>from v5.12.3 to v5.15.8</td>
</tr>
<tr>
<td>HBase</td>
<td>Distributed scalable data store</td>
<td>3.9K-4.5K</td>
<td>497-643</td>
<td>4.1%-6.2%</td>
<td>from v5.5.2 to v7.6.0</td>
</tr>
<tr>
<td>Lucene</td>
<td>Text search engine library</td>
<td>4.6K-7.7K</td>
<td>133-362</td>
<td>3.3%-9.6%</td>
<td>from v2.17.7 to v2.23.0</td>
</tr>
<tr>
<td>Camel</td>
<td>Enterprise integration framework</td>
<td>1.5K-1.8K</td>
<td>364-525</td>
<td>6.2%-15.5%</td>
<td>from v1.1.10 to v2.0.3</td>
</tr>
<tr>
<td>Poi</td>
<td>Java library for handling Microsoft Office docs</td>
<td>9.7%-16.4%</td>
<td>6.2%-20.5%</td>
<td>1.5K-1.8K</td>
<td>from v5.12.3 to v5.15.8</td>
</tr>
<tr>
<td>Ant</td>
<td>Java project building framework</td>
<td>1.8K-2.3K</td>
<td>199-244</td>
<td>7.2%-13.1%</td>
<td>from v3.10.1 to v4.0.1</td>
</tr>
<tr>
<td>ActiveMQ</td>
<td>Java based open source message broker</td>
<td>19K-36K</td>
<td>142-299</td>
<td>6.2%-15.5%</td>
<td>from v5.12.3 to v5.15.8</td>
</tr>
<tr>
<td>HBase</td>
<td>Distributed scalable data store</td>
<td>3.9K-4.5K</td>
<td>497-643</td>
<td>4.1%-6.2%</td>
<td>from v5.5.2 to v7.6.0</td>
</tr>
<tr>
<td>Lucene</td>
<td>Text search engine library</td>
<td>4.6K-7.7K</td>
<td>133-362</td>
<td>3.3%-9.6%</td>
<td>from v2.17.7 to v2.23.0</td>
</tr>
</tbody>
</table>

\(^{2}\text{https://scitools.com}\)
Table 3: Details of the metrics used to build software bug prediction models in this work.

<table>
<thead>
<tr>
<th>Type</th>
<th>Metrics</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>CountClassBase, CountClassCoupled, CountClassDerived, MaxInheritanceTree, PercentLackOfCohesion</td>
<td>5</td>
</tr>
<tr>
<td>Method</td>
<td>CountInput {[Min, Mean, Max]}, Countoutput {[Min, Mean, Max]}, CountPath {[Min, Mean, Max]}, MaxNesting {[Min, Mean, Max]}</td>
<td>12</td>
</tr>
</tbody>
</table>

RQ2: How effective is ConUpdate in guiding model reuse and update in continuous software development?

RQ3: How much ConEA can improve existing effort-aware bug prediction approaches in continuous software development?

RQ1 explores the effectiveness of our approach for training version selection for building bug prediction models proposed in Algorithm 1. In RQ2, we aim to understand the performance of our model update or reuse recommendation approach proposed in Algorithm 2. In RQ3, we investigate the performance of our effort-aware bug evaluation approach for continuous software bug prediction models proposed in Algorithm 3.

4.6 Terminologies

This section presents the basic terminologies used to evaluate our proposed approach. The best model is the bug prediction model that achieves the best performance (i.e., AUC) on a given test version. Applicable models include the best models and other models with less than 5% deviation compared to the best models on a given test version, since 5% deviation is considered as acceptable in statistics [34]. Other models are inapplicable models.

5 RESULT ANALYSIS

5.1 RQ1: Performance of ConBuild

To evaluate ConBuild (Algorithm 1), we select the latest ten release versions (from v1 to v20) from each project (listed in Table 2) as the test datasets. For each test dataset, we then use the model trained with the available bug information when the test version is released and collect features for each version, then we use early release versions to build and tune ADTree based bug prediction models. After that, we obtain its best models, applicable models and inapplicable models as the ground truth. We use Algorithm 1 to recommend the training version for each test version and check how many test versions could obtain the applicable models with our recommendations. In addition, for the ones that our approach cannot recommend an applicable model, we further calculate the performance difference between the bug prediction models trained on our recommended versions and the best models.

Table 4 shows the results of ConBuild on the 60 experimental test versions (from v1 to v20) from the six projects. Overall, of 78.33% (47 out of 60) experiment runs, ConBuild can help find an applicable model. We can also observe that for the left 21.77% (13 out of 60) experiment runs, ConBuild cannot find their applicable models. Since an inappropriate training version could introduce a performance decline, thus for these unsuccessful recommendations we further check the performance differences between the models built on recommended training versions and the best models, which are also presented in Table 4. Overall, the performance differences between models built with the unsuccessful recommended training versions and the best models range from 5.5% to 14.5% and on average is 8.6%, which means ConBuild does not lose too much predictive power for the unsuccessful recommendations.

We further evaluate ConBuild by comparing it to two benchmarks, i.e., FSSBagging [20] and RandomBuild. Note that we use the recommended parameter values [20] and ADTree classifier to drive FSSBagging on each of the test versions from v1 to v20 on the six experiment projects. To compare the performance of ConBuild and RandomBuild, for each of the test versions from v1 to v20, we randomly select one of its previous versions to build a bug prediction model and further evaluate its performance on the test version. We repeat RandomBuild on each test version 100 times to remove potential bias. We also collect the performance of the best models across all the test versions in the six projects as the best baseline.

The distributions of performance of the best model (○), ConBuild (●), FSSBagging (■), and RandomBuild (▲) on the test versions are shown in Figure 4. Overall, ConBuild outperforms both FSSBagging and RandomBuild on the six experiment projects. FSSBagging delivers better results than RandomBuild on five out of the six experiment projects. Our statistical test (i.e., Wilcoxon signed-rank test) shows that ConBuild is significantly better than both FSSBagging and RandomBuild.

One possible reason that FSSBagging does not perform well in continuous bug prediction scenarios is that the assumption of FSSBagging, i.e., combining multiple bug prediction models built with different training versions from different projects could generate better results than the model built with insufficient training data from the same project, does not hold in continuous software development scenarios, since in continuous software development, a project often has sufficient history training data to build effective bug prediction models.
Table 4: Results of ConBuild on the ten test versions (from $v_{11}$ to $v_{20}$) on each project. ‘√’ denotes that the bug prediction model built on the recommended training version is an applicable model. ‘×’ denotes an unsuccessful recommendation. Numbers in the brackets are the relative performance difference (i.e., AUC difference) between bug prediction models built on the recommended training versions and the best model.

<table>
<thead>
<tr>
<th>Project</th>
<th>$v_{11}$</th>
<th>$v_{12}$</th>
<th>$v_{13}$</th>
<th>$v_{14}$</th>
<th>$v_{15}$</th>
<th>$v_{16}$</th>
<th>$v_{17}$</th>
<th>$v_{18}$</th>
<th>$v_{19}$</th>
<th>$v_{20}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant</td>
<td>√</td>
<td>× (-9.9%)</td>
<td>√</td>
<td>× (-6.3%)</td>
<td>√</td>
<td>× (-7.1%)</td>
<td>√</td>
<td>× (-14.5%)</td>
<td>× (-8.1%)</td>
<td>√</td>
</tr>
<tr>
<td>Poi</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>× (-10.1%)</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>ActiveMQ</td>
<td>√</td>
<td>√</td>
<td>× (-5.5%)</td>
<td>√</td>
<td>× (-13.5%)</td>
<td>√</td>
<td>× (-9.8%)</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>HBase</td>
<td>√</td>
<td>√</td>
<td>× (-5.8%)</td>
<td>√</td>
<td>× (-6.7%)</td>
<td>√</td>
<td>√</td>
<td>× (-6.0%)</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Lucene</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Camel</td>
<td>√</td>
<td>× (-5.8%)</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

ConBuild can help find the best training versions for 78.3% test versions. In addition, ConBuild significantly outperforms existing approaches, which shows its practical value.

5.2 RQ2: Performance of ConUpdate

To evaluate ConUpdate (Algorithm 2), we use the same experiment settings as RQ1 (Section 5.1), e.g., we select the latest ten release versions (from $v_{11}$ to $v_{20}$) as the test datasets and for each test dataset, we label its early release versions with the available bug information when the version is released, then we use early versions to build and tune ADTree based bug prediction models. After that, we obtain its best model, applicable models, and inapplicable models as the ground truth. Note that for the first test version, i.e., $v_{11}$, we use the best model of its previous version $v_{10}$ as the applied model to drive ConUpdate, for a later test version (i.e., from $v_{12}$ to $v_{20}$), if ConUpdate recommends an Update operation, we use the training version selected by ConBuild (details are in Section 3.1) as its applied model.

Given a test version $v_n$, we define a Reuse recommendation is valid, if the applied model of previous version $v_{n-1}$ (i.e., the bug prediction model that used to predict bugs on $v_{n-1}$) is in the applicable model set (i.e., ground truth) of $v_n$. Analogically, we define an Update recommendation is valid, if the applied model of previous version $v_{n-1}$ is not in the applicable model set of $v_n$. To find the best $k$ in Algorithm 2, we experiment $k$ from 1 to 5 and check the accuracy of our recommendation. Results show that when $k$ equals to 3, ConUpdate generates the best performance.

Table 5 shows the detailed recommendation results on the six projects when $k$ equals to 3. Specifically, ConUpdate makes 60 recommendations, 29 of them are Reuse and 31 are Update. We further calculate the performance of our recommendation of Reuse and Update regarding the accuracy of the recommendation results. For the Reuse, ConUpdate achieves an accuracy of 62.1% and for the Update, ConUpdate achieves an accuracy of 83.9%. Overall, ConUpdate achieves an accuracy of 73.3% on these two types of recommendations. Since an unsuccessful Reuse may introduce a performance decline, for these unsuccessful recommendations, we further check the performance differences between the recommended previous applied model and the ground truth best model, which are shown in Table 5. Overall, the performance declines range from 5.5% to 15.0% across the six projects and on average is 9.5%.

In addition, as the goal of ConUpdate is to help developers save unnecessary modeling building effort. Without ConUpdate, for each newly released version, developers would either randomly reuse previous models (which could bring dramatic performance decline as shown in Section 2.2.1) or use ConBuild to find an appropriate training version to build a bug prediction model. ConBuild requires 10 model-building runs on test versions from $v_{11}$ to $v_{20}$ in each of the six projects. To show how much effort our approach could save on each project, we calculate the model building runs that our approach could save (i.e., a successful Reuse recommendation could save one model building run), details are in Figure 5. As we can see from the figure, the effort our approach saved ranges from 10.0% to 60.0% across the six experimental projects and on average is 28.3%, which shows the practical value of ConUpdate.

ConUpdate can guide bug prediction model reuse or update for 73.3% test versions, and save 28.3% model-building effort.

5.3 RQ3: Performance of ConEA

To evaluate ConEA (Algorithm 3), we use the same experiment settings as RQ1 (Section 5.1) and RQ2 (Section 5.2), i.e., select the latest ten release versions (from $v_{11}$ to $v_{20}$) as the test versions from each project. For each of the test version, we use early versions to build and tune ADTree based bug prediction models, after that we obtain its best model, finally we use the best model to predict bugs on each test version. Based on the prediction results we then calculate the widely used effort-aware evaluation metric, i.e., PofB20 (details are in Section 4.3), by following the existing studies [24, 72] (denoted as EA) and our proposed ConEA.

Table 6 shows the comparison of the PofB20 values generated by EA [24, 72] and ConEA. Overall, on 85% of the experiment pairs, ConEA can generate better or equal PofB20 values to EA. In addition on the 65% of the experiment pairs, ConEA generates better results than that of EA. The improvements can be up to 58.83 percentage points (i.e., $v_{20}$ of project Ant), and on average the improvement is 9.5 percentage points. Results of our statistical test (i.e., Wilcoxon signed-rank test) suggests that the proposed ConEA generates significantly better PofB20 values than EA.
Comparison of PofB20 between our proposed continuous effort-aware evaluation (i.e., ConEA) and current effort-aware evaluation approach (i.e., EA) [24, 72]. Better or equal PofB20 values generated by ConEA are shown in bold.

<table>
<thead>
<tr>
<th>Project</th>
<th>v11</th>
<th>v12</th>
<th>v13</th>
<th>v14</th>
<th>v15</th>
<th>v16</th>
<th>v17</th>
<th>v18</th>
<th>v19</th>
<th>v20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ant</td>
<td>R √</td>
<td>R × (-8.1%)</td>
<td>R × (-14.3%)</td>
<td>R √</td>
<td>U √</td>
<td>R √</td>
<td>R √</td>
<td>R × (-8.3%)</td>
<td>R × (-6.5%)</td>
<td>U √</td>
</tr>
<tr>
<td>Poi</td>
<td>R × (-8.7%)</td>
<td>R × (-13.4%)</td>
<td>U √</td>
<td>R ×</td>
<td>U √</td>
<td>U √</td>
<td>U ×</td>
<td>R × (-5.2%)</td>
<td>R ×</td>
<td>U √</td>
</tr>
<tr>
<td>ActiveMQ</td>
<td>R × (-8.7%)</td>
<td>U √</td>
<td>R ×</td>
<td>U √</td>
<td>R ×</td>
<td>R ×</td>
<td>U √</td>
<td>R × (-5.4%)</td>
<td>U ×</td>
<td>R ×</td>
</tr>
<tr>
<td>HBase</td>
<td>U √</td>
<td>R ×</td>
<td>U √</td>
<td>U √</td>
<td>U ×</td>
<td>U ×</td>
<td>U √</td>
<td>U √</td>
<td>U ×</td>
<td>R ×</td>
</tr>
<tr>
<td>Lucene</td>
<td>U √</td>
<td>R × (-5.5%)</td>
<td>U √</td>
<td>U ×</td>
<td>R × (-6.8%)</td>
<td>U ×</td>
<td>U ×</td>
<td>U √</td>
<td>U √</td>
<td>R ×</td>
</tr>
<tr>
<td>Camel</td>
<td>U √</td>
<td>R × (-5.5%)</td>
<td>U √</td>
<td>R × (-8.0%)</td>
<td>U √</td>
<td>U √</td>
<td>U ×</td>
<td>U √</td>
<td>U √</td>
<td>R ×</td>
</tr>
</tbody>
</table>

Table 6: Comparison of PofB20 between our proposed continuous effort-aware evaluation (i.e., ConEA) and current effort-aware evaluation approach (i.e., EA) [24, 72]. Better or equal PofB20 values generated by ConEA are shown in bold.

![Comparison of PofB20 between ConBuild, ConUpdate, and ConEA](image)

Figure 6: The distribution of performance of ConBuild, ConUpdate, and ConEA with different bug prediction models.

Table 7: Performance of ConBuild, ConUpdate, and ConEA with different bug prediction models built on different machine learning classifiers.

<table>
<thead>
<tr>
<th>Project</th>
<th>ADTree</th>
<th>RF</th>
<th>NB</th>
<th>LR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConBuild</td>
<td>78.3%</td>
<td>66.7%</td>
<td>95.0%</td>
<td>88.3%</td>
</tr>
<tr>
<td>ConUpdate</td>
<td>73.5%</td>
<td>65.0%</td>
<td>68.3%</td>
<td>60.0%</td>
</tr>
<tr>
<td>ConEA</td>
<td>9.5%</td>
<td>14.4%</td>
<td>9.2%</td>
<td>14.7%</td>
</tr>
</tbody>
</table>

As we can see from Table 7, with all the four examined bug prediction models, ConBuild can help at least 40 out of 60 experiment runs (i.e., accuracy is 66.7%) find the applicable models. Statistic test (i.e., Wilcoxon signed-rank test) also shows that all of them are significantly better than FSSBagging. With different machine learning based bug prediction models, ConUpdate can achieve accuracy larger than 60.0% regarding model reuse or update recommendations. Compared to current effort-aware evaluation approach (i.e., EA), the average improvement of ConEA variants can be up to 14.7% (i.e., ConEA with bug prediction models built on LR).

Overall, the fact that the performance of ConBuild, ConUpdate, and ConEA does not stick to a specific machine learning classifier suggests that our approaches are generalizable for software bug prediction models built on different machine learning algorithms.

### 6.2 Compare to Cross-Project Bug Prediction

The most similar work to this study is cross-project bug prediction (CPBP) [2, 3, 5, 7, 27, 57, 82, 84]. The difference between CPBP and continuous bug prediction can be summarized as follows: CPBP is designed for projects with insufficient training data (e.g., new projects with no prior data), continuous bug prediction is designed for solving the problems encountered when building prediction models in continuous software development scenarios, i.e., the projects involved often have sufficient history training data. Nevertheless,
similar to our continuous bug prediction, one common solution for CPBP is choosing a proper training dataset. The state-of-the-art training version selection approach in CPBP, i.e., FSSBagging [20], utilized bagging ensemble method to build multiple bug prediction models with training versions selected from different projects and then combine their results for achieving better performance. Our comparison between FSSBagging [20] and ConBu1d shows that ConBu1d delivers significant better results than FSSBagging on training version selection in continuous software development scenarios.

Note that, there also exist instance selection based approaches for training data selection in CPBP [22, 59, 68], which create a new training dataset by selecting data instances from different projects. We do not compare ConBu1d to instance selection based approaches, since these approaches often require non-trivial time cost, e.g., [22] required 28.3 hours to finish its experiments.

6.3 Threats to Validity

Internal Validity: We only used code metrics as available from SciTool’s popular Understand tool, which does not generate all possible code metrics ever reported or used in literature. However, it does produce a large set of diverse code metrics.

External Validity: In this work, all the experiment subjects are open-source projects from Apache Software Foundation (ASF) and written in Java. Although they are popular projects and widely used in existing software bug prediction studies, our findings may not be generalizable to commercial projects or projects in other ecosystems.

In addition, the conclusions of our case study rely on one defect prediction scenario (i.e., within-project bug prediction models). However, there are a variety of bug prediction scenarios in the literature (e.g., change-level bug prediction [24, 28, 65] and method-level bug prediction [15]). Therefore, the conclusions may differ for other scenarios.

7 RELATED WORK

Software bug prediction techniques leverage various metrics to build machine learning models to predict unknown defects in software projects [7, 19, 33, 56, 86].

Most bug prediction techniques leverage features that are manually extracted from labeled historical defect data to train machine learning based classifiers [44]. Software prediction features can be divided into static code features (e.g., Halstead features [17], McCabe features [40], MOOD features [18], etc.), process features [24, 45, 48, 56, 65], semantic features [62, 73], context metrics [31], etc. Most of the above studies were evaluated on the widely used and publicly available datasets, e.g., PROMISE dataset [26], NASA dataset [60], AEEEM dataset [8], SOFTLAB dataset [68], or ReLink dataset [50, 75], etc. While these datasets contain randomly selected discrete versions as shown in Section 2.2.1, evaluating bug prediction models on these datasets could introduce non-trivial bias in performance. Some other studies also consider leveraging information between different versions to predict bugs, e.g., Kastro et al. [29] used the change information between versions to predict the number of bugs that would be appeared in a new version. Krishna et al. [32] used a time series analysis of the last 4 months of issues to forecast how many bug reports and enhancement requests will be generated next month. Liu et al. [37] used the difference of code metrics between two versions to build bug prediction models, which were evaluated on projects from PROMISE dataset [26]. Different from the above studies, this work explores the challenging issues of file-level software bug prediction in the context of continuous software development regarding model building, updating, and evaluation.

Most of the above studies were evaluated without considering the effort to check the bug prediction results. Mende et al. [42] first introduced the notion of effort into a bug prediction model. Along this line, many effort-aware bug prediction models have been proposed and well studied [4, 5, 23, 24, 61, 77]. In most of the above studies, the number of lines of code in a file was used as the de facto measure of effort required to inspect the file. Files are ranked based on their buggy probability to be inspected for finding potential bugs. Other studies related to effort estimation focus on estimating the time required to fix software bugs [1, 16]. Different from these studies, in this work we leverage the evolution of the buggy probability of files between two sequential versions to redefine effort-aware evaluation in continuous software development.

Software bug prediction models can be categorized into file-level [19, 86], change-level [6, 24, 41, 65, 87], or method-level [15] based on the prediction granularities. As the first study on exploring software bug prediction models in the context of continuous software development, we only focus on file-level bug prediction. We plan to extend our approach to other bug prediction models in our future work.

8 CONCLUSION

In this paper, we revisit software bug prediction in the real-world continuous software development scenarios regarding bug prediction model building, updating, and evaluation. Specifically, for model building, we propose ConBu1d, which leverages the distributional characteristics of bug prediction data to redefine the process of training data selection. For model updating, we propose ConUpdate, which leverages the evolution of distributional characteristics of bug prediction data to guide the reuse or update of bug prediction models. For model evaluation, we propose ConEA, which leverages the evolution of the buggy probability of files to redefine effort-aware evaluation in continuous software development. Experiments on 120 continuous releases that span across six large-scale open-source software systems show the practical value of our approach. Although we focus on software bug prediction models in this study, we believe this study opens the door to rethink existing software analysis models in the context of continuous software development, e.g., software effort estimation, reviewer recommendation, and bug triage models, in the context of continuous software development.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their feedback which helped improve this paper. This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the National Natural Science Foundation of China under grant No.62072442.
[84] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. [n.d.]. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In FSE’09. 91–100.