Empirical Software Engineering (2023) 28:37
https://doi.org/10.1007/s10664-022-10261-z

Enhancing the defectiveness prediction of methods and classes via JIT

Davide Falessi[1] · Simone Mesiano Laureani[1] · Jonida Çarka[1] · Matteo Esposito[1] · Daniel Alencar da Costa[2]

Accepted: 7 November 2022 / Published online: 31 January 2023

Abstract

Context Defect prediction can help at prioritizing testing tasks by, for instance, ranking a list of items (methods and classes) according to their likelihood to be defective. While many studies investigated how to predict the defectiveness of commits, methods, or classes separately, no study investigated how these predictions differ or benefit each other. Specifically, at the end of a release, before the code is shipped to production, testing can be aided by ranking methods or classes, and we do not know which of the two approaches is more accurate. Moreover, every commit touches one or more methods in one or more classes; hence, the likelihood of a method and a class being defective can be associated with the likelihood of the touching commits being defective. Thus, it is reasonable to assume that the accuracy of methods-defectiveness-predictions (MDP) and the class-defectiveness-predictions (CDP) are increased by leveraging commits-defectiveness-predictions (aka JIT).

Objective The contribution of this paper is fourfold: (i) We compare methods and classes in terms of defectiveness and (ii) of accuracy in defectiveness prediction, (iii) we propose and evaluate a first and simple approach that leverages JIT to increase MDP accuracy and (iv) CDP accuracy.

Method We analyse accuracy using two types of metrics (threshold-independent and effort-aware). We also use feature selection metrics, nine machine learning defect prediction classifiers, more than 2.000 defects related to 38 releases of nine open source projects from the Apache ecosystem. Our results are based on a ground truth with a total of 285,139 data points and 46 features among commits, methods and classes.

Results Our results show that leveraging JIT by using a simple median approach increases the accuracy of MDP by an average of 17% AUC and 46% PofB10 while it increases the accuracy of CDP by an average of 31% AUC and 38% PofB20.

Communicated by: Yasutaka Kamei

✉ Davide Falessi
falessi@ing.uniroma2.it

Extended author information available on the last page of the article.

Conclusions From a practitioner's perspective, it is better to predict and rank defective methods than defective classes. From a researcher's perspective, there is a high potential for leveraging statement-defectiveness-prediction (SDP) to aid MDP and CDP.

Keywords Defect prediction · Accuracy metrics · Effort-aware metrics

1 Introduction

The manner in which defects are introduced into code, and the sheer volume of defects in software, are typically beyond the capability and resources of most development teams (Tantithamthavorn et al. 2016a; Nam et al. 2017; Kamei et al. 2012; Ghotra et al. 2017; Kondo et al. 2019). Defect prediction models aim to identify software artifacts that are likely to be defective (Ohlsson and Alberg 1996; Menzies et al. 2010; Turhan et al. 2009; Weyuker et al. 2010; Ostrand and Weyuker 2004; Ostrand et al. 2005). The main purpose of defect prediction is to reduce the cost of testing, analysis, and code reviews, by prioritizing developers' effort on specific artifacts such as commits, methods, or classes.

In this work, we study defect prediction that aims to support the testing phase, i.e., the predictions that are performed after the coding phase and before the deployment phase. During the testing phase, developers work to identify and eventually fix defects in the code before these defects can reach the deployment phase and, hence, become production defects. The focus of our paper is not on the *Just in Time* (JIT) defect predictions, which are usually performed *during* the development phase (Herbold 2019, 2020; McIntosh and Kamei 2018; Pascarella et al. 2019; Kondo et al. 2020; Huang et al. 2019; Fan et al. 2019; Tu et al. 2020; Rodriguezperez et al. 2020).

In the defect prediction area, researchers have proposed the use of different models to predict defects, leveraging, for example, product metrics (Basili et al. 1996; Gyimóthy et al. 2005; Khoshgoftaar et al. 1996; Nagappan and Ball 2005; Hassan 2009), process metrics (Moser et al. 2008), knowledge from where previous defects occurred (Ostrand et al. 2005; Kim et al. 2007), information about change-inducing fixes (Kim et al. 2008; Fukushima et al. 2014) and, recently, deep learning techniques to automatically engineer features from source code elements (Wang et al. 2016). While many studies investigated how to predict the defectiveness of commits (Herbold et al. 2019, 2020; McIntosh and Kamei 2018; Pascarella et al. 2019, 2020; Kondo et al. 2020; Huang et al. 2019; Fan et al. 2019; Tu et al. 2020; Rodriguezperez et al. 2020; Giger et al. 2012), or classes (Kamei et al. 2016; Tantithamthavorn et al. 2016b, 2019, 2020; Bennin et al. 2018, 2019; Herbold et al. 2017, 2018, 2019; Hosseini et al. 2019; Yan et al. 2017; Liu et al. 2017; Chi et al. 2017; Jing et al. 2017; Di Nucci et al. 2018; Palomba et al. 2019; Song et al. 2019; Zhang et al. 2016, 2017; Lee et al. 2016; Yu et al. 2019a; Peters et al. 2019; Qu et al. 2021; Shepperd et al. 2018; Amasaki 2020; Bangash et al. 2020; Kondo et al. 2019; Morasca and Lavazza 2020; Mori and Uchihira 2019; Tian et al. 2015; Jiarpakdee et al. 2020; Chen et al. 5555; Dalla Palma et al. 2021) independently, no study other than Pascarella et al. (2019), investigated how these predictions differ or can benefit each other. Specifically, every commit touches one or more methods in one or more classes; hence, the likelihood of a method, and of a class, to be defective depends on the likelihood of the touched commits being defective. Thus, it is reasonable to conjecture that method-defectiveness-predictions (MDP) and the class-defectiveness-predictions (CDP) may become more accurate if we leverage the commit-defectiveness-predictions (JIT). Furthermore, before code is shipped to production, developers may decide whether to test methods or classes. To the best of our knowledge, no study has investigated whether it is more useful for developers to rank methods or classes. Pascarella et al. (2019) reports a fine-grained just-in-time defect prediction technique that seems to enhance the performance of Kamei et al. (2012). We took inspiration from Pascarella et al. (2019) regarding the fact that they leverage CDP to refine JIT. However, we investigated the use of JIT to improve the accuracy of MDP and CDP. Moreover, compared to Pascarella et al. (2019), we use a more advanced SZZ algorithm (RA-SZZ), and we assume a different scenario: end of a release versus after a commit.

To better explain the purpose of our paper, we elaborate on an example. Let's consider a defect of project *OPENJPA*[1] that is documented and tracked via the *OPENJPA-2414* ticket, which was opened on *24/07/13* and closed on *29/10/13*.[2] The commit that fixes this defect has the ID *ee6f4acc3ff9ac43ea4e98579b478e55767aef24* and was committed on *23/08/2013* in release *2.3.0*. Figure 1 shows the log of this commit, which reports the defect ID. The commit containing the defect ID is defined as the fix-commit of the defect. Figure 1 also shows a fragment of the code statements touched (in green). The commit touched 1,494 LOC and removed 8 LOC over the following five classes:

- openjpa-jdbc/src/main/java/org/apache/openjpa/jdbc/kernel/FinderCacheImpl.java
- openjpa-kernel/src/main/java/org/apache/openjpa/kernel/DelegatingFetchConfiguration.java
- openjpa-kernel/src/main/java/org/apache/openjpa/kernel/FetchConfiguration.java,
- openjpa-kernel/src/main/java/org/apache/openjpa/kernel/FetchConfigurationImpl.java
- openjpa-persistence-jdbc/src/test/java/org/apache/openjpa/persistence/fetchgroups/TestFetchGroups.java

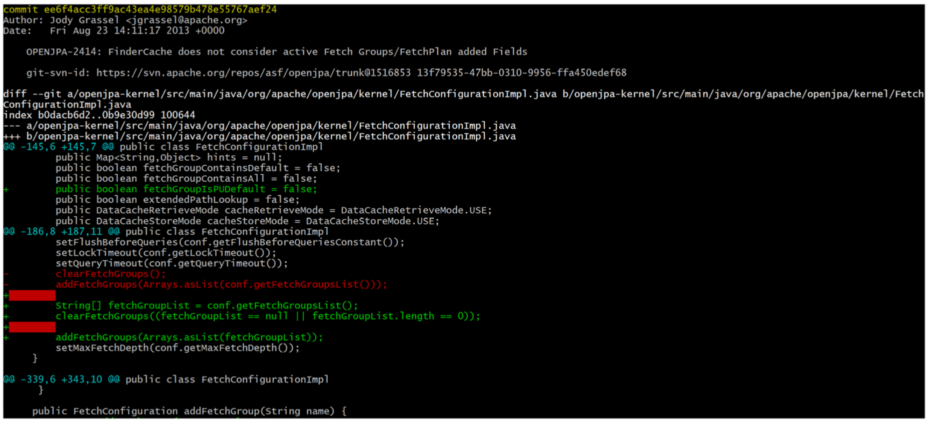
By applying RA-SZZ (Neto et al. 2019) to commit `ee6f4acc3ff9ac43ea4e98579b478e55767aef24`, we retrieve a list of commits that may have caused the defect (i.e., the defect-inducing commits). Specifically, RA-SZZ retrieved two defect-inducing commits, which touched 773 LOC and 139 classes. These two defect-inducing commits have the IDs `1fede626e2cad16f7bb4d77dd9fc3270a8b6b331` and `979d2340e93eaaa9f273a100dbe78e42ea9ed400`, respectively (both in release 0.9.0). However, not all statements in these two defect-inducing commits are defective. Specifically, only class `openjpa-kernel/src/main/java/org/apache/openjpa/kernel/FetchConfigurationImpl.java` is deemed defective among the classes retrieved by RA-SZZ; i.e, the two defect-inducing commits touched the same class. Afterwards, the class `openjpa-kernel/src/main/java/org/apache/openjpa/kernel/FetchConfigurationImpl.java` is labeled as defective from release `0.9.0 to release 2.3.0` (excluded). Since we have two defect-inducing commits that may have contributed to the defectiveness of class `openjpa-kernel/src/main/java/org/apache/openjpa/kernel/FetchConfigurationImpl.java`, we believe that information embedded in these commits can contribute to both class-defectiveness-predictions.

The contribution of our paper is fourfold:

1. We compare methods and classes in terms of defectiveness.
2. We compare the accuracy of MDP versus CDP in terms of effort-aware metrics, thus supporting the decision of whether to perform MDP or CDP.

[1] https://github.com/apache/openjpa

[2] https://issues.apache.org/jira/browse/OPENJPA-2414



```
commit ee6f4acc3ff9ac43ea4e98579b478e55767aef24
Author: Jody Grassel <jgrassel@apache.org>
Date:   Fri Aug 23 14:11:17 2013 +0000

    OPENJPA-2414: FinderCache does not consider active Fetch Groups/FetchPlan added Fields

    git-svn-id: https://svn.apache.org/repos/asf/openjpa/trunk@1516853 13f79535-47bb-0310-9956-ffa450edef68

diff --git a/openjpa-kernel/src/main/java/org/apache/openjpa/kernel/FetchConfigurationImpl.java b/openjpa-kernel/src/main/java/org/apache/openjpa/kernel/Fetch
ConfigurationImpl.java
index b0dacb6d2..0b9e30d99 100644
--- a/openjpa-kernel/src/main/java/org/apache/openjpa/kernel/FetchConfigurationImpl.java
+++ b/openjpa-kernel/src/main/java/org/apache/openjpa/kernel/FetchConfigurationImpl.java
@@ -145,6 +145,7 @@ public class FetchConfigurationImpl
         public Map<String,Object> hints = null;
         public boolean fetchGroupContainsDefault = false;
         public boolean fetchGroupContainsAll = false;
+        public boolean fetchGroupIsPUDefault = false;
         public boolean extendedPathLookup = false;
         public DataCacheRetrieveMode cacheRetrieveMode = DataCacheRetrieveMode.USE;
         public DataCacheStoreMode cacheStoreMode = DataCacheStoreMode.USE;
@@ -186,8 +187,11 @@ public class FetchConfigurationImpl
         setFlushBeforeQueries(conf.getFlushBeforeQueriesConstant());
         setLockTimeout(conf.getLockTimeout());
         setQueryTimeout(conf.getQueryTimeout());
-        clearFetchGroups();
-        addFetchGroups(Arrays.asList(conf.getFetchGroupsList()));
+
+        String[] fetchGroupList = conf.getFetchGroupsList();
+        clearFetchGroups((fetchGroupList == null || fetchGroupList.length == 0));
+
+        addFetchGroups(Arrays.asList(fetchGroupList));
         setMaxFetchDepth(conf.getMaxFetchDepth());
     }

@@ -339,6 +343,10 @@ public class FetchConfigurationImpl
      }

     public FetchConfiguration addFetchGroup(String name) {
```

Fig. 1 Example of fix commit

3. We propose and evaluate the increase in MDP accuracy by leveraging JIT information.
4. We propose and evaluate the increase in CDP accuracy by leveraging JIT information.

Our analysis features two types of accuracy metrics (threshold-independent and effort-aware), nine machine learning defect prediction classifiers, 1,860 defects related to 35 releases of 9 open source projects from the Apache ecosystem. MDP, CDP, and JIT predictions are performed using state-of-the-art approaches and rely on a total of 15 features from commits, 15 features from methods, and 16 features from classes. Our results rely on a ground truth featuring a total of 269,004 data points.

Our results reveal that MDP is substantially more accurate than CDP (+5% AUC and +62% PofB10). These results indicate that it is more rewarding to predict and rank defective methods instead of defective classes. Moreover, leveraging JIT information by using a simple median approach increases the accuracy of MDP by an average of 18% AUC and 49% PofB10 while increasing the accuracy of CDP by an average of 28% AUC and 31% PofB20.

The remainder of this paper is structured as follows: Section 2 reports the design of our research questions. Section 3 describe the results of our study. Section 4 provides a discussion of our results. Section 5 reports on the threats to the study validity. Section 6 discusses the related literature, focusing in particular on MDP and CDP. Finally, Section 7 concludes the paper and outlines directions for future work.

2 Study Design

In this section, we explain the design of our study, which includes the procedures to choose our subject projects, our research questions (RQs) and our approaches to answer our RQs.

2.1 Subject Projects

We describe in the following how we chose the 10 datasets we used in this study.

1. We first retrieved the JIRA and Git URL of all existing Apache projects.[3] We focused on Apache projects instead of GitHub projects because Apache projects have a higher quality of defect annotation and are unlikely to be toy datasets (Munaiah et al. 2017).
2. We filtered out projects that are not tracked in JIRA or not versioned in Git.
3. We filtered out projects that are small and therefore not representative of a medium size industrial project. Specifically, we filtered out projects having: less than 20 releases, less than 20 commits per release, less than 20 linked and fixed defects per release, less than 50% of commits in the Java programming language. We define a commit in Java as a commit touching at least one Java file.
4. We filtered out two projects that although hosted on GitHub, their historical data did not cover a significant portion of their life time (which were actually covered in JIRA). These two projects were Openmeetings and Camel.
5. We selected the nine projects having the highest linkage rate, i.e., the highest proportion of defects with linked commits. A defect is considered to be linked if such a defect can be associated with at least one commit from the source code commit log. We selected nine projects due to the manual effort constraints required to perform the study.

Table 1 reports the details of the our 9 selected projects in terms of commits, percent of defective commits, methods, percent of defective methods, classes, percent of defective classes, defects, and releases.

In the reminder of this section we report the design of each research question.

2.2 RQ1: Do Methods and Classes vary in Defectiveness?

Many studies evaluated techniques to perform MDP and CDP. However, there is a lack of studies that have compared the defectiveness across different granular levels (e.g., methods vs classes). Studying the defectiveness of methods and classes on the same releases can help us understand whether the defectiveness at a certain granularity is harder or easier to predict compared to other granularities. Specifically, despite we know that, by definition, defective classes are more frequent than defective methods, we do not know if defective classes are statistically more frequent than defective methods. In this study, we compare methods and classes in terms of defectiveness.

In this research question, we test the following null hypotheses:

- H_{10}: *the number of defective methods is equivalent to the number of defective classes.*
- H_{20}: *the proportion of defective methods is equivalent to the proportion of defective classes*

Independent Variables The independent variable is the type of defective entity, either methods or classes.

Dependent Variables The dependent variables are the number and the proportion of detective entities. The proportion of defective entities of a type is computed as the number of defective entities of that type divided by the total number of entities of that type.

An Extended RA-SZZ In order to label commits, methods and classes as defective we performed the following steps, as detailed in Fig. 2:

[3] https://people.apache.org/phonebook.html

Table 1 Details of the datasets used in terms of commits, percent of defective commits, methods, percent of defective methods, classes, percent of defective classes, defects, linkage, releases, and manually validated commits

Dataset	CMT	D. CMT	Meth	D. Meth	CLS	D. CLS	Defects	Linkage	Releases	V. CMT
ARTEMIS	589	34%	83880	0.14%	9425	0.41%	133	82%	1.0.0;1.0.1;1.2.0	21
DIRSERVER	367	6%	19335	0.11%	3367	1.85%	14	83%	1.01;1.02;1.03;1.04	10
GROOVY	701	2%	14310	0.12%	1496	1.54%	20	77%	1.0-1;1.0-2;1.0-3;1.0-4;1.0-5	12
MNG	1835	39%	14519	2.33%	2778	8.8%	582	51%	2.0a1;2.0a2;2.0a3;2.0-1;2.0-2	24
NUTCH	223	65%	6934	4.86%	1078	13.22%	135	70%	0.7;0.7.1;0.7.2	21
OPENJPA	538	54%	44997	0.48%	3119	2.65%	263	92%	0.9.0;0.9.6;0.9.7	23
QPID	1443	44%	43049	3.42%	5423	12.36%	617	82%	M1;M2;M2.1;M3;M4	24
TIKA	246	29%	1844	0.64%	369	6.32%	69	75%	0.1-incubating;0.2;0.3	19
ZOOKEEPER	174	21%	6249	0.49%	838	7.01%	27	76%	3.0.0;3.0.1;3.1.0;3.1.1	13

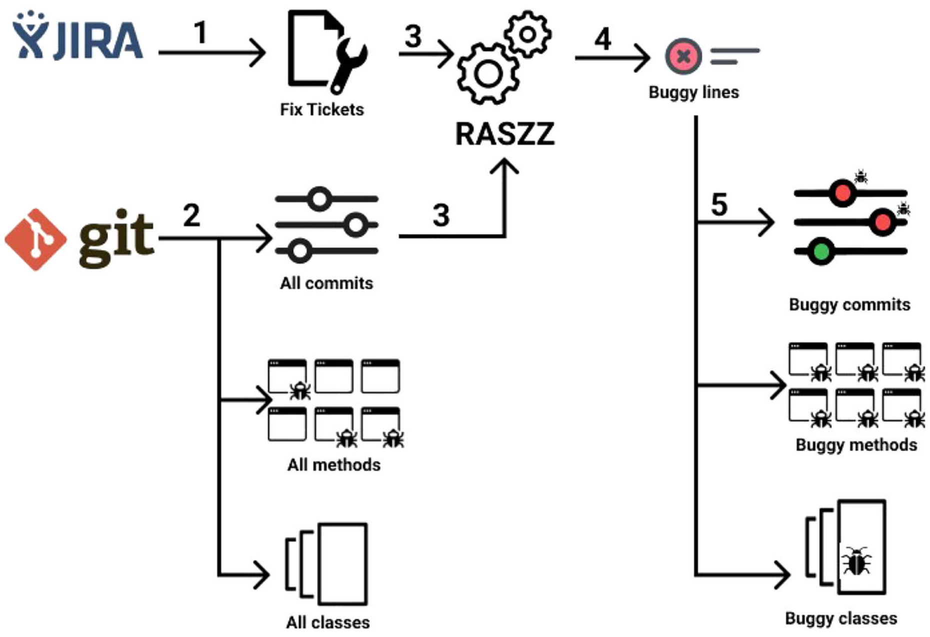


Fig. 2 Steps of RQ1

1. Defect identification: For each project, we find defect tickets on JIRA where type == "defect" AND (status == "Closed" OR status =="Resolved") AND Resolution =="Fixed".
2. Labelling statements: We run RA-SZZ, which provides us with the defect-introducing statements.
3. Labelling commits: From the defect introducing statements we retrieved their defect-introducing commits and, hence, the defect-introducing releases.
4. Labelling methods: To identify which methods have been impacted by the defect-introducing statements (as retrieved by RA-SZZ), we developed a tool that uses the javaparser library[4] to parse the Java files implicated by the defect-introducing statements. Once the java files are parsed, the tool becomes aware of the *start* and *end* statements of each method within the Java files. Then, our tool identifies the methods that contain the defect-introducing statements that were previously retrieved. In doing so, we are able to identify the defective methods of a set of Java classes. Thus, in this study, we use an extended release of RA-SZZ, which provides a ground truth at the method granularity (in addition to the code statement and commit granularities).
5. Labelling classes: We labelled classes as defective if they contained at least one defective method.

Note that, in this study, an element at any granularity (i.e., a class, a method, or a statement) is defined as defective in all releases containing a defective statement.

[4]https://javaparser.org/

Manual Validation We have manually validated the defect-inducing commits used in this work. We used a 95% confidence level and a 20% margin of error; the resulting number of manually analyzed commits per project is reported in column 11 of Table 1. The third and fourth authors performed the manual validation of defect-inducing commits independently; a discussion resolved the disagreements between the two authors. The two authors resulted in agreement 95% of the cases (Kappa equals to 0.69) and the tool resulted in an agreement with the two authors 93% of the cases. The replication package reports the following information for each validated commit: bug ID, defect-fixing commit SHA, defect-inducing commit SHA, the label of the third author, the label of the fourth author, and the tool label.

Hypothesis Testing To test our hypotheses, we used the paired Wilcoxon signed-rank test (Rey and Neuhäuser 2011), which is a non-parametric test (i.e., there are no assumptions regarding the underlying distribution) to check whether two paired distributions are significantly different. We chose the non-parametric Kruskal–Wallis because our metrics (e.g., number and proportion of defective entities) do not follow a normal distribution (as we noted when performing Shapiro–Wilk tests (Shapiro and Wilk 1965)). Therefore, our approach is compliant to the suggestion to avoid using ScottKnottESD in case of non-normal distributions (Herbold 2017). We use standard value of alpha $\alpha = 0.05$. To account for the chance of errors due to multiple comparisons, in all research questions we perform a Holm-Bonferroni correction of our $p - values$ (Holm 1979). We performed effect size analysis by using Cohen's d (Herbold 2017) which shall be interpreted as in Table 2.

2.3 RQ2: Does Leveraging JIT Information Increase the Accuracy of MDP?

Every commit touches one or more methods; hence, the likelihood that a method is defective depends on the likelihood that the touched commits are defective. While many studies investigated techniques for MDP, no previous study investigated whether MDP may become more accurate if JIT information is used. In this paper, we propose and evaluate techniques to increase MDP accuracy by leveraging JIT.

In this research question, we test the following null hypothesis:

- H_{30}: *leveraging JIT does not improve the accuracy of MDP.*

Independent Variables The independent variable is MDP with JIT information versus without JIT information. Specifically, we investigate the following MDP approaches:

- **Single**: It uses state of the art approach for MDP (Giger et al. 2012; Pascarella et al. 2020). Specifically, we used the following set of features as input to a machine learning classifier:
    - size: LOC of a method.

Table 2 Intepretation of Cohen's d

Effect size	d
Very small	< 0.01
Small	< 0.20
Medium	< 0.50
Large	< 0.80
Very Large	≥ 0.80

  - methodHistories: number of times a method was changed.
  - authors: number of distinct authors that changed a method.
  - stmtAdded: sum of all source code statements added to a method body over all method histories.
  - maxStmtAdded: maximum number of source code statements added to a method body throughout the method's change history.
  - avgStmtAdded: average number of source code statements added to a method body per change to the method.
  - stmtDeleted: sum of all source code statements deleted from a method body over all method histories.
  - maxStmtDeleted: maximum number of source code statements deleted from a method body for all method histories.
  - avgStmtDeleted: Average number of source code statements deleted from a method body per method history.
  - churn: sum of stmtAdded plus stmtDeleted over all method histories.
  - maxChurn: maximum churn for all method histories.
  - avgChurn: average churn per method history
  - cond: number of condition expression changes in a method body over all revisions.
  - elseAdded: number of added else-parts in a method body over all revisions.
  - elseDeleted: number of deleted else-parts from a method body over all revisions.

- **Combined:** It takes the **median** to combine the previously mentioned Single approach with two other scores:

  - SumC is the sum of defectiveness probabilities of the commits touching the method. The rationale is that the probability that a method is defective is related to the sum of probabilities of the commits touching the method.
  - MaxC is the max of defectiveness probabilities of the commits touching the method. The rationale is that the probability that a method is defective is related to the max of probabilities of the commits touching the method.

  The rationale of the Combined approach is that a defective commit incurs defective methods (i.e., those methods that are touched by the commit). We use the median as the combination mechanism because it is a simple way to combine several probabilities.

  We use a standard JIT approach (Kamei et al. 2012) to obtain probabilities of defectiveness of the commits that touch the methods. Specifically, we used the following set of features as input to a machine learning classifier:

  - Size: lines of code modified.
  - Number of modified subsystems (NS): changes modifying many subsystem are more likely to be defect-prone.
  - Number of modified directories (ND): changes that modify many directories are more likely to be defect-prone.
  - Number of modified files (NF): changes touching many files are more likely to be defect prone.
  - Distribution of modified code across each file (Entropy): changes with high entropy are more likely to be defect-prone, because a developer will have to recall and track higher numbers of scattered changes across each file.

- Lines of code added (LA): the more lines of code added the more likely a defect is introduced.
- Lines of code deleted (LD): the more lines of code deleted the higher the chance of a defect to occur.
- Lines of code in a file before the change (LT): the larger a file the more likely a change might introduce a defect.
- Whether or not the change is a defect fix (FIX): fixing a defect means that an error was made in an earlier implementation, therefore it may indicate an area where errors are more likely.
- Number of developers that changed the modified files (NDEV): the larger the NDEV, the more likely a defect is introduced because files revised by many developers often contain different thoughts and coding styles.
- Average time interval between the last and the current change (AGE): the lower the AGE, the more likely a defect will be introduced.
- Number of unique changes to the modified files (NUC): the larger the NUC, the more likely a defect is introduced, because a developer will have to recall and track many previous changes.
- Developer experience (EXP): more experienced developers are less likely to introduce a defect.
- Recent developer experience (REXP): developers that have often modified the files in recent months are less likely to introduce a defect because they will be more familiar with the recent developments in the system.
- Developer experience on a subsystem (SEXP): developers that are familiar with the subsystems modified by a change are less likely to introduce a defect.

Dependent Variables The main dependent variable is the accuracy of MDP. As performance indicators of defect prediction we used the following metrics:

- AUC: Area Under the Receiving Operating Characteristic Curve (Powers 2007) is the area under the curve of true positives rate versus false positive rate, which is defined by setting multiple thresholds. A positive instance is a defective entity, whereas a negative instance is a defect-free entity. AUC has the advantage to be threshold independent and, therefore, it is recommended for evaluating defect prediction techniques (Lessmann et al. 2008). We decided to avoid metrics such as Precision, Recall and F1, since they are threshold dependent.
- PofBX: as the effort-aware metric, we used PofBx (Chen et al. 2017; Wang et al. 2020; Xia et al. 2016; Tu et al. 2020). PofBx is defined as the proportion of defective entities identified by analyzing the first x% of the code base. For instance, a PofB10 of 30% signifies that 30% of defective entities have been found by analyzing 10% of the code base. We explored PofBx with an x in the range of [10, 50]. While previous studies only focused on $x = 20$, we investigated a wider range to obtain more informative results. Note that PofB is different than Popt (Kamei et al. 2010, 2013; Mende and Koschke 2010) in two aspects: normalization and range of x. Regarding normalization, while Popt normalizes the value according to a random approach, PofB does not perform such analysis, which aligns with our goals for two reasons:

  1) The comparison against a random approach is already provided by AUC, since an AUC higher than 0.5 indicates that a classifier performed better than a random classifier,

2) In our study, we are interested in comparing classifiers that rank entities at different levels of granularity. Specifically, since methods and classes have a different defectiveness proportion (see Table 1), a random ranking would perform differently across methods and classes. Regarding the value of x, Popt represents an average of the complete spectrum of x, but we decided to neglect high values of x, as we believe that high values of x would be unrealistic for practitioners when indicating which code should be inspected during testing. Specifically, the lower the amount of code tested, the higher the impact of the ranking approach; i.e., if 100% of the code needs to inspected the ranking approach is effectively useless. Thus, we envisioned a metric that express the return of investing a specific amount of time in testing $x\%$ of the code as suggested by the ranking from a classifier. For all these reasons, PofB is a better match to our needs than Popt.

As an additional dependent variable, we measured the proportion of times features related to JIT are chosen to predict MDP. We performed this feature selection to complement the accuracy analysis, since it is important to know whether our approach of using the median to leverage JIT information is beneficial. For example, if JIT information is selected as features for MDP but do not result in improving the accuracy of MDP, it indicates that using the median may not be the right approach to leverage JIT information.

Measurement Procedure In this section, we describe the steps we performed to compute the accuracy metrics related to RQ2. As described in Fig. 3, for each project, we:

1. Compute for each commit the above mentioned features.
2. Label commits as defective or not by using RA-SZZ (as described in RQ1)
3. Create a commit dataset by combining features and defectiveness labels
4. Perform preprocessing:
   - Normalization: we normalize the data of all features with log10 since performed in many similar studies (Jiang et al. 2008; Tantithamthavorn et al. 2019),
   - Feature Selection: we filter the independent variables described above by using the correlation-based feature subset selection (Hall 1998; Ghotra et al. 2017; Kondo et al. 2019). The approach evaluates the worth of a subset of features by considering the individual predictive capability of each feature, as well as the degree of redundancy between different features. The approach searches the space of feature subsets by a greedy hill-climbing augmented with a backtracking facility. The approach starts with an empty set of features and performs a forward search by considering all possible single feature additions and deletions at a given point.
   - Balancing: we apply SMOTE (Chawla et al. 2002; Agrawal and Menzies 2018) so that each dataset is perfectly balanced. As suggested (Witten et al. 2011), we apply feature selection and balancing to the training set only.
5. Create, Train, and Test commit datasets by splitting the preprocessed dataset into about 66% of releases as the training set and about 33% of releases as the testing set, while preserving the order of data (Falessi et al. 2020). We chose this split proportion since suggested in ML (Witten et al. 2011) and because it resemble the split proportion of the bootstrap approach (Falessi et al. 2020). Note that since the split is

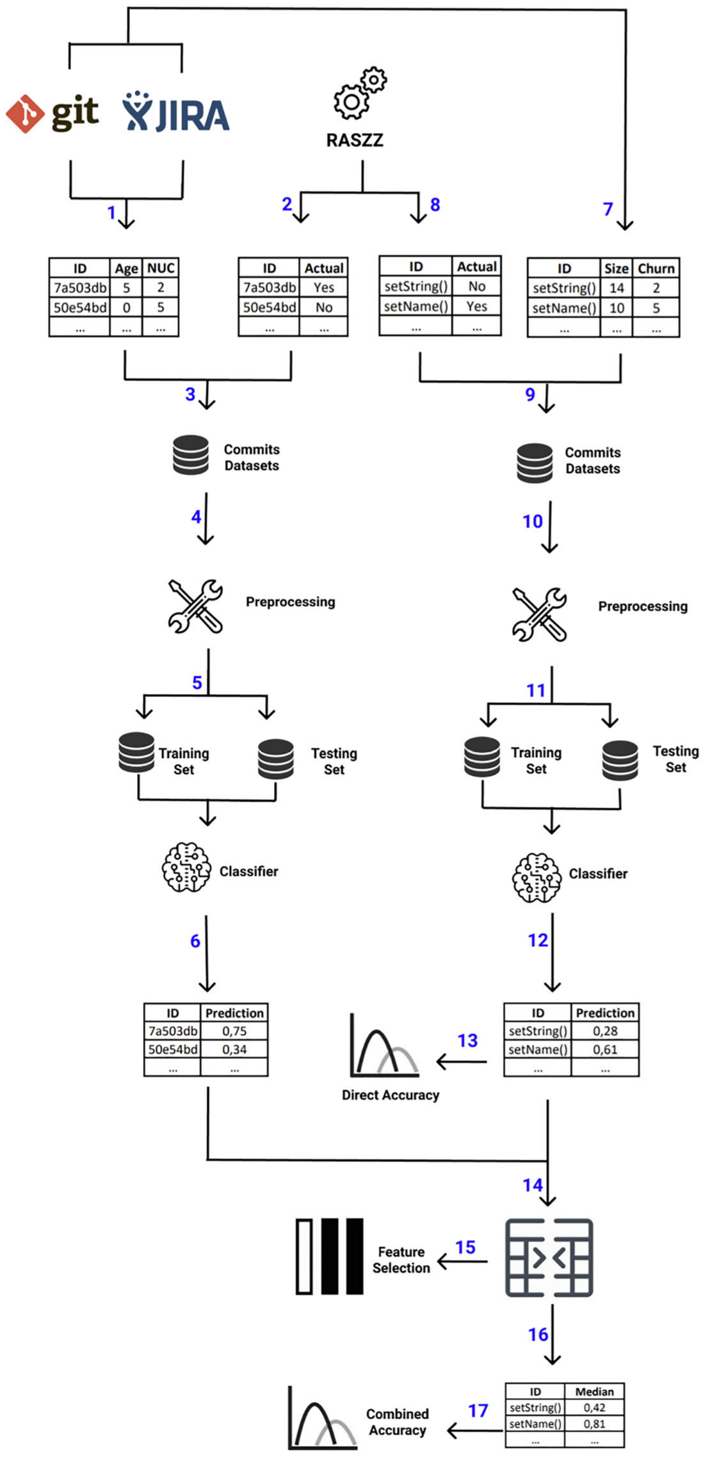
at the level of the release, and since different releases have a different number of commits, methods and classes, the specific proportion of spilt in training set and testing set vary across datasets and types of entities. Commits are assigned to the different releases given their time-stamp. The status of methods and classes in a release takes into account the commits of that release.

6. Compute predicted probability of defectiveness for each commit by using each of the 9 classifiers.
7. Compute the above mentioned features for each method.
8. Label methods as defective or not by using RA-SZZ (as described in RQ1).
9. Create a dataset of methods by combining features and defectiveness labels.
10. Perform preprocessing.
11. Create, Train, and Test method datasets by splitting the preprocessed dataset into 66% as the training set and 33% as the testing set, while preserving the order of data (Falessi et al. 2020).
12. Compute *Direct* predicted probability of defectiveness for each method by using each of the 9 classifiers.
13. Compute accuracy metrics of Direct.
14. Find commits related to each method. For the set of commits related to a method compute MaxC and SumC given the above commits predictions performed.
15. Perform feature selection, with the same technique described in item 4 (above), of the following features: Direct, MaxC and SumC.
16. Compute *Combined* by computing the median between Direct, MaxC and SumC.
17. Compute accuracy metrics of Combined.

In Fig. 3, the dataset is split twice and some actions are repeated twice because one flow is for the data about commits and another is for data about the entity we want to predict (i.e., methods and classes).

In this paper, we used the following set of classifiers, since they have been successfully adopted in a previous study (Falessi et al. 2020):

– Random Forest: It generates a number of separate, randomized decision trees and provides as classification the mode of the classifications. It has proven to be highly accurate and robust against noise (Breiman 2001). However, it can be computationally expensive as it requires the building of several trees.
– Logistic Regression: It estimates the probabilities of the different possible outcomes of a categorically distributed dependent variable, given a set of independent variables. The estimation is performed through the logistic distribution function (Cessie and Houwelingen 1992).
– Naïve Bayes: It uses the Bayes theorem, i.e., it assumes that the contribution of an individual feature towards deciding the probability of a particular class is independent of other features in that dataset instance (Mccallum and Nigam 2001).
– HyperPipes: It simply constructs a hyper-rectangle for each label that records the bounds for each numeric feature and what values occur for nominal features. During the classifier application, the label is chosen by whose hyper-rectangle most contains the instance (i.e., that which has the highest number of feature values of the test instance fall within the corresponding bounds of the hyper-rectangle) .
– IBK: Also known as the k-nearest neighbors' algorithm (k-NN) which is a non-parametric method. The classification is based on the majority vote of its neighbors,



ID	Age	NUC
7a503db	5	2
50e54bd	0	5
...	...	...

ID	Actual
7a503db	Yes
50e54bd	No
...	...

ID	Actual
setString()	No
setName()	Yes
...	...

ID	Size	Churn
setString()	14	2
setName()	10	5
...	...	...

ID	Prediction
7a503db	0,75
50e54bd	0,34
...	...

ID	Prediction
setString()	0,28
setName()	0,61
...	...

ID	Median
setString()	0,42
setName()	0,81
...	...

Fig. 3 Measurement procedure of RQ2

with the object being assigned to the class most common among its k nearest neighbors (Altman 1992).
- IB1: It is a special case of IBK with K = 1, i.e., it uses the closest neighbor (Altman 1992).
- J48: Builds decision trees from a set of training data. It extends the Iterative Dichotomiser 3 classifier (Quinlan 1986) by accounting for missing values, decision trees pruning, continuous feature value ranges and the derivation of rules.
- VFI: Also known as voting feature intervals (Demiröz and Güvenir 1997). A set of feature intervals represents a concept on each feature dimension separately. Afterwards, each feature is used by distributing votes among classes. The predicted class is the class receiving the highest vote (Demiröz and Güvenir 1997).
- Voted Perceptron: It uses a new perceptron every time an example is wrongly classified, initializing the weights vector with the final weights of the last perceptron. Each perceptron will also be given another weight corresponding to how many examples they correctly classify before wrongly classifying one, and at the end, the output will be a weighted vote on all perceptrons (Freund and Schapire 1999).

Hypothesis Testing As in RQ1, we use the paired Wilcoxon signed-rank test (Rey and Neuhäuser 2011) to test our hypothesis, H_{30}.

2.4 RQ3: Does Leveraging JIT Information increase the Accuracy of CDP?

Similar to RQ2, as no previous work investigated whether CDP may become more accurate by using JIT information, we propose and evaluate a technique to increase CDP accuracy by leveraging JIT information.

In this research question, we test the following null hypothesis:

- H_{40}: *leveraging JIT does not improve the accuracy of CDP.*

Independent Variables The independent variable is the use of JIT information in CDP. As in RQ2, we investigate the following MDP approaches:

- **Single**: It uses state of the art approach for CDP (Falessi et al. 2021). Specifically, we used the following set of features as input to a machine learning classifier:
    - Size (LOC): lines of code.
    - LOC Touched: sum over revisions of LOC added and deleted
    - NR: number of revisions.
    - Nfix: number of defect fixes.
    - Nauth: number of authors.
    - LOC Added: sum over revisions of LOC added and deleted.
    - Max LOC Added: maximum over revisions of LOC added.
    - Average LOC Added: average LOC added per revision.
    - Churn: sum over revisions of added deleted LOC.
    - Max Churn: maximum churn over revisions.
    - Average Churn: average churn over revisions.
    - Change Set Size: number of files committed together.
    - Max Change Set: maximum change set size over revisions.
    - Average Change Set: average change set size over revisions.
    - Age: age of release.

  - Weighted Age: age of release weighted by LOC touched.
- **Combined:** Similar to RQ2, it takes the median to combine the previously described Direct approach with JIT information.

Dependent Variables As in RQ2, the main dependent variable is the accuracy of CDP. Again, as an additional dependent variable, we measured the proportion of times features related to JIT are chosen to predict CDP.

Measurement Procedure In this section, we describe the steps we performed to compute the accuracy metrics related to RQ3. We performed the same exact set of steps of RQ2 with the only difference being that we used the above mentioned features related to classes rather than features related to methods. Therefore, in RQ3, *median* is computed among the direct probability of a class to be defective, MaxC and SumC.

Hypothesis Testing As in RQ1 and RQ2, we use the paired Wilcoxon signed-rank test (Rey and Neuhäuser 2011) to test our hypotheses, H_{40}.

2.5 RQ4: Are we more Accurate in MDP or CDP?

When using defect predictions, developers may choose to inspect methods or classes during testing (i.e., before code is shipped to production). However, no previous study has investigated which prediction granularity is more advantageous (methods or classes?) in terms of accuracy and effort. In this study, we compare the accuracy of MDP against CDP, using also effort aware metrics.

In this research question, we test the following null hypotheses:

- H_{50}: *MDP is as accurate as CDP.*

Independent Variables The independent variable is the granularity of the entity that is subject to the defectiveness prediction. The independent variable can have the following values: methods or classes.

Dependent Variables The dependent variable is the accuracy of MDP and CDP as measured by the same performance metrics we used in RQ2 and RQ3.

Measurement Procedure We used the data already used in RQ2 and RQ3 related to the Combined approach.

Hypothesis Testing As in our previous three research questions, we use the paired Wilcoxon signed-rank test (Rey and Neuhäuser 2011) to test our hypotheses, H_{50}.

2.6 Replication Package

For the interested researchers, the present study can be replicated using the replication package available online.[5] The replication package provides the scripts used to measure the data and the data itself. Both scripts and data are organized by research questions.

[5]https://doi.org/10.5281/zenodo.7213412

3 Study Results

3.1 RQ1: Do Methods and Classes vary in Defectiveness?

Figure 4 reports the number of defective entities (x-axis) in specific projects (x-axis) across different granularities of entities (color). According to Fig. 4 there is no entity that is more defective than another in all projects. Moreover, the number of defective entities varies across projects.

Figure 5 reports the proportion of defective entities (x-axis) in specific datasets (x-axis) across different types of entities (color). According to Fig. 5: in all nine projects, the proportion of defective classes is higher than the proportion of defective methods. Therefore, it is more likely to find by chance a defective class than a defective method. Moreover, the number of defective entities varies across datasets.

Table 3 reports the statistical results (p-value) comparing the number of defective classes versus the number of defective methods. According to Table 3, there is a statistical difference between the number of defective methods and the number of defective classes; therefore, we can reject H_{10}. Moreover, the effect size is large.

Table 4 reports the statistical results (p-value) comparing the proportion of defective methods against the proportion of defective classes. According to Table 4, methods are statistically less frequently defective than classes. Therefore, we can reject H_{20}. Moreover, the effect size is medium.

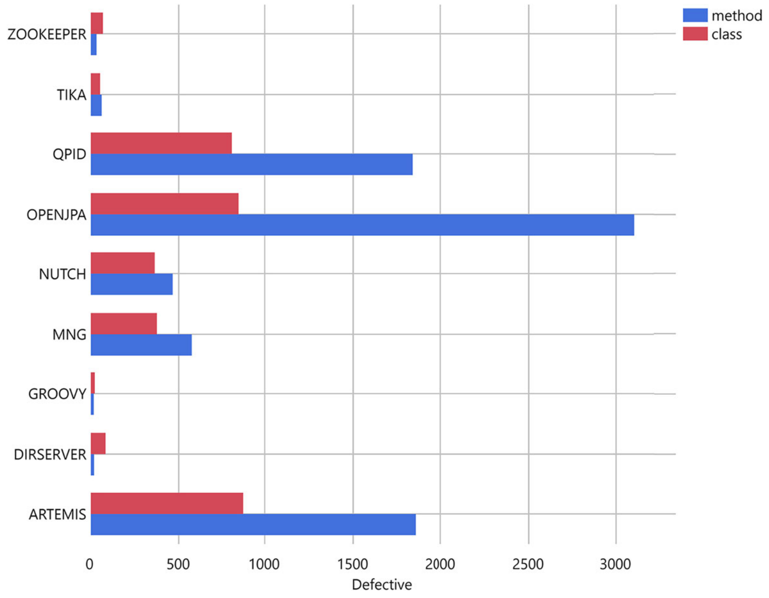


Fig. 4 Number of defective entities (x-axis) in specific projects (y-axis) across different granularities of entities (color)

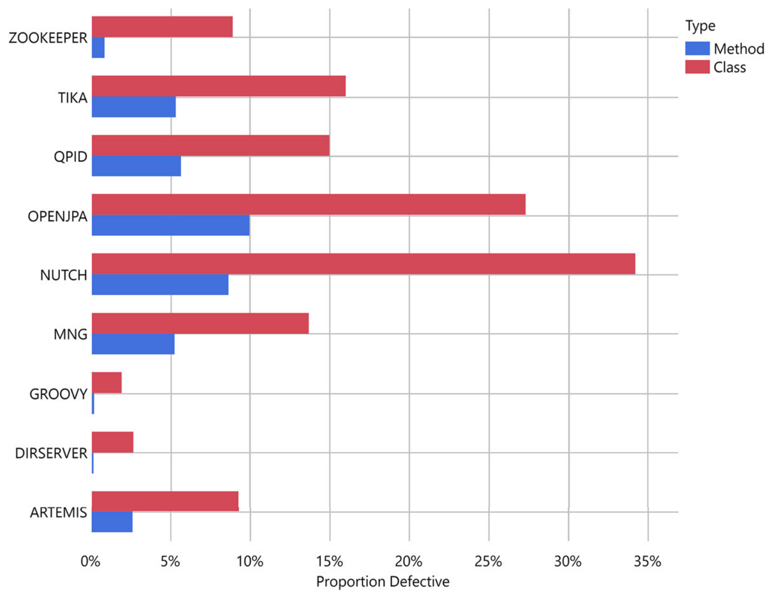


Fig. 5 Proportion of defective entities (x-axis) in specific datasets (x-axis) across different types of entities (color)

3.2 RQ2: Does Leveraging JIT Information Increase the Accuracy of MDP?

Accuracy Figure 6 reports the distribution of AUC values across classifiers (y-axis) achieved by our proposed approaches (i.e., Combined and Direct) for MDP in the subject projects (x-axis). According to Fig. 6, the Combined approach is more accurate than Direct in all projects, except DRSERVER. Moreover, the distribution of AUC values across classifiers of Combined is narrower than the distribution of Direct.

Figure 7 reports the distribution, across classifiers, of PofB values (x-axis) achieved by the Combined and Direct approaches (colors) for MDP in our subject projects (quadrant). According to Fig. 7:

- Combined is better than Direct in all PofBs in Groovy, MNG, NUTCH, OPENJPA, QPID and TIKA.
- Combined is worse than Direct in all PofBs in DERSERVER.
- Similar to the previous results related to the AUC, the distribution of values from Combined is substantially more narrow than the distribution of values from Direct. This result indicates that the choice of classifiers is not as important when using Combined.

Figure 8 reports the mean of the relative gain in MDP by leveraging JIT across classifiers and projects. According to Fig. 8: leveraging JIT increases the accuracy of MDP by an

Table 3 Statistical result (p-value) and Cohen's d effect size, comparing the number of defective methods and the number of defective classes in our projects

Entity	Pvalue	Cohen's d
Class Vs Method	0.0488	0.598

Table 4 Statistical result (p-value) and Cohen's d effect size, comparing the proportion of defective methods against the proportion of defective classes

Entity	Pvalue	Cohen's d
Class Vs Method	0.0488	0.359

average of 17% in AUC and 46% in PofB10. It is interesting to note that the relative gain is inversely correlated with PofB; this is due to the fact that the margin of performance, and hence the relative gain, is reduced when considering a larger code base.

Feature Selection Figure 9 reports the percent (x-axis) of times, across the nine classifiers, that a given feature (color) has been chosen in a given project (y-axis). According to Fig. 9:

- MaxC or SumC have been selected in all projects, except Zookeeper.
- Direct has been selected in seven out of the nine projects.
- MaxC or SumC have been selected more than Direct in five out of the nine projects.

Table 5 reports the statistical results (p-value) comparing the MDP accuracy of the combined versus direct approach. An asterisk identifies cases where the pvalue is lower than alpha according to the Holm-Bonferroni correction and hence we can reject the null hypothesis. We note that MDP is statistically more accurate by leveraging JIT in AUC and in seven out of nine PofB. Therefore, we can reject H_{30} and claim that leveraging JIT statistically and significantly improves the accuracy of MDP. Moreover, the effect size is at least medium in most of the metrics.

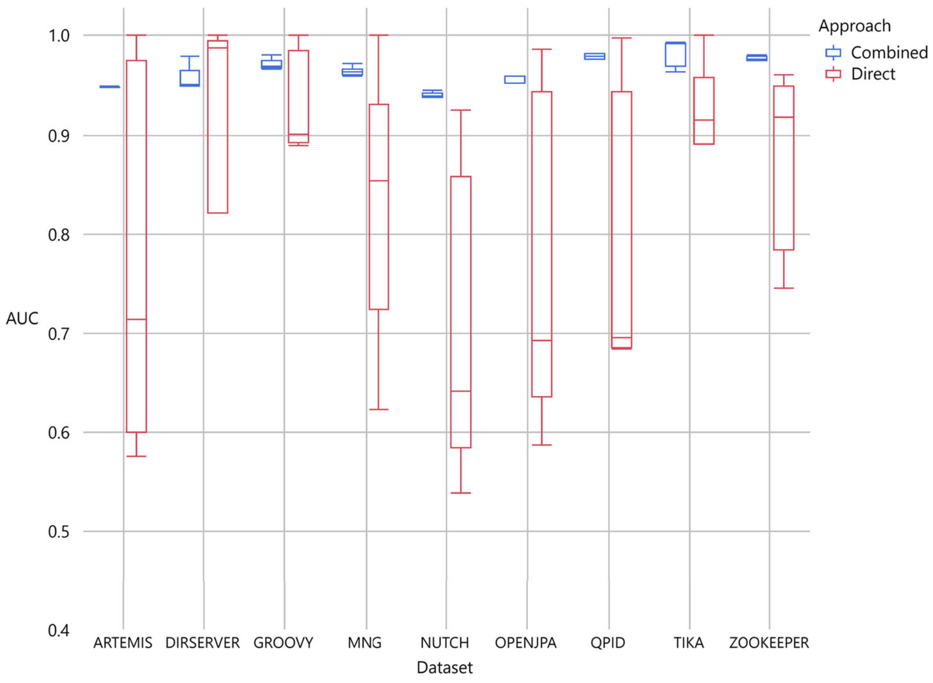


Fig. 6 Distributions across classifiers of AUC values (y-axis) achieved by Combined and Direct (colors) for MDP in our subject projects (x-axis)

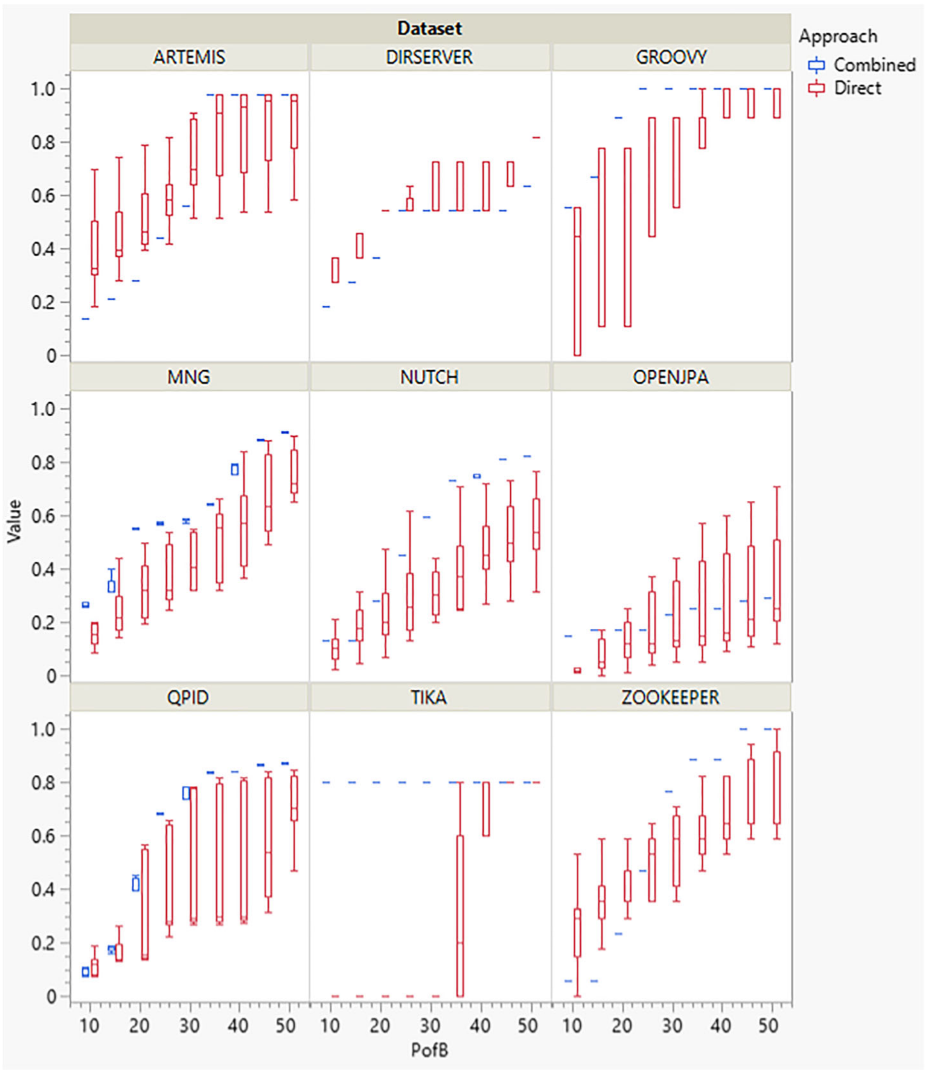


Fig. 7 Distribution across classifiers of PofB values (x-axis) achieved by the Direct and Combined approaches (colors) for MDP in our studied projects (quadrant)

3.3 RQ3: Does Leveraging JIT Information Increase the Accuracy of CDP?

Accuracy Figure 10 reports the distribution of the AUC values across classifiers (y-axis) achieved by different our approaches (i.e., Combined and Direct) for CDP in our subject projects (x-axis). According to Fig. 10:

- The median of Combined is more accurate than Direct in all nine projects.
- As in RQ2, the distribution of values across classifiers of Combined is extremely narrower than the distribution of Direct. Therefore, the choice of classifiers is not as important when using the Combined approach.

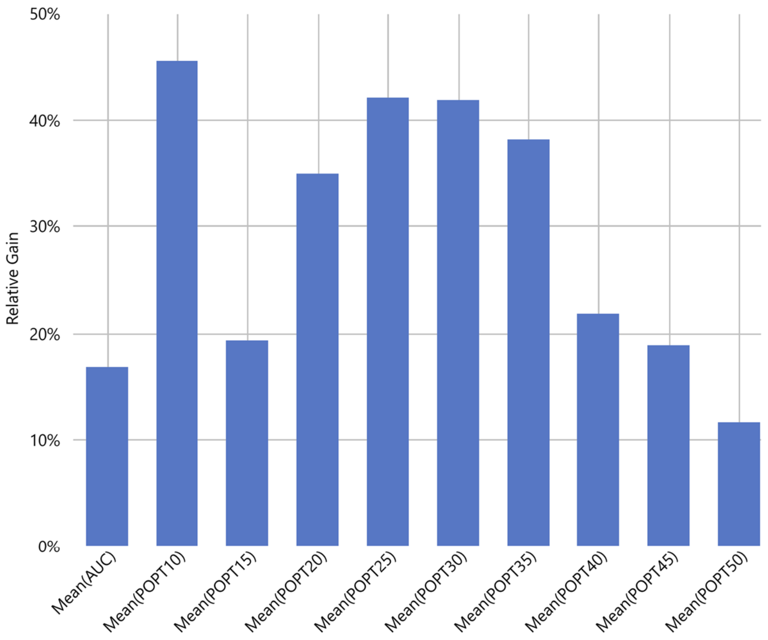


Fig. 8 Mean relative gain in MDP by leveraging JIT across classifiers and datasets

Figure 11 reports the distribution across classifiers of PofB values (x-axis) achieved by different approaches (Combined and Direct) for CDP in our subject projects (quadrant). According to Fig. 11:

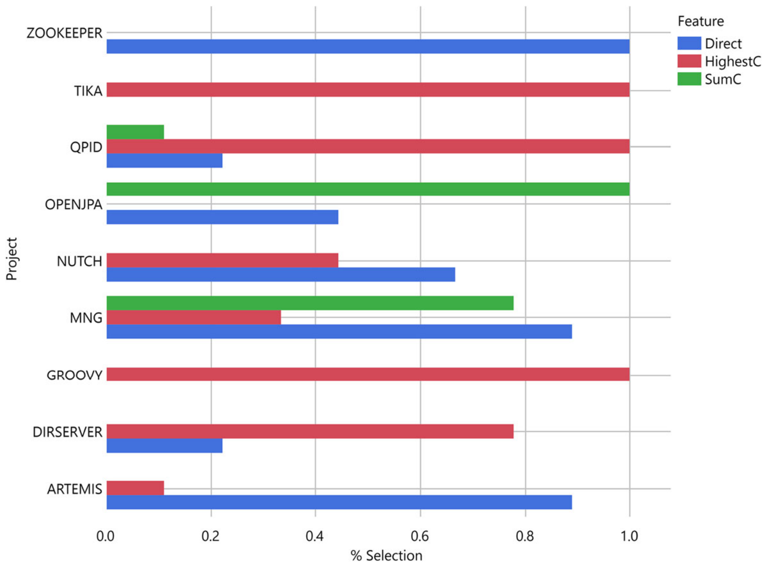


Fig. 9 Percent (x-axis) of times, across the nine classifiers, that a given feature (color) has been chosen in a given project (y-axis), in MDP

Table 5 Statistical result (p-value) and Cohen's d effect size, comparing the MDP accuracy of the combined versus direct approach

	AUC	Popt10	Popt15	Popt20	Popt25	Popt30	Popt35	Popt40	Popt45	Popt50
Pvalue	0.0001*	0.1082	0.5813	0.0275*	0.0001*	0.0001*	0.0001*	0.0001*	0.0001*	0.0001*
Cohen's d	1.1746	0.4052	0.2263	0.4900	0.7130	0.7970	0.8241	0.5896	0.5567	0.3949

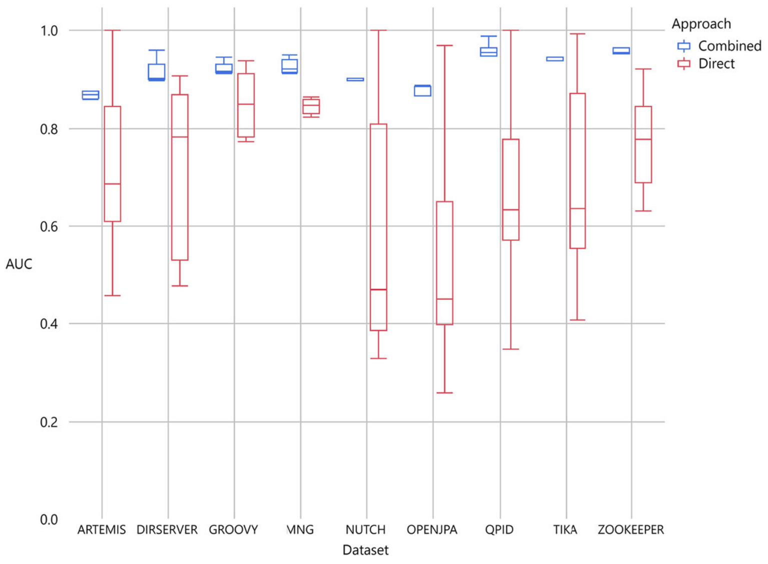


Fig. 10 Distribution of the AUC values across classifiers (y-axis) achieved by different approaches (Combined and Direct) for CDP in our subject projects (x-axis)

- Combined is better than Direct in all PofB values in seven out of nine projects: ARTEMIS, DIRSERVER, MNG, NUTCH, OPENJPA, QPID and TIKA.
- There is no dataset where Combined is worse than Direct in all PofB values.
- Similar to the results for AUC, the distribution of values across classifiers from Combined is extremely narrower than the distribution from Direct. Therefore, the choice of classifiers is not as important when using Combined.

Figure 12 reports the mean relative gain in CDP by leveraging JIT across classifiers and projects. According to Fig. 12 leveraging JIT increases the accuracy of CDP by an average of 31% in AUC and 38% in PofB20. It is interesting to note that the relative gain is not inversely correlated with PofB (as observed in RQ2).

Feature Selection Figure 13 reports the distribution across datasets of feature selection and reports which feature is selected. According to Fig. 13:

- MaxC or SumC have been selected in all nine projects.
- Direct has not been selected in two out of the nine projects.
- MaxC or SumC were selected more than Direct in five out of the nine projects.

Table 6 reports the statistical results (p-value) comparing the CDP accuracy of the combined versus direct approach. An asterisk identifies cases where the pvalue is lower than alpha according to the Holm-Bonferroni correction and hence we can reject the null hypothesis. We note that MDP is statistically more accurate by leveraging JIT in AUC and in seven out of nine PofB. Therefore, we can reject H_{40} and claim that leveraging JIT statistically and significantly improves the accuracy of CDP. Moreover, the effect size is at least medium in most of the metrics.

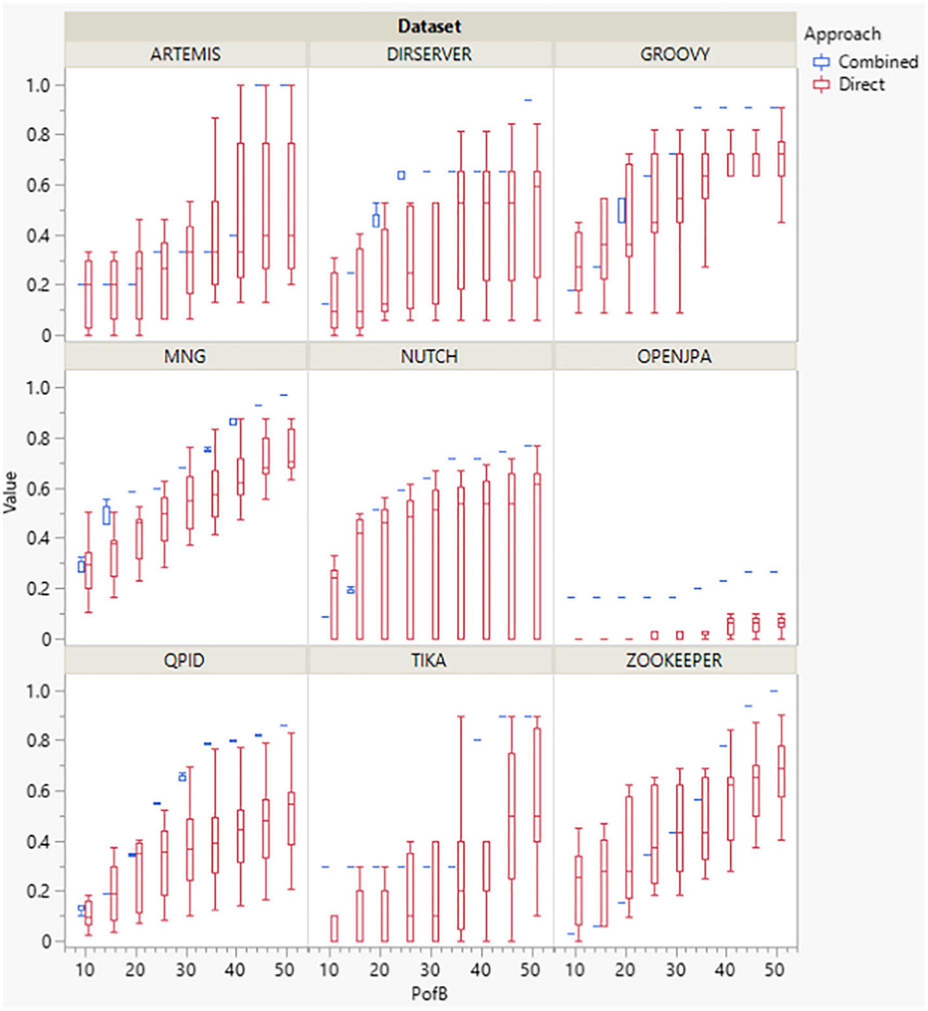


Fig. 11 Distribution across classifiers of PofB values (x-axis) achieved by different approaches (Combined and Direct) for CDP in our subject projects (quadrant)

3.4 RQ4: Are we more Accurate in MDP or CDP?

Figure 14 reports the distribution across classifiers of AUC values (y-axis) achieved for MDP or CDP (colors) in our subject projects (x-axis). According to Fig. 14:

– MDP is more accurate than CDP in all nine projects.
– The distribution of values across classifiers for MDP is extremely narrower than the distribution for CDP. Therefore, the choice of classifiers is less important in MDP than it is in CDP.

Since Fig. 14 results are interesting considering that defective methods are harder to find by chance than defective classes, Fig. 15 reports the distribution across projects of AUC

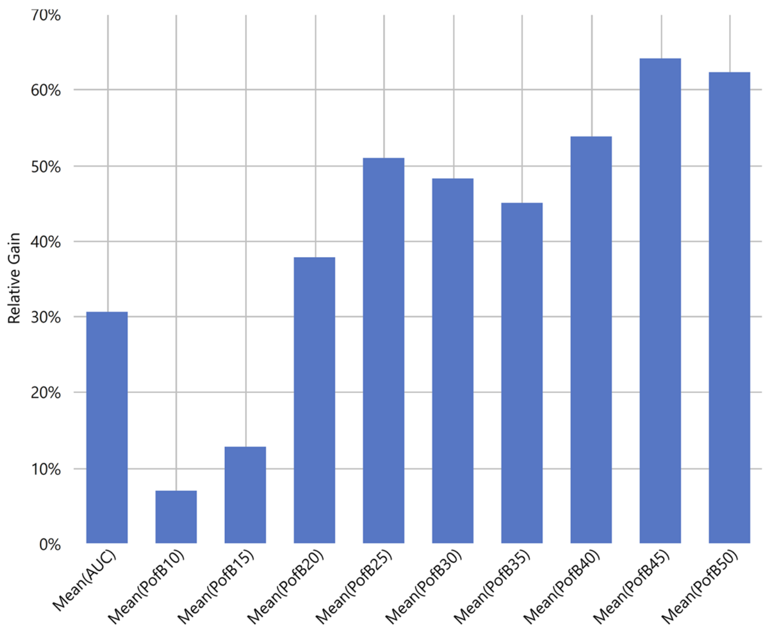


Fig. 12 Mean relative gain in CDP by leveraging JIT across classifiers and datasets

values (y-axis) achieved for MDP or CDP (colors) by our classifiers (x-axis). According to Fig. 15:

– MDP is more accurate than CDP in all nine classifiers.
– The distribution of values across projects for MDP is extremely narrower than the distribution for CDP. Thus, classifiers are more stable in MDP than CDP.

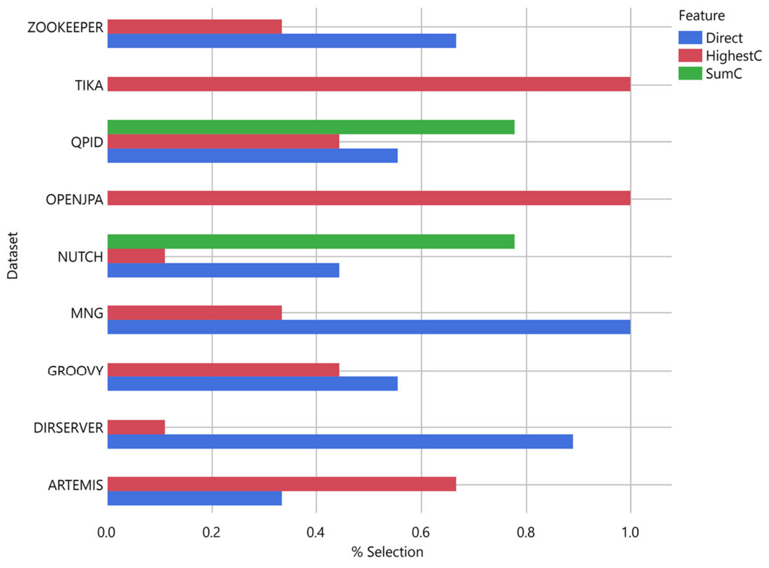


Fig. 13 Distribution across datasets of feature selection for CDP

Table 6 Statistical result (p-value) and Cohen's d effect size, comparing the CDP accuracy of the combined versus direct approach

	AUC	Popt10	Popt15	Popt20	Popt25	Popt30	Popt35	Popt40	Popt45	Popt50
Pvalue	0.0001*	0.2223	0.0913	0.0001*	0.0001*	0.0001*	0.0001*	0.0001*	0.0001*	0.0001*
Cohen's d	1.5737	0.0943	0.1787	0.5269	0.7842	0.7590	0.7295	0.9890	1.2621	1.2941

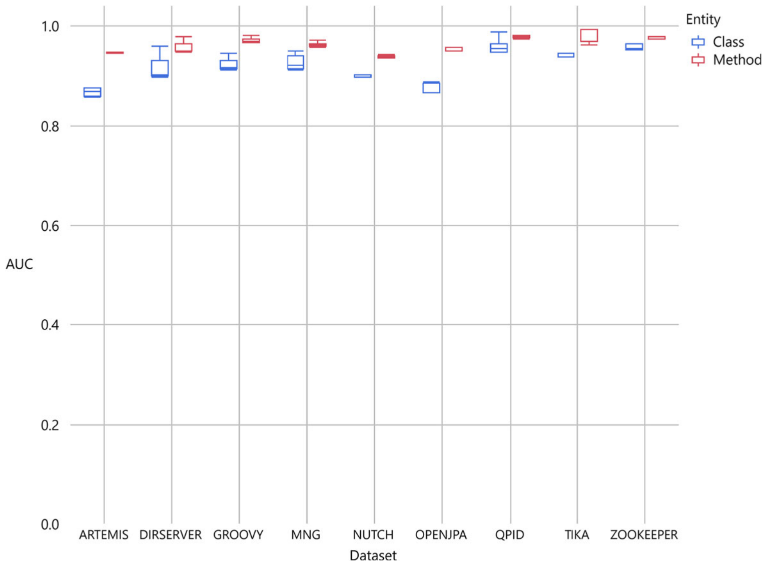


Fig. 14 Distribution across classifiers of AUC values (y-axis) achieved for MDP or CDP (colors) in our subject projects (x-axis)

Figure 16 reports the mean across classifiers for a specific value of PofB (x-axis) achieved by MDP or CDP (colors) in our subject projects (quadrant). We reported the results in bar-charts instead of box-plots (as done in RQ2 and RQ3) because the distributions are very narrow, which hinders the visualization of results in this case. These distributions can

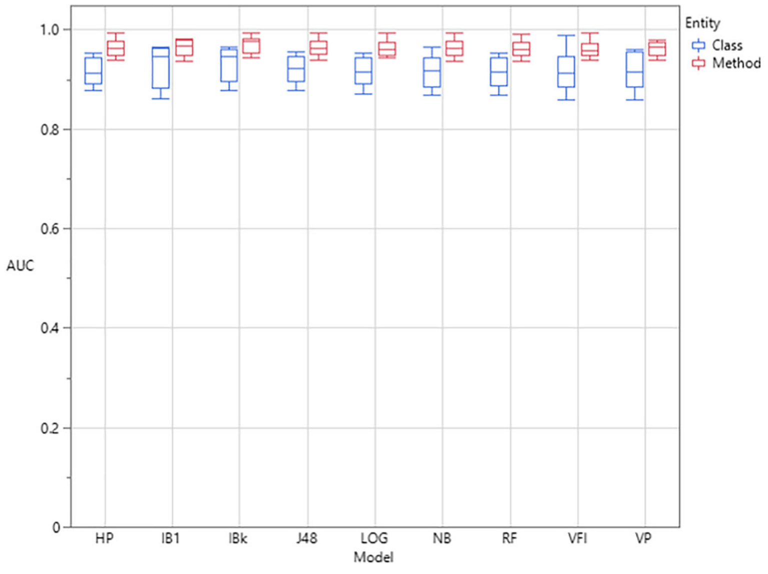


Fig. 15 Distribution across projects of AUC values (y-axis) achieved for MDP or CDP (colors) by our classifiers (x-axis)

also bee seen in previous figures (Fig. 7 for MDP and Fig. 11 for CDP). According to Fig. 16:

- MDP is better than CDP in all PofB values in four projects.
- MDP is worse than CDP in all PofB values only in the Groovy project.

Figure 17 reports the mean of the relative gain in performing MDP over CDP across classifiers and projects. According to Fig. 17 MDP is more accurate than CDP by an average of 5% in AUC and 62% in PofB10. It is worth noting that the relative gain is inversely correlated with PofB again.

Table 7 reports the statistical results (p-value) comparing the accuracy of MDP versus CDP. An asterisk identifies the eight cases out of ten where the pvalue is lower than alpha

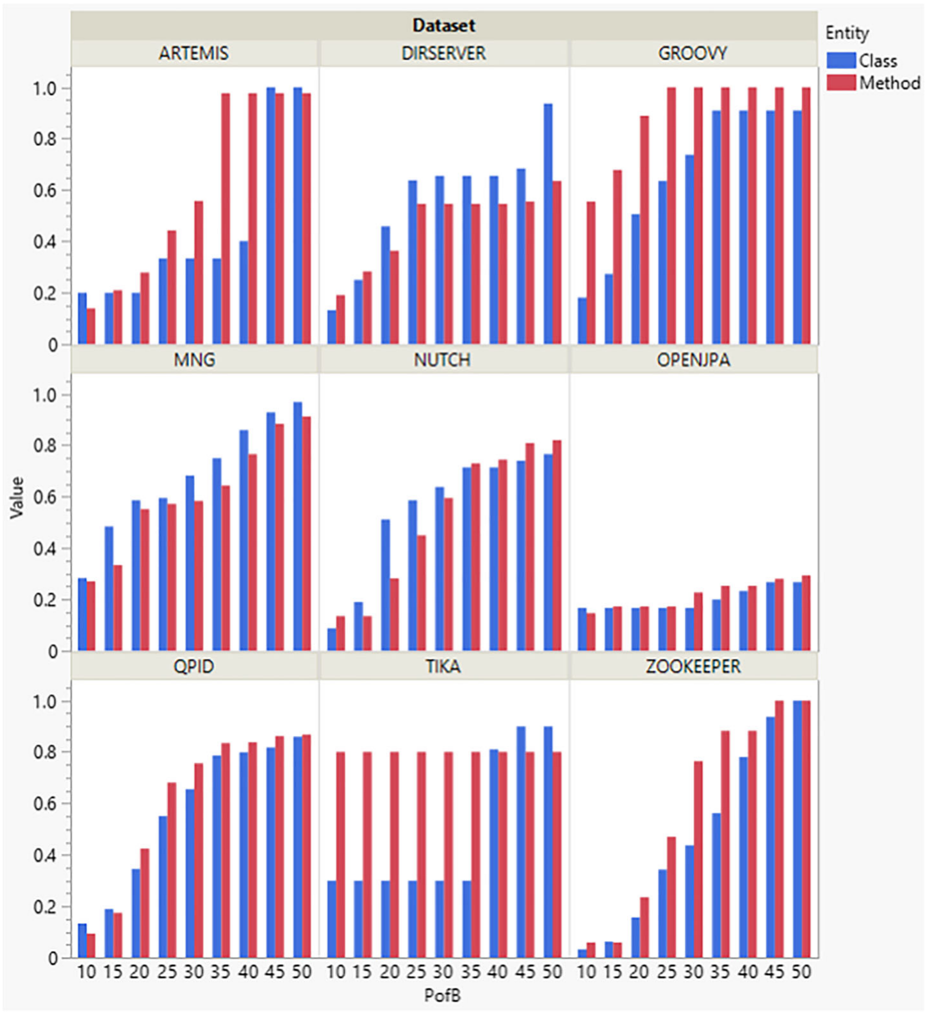


Fig. 16 Mean across classifiers of PofB values (x-axis) achieved for MDP or CDP (colors) in our subject projects (quadrant)

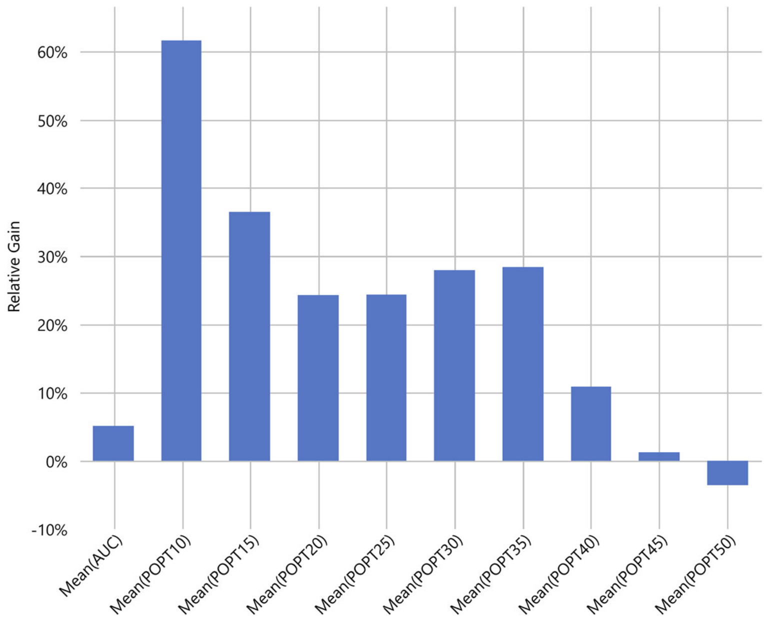


Fig. 17 Mean of the relative gain in performing MDP over CDP across classifiers and projects

according to the Holm-Bonferroni correction and hence we can reject the null hypothesis. We note that MDP is statistically more accurate than CDP in AUC and in seven out of nine PofB. Therefore, we can reject H_{50} and claim that MDP is more accurate than CDP. Moreover, the effect size is at least medium in four out of ten metrics.

4 Discussion

This section discuss our main results, offering possible explanations for the results, implications, and guidelines for practitioners and researchers.

4.1 Main Results and Possible Explanations

The main result of RQ1 is that defective methods are significantly less frequent than defective classes. This means that it is harder to find by chance defective entities if they are methods rather than classes. This result confirms common wisdom for which MDP is more challenging than CDP because defective methods are rare. It was surprising at first to observe that in three out of nine projects the number of defective classes is higher than defective methods. We triple-checked the results and we found no mistakes. Our investigation revealed that such a surprising result was due to defects pertaining only to attributes and that such defects are particularly numerous in those three projects. Let us take the DIRSERVER-1019 as an example defect, which affects the core/src/main/java/org/apache/directory/server/core/jndi/ServerDirContext.java class. By observing the content of the DIRSERVER-1019 fix commit, which has ID 09cc2c065fb36662ebf9f56486af28d87ad09d4c, we realize that developers removed the

Table 7 Statistical result (p-value) and Cohen's d effect size, comparing the MDP versus CDP accuracy

	AUC	Popt10	Popt15	Popt20	Popt25	Popt30	Popt35	Popt40	Popt45	Popt50
Pvalue	0.0001*	0.0001*	0.0001*	0.0002*	0.0001*	0.0001*	0.0001*	0.0003*	0.1381	0.9942
Cohen's d	1.7579	0.5661	0.4334	0.3894	0.4737	0.5959	0.6106	0.2784	0.0367	0.1196

An asterisk identifies a pvalue lower than alpha static keyword from the attribute "static final FilterParserImpl filterParser = new FilterParserImpl();". Such an attribute, and hence the DIRSERVER-1019 defect, was inserted in the commit b392e8f69e2c6f30116459152db612e414f18724. The fact that defective classes can be more than defective methods supports the need for CDP as it can identify defects that MDP cannot.

The main results of RQ4 are that MDP is substantially more accurate than CDP (a mean of +5% in AUC and +62% in PofB10). The higher accuracy of MDP in comparison with CDP is visible in all datasets in terms of AUC. This might be due to the coarser granularity of the classes that, by definition, are larger than methods and, hence, harder to rank as only partially defective. Another possible reason is that defective methods are lower in proportion than defective classes and, hence, a better ranking is more visible in effort-aware metrics like PofB.

The main result of RQ2 is that leveraging JIT increases the accuracy of MDP by an average of 17% in AUC and 46% in PofB10. Similarly, in RQ3, we observe that leveraging JIT increases the accuracy of CDP by an average of 31% in AUC and 38% in PofB20. Moreover, our statistical tests reveal that leveraging JIT increases the accuracy of MDP and CDP in AUC and in PofB (from PofB20 to PofB50). We note that leveraging JIT increases the accuracy of MDP and CDP in all datasets in terms of AUC.

Regarding PofB, in some Combined datasets - PofBX-JIT decreases the accuracy of MDP and CDP. For instance, in Zookeeper, JIT decreases the accuracy of CDP in PofB (from PofB10 to PofB25).

Regarding ZOOKEEPER, it was surprising at first to observe that according to Fig. 6 Combined is better than Direct whereas for the same project in Fig. 9 HighestC and SumC have never been selected. We triple-checked the results and we found no mistakes. If HighestC and SumC have not been selected, Combined cannot be better than Direct. If on the one side, this reasoning is correct, on the other side the datasets upon which the feature selection is applied differ between Figs. 6 and 9. To compute results in Fig. 6, the feature selection is applied to the training set, as it aims at supporting the prediction on the testing set. To compute results in Fig. 9, the feature selection is applied to the entire dataset, as it aims at providing results on a dataset. Thus, there is no inconsistency between results in Figs. 6 and 9.

The Role of Partially Defective Commits We note that leveraging JIT increases the accuracy of MDP or CDP under two conditions: 1) the JIT is accurate, 2)the entities touched by a defective commit are defective. This last point is important, since defective commits usually have only a small proportion of statements that are defect-inducing (Pascarella et al. 2019). Thus, the methods and classes touched by the non-defect-inducing statements from a defective commit are actually not defective. In this study, when we leverage JIT, the defectiveness of a commit is cascaded over all the touching entities, therefore we cascade this defectiveness also over the entities touched by the non-defect-inducing statements of the commit. In other words, while our JIT prediction is performed at the commit level, our ground truth is computed at the defect-inducing-statements level. Thus, the positive impact of leveraging JIT to support MDP or CDP is correlated with the percent of defective entities (classes or methods) touched by defective commits.

To investigate whether the effect of partially defective commits is substantial in our results, we report in Fig. 18 the mean percent, across commits, of defective entities touched by defective commits. A low value of this percentage indicates that defective commits were only partially defective and, hence, they had non-defect-inducing statements touching several entities.

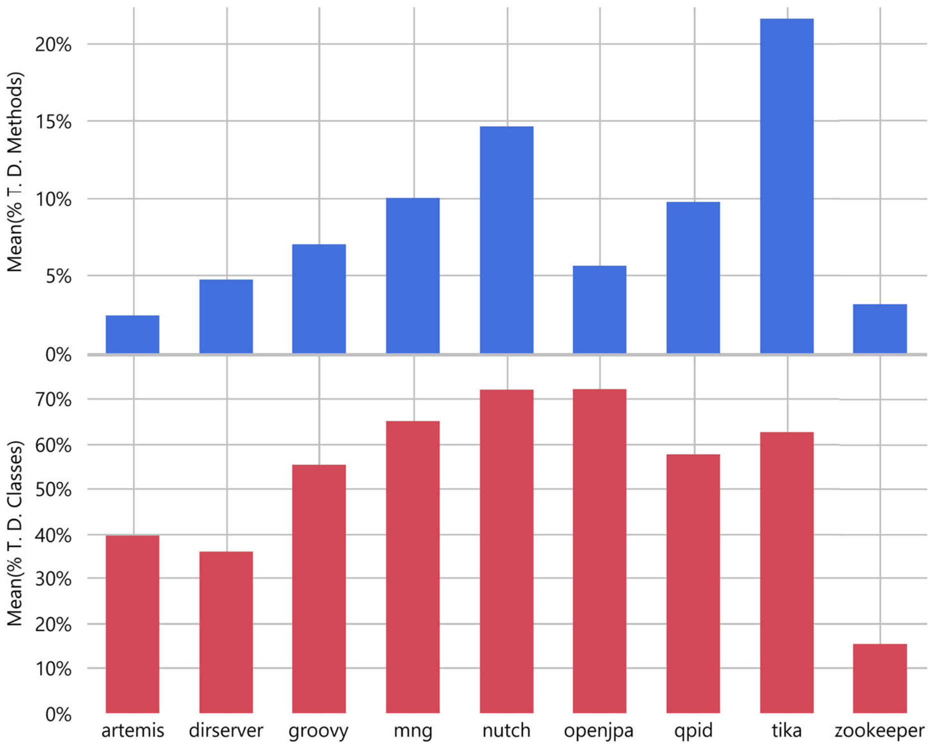


Fig. 18 Mean percent across commits of defective entities that are touched by a defective commit

Regarding RQ2, according to Fig. 18, Artemis, Dirserver and Zookeeper have the lowest proportion of actually defective methods touched by a defective commit; this is in line with the Figs. 7 and 9 as, in Zookeeper, neither HighestC nor SumC have been selected in MDP. Moreover, according to Fig. 18, Tika has the highest proportion of actually defective methods touched by a defective commit; this is in line with Fig. 9 as, in Tika, Direct has never been selected in MDP.

Regarding RQ3, according to Fig. 18, Zookeper has the lowest proportion of actually defective classes; this is in line with Fig. 11 since in this project Combined has a lower PofB10 to PofB25 than Direct. Moreover, according to Fig. 18, Nutch and Openjpa have the highest proportion of actually defective classed touched by a defective commit; this is in line with Fig. 9 as, in Tika, Direct has never been selected in MDP.

Comparing RQ2 to RQ3, we observe that JIT helped more CDP than MDP. This can be explained by Fig. 18, since the percent of entities that are actually defective when touched by a defective commit is about five times higher for classes than it is for methods. This means that leveraging a finer grained defect prediction than JIT, i.e. a statement-defectiveness-prediction (Pornprasit and Tantithamthavorn 2021), would likely benefit more MDP than CDP.

The Narrower Distributions Other important results from RQ2 and RQ3 are that the distributions of accuracy are extremely narrower in both MDP and CDP when leveraging JIT. This indicates that the choice of classifiers does not impact Accuracy as much and, hence,

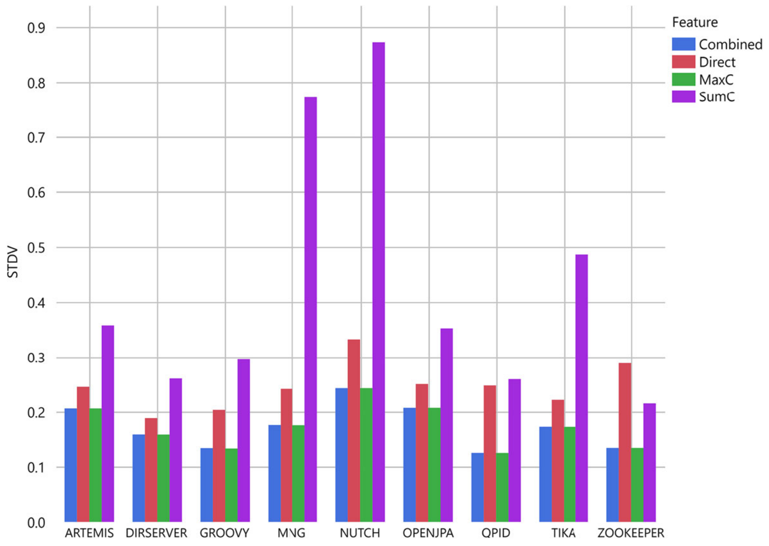


Fig. 19 Distribution across classifiers of standard deviation achieved by different features (colors) for MDP in our projects

leveraging JIT does not only increases the accuracy of both MDP and CDP, but also makes them much more stable across a set of different classifiers. In order to analyze possible reasons as to why the distributions become narrower, Figs. 19 and 20 report the STDV of Combined, Direct, MaxC and SumC in each specific dataset for MDP and CDP, respectively. According to Fig. 19 and 20, Combined has a lower STDV than Direct in all datasets

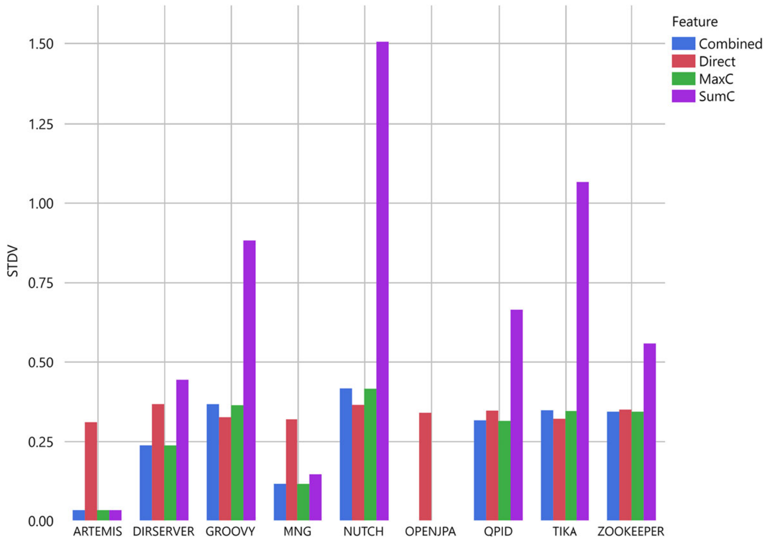


Fig. 20 Distribution across classifiers of standard deviation achieved by different features (colors) for CDP in our projects

in both MDP and CDP. However, the difference in the STDV among Combined and Direct is not as big to explain the significant difference in the resulting accuracy metrics.

We do not see any correlation between defective commits ratio and the benefits of Combined for MDP or CDP. Specifically, Combined is better than Direct also in Groovy, see Figs. 6 and 7 for MDP and see Figs. 10 and 11 for CDP; i.e., leveraging JIT increases the accuracy of MDP and CDP even in a project with only 2% of defective commits. Moreover, we note that large projects might bias results across projects, e.g., Figs. 8 and 12.

4.2 Implications

The main implication of RQ1 is that if practitioners do not use any support system, they should prefer testing instances that are classes instead of methods. Conversely, RQ4 demonstrates that provided the same amount of effort, practitioners would find a much higher percentage of defective entities using a ranked list of methods instead of a list of classes. Specifically, analyzing results across datasets, the lowest accuracy achieved by MDP is higher than the highest accuracy achieved by CDP. Thus, the main implication of RQ4 to practitioners is that it is better to predict and rank defective methods rather than defective classes. The main implications for practitioners of RQ2 and RQ3 is that overall, ranking and classification of both methods and classes shall be done by leveraging JIT information.

Regarding implications to researchers, we found a very limited number of related papers that compared MDP to CDP. For instance, in an ongoing systematic mapping study, we found that, in the last five years, no existing work investigated the accuracy of MDP in three of the major software engineering journals (IEEE Transactions of Software Engineering, ACM Transactions on Software engineering and Methodologies, and Empirical Software Engineering). Hence, there is a high potential for the research community in improving MDP instead of focusing on CDP. Moreover, given the observed low percent of defective methods touched by defective commits, i.e., the defective commits are only partially defective, there is a high potential in leveraging statement-level-defectiveness (Pornprasit and Tantithamthavorn 2021) for MDP.

5 Threats to Validity

In this section, we report the threats to validity of our study. The section is organized by threat type, i.e., Conclusion, Internal, Construct, and External.

5.1 Conclusion

Conclusion validity concerns issues that affect the ability to draw accurate conclusions regarding the observed relationships between the independent and dependent variables (Wohlin et al. 2012).

We tested all hypotheses with non-parametric tests (e.g., Kruskal–Wallis) which are prone to type-2 error, i.e., not rejecting a false hypothesis. We have been able to reject the hypotheses in most of the cases; therefore, the likelihood of a type-2 error is low. Moreover, the alternative would have been using parametric tests (e.g., ANOVA) which are prone to type-1 error, i.e., rejecting a true hypothesis, which, in our context, is less desirable than type-2 error. Also, we acknowledge that our proposed method (i.e., median) to combine JIT with MDP and CDP is a simple and effective baseline to start with (as demonstrated by our results).

There is in the literature a gap between defect prediction accuracy and its actual value in software quality. To face this threat we adopted nine effort-aware metrics; these metrics relate to the effort savings provided by adopting defect prediction models.

Since AUC is sensitive to imbalanced data, and since our datasets are highly unbalanced, AUC results must be interpreted with care. The remaining nine performance metrics are insensitive to imbalanced data; thus, our overall results are not impacted by the imbalanced nature of our data."

5.2 Internal

Internal validity is concerned with influences that can affect the independent variables with respect to causality (Wohlin et al. 2012). A threat to *internal validity* is the lack of ground truth for commits, methods and class defectiveness. In other words, the used RA-SZZ is not perfectly accurate. Nevertheless, we would argue that this is a common threat in most of empirical research in the area of software engineering (Kamei and Shihab 2016). Moreover, to face this threat, we have manually analyzed many commits.

Still regarding the lack of ground truth for commits, methods and class defectiveness, one relevant threat to validity is the possibility that non-linked tickets are related to defect fixes or injections. To face this threat, we tried our best to select projects with the highest linkage. Moreover, we reported the linkage proportion of each project in Table 1 to allow the reader to reason about the validity of the results to specific projects. Finally, we believe that the presence of non-linked commits could have inhibited the observed positive effects of using CDP to support MDP or CDP. Specifically, the inaccuracy in commits labeling, caused by the non-linked commits, reasonably inhibits the accuracy of the commits defectiveness prediction and hence its use.

The execution of a prediction study on defect prediction entails many, often subjective, design decisions such as validation technique, balancing, normalization, tuning, and many more, which might influence the prediction results. We do not expect that our design choices coincide with the choices of all readers, our intent is to use state-of-the-art techniques. We documented all our design choices in Section 2; moreover, we made our replication package available to researchers. Regarding tuning, many studies suggest the tuning of hyperparameters (Fu et al. 2016; Tantithamthavorn et al. 2019); however, in the present study, we use default hyperparameters due to resource constraints and due to the static time-ordering design of our evaluation. In the future, we plan to evaluate the interaction factor between hyperparameters tuning and the benefits of using JIT for MDT and CDP.

In this work, as in many other similar ones (Tantithamthavorn et al. 2016b, 2019; Falessi et al. 2020, 2021; Turhan et al. 2009; Fukushima et al. 2014; Vandehei et al. 2021), we do not differentiate among the severity of defects. If, on one side, the severity of defects is important and practical, on the other side, to the best of our knowledge, there is no study suggesting that the severity of defects impacts defect prediction accuracy. Therefore, in the future, we plan to extend this work by analyzing the sensitivity of the current results to the severity or priority of the considered defects.

5.3 Construct

Construct validity is concerned with the degree to which our measurements indeed reflect what we claim to measure (Wohlin et al. 2012).

In order to avoid that dormant defects would impact our ground-truth, we neglected the last 90% of the releases. This provides us the confidence that snoring is only about 1% in our datasets (Falessi et al. 2021; Ahluwalia et al. 2019).

Our results could also be impacted by our specific design choices including classifiers, features, and accuracy metrics. In order to face these threats, we based our choice on past studies.

One relevant threat to validity is the possibility that non-linked tickets are related to defect fixes or injection; this would bias our ground truth. To face this threat, we tried our best to select projects with the highest linkage. Moreover, we reported the linkage proportion of each project in Table 1 to allow the reader to reason about the validity of the results to specific projects. Finally, we believe that the presence of non-linked commits could have inhibited the observed positive effects of using CDP to support MDP or CDP. Specifically, the inaccuracy in commits labeling, caused by the non-linked commits, reasonably inhibits the accuracy of the commits defectiveness prediction and hence its use.

5.4 External

External validity is concerned with the extent to which the research elements (subjects, artifacts, etc.) are representative of actual elements (Wohlin et al. 2012).

This study used a large set of datasets and, hence, could be deemed of high generalization compared to similar studies. Of course, our results cannot be generalized by projects that would significantly differ from the settings used in this present study. Moreover, since we focused on open-source projects, due to their high availability, we recommend care in generalizing these findings to industrial projects. Please note that considering only mature projects is a threat to external generalizability as results might not generalize to immature projects. One could argue that since only nine nontrivial Apache projects have issue linkage rate above 50%, then no other Apache project might reasonably benefit from the proposed method. We note that the linkage impacts the labeling mechanism, which might have no impact on our results. In practice, our independent variables are orthogonal to the labeling mechanism. In other words, if in the future we will be able to label the defectiveness of commits, classes, and methods according to a linkage-agnostic mechanism, then we believe that our results will still reasonably hold.

Finally, in order to promote reproducible research, all datasets, results, and scripts for this paper are available in our replication package.[6]

6 Related Work

6.1 Combining Heterogeneous Predictions

While countless studies investigated how to predict the defectiveness of commits (Herbold et al. 2020; McIntosh and Kamei 2018; Pascarella et al. 2019; Herbold 2019; Kondo et al. 2020; Huang et al. 2019; Fan et al. 2019; Tu et al. 2020; Rodriguezperez et al. 2020; Pascarella et al. 2020; Giger et al. 2012), or classes (Kamei et al. 2016; Tantithamthavorn et al. 2016b, 2019, 2020; Bennin et al. 2018, 2019; Herbold et al. 2017, 2018; 2019; Hosseini et al. 2019; Yan et al. 2017; Liu et al. 2017; Chi et al. 2017; Jing et al. 2017; Di Nucci et al.

[6]5

2018; Palomba et al. 2019; Song et al. 2019; Zhang et al. 2016, 2017; Lee et al. 2016; Yu et al. 2019a; Peters et al. 2019; Qu et al. 2021; Shepperd et al. 2018; Amasaki 2020; Bangash et al. 2020; Kondo et al. 2019; Morasca and Lavazza 2020; Mori and Uchihira 2019; Tian et al. 2015; Jiarpakdee et al. 2020; Chen et al. 5555; Dalla Palma et al. 2021) in a separate fashion, to the best of our knowledge, no study other than Pascarella et al. (2019), investigated how heterogeneous predictions can benefit one another.

Another family of studies that combines heterogeneous information is the ensemble model, which has been used in the context of defect prediction, as a way to combine the prediction of several classifiers (Laradji et al. 2015; Petric et al. 2016; Tosun et al. 2008; Yang et al. 2017).

6.2 Method Defectiveness Prediction

The first proposing of lowering the granularity of defective prediction have been Menzies et al. (2007) and Tosun et al. (2010). Giger et al. (2012) were the first to perform an MDP study. Specifically, Giger et al. (2012) defined a set of product and process features and found that both product and process features support MDP (i.e., F-Measure=86%).

Our paper has been highly inspired by Pascarella et al. (2020). Specifically, Pascarella et al. (2020) provide negative results regarding the performance of MDP. In other words, using the same design of Giger et al. (2012), they show that MDP is as accurate as a random approach, i.e., the obtained AUC is about 0.51. We share with them several design decisions, including:

– The use of process metrics as features for MDP. Specifically, "The addition of alternative features based on textual, code smells, and developer-related factors improve the performance of the existing models only marginally, if at all." (Pascarella et al. 2020)
– The use of a realistic validation procedure. However, they performed a walk-forward procedure, whereas we performed a simple split by preserving. However, both procedures are realistic since they preserve the order of data (Falessi et al. 2020).

The differences in design include:

– We use an advanced SZZ implementation (RA-SZZ) whereas they use Relink (Wu et al. 2011).
– The use of a different definition of a defective entity. In our research, an entity is defective from when the defect was injected until the last release before the defect has been fixed.
– We use a different set of classifiers.
– We use effort aware metrics such as PofB.
– We selected a different set of projects from which we derived the datasets. The change was due to the fact that we needed the same dataset to produce commit, method, and class data.

The differences in results include that the accuracy achieved by MDP, even without leveraging JIT, is much better than a random approach. Specifically, According to Figs. 6, the median AUC across classifiers and datasets is of 0.81 without leveraging JIT and 0.96 when leveraging JIT. Moreover, the proportion of defective methods is lower by about an order of magnitude in our datasets than in their datasets. Those differences are due to the set of changes in the design, and we prefer not to speculate on which specific change caused the difference in results.

To the best of our knowledge, there is no other study investigating MDP other than Pascarella et al. (2020) and Giger et al. (2012).

Defect prediction can focus on finer-grained software entities other than methods, such as commits (JIT) and statements (Pornprasit and Tantithamthavorn 2021). However, these types of entities (commits and statements) seem more useful to rank at the moment of a commit instead of during the testing phase (which is our target phase in this paper).

7 Conclusion

In this study, we: (i) compare methods and classes in terms of defectiveness; (ii) compare methods and classes in terms of accuracy in defectiveness prediction; (iii) propose and evaluate a first and simple approach that leverages JIT information to increase MDP and (iv) CDP accuracy.

Our analysis features two types of accuracy metrics (threshold-independent and effort-aware) and feature selection metrics, nine machine learning defect prediction classifiers, 1,860 defects related to 35 releases of 9 open source projects from the Apache ecosystem. Our results rely on a ground truth featuring a total of 269,004 data points and 46 features among commits, methods and classes.

Our results reveal that:

– MDP is significantly more accurate than CDP (+5% AUC and 62% PofB10). Thus, it is better to predict and rank defective methods instead of than defective classes from a practitioner's perspective. From a researcher's perspective, given the scarce number of MDP studies, there is a high potential for improving MDP accuracy.
– Leveraging JIT by using a simple median approach increases the accuracy of MDP by an average of 17% in AUC and 46% in PofB10 and increases the accuracy of CDP by an average of 28% in AUC and 31% in PofB20. However, in a few cases, leveraging JIT decreased the accuracy of MDP and CDP.
– Since many defective commits were only partially defective, only a small percent of methods touched by defective commits were actually defective. Therefore, we expect that leveraging statement-defectiveness-prediction (Pornprasit and Tantithamthavorn 2021) would better enhance MDP than JIT.

In conclusion, from a practitioner's perspective, it is better to predict and rank defective methods than defective classes. From a researcher's perspective, there is a high potential for leveraging statement-defectiveness-prediction (SDP) to aid MDP and CDP.

In the future we plan to:

– Propose and evaluate new approaches to improve MDP by leveraging JIT. Specifically, instead of using a static approach like median, we could use a machine learning approach to combine MDP with JIT information.
– Use smell information (Fowler 2018) to support MDP as suggested by previous works (Khomh et al. 2012; Palomba et al. 2018, 2019; Taba et al. 2013).
– Leverage statement level defect prediction (Pornprasit and Tantithamthavorn 2021) to augment MDP and CDP.
– Investigate whether dormant defects (Ahluwalia et al. 2019; Falessi et al. 2021) or other types of noise in the datasets (Chen et al. 2014; Tantithamthavorn et al. 2015; Herzig et al. 2013; Rahman et al. 2013; Bird et al. 2009) have more impact on MDP or CDP.

- Replicate the approach in the context of dependence (Cogo et al. 2019), performance (Chen et al. 2020) or security (Yu et al. 2019b) defects.
- Using multi-level features in a single prediction model. While in this work we evaluated the benefits of combining two predictions, e.g., commits with methods, in the future, we plan to investigate the benefits of performing a single prediction that uses features at different levels, i.e., features at commits and methods levels).

Acknowledgements We thank Bailey Renee Vandehei for proof-reading this article.

Funding Open access funding provided by Università degli Studi di Roma Tor Vergata within the CRUI-CARE Agreement. Open access funding provided by Università degli Studi di Roma Tor Vergata within the CRUI-CARE Agreement.

Declarations

Conflict of Interests The authors have no relevant financial or non-financial interests to disclose. The authors have no conflicts of interest to declare that are relevant to the content of this article.

References

Agrawal A, Menzies T (2018) Is "better data" better than "better data miners"?: on the benefits of tuning SMOTE for defect prediction. In: Proceedings of the 40th international conference on software engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, pp 1050–1061

Ahluwalia A, Falessi D, Di Penta M (2019) Snoring: a noise in defect prediction datasets. In: Storey MD, Adams B, Haiduc S (eds) Proceedings of the 16th international conference on mining software repositories, MSR 2019, 26-27 May 2019. IEEE / ACM, Canada, pp 63–67. https://doi.org/10.1109/MSR.2019.00019

Altman NS (1992) An introduction to kernel and nearest-neighbor nonparametric regression. Am Stat 46(3):175–185. http://www.jstor.org/stable/2685209

Amasaki S (2020) Cross-version defect prediction: use historical data, cross-project data, or both? Empir Softw Eng 25(2):1573–1595

Bangash AA, Sahar H, Hindle A, Ali K (2020) On the time-based conclusion stability of cross-project defect prediction models. Empir Softw Eng 25(6):5047–5083

Basili VR, Briand LC, Melo WL (1996) A validation of object-oriented design metrics as quality indicators. IEEE Trans Software Eng 22(10):751–761

Bennin KE, Keung J, Phannachitta P, Monden A, Mensah S (2018) MAHAKIL: diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction. IEEE Trans Software Eng 44(6):534–550. https://doi.org/10.1109/TSE.2017.2731766

Bennin KE, Keung JW, Monden A (2019) On the relative value of data resampling approaches for software defect prediction. Empir Softw Eng 24(2):602–636

Bird C, Bachmann A, Aune E, Duffy J, Bernstein A, Filkov V, Devanbu P (2009) Fair and balanced?: Bias in bug-fix datasets. In: Proceedings of the the 7th Joint meeting of the european software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, ESEC/FSE '09. ACM, New York, pp 121–130. https://doi.org/10.1145/1595696.1595716

Breiman L (2001) Random forests. Mach Learn 45(1):5–32

Cessie SL, Houwelingen JCV (1992) Ridge estimators in logistic regression. J R Stat Soc Ser C (Appl Stat) 41(1):191–201. http://www.jstor.org/stable/2347628

Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP (2002) Smote: synthetic minority over-sampling technique. J Artif Intell Res 16:321–357

Chen H, Liu W, Gao D, Peng X, Zhao W (2017) Personalized defect prediction for individual source files. 3574531639264263118523398 44(4):90–95. https://doi.org/10.11896/j.issn.1002-137X.2017.04.020

Chen H, Jing X, Li Z, Wu D, Peng Y, Huang Z (5555) An empirical study on heterogeneous defect prediction approaches. IEEE Trans Software Eng (01):1–1. https://doi.org/10.1109/TSE.2020.2968520

Chen J, Shang W, Shihab E (2020) PerfJIT: Test-level just-in-time prediction for performance regression introducing commits. IEEE Trans Softw Eng :1–1. https://doi.org/10.1109/TSE.2020.3023955

Chen TH, Nagappan M, Shihab E, Hassan AE (2014) An empirical study of dormant bugs. Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014. https://doi.org/10.1145/2597073.2597108

Chi J, Honda K, Washizaki H, Fukazawa Y, Munakata K, Morita S, Uehara T, Yamamoto R (2017) Defect analysis and prediction by applying the multistage software reliability growth model. In: IWESEP. IEEE Computer Society, pp 7–11

Cogo FR, Oliva GA, Hassan AE (2019) An empirical study of dependency downgrades in the NPM ecosystem. IEEE Trans Softw Eng :1–1. https://doi.org/10.1109/TSE.2019.2952130

Dalla Palma S, Di Nucci D, Palomba F, Tamburri DA (2021) Within-project defect prediction of infrastructure-as-code using product and process metrics. IEEE Trans Softw Eng :1–1. https://doi.org/10.1109/TSE.2021.3051492

Demiröz G, Güvenir HA (1997) Classification by voting feature intervals. In: ECML, Springer, Lecture Notes in Computer Science, vol 1224, pp 85–92

Di Nucci D, Palomba F, De Rosa G, Bavota G, Oliveto R, De Lucia A (2018) A developer centered bug prediction model. IEEE Trans Softw Eng 44(1):5–24

Falessi D, Huang J, Narayana L, Thai JF, Turhan B (2020) On the need of preserving order of data when validating within-project defect classifiers. Empir Softw Eng 25(6):4805–4830. https://doi.org/10.1007/s10664-020-09868-x

Falessi D, Ahluwalia A, Penta MD (2021) The impact of dormant defects on defect prediction: A study of 19 apache projects. ACM Trans Softw Eng Methodol (TOSEM) 31(1):1–26

Fan Y, Xia X, Alencar da Costa D, Lo D, Hassan AE, Li S (2019) The impact of changes mislabeled by SZZ on just-in-time defect prediction. IEEE Trans Softw Eng :1–1. https://doi.org/10.1109/TSE.2019.2929761

Fowler M (2018) Refactoring: improving the design of existing code. Addison-Wesley Professional, Reading

Freund Y, Schapire RE (1999) Large margin classification using the perceptron algorithm. Mach Learn 37(3):277–296

Fu W, Menzies T, Shen X (2016) Tuning for software analytics: Is it really necessary? Inf Softw Technol 76:135–146

Fukushima T, Kamei Y, McIntosh S, Yamashita K, Ubayashi N (2014) An empirical study of just-in-time defect prediction using cross-project models. In: Proceedings of the 11th working conference on mining software repositories, pp 172–181

Ghotra B, McIntosh S, Hassan AE (2017) A large-scale study of the impact of feature selection techniques on defect classification models. In: 2017 IEEE/ACM 14th international conference on mining software repositories (MSR). IEEE, pp 146–157

Giger E, D'Ambros M, Pinzger M, Gall H (2012) Method-level bug prediction. pp 171–180. https://doi.org/10.1145/2372251.2372285

Gyimóthy T, Ferenc R, Siket I (2005) Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Trans Softw Eng 31(10):897–910

Hall MA (1998) Correlation-based feature subset selection for machine learning. PhD thesis University of Waikato, Hamilton, New Zealand

Hassan AE (2009) Predicting faults using the complexity of code changes. In: Proceedings of the 31st international conference on software engineering, ICSE 2009, May 16-24, 2009. IEEE, Canada, pp 78–88. https://doi.org/10.1109/ICSE.2009.5070510

Herbold S (2017) Comments on scottknottesd in response to "an empirical comparison of model validation techniques for defect prediction models". IEEE Trans Softw Eng 43(11):1091–1094. https://doi.org/10.1109/TSE.2017.2748129

Herbold S (2019) On the costs and profit of software defect prediction. arXiv:1911.04309

Herbold S, Trautsch A, Grabowski J (2017) Global vs. local models for cross-project defect prediction - A replication study. Empir Softw Eng 22(4):1866–1902

Herbold S, Trautsch A, Grabowski J (2018) A comparative study to benchmark cross-project defect prediction approaches. IEEE Trans Softw Eng 44(9):811–833. https://doi.org/10.1109/TSE.2017.2724538

Herbold S, Trautsch A, Grabowski J (2019) Correction of “a comparative study to benchmark cross-project defect prediction approaches”. IEEE Trans Softw Eng 45(6):632–636

Herbold S, Trautsch A, Trautsch F (2020) On the feasibility of automated prediction of bug and non-bug issues. Empir Softw Eng 25(6):5333–5369

Herzig K, Just S, Zeller A (2013) It’s not a bug, it’s a feature: how misclassification impacts bug prediction. In: Proceedings of the 2013 international conference on software engineering, ICSE ’13. IEEE Press, Piscataway, pp 392–401. http://dl.acm.org/citation.cfm?id=2486788.2486840

Holm S (1979) A simple sequentially rejective multiple test procedure. Scand J Stat :65–70

Hosseini S, Turhan B, Gunarathna D (2019) A systematic literature review and meta-analysis on cross project defect prediction. IEEE Trans Softw Eng 45(2):111–147

Huang Q, Xia X, Lo D (2019) Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction. Empir Softw Eng 24(5):2823–2862

Jiang Y, Cukic B, Menzies T (2008) Can data transformation help in the detection of fault-prone modules?. In: Devanbu PT, Murphy B, Nagappan N, Zimmermann T (eds) Proceedings of the 2008 Workshop on Defects in Large Software Systems, held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008), DEFECTS 2008, Seattle, Washington, USA, July 20, 2008. ACM, pp 16–20. https://doi.org/10.1145/1390817.1390822

Jiarpakdee J, Tantithamthavorn C, Dam HK, Grundy J (2020) An empirical study of model-agnostic techniques for defect prediction models. IEEE Trans Softw Eng :1–1 https://doi.org/10.1109/TSE.2020.2982385

Jing X, Wu F, Dong X, Xu B (2017) An improved SDA based defect prediction framework for both within-project and cross-project class-imbalance problems. IEEE Trans Softw Eng 43(4):321–339

Kamei Y, Shihab E (2016) Defect prediction: Accomplishments and future challenges. In: 2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER), vol 5. IEEE, pp 33–45

Kamei Y, Matsumoto S, Monden A, Matsumoto K, Adams B, Hassan AE (2010) Revisiting common bug prediction findings using effort-aware models. In: 26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010. IEEE Computer Society, Timisoara, pp 1–10. https://doi.org/10.1109/ICSM.2010.5609530

Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N (2012) A large-scale empirical study of just-in-time quality assurance. IEEE Trans Softw Eng 39(6):757–773

Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N (2013) A large-scale empirical study of just-in-time quality assurance. IEEE Trans Softw Eng 39(6):757–773. https://doi.org/10.1109/TSE.2012.70

Kamei Y, Fukushima T, McIntosh S, Yamashita K, Ubayashi N, Hassan AE (2016) Studying just-in-time defect prediction using cross-project models. Empir Softw Eng 21(5):2072–2106

Khomh F, Di Penta M, Guéhéneuc YG, Antoniol G (2012) An exploratory study of the impact of antipatterns on class change-and fault-proneness. Empir Softw Eng 17(3):243–275

Khoshgoftaar TM, Allen EB, Goel N, Nandi A, McMullan J (1996) Detection of software modules with high debug code churn in a very large legacy system. In: 7th international symposium on software reliability engineering, ISSRE 1996, October 30, 1996-Nov. 2, 1996. IEEE Computer Society, White Plains, pp 364–371. https://doi.org/10.1109/ISSRE.1996.558896

Kim S, Zimmermann T, Whitehead Jr EJ, Zeller A (2007) Predicting faults from cached history. In: 29th international conference on software engineering (ICSE 2007), May 20-26, 2007. IEEE Computer Society, Minneapolis, pp 489–498. https://doi.org/10.1109/ICSE.2007.66

Kim S, Whitehead Jr EJ, Zhang Y (2008) Classifying software changes: Clean or buggy? IEEE Trans Softw Eng 34(2):181–196

Kondo M, Bezemer CP, Kamei Y, Hassan AE, Mizuno O (2019) The impact of feature reduction techniques on defect prediction models. Empir Softw Eng 24(4):1925–1963

Kondo M, German D, Mizuno O, Choi E (2020) The impact of context metrics on just-in-time defect prediction. Empir Softw Eng 25(1):890–939. https://doi.org/10.1007/s10664-019-09736-3. Funding Information: This work was partially supported by NSERC Canada as well as JSPS KAKENHI Japan (Grant Numbers: JP16K12415). Publisher Copyright: © 2019, Springer Science+Business Media, LLC, part of Springer Nature.

Laradji IH, Alshayeb M, Ghouti L (2015) Software defect prediction using ensemble learning on selected features. Inf Softw Technol 58:388–402. https://doi.org/10.1016/j.infsof.2014.07.005

Lee T, Nam J, Han D, Kim S, In HP (2016) Developer micro interaction metrics for software defect prediction. IEEE Trans Softw Eng 42(11):1015–1035

Lessmann S, Baesens B, Mues C, Pietsch S (2008) Benchmarking classification models for software defect prediction: A proposed framework and novel findings. IEEE Trans Softw Eng 34(4):485–496. https://doi.org/10.1109/TSE.2008.35

Liu J, Zhou Y, Yang Y, Lu H, Xu B (2017) Code churn: A neglected metric in effort-aware just-in-time defect prediction. In: ESEM. IEEE Computer Society, pp 11–19

Mccallum A, Nigam K (2001) A comparison of event models for naive bayes text classification. Work Learn Text Categ :752

McIntosh S, Kamei Y (2018) Are fix-inducing changes a moving target? A longitudinal case study of just-in-time defect prediction. IEEE Trans Softw Eng 44(5):412–428

Mende T, Koschke R (2010) Effort-aware defect prediction models. In: Capilla R, Ferenc R, Dueñas JC (eds) 14th European conference on software maintenance and reengineering, CSMR 2010, 15-18 March 2010. IEEE Computer Society, Spain, pp 107–116, https://doi.org/10.1109/CSMR.2010.18

Menzies T, Greenwald J, Frank A (2007) Data mining static code attributes to learn defect predictors. IEEE Trans Softw Eng 33(1):2–13. https://doi.org/10.1109/TSE.2007.256941

Menzies T, Milton Z, Turhan B, Cukic B, Jiang Y, Basar Bener A (2010) Defect prediction from static code features: current results, limitations, new approaches. Autom Softw Eng 17(4):375–407

Morasca S, Lavazza L (2020) On the assessment of software defect prediction models via ROC curves. Empir Softw Eng 25(5):3977–4019

Mori T, Uchihira N (2019) Balancing the trade-off between accuracy and interpretability in software defect prediction. Empir Softw Eng 24(2):779–825

Moser R, Pedrycz W, Succi G (2008) A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: 30th international conference on software engineering (ICSE 2008), May 10-18, 2008, Leipzig, pp 181–190

Munaiah N, Kroh S, Cabrey C, Nagappan M (2017) Curating github for engineered software projects. Empir Softw Engg 22(6):3219–3253. https://doi.org/10.1007/s10664-017-9512-6

Nagappan N, Ball T (2005) Use of relative code churn measures to predict system defect density. In: Roman G, Griswold WG, Nuseibeh B (eds) 27th international conference on software engineering (ICSE 2005), 15-21 May 2005. ACM, St. Louis, Missouri, pp 284–292. https://doi.org/10.1145/1062455.1062514

Nam J, Fu W, Kim S, Menzies T, Tan L (2017) Heterogeneous defect prediction. IEEE Trans Softw Eng 44(9):874–896

Neto EC, da Costa DA, Kulesza U (2019) Revisiting and improving SZZ implementations. In: 2019 ACM/IEEE international symposium on empirical software engineering and measurement (ESEM), IEEE, pp 1–12

Ohlsson N, Alberg H (1996) Predicting fault-prone software modules in telephone switches. IEEE Trans Softw Eng 22(12):886–894. https://doi.org/10.1109/32.553637

Ostrand TJ, Weyuker EJ (2004) A tool for mining defect-tracking systems to predict fault-prone files. In: Hassan AE, Holt RC, Mockus A (eds) Proceedings of the 1st international workshop on mining software repositories, MSR@ICSE 2004, Edinburgh, Scotland, UK, 25th May 2004, pp 85–89

Ostrand TJ, Weyuker EJ, Bell RM (2005) Predicting the location and number of faults in large software systems. IEEE Trans Softw Eng 31(4):340–355

Palomba F, Bavota G, Di Penta M, Fasano F, Oliveto R, De Lucia A (2018) On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empir Softw Eng 23(3):1188–1221

Palomba F, Zanoni M, Fontana FA, De Lucia A, Oliveto R (2019) Toward a smell-aware bug prediction model. IEEE Trans Softw Eng 45(2):194–218. https://doi.org/10.1109/TSE.2017.2770122

Pascarella L, Palomba F, Bacchelli A (2019) Fine-grained just-in-time defect prediction. J Syst Softw 150:22–36

Pascarella L, Palomba F, Bacchelli A (2020) On the performance of method-level bug prediction: A negative result. J Syst Softw :161

Peters F, Tun TT, Yu Y, Nuseibeh B (2019) Text filtering and ranking for security bug report prediction. IEEE Trans Softw Eng 45(6):615–631

Petric J, Bowes D, Hall T, Christianson B, Baddoo N (2016) Building an ensemble for software defect prediction based on diversity selection. In: Proceedings of the 10th ACM/IEEE international symposium on empirical September 8-9, 2016. ACM, pp 46:1–46:10. https://doi.org/10.1145/2961111.2962610

Pornprasit C, Tantithamthavorn C (2021) Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction. arXiv:210307068

Powers DMW (2007) Evaluation: From precision, recall and F-measure to ROC, informedness, markedness & correlation. J Mach Learn Technol 2(1):37–63

Qu Y, Zheng Q, Chi J, Jin Y, He A, Cui D, Zhang H, Liu T (2021) Using k-core decomposition on class dependency networks to improve bug prediction model's practical performance. IEEE Trans Softw Eng 47(2):348–366

Quinlan JR (1986) Induction of decision trees. Mach Learn 1(1):81–106. https://doi.org/10.1023/A:1022643204877

Rahman F, Posnett D, Herraiz I, Devanbu P (2013) Sample size vs. bias in defect prediction. In: Proceedings of the 2013 9th joint meeting on foundations of software engineering, ESEC/FSE 2013. ACM, New York, pp 147–157

Rey D, Neuhäuser M (2011) Wilcoxon-Signed-Rank Test. Springer, Berlin, pp 1658–1659. https://doi.org/10.1007/978-3-642-04898-2_616

Rodriguezperez G, Nagappan M, Robles G (2020) Watch out for extrinsic bugs! a case study of their impact in just-in-time bug prediction models on the openstack project. IEEE Trans Softw Eng :1–1. https://doi.org/10.1109/TSE.2020.3021380

Shapiro SS, Wilk MB (1965) An analysis of variance test for normality (complete samples). Biometrika 52(3/4):591–611

Shepperd MJ, Hall T, Bowes D (2018) Authors' reply to "comments on 'researcher bias: The use of machine learning in software defect prediction"'. IEEE Trans Softw Eng 44(11):1129–1131

Song Q, Guo Y, Shepperd MJ (2019) A comprehensive investigation of the role of imbalanced learning for software defect prediction. IEEE Trans Softw Eng 45(12):1253–1269

Taba SES, Khomh F, Zou Y, Hassan AE, Nagappan M (2013) Predicting bugs using antipatterns. In: ICSM. IEEE Computer Society, pp 270–279

Tantithamthavorn C, McIntosh S, Hassan AE, Ihara A, Matsumoto K (2015) The impact of mislabelling on the performance and interpretation of defect prediction models. In: Proceedings of the 37th international conference on software engineering, ICSE '15, vol 1. IEEE Press, Piscataway, pp 812–823. http://dl.acm.org/citation.cfm?id=2818754.2818852

Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2016a) Automated parameter optimization of classification techniques for defect prediction models. In: Proceedings of the 38th international conference on software engineering, ICSE 2016, May 14-22, 2016, Austin, pp 321–332

Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2016b) An empirical comparison of model validation techniques for defect prediction models. IEEE Trans Softw Eng 43(1):1–18

Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2019) The impact of automated parameter optimization on defect prediction models. IEEE Trans Softw Eng 45(7):683–711. https://doi.org/10.1109/TSE.2018.2794977

Tantithamthavorn C, Hassan AE, Matsumoto K (2020) The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. IEEE Trans Softw Eng 46(11):1200–1219. https://doi.org/10.1109/TSE.2018.2876537

Tian Y, Lo D, Xia X, Sun C (2015) Automated prediction of bug report priority using multi-factor analysis. Empir Softw Eng 20(5):1354–1383

Tosun A, Turhan B, Bener AB (2008) Ensemble of software defect predictors: a case study. In: Rombach HD, Elbaum SG, Münch J (eds) Proceedings of the 2nd international symposium on empirical software engineering and measurement, ESEM 2008, October 9-10 2008. ACM, Kaiserslautern, pp 318–320, https://doi.org/10.1145/1414004.1414066

Tosun A, Bener A, Turhan B, Menzies T (2010) Practical considerations in deploying statistical methods for defect prediction: A case study within the Turkish telecommunications industry. Inf Softw Technol 52(11):1242–1257. https://doi.org/10.1016/j.infsof.2010.06.006, https://www.sciencedirect.com/science/article/pii/S0950584910001163, special Section on Best Papers PROMISE 2009

Tu H, Yu Z, Menzies T (2020) Better data labelling with emblem (and how that impacts defect prediction). IEEE Trans Softw Eng :1–1 https://doi.org/10.1109/TSE.2020.2986415

Turhan B, Menzies T, Basar Bener A, Di Stefano JS (2009) On the relative value of cross-company and within-company data for defect prediction. Empir Softw Eng 14(5):540–578

Vandehei B, da Costa DA, Falessi D (2021) Leveraging the defects life cycle to label affected versions and defective classes. ACM Trans Softw Eng Methodol 30(2):24:1–24:35

Wang S, Liu T, Tan L (2016) Automatically learning semantic features for defect prediction. In: Proceedings of the 38th international conference on software engineering, ICSE 2016, May 14-22, 2016, Austin, pp 297–308

Wang S, Liu T, Nam J, Tan L (2020) Deep semantic feature learning for software defect prediction. IEEE Trans Software Eng 46(12):1267–1293. https://doi.org/10.1109/TSE.2018.2877612

Weyuker EJ, Ostrand TJ, Bell RM (2010) Comparing the effectiveness of several modeling methods for fault prediction. Empir Softw Eng 15(3):277–295. https://doi.org/10.1007/s10664-009-9111-2

Witten IH, Frank E, Hall MA (2011) Data mining: Practical machine learning tools and techniques, 3rd edn. Morgan Kaufmann Publishers Inc., San Francisco

Wohlin C, Runeson P, Hst M, Ohlsson MC, Regnell B, Wessln A (2012) Experimentation in software engineering. Springer Publishing Company Incorporated, Berlin

Wu R, Zhang H, Kim S, Cheung S (2011) Relink: recovering links between bugs and changes. In: Gyimóthy T, Zeller A (eds) SIGSOFT/FSE'11 19th ACM SIGSOFT symposium on the foundations of software engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011. ACM, pp 15–25, https://doi.org/10.1145/2025113.2025120

Xia X, Lo D, Pan SJ, Nagappan N, Wang X (2016) HYDRA: massively compositional model for cross-project defect prediction. IEEE Trans Softw Eng 42(10):977–998. https://doi.org/10.1109/TSE.2016.2543218

Yan M, Fang Y, Lo D, Xia X, Zhang X (2017) File-level defect prediction: Unsupervised vs. supervised models. In: ESEM. IEEE Computer Society, pp 344–353

Yang X, Lo D, Xia X, Sun J (2017) TLEL: A two-layer ensemble learning approach for just-in-time defect prediction. Inf Softw Technol 87:206–220. https://doi.org/10.1016/j.infsof.2017.03.007

Yu T, Wen W, Han X, Hayes JH (2019a) Conpredictor: Concurrency defect prediction in real-world applications. IEEE Trans Softw Eng 45(6):558–575

Yu Z, Theisen C, Williams L, Menzies T (2019b) Improving vulnerability inspection efficiency using active learning. IEEE Trans Softw Eng :1–1. https://doi.org/10.1109/TSE.2019.2949275

Zhang F, Mockus A, Keivanloo I, Zou Y (2016) Towards building a universal defect prediction model with rank transformed predictors. Empir Softw Eng 21(5):2107–2145

Zhang F, Hassan AE, McIntosh S, Zou Y (2017) The use of summation to aggregate software metrics hinders the performance of defect prediction models. IEEE Trans Softw Eng 43(5):476–491

Affiliations

Davide Falessi[1] · Simone Mesiano Laureani[1] · Jonida Çarka[1] · Matteo Esposito[1] · Daniel Alencar da Costa[2]

Simone Mesiano Laureani
simone.mesianolaureani@alumni.uniroma2.eu

Jonida Çarka
jonida.carka@students.uniroma2.eu

Matteo Esposito
m.esposito@ing.uniroma2.it

Daniel Alencar da Costa
danielcalencar@otago.ac.nz

[1] University of Rome Tor Vergata, Rome, Italy

[2] University of Otago, Otago, New Zealand