Continuous Software Bug Prediction

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

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

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

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**.

ESEM 2021, 11st - 15th October 2021, Bari

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

KEYWORDS

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

ACM Reference Format:
Song Wang, Junjie Wang, Jaechang Nam, and Nachiappan Nagappan. 2021. Continuous Software Bug Prediction. In *Proceedings of ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2021).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.475/123_4

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 [35], 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 `0.0` on an early release version v_1. Then it was predicted as buggy with a probability of `0.51` on a later release version v_2. 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 v_1 to v_2). Existing effort-aware evaluation approaches give higher priority to `f2` since `f2` has a higher buggy probability in v_2, 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[1].

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.

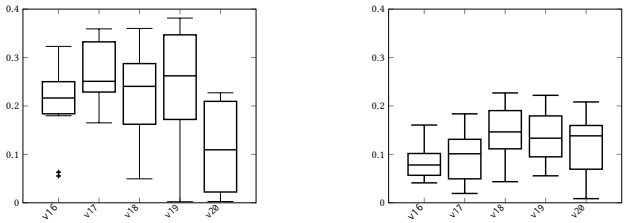
2.2 Problems of Existing Bug Prediction Process

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 incontinuous. 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.



(a) Camel **(b) Lucene**

Figure 1: The distributions of performance difference of AUC values between the best model and other models.

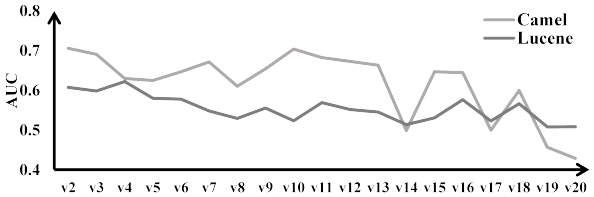


Figure 2: Performance evolution of bug prediction models built on the first versions on Camel and Lucene.

> The problems of current widely used bug prediction benchmark datasets, i.e., the prevalence of randomness regarding data collection in these benchmark datasets, could introduce non-trivial performance bias and do not fit in with real-world continuous software development.

2.2.2 ***Pilot Study II: Performance Evolution of Bug Prediction Models.*** As reported by existing studies, software analysis models for predicting buggy commits [6, 41] or monthly bug frequency [10] lost the predictive power over time due to the data drift issue during the evolution of software projects, i.e., the distribution of software artificial data (e.g., bug types, code changes, functionalities, etc.) can be changed over time. In this pilot study, we set out to investigate whether a release-based file-level bug prediction model also suffers from the same issue. Specifically, we trained bug prediction models on the first release versions (i.e., v_1) from Camel and Lucene, and then we use a later release version (from v_2 to v_{20}) as the test data to examine the performance of the models.

Figure 2 shows the performance of the two bug prediction models on test data from different release versions over time. Overall, the performance of bug prediction models declines on both projects with the evolution of projects, i.e., the AUC values drop from 0.775 (on v_2) to 0.428 (on v_{20}) on Camel and from 0.671 (on v_4) to 0.508 (on v_{20}) on Lucene. This confirms that release-based file-level bug prediction models also suffer from the data drift issue, which makes the prediction performance decline over time. In addition, we can also find that a bug prediction model does not lose the predictive power sharply. For example, the performance of the model has a stable AUC values (0.63 ± 0.014) on test versions from v_4 to v_7 on Camel, the same phenomenon is observed on Lucene, e.g., test versions v_{11}-v_{13}. This sheds light on the reuse of bug prediction models built on early versions for saving model-building effort.

One of the possible reasons for the performance evolution is the different distribution of features among release versions. Test

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 (v_1) and a test version (from v_2 to v_{20}). 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 (v_1) 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.

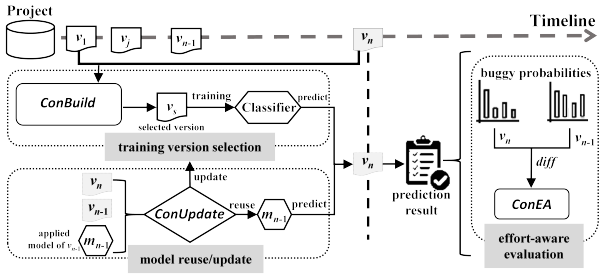


Figure 3: The overview of our proposed approaches for building, updating, and evaluating bug prediction models in continuous software development.

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 ConUpate: 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 $V_{trainings}$;
Ensure:
recommended version v_r;
1: initialize a Map S;
2: **for** each version v in $V_{trainings}$ **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.

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

prediction model that used to predict bugs on v_{n-1}) have similar distributional characteristics to v_n, ConUpdate recommends to reuse the `applied model` of v_{n-1} to conduct bug prediction tasks on v_n, otherwise, ConUpdate recommends to update bug prediction models and developers can find an appropriate training version to rebuild a bug prediction model with ConBuild (in Section 3.1).

ConUpdate first calculates the similarities between each previous release version (including version v_{n-1} and the training version of its `applied model`) and v_n. Second, ConUpdate ranks all previous versions based on the similarity values. If both v_{n-1} and the training version of its `applied model` are ranked in top k ($1 < k < n$), ConUpdate recommends a Reuse operation. Otherwise, ConUpdate recommends an Update operation. For the Update operation, developers could use ConBuild algorithm in Section 3.1 to find an appropriate training version to rebuild a bug prediction model for version v_n. Algorithm 2 describes the details of ConUpdate.

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 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 $Applied_{n-1}$;
all existing version list V_{all};
threshold k;
Ensure:
recommended operation;
1: initialize a Map S;
2: **for** each version v in V_{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 $Applied_{n-1}$ 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 on 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 on the fault proneness (i.e., buggy probability) between 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.

4 EXPERIMENT SETUP

4.1 Data Collection

In this work, we set out to investigate software bug prediction models in real-world continuous software development, while as shown in Section 2.2, most of the existing benchmark datasets do not satisfy our criterion, i.e., a dataset of continuously released versions. Thus, we follow the process described in Section 2.1 to collect experiment data. Specifically, we first collect a set of bug-fixing commits within a given period by using the heuristic approach proposed in [13], i.e., using regular expression to search for specific keywords (e.g.,

Table 2: Details of the experimental subjects and the 20 recent continuous release versions from each project.

Project	Description	#Files	#KLOC	Bug rate	Studied Releases
Ant	Java project building framework	0.71K-0.86K	94-107	9.7%-16.4%	from v1.7.1 to v1.9.13
Poi	Java library for handling Microsoft Office docs	1.8K-2.3K	199-244	7.2%-13.1%	from v3.10.1 to v4.0.1
ActiveMQ	Java based open source message broker	19K-36K	142-299	6.2%-15.5%	from v5.12.3 to v5.15.8
HBase	Distributed scalable data store	1.5K-1.8K	364-525	6.1%-20.5%	from v1.1.10 to v2.0.3
Lucene	Text search engine library	3.9K-4.5K	497-643	4.1%-6.2%	from v5.5.4 to v7.6.0
Camel	Enterprise integration framework	4.6K-7.7K	133-362	3.3%-9.6%	from v2.17.7 to v2.23.0

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
for each file f in R_n **do**
  if f in R_{n-1} and f is true buggy in R_{n-1} **then**
    calculate the probability diff $diff_f$ in R_n and R_{n-1};
    put f and $diff_f$ into F;
  end if
  if f not in R_{n-1} and f is predicted as buggy in R_n **then**
    put f and its probability from R_n into F;
  end if
end for
sort map F by $diff$;
return F;

$fix(e[ds])?, bugs?, defects?$) 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 YatishData [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.

[2]https://scitools.com

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 3: Details of the metrics used to build software bug prediction models in this work.

Type	Metrics	Count
File	AvgCyclomatic, AvgCyclomaticModified, AvgCyclomaticStrict, AvgEssential, AvgLine, AvgLineBlank, AvgLineCode, AvgLineComment, CountDeclClass, CountDeclClassMethod, CountDeclClassVariable, CountDeclFunction, CountDeclInstanceMethod, CountDeclInstanceVariable, CountDeclMethod, CountDeclMethodDefault, CountDeclMethodPrivate, CountDeclMethodProtected, CountDeclMethodPublic, CountLine, CountLineBlank, CountLineCode, CountLineCodeDecl, CountLineCodeExe, CountLineComment, CountSemicolon, CountStmt, CountStmtDecl, CountStmtExe, MaxCyclomatic, MaxCyclomaticModified, MaxCyclomaticStrict, RatioCommentToCode, SumCyclomatic, SumCyclomaticModified, SumCyclomaticStrict, SumEssential	37
Class	CountClassBase, CountClassCoupled, CountClassDerived, MaxInheritanceTree, PercentLackOfCohesion	5
Method	CountInput_{Min, Mean, Max}, CountOutput_{Min, Mean, Max}, CountPath_{Min, Mean, Max}, MaxNesting_{Min, Mean, Max}	12

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.

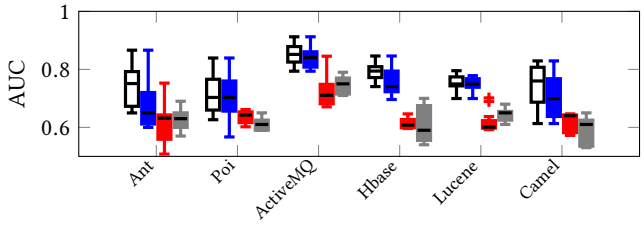


Figure 4: The distributions of performance of the `best model` (○), `ConBuild` (●), `FSSBagging` (●), and `RandomBuild` (●).

4.6 Terminologies

This section presents the basic terminologies used to evaluate our proposed approach.

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 v_{11} to v_{20}) from each project (listed in Table 2) as the test datasets. For each test dataset, we label its early release versions 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 v_{11} to v_{20}) from the six projects. Overall, of 78.3% (47 out of 60) experiment runs, `ConBuild` can help find an `applicable model`. We can also observe that for the left 21.7% (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 v_{11} to v_{20} on the six experiment projects. To compare the performance of `ConBuild` and `RandomBuild`, for each of the test versions from v_{11} to v_{20}, 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`.

Project	v_{11}	v_{12}	v_{13}	v_{14}	v_{15}	v_{16}	v_{17}	v_{18}	v_{19}	v_{20}
Ant	√	× (-9.9%)	√	× (-6.3%)	√	× (-7.9%)	× (-14.5%)	× (-8.1%)	√	√
Poi	√	√	√	√	× (-7.1%)	√	√	√	√	√
ActiveMQ	√	√	√	√	× (-10.1%)	√	√	√	√	√
HBase	√	√	× (-5.5%)	√	√	√	× (-13.5%)	√	√	× (-9.8%)
Lucene	√	√	√	√	√	× (-6.7%)	√	√	√	√
Camel	√	× (-5.8%)	√	√	√	√	√	× (-6.0%)	√	√

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.

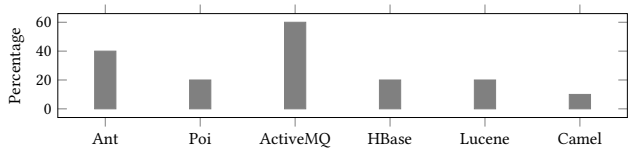


Figure 5: Model-building effort saved (i.e., the percentage of saved model building runs) by ConUpdate.

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.

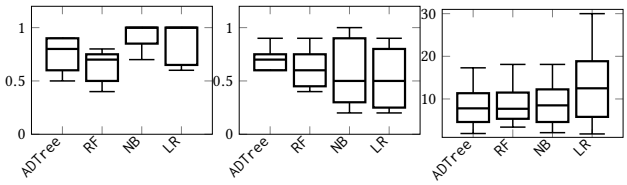
Table 6 shows the comparison of the **PofB20** values generated by EA [24, 72] and ConEA. Overall, on 85% of the experiment pairs, ConEA can generate better or equal **PofB20** values to EA. In addition on the 65% of the experiment pairs, ConEA generates better results than that of EA. The improvements 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.

Table 5: Results of `ConUpdate` on deciding whether to reuse previous `applied model` (i.e., R) or to update bug prediction models, i.e., finding an appropriate training version and building a new bug prediction model (i.e., U). Numbers in the brackets are the performance difference between the recommended previous `applied model` and the ground truth `best model` of the versions to be predicted. '√' denotes a successful recommendation and '×' represents an unsuccessful recommendation.

Project	v_{11}	v_{12}	v_{13}	v_{14}	v_{15}	v_{16}	v_{17}	v_{18}	v_{19}	v_{20}
Ant	R √	R × (-8.1%)	R × (-14.3%)	R √	U √	R √	R √	R × (-8.3%)	R × (-6.5%)	U √
Poi	R × (-8.7%)	R × (-13.4%)	U √	R √	U ×	U √	R × (-15.0%)	U √	R √	U √
ActiveMQ	R √	U √	R √	U √	R √	R × (-5.2%)	R√	U √	R √	R √
HBase	U √	R √	R √	U √	U √	U √	U ×	U √	R × (-5.4%)	U ×
Lucene	U √	R √	U √	U √	R × (-6.8%)	U ×	U ×	U √	U √	R √
Camel	U √	R × (-5.5%)	U √	R × (-8.0%)	U √	U √	U √	U ×	U √	R √

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.

Project		Ant	Poi	ActiveMQ	HBase	Lucene	Camel
v_{11}	EA	15.00	19.50	38.89	3.45	19.64	30.69
	ConEA	**15.00**	**19.50**	**61.11**	**3.45**	**31.55**	**45.58**
v_{12}	EA	24.24	20.68	58.62	33.33	28.57	26.34
	ConEA	**24.24**	**31.28**	**72.41**	**33.33**	27.33	**38.68**
v_{13}	EA	62.50	28.45	71.42	47.27	28.47	20.87
	ConEA	**68.75**	**38.12**	**71.42**	**49.09**	**58.33**	**55.39**
v_{14}	EA	21.81	28.37	69.23	74.19	23.17	14.87
	ConEA	10.90	**28.71**	**69.23**	**74.19**	**33.55**	**14.87**
v_{15}	EA	6.25	14.41	40.00	37.63	32.65	22.44
	ConEA	**12.50**	**16.59**	11.11	**40.86**	**65.30**	20.95
v_{16}	EA	40.00	22.25	48.78	30.50	21.62	23.12
	ConEA	**60.00**	**45.07**	**75.61**	**37.29**	**61.26**	**27.81**
v_{17}	EA	22.22	24.34	44.44	55.93	18.09	12.59
	ConEA	**30.56**	**28.98**	**66.66**	**55.93**	**36.67**	**33.01**
v_{18}	EA	24.52	20.05	60.71	53.03	28.17	20.45
	ConEA	**24.52**	**25.13**	**92.86**	**53.03**	26.76	**20.45**
v_{19}	EA	31.57	22.64	51.61	18.70	22.22	16.56
	ConEA	15.78	18.24	**70.96**	**23.58**	**25.00**	**17.18**
v_{20}	EA	35.29	27.11	55.55	43.47	23.72	28.65
	ConEA	**94.12**	**55.42**	**96.29**	**46.09**	**53.85**	22.16
Improvement		7.30	7.79	14.8	1.93	17.33	7.94



(a) `ConBuild` **(b) `ConUpdate`** **(c) `ConEA`**

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

`ConEA` significantly improves the existing effort-aware evaluation approaches and the improvements can be up to 58.8% and on average are 9.5%.

6 DISCUSSION

6.1 Generalizability of Our Approaches

In this work, following existing studies [24, 49, 65, 73, 76, 79], we build bug prediction models with ADTree. However software bug prediction models with Random Forest (RF), Naive Bayes (NB), and Logistic Regression (LR) have also been adopted in existing work [11, 25, 79]. To explore the generalizability of our techniques, we further examine the performance of the proposed `ConBuild`, `ConUpdate`, and `ConEA` with bug prediction models based on three other classification techniques, i.e., RF, NB, and LR. We use the accuracy to measure the performance of `ConBuild` and `ConUpdate`. We use the average improvement on **PofB20** to measure the performance of `ConEA` with different machine learning classifiers. The results are shown in Table 7. We also show the detailed distribution of the improvements of `ConBuild`, `ConUpdate`, and `ConEA` with different bug prediction models in Figure 6.

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

	ADTree	RF	NB	LR
`ConBuild`	78.3%	66.7%	95.0%	88.3%
`ConUpdate`	73.3%	65.0%	68.3%	60.0%
`ConEA`	9.5%	14.4%	9.2%	14.7%

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, 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 ConBuild shows that ConBuild 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 ConBuild 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 ConBuild, 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 probabilities 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.

REFERENCES

[1] Syed Nadeem Ahsan, Javed Ferzund, and Franz Wotawa. 2009. Program File Bug Fix Effort Estimation Using Machine Learning Methods for OSS.. In *SEKE'09*. 129–134.

[2] Sousuke Amasaki. 2017. On Applicability of Cross-project Defect Prediction Method for Multi-Versions Projects. In *PROMISE'17*. 93–96.

[3] Sousuke Amasaki. 2018. Cross-Version Defect Prediction using Cross-Project Defect Prediction Approaches: Does it work?. In *PROMISE'18*. 32–41.

[4] Kwabena Ebo Bennin, Jacky Keung, Akito Monden, Yasutaka Kamei, and Naoyasu Ubayashi. 2016. Investigating the effects of balanced training and testing datasets on effort-aware fault prediction models. In *COMPSAC'16*, Vol. 1. 154–163.

[5] Kwabena Ebo Bennin, Koji Toda, Yasutaka Kamei, Jacky Keung, Akito Monden, and Naoyasu Ubayashi. 2016. Empirical evaluation of cross-release effort-aware defect prediction models. In *QRS'16*. 214–221.

[6] George G Cabral, Leandro L Minku, Emad Shihab, and Suhaib Mujahid. 2019. Class imbalance evolution and verification latency in just-in-time software defect prediction. In *ICSE'19*. 666–676.

[7] Maria Caulo and Giuseppe Scanniello. 2019. On the Use of Commit Messages to Support the Creation of Datasets for Fault Prediction: an Empirical Assessment. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 193–198.

[8] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2010. An extensive comparison of bug prediction approaches. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. 31–41.

[9] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2012. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *EMSE'12* 17, 4-5 (2012), 531–577.

[10] Jayalath Ekanayake, Jonas Tappolet, Harald C Gall, and Abraham Bernstein. 2009. Tracking concept drift of software projects using defect prediction quality. In *MSR'09*. 51–60.

[11] Karim O. Elish and Mahmoud O. Elish. 2008. Predicting defect-prone software modules using support vector machines. *JSS'08* 81, 5 (2008), 649–660.

[12] Justin R Erenkrantz. 2003. Release management within open source projects. In *Proc. 3rd. Workshop on Open Source Software Engineering*.

[13] Michael Fischer, Martin Pinzger, and Harald Gall. 2003. Populating a release history database from version control and bug tracking systems. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* IEEE, 23–32.

[14] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. 2001. *The elements of statistical learning*. Vol. 1.

[15] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C Gall. 2012. Method-level bug prediction. In *ESEM'12*. 171–180.

[16] Emanuel Giger, Martin Pinzger, and Harald Gall. 2010. Predicting the fix time of bugs. In *RSSE'10*. 52–56.

[17] Maurice H Halstead. 1977. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc.

[18] Rachel Harrison, Steve J Counsell, and Reuben V Nithi. 1998. An evaluation of the MOOD set of object-oriented software metrics. *TSE'98* 24, 6 (1998), 491–496.

[19] Ahmed E. Hassan. 2009. Predicting Faults Using the Complexity of Code Changes. In *ICSE'09*. 78–88.

[20] Zhimin He, F. Peters, T. Menzies, and Ye Yang. 2013. Learning from Open-Source Projects: An Empirical Study on Defect Prediction. In *ESEM'13*. 45–54.

[21] Zhimin He, Fengdi Shu, Ye Yang, Mingshu Li, and Qing Wang. 2012. An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering* 19, 2 (2012), 167–199.

[22] Seyedrebvar Hosseini, Burak Turhan, and Mika Mäntylä. 2018. A benchmark study on the effectiveness of search-based data selection and feature selection for cross project defect prediction. *IST'18* 95 (2018), 296–312.

[23] Qiao Huang, Xin Xia, and David Lo. 2017. Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In *ICSME'17*. 159–170.

[24] Tian Jiang, Lin Tan, and Sunghun Kim. 2013. Personalized defect prediction. In *ASE'13*. 279–289.

[25] Xiao-Yuan Jing, Shi Ying, Zhi-Wu Zhang, Shan-Shan Wu, and Jin Liu. 2014. Dictionary learning based software defect prediction. In *ICSE'14*. 414–423.

[26] Marian Jureczko and Lech Madeyski. 2010. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*. 9.

[27] Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E Hassan. 2016. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering* 21, 5 (2016), 2072–2106.

[28] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2012. A large-scale empirical study of just-in-time quality assurance. *TSE'12* 39, 6 (2012), 757–773.

[29] Yomi Kastro and Ayşe Basar Bener. 2008. A defect prediction method for software versioning. *Software Quality Journal* 16, 4 (2008), 543–562.

[30] Mijung Kim, Jaechang Nam, Jaehyuk Yeon, Soonhwang Choi, and Sunghun Kim. 2015. REMI: defect prediction for efficient API testing. In *FSE'15*. 990–993.

[31] Masanari Kondo, Daniel M German, Osamu Mizuno, and Eun-Hye Choi. 2019. The impact of context metrics on just-in-time defect prediction. *EMSE'19* (2019), 1–50.

[32] Rahul Krishna, Amritanshu Agrawal, Akond Rahman, Alexander Sobran, and Tim Menzies. 2018. What is the connection between issues, bugs, and enhancements?: Lessons learned from 800+ software projects. In *ICSE-SEIP'18*. 306–315.

[33] Taek Lee, Jaechang Nam, DongGyun Han, Sunghun Kim, and Hoh Peter In. 2011. Micro interaction metrics for defect prediction. In *FSE'11*. 311–321.

[34] Erich L Lehmann and Joseph P Romano. 2006. *Testing statistical hypotheses*. Springer Science & Business Media.

[35] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E James Whitehead. 2013. Does bug prediction support human developers? findings from a google case study. In *ICSE'13*. 372–381.

[36] Myron Lipow. 1982. Number of faults per line of code. *IEEE Transactions on software Engineering* 4 (1982), 437–439.

[37] Yibin Liu, Yanhui Li, Jianbo Guo, Yuming Zhou, and Baowen Xu. 2018. Connecting software metrics across versions to predict defects. In *SANER'18*. 232–243.

[38] Kanti V Mardia. 1970. Measures of multivariate skewness and kurtosis with applications. *Biometrika* 57, 3 (1970), 519–530.

[39] Matias Martinez and Martin Monperrus. 2015. Mining software repair models for reasoning on the search space of automated program fixing. *EMSE'15* 20, 1 (2015), 176–205.

[40] Thomas J McCabe. 1976. A complexity measure. *TSE'76* 4 (1976), 308–320.

[41] Shane McIntosh and Yasutaka Kamei. 2017. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *TSE'17* 44, 5 (2017), 412–428.

[42] Thilo Mende and Rainer Koschke. 2010. Effort-aware defect prediction models. In *CSMR'10*. 107–116.

[43] Andrew Meneely, Pete Rotella, and Laurie Williams. [n.d.]. Does adding manpower also affect quality? an empirical, longitudinal analysis. In *FSE'11*. 81–90.

[44] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. 2010. Defect prediction from static code features: current results, limitations, new approaches. *ASE'10* 17, 4 (2010), 375–407.

[45] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. 2008. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *ICSE'08*. 181–190.

[46] Jerome L Myers, Arnold D Well, and Robert F Lorch Jr. 2013. *Research design and statistical analysis*.

[47] Nachiappan Nagappan and Thomas Ball. 2005. Use of relative code churn measures to predict system defect density. In *ICSE'05*. 284–292.

[48] Nachiappan Nagappan and Thomas Ball. 2007. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *ESEM'07*. 364–373.

[49] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining metrics to predict component failures. In *ICSE'06*. 452–461.

[50] Jaechang Nam, Wei Fu, Sunghun Kim, Tim Menzies, and Lin Tan. 2017. Heterogeneous defect prediction. *TSE'17* (2017).

[51] Jaechang Nam and Sunghun Kim. 2015. CLAMI: Defect Prediction on Unlabeled Datasets. In *ASE'15*. 452–463.

[52] Mangasarian Olvi and Glenn Fung. 2000. *Data selection for support vector machine classifiers*. Technical Report.

[53] Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. 2005. Predicting the location and number of faults in large software systems. *TSE'05* 31, 4 (2005), 340–355.

[54] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. 2008. Can Developer-module Networks Predict Failures?. In *FSE'08*. 2–12.

[55] Jens C Pruessner, Clemens Kirschbaum, Gunther Meinlschmid, and Dirk H Hellhammer. 2003. Two formulas for computation of the area under the curve represent measures of total hormone concentration versus time-dependent change. *Psychoneuroendocrinology* 28, 7 (2003), 916–931.

[56] Foyzur Rahman and Premkumar Devanbu. 2013. How, and why, process metrics are better. In *ICSE'13*. 432–441.

[57] Foyzur Rahman, Daryl Posnett, Israel Herraiz, and Premkumar Devanbu. [n.d.]. Sample size vs. bias in defect prediction. In *FSE'13*. 147–157.

[58] Pete Rotella and Sunita Chulani. [n.d.]. Implementing quality metrics and goals at the corporate level. In *MSR'11*. 113–122.

[59] Duksan Ryu, Jong-In Jang, and Jongmoon Baik. 2015. A hybrid instance selection using nearest-neighbor for cross-project defect prediction. *JCST'15* 30, 5 (2015), 969–980.

[60] Martin Shepperd, Qinbao Song, Zhongbin Sun, and Carolyn Mair. 2013. Data quality: Some comments on the nasa software defect datasets. *IEEE Transactions on Software Engineering* 39, 9 (2013), 1208–1215.

[61] Emad Shihab, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2013. Is lines of code a good measure of effort in effort-aware models? *IST'13* 55, 11 (2013), 1981–1993.

[62] Thomas Shippey, David Bowes, and Tracy Hall. 2019. Automatically identifying code features for software defect prediction: Using AST N-grams. *IST'19* 106 (2019), 142–160.

[63] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes?. In *FSE'05*, Vol. 30. 1–5.

[64] Qinbao Song, Martin Shepperd, Michelle Cartwright, and Carolyn Mair. 2006. Software defect association mining and defect correction effort prediction. *TSE'06* 32, 2 (2006), 69–82.

[65] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. 2015. Online Defect Prediction for Imbalanced Data. In *ICSE'15*. 99–108.

[66] Chakkrit Tantithamthavorn and Ahmed E Hassan. [n.d.]. An experience report on defect modelling in practice: Pitfalls and challenges. In *ICSE-SEIP'18*. 286–295.

[67] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. Automated parameter optimization of classification techniques for defect prediction models. In *ICSE'16*. 321–332.

[68] Burak Turhan, Tim Menzies, Ayşe B Bener, and Justin Di Stefano. 2009. On the relative value of cross-company and within-company data for defect prediction. *EMSE'09* 14, 5 (2009), 540–578.

[69] Junjie Wang, Song Wang, Jianfeng Chen, Tim Menzies, Qiang Cui, Miao Xie, and Qing Wang. 2019. Characterizing crowds to better optimize worker recommendation in crowdsourced testing. *IEEE Transactions on Software Engineering* (2019).

[70] Junjie Wang, Song Wang, Qiang Cui, and Qing Wang. 2016. Local-based active classification of test report to assist crowdsourced testing. In *ASE'16*. 190–201.

[71] Song Wang, Chetan Bansal, Nachiappan Nagappan, and Adithya Abraham Philip. 2019. Leveraging change intents for characterizing and identifying large-review-effort changes. In *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*. 46–55.

[72] Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. 2018. Deep semantic feature learning for software defect prediction. *IEEE Transactions on Software Engineering* 46, 12 (2018), 1267–1293.

[73] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *ICSE'16*. 297–308.

[74] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. 2016. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.

[75] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. 2011. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 15–25.

[76] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep learning for just-in-time defect prediction. In *QRS'15*. 17–26.

[77] Yibiao Yang, Mark Harman, Jens Krinke, Syed Islam, David Binkley, Yuming Zhou, and Baowen Xu. 2016. An empirical study on dependence clusters for effort-aware fault-proneness prediction. In *ASE'16*. 296–307.

[78] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. 2016. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In *FSE'16*. 157–168.

[79] Suraj Yatish, Jirayus Jiarpakdee, Patanamon Thongtanunam, and Chakkrit Tantithamthavorn. 2019. Mining software defects: should we consider affected releases?. In *ICSE'19*. 654–665.

[80] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. 2014. Towards building a universal defect prediction model. In *MSR'04*. 182–191.

[81] Feng Zhang, Quan Zheng, Ying Zou, and Ahmed E Hassan. 2016. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *ICSE'16*. 309–320.

[82] Jie Zhang, Jiajing Wu, Chuan Chen, Zibin Zheng, and Michael R Lyu. 2020. CDS: A Cross–Version Software Defect Prediction Model With Data Selection. *IEEE Access* 8 (2020), 110059–110072.

[83] Thomas Zimmermann and Nachiappan Nagappan. 2008. Predicting defects using network analysis on dependency graphs. In *ICSE'08*. 531–540.

[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.

[85] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. 2010. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *ICST'10*. 421–428.

[86] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting defects for eclipse. In *PROMISE'07*. 9–9.

[87] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. 2019. An empirical study of fault localization families and their combinations. *TSE'19* (2019).