Static source code metrics and static analysis warnings for fine-grained just-in-time defect prediction

Alexander Trautsch
Institute of Computer Science
University of Goettingen, Germany
alexander.trautsch@cs.uni-goettingen.de

Steffen Herbold
Institute of Computer Science
University of Goettingen, Germany
herbold@cs.uni-goettingen.de

Jens Grabowski
Institute of Computer Science
University of Goettingen, Germany
grabowski@cs.uni-goettingen.de

***Abstract*—Software quality evolution and predictive models to support decisions about resource distribution in software quality assurance tasks are an important part of software engineering research. Recently, a fine-grained just-in-time defect prediction approach was proposed which has the ability to find bug-inducing files within changes instead of only complete changes. In this work, we utilize this approach and improve it in multiple places: data collection, labeling and features. We include manually validated issue types, an improved SZZ algorithm which discards comments, whitespaces and refactorings. Additionally, we include static source code metrics as well as static analysis warnings and warning density derived metrics as features. To assess whether we can save cost we incorporate a specialized defect prediction cost model. To evaluate our proposed improvements of the fine-grained just-in-time defect prediction approach we conduct a case study that encompasses 38 Java projects, 492,241 file changes in 73,598 commits and spans 15 years. We find that static source code metrics and static analysis warnings are correlated with bugs and that they can improve the quality and cost saving potential of just-in-time defect prediction models.**

***Index Terms*—Software quality, Software metrics**

I. Introduction

Quality assurance budgets are limited. A risk analysis for changes introduced to software would provide hints for quality assurance personal on how to make the most of their limited resources. Just-in-time defect prediction models are predictive models that assign a risk to changes, or files within a change, of being defect-inducing. Because just-in-time models are able to provide feedback directly after a change happened, they can reduce the cost of bug removal.

Just-in-time defect prediction is an active research topic which tries to enable the aforementioned theoretical risk probabilities on a per-change basis. A lot of research is being conducted in this area, e.g., improving the granularity of the predictions [1], adding features, e.g., code review [2], change context [3], or applying deep learning models [4].

Just-in-time defect prediction models are trained on bug-inducing changes, which are found by tracing back bug-fixing changes, e.g., with the SZZ algorithm [5]. Some researchers utilize keyword only approaches that scan the commit messages for certain keywords, e.g., fixes, fixed or patch, to find bug-fixing commits, e.g., [1], [3], [6]. We refer to this approach as ad-hoc SZZ. Others apply full SZZ which requires a link from the commit to the Issue Tracking System (ITS) for identifying bug-fixing commits, e.g., [2], [7], [8]. We refer to this approach as ITS SZZ. In addition to this information, features that describe the change are used as independent variables, e.g., size of the change or diffusion, i.e., how may different subsystems are changed [1], [7], [9].

In contrast to just-in-time defect prediction, release-level defect prediction utilizes features describing files, classes or methods. These features encompass size and complexity metrics as well as object oriented metrics extracted from the files. D'Ambros et al. [10] incorporated change level features into release level defect prediction by including a time-frame before the release for change metric calculation. In addition to static, size and complexity metrics, static analysis warnings were also investigated in the context of quality insurance. Rahman et al. [11] investigated static analysis warnings in the context of release-level defect prediction. Zheng et al. [12] found that the number of static analysis warnings may help to identify defective modules.

Static analysis warnings are generated by tools which inspect different source code representations, e.g., Abstract Syntax Trees (ASTs) or call graphs and find patterns that are known to be problematic. If a problematic pattern is found a warning for the line in the code is generated for the developer. The combination of pattern and generated warning is defined in a rule, these tools allow the developers to define which rules should be utilized by the tool. The rules depend on the tool but most are concerned with readability, common coding mistakes as well as size and complexity thresholds. Results of questionnaires show that developers assign importance to static analysis software, especially at code review time [13], see also Panichella et al. [14].

Recently, Pascarella et al. [1] introduced a fine-grained just-in-time defect prediction approach where instead of complete changes the files contained in these changes are used to train predictive models. Static analysis warnings were not included, however the authors adopted change features for their fine-grained approach together with an ad-hoc SZZ implementation. While Querel et al. [15] included static analysis warnings

for just-in-time defect prediction models, this information has not yet been included in a fine-grained approach. Fan et al. [16] investigated the impact of mislabels by SZZ on just-in-time defect prediction models and found that, depending on the SZZ variant, there can be a significant impact on the models performance.

Independent of the implemented SZZ variant, if a valid link to the ITS is required, there may be additional data validity problems regarding the chosen type of the issue. Prior research by multiple groups found that not every issue classified as a bug in the ITS is actually a bug [17]–[19].

In this work we combine the fine-grained approach by Pascarella et al. [1] with static source code metrics and static analysis warnings. In addition, we include an improved SZZ algorithm which works similar to the approach proposed by Neto et al. [20]. Instead of keyword matches this approach requires valid links to the ITS for each bug-fixing commit. Similar to the approach used by Pascarella et al., its implementation ignores whitespace and comment changes. In addition, it also ignores refactorings. The links between commits and the ITS as well as the types of the linked issues are manually validated. We explore the impact that this SZZ approach has on the resulting models performance in comparison to a keyword based ad-hoc SZZ approach.

We are interested in the impact of additional features on the performance of fine-grained just-in-time defect prediction models. Similar to previous just-in-time defect prediction approaches we include a model to estimate effort. In contrast to effort based on lines of code, the cost model we incorporate is a specialized defect prediction model by Herbold [21] which calculates whether we can save cost by implementing our predictive model.

To summarize the contributions of this work:

- An evaluation of the impact of static source code metrics and static analysis warnings on fine-grained just-in-time defect prediction models.
- Three novel features based on static analysis warning density designed to capture quality evolution regarding static analysis warnings.
- Combination of a specialized defect prediction cost model [21] with a fine-grained just-in-time approach.
- Evaluation of two common labeling strategies in a fine-grained approach, ad-hoc SZZ [1], [3], [6] and ITS SZZ [7], [9], [22].

The rest of this article is structured as follows. In Section II, we introduce prior publications on the topic and relate our current article to it. In Section III, we motivate and define the research questions that we want to answer. Afterwards, we define our case study in Section IV. Section V presents the results of the case study which we discuss in Section VI. After that, we describe threats to validity in Section VII. Finally, we present a short conclusion in Section VIII.

II. Related work

Just-in-time defect prediction has been an active area of research. In this section, we discuss the relevant related work and draw comparisons with our own.

Kamei et al. [7] performed a large-scale empirical study of just-in-time defect prediction. They build predictive models for bug-inducing changes, including effort awareness and also investigate the difference between bug-inducing changes and the rest of the changes. To this end the authors introduced change based metrics which incorporate size, diffusion, purpose, and the history of a change as well as developer experience. The authors use ITS SZZ to find bug-inducing commits without falling back on a keyword based approach. This has a detrimental effect on the performance of their models due to heavy class imbalance as the authors note in the discussion. The predictive models are evaluated with 10-fold-cross-validation.

Tan et al. [8] apply online defect prediction where the window for the training data expands stepwise from the beginning of the project. The authors utilize the commit message, the characteristic vector [23], and churn metrics to build models for 6 open source and one proprietary project. Ad-hoc SZZ is utilized to find bug-fixing commits. The authors point out that cross-validation is not a realistic scenario for just-in-time defect prediction due to it including information from the future. They point out that due to this limitation their model performance is impacted negatively. To mitigate class-imbalance the authors apply and discuss four sampling variants.

Yang et al. [9] further investigated the model complexity utilizing the same data as Kamei et al. [7]. They found that sometimes simple unsupervised models are better than the model introduced by Kamei et al. [7]. The authors also found no big difference between cross-validation and a time-sensitive approach when evaluating the models.

McIntosh et al. [2] investigate whether the properties of bug-inducing changes change over time. The authors utilize change metrics proposed by Kamei et al. [7] and include code review metrics for the predictive models. They analyze the evolution of two open source projects.

Pascarella et al. [1] combine the features used previously by Kamei et al. [7] and Rahman et al. [24] with a fine-grained approach. Instead of predicting bug-inducing changes at the commit level they predict bug-inducing files within one commit. An ad-hoc SZZ implementation is used. To predict one instance the authors use the previous three months of data as training data.

Querel et al. [15] present an addition to commit guru [25] which includes static analysis warnings for building just-in-time defect prediction models. They show that they are able to improve the predictive models with the additional information. Their result complements the results of our case study.

Almost every prior work regarding just-in-time defect prediction relies on some variant of the SZZ algorithm. Although there are differences in its implementation. Some publications use a modified version of the SZZ algorithm which does not utilize an ITS. The original SZZ algorithm does not work as well without an ITS. Without an ITS there is no way to define a suspect boundary date [5]. This results in more bug-fixes

and bug-inducing changes with keyword only ad-hoc SZZ approaches.

In our case study, we investigate this difference by including the ad-hoc SZZ keyword based approach as well as the ITS SZZ approach which requires bug-fixing changes to have a link to a valid ITS issue. The dataset we build upon contains manually validated issue types for every issue that is linked to a bug-fixing commit to account for wrongly classified issues [19].

None of the prior work investigated if static source code metrics or static analysis warnings can improve just-in-time defect prediction models in a fine-grained context. Moreover, none of the prior studies compared how the difference between ad-hoc SZZ and ITS SZZ impact the results of defect prediction. Finally, while some publications considered aspects related to the costs [1], [9], [22] this is the first publication that applies a complete cost model [21] to evaluate the cost saving potential of just-in-time defect prediction.

III. Research questions and analysis procedure

We hypothesize that additional features in the form of static source code metrics and static analysis warnings may improve just-in-time defect prediction models. The commonly used features are change metrics, e.g., [7], [24]. They capture information about the change and itself and the process, e.g., developer experience, size and diffusion of the change. Static source code metrics, e.g., Logical Lines of Code (LLOC), McCabe complexity [26] or object oriented metrics [27] would add additional information about the structure of the files that are contained in the change. Static analysis warnings can add information about violated best practices or naming conventions within the changed files. These features are commonly used for release-level defect prediction and perform well [28]. Moreover, a combination of different sets of features seem promising [29]. Both of the additional sets of features offer different additional information that might positively affect just-in-time defect prediction models. Our investigation into the impact of different features and SZZ approaches on just-in-time defect prediction is driven by the following research questions.

RQ1: Which feature types are correlated with bug-inducing changes?
This question aims to quantify the impact of the features we chose on identifying bug-inducing changes. We are utilizing a linear model which regularizes collinearity between features so that we can focus on the direct impact of each feature on the dependent variable. To broaden our view we also utilize a non-linear model which can also handle collinear features in addition to the linear model.

RQ2: Which feature types improve just-in-time defect prediction?
For this question, we combine recent state-of-the art data extraction and features for just-in-time defect prediction with features that are commonly used for release-level defect prediction. We hope to shed some light on how much release-level feature sets, including static analysis warnings, can improve just-in-time defect prediction.

RQ3: Are static features improving cost effectiveness in just-in-time defect prediction?
Cost awareness is important to estimate the usefulness of the created models. To estimate the cost effectiveness we utilize a cost model explicitly created for defect prediction.

To answer *RQ1* we build two models, a linear logistic regression model and a non-linear random forest [30] model. The data for the linear model is scaled by a z-transformation [31] to prevent features with different scale magnitudes to dominate the objective function. The linear model is also regularized via elastic net to remove collinear features.

The data is not scaled for the random forest as it is robust to scale differences. The random forest choses relevant features via gini impurity which also mitigates collinear features.

The models that are build for *RQ1* contain perfect knowledge, e.g., all information independent of the time it became available is included. As both models get all data, we do not perform sampling to mitigate class imbalance here. Both models are used to rank the features by their importance in the predictive model. The random forest provides this information directly via a feature importance score. For the regularized logistic regression we determine the feature importance by the absolute value of the coefficients, i. e., their impact on the prediction.

To answer *RQ2* we utilize both models as they were used previously in *RQ1*. As the first performance metric for the evaluation of our models we use the harmonic mean of precision and recall, F-measure.

$$\text{precision} = \frac{TP}{TP + FP}$$

$$\text{recall} = \frac{TP}{TP + FN}$$

$$\text{F-measure} = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

TP are the bug-inducing instances correctly predicted by the model, FP non bug-inducing instances incorrectly predicted as bug-inducing. TN are non bug-inducing instances correctly predicted as such and FN are bug-inducing instances incorrectly predicted as non bug-inducing.

Additionally, we include AUC as a model performance measure that is not as impacted by highly imbalanced data. AUC is defined as the area under the Receiver Operating Characteristic (ROC) curve which is a plot of the false positive rate, against the true positive rate. AUC values range from 0 to 1 with 0.5 being equivalent to random guessing and 1 being the perfect value with no false positives and every positive correctly identified. AUC and F-measure are common choices in evaluating model performance, e.g., [1], [4], [7], [8]

To mitigate the class imbalance in our data, we perform SMOTE [32] sampling. SMOTE performs an oversampling of the minority class by creating additional instances which

are similar but not identical to the existing instances of the respective class.

Both classifiers are trained and evaluated on all projects. The models are evaluated for both labeling strategies (ad-hoc SZZ and ITS SZZ) as well as a train/test split as it is done in the replication kit by Pascarella et al. [1]. Additionally, to allow a comparison we replicate the commit label used in the replication kit. The commit label marks every commit in which a file was found inducing as bug-inducing, subsequently every file changed in an inducing commit as inducing.

Additionally, we include a time-sensitive interval approach where we use 3 months as training data and 1 month as testing data. The choice of 3 months is common in related literature, e.g., [1], [7], [8]. The first and last 3 months of each study subject are dropped from the analysis. After that, a sliding window approach is used to train and evaluate a model over the remaining time frame for each study subject.

Restricting the time frame in which training and test data is collected has certain drawbacks. Most prominently we may simply not have enough data to train a model. Therefore we relax the timeframe for the sliding window in certain conditions. 1) Sample size: we select a minimum sample size of the mean number of commits for one month over the project history. For training data this number is multiplied with 3 because the train window size is 3 months. 2) Insufficient positive instances: to perform SMOTE on the training data a minimum number of 5 instances of the minority class is needed. If the training data does not fulfill these conditions, we extend the timeframe of the training data further until we met the conditions.

To compare their performance with different sets of features the results are first combined into boxplots. After that, we perform prerequisite tests for selecting a statistical test to compare the difference between the feature sets. We use autorank [33] to conduct the statistical tests. Autorank implements Demsar's guidelines [34] for the comparison of classifiers. It tests the data for normality and homoscedacity and then automatically selects suitable tests for the data: repeated measures ANOVA as omnibus tests with a post-hoc Tukey HSD test [35] in case the data is normal and homoscedastic and Friedman test [36] as omnibus test with a post-hoc Nemenyi test [37] otherwise. In case of normally distributed data we calculate effects sizes with Cohen's d [38]. If the data is not normal we use Cliff's δ [39] for effect sizes.

We chose a significance level of $\alpha = 0.05$. After Bonferroni [40] correction for 16 statistical tests (4 model performance metrics, 2 labeling strategies for both train/test split and interval approach) we reject the H_0 hypothesis that there is no difference in model performance at $p < 0.003$. We also include critical distance diagrams and plots for the confidence intervals for a combination of both classifiers for both labels, all performance metrics and all feature sets.

For *RQ3* we calculate whether costs can be saved by utilizing a predictive model for directing quality assurance with a cost model introduced by Herbold [21]. The model calculates boundaries for the ratio of cost between quality assurance cost and the cost of the defects. The model takes different relationships between bugs and inducing files into account, e.g., one file may be inducing for multiple bugs.

We count for how many projects our models can save costs and include the upper and lower cost boundaries as further model performance metrics in our ranking.

TABLE I
NUMBER OF COMMITS, FILES AND DEFECTIVE RATES OF OUR STUDY SUBJECTS FOR AD-HOC SZZ AND ITS SZZ

Project	#com	%its	%adh	#files	%its	%adh
ant-ivy	1917	3.29%	25.98%	11581	4.83%	29.91%
archiva	3873	2.89%	11.72%	23899	3.46%	12.25%
calcite	2056	1.07%	12.89%	24653	4.53%	29.81%
cayenne	4157	1.95%	7.60%	42203	3.18%	9.54%
c-bcel	957	1.57%	7.21%	10842	1.02%	10.15%
c-beanutils	741	1.35%	13.09%	4760	0.82%	10.23%
c-codec	1093	0.73%	12.63%	3299	1.76%	14.55%
c-collections	2229	0.58%	8.97%	18362	0.43%	7.26%
c-compress	1765	2.38%	5.38%	5026	4.12%	7.54%
c-configuration	2010	1.24%	8.71%	7011	2.31%	12.22%
c-dbcp	1004	1.99%	21.61%	3459	2.66%	22.95%
c-digester	1375	0.73%	8.44%	5684	0.48%	6.30%
c-io	1411	1.42%	9.99%	4912	1.71%	9.85%
c-jcs	942	2.02%	14.01%	10905	2.60%	20.02%
c-jexl	884	2.15%	15.05%	3962	5.98%	20.92%
c-lang	3966	1.64%	10.26%	11962	1.74%	10.08%
c-math	5098	0.82%	8.14%	32421	1.55%	9.51%
c-net	1435	4.46%	5.64%	6645	2.48%	5.64%
c-scxml	620	1.13%	29.35%	2898	3.11%	39.41%
c-validator	724	2.07%	15.47%	2356	2.12%	13.03%
c-vfs	1378	1.45%	17.78%	9360	1.63%	14.97%
deltaspike	1519	1.97%	3.75%	7464	3.56%	8.87%
eagle	609	0.82%	10.67%	8989	3.39%	32.86%
giraph	861	1.86%	8.83%	9760	3.65%	15.28%
gora	568	1.41%	4.58%	3250	2.62%	7.45%
jspwiki	5086	2.22%	22.87%	20233	1.57%	15.27%
knox	841	2.02%	5.23%	7667	5.16%	9.69%
kylin	4362	3.07%	7.47%	31027	3.64%	11.09%
lens	1491	2.15%	12.41%	11207	5.84%	24.19%
mahout	2393	1.55%	10.03%	26713	2.74%	18.11%
manifoldcf	2609	7.51%	19.93%	17096	4.01%	11.61%
nutch	1536	6.45%	15.17%	7805	6.43%	21.42%
opennlp	1288	2.72%	12.66%	11490	2.04%	10.53%
parquet-mr	1184	1.10%	40.79%	8016	2.88%	47.32%
santuario-java	1432	1.19%	20.11%	12190	1.14%	16.46%
systemml	3645	1.48%	19.56%	36761	2.33%	24.23%
tika	2640	3.64%	10.34%	8797	6.45%	18.17%
wss4j	1899	1.42%	8.64%	17576	1.92%	10.55%

IV. CASE STUDY

In this case study, we investigate the changes introduced into a codebase over a multi-year time period.

A. Data

In this case study we use 38 Java open source projects of the Apache Software Foundation from Herbold et al. [19]. Table I shows the summary statistics of the projects. The number of bug-inducing commits and files is small when we only consider ITS SZZ labels (denoted %its). If we consider ad-hoc SZZ fixes (denoted %adh) the number of bug-inducing commits and files increases significantly. The number of commits only shows the commits where Java source code files were changed. All other commits are ignored.

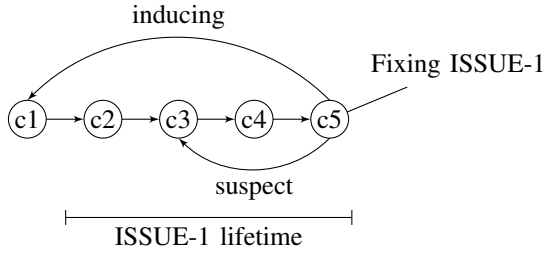

Fig. 1. SZZ algorithm

The data collection by Herbold et al. [19] was performed by SmartSHARK [41]. To utilize the data for a just-in-time defect prediction case study, we implemented an extraction on top of the SmartSHARK database snapshot provided in [19].

We base our extraction on the replication kit by Pascarella et al. [1]. In addition to change features, the extraction provides static source code metrics as well as static analysis warnings as additional features from [19]. Moreover, it provides bug-fixing commits with valid links to the ITS and manually validated bug issues from [19]. This data is integrated into our approach as ITS SZZ labels. Ad-hoc SZZ labels are extracted analogous to [1].

Although the extraction is based on Pascarella et al. [1] we extend it with three improvements. First, to improve the linking between bug-fixing and bug-inducing files, we directly utilize the underlying GitPython[1] instead of the wrapper provided by Pydriller [42]. This allows us to directly access the name of the file at the time when the bug-inducing change happened instead of the current file name.

Second, we implemented a traversal algorithm on top of a Directed Acyclic Graph (DAG) constructed from a complete traversal by Pydriller. By traversing the constructed graph instead of a date ordered list of commits we gain improvements with regard to changes on different branches. We can keep track of files during subsequent renaming or additions and deletions happening on different branches. Furthermore, we can accumulate state information, e.g., number of changes to a file, even if it was renamed on a different branch.

Third, due to the implemented traversal algorithm we can not just ignore merge commits. As Pydriller currently does not support returning modifications on merge commits we implemented a shortcut to the underlying GitPython library to directly access the modifications.

We now describe details of the data collection for our predictive models, namely labels and features. We start by introducing the basic SZZ [5] algorithm, the improvements available from [19] and then both of our labeling strategies. After that we introduce the additional features our models use.

1) Label: Supervised learning models require labeled data, which in our case would be whether a change introduced a bug or not.

The purpose of SZZ is to link bug-fixing commits with their respective bug-reports in the ITS and to link each bug-fixing change to a list of probable bug-inducing changes. Figure 1 shows the basic SZZ algorithm. Changes are denoted as *c1-5*, where *c5* is a bug-fixing change. The time in which ISSUE-1 is created is defined as the suspect boundary, changes that happen before the suspect boundary are bug-inducing changes. Changes after the suspect boundary are suspects and further divided. A suspect change is a partial fix if the suspect change is a fix for another bug. A suspect change is a weak suspect if it is a bug-inducing change (non suspect) for another bug. A suspect change is a hard suspect if it is neither a partial fix or a weak suspect. Hard suspects are discarded while partial fixes and changes inducing another bug are both counted towards bug-inducing changes.

TABLE II
LABELING STRATEGIES USED IN THIS CASE STUDY.

Label	Description
ITS SZZ	Only links to ITS, manually validated issue types and links, discard whitespace, comments and refactorings.
Ad-hoc SZZ	Keywords only (fix, bug, repair, issue, error), discard whitespace and comments.

In this work we utilize two labeling strategies. One uses the ITS and discards hard suspects as described above. The ITS SZZ approach discards whitespace, comments and refactorings and also utilizes manually validated data as it uses the data from [19]. The second uses an ad-hoc SZZ keyword only approach. It filters whitespace changes and comment only changes but does not filter refactoring changes as it is based on Pydriller [42] and not part of the SmartSHARK infrastructure. This approach is similar to the data collection used by Pascarella et al. [1].

Table II provides an overview the labeling strategies. Table I shows the defective rates for our study subjects of the commits and files. A commit is counted as defective if at least one file contained in the commit is defect-inducing. A file is defective if at least one line in the change for that file is defect-inducing.

2) Features: The features our supervised learning models use to predict potential bug-inducing changes are based on prior publications. We include all features used by Pascarella et al. [1]. They consist of features introduced by Kamei et al. [7] and Rahman et al. [24] adopted for fine-grained just-in-time defect prediction. The features introduced by Kamei et al. are used frequently in just-in-time defect prediction, e.g., [9], [43], [44]. They contain features such as the number of lines added, experience of developers and ages on a per file basis.

Additionally, we include features consisting of static analysis warnings by PMD[2] and static source code metrics by OpenStaticAnalyzer[3]. The static source code metrics include object oriented metrics as well as size and complexity metrics, a full list is available online[4].

The static analysis warnings by PMD contain a broad range of rules. From formatting rules, e.g., class names must be in

[1]https://pypi.org/project/GitPython/

[2]https://pmd.github.io/

[3]https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer

[4]https://www.sourcemeter.com/resources/java/

TABLE III
FEATURES USED IN THE FEATURE SETS.

Name	Features
jit	COMM, ADEV, ADD, DEL, OWN, MINOR, SCTR, NADEV, NCOMM, NSCTR, OEXP, EXP, ND, ENTROPY, LA, LD, LT, AGE, NUC, CEXP, SEXP, REXP, FIX_BUG
static	PDA, LOC, CLOC, PUA, McCC, LLOC, LDC, NOS, MISM, CCL, TNOS, TLLOC, NLE, CI, HPL, MI, HPV, CD, NOI, NUMPAR, MISEI, CC, LLDC, NII, CCO, CLC, TCD, NL, TLOC, CLLC, TCLOC, MIMS, HDIF, DLOC, NLM, DIT, NPA, TNLPM, TNLA, NLA, AD, TNLPA, NM, TNG, NLPM, TNM, NOC, NOD, NOP, NLS, NG, TNLG, CBOI, RFC, NLG, TNLS, TNA, NLPA, NOA, WMC, NPM, TNPM, TNS, NA, LCOM5, NS, CBO, TNLM, TNPAA
pmd	ABSALIL, ADLIBDC, AMUO, ATG, AUHCIP, AUOV, BII, BI, BNC, CRS, CSR, CCEWTA, CIS, DCTR, DUFTFLI, DCL, ECB, EFB, EIS, EmSB, ESNIL, ESI, ESS, ESB, ETB, EWS, EO, FLSBWL, JI, MNC, OBEAH, RFFB, UIS, UCT, UNCIE, UOOI, UOM, FLMUB, IESMUB, ISMUB, WLMUB, CTCNSE, PCI, AIO, AAA, APMP, AUNC, DP, DNCGCE, DIS, ODPL, SOE, UC, ACWAM, AbCWAM, ATNFS, ACI, AICICC, APFIFC, APMIFCNE, ARP, ASAML, BC, CWOPCSBF, ClR, CCOM, DLNLISS, EMIACSBA, EN, FDSBASOC, FFCBS, IO, IF, ITGC, LI, MBIS, MSMINIC, NCLISS, NSI, NTSS, OTAC, PLFICIC, PLFIC, PST, REARTN, SDFNL, SBE, SBR, SC, SF, SSSHD, TFBFASS, UEC, UEM, ULBR, USDF, UCIE, ULWCC, UNAION, UV, ACF, EF, FDNCSF, FOCSF, FO, FSBP, DIJL, DI, IFSP, TMSI, UFQN, DNCSE, LHNC, LISNC, MDBASBNC, RINC, RSINC, SEJBFSBF, JUASIM, JUS, JUSS, JUTCTMA, JUTSIA, SBA, TCWTC, UBA, UAEIOAT, UANIOAT, UASIOAT, UATIOAE, GDL, GLS, PL, UCEL, APST, GLSJU, LINSF, MTOL, SP, MSVUID, ADS, AFNMMN, AFNMTN, BGMN, CNC, GN, MeNC, MWSNAEC, NP, PC, SCN, SMN, SCFN, SEMN, SHMN, VNC, AES, AAL, RFI, UWOC, UALIOV, UAAL, USBFSA, AISD, MRIA, ACGE, ACNPE, ACT, ALEI, ARE, ATNIOSE, ATNPE, ATRET, DNEJLE, DNTEIF, EAFC, ADL, ASBF, CASR, CLA, ISB, SBIWC, StI, STS, UCC, UETCS, ClMMIC, LoC, SiDTE, UnI, ULV, UPF, UPM, System/WD, File/System/WD, Author/Delta/WD

TABLE IV
FEATURE SETS USED IN OUR CASE STUDY.

Name	Feature set description
combined	All features combined
jit	Change features commonly used in just-in-time defect prediction adopted for a fine-grained scenario by Pascarella et al. [1].
static	Static source code metrics by OpenStaticAnalyzer. A full list is available online[4]
pmd	Static analysis warnings by PMD also collected via OpenStaticAnalyzer. A full list is available online[4]

TABLE V
ADDITIONAL WARNING DENSITY BASED FEATURES INTRODUCED IN OUR CASE STUDY.

Name	Description
System/WD	The warning density of the project.
File/System/WD	The cumulative difference between warning density of the file and the project as a whole.
Author/Delta/WD	The cumulative sum of the changes in warning density by the author.

CamelCase over rules regarding empty catch blocks up to very specific rules regarding BigDecimal usage. The static analysis warnings and source code metrics are collected for each change and its parent, then a delta is calculated from the current change to its parent change. This allows the included feature to quantify the impact of the change as well as its current and previous value. Table III shows all features included in our case study and their respective feature set. Table IV shows the all feature sets and a short description. In addition to the sum of static metrics and static analysis warnings we introduce new change based metrics utilizing warning density.

$$\text{Warning density} = \frac{\text{Number of static analysis warnings}}{\text{Product size}}$$

Warning density, analogous to defect density [45], describes the ratio of the sum of static analysis warnings to the size of the product, in our cases the LLOC of a file or a whole project.

Table V describes the additional features we introduce based on warning density. With these additional features we hope to capture quality evolution regarding static analysis warnings. If a modified file is consistently below the warning density of the whole project, i.e., contains less static analysis warnings per LLoC, it may be helpful in estimating its quality. Analogous, if the author of a commit consistently lowers the warning density it may also be a good indicator whether a commit by that author may induce defects or not.

B. Replication kit

All data can be found in our replication kit[5]. It also contains code for the creation of the models as well as the plots and tables used in this paper.

V. RESULTS

To answer *RQ1*, we applied a linear logistic regression and a non-linear random forest classification model to a complete set of our data.

Table VI shows the top ten features for both of our classifiers. We note that new static and pmd features are in the top ten for all combinations except for the logistic regression with ad-hoc SZZ labels. The random forest classifier contains warning density based features in its most important features. Jit features, e.g., lines added, deleted or author experience remain important for both classifiers and both labeling strategies. However, this result indicates that we may be able to improve just-in-time defect prediction models with additional static source code metrics and static analysis warnings.

> *RQ1 Summary*: Static as well as pmd warning density based features appear in the top 10 features in 3 out of 4 combinations.

To answer *RQ2*, we start with first replicating the approach utilized in the replication kit by Pascarella et al. [1], i.e., the commit label in a train/test split. Figure 2 shows both performance metrics of both of our models. We can see, that the random forest model performs best with only the jit features while the logistic regression model somewhat performs the same, with the combined features. Our results are consistent with the results obtained by Pascarella et al. [1].

We restrict the labeling now to ad-hoc and ITS SZZ labels. As described in Section IV-A1 with ad-hoc and ITS SZZ we only label the bug-inducing files themselves as bug-inducing independent of the commit. This reduces our positive instances significantly as shown in Table I and also impacts the overall performance.

[5]https://www.github.com/atrautsch/icsme2020_replication

TABLE VI
TOP 10 FEATURES OF BOTH CLASSIFIERS

Logistic regression				Random forest			
ITS SZZ label		Ad-hoc SZZ label		ITS SZZ label		Ad-hoc SZZ label	
la (jit)	0.1085	la (jit)	0.3325	la (jit)	0.0143	la (jit)	0.0276
add (jit)	0.0812	age (jit)	-0.2253	file/system/WD (pmd)	0.0101	add (jit)	0.0208
del (jit)	0.0689	sctr (jit)	-0.2053	system/WD (pmd)	0.0101	exp (jit)	0.0142
entropy (jit)	-0.0656	add (jit)	0.1926	add (jit)	0.0099	oexp (jit)	0.0135
delta_CBO (static)	0.0472	nsctr (jit)	-0.1753	author/delta/WD (pmd)	0.0096	system/WD (pmd)	0.0134
current_NL (static)	0.0438	oexp (jit)	0.1722	ld (jit)	0.0095	entropy (jit)	0.0133
age (jit)	-0.0421	fix_bug (jit)	0.1335	exp (jit)	0.0094	author/delta/WD (pmd)	0.0126
current_NLE (static)	0.0398	ld (jit)	-0.1176	oexp (jit)	0.0085	sctr (jit)	0.0114
system/WD (pmd)	-0.0343	minor (jit)	-0.1113	entropy (jit)	0.0085	delta_HPL (static)	0.0106
current_NUMPAR (static)	0.0340	own (jit)	0.1087	sctr (jit)	0.0078	nd (jit)	0.0101

Figure 3 shows the performance metrics on a train/test split on all of our available data. In comparison with the commit label we can see that with the ad-hoc label both classifiers performance improves slightly with regards to the combined feature set. The combined feature set does not perform best with ad-hoc but the improvement may be an indication that there is a possibility of the model performing better with the combined features. Our next step is to restrict analysis to the ITS SZZ label.

Figure 4 shows the performance metrics for the ITS SZZ label. We observe that the F-measure is significantly lower than with the ad-hoc SZZ labels. However, we can see that both classifiers perform slightly better for the combined feature set.

While until now we performed a train/test split of our data as is done in the replication kit of Pascarella et al. [1]. We now explore whether our assumption holds when evaluating our models in a time-sensitive approach.

Figure 5 and Figure 6 show both classifiers with the interval approach. There is a drop in model performance, especially the F-measure. Regardless of the limited power of the predictive models, as shown by their F-measure, we can see that what we previously demonstrated holds. Adding additional features consisting of static source code metrics and static analysis warnings can improve fine-grained just-in-time defect prediction models, especially if we consider the ITS SZZ labels.

We now rank the performance of both classifiers for all feature sets for each model performance metric using statistical tests. If the data is normally distributed and homoscedastic, we plot the confidence interval and mean for each feature set. Otherwise we plot the critical distance diagram. Figure 7 shows the confidence intervals as well as the critical distance diagrams for both classifiers combined and all feature sets in the train/test split setting. For AUC and F-measure the combined feature set is ranked first. The difference to the second rank is not significant for the ad-hoc SZZ label. However, the difference between first and second rank is significant for the ITS SZZ label for AUC and close to significant for F-measure. Moreover, while the jit feature set is second for the ad-hoc SZZ label this rank is occupied by the static feature set for the ITS SZZ label. Figure 8 shows the critical distance diagrams for both classifiers combined and all feature sets for the interval approach. Again, the combined feature set is ranked first for ad-hoc as well as ITS SZZ. However, the difference to the static features is not significant for the F-measure. We notice that for the ITS SZZ label the static metrics are more important than the jit metrics as was the case for the train/test split.

So far we determined that the combined feature set is ranked first for both AUC and F-measure for both labeling strategies as well as train/test split and interval approaches. However the difference is only significant in some cases. Table VII provides additional details. In addition to mean, standard deviation or in the case of critical distance diagrams, median and median absolute deviation, they provide effect sizes in the form of Cohen's d and Cliffs δ as well as the confidence intervals.

The effect sizes indicate that the differences between the best ranked combined feature set and the second ranked feature set is often negligible. Thus, there is always at least one other feature set that performs similar to the combined feature set. For the ad-hoc SZZ labels, this is the jit feature set, for the ITS SZZ labels, this is the static feature set. However, the difference between combined features and the jit features is large with AUC with the ITS SZZ labels. Similarly, the difference between the combined features and the static features is large with AUC and medium with F-Measure for the ad-hoc labels. This means the combined feature set is the save choice that seems to work best, regardless of the type of labels.

> *RQ2 Summary*: The combined feature set ranks first for AUC and F-measure in every configuration. While the difference to the second ranked feature set is negligible, all other feature sets rank significantly worse with a large effect size for at least once, indicating that the combined features improve the stability of just-in-time defect prediction.

For *RQ3* we calculate if cost savings are possible with the cost model for defect prediction introduced by Herbold [21]. The cost model provides boundary conditions for saving cost by taking predictions for bug-inducing files and the number of bugs into account. By inspecting lower and upper boundaries for each project we can see if we are able to save costs in more projects if we train the predictive model with more features.

Table VIII shows the end result of the cost model boundary calculations. For each label, feature set and classifier it shows the number of projects for which cost saving is possible

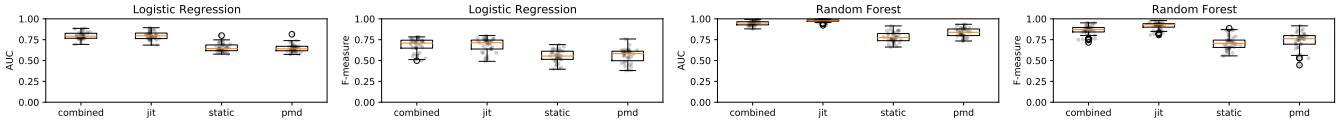

Fig. 2. Model performance metrics with ad-hoc SZZ commit label and train/test split

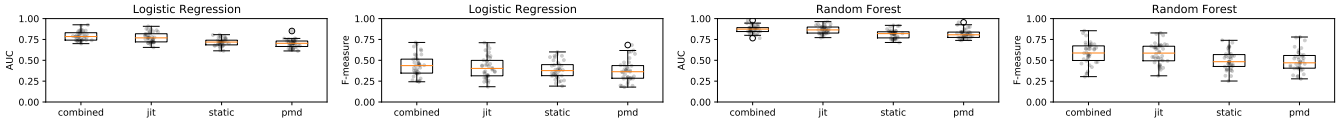

Fig. 3. Model performance metrics with ad-hoc SZZ label and train/test split

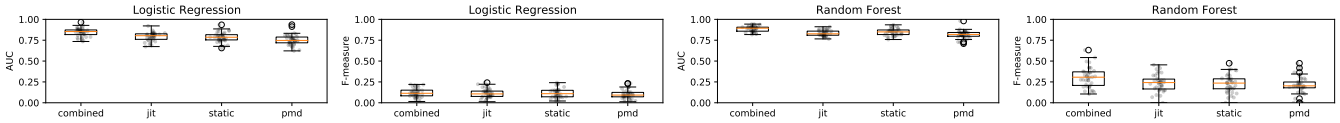

Fig. 4. Model performance metrics with ITS SZZ label and train/test split

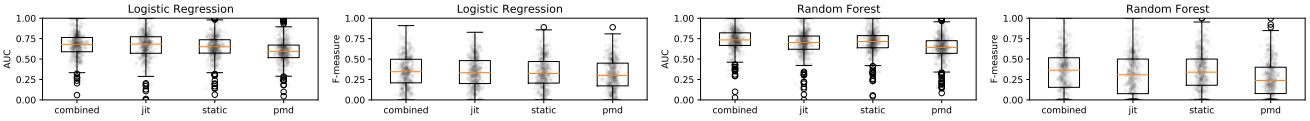

Fig. 5. Model performance metrics with ad-hoc SZZ label and interval approach

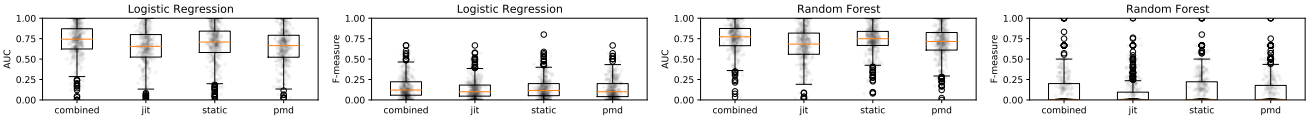

Fig. 6. Model performance metrics with ITS SZZ label and interval approach

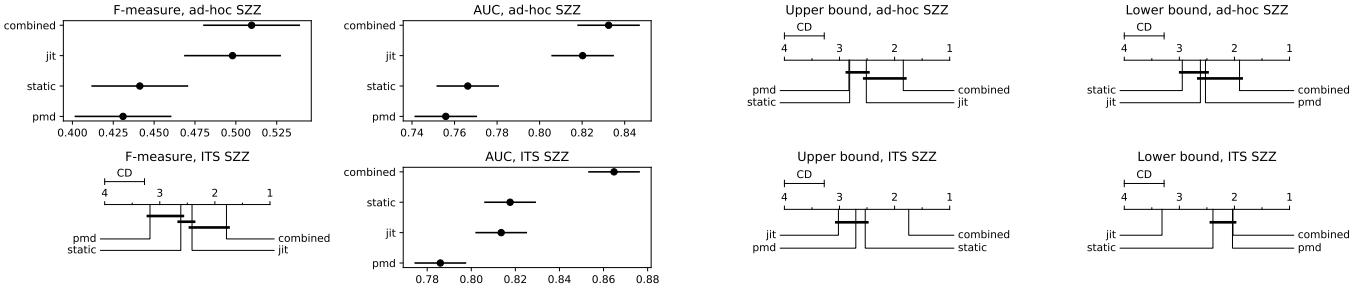

Fig. 7. Ranking of model performance metrics and cost boundaries for the train/test split

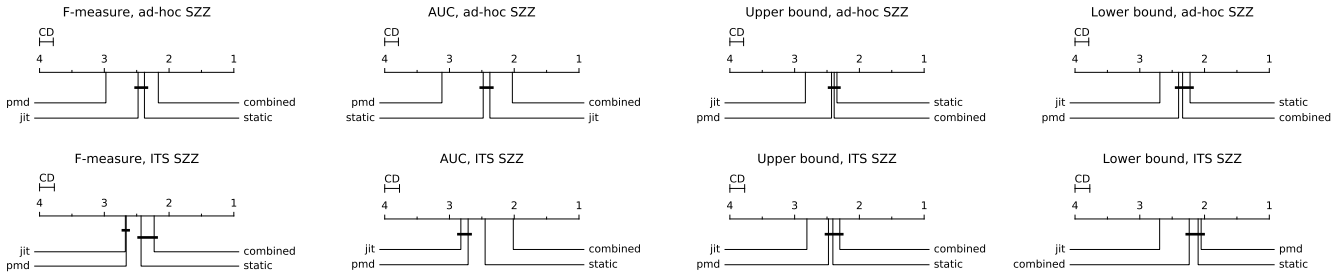

Fig. 8. Ranking of model performance metrics and cost boundaries for the interval approach

TABLE VII

RANKING OF MODEL PERFORMANCE METRICS, MEAN (M), STANDARD DEVIATION (SD), MEDIAN (MED), MEAN ABSOLUTE ERROR (MAD), CONFIDENCE INTERVAL (CI), COHEN'S d (d), CLIFF'S δ (δ) AND EFFECT SIZE MAGNITUDES NEGLIGIBLE (n), SMALL (s), MEDIUM (m), LARGE (l). BOLDING DENOTES A STATISTICALLY SIGNIFICANT DIFFERENCE TO THE FIRST RANK

Train/test split

			M	SD	CI	d
ad-hoc SZZ	AUC	combined	0.832	0.064	[0.818, 0.847]	0.000 (n)
		jit	0.820	0.071	[0.806, 0.835]	0.181 (n)
		static	0.766	0.070	[0.752, 0.781]	**0.988** (l)
		pmd	0.756	0.075	[0.741, 0.771]	**1.091** (l)
			M	SD	CI	d
	F-measure	combined	0.510	0.149	[0.480, 0.539]	0.000 (n)
		jit	0.498	0.152	[0.468, 0.528]	0.078 (n)
		static	0.441	0.123	[0.412, 0.471]	**0.500** (m)
		pmd	0.431	0.139	[0.401, 0.461]	**0.545** (m)
			M	SD	CI	d
ITS SZZ	AUC	combined	0.865	0.048	[0.853, 0.876]	0.000 (n)
		static	0.818	0.059	[0.806, 0.829]	**0.876** (l)
		jit	0.814	0.050	[0.802, 0.825]	**1.051** (l)
		pmd	0.786	0.065	[0.774, 0.798]	**1.376** (l)
			MED	MAD	CI	δ
	F-measure	combined	0.190	0.126	[0.110, 0.320]	0 (n)
		jit	0.151	0.103	[0.090, 0.250]	0.152 (s)
		static	0.152	0.111	[0.094, 0.247]	**0.156** (s)
		pmd	0.134	0.089	[0.080, 0.223]	**0.249** (s)

Interval approach

			MED	MAD	CI	δ
ad-hoc SZZ	AUC	combined	0.707	0.121	[0.685, 0.732]	0.000 (n)
		jit	0.695	0.136	[0.664, 0.716]	**0.078** (n)
		static	0.681	0.126	[0.657, 0.709]	**0.110** (n)
		pmd	0.625	0.123	[0.597, 0.645]	**0.351** (n)
			MED	MAD	CI	δ
	F-measure	combined	0.350	0.236	[0.304, 0.400]	-0.000 (n)
		static	0.333	0.225	[0.286, 0.382]	**0.015** (n)
		jit	0.320	0.250	[0.273, 0.370]	**0.063** (n)
		pmd	0.272	0.227	[0.233, 0.320]	**0.158** (s)
			MED	MAD	CI	δ
ITS SZZ	AUC	combined	0.759	0.170	[0.730, 0.795]	0.000 (n)
		static	0.733	0.162	[0.703, 0.773]	**0.088** (n)
		pmd	0.697	0.186	[0.657, 0.727]	**0.202** (s)
		jit	0.672	0.199	[0.632, 0.716]	**0.247** (s)
			MED	MAD	CI	δ
	F-measure	combined	0.086	0.128	[0.049, 0.126]	0.000 (n)
		static	0.091	0.135	[0.055, 0.127]	-0.011 (n)
		pmd	0.062	0.091	[0.029, 0.100]	**0.057** (n)
		jit	0.054	0.080	[0.000, 0.087]	**0.119** (n)

depending on the costs of defects. For the interval approach it shows the number of intervals for which cost saving is possible. We can see that the number increases between the jit and combined feature sets for both classifiers and both labels. This further indicates that by including static source code metrics and static analysis warnings we may improve a fine-grained just-in-time defect prediction approach and also save cost for software projects using the approach.

In addition to Table VIII, Figure 7 and Figure 8 show the upper and lower bounds ranked for each feature set. As the cost model defines the potential for cost saving the lower bound should be as low as possible while the upper bound as high as possible. To simplify a visual ranking we reversed the rank order for the lower bound the plots. For the train/test split

TABLE VIII

NUMBER OF PROJECTS/INTERVALS WHERE COST CAN BE SAVED FOR BOTH CLASSIFIERS AND MEAN NUMBER OF PROJECTS.

Label		Feature set	$\frac{1}{2}$(#LR + #RF)	#LR	#RF
train/test split	ad-hoc SZZ	jit	23.0	24	22
		static	19.5	15	24
		pmd	20.5	13	28
		combined	34	26	31
	ITS SZZ	jit	24.5	23	26
		static	35.0	37	33
		pmd	32	35	29
		combined	34	32	36
interval	ad-hoc SZZ	jit	109	111	107
		static	170.5	162	179
		pmd	160.0	152	168
		combined	162.5	150	175
	ITS SZZ	jit	87.5	109	66
		static	129.5	146	113
		pmd	137	168	106
		combined	121	99	143

in Figure 7 we can see that for the ad-hoc SZZ label the combined feature set is ranked first for upper and lower bound. While this indicates that more cost savings are possible with the combined feature set the critical distance to the next rank is not exceeded. For the ITS SZZ label we see that while the combined feature set is ranked first for the upper bound the best feature set for the lower bound is static. Although we note the large difference of the jit feature set to the other feature sets.

For the interval approach depicted in Figure 8 we can see that static performs best for the ad-hoc SZZ, with combined second. However the critical distance is not exceeded except for the jit feature set which performs worse. For the ITS SZZ label, combined is again best, although critical distance is only exceeded again for jit which performs worse. The lower bounds show that static is the best feature set for ad-hoc SZZ and pmd for ITS SZZ. However the critical distance to the combined feature set is not exceeded. This is also shown in Table VIII, we can see that models build with ITS SZZ and static/pmd features are able to save cost in more projects.

RQ3 Summary: The potential for cost saving is higher with a combined feature set than with only jit features. However, static and pmd features perform better with the ITS SZZ labeling strategy.

VI. DISCUSSION

In the answer to our first research question regarding the importance of adding static features and static analysis warnings to just-in-time defect prediction we first find that the top 10 features for our regularized linear model and random forest contain static and pmd features in 3 out of 4 combinations. The linear model with ad-hoc SZZ labels is the only one which contains only jit features. This analysis shows that, given perfect knowledge, both ways to measure the importance of features indicate that static metrics can have correlations with defects. Since we use regularization to account for collinearity these correlations provide an indication that static source code

metrics carry useful information about defects that is not contained in the features proposed by Kamei et al. [7] which are the standard choice for just-in-time defect prediction.

The results of the model evaluation show that for the label (commit) also used by Pascarella et al. [1] in their replication kit there is no performance gain when using additional metrics. Although, the more detailed the labeling process gets, i.e., ad-hoc SZZ for keyword only SZZ, ITS SZZ for full SZZ, the more positive impact additional static source code metrics and static analysis warnings as features have on the predictive models. This is also reflected by our final analysis which incorporates a sliding window approach for time-sensitive analysis. The combined feature set is ranked first in every case. The performance drops between train/test split and interval are also in line with the literature, e.g., Tan et al. [32].

The results for *RQ3* show that in our reproduction of Pascarella et al. [1] our created models can save cost. We see that the combined feature set allows us to utilize the predictive model to save cost in more cases than the jit feature set. However, with ITS SZZ labels we see that the models built with static and pmd feature sets are able to save cost in more cases than the combined feature set. This is another indication that file-based metrics are more important for ITS SZZ labels than in a ad-hoc SZZ labeling strategy.

As a final note, we believe that both labeling strategies have their use. Ad-hoc SZZ labels can be used to distinguish possible quick fixes developers apply from possible bigger issues that are more indicative of an entry in an ITS. However, we have shown that it is very important to be aware of this as the defective rates for both approaches differ significantly, especially in a fine-grained scenario.

VII. Threats to Validity

In this section we discuss the threats to validity we identified for our work. To structure this section we discuss four basic types of validity, as suggested by Wohlin et al. [46].

A. Construct Validity

The link between bug-fixing and bug-inducing commits is at the heart of this study. We are aware that some variants of the SZZ [5] algorithm have a certain imprecision [16], [47]. The ad-hoc SZZ label in our study ignores whitespace and comment changes while the ITS SZZ label additionally ignores refactoring changes which removes more false positives [20].

The ITS SZZ label in our study relies on a link between the ITS and the Version Control System (VCS), i.e., the bug-fixing commit must be linked to a valid issue of the type bug in the ITS. The type of the issue in the ITS may not reflect the real type but instead feature requests or other change requests [17], [18]. To mitigate this threat, our study subjects are based on a convenience sample of the Apache Software Foundation ecosystem. Not only do the ASF developers a good job of linking changes to issues, this sample also includes manually validated issue types and links [19].

B. Internal Validity

Our results are influenced by the data collected from our study subjects. Factors that we are not able to change include the number of changes over time. This has a pronounced impact on model performance as can be seen in Figure 5. As we do not want to chose our study subjects based on their commit history we are forced to handle fluctuating change histories. We do this by relaxing a strict time window as used in prior publications by also requiring a minimum number of changes for the dataset. Instead of choosing hard values for the number of changes we require the average number of changes for that time frame over the complete change history of the considered study subject. This improves the performance of the models and, in our eyes, is a reasonable choice. Nevertheless, this still is a factor that impacts our internal validity and warrants future research, i.e., how can just-in-time defect prediction work with all kinds of projects.

C. External Validity

A threat to external validity is our project selection. Although the projects are all Java and originate from the same organization they contain a variety of developers due to their open source nature. Moreover, our sample contains a diverse set of application domains, e.g., wiki software, math libraries and build systems. Nevertheless, our results may not be applicable to all Java projects of the Apache Software Foundation much less all Java projects in existence.

D. Conclusion Validity

As our study investigates many features we perform regularization on our linear logistic regression classifier. To further complete our view we additionally include a non linear random forest classifier. Both should be able to handle collinear features. Moreover, we apply statistical tests to enhance the validity of our conclusion for *RQ2* and *RQ3*.

VIII. Conclusion

In this work we combined a state-of-the art just-in-time defect prediction approach with additional static source code metrics from OpenStaticAnalyzer and static analysis warnings from a well known Java static analysis tool (PMD). We create additional features based on warning density and show that additional features can improve just-in-time defect prediction models depending on the granularity of the labeling strategy. We investigated two labeling strategies in depth, ad-hoc SZZ and ITS SZZ and found that the more targeted the label the more the models performance is positively impacted by the additional features. We conclude that highly targeted models, i.e., models that target bugs linked to an ITS profit from the additional features.

We applied a defect prediction cost model to investigate if cost saving is possible with our created models. The number of projects where cost can be saved increases between jit only and combined feature sets. For ITS SZZ static and pmd feature sets provide more cost saving opportunities.

REFERENCES

[1] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," *Journal of Systems and Software*, vol. 150, pp. 22 – 36, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121218302656

[2] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," *IEEE Transactions on Software Engineering*, vol. 44, no. 5, pp. 412–428, May 2018.

[3] M. Kondo, D. M. German, O. Mizuno, and E.-H. Choi, "The impact of context metrics on just-in-time defect prediction," *Empirical Software Engineering*, vol. 25, no. 1, pp. 890–939, 2020.

[4] T. Hoang, H. Khanh Dam, Y. Kamei, D. Lo, and N. Ubayashi, "Deepjit: An end-to-end deep learning framework for just-in-time defect prediction," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 34–45.

[5] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, May 2005.

[6] G. G. Cabral, L. L. Minku, E. Shihab, and S. Mujahid, "Class imbalance evolution and verification latency in just-in-time software defect prediction," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, May 2019, pp. 666–676.

[7] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, June 2013.

[8] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, May 2015, pp. 99–108.

[9] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, "Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 157–168. [Online]. Available: https://doi.org/10.1145/2950290.2950353

[10] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison," *Empirical Softw. Engg.*, vol. 17, no. 4-5, pp. 531–577, Aug. 2012. [Online]. Available: http://dx.doi.org/10.1007/s10664-011-9173-9

[11] F. Rahman, S. Khatri, E. T. Barr, and P. Devanbu, "Comparing static bug finders and statistical prediction," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 424–434. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568269

[12] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE Transactions on Software Engineering*, vol. 32, no. 4, pp. 240–253, April 2006.

[13] P. Devanbu, T. Zimmermann, and C. Bird, "Belief evidence in empirical software engineering," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, pp. 108–119.

[14] S. Panichella, V. Arnaoudova, M. D. Penta, and G. Antoniol, "Would static analysis tools help developers with code reviews?" in *2015 IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, vol. 00, March 2015, pp. 161–170.

[15] L.-P. Querel and P. C. Rigby, "Warningsguru: Integrating statistical bug models with static analysis to provide timely and specific bug warnings," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 892–895. [Online]. Available: https://doi.org/10.1145/3236024.3264599

[16] Y. Fan, D. Alencar da Costa, D. Lo, A. E. Hassan, and L. Shanping, "The impact of mislabeled changes by szz on just-in-time defect prediction," *IEEE Transactions on Software Engineering*, 2020.

[17] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: A text-based approach to classify change requests," in *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, ser. CASCON '08. New York, NY, USA: ACM, 2008, pp. 23:304–23:318. [Online]. Available: http://doi.acm.org/10.1145/1463788.1463819

[18] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: How misclassification impacts bug prediction," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 392–401. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486840

[19] S. Herbold, A. Trautsch, F. Trautsch, and B. Ledel, "Issues with szz: An empirical study of the state of practice of defect prediction data collection," *Submitted to: ACM Transactions on Software Engineering and Methodology*, 2020. [Online]. Available: https://arxiv.org/abs/1911.08938

[20] E. C. Neto, D. A. da Costa, and U. Kulesza, "The impact of refactoring changes on the szz algorithm: An empirical study," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2018, pp. 380–390.

[21] S. Herbold, "On the costs and profit of software defect prediction," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[22] Q. Huang, X. Xia, and D. Lo, "Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 159–170.

[23] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2013, pp. 279–289.

[24] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 432–441.

[25] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: Analytics and risk prediction of software commits," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 966–969. [Online]. Available: https://doi.org/10.1145/2786805.2803183

[26] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, pp. 308–320, Jul. 1976.

[27] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[28] S. Hosseini, B. Turhan, and D. Gunarathna, "A systematic literature review and meta-analysis on cross project defect prediction," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2017.

[29] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1276–1304, Nov 2012.

[30] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, Oct. 2001. [Online]. Available: http://dx.doi.org/10.1023/A:1010933404324

[31] E. Kreyszig, *Advanced Engineering Mathematics: Maple Computer Guide*, 8th ed. New York, NY, USA: John Wiley & Sons, Inc., 2000.

[32] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 99–108.

[33] S. Herbold, "Autorank: A python package for automated ranking of classifiers," *Journal of Open Source Software*, vol. 5, no. 48, p. 2173, 2020. [Online]. Available: https://doi.org/10.21105/joss.02173

[34] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *J. Mach. Learn. Res.*, vol. 7, pp. 1–30, Dec. 2006.

[35] J. W. Tukey, "Comparing individual means in the analysis of variance," *Biometrics*, vol. 5, no. 2, pp. 99–114, 1949. [Online]. Available: http://www.jstor.org/stable/3001913

[36] M. Friedman, "A comparison of alternative tests of significance for the problem of m rankings," *The Annals of Mathematical Statistics*, vol. 11, no. 1, pp. 86–92, 1940.

[37] P. Nemenyi, "Distribution-free multiple comparison," Ph.D. dissertation, Princeton University, 1963.

[38] J. Cohen, *Statistical power analysis for the behavioral sciences*. L. Erlbaum Associates, 1988.

[39] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological Bulletin*, vol. 114, no. 3, p. 494, 1993.

[40] H. Abdi, "Bonferroni and Sidak corrections for multiple comparisons," in *Encyclopedia of Measurement and Statistics*. Sage, Thousand Oaks, CA, 2007, pp. 103–107.

[41] F. Trautsch, S. Herbold, P. Makedonski, and J. Grabowski, "Addressing problems with replicability and validity of repository mining studies

through a smart data platform," *Empirical Software Engineering*, Aug. 2017.

[42] D. Spadini, M. Aniche, and A. Bacchelli, "PyDriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. New York, New York, USA: ACM Press, 2018, pp. 908–911. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3236024.3264598

[43] Q. Huang, X. Xia, and D. Lo, "Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction," *Empirical Software Engineering*, pp. 1–40, 2018.

[44] X. Yang, D. Lo, X. Xia, and J. Sun, "Tlel: A two-layer ensemble learning approach for just-in-time defect prediction," *Information and Software Technology*, vol. 87, pp. 206 – 220, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584917302501

[45] N. Fenton and J. Bieman, *Software Metrics: A Rigorous and Practical Approach, Third Edition*, 3rd ed. Boca Raton, FL, USA: CRC Press, Inc., 2014.

[46] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[47] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the szz approach for identifying bug-introducing changes," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, 2017.