Enhancing Performance Bug Prediction Using Performance Code Metrics

Guoliang Zhao
IBM
Markham, Canada
Guoliang.Z@ca.ibm.com

Stefanos Georgiou
simpleTechs
Budapest, Hungary
stefanos@simpletechs.net

Ying Zou
Department of Electrical and Computer Engineering, Queen's University
Kingston, Canada
ying.zou@queensu.ca

Safwat Hassan
Faculty of Information, University of Toronto
Toronto, Canada
Safwat.hassan@utoronto.ca

Derek Truong
IBM
Markham, Canada
trong@ca.ibm.com

Toby Corbin
IBM
Southampton, United Kingdom
corbint@uk.ibm.com

ABSTRACT

Performance bugs are non-functional defects that can significantly reduce the performance of an application (e.g., software hanging or freezing) and lead to poor user experience. Prior studies found that each type of performance bugs follows a unique code-based *performance anti-pattern* and proposed different approaches to detect such anti-patterns by analyzing the source code of a program. However, each approach can only recognize one performance anti-pattern. Different approaches need to be applied separately to identify different performance anti-patterns. To predict a large variety of performance bug types using a unified approach, we propose an approach that predicts performance bugs by leveraging various historical data (e.g., source code and code change history). We collect performance bugs from 80 popular Java projects. Next, we propose performance code metrics to capture the code characteristics of performance bugs. We build performance bug predictors using machine learning models, such as Random Forest, eXtreme Gradient Boosting, and Linear Regressions. We observe that: (1) Random Forest and eXtreme Gradient Boosting are the best algorithms for predicting performance bugs at a file level with a median of 0.84 AUC, 0.21 PR-AUC, and 0.38 MCC; (2) The proposed performance code metrics have the most significant impact on the performance of our models compared to code and process metrics. In particular, the median AUC, PR-AUC, and MCC of the studied machine learning models drop by 7.7%, 25.4%, and 20.2% without using the proposed performance code metrics; and (3) Our approach can predict additional performance bugs that are not covered by the anti-patterns proposed in the prior studies.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Software testing and debugging**.

KEYWORDS

Performance bugs, Performance anti-patterns, Performance code metrics, Performance bug prediction

ACM Reference Format:
Guoliang Zhao, Stefanos Georgiou, Ying Zou, Safwat Hassan, Derek Truong, and Toby Corbin. 2024. Enhancing Performance Bug Prediction Using Performance Code Metrics. In *21st International Conference on Mining Software Repositories (MSR '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3643991.3644920

1 INTRODUCTION

Large-scale software systems are becoming increasingly more prominent in our society. High performance is critical for the user perception and the quality of software systems. However, user perception and software system's quality can be affected negatively by performance bugs such as software hanging or freezing [70]. For instance, performance bugs (e.g., memory leak bugs) can deteriorate the responsiveness and throughput of software systems, which results in poor user satisfaction and waste of computation [66, 69]. Such bugs exist widely in released software systems, even in well tested commercial products such as Windows 7's Windows Explorer [47, 65].

Prior studies [21, 61, 69, 90, 91] study the characteristics of performance bugs and find that fixing performance bugs is more time-consuming compared to non-performance bugs. Similarly, prior study [98] finds that most performance issues are caused by poor architectural decisions and fixes usually require design-level optimizations instead of simple code changes. Thus, it is important to predict performance bugs to provide developers warnings at an early stage of software development phase and help developers fix performance bugs (e.g., conduct design-level optimization [98]).

Various prior approaches [25, 38, 68, 71, 93] have been proposed to recognize performance bugs at the development phase. For example, anti-patterns are design and implementation styles which lead to poor source code quality [80] and existing studies identify anti-patterns that lead to performance bugs, such as data corruption hang bugs [25], redundant traversal bugs [71], memory and resource leak bugs [38], and synchronization bugs [93]. The anti-patterns (e.g., the exit condition of a loop depends on I/O operations) are used as restricted rules to check if source code contains performance bugs. In addition, prior defect prediction studies [12, 13, 15, 31, 49, 60, 64, 94, 96, 97] use code and process metrics that are derived from source code and code change history to predict defects (not specific to performance bugs). However, the prior approaches have the following limitations.

- **Limitation of prior defect prediction approaches**. The code and process metrics used in defect prediction approaches are designed to detect any type of bugs. Therefore, the prior defect prediction approaches [12, 13, 15, 31, 49, 60, 64, 94, 96, 97] do not consider the code characteristics specific to performance bugs, which may impact prediction accuracy.
- **Limitations of prior anti-pattern based approaches**. Each approach can only identify one performance anti-pattern. It is time-consuming to configure and apply different approaches separately to identify different performance anti-patterns. Moreover, prior approaches [25, 38, 68, 71, 93] cannot predict the performance bugs that do not follow the identified anti-patterns.

To address the mentioned limitations, we propose a set of performance code metrics that capture the code characteristics that can lead to poor performance. Our work combines performance code metrics with the source code and process metrics used in defect prediction studies to build models for predicting performance bugs at file level. Different from the prior studies [25, 38, 68, 71, 93], the proposed performance code metrics measure code features (e.g., the number of I/O operations and the number of loops) instead of finding restricted rules that match anti-patterns. We conduct experiments on 80 open-source Java projects obtained from GitHub. Our work aims to address the following research questions (RQs):

RQ1. What is the performance of our approach in predicting performance bugs at file level?

We utilize seven well-known machine learning algorithms used in prior defect prediction studies [3, 82] such as Random Forest, Naive Bayes, and Logistic Regression. Our findings suggest that Random Forest and eXtreme Gradient Boosting algorithms achieve the best performance with a median value of 0.84 AUC, 0.21 Precision-Recall AUC (PR-AUC), and 0.38 Matthews Correlation Coefficient (MCC) for predicting performance bugs at file level.

RQ2. Which group of metrics affect the performance of our models the most?

In this RQ, we study the performance effects of the group metrics on seven machine learning models. To build machine learning models, we use three groups of metrics (i.e., code metrics, process metrics, and performance code metrics). We observe that the proposed performance code metrics are the most important metrics in the studied machine learning models. Specifically, the AUC, PR-AUC, and MCC of the seven studied machine learning models drop a median of 7.7%, 25.4%, and 20.2% without using the proposed performance code metrics.

RQ3. What are the different types of performance bugs that our approach can predict and fail to predict?

In this RQ, we study the types of performance bugs that our approach can predict and fail to predict. We find that our approach can predict various types of performance bugs, including *performance regression bugs, memory leak bugs, infinite loop bugs, deadlock bugs, and hang bugs*. In addition, our approach can predict additional performance bugs that are not covered by the anti-patterns proposed in the prior studies [25, 38, 68, 71, 93]. On the other hand, we find that our approach fails to predict performance bugs related to running time input or system-specific practices.

Paper organization: The rest of the paper is organized as follows. Section 2 describes the experimental setup, while Section 3 presents our results. We discuss the usages and limitations of our approach in Section 4. The threats to validate are discussed in Section 5 and related work in Section 6. Finally, we conclude and discuss future work in Section 7.

2 EXPERIMENT SETUP

In this section, we introduce the overview of our experimental setup. As shown in Figure 1, we first collect data from GitHub and identify performance bugs. Then, we calculate various types of metrics to capture the characteristics of source code and build performance bug prediction models.

2.1 Selecting projects

As shown in Figure 1, we first query the project information hosted on GHTorrent [42–44, 89] to select our projects. Then, we clone the selected projects from GitHub. We define the following criteria to select our projects.

Selection Criteria. As Java is one of the most popular programming languages according to programming language popularity websites (e.g., Tiobe [83], GitHut [40], and PYPL [74]), we restrict our analysis only to Java projects. Nevertheless, we believe that our approach can be adapted to other programming languages, assuming that the metrics are properly calculated. To avoid working on personal or toy projects [55], we select 1,697 Java projects that each has at least 2,000 commits [76]. Next, we apply the following criteria to further exclude projects from the initial selection:

- In the work by Gousios et al. [41], more than half of the GitHub projects are forked from others. Therefore, we exclude projects that have been archived [39] or forked.
- We exclude projects that do not have bug reports or are managed by bug tracking systems (e.g., Jira and Bugzilla) [55].
- We filter out projects with a lifespan less than a year, otherwise we can not have enough performance bug reports and metrics to build accurate prediction models [94, 95, 99].
- We exclude projects that have limited performance bug fixing commits. Following the prior studies [95], we count the number of performance bug fixing commits from a year period in each project and choose the 75% percentile of the number of performance bug fixing commits (i.e., 74) as the threshold to filter out projects.
- We remove projects without performance bug fixing commits in their last six month periods.

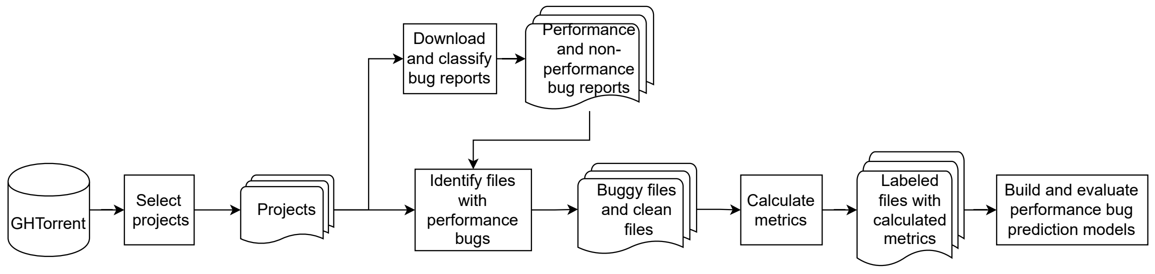
GHTorrent
Select projects
Projects
Download and classify bug reports
Performance and non-performance bug reports
Identify files with performance bugs
Buggy files and clean files
Calculate metrics
Labeled files with calculated metrics
Build and evaluate performance bug prediction models

Figure 1: The overview of our approach.

After applying the aforementioned criteria, we reduce our projects to 80. Our projects are from different fields, such as Elasticsearch search engine [36] OpenLiberty cloud micro-service [51], and so on. On average, the collected projects have 304,619 source code files and 34,268,345 lines of code.

2.2 Identifying performance bug fixing commits

As suggested in prior literature [26, 31], we use the performance bug fixing commits to identify the source code files that contain performance bugs. We use two methods to identify the commits that fix performance bugs.

- Searching for commits containing any of the performance-related keywords (i.e., *deadlock*, *contention*, *infinite loop*, *memory leak*, *performance*, *high memory*, *stuck*, *hang*, *slow*, *speed up*, and *100% CPU*) in the commit message.
- Identifying commits fixing bug reports that contain the performance-related keywords.

For each project, we include the performance bug fixing commits in the latest history year of the project. We filter out performance bug fixing commits by keeping only the commits that have source code changes, i.e., changes to **.java* files. After filtering, there is a median of 175 performance bug fixing commits in each project.

2.3 Identifying the files with performance bugs

For each project, we exclude the test files since we are interested in studying the implications of performance bugs. A file is considered as a test file if the filename contains the *test* keyword.

Figure 2 shows the approach that we use to label the files that have performance bugs. Following the prior studies [95, 96], we collect the source code files of each project six months before the latest commit time in the project. To identify the files that have performance bugs, we make use of the commits collected in the previous step. Then, we mark the source code files that are modified in the performance bug fixing commits as performance bug files. However, there are performance bug files introduced after the time we collect the source code files. Therefore, we use SZZ [77] algorithm to search for the bug introducing commits. We exclude the buggy files that are introduced after the time that we collect the source code files.

In our projects, we find that a median ratio of 0.84% files have performance bugs, which means there is one performance bug in every 119 files. Performance bugs are not frequent, however, they caused many severe failures in production and resulted in software worth hundreds of millions being abandoned [67, 75].

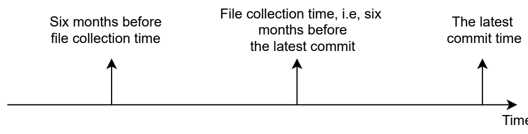
Six months before file collection time
File collection time, i.e, six months before the latest commit
The latest commit time
Time

Figure 2: The approach that is used to label the files that have performance bugs

2.4 Capturing the code characteristics of performance bugs

In this section, we propose a number of metrics to capture the code characteristics of performance bugs. The proposed performance code metrics are shown in Table 1. Below, we explain the rationale of each performance code metric.

Table 1: The proposed performance code metrics. The metrics are aggregated to a file level using average scheme.

Performance code metric	Description
Num_if_in_loop	Number of if-conditions that are inside loops
Num_loop_in_if	Number of loops that are inside if-conditions
Num_file_operations	Number of file operations
Num_file_operations_in_loop	Number of file operations inside loops
Num_database_operations	Number of databases operations
Num_database_operations_in_loop	Number of databases operations inside loops
Num_collection	Number of operations on collection data structures
Num_collection_in_loop	Number of operations on collection data structure inside loops
Num_synchronization	Number of synchronization operations
Num_nested_loop	Number of nested loops
Num_nested_loop_in_crit	Number of nested loops in critical sections in synchronization
Num_thread	Number of thread operations

Performance bugs with non-intrusive fixes. Nistor et al. [68] analyze performance bugs that have *non-intrusive fixes* and find that each performance bug in the family is associated with a loop and an if-condition. For example, in the situation that an if-condition inside a running loop is met and there is no break command to exit from the loop, the loop continues its execution until an end condition is met. Such cases result to unnecessary execution and performance deterioration. To represent the code characteristics of such performance bugs, we propose the *Num_if_in_loop* and *Num_loop_in_if* metrics (as listed in Table 1) to count the number of if-conditions in loops and the number of loops in if-conditions. The rationale is that when loops and if-conditions are heavily nested, developers are likely to miss exit-conditions to stop the loops.

```
1  boolean elExp = False;
2  while (nodes.hasNext()) {
3       ELNode node = nodes.next();
4       if (node instanceof ELNode.Root) {
5            if (((ELNode.Root) node).getType() == '$'){
6                 elExp = true;
7                 }
8            }
9  }
```

(a) The performance bug with non-intrusive fix

```
1  boolean elExp = False;
2  while (nodes.hasNext()) {
3       + if (elExp) break; // FIX
4       ELNode node = nodes.next();
5       if (node instanceof ELNode.Root) {
6            if (((ELNode.Root) node).getType() == '$'){
7                 elExp = true;
8                 }
9            }
10 }
```

(b) The fix for the performance bug example

Figure 3: An example of performance bugs that have non-intrusive fix and its resolution in project Tomcat [84]

Figure 3 shows a performance bug that has non-intrusive fix. When the `if-condition` at line 6 is met, the remaining computations of the loop are unnecessary because there is no point to set the `elExp` variable to true again. Such cases result in performance deterioration. Thus, developers fix the performance by adding a break statement at line 3 as shown in Figure 3(b). The proposed performance code metric *Num_if_in_loop* can help us model such performance bugs.

Data corruption hang bugs. Dai et al. [25] report that the *data corruption hang bugs* can cause infinite loops in software and make software unavailable to its users, which is among the most common performance issues [24, 28, 29, 53]. Moreover, data corruption hang bugs happen when a loop's exit-condition is affected by an I/O operation, e.g., reading from a file or database. For example, if a file read operation inside a loop is executed and the file is corrupted, then this can lead to an infinite loop. Therefore, we propose the *Num_file_operations*, *Num_file_operations_in_loop*, *Num_database_operations*, and *Num_database_operations_in_loop* metrics (as listed in Table 1) to count the number of I/O operations inside a method and loop. The rationale is that the execution of a loop is likely to be affected by the I/O operations in the loop or method.

Figure 4 illustrates a data corruption hang bug in file Benchmark-Throughput.java [9] of project Hadoop Distributed File System.

```
78 private void readLocalFile(Path path, ...) throws
   IOException {
     ...
83   InputStream in = new FileInputStream(…);
84   byte[] data = new byte[BUFFER_SIZE];
85   long size = 0;
86   while (size >= 0){
87        size = in.read(data);
     }}
```

Figure 4: An example of data corruption hang performance bugs in file BenchmarkThroughput.java [9] of project Hadoop Distributed File System

The data corruption hang bug is reported in issue #13514 [10]. As shown in Figure 4, when the `BUFFER_SIZE` variable at line 84 is 0, the `InputStream` in at line 87 reads a zero-size byte array and returns 0. The exit-condition of the while loop at line 86 become infeasible because `size < 0` is never satisfied. The `in.read(data)` at line 87 is a file operation. Thus, the proposed performance code metrics *Num_file_operations* and *Num_file_operations_in_loop* can help us model this data corruption hang bug.

Redundant traversal bugs. Olivo et al. [71] have studied the *redundant traversal bugs* that appear when methods repeatedly iterate over a data structure, without modifying it after a successive traversal. Since a data structure is not modified between traversals, the results from one traversal can be reused and the repeatedly traversals are a waste of computational resources, which results in performance degradation. Ghanavati et al. [38] find that inefficient usage of data structures (e.g., not removing stale objects from data collections) is one of the common root causes of memory leak bugs. Inspired by the above studies [38, 71], we propose the *Num_collection* and *Num_collection_in_loop* metrics (as listed in Table 1) to count the number of operations on data structures. The rationale is that the extensive usage of data structures makes it challenging for developers to correctly manage them, which can lead to inefficient usages of data structures [88].

```
583 public void drawItem(XYDataset dataset, int series, ...) {
      ...
606   OHLCDataset highLowData = (OHLCDataset) dataset;
      ...
648   itemCount = highLowData.getItemCount(series);
      ...
651   for (int i=0; i< itemCount; i++) {
652        double pos = domainAxis.valueToJava2D(
653                  highLowData.getXValue(series, i), dataArea,
654                  domainEdge);
655        if (lastPos != -1) {
656             xxWidth = Math.min(xxWidth,
657                       Math.abs(pos - lastPos));
658        }
659        lastPos = pos;
      }}}
```

Figure 5: An example of redundant traversal performance bugs in file CandlestickRenderer.java [52] of project JFreeChart

Figure 5 depicts a redundant traversal bug in file Candlestick-Renderer.java [52] in JFreeChart. The redundant traversal bug is reported by Olivo et al. [71]. As shown in Figure 5, the `drawItem()` method iterates over all points in the dataset (i.e., `XYDataset dataset` at line 583) in order to draw a single data point. The `drawItem()` method traverses all data points to compute a variable called `xxWidth` at line 656. The `xxWidth` variable records the minimum gap between adjacent x-coordinates of all points in the dataset. If a dataset is not modified between successive calls to the `drawItem()` method, the re-computation of `xxWidth` in each call to the `drawItem()` method is unnecessary. The `getXValue` at line 653 is an operation on `highLowData` which is a set object as defined by `OHLCDataset highLowData` at line 606. Thus, the proposed performance code metrics *Num_collection*, *Num_collection_in_loop*, and *Num_if_in_loop* can help us model this redundant traversal bug.

Synchronization performance bugs. Thread synchronization is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute a particular program segment known as critical section. However, synchronization performance bugs can cause synchronization issues and degrade the performance of a multi-threaded system. To this end, Zhang et al. [93] investigate synchronization performance bugs and find that the nested loops in critical sections are likely to cause synchronization issues. This is because nested loops are time-consuming and once a thread enters the critical section and starts to execute the nested loops, then all other threads have to wait for the nested loops to finish. Therefore, we propose the *Num_nested_loop* and *Num_nested_loop_in_crit* metrics (as listed in Table 1) to count the number of nested loops found in methods' critical sections. In addition, we propose the *Num_synchronization* and *Num_thread* metrics (as listed in Table 1) to represent the number of synchronization operations (e.g., acquiring a mutex) and thread operations (e.g., creating a thread) in a method. The rationale is that a method is more likely to have synchronization issues if it has more synchronization and thread operations.

```
993   public void handle(JobEvent event) {
          ...
998     try {
                  writeLock.lock();
999               ...
1002              doTransition(event.getType(), event);
1003        } catch
              ...
1018     finally {
1019       writeLock.unlock();
      }}
```

Figure 6: An example of synchronization performance bugs in file JobImpl.java [11] of project Hadoop MapReduce

Figure 6 shows a synchronization bug in file JobImpl.java [11] of project Hadoop MapReduce. The synchronization bug is reported in issue #4813. As shown in Figure 6, the `handle()` method acquires the `writeLock` lock at line 999, executes the method `doTransition()` at line 1002, and releases the `writeLock` lock at line 1019. However, if the method `doTransition()` takes too long to finish, some threads in the system have to wait for the release of the `writeLock` lock and the system might become unresponsive to users. The code conducts synchronization operations (e.g., acquiring and releasing a lock). Thus, the proposed performance code metric *Num_synchronization* can be used to model such synchronization bug.

The proposed performance code metrics can be calculated via static source code analysis and require no system specific knowledge. We focus on loops when we propose the performance code metrics because most computation time is spent inside loops and most performance bugs involve loops [46, 70, 86, 92].

Prior studies [19, 27, 30] propose metrics to capture the performance regression or improvement introduced by source code changes. Compared with the metrics proposed in the prior studies [19, 27, 30], the proposed performance code metrics in our paper are oriented to capture the code characteristics of performance bugs. For example, the prior studies [19, 27, 30] count only the number of loops in a method, while we count (1) the number of loops in if-conditions, (2) the number of file operations in loops, and (3) the number of database operations in loops to capture the code characteristics of data corruption hang bugs [25].

2.5 The code and process metrics

To represent the source code and code change history, we include the code and process metrics that are widely used in the prior defect prediction studies [12, 13, 15, 31, 49, 60, 64, 94, 96, 97]. Table 2 summarizes the code and process metrics.

Table 2: The code and process metrics used in our study. The last column refers to the scheme to aggregate method-level metrics to a file level ("none means that no aggregation is performed for metrics that are calculated at a file level").

Code Metrics		
Metric name	**Description**	**Aggregation scheme**
LOC	Lines of code in a method	average
CL	Comment lines in a method	average
NSTMT	Number of statements in a method	average
RCC	Ratio comments to codes of a method	average
MNL	Max nesting level of a method	average
CC	McCable cyclomatic complexity of a method	average
FANIN	Number of input data of a method	average
FANOUT	Number of output data of a method	average
Process Metrics		
Num_rev	Number of revisions	None
Num_perf_rev	Number of revisions a file was involved in fixing performance bugs	None
Num_non_perf_rev	Number of revisions a file was involved in fixing non-performance bugs	None
Num_perf_bug	Number of performance bugs happened in a file	None
Num_non_perf_bug	Number of non-performance bugs happened in a file	None
Added_loc	Lines of code added in a file in the history commits	average
Deleted_loc	Lines of code deleted in a file in the history commits	average

3 EXPERIMENT RESULTS

In this section, we present motivations, approaches, and results of the studied research questions.

3.1 RQ1. What is the performance of our approach in predicting performance bugs at file level?

To help developers in identifying performance bugs at the development phase, we provide a unified approach that can predict various types of performance bugs. Specifically, we combine the performance code metrics with the code and process metrics used in defect prediction studies [12, 13, 15, 31, 49, 60, 64, 94, 96, 97] to build models for predicting performance bugs at file level.

Selecting machine learning algorithms. In our experiment, we select seven machine learning algorithms, i.e., Random Forest (RF), eXtreme Gradient Boosting (XGBoost), Logistic Regression (LR), Support Vector Machines (SVM), Complement Naive Bayes (CNB), MultiLayer Perceptron neural network (MLP), and Decision Tree (DT), that are widely used to predict bugs [3, 82].

Creating training and testing datasets. In our study, the performance bug prediction models are within-project models. We apply the out-of-sample bootstrap technique [3, 32, 81, 82] to create training and testing datasets for each project. The out-of-sample bootstrap is recommended for highly-skewed datasets [50, 82], which is the case in performance bug prediction where the number of the files that have performance bugs is small comparing with clean files. The out-of-sample bootstrap is composed from two steps:

- For a project with N files, a bootstrap sample of size N files is randomly drawn with replacement from the original files of the project.
- A model is trained and validated using the bootstrap sample and tested using the files that do not appear in the bootstrap sample. On average, 36.8% of the files do not appear in the bootstrap sample, since the sample is drawn with replacement [32].

We repeat the out-of-sample bootstrap process for 100 times for each project.

Conducting class re-balancing on the training datasets. As suggested by the prior study [82], class re-balancing is able to improve the performance of defect prediction models. Following the prior study [82], we use the SMOTE class re-balancing to balance the number of files that have performance bugs in our training datasets.

Performance metrics. We use three metrics to evaluate the performance of our models, including Area Under the receiver operator characteristic Curve (AUC), Matthews Correlation Coefficient (MCC), and Area under the Precision-Recall curve (PR-AUC).

Training and evaluating performance bug prediction models. To find the best hyper-parameters for our models and the SMOTE class re-balancing technique, we use the grid search optimization approach. The grid search parameter optimization approach consists of three steps.

- **Set candidate values for parameters**. We manually input a set of candidate values for every parameter in each machine learning algorithm and the SMOTE technique.
- **Iterate candidate parameter values**. We iterate all possible combinations of candidate parameter values. For each possible combination, we train models using 80% of the training dataset and test them on the rest 20%.
- **Select the optimal parameter values**. After iterating all the combinations of parameters, we select the ones that achieves the highest performance.

In a project, for each training and testing datasets that are generated from the out-of-sample bootstrap process, we train a model on the training dataset using the optimal parameter values and test the AUC, PR-AUC, and MCC of the model for predicting performance bugs in the testing dataset. Since we have seven machine learning algorithms, 80 projects, and 100 datasets from each project, we build and test 56,000 (i.e., 7*80*100) models, in total.

Comparing the performance of the models. After evaluating the performance of machine learning models on our projects, we draw beanplots of the AUCs, PR-AUCs, and MCCs to visualize the performance of the models. Next, we conduct Friedman tests followed by Nemenyi's post-hoc tests to compare the performance of different models. Both Friedman test and Nemenyi's post-hoc test are non-parametric tests that do not require the analyzed data to meet any assumptions [72]. We use 0.01 as the significant level when applying Friedman tests and Nemenyi's post-hoc tests.

Table 3: The p-values of the Friedman tests of comparing the performance of machine learning algorithms

Performance metrics		
MCC	PR-AUC	AUC
5.7e-78	2.0e-137	1.5e-47

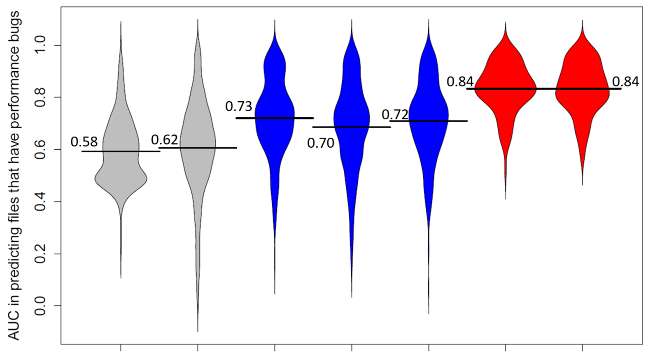

Figure 7: The AUC of machine learning (ML) algorithms for predicting performance bugs at file level. The ML algorithms are grouped in Gray, Red, and Blue colors based on the results from Nemenyi's post-hoc tests.

Results: The Random Forest and eXtreme Gradient Boosting algorithms achieve the best performance in predicting files with performance bugs. As shown in Table 3, the p-values of the Friedman tests are all smaller than 0.01, which means that the performance of the studied machine learning algorithms are significantly different under all performance metrics. Figures 7, 8, and 9 show the AUCs, PR-AUCs, and MCCs of the studied algorithms for predicting files that have performance bugs. As shown

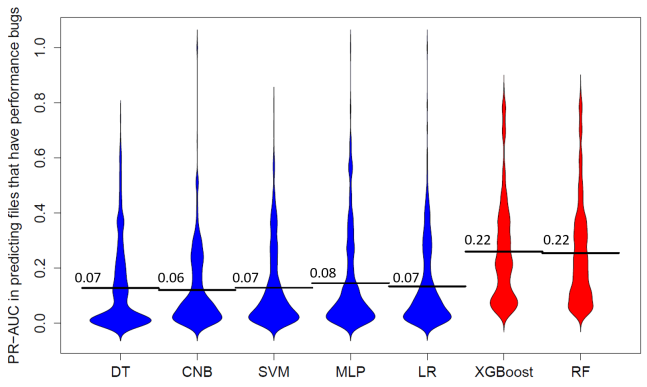

Figure 8: The PR-AUC of machine learning (ML) algorithms for predicting performance bugs at file level. The ML algorithms are grouped in Gray, Red, and Blue colors based on the results from Nemenyi's post-hoc tests.

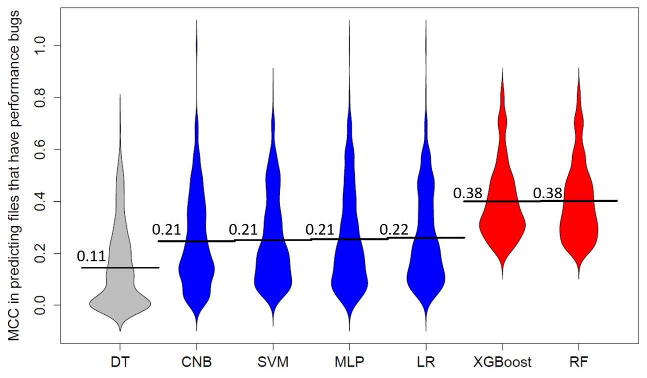

Figure 9: The MCC of machine learning (ML) algorithms for predicting performance bugs at file level. The ML algorithms are grouped in Gray, Red, and Blue colors based on the results from Nemenyi's post-hoc tests.

in Figures 7, 8, and 9, the Random Forest and eXtreme Gradient Boosting algorithms achieve the best performance with a median of 0.38 MCC, 0.22 PR-AUC, and 0.84 AUC. The 0.84 median AUC value (ranges from 0 to 1) suggests that the algorithms achieve a high quality of performance for distinguishing the files with performance bugs from the files without performance bugs. As suggested by the existing studies [17, 22, 78], the median 0.22 PR-AUC (ranges from 0 to 1) and 0.38 MCC (ranges from -1 to 1) suggest relatively mediocre performance. Overall, the Random Forest and eXtreme Gradient Boosting algorithms achieve acceptable performance for predicting performance bugs at the file level. One explanation to the mediocre PR-AUC and MCC values is that the ratio of source code files that have performance bugs is very low, i.e., 0.84%, in our studied projects. It is a challenging task to train machine learning algorithms to predict performance bugs with such a low ratio.

The Complement Naive Bayes and Decision Tree algorithms have the worst performance for predicting performance bugs. A possible reason is that the Complement Naive Bayes and Decision Tree algorithms are not suitable for predicting highly-skewed performance bugs, since there is a very small number of performance bug files.

Summary of RQ 1

Among the examined algorithms, the Random Forest and eXtreme Gradient Boosting algorithms achieve the best performance in predicting performance bugs at file level.

3.2 RQ2. Which group of metrics affect the performance of our models the most?

To build performance bug prediction models, we use three groups of metrics including code metrics, process metrics, and the proposed performance code metrics. To understand whether the proposed performance code metrics are indeed useful in predicting performance bugs, we analyze the effects that each group of metrics has on the prediction models.

Approach: To test the effect of a group of metrics on the machine learning models, we apply the effect calculation process as suggested by the prior study [82]. The calculation of the effect of each group of metrics on a machine learning algorithm is made of two steps:

- For each dataset (i.e., the files and their metrics), we first randomly permute all the values of a group of metrics and obtain a new dataset.
- We apply out-of-sample bootstrap process on the new dataset and conduct the parameter optimization to select the optimal parameter values for the machine learning algorithm. Next, we evaluate the performance of the machine learning algorithm for predicting performance bugs on the new dataset. We compute the differences between the performance of the models that are built on the original dataset and the dataset with the randomly-permuted metrics. The performance differences are measured in AUC, PR-AUC, and MCC and are used as the effect of the group of metrics.

We test the effects of all groups of metrics on all studied machine learning algorithms except the Decision Trees and Complement Naive Bayes algorithms. We did not include the Decision Trees and Complement Naive Bayes algorithms in the metrics effects study because these two algorithms have poor performance for predicting performance bugs as discussed in Section 3.1.

Results: The proposed performance code metrics impact the performance of our models the most. Table 5 shows the results of Friedman tests followed by Nemenyi's post-hoc tests of comparing the effects of the groups of metrics. As shown in Table 5, the proposed performance code metrics are consistently ranked in the best performing group (i.e., first group). Table 4 shows the effect that each group of metrics has on the five studied machine learning algorithms, measured in AUC, PR-AUC, and MCC. As shown in Table 4, the proposed performance code metrics have the highest effects for our models. For example, building a Random Forest model without the proposed performance code metrics can reduce the model's median AUC, PR-AUC, and MCC by 7.0%, 27.2%, and 19.0%, respectively.

Table 4: The effects of groups of metrics on performance bug prediction models. We highlight the highest effects of groups of metrics in each algorithm.

Groups of metrics		Median of performance drop percentage				
		RF	XGB	LR	MLP	SVM
Performance code metrics	MCC	**19.0%**	**18.5%**	**20.3%**	**20.0%**	13.2%
	PR-AUC	**27.2%**	**24.7%**	13.3%	**23.2%**	14.3%
	AUC	**7.0%**	**6.9%**	**9.2%**	**11.2%**	**5.5%**
Code metrics	MCC	5.9%	4.3%	3.1%	5.7%	4.1%
	PR-AUC	10.0%	10.4%	7.9%	0.1%	6.6%
	AUC	6.2%	6.7%	1.4%	0.1%	0.2%
Process metrics	MCC	7.8%	4.4%	14.2%	2.7%	**18.0%**
	PR-AUC	11.9%	8.1%	**17.7%**	10.6%	**22.7%**
	AUC	1.3%	1.2%	2.7%	1.9%	3.0%

Table 5: The classified groups based on the results from Nemenyi's post-hoc tests

Performance metrics	Groups assigned based on the results from Nemenyi's post-hoc tests		
	First group	Second group	Third group
MCC	Performance code metrics	Process metrics	Code metrics
PR-AUC	Performance code metrics	Process metrics	Code metrics
AUC	Performance code metrics	Code metrics	Process metrics

Summary of RQ 2

The proposed performance code metrics impact the performance of our prediction models the most. Specifically, the AUC, PR-AUC, and MCC of the five studied machine learning models drop a median of 7.7%, 25.4%, and 20.2% without using the proposed performance code metrics.

3.3 RQ3. What are the different types of performance bugs that our approach can predict and fail to predict?

To understand the types of performance bugs that our approach can predict, we analyze the performance bugs predicted by our approach.

Collecting performance bugs predicted by our approach. First, we build Random Forest models using its optimal parameters values and training datasets selected in the experiments in Section 3.1. Then, we use the Random Forest models to predict files that have performance bugs in the testing datasets from the 80 experiment projects. Random Forest model predicts the probabilities of files to have performance bugs and we use 0.5 as the threshold to convert the probabilities to labels (i.e., clean files and files containing performance bugs). To the end, we find that 31,556 files are predicted to have performance bugs in the testing datasets from the 80 experiment projects.

Analyzing the predicted performance bugs. To have 95% confidence level and 5% confidence interval, we randomly select a sample of 340 predicted to have performance bugs. Next, we find the performance bug fixing commits for fixing the performance bugs in the sampled files. Then, the first and second authors of the paper manually go through the files and performance bug fixing commits to study the types of the performance bugs. We compare the results from the two authors and obtain a 0.85 Cohen's Kappa score (i.e., a almost perfect agreement between two authors [62]).

Table 6: The result of the manual analysis about the types of performance bugs predicted by our approach

Types of performance bugs	Number of files	Percentage of files
Performance regression bug	236	69.4%
Memory leak bug	26	7.6%
Infinite loop bug	23	6.8%
Deadlock bug	21	6.2%
Stuck/hang bug	21	6.2%
Non-performance bug	13	3.8%

Results: Our approach can predict various types of performance bugs. The result of the manual analysis is shown in Table 6. Our approach predicts performance regression bugs (i.e., code that introduces response time degradation and increases resource utilization [18]), memory leak bugs, infinite loop bugs, deadlock bugs, and hang bugs. As shown in Table 6, 236 (i.e., 236/340 = 69.4%) files contain inefficient code that results in performance regression bugs. For example, file *PublicationTransportHandler.java* [35] in project Elasticsearch causes significant system response time degradation and is predicted to have performance bugs by our approach. The performance regression bug is fixed in commit c5315744 [34]. There are 13 (i.e., 236/340 = 3.8%) false positive predictions, i.e., non-performance bugs, made by our approach as shown in Table 6.

Our approach can capture more performance bugs than the prior anti-pattern studies. From analyzing the files that have memory leak bugs, deadlock bugs, and hang bugs, we find that our approach can predict additional performance bugs that are not covered by the anti-patterns proposed in the prior studies [25, 38, 68, 71, 93]. We show examples as follows.

- The predicted buggy file *JournaledGroup.java* [2] in project Alluxio contains a memory leak bug. The memory leak bug is caused by unclosed threats when users iterate directories (detailed description can be found in commit 58c24eb8 [1]).
- The predicted buggy file *ConnectivityService.java* [6] in project platform_frameworks_base contains a deadlock bug. The deadlock bug is caused by lock acquiring sequences (detailed description can be found in commit 465088ed [5]).
- The predicted buggy file *AbstractBuilding.java* [58] in project minecolonies has a hang bug that is caused by keeping searching for objects that do not exist (mentioned in commits 508e7638 [59]).

Our approach is able to predict more performance bugs because we measure code characteristics that lead to performance bugs instead of measuring the restricted rules. In addition, we combine performance code metrics with code and process metrics extracted from development history to predict performance bugs.

We intend to study the types of performance bugs that our approach fails to predict. We are also interested in understanding the root causes of the performance bugs that cannot be predicted using our approach. Thus, we conduct a manual analysis of the false negatives (i.e., files with performance bugs that our approach fails to predict). We select a sample of 340 false negatives using 95% confidence level and 5% confidence interval from the prediction results of the Random Forest models.

Table 7: The result of the manual analysis on the types of performance bugs that our approach fails to predict

Type of performance bugs	Number of files	Main root causes
Performance regression bug	251	Inefficient functional logic, inefficient memory usage
Memory leak bug	12	Specific program input, unclosed threats at runtime
Infinite loop bug	18	Specific program input, wrong API usage
Deadlock bug	13	Database binary storage, JDK related
Stuck/hang bug	17	Specific program input, configuration problem, device incompatibility
Non-performance bug	29	None

We go through the files and performance bug fixing commits to study the types of the performance bugs and the root causes of the performance bugs. The result of the manual analysis is shown in Table 7. We show examples of each root causes as follows.

Inefficient function logic: code contains inefficient algorithms that introduce unnecessary calculations. The file, namely MekanismRenderType.java in project Mekanism, contains a performance regression bug caused by frequent unnecessary data searches (detailed description can be found in commit ee83cb14 [63]).

Inefficient memory usage: code holds rarely used objects in memory. The file, namely AssetFeeView.java in project bisq, contains a performance regression bug caused by building and holding a not used object (about 40MB) in memory (detailed description can be found in commit 55b070f9 [16]).

Specific input: performance bugs that happen when specific data are input. The file, namely ColorExtractor.java in project platform_frameworks_base, contains a stuck bug that happens only when importing an image as a wallpaper (detailed description can be found in commit 5abc71b2 [7]). In addition, the file, namely DefaultSearchContext.java, in project Elasticsearch contains a memory leak bug that occurs only when a search query has certain fields enabled (detailed description can be found in commit 6b51d85c [33]).

Wrong API usage: performance bugs that are caused by wrong API usages. The file, namely LayoutUtil.java in project webfx, contains an infinite loop bug that is caused by the mixing usage of two API methods (detailed description can be found in commit c241aa13 [85]).

Device incompatibility: performance bugs that happen because of the environment on specific devices. The file, namely BiometricsEnrollEnrolling.java in project platform_packages_apps _settings, contains a stuck bug that happens on only fingerprint devices (detailed description can be found in commit aea1bdec [8]).

Database binary storage operations: performance bugs that are caused by wrong usages of database operations APIs. The file, namely BinaryStorageEntity.java in project hapi-fhir, contains a deadlock bug that happens when the database binary storage is used (detailed description can be found in commit c6777578 [48]).

JDK related: performance bugs that are caused by JDK issues. The file, namely DirtyPrintStreamDecorator.java in project buck, contains a deadlock bug when the code is executed using JDK 11 (detailed description can be found in commit 2258ba13 [37]).

Based on the root causes of the false negative performance bugs shown in Table 7, we find that there are performance bugs that can only be detected using *the information collected from running time (e.g., testing phase)*, such as performance bugs caused by specific input, device incompatibility, and JDK issues. For the performance bugs caused by other root causes (i.e., inefficient function logic, inefficient memory usage, wrong API usage, and database binary storage operations), the code characteristics of the root causes can be system-specific as different systems have different coding practices. In the future, we plan to propose *system-specific* performance code metrics to measure the code characteristics of inefficient function logic, inefficient memory usage, wrong API usage, and database binary storage operations in software systems

Summary of RQ 3

Our approach is able to predict various types of performance bugs, while 69.4% predicted bugs are performance regression bugs. Our approach fails to predict performance bugs that are related to running time input or system-specific practices.

4 DISCUSSION

In this section, we discuss the limitations and usage of our approach.

4.1 Limitations of our approach

A limitation of our approach is to predict performance bugs in new project with limited development history. This is because our approach relies on the performance bugs history of a project to train machine learning models in predicting hidden performance bugs in the same project.

4.2 Usage of our approach

In practice, developers may not have sufficient time to design performance test cases to cover each file or function in a software system [27]. To this end, developers can use our approach to prioritize their testing effort on a small set of source code files that are predicted to have performance bugs.

As the first step, developers can use our approach to predict files that have performance bugs. Then, they can introduce performance test cases to test the performance (e.g., execution time, CPU usages, and memory usages) of files that are considered to have performance bugs. In addition, developers can use approaches proposed in the existing studies [19, 30] to test the performance.

Chen et al. [19] propose an approach to predict which test cases are likely to manifest performance regression in a commit and Ding et al. [30] propose an approach to predict whether a function test case is able to demonstrate the performance improvement after fixing a performance bug. Developers can use the approach proposed by Chen et al. [19] to select the existing test cases that can manifest the performance regression in the files. After fixing the performance bugs, developers can use the approach proposed by Ding et al. [30] to identify the test cases that can demonstrate the performance improvement.

5 THREATS TO VALIDITY

Threats to external validity are related to the generalizability of our results. To ensure the generalizability of our study, we conduct experiments on 80 popular Java projects. The 80 Java projects are selected following the criteria described in Section 2.1. These projects belong to different domains. We believe that our approach can be adapted to other projects (e.g., projects developed in different programming languages or projects using Jira issue tracking systems), assuming that the metrics are properly calculated.

A limitation of our approach is to predict performance bugs in new project with limited development history. This is because our approach relies on the performance bugs history of a project to train machine learning models in predicting hidden performance bugs in the same project.

Threats to internal validity concern factors out of our control that may affect the experiment results. We use keyword searching to identify the performance bug reports and the performance bug fixing commits following the prior studies. However, there might be *false positive* performance bug reports and performance bug fixing commits in our dataset. To mitigate this issue, we manually labelled a random statistical sample to ensure that our results does not have many false positives.

Similarly, the keyword *test* is applied to filter out test files. However, there may be test files within our dataset that do not contain the keyword *test*, such as the *BenchmarkThroughput.java* file shown in Figure 4.

6 RELATED WORK

In this section, we discuss prior studies that aim to predict performance bugs using software testing and source code analysis.

6.1 Predicting performance bugs via software testing

Grechanik et al. [45] propose a solution for automatically identifying performance problems in applications using black-box software testing. Jovic et al. [54] introduce an approach that can identify performance bugs in an application by monitoring the behavior of the application from different deployments (e.g., application deployed on machines with different size of memory). Xiao et al. [86] suggest an approach to detect workload-dependent loops that contain time-consuming operations via monitoring the behavior of applications under large workloads. Nistor et al. [70] introduce an automated approach that reports code loops whose computations are repetitive and unnecessary. Killian et al. [56] propose an approach to predict latent performance bugs by combining state-space exploration with time-based event simulation. Xu et al. [87] introduce a technique that summarizes the run-time activity in terms of data copies and identifies unnecessary operations in software systems. Coppa et al. [23] present a method for helping developers to discover hidden inefficiencies in source code. Altman et al. [4] present a tool to abstract the concrete execution states of Java applications and diagnose the root cause of idle time in applications. Han et al. [47] propose an approach that mines call-stack traces to debug the performance problems in order to identify performance bugs. Pradel et al. [73] present a performance regression testing technique to test the performance of thread-safe classes. The aforementioned studies predict performance bugs on running applications and analyzing their run-time information (e.g., call-stack). In contrast to the aforementioned studies, our approach leverages different source of data, i.e., source code and post-release historical information, to predict performance bugs.

Laaber et al. [57] propose an approach to predict the stability of a performance benchmark testing using statically-computed source code features without running the benchmark testing. Different from the existing approach [57], our approach aims to predict the performance bugs in software systems without executing any test cases. Oliveira et al. [27] propose an approach to predict if a new commit affects the performance of a benchmark using the run-time monitoring information collected from running the same benchmark from previous commits. Chen et al. [19] propose prediction models to select the test cases that can manifest the performance regression in a commit. Ding et al. [30] propose an approach to predict the test cases that can demonstrate the performance improvement after fixing a performance bug. These approaches [19, 27, 30] aim to identify test cases that can manifest the existence of performance bugs in software systems. The identified test cases do not reveal the exact locations of performance bugs. Unlike the existing studies [19, 27, 30], our approach aims to identify the locations (e.g., files) of the performance bugs in software systems using static source code analysis.

6.2 Predicting performance bugs via source code analysis

To predict performance bugs, Jin et al. [53] use efficiency rules (e.g., function f1 is always followed by f2) that are extracted from the fixes of 109 performance bugs. Specifically, the authors find 332 previously unknown performance bugs in MySQL, Apache, and Mozilla applications. They identify 219 out of 332 performance bugs by applying the extracted efficiency rules across applications. Zhang et al. [93] propose a tool to predict synchronization performance bugs based on common anti-patterns of synchronization performance bugs. Chen et al. [20] propose an automated framework to predict performance anti-patterns in Object-Relational Mapping by analyzing global call graphs and data graphs. Nistor et al. [68] predict performance bugs that can be fixed by adding one line of code inside a loop. Song et al. [79] design a root-cause and fix-strategy taxonomy for inefficient loops and, then, propose a static analysis approach to automatically predict whether a loop is inefficient based on the proposed taxonomy. Dai et al. [25] propose Dscope, a tool to predict data-corruption related performance bugs. Dscope analyzes I/O operations and loops in software packages and identifies loops whose exit-conditions can be affected by I/O operations. Xu et al. [88] propose a tool to find problematic uses of data structures by identifying the objects that are added to collections. Similarly, Bhattacharya et al. [14] present an algorithm that can predict the excessive generation of temporary data structures within a loop by determining which objects created within the loop can be reused. Olivo et al. [71] introduce a tool to identify redundant traversal performance bugs, i.e., a collection of data structures is repeatedly iterated but the collection has not been modified between successive traversals.

The aforementioned studies use anti-patterns to predict performance bugs. However, each type of performance bugs has a unique anti-pattern and requires a different approach to identify it. Adopting a different approach to predict each type of performance bug is time-consuming. Compared to the aforementioned studies, we combine performance code metrics with code and process metrics used in defect prediction to predict performance bugs.

7 CONCLUSIONS

In this paper, we present an approach that predicts performance bugs in software systems by integrating source code analysis and mining code change history. We propose performance code metrics to capture the code characteristics of performance bugs. Models are built to predict performance bugs using code, process, and performance code metrics. We conduct extensive experiments on 80 Java projects to evaluate the performance of seven machine learning algorithms for predicting performance bugs.

Overall, the contributions of this paper are listed as follow:

- We propose performance code metrics to measure characteristics of poor performance code instead of measuring restricted performance anti-patterns [25, 38, 68, 71, 93].
- We propose a unified approach that can predict various types of performance bugs by combining our proposed performance code metrics with code metrics and process metrics used in prior defect prediction studies [12, 13, 15, 31, 49, 60, 64, 94, 96, 97].

We provide a replication package[1] of our approach. The replication package includes a list of our experiment GitHub projects along with their versions, the experiment dataset, the metrics calculation scripts, and the experiment scripts to replicate our study. In the future, we will extend our dataset to include more projects with the newest versions of the studied projects. We intend to report the predicted files with the performance bugs to the developers and ask developers' feedback on the performance of our approach in order to further improve our approach. In addition, we plan to study the evolution of performance bugs in software systems over time and test the performance of our approach over different software versions.

REFERENCES

[1] Alluxio. [n. d.]. Commit 58c24eb8 in project alluxio. Retrieved December 02, 2021 from https://github.com/Alluxio/alluxio/commit/58c24eb80baca6c2f4efbb9fb7349fd80c1d3c0a

[2] Alluxio. [n. d.]. JournaledGroup.java in project alluxio. Retrieved December 02, 2021 from https://github.com/Alluxio/alluxio/blob/master/core/server/common/src/main/java/alluxio/master/journal/JournaledGroup.java

[3] Douglas G Altman, Berthold Lausen, Willi Sauerbrei, and Martin Schumacher. 1994. Dangers of using "optimal" cutpoints in the evaluation of prognostic factors. *JNCI: Journal of the National Cancer Institute* 86, 11 (1994), 829–835.

[4] Erik Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. 2010. Performance analysis of idle programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 739–753.

[5] Android. [n. d.]. Commit 465088ed in project android_platform_frameworks. Retrieved December 02, 2021 from https://android.googlesource.com/platform/frameworks/base/+/465088ed2f4591d08738b2306f213c5149b3484b

[6] Android. [n. d.]. ConnectivityService.java in project android_platform_frameworks. Retrieved December 02, 2021 from https://android.googlesource.com/platform/frameworks/base/+/465088ed2f4591d08738b2306f213c5149b3484b/services/core/java/com/android/server/ConnectivityService.java

[7] Aosp-mirror. [n. d.]. Commit 5abc71b2 in project platform_frameworks_base. Retrieved May 07, 2022 from https://github.com/aosp-mirror/platform_frameworks_base/commit/5abc71b27b5cd08148a277a8f378bbb5f4029835

[8] Aosp-mirror. [n. d.]. Commit aea1bdec in project platform_packages_apps_settings. Retrieved May 07, 2022 from https://github.com/aosp-mirror/platform_packages_apps_settings/commit/aea1bdec2d20753bbb64b53ac95b347395877493

[9] Apache. [n. d.]. BenchmarkThroughput.java in Hadoop. Retrieved December 02, 2021 from https://github.com/apache/hadoop/blob/3427bc1380ab4455a311c1848a83a966996bbc95/hadoop-hdfs-project/hadoop-hdfs/src/test/java/org/apache/hadoop/hdfs/BenchmarkThroughput.java

[10] Apache. [n. d.]. Issue 13514 in HDFS. Retrieved December 02, 2021 from https://issues.apache.org/jira/browse/HDFS-13514

[11] Apache. [n. d.]. JobImpl.java in project Hadoop. Retrieved December 02, 2021 from https://github.com/Jerry-Xin/hadoop/blob/master/hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-mapreduce-client-app/src/main/java/org/apache/hadoop/mapreduce/v2/app/job/impl/JobImpl.java

[12] Erik Arisholm, Lionel C Briand, and Eivind B Johannessen. 2010. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software* 83, 1 (2010), 2–17.

[13] Alberto Bacchelli, Marco D'Ambros, and Michele Lanza. 2010. Are popular classes more defect prone?. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 59–73.

[14] Suparna Bhattacharya, Mangala Gowri Nanda, Kanchi Gopinath, and Manish Gupta. 2011. Reuse, recycle to de-bloat software. In *European Conference on Object-Oriented Programming*. Springer, 408–432.

[15] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 2011. Don't touch my code! Examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 4–14.

[16] Bisq-network. [n. d.]. Commit 55b070f9 in project bisq. Retrieved May 07, 2022 from https://github.com/bisq-network/bisq/commit/55b070f9556977aa6ec4ebf878498a03b44f2f0

[17] Kendrick Boyd, Vitor Santos Costa, Jesse Davis, and C David Page. 2012. Unachievable region in precision-recall space and its effect on empirical evaluation. In *Proceedings of the... International Conference on Machine Learning. International Conference on Machine Learning*, Vol. 2012. NIH Public Access, 349.

[18] Jinfu Chen and Weiyi Shang. 2017. An exploratory study of performance regression introducing code changes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 341–352.

[19] Jinfu Chen, Weiyi Shang, and Emad Shihab. 2020. PerfJIT: Test-level just-in-time prediction for performance regression introducing commits. *IEEE Transactions on Software Engineering* (2020).

[20] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. 2014. Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proceedings of the 36th International Conference on Software Engineering*. 1001–1012.

[21] Yiqun Chen, Stefan Winter, and Neeraj Suri. 2019. Inferring Performance Bug Patterns from Developer Commits. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 70–81.

[22] Davide Chicco and Giuseppe Jurman. 2020. The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. *BMC genomics* 21, 1 (2020), 1–13.

[23] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. 2012. Input-sensitive profiling. *ACM SIGPLAN Notices* 47, 6 (2012), 89–98.

[24] Ting Dai, Daniel Dean, Peipei Wang, Xiaohui Gu, and Shan Lu. 2018. Hytrace: a hybrid approach to performance bug diagnosis in production cloud infrastructures. *IEEE Transactions on Parallel and Distributed Systems* 30, 1 (2018), 107–118.

[25] Ting Dai, Jingzhu He, Xiaohui Gu, Shan Lu, and Peipei Wang. 2018. Dscope: Detecting real-world data corruption hang bugs in cloud server systems. In *Proceedings of the ACM Symposium on Cloud Computing*. 313–325.

[1] https://github.com/NintyFive/MSR2024-Replication-Package

[26] 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)*. IEEE, 31–41.
[27] Augusto Born De Oliveira, Sebastian Fischmeister, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. 2017. Perphecy: performance regression test selection made simple but effective. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 103–113.
[28] Daniel J Dean, Hiep Nguyen, Xiaohui Gu, Hui Zhang, Junghwan Rhee, Nipun Arora, and Geoff Jiang. 2014. Perfscope: Practical online server performance bug inference in production cloud computing infrastructures. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–13.
[29] Daniel J Dean, Peipei Wang, Xiaohui Gu, William Enck, and Guoliang Jin. 2015. Automatic server hang bug diagnosis: Feasible reality or pipe dream?. In *2015 IEEE International Conference on Autonomic Computing*. IEEE, 127–132.
[30] Zishuo Ding, Jinfu Chen, and Weiyi Shang. 2020. Towards the Use of the Readily Available Tests from the Release Pipeline as Performance Tests. Are We There Yet?. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1435–1446.
[31] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2012. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering* 17, 4-5 (2012), 531–577.
[32] Bradley Efron. 1983. Estimating the error rate of a prediction rule: improvement on cross-validation. *Journal of the American statistical association* 78, 382 (1983), 316–331.
[33] Elastic. [n. d.]. Commit 6b51d85c in project Elasticsearch. Retrieved May 07, 2022 from https://github.com/elastic/elasticsearch/commit/6b51d85cbde8e0ddea020dcccb1e798dcb4ef27a
[34] Elastic. [n. d.]. Commit c5315744 in project elasticsearch. Retrieved December 02, 2021 from https://github.com/elastic/elasticsearch/commit/c531574407c5547fc742b168f61c03fd00b0c530
[35] Elastic. [n. d.]. PublicationTransportHandler.java in project elasticsearch. Retrieved December 02, 2021 from https://github.com/elastic/elasticsearch/blob/master/server/src/main/java/org/elasticsearch/cluster/coordination/PublicationTransportHandler.java
[36] ElasticSearch. [n. d.]. Elasticsearch Reference. Retrieved January 21, 2021 from https://www.elastic.co/
[37] Facebook. [n. d.]. Commit 2258ba13 in project buck. Retrieved May 07, 2022 from https://github.com/facebook/buck/pull/2553/commits/2258ba13d8c4ed7bce41975571609a05c3939bba
[38] Mohammadreza Ghanavati, Diego Costa, Janos Seboek, David Lo, and Artur Andrzejak. 2020. Memory and resource leak defects and their repairs in Java projects. *Empirical Software Engineering* 25, 1 (2020), 678–718.
[39] GitHub. [n. d.]. Archiving repositories on GitHub. Retrieved December 02, 2021 from https://docs.github.com/en/free-pro-team@latest/github/creating-cloning-and-archiving-repositories/about-archiving-repositories
[40] GitHut. [n. d.]. Top Active Programming Languages by GitHut. Retrieved December 02, 2021 from https://githut.info/
[41] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 345–355.
[42] Georgios Gousios and Diomidis Spinellis. 2012. GHTorrent: GitHub's data from a firehose. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 12–21.
[43] Georgios Gousios and Diomidis Spinellis. 2017. Mining software engineering data from GitHub. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 501–502.
[44] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. 2014. Lean GHTorrent: GitHub data on demand. In *Proceedings of the 11th working conference on mining software repositories*. 384–387.
[45] Mark Grechanik, Chen Fu, and Qing Xie. 2012. Automatically finding performance problems with feedback-directed learning software testing. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 156–166.
[46] Sumit Gulwani, Krishna K Mehra, and Trishul Chilimbi. 2009. SPEED: precise and efficient static estimation of program computational complexity. *ACM Sigplan Notices* 44, 1 (2009), 127–139.
[47] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. 2012. Performance debugging in the large via mining millions of stack traces. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 145–155.
[48] Hapifhir. [n. d.]. Commit c6777578 in project hapi-fhir. Retrieved May 07, 2022 from https://github.com/hapifhir/hapi-fhir/commit/c6777578a8b96eb5d9ea38bd280c40b8ca527e62
[49] Ahmed E Hassan. 2009. Predicting faults using the complexity of code changes. In *2009 IEEE 31st international conference on software engineering*. IEEE, 78–88.
[50] Steffen Herbold. 2017. Comments on ScottKnottESD in response to" An empirical comparison of model validation techniques for defect prediction models". *IEEE Transactions on Software Engineering* 43, 11 (2017), 1091–1094.
[51] IBM. [n. d.]. OpenLiberty Reference. Retrieved January 21, 2021 from https://openliberty.io/
[52] Jfree. [n. d.]. CandlestickRenderer.java in Jfreechart. Retrieved December 02, 2021 from https://github.com/jfree/jfreechart/blob/master/src/main/java/org/jfree/chart/renderer/xy/CandlestickRenderer.java
[53] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices* 47, 6 (2012), 77–88.
[54] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. 2011. Catch me if you can: performance bug detection in the wild. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. 155–170.
[55] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2014. The promises and perils of mining GitHub. In *Proceedings of the 11th working conference on mining software repositories*. ACM, 92–101.
[56] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W Anderson, and Ranjit Jhala. 2010. Finding latent performance bugs in systems implementations. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. 17–26.
[57] Christoph Laaber, Mikael Basmaci, and Pasquale Salza. 2021. Predicting unstable software benchmarks using static source code features. *Empirical Software Engineering* 26, 6 (2021), 1–53.
[58] Let's Dev Together (LDT). [n. d.]. AbstractBuilding.java in project minecolonies. Retrieved December 02, 2021 from https://github.com/ldtteam/minecolonies/blob/version/main/src/main/java/com/minecolonies/coremod/colony/buildings/AbstractBuilding.java
[59] Let's Dev Together (LDT). [n. d.]. Commit 508e7638 in project minecolonies. Retrieved December 02, 2021 from https://github.com/ldtteam/minecolonies/pull/6076/commits/508e763806951d74f4e8c2cafc87490d3a6d0ada
[60] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. 2008. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering* 34, 4 (2008), 485–496.
[61] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th international conference on software engineering*. 1013–1024.
[62] Mary L McHugh. 2012. Interrater reliability: the kappa statistic. *Biochemia medica* 22, 3 (2012), 276–282.
[63] Mekanism. [n. d.]. Commit ee83cb14 in project Mekanism. Retrieved May 07, 2022 from https://github.com/mekanism/Mekanism/commit/ee83cb142fc4c341750300a0e651cf79058a652b
[64] Tim Menzies, Jeremy Greenwald, and Art Frank. 2006. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering* 33, 1 (2006), 2–13.
[65] Domas Mituzas. 2009. Embarrassment. *Blog post: Embarrassment* (2009).
[66] I MOLYNEAUX. 2009. The Art of Application Performance Testing: Help for Programmers and Quality Assurance.
[67] Glen Emerson Morris. 2004. "Lessons from the Colorado benefits management system disaster. *Advertising and Marketing Review* (2004).
[68] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 902–912.
[69] Adrian Nistor, Tian Jiang, and Lin Tan. 2013. Discovering, reporting, and fixing performance bugs. In *2013 10th working conference on mining software repositories (MSR)*. IEEE, 237–246.
[70] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting performance problems via similar memory-access patterns. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 562–571.
[71] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Static detection of asymptotic performance bugs in collection traversals. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 369–378.
[72] Thorsten Pohlert. 2014. The pairwise multiple comparison of mean ranks package (PMCMR). *R package* 27, 2019 (2014), 9.
[73] Michael Pradel, Markus Huggler, and Thomas R Gross. 2014. Performance regression testing of concurrent classes. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 13–25.
[74] PYPL. [n. d.]. Popularity of Programming Language. Retrieved December 02, 2021 from https://pypl.github.io/PYPL.html
[75] Tim Richardson. 1901. census site still down after six months," 2002.
[76] Leif Singer and Kurt Schneider. 2012. It was a bit of a race: Gamification of version control. In *2012 Second International Workshop on Games and Software Engineering: Realizing User Engagement with Game Engineering Techniques (GAS)*. IEEE, 5–8.
[77] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *ACM sigsoft software engineering notes* 30, 4 (2005), 1–5.
[78] Helen R Sofaer, Jennifer A Hoeting, and Catherine S Jarnevich. 2019. The area under the precision-recall curve as a performance metric for rare binary events. *Methods in Ecology and Evolution* 10, 4 (2019), 565–577.

[79] Linhai Song and Shan Lu. 2017. Performance diagnosis for inefficient loops. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 370–380.
[80] Seyyed Ehsan Salamati Taba, Foutse Khomh, Ying Zou, Ahmed E Hassan, and Meiyappan Nagappan. 2013. Predicting bugs using antipatterns. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 270–279.
[81] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering* 43, 1 (2016), 1–18.
[82] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2018. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering* 45, 7 (2018), 683–711.
[83] Tiobe. [n. d.]. TIOBE Index For Popular Programming Languages. Retrieved December 02, 2021 from https://www.tiobe.com/tiobe-index/
[84] Apache Tomcat. [n. d.]. Apache Tomcat. Retrieved January 15, 2021 from https://tomcat.apache.org/
[85] Webfx-project. [n. d.]. Commit c241aa13 in project webfx. Retrieved May 07, 2022 from https://github.com/webfx-project/webfx/commit/c241aa134be0f744b213ab4380ff70a25dd25533
[86] Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. 2013. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. 90–100.
[87] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. 2009. Go with the flow: profiling copies to find runtime bloat. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 419–430.
[88] Guoqing Xu and Atanas Rountev. 2010. Detecting inefficiently-used containers to avoid bloat. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 160–173.
[89] Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. 2016. Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment? *Information and Software Technology* 74 (2016), 204–218.
[90] Shahed Zaman, Bram Adams, and Ahmed E Hassan. 2011. Security versus performance bugs: a case study on firefox. In *Proceedings of the 8th working conference on mining software repositories*. 93–102.
[91] Shahed Zaman, Bram Adams, and Ahmed E Hassan. 2012. A qualitative study on performance bugs. In *2012 9th IEEE working conference on mining software repositories (MSR)*. IEEE, 199–208.
[92] Dmitrijs Zaparanuks and Matthias Hauswirth. 2012. Algorithmic profiling. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 67–76.
[93] Chen Zhang, Jiaxin Li, Dongsheng Li, and Xicheng Lu. 2019. Understanding and Statically Detecting Synchronization Performance Bugs in Distributed Cloud Systems. *IEEE Access* 7 (2019), 99123–99135.
[94] Feng Zhang, Ahmed E Hassan, Shane McIntosh, and Ying Zou. 2016. The use of summation to aggregate software metrics hinders the performance of defect prediction models. *IEEE Transactions on Software Engineering* 43, 5 (2016), 476–491.
[95] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. 2014. Towards building a universal defect prediction model. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. 182–191.
[96] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. 2016. Towards building a universal defect prediction model with rank transformed predictors. *Empirical Software Engineering* 21, 5 (2016), 2107–2145.
[97] Feng Zhang, Quan Zheng, Ying Zou, and Ahmed E Hassan. 2016. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 309–320.
[98] Yutong Zhao, Lu Xiao, Xiao Wang, Lei Sun, Bihuan Chen, Yang Liu, and Andre B Bondi. 2020. How Are Performance Issues Caused and Resolved?-An Empirical Study from a Design Perspective. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. 181–192.
[99] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting defects for eclipse. In *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007)*. IEEE, 9–9.