# Feature-Oriented Defect Prediction

Stefan Strüder*
University of Koblenz-Landau, Germany

Mukelabai Mukelabai
Chalmers | University of Gothenburg, Sweden

Daniel Strüber†
Radboud University Nijmegen, Netherlands

Thorsten Berger
Chalmers | University of Gothenburg, Sweden

## ABSTRACT

Software errors are a major nuisance in software development and can lead not only to reputation damages, but also to considerable financial losses for companies. Therefore, numerous techniques for predicting software defects, largely based on machine learning methods, have been developed over the past decades. These techniques usually rely on code and process metrics in order to predict defects at the granularity of typical software assets, such as subsystems, components, and files. In this paper, we present the first systematic investigation of *feature-oriented* defect prediction: the prediction of defects at the granularity of features—domain-oriented entities abstractly representing (and often cross-cutting) typical software assets. Feature-oriented prediction can be beneficial, since: (i) particular features might be more error-prone than others, (ii) characteristics of features known as defective might be useful to predict other error-prone features, (iii) feature-specific code might be especially prone to faults arising from feature interactions. We present a dataset derived from 12 software projects and introduce two metric sets for feature-oriented defect prediction. We evaluated seven machine learning classifiers with three different attribute sets each, using our two new metric sets as well as an existing metric set from the literature. We observe precision and recall values of around 85% and better robustness when more diverse metrics sets with richer feature information are used.

## CCS CONCEPTS

• **Software and its engineering** → *Software product lines*; **Software defect analysis**; • **Computing methodologies** → **Supervised learning by classification**.

## KEYWORDS

feature, defect, prediction, classification

*Also with Chalmers | University of Gothenburg, Sweden during a research visit for conducting this work.

†Also with Chalmers | University of Gothenburg, Sweden for this work before moving to Radboud University Nijmegen.


*SPLC '20, October 19–23, 2020, MONTREAL, QC, Canada*

ACM ISBN 978-1-4503-7569-6/20/10…$15.00
https://doi.org/10.1145/3382025.3414960

**ACM Reference Format:**
Stefan Strüder, Mukelabai Mukelabai, Daniel Strüber, and Thorsten Berger. 2020. Feature-Oriented Defect Prediction. In *24th ACM International Systems and Software Product Line Conference (SPLC '20), October 19–23, 2020, MONTREAL, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3382025.3414960

## 1 INTRODUCTION

Software errors are a significant cause of financial and reputation damage to companies. Such errors range from minor bugs to serious security vulnerabilities. Therefore, there is a high interest in warning a developer when they release updated software code that may be affected by errors.

To this end, over the past decade, a large variety of techniques for error detection and prediction has been developed, largely based on machine learning techniques [16]. These techniques use historical data of *defective* and *clean* (defect-free) changes to software systems in combination with a carefully compiled set of attributes (usually called *attributes* or *features*[1]) to train a given classifier [5, 28]. This can then be used to make an accurate prediction of whether a new change to a piece of software is defective or clean. The choice of algorithms for classification is large. Studies show that, out of the pool of available algorithms, both tree-based (e.g., J48, CART or Random Forest) and Bayesian algorithms (e.g., Naïve Bayes (NB), Bernoulli-NB or multinomial NB) are the most widely used [69]. Alternatives include logistic regression, k-nearest-neighbors or artificial neural networks [16]. The vast majority of existing work uses these techniques for defect prediction at the granularity of sub-systems, components, and files, and does not come to a definitive consensus on their usefulness—the "best" classifier generally seems to depend on the considered prediction scenario.

In this work, we present a systematic investigation of *feature-oriented defect prediction*—predicting errors on the granularity of software features. Features are a primary unit of abstraction in software product lines and configurable systems [7, 12, 37, 56], but also play a crucial role in agile development processes, where organizations strive towards feature teams and organize sprints around feature requests, for shorter release cycles [42]. Notably, features abstract over traditional software assets (e.g., source files) and often cross-cut them [59], constituting more coherent entities from a domain perspective. Predicting defects at feature granularity is promising for several reasons: First, since a given feature might be historically more or less error-prone, a change that updates the feature may be more or less error-prone as well. Second, features more or less likely to be error-prone might have certain characteristics that can be harnessed for defect prediction. Third, code that

[1]To avoid ambiguity, throughout this paper, we use the term "attribute" instead of "feature" to describe dataset characteristics in the context of machine learning.

contains a lot of feature-specific code (including feature-interaction code [6, 15, 77]) might be more error-prone than others.

We make the following contributions:

- We present a dataset for feature-oriented defect prediction. The dataset is based on twelve projects that we selected due to their usage in previous feature-oriented research [33, 45, 61, 62]. The dataset contains features in specific versions, labeled as either *defective* or *clean*. Feature information was extracted from preprocessor instructions (`#ifdef` and `#ifndef`) in the projects' source code files. The labels were determined using existing automated heuristics targeting file-based defect prediction, which we refined to obtain more accurate results in the considered projects.
- We introduce two metric sets designed for use in the training of machine learning classifiers for feature-oriented defect prediction. The first metric set is comprised of eight feature-based process metrics, whereas the second additionally contains six feature-based structure metrics.
- We present an evaluation of feature-oriented defect prediction, based on our dataset, three metric sets (our proposed two and an existing one) and seven classifiers that were selected due to their frequent usage in the literature. A replication package with all data and code is publicly available in our online appendix [75].

The only previous work investigating feature granularity is a short workshop paper by Queiroz et al. [61]. This earlier work only considered a single software project, a fixed set of five metrics (restricted to process metrics), and three classifiers. In this paper, we consider a significantly greater selection of projects (12), metrics (two fundamentally different sets, 14 in total), and classifiers (7). Furthermore, we took a mitigation measure to deal with the drawbacks of our inherently imbalanced dataset, and empirically compare our results to those produced by using Queiroz et al.'s [61] metric set on our substantially larger dataset. While we focus on systems using annotative variability, in principle, our technique is programming language independent and could be applied to any system with a defined approach for feature extraction.

## 2 BACKGROUND AND RELATED WORK

**Software defect prediction.** Defect prediction is an active research area in software engineering that has been studied for the past five decades [18, 50, 55, 64, 78], with the earliest studies beginning in the 1970s by Akiyama [3], McCabe [47], and Halstead [27], who used code complexity metrics to estimate defects (without machine learning). The vast majority of recent studies relies on machine learning techniques [9, 14, 19, 29, 43, 50, 54] and follows standard procedures of (i) extracting instances (dataset records) from software archives based on the chosen granularity level (e.g., file, class, or method level), (ii) labeling the instances (e.g., as defective or clean) and applying metrics, (iii) optionally applying preprocessing techniques, such as feature selection [66] or normalization [50], and (iv) making predictions for unknown instances—predicting either bug-proneness of source code (classification) or the number of defects in source code (regression).

To characterize the defect-proneness of source code, several metrics have been proposed, including structure and process metrics. While structure metrics generally measure the complexity and size of code, process metrics quantify several aspects of the development process, such as changes of source code, code ownership, and developer interactions. The onset of version control systems has facilitated the application of process metrics to defect prediction [29, 43, 51, 63], which have been demonstrated to outperform structure metrics in many cases [51, 54, 63]. Different measures are used to assess the performance of classification models; the most common being precision, recall, and f-measure (see Section 4.1). However, since most prediction models predict probabilities of defect-proneness, these measures require the use of a minimum probability threshold to declare an instance defective or not. Such performance measures that require the use of threshold values are discouraged [44], since results may vary and are hard to reproduce [49]. A more reliable, threshold-invariant metric is the area under the receiver operating characteristic curve (AUC-ROC). It plots the true positive rate against the false positive rate taking into account all possible threshold values (between 0 and 1). Thus, AUC-ROC indicates how much a prediction model is capable of distinguishing between classes. Furthermore, the area under cost-effective curve (AUCEC) [8, 64] is sometimes used to measure how many defects can be found in the top $n\%$ lines of code so as to provide priorities to quality assurance teams and developers.

Defect prediction models may target quality assurance before product release [63] (a.k.a., *release-based*), or prediction of defects whenever the source code is changed (i.e., predicting bug-inducing changes, a.k.a., *just-in-time (JIT)* defect prediction models [35, 36, 38]). In general, JIT models suffer from insufficient training data. To overcome this limitation for new projects or projects with less historical data, the notion of cross-project defect prediction has been studied as well [31, 55, 79]. To achieve better performance, cross-project predictions generally require careful selection of training data [35, 79], e.g., from similar projects to create a large training dataset, or they require ensembles of models from several projects.

Defect prediction models have been constructed at various granularity levels, including sub-system [22, 29, 39], component/package [44, 54, 78], file/class [46, 50, 55, 57, 79], method [26, 30], and change (hunk) [38] level. Only one study [61] has considered the feature granularity level. Yet, developers commonly use features to develop, maintain, and evolve software systems [7, 12]. In fact, almost all agile software development methodologies, such as SCRUM and XP organize teams, sprints, and releases around features [42]. Therefore, our study investigates defect prediction for features and takes into account feature process and structure metrics to characterize the defect-proneness of features. We use release-based prediction and combine data from several pre-processor-based projects. We rely on the AUC-ROC measure to assess the performance of our selected classification algorithms.

**Machine learning and software product lines.** Defect prediction presents a natural application avenue for machine learning in product line engineering. Most existing work in this area has focused on the sampling of configurations for various use cases; the recent survey by Pereira et al. [60] provides an overview. Focusing on performance predictions, Siegmund et al. [73] use machine learning and sampling techniques to build performance influence models, quantifying the performance impact of specific features and interactions. Temple et al. [74] use machine learning to infer missing product line constraints [53], based on a random sampling

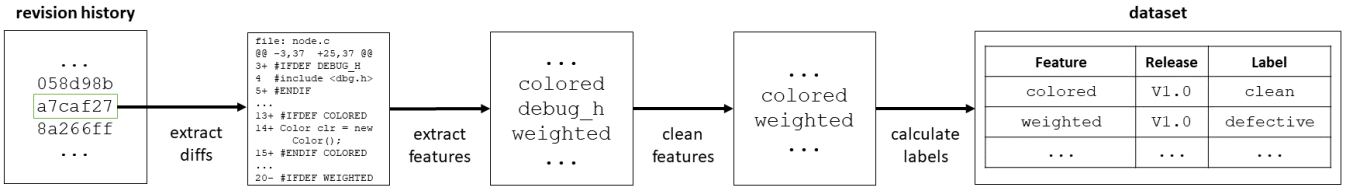



**Figure 1: Dataset creation**

of products and an oracle that assesses whether a particular configuration leads to a valid product. In contrast to us, they are interested in faulty feature combinations, rather than erroneous features. The same authors also investigated the use of learned adversarial configurations in the context of quality assurance [73]. Considering the reverse direction of applying variability concepts to machine learning, Ghofrani et al. [25] propose to investigate product lines of deep neural networks, which establish reuse of existing trained networks by identifying features and composing them. They investigate the reuse potential in an associated empirical study [24].

**Feature metrics.** Several variability-aware feature metrics have been proposed in the literature [11, 21, 45, 58] that measure characteristics of feature specifications (variability models) [37, 56], code, or of the mapping between the feature specification and code artifacts. These metrics target specific variability implementation mechanisms, typically classified [72] into annotative mechanisms—such as the C preprocessor (e.g., `#ifdef` [48])—and compositional mechanisms—such as feature modules (e.g., AHEAD [10]). Using metrics allows conceiving lightweight analysis techniques [52] for systems as complex systems as product lines.

Common among annotation-based metrics are code-related metrics [33, 45, 62] that measure the nesting depth of features, lines of feature code, feature scattering degree (to what extent a feature's implementation is spread across the codebase [59]), and tangling degree (to what extent a feature's implementation is mixed with that of other features, implying feature interactions [6, 15, 77]). We use these four kinds of structural metrics to characterize the defect-proneness of a feature, hypothesizing that the higher the value of each metric is, the more likely a feature is to be defective. All of these existing metrics are structure-based. With the exception of Queiroz et al. 's defect prediction work [61] (which we consider during metric engineering), no existing work proposes dedicated process metrics for features.

## 3 METHODOLOGY

Our methodology comprises: (i) creating a dataset of feature labels over the history of 12 software projects; (ii) creating two new metric sets designed for feature-oriented defect prediction; and (iii) selecting and training seven classifiers we considered for our evaluation.

### 3.1 Dataset Creation

Our study aims to create a machine learning model for predicting the defect-proneness of software features. To this end, we created training and test datasets whose instances are features, in contrast with commonly used granularity levels such as components, files or methods. For this purpose, we relied on software projects with available revision histories, and use preprocessor macros (e.g., `#ifdef`) to annotate source code with features. We followed the process outlined in Figure 1 to extract feature references (by pattern matching preprocessor macros) in files that were modified during commits, and labeling these features as defective or clean based on whether one or more files implementing each feature was identified to be defective. Below, we describe this process in more detail.

**Software projects.** We generated datasets based on data from the full revision histories of 12 preprocessor-based software projects—these projects have been subjects of prior research on features and software product lines [45, 61**?** , 62]. From these papers, we obtained an initial set of 44 projects, which we filtered by applying the following inclusion criteria: First, the project's source code uses preprocessor directives as variability mechanism. Second, meta-data on release versions is available in the form of several tags specifying release versions. Third, the project has a nontrivial (greater than 5) number of features. Fourth, the project's commit messages are given in English—a prerequisite for the heuristics we used for detecting bug-fixing commits. We checked these criteria manually, yielding a selection of 12 projects, which we list in Table 1, together with context, repository sources, and additional information.

**Retrieval.** To retrieve our subject projects' revision histories, we used the library PyDriller [70]). It allows easy data extraction from Git repositories to obtain commits, commit messages, commit authors, diffs, and more (called "metadata" in the following). To this end, we created Python scripts for receiving the commit metadata, including the release number to which each commit belonged.

For each modified file within a commit, we collected metadata, such as *commit hash (unique commit identifier), commit author, commit message, filename, and diff (changeset)*, that we used for calculating metrics (Section 3.2) and labeling of instances in our datasets. This metadata was saved in a MySQL database, available as part of our online appendix [75]. For each of our subject projects, we create a separate table in the database in which we store the above metadata for each file, including the name of the project and the release number associated with the commit in which the file was changed.

**Feature reference extraction and cleaning.** Using regular expressions, we extracted feature references in each modified file within a commit changeset, by pattern-matching the preprocessor macros `#ifdef` and `#ifndef`. Combinations of features (e.g., `#ifdef A & B`) are stored in their identified form.

This way of identification has some obstacles. In some C programming paradigms, it is common to include header files in the source code using preprocessor directives, in the same way as features. However, we ignored these "header macros", as they will be referred to in the remainder, as they do not represent actual features. In general, these header macros are identifiable through their suffix `_h_` to the name, such as `macroname_h_`. Through a manual

**Table 1: Subject systems**

| project | description | corrective commits | bug-introducing commits | features | training-set releases | test-set releases | split ratio[1] | URL |
|---|---|---|---|---|---|---|---|---|
| **Blender** | 3D-modeling tool | 7,760 | 3,776 | 1,400 | 2.70 - 2.77 | 2.78 - 2.80 | 73 : 27 | github.com/sobotka/blender |
| **Busybox** | UNIX toolkit | 1,236 | 802 | 628 | 1_16_0 - 1_25_0 | 1_26_0 - 1_30_0 | 71 : 29 | git.busybox.net/busybox/ |
| **Emacs** | text editor | 4,269 | 2,532 | 718 | 25.0 - 26.0 | 26.1 - 26.2 | 71 : 29 | github.com/emacs-mirror/emacs |
| **GIMP** | graphics editor | 1,380 | 854 | 204 | 2_8_2 - 2_10_4 | 2_10_6 - 2_10_12 | 71 : 29 | gitlab.gnome.org/GNOME/gimp |
| **Gnumeric** | spreadsheet | 1,498 | 1,191 | 637 | 1_10_0 - 1_12_10 | 1_12_20 - 1_12_30 | 75 : 25 | gitlab.gnome.org/GNOME/gnumeric |
| **gnuplot** | plotting tool | 854 | 1,215 | 558 | 4.0.0 - 4.6.0 | 5.0.0 | 80 : 20 | github.com/gnuplot/gnuplot |
| **Irssi** | IRC client | 52 | 22 | 9 | 1.0.0 - 1.0.4 | 1.0.5 - 1.0.6 | 71 : 29 | github.com/irssi/irssi |
| **libxml2** | XML parser | 324 | 88 | 200 | 2.9.0 - 2.9.7 | 2.9.8 - 2.9.9 | 80 : 20 | gitlab.gnome.org/GNOME/libxml2 |
| **lighttpd** | web server | 1,078 | 929 | 230 | 1.3.10 - 1.4.20 | 1.4.30 - 1.4.40 | 67 : 33 | git.lighttpd.net/lighttpd/lighttpd1.4.git/ |
| **MPSolve** | polynom solver | 151 | 211 | 54 | 3.0.1 - 3.1.5 | 3.1.6 - 3.1.7 | 75 : 25 | github.com/robol/MPSolve |
| **Parrot** | virtual machine | 3,109 | 3,072 | 397 | 1_0_0 - 5_0_0 | 6_0_0 - 7_0_0 | 71 : 29 | github.com/parrot/parrot |
| **Vim** | text editor | 371 | 696 | 1,158 | 7.0 - 7.4 | 8.0 - 8.1 | 71 : 29 | github.com/vim/vim |

[1] percentage of training and test releases

review of the identified feature references, we also ignored feature references when the preprocessor directives occurred in comments.

**Label calculation.** For each identified feature in each revision, we calculated a label, specifying if the feature is *defective* or *clean*. To this end, we relied on a common automated heuristic for identifying corrective and bug-introducing commits [80]. We modified it for our purpose and mapped the results to features, as explained below.

The heuristic scans commit messages for the presence of the keywords "bug," "bugs," "bugfix," "error," "fail," "fix," "fixed," and "fixes." In a manual inspection of the results, we noticed many false positives. Especially in lengthy commit messages, we noticed an increased probability that our keywords are used in an irrelevant context, e.g., handling of "fixed fonts" in the implementation of *emacs*. We modified the heuristic to only consider the first line of each commit since the main purpose of the commit is usually stated in the first line or sentence. We then took a sample of about 50-100 commits per project (approx 500 in total) to evaluate the modified heuristic, and found that it decreased the number of false positives significantly. However, as a general limitation of our technique (similar to other techniques used in defect prediction studies) we do not guarantee that a commit does not have bugs, but instead focus on confirmed bugs specified by developers.

We used the corrective commits to identify the corresponding bug-introducing commits. The state-of-the-art algorithm for this purpose is the SZZ algorithm according to Sliwerski, Zimmermann and Zeller [68, 70], which uses heuristics to identify the commits in which the lines leading to the later-fixed bug have been introduced. We used the available SZZ implementation of PyDriller. Table 1 gives an overview of the number of corrective and bug-introducing commits and the number of features identified per project.

Finally, for labeling, we first compute labels for files, and then use these labels to calculate the labels for associated features. A file is labeled as *defective* in a particular release if there is at least one bug-introducing commit that changes the file, and as *clean* otherwise. A feature is labeled as *defective* in a particular release if it is associated with at least one defective file, and as *clean* otherwise. Corrective commits are not reflected directly in labels, since we are interested in the error-proneness of particular features. Key figures giving an overview of the created dataset are listed in Table 2.

**Table 2: Key characteristics of the dataset**

| instances | defective | clean |
|---|---|---|
| 14,735 | 2,988 | 11,747 |

Figure 3 shows the diffs of a corrective (A) and a bug-introducing (B) commit to a feature `FEAT_TEXT_PROP` from the project *Vim*. The diff of commit A shows that the arguments of the method call `vim_memset` have been replaced. According to the associated commit message, the original method call caused a "memory access error." Commit A was, therefore, identified as corrective because the commit message contains the keyword "error." To identify the bug-introducing commit B of the file concerned, we specify the hash of the corrective commit A to the SZZ algorithm. In its portion of the diff, we can see that commit B has put the feature `FEAT_TEXT_PROP` in the file with the incorrect method call. Consequently, we consider the commit to be bug-introducing, and the associated file and feature to be defective in that particular release.

## 3.2 Selection of Metrics

Selecting an effective set of attributes for classifier training (a.k.a., *feature engineering*) is commonly considered the decisive factor for the success or failure of machine learning applications [20]. To reflect this crucial role, we iteratively designed a suitable set of attributes, following a design-science approach [32].

**Metrics as attributes.** To map the available feature information to attributes, we need to design a set of software metrics—numerical values that quantify properties of a software project. We consider both *structure metrics*, which are used to measure certain qualities of the software code of a specific revision, and *process metrics*, which are used to measure properties of metadata taken from software

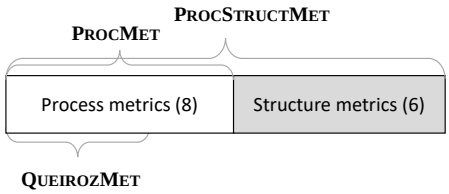



**Figure 2: Metric sets**

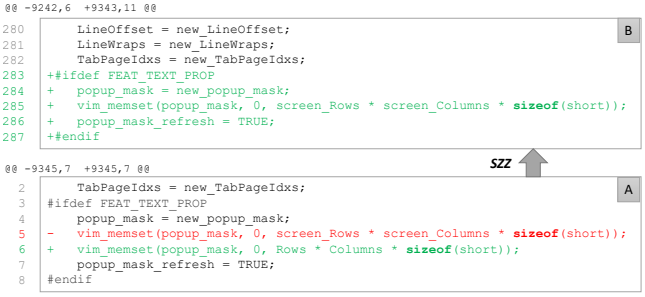



**Figure 3: Example of a defect with corrective (A) and bug-introducing (B) commit**

repositories [63], or take the evaluation of characteristics over revisions into account. In the context of features, an example structure metric is: scattering degree (counting all preprocessor macro references to the feature—e.g., `#ifdef A`). An example process metric is: the number of committers who changed the feature in the release.

**Metric sets.** We obtained three metric sets: an existing one from the literature (QueirozMet) and two new metric sets (ProcMet, ProcStructMet) we obtained by incrementally refining the available metrics. ProcMet augments QueirozMet with additional process metrics, whereas ProcStructMet extends ProcMet with structure metrics. Figure 2 illustrates the metric sets and their relationships. Table 3 gives a detailed overview of the resulting fourteen metrics and their descriptions.

- **QueirozMet**: The original metric set by Queiroz et al. [61] consists of five process metrics, based on the rationale that process metrics are deemed particularly beneficial in defect prediction [63]. The included metrics quantify basic information such as the number of commits associated with the feature, developers contributing to the feature's implementation, and the experience of these developers (based on previous involvement).
- **ProcMet**: In the first iteration, we systematically investigated additional process metrics. In the absence of dedicated feature process metrics in the literature (see Section 2), we derived three new ones from existing non-feature process metrics. These metrics quantify more involved feature-related process information, such as the average number of lines of code added to the files associated with the feature in the release. The original, file-based versions of these metrics were assessed as beneficial for defect prediction in earlier work [63]. Consequently, we obtained a new metric set ProcMet, consisting of eight process metrics.
- **ProcStructMet**: In the second iteration, we systematically investigated feature structure metrics, aiming to benefit from two complementary types of metrics by bundling them. We added six structure metrics: four custom feature structure metrics identified in related work (see Section 2), and two new ones we derived from particularly common structure metrics. The custom metrics include metrics such as the nesting depth of a feature (number of other preprocessor macros nested within the macro of a given feature). The newly derived metrics, based on LoC and cyclomatic complexity, represent the two main dimensions usually considered by structure metrics: size and complexity. Overall, our metric set ProcStructMet comprises eight process and six structure metrics.

Generally, the metric values for each feature are aggregated over a release as described in Table 3. The values of the metrics are calculated for the data of each subject project, in some cases directly using SQL queries, in some cases by combining SQL queries and a Python script, and in other cases by using an available tool.

## 3.3 Selection and Training of Classifiers

We selected seven classifiers based on their use in previous studies. Table 4 provides an overview. A key informative work for our selection was the empirical study by Son et al. [69], who determine the six most commonly used classifier types in 156 defect-prediction studies: Decision Tree, Random Forest, Bayesian, Regression, Support Vector Machines and Neural Networks. We used typical representative learners for each of the broader categories: J48 (Decision Tree), LR (Regression), NB (Bayesian). As an example for learners that are commonly used in classification, but less so in defect prediction, we included k-Nearest Neighbor (KNN).

**Tool and configuration.** To train and test our classifiers, we used the WEKA workbench[2] due to its widespread application in scientific studies, including defect prediction [28, 61, 65]. WEKA offers a large collection of machine learning algorithms and preprocessing tools for use via a graphical user interface. All classification algorithms presented above are already integrated in the WEKA tool. WEKA takes as input the dataset (saved in `CSV` or the proprietary `ARFF` format) and executes it against selected classifiers.

We trained each classification algorithm in WEKA with the respective standard settings, except for NN and RF. For RF, we set the number of decision trees for parallel processing to 200. There are no clear recommendations on how many trees should be specified. We, therefore, select the value of 200 independently, taking into account the scope of the datasets and the high number of attributes. For the NN algorithm, we independently specify a hidden layer structure of `(13,13,13)`. This means that the artificial neural network has three hidden layers of 13 hidden layer neurons each. This allows them to process the large number of attributes more efficiently.

We trained each of these seven classifiers using the dataset with each of the three metric sets, leading to 21 instances of training in total. We used a Windows 10 system (Intel Core i7-6500U, 16GB RAM) for all experiments. Depending on the metric set, the training times (given in Table 5) were between a few seconds and a bit more than a minute for the entire dataset. Specifically, the longest time taken was for NN (between 52 and 65 seconds) and the shortest was for KNN (1 to 2 seconds).

The results obtained using the test data, which reflect the performance of the individual classifiers, are presented in the following section as part of the evaluation.

**Test vs. training set.** Before conducting the training, we determined the ratio of training data to test data for each individual project based on the number of available releases. We aimed to approximate the commonly used split ratios of between 80 : 20% and 70 : 30%. The resulting ratio splits, as shown in Table 2, range from 67 : 33% to 80 : 20%.

In general, we assigned earlier releases to the training data and later ones to test data. In doing so, we avoid the implausible situation of "using the future to predict the past", which is unrealistic in

[2] https://www.cs.waikato.ac.nz/ml/weka/

**Table 3: Metrics of metric sets ProcMet (process metrics) and ProcStructMet (process and structure metrics)**

**Legend**: Let $R = \{c_1, c_2, ...c_q\}$ be a release set consisting of $q$ commits; $C$ be the set of all commits from previous releases plus those in $R$; $F = \{f_1, f_2, ...f_p\}$ be the set of all files changed by commits in $R$. Let $T = \{feat_1, feat_2, ...feat_n\}$ be the set of all features affected by changes in $R$ (i.e., features included in diffs), where each feature $feat \in T$ has a set $A = \{featfile_1, featfile_2, ...featfile_m\}$ of files implementing it, and $A \subseteq F$. We define our metrics for each feature *feat*, with respect to release $R$ as follows:

| | metric[1] | description | function signature |
|---|---|---|---|
| process metrics | *FCOMM* | Count of all commits in which a feature was changed within a release. | $comm(feat, R)$ |
| | *FADEV* | Count of all developers who changed a feature within a release. | $adev(feat, R)$ |
| | *FDDEV* | Count of all distinct developers who changed the feature up to the current release | $ddev(feat, C)$ |
| | *FEXP*[2] | Average experience[3] of all developers who changed a feature within a release. | $exp(feat, R)$ |
| | *FOEXP* | Average experience of the developer who changed the features of a file most often within a release. | $oexp(feat, R)$ |
| | *FMODD* | Average scattering degree of a feature in changesets within a release—counts number of #ifdef references to a feature within each changeset and averages this over the release | $modd(feat, R)$ |
| | *FADDL* | Average number of lines of code added to the files associated with a feature within a release. | $addl(feat, A)$ |
| | *FREML* | Average number of lines of code deleted from the files associated with a feature within a release. | $reml(feat, A)$ |
| structure metrics | *FNLOC* | Average number of lines of code of the files associated with a feature within a release. | $nloc(feat, A)$ |
| | *FCYCO* | Average cyclomatic complexity of the files associated with a feature within a release. | $cyco(feat, A)$ |
| | *LOFC* | Number of lines of code associated with a feature in a release (calculated from the last commit in $R$) | $lofc(feat, c_q)$ |
| | *NDEP* | Maximum nesting depth of #ifdef directives that the feature is involved in (calculated from the last commit in $R$) | $ndep((feat, c_q)$ |
| | *SCAT* | Scattering degree of a feature—count of all #ifdef references to the feature (calculated from the last commit in $R$) | $scat(feat, c_q)$ |
| | *TANGA* | Tangling degree of a feature—count of all other features mentioned the #ifdef reference as the feature, e.g., *#ifdef featA & featB* (calculated from the last commit in $R$) | $tang(feat, c_q)$ |

[1] The first five process metrics (FCOMM, FADEV, FDDEV, FEXP, and FOEXP) were introduced by Queiroz et al. [61] with a slight modification in names; here we prefixed them with F to indicate that they are calculated over features unlike commonly done with files e.g., by Rahman et al. [63]. We refer to this set of metrics as QueirozMet.

[2] $exp(feat, R)$ returns the geometric mean of the experience[3] of all developers who changed the feature within a release.

[3] Experience is the sum of the changed, deleted or added lines in the commits associated with the files (set $A$) implementing *feat*.

practice [34]. For the same reason, we do not use cross-validation to evaluate our selected classifiers.

**Inbalanced dataset.** Table 2 reveals that our dataset is imbalanced: *clean* instances outnumber *defective* ones by a factor of 4.14. Using imbalanced datasets for training is generally known to skew the classifier towards misclassification of the under-represented class. A common mitigation strategy is to apply over-sampling, by generating synthetic examples of the minority class. To this end, we apply the SMOTE [17] algorithm to our training dataset by using the available implementation in WEKA, in its standard configuration.

## 4 EVALUATION

Using the classifiers we trained based on our dataset (see subsection 3.1) and the three considered metric sets (see subsection 3.2), we studied two research questions:

- **RQ1**: What is the effect of using different types of feature metrics (structural and process) on prediction quality?
- **RQ2**: Which particular feature metrics contribute most strongly to prediction quality?
- **RQ3**: What is the effect of using different classifiers on prediction quality?

**Table 4: Selection of classification algorithms**

| classifier | abbreviation |
|---|---|
| J48 Decision Trees | J48 |
| k-Nearest-Neighbors | KNN |
| Logistic Regression | LR |
| Naïve Bayes Bayes | NB |
| Artificial Neural Networks | NN |
| Random Forest | RF |
| Support Vector Machines | SVM |

Within RQ1, we implicitly compare our contribution to the most closely related work: our two new metric sets are compared to the one from Queiroz et al.
,[61], who proposed the only other dedicated metric set for feature-oriented defect prediction.

In what follows, first, we present our evaluation metrics, second, we present our results and discuss their implications.

### 4.1 Evaluation Metrics

To compare the classifiers with regard to prediction quality, we consider two types of evaluation metrics, commonly used for this purpose in the field of information retrieval [1]. First, *precision*, *recall*, and *F-score*, which quantify information about the percentage of true and false predictions, based on an available *confusion matrix*. Second, *receiver operating characteristic* (ROC) curves and the associated *area under curve* (AUC), which provide a visual and more robust way for assessing prediction quality than confusion-matrix-based metrics.

All our evaluation metrics assume a *ground truth*, specifying for each given class the entries that belong to it (positives) and those that do not (negatives). In our case, entries are features with regard to a given release. The classes are *defective* and *clean*. The ground truth was constructed in the labeling step during dataset construction (see Sect. 3.1).

**Recall, Precision, and F-score.** We follow the standard definition of precision, recall, and F-score. Intuitively, recall quantifies how exhaustively the classifier identified all entries of the class, comprised of true positives (TP) and false negatives (FN), respectively. Precision quantifies the percentage of true positives (TP) among all entries assigned to a particular class (also including false negatives, FN). The F-score is the harmonic mean of precision and recall, representing a balance between both. In contrast to other confusion-based-matrix (e.g., accuracy), these metrics are considered as useful on imbalanced datasets, such as ours. These metrics are computed as follows:

$$\text{Recall} = \frac{TP}{TP+FN} \quad \text{Precision} = \frac{TP}{TP+FP} \quad \text{F-score} = \frac{2TP}{2TP+FP+FN}$$

**ROC-AUC.** We determined the ROCs and AUCs of the individual classifiers. These have the benefit that they represent performance in a visual, understandable way, while at the same time making the quality assessment more robust: Precision, recall, and F measure depend on a predefined threshold, which is used in the classifiers to assign each instance to a class. A robust classifier shows good predictive ability regardless of the chosen threshold value.

ROC curves encode this intuition, by describing the relationship between the TP rate (a.k.a. recall, y axis) and the FP rate (x axis), indicating the proportion of predictions that are incorrectly evaluated as positive [1, 4]. The FP rate is calculated as follows:

$$\text{FP rate} = \frac{FP}{FP+TN}$$

Datapoints on the curve are obtained by taking into account all possible values for the threshold that determines when an instance is assigned to a particular class.

The AUC area indicates the extent to which a classifier is able to make correct predictions under a changing threshold value. The higher this value is, the more robust the classifier is in making correct predictions. The ideal value is 1.0, whereas a value of 0.5 indicates a predictive ability on the same level as random guessing.

## 4.2 Results

Table 7 and Fig. 4 in combination give an overview of our results. Table 7 provides all calculated precision, recall, F-score, and AUC values. For each classifier and evaluation metric, the top value (best-perfoming metric set) is highlighted in bold. Figure 4 shows ROCs for three representative classifiers (top performer, average performer, worst performer in terms of AUC) in combination with all three metrics sets. For reference, AUC values of all cases are shown in the table.

**Table 5: Training times per metric set (in seconds)**

| | QueirozMet | ProcMet | ProcStructMet |
|---|---|---|---|
| J48 | 0.44 | 0.24 | 0.46 |
| KNN | 0.01 | 0.02 | 0.01 |
| LR | 0.29 | 0.09 | 0.14 |
| NB | 0.03 | 0.03 | 0.03 |
| NN | 51.85 | 53.13 | 65.34 |
| RF | 10.89 | 6.04 | 5.83 |
| SVM | 0.6 | 0.75 | 1.65 |

**RQ1: Effect of metric sets.** Based on precision, recall, and F-score, we generally observe a moderate tendency that classifiers performed best when using ProcStructMet, for which we observed weighted averages between 0.66–0.85, 0.70–0.85 and 0.68–0.83 respectively. The corresponding ranges for the case of ProcMet and QueirozMet are 0.58–0.84, 0.70–0.83 and 0.63–0.82, and 0.55–0.84, 0.71–0.84 and 0.61–0.82, respectively. The quality difference is particularly pronounced when considering the top values (printed in bold): In all classifiers except for SVM, ProcStructMet shows the top value for precision, recall, and F-score. Two noteworthy observations are the case of NB, where all evaluation metrics take the same values over all metrics sets, and SVM, where QueirozMet outperforms ProcMet and ProcStructMet. Considering the two classes *clean* and *defective*, we generally find higher F-scores in the more advanced metric sets, and a better ability to predict clean than defective instances for all metric sets.

Considering ROCs and AUCs sheds light on the effect of the metric sets on robustness. We generally find a clear tendency of ProcStructMet to highest robustness, i.e., more stability with regard to different values for the threshold used for assigning instances to classes (reflected by a steeper initial incline in the ROC curves). In 5 out of 7 cases, the AUC for ProcStructMet shows a solid value between 0.74 and 0.82. The AUC for ProcStructMet is consistently greater or equal to that of ProcMet in some cases strongly so, including the top performer NN (0.79 vs. 0.61). The highest achieved value for QueirozMet is 0.64. The SVM classifier is an exception to all other cases: for ProcMet and ProcStructMet we observe worse performance (0.49) than from random guessing (0.5); corresponding to a nearly-linear ROC. A possible explanation for the preferable robustness of ProcStructMet in most classifiers is the availability of more diverse metrics, providing a richer information source for predictions.

**RQ2: Effect of individual metrics.** We determined the effect of individual metrics by applying an attribute selection method. Such methods heuristically determine the effect of attributes, in our case metrics, with regard to a classifier's predictive ability. We used a standard method provided by Weka (`weka.attributeSelection.ClassifierAttributeEval` together with the `Ranker` class). This method runs the considered classifier several times with different subsets of the entire metric set, and outputs an influence measure between 1.0 and -1.0 for each considered metric, quantifying the influence of the metric to the prediction result. We applied the method to all 21 classifier instances (7 classifiers with 3 metric sets).

We present an overview of the results in Table 7, showing the three top performers from RQ1 and the average over all 21 classifier instances. The most influential metric for each classifier is highlighted in bold. Generally, the obtained values are very similar for RF and J48, perhaps unsurprisingly, since RF and J48 are both based on the decision tree paradigm. For NN, only three non-zero values are reported, which, however, agree with the reported values for the other two top classifiers. We observe striking cases of large standard deviations, most pronounced in the case of FADDL, which has the most positive impact for the RF classifier (0.058), while, on average, leading to a strong negative influence (-0.03). Despite the observation in RQ1 that the inclusion of structure metrics leads to improved results compared to only process metrics, the effect

**Table 6: Results RQ1 and RQ3: evaluation metrics for the classes "defective" and "clean," and the weighted average "w.a."**

| | | Metric set | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | QueirozMet | | | ProcMet | | | ProcStructMet | | |
| **Classifier** | **Eval. metric** | defective | clean | w.a. | defective | clean | w.a. | defective | clean | w.a. |
| J48 | Recall | 0.57 | 0.66 | 0.64 | 0.61 | **0.85** | 0.80 | **0.65** | **0.85** | **0.81** |
| | Precision | 0.27 | 0.87 | 0.77 | 0.47 | 0.91 | 0.83 | **0.49** | **0.92** | **0.84** |
| | F score | 0.37 | 0.75 | 0.68 | 0.53 | **0.88** | 0.81 | **0.56** | **0.88** | **0.82** |
| | AUC area | 0.57 | 0.57 | 0.57 | **0.79** | **0.79** | **0.79** | 0.78 | 0.78 | 0.78 |
| KNN | Recall | 0.53 | 0.56 | 0.55 | **0.57** | 0.58 | 0.58 | 0.55 | **0.81** | **0.77** |
| | Precision | 0.21 | 0.84 | 0.73 | 0.23 | 0.86 | 0.75 | **0.39** | **0.89** | **0.80** |
| | F score | 0.30 | 0.67 | 0.61 | 0.33 | 0.69 | 0.63 | **0.46** | **0.85** | **0.78** |
| | AUC area | 0.50 | 0.50 | 0.50 | 0.52 | 0.52 | 0.52 | **0.74** | **0.74** | **0.74** |
| LR | Recall | 0.40 | **0.73** | **0.67** | 0.43 | 0.72 | **0.67** | **0.45** | 0.72 | **0.67** |
| | Precision | 0.25 | **0.85** | 0.74 | 0.25 | **0.85** | 0.74 | **0.26** | **0.85** | **0.75** |
| | F score | 0.30 | **0.78** | **0.70** | 0.32 | **0.78** | **0.70** | **0.33** | **0.78** | **0.70** |
| | AUC area | **0.64** | **0.64** | **0.64** | 0.60 | 0.60 | 0.60 | 0.60 | 0.60 | 0.60 |
| NB | Recall | 0.38 | **0.94** | **0.84** | **0.40** | 0.93 | **0.84** | 0.37 | **0.94** | **0.84** |
| | Precision | **0.58** | 0.87 | **0.82** | 0.57 | **0.88** | **0.82** | 0.57 | 0.87 | **0.82** |
| | F score | **0.50** | **0.91** | **0.82** | 0.47 | 0.90 | **0.82** | 0.45 | 0.90 | **0.82** |
| | AUC area | 0.61 | 0.61 | 0.61 | 0.77 | 0.77 | 0.77 | **0.78** | **0.78** | **0.78** |
| NN | Recall | 0.28 | 0.75 | 0.66 | 0.30 | 0.75 | 0.67 | **0.33** | **0.97** | **0.85** |
| | Precision | 0.20 | 0.82 | 0.71 | 0.21 | 0.83 | 0.72 | **0.69** | **0.87** | **0.84** |
| | F score | 0.23 | 0.78 | 0.68 | 0.25 | 0.79 | 0.69 | **0.45** | **0.92** | **0.83** |
| | AUC area | 0.55 | 0.55 | 0.55 | 0.61 | 0.61 | 0.61 | **0.79** | **0.79** | **0.79** |
| RF | Recall | 0.57 | 0.63 | 0.62 | 0.62 | 0.83 | 0.80 | **0.68** | **0.85** | **0.82** |
| | Precision | 0.26 | 0.87 | 0.76 | 0.45 | 0.91 | 0.83 | **0.51** | **0.92** | **0.85** |
| | F score | 0.35 | 0.73 | 0.66 | 0.52 | 0.87 | 0.81 | **0.58** | **0.89** | **0.83** |
| | AUC area | 0.59 | 0.59 | 0.59 | 0.75 | 0.75 | 0.75 | **0.82** | **0.82** | **0.82** |
| SVM | Recall | 0.12 | **1.00** | **0.84** | 0.22 | 0.76 | 0.66 | **0.23** | 0.76 | 0.66 |
| | Precision | **0.83** | **0.84** | **0.84** | 0.17 | 0.82 | 0.70 | 0.17 | 0.82 | 0.70 |
| | F score | **0.21** | **0.91** | **0.78** | 0.19 | 0.79 | 0.68 | 0.20 | 0.79 | 0.68 |
| | AUC area | **0.56** | **0.56** | **0.56** | 0.49 | 0.49 | 0.49 | 0.49 | 0.49 | 0.49 |

of each individual structure metric is moderate compared to the process metrics. This indicates that structure metrics seem to play a non-negligible, but supplementary role for the observed results.

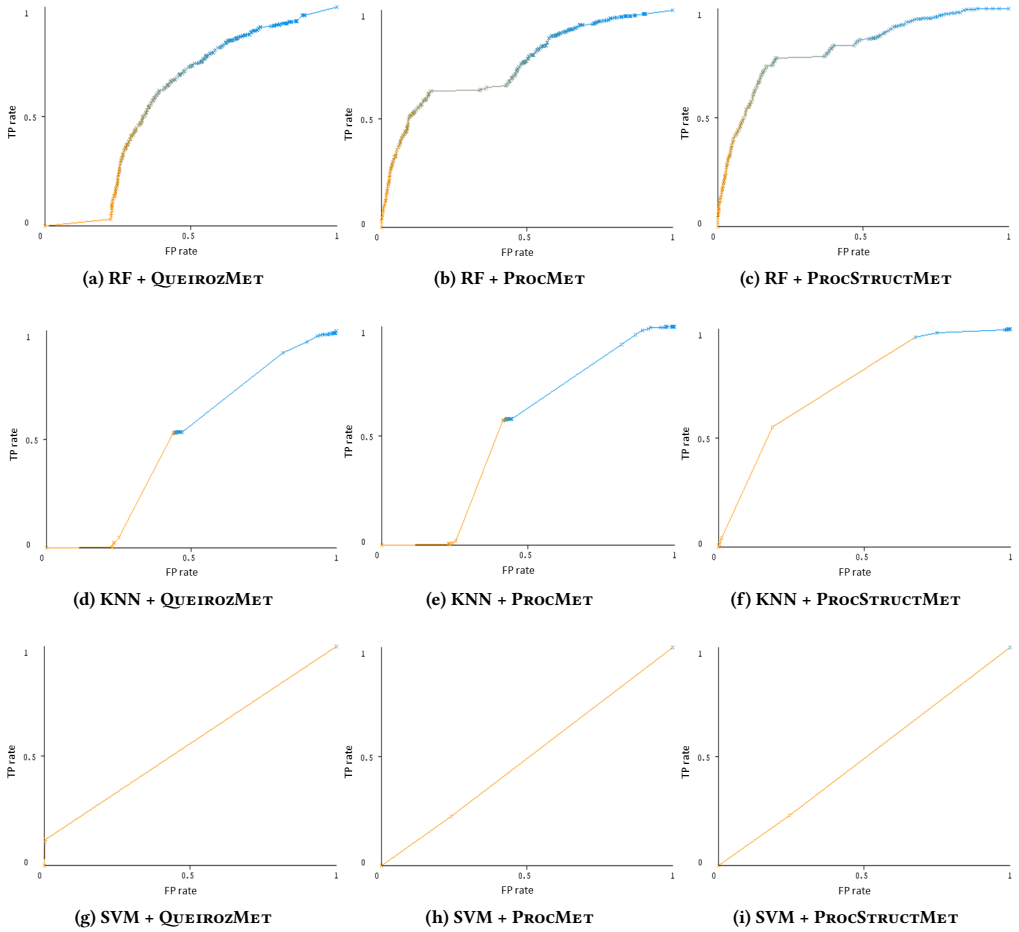
**RQ3: Effect of classifiers.** As a general observation, in most cases, the prediction quality of the same classifier varied strongly based on the considered metrics set (see RQ1). It is, therefore, more meaningful to compare combinations of classifiers and metrics, rather than classifiers alone. Considering weighted averages, we observe values for precision between 0.70–0.85, for recall between 0.62–0.85, and for F-measure between 0.61–0.83. The best-performing classifiers with regard to F-measure were NN and RF, both in combination with ProcStructMet showing an F-measure of 0.83. NN also shows the best average-weighted recall (0.85), while RF shows the top average-weighted precision (0.85). With a value of 0.82 for ProcStructMet, J48 also achieved above-average performance.

Considering individual classes, similar to the comparison between metrics, the results for the label *defective* are generally worse than those of the label *clean*. The best precision for predicting clean files, 0.83, is observed for SVM in combination with QueirozMet. However, this value is traded off for the worst observed recall for that label (0.12). Note that the seemingly contra-intuitive average score of 0.78 for this combination results from averaging over the individual F-scores for class labels (0.21 and 0.91).

Considering ROC curves and AUC areas, we again observe a large inter-classifier variability. Still, the two top classifiers with regard to precision, recall, and F-balance also have the two highest observed AUC values: RF with 0.82, and NN with 0.79, indicating that these classifiers a good robustness while ensuring high predictive ability. An interesting observation is that the minimal AUC value achieved per classifier was never higher than 0.61, close to random guessing. In contrast, in four out of seven cases, a maximal

**Table 7: Results RQ2: Influence of metrics for top classifiers (RF, NN, J48 with ProcStructMet) and all classifiers**

| | | | | | All classifiers | |
|---|---|---|---|---|---|---|
| | **Metric** | **RF** | **NN** | **J48** | mean | std.dv |
| process metrics | *FCOMM* | 0.048 | 0 | **0.049** | **0.034** | 0.02 |
| | *FADEV* | 0.025 | **0.024** | 0.025 | 0.022 | 0.004 |
| | *FDDEV* | 0.009 | 0.008 | 0.01 | 0.009 | 0.003 |
| | *FEXP* | 0.047 | 0 | 0.018 | 0.006 | 0.036 |
| | *FOEXP* | 0.057 | 0 | 0.036 | 0.021 | 0.026 |
| | *FMODD* | 0.042 | 0 | 0.041 | 0.028 | 0.019 |
| | *FADDL* | **0.058** | 0 | 0.035 | -0.03 | 0.142 |
| | *FREML* | 0.03 | 0 | 0.016 | 0.011 | 0.014 |
| structure metrics | *FNLOC* | 0.014 | 0 | 0.002 | 0.004 | 0.006 |
| | *FCYCO* | 0.013 | 0 | 0.007 | 0.004 | 0.005 |
| | *LOFC* | 0.002 | 0 | 0 | 0 | 0.001 |
| | *NDEP* | 0.002 | 0 | 0 | 0 | 0.001 |
| | *SCAT* | 0.005 | -0.001 | 0 | 0.001 | 0.002 |
| | *TANGA* | 0 | 0 | 0 | 0 | 0 |




(a) RF + QueirozMet (b) RF + ProcMet (c) RF + ProcStructMet

(d) KNN + QueirozMet (e) KNN + ProcMet (f) KNN + ProcStructMet

(g) SVM + QueirozMet (h) SVM + ProcMet (i) SVM + ProcStructMet

**Figure 4: Results RQ1 and RQ3: ROC curves for three selected classifiers (top, average, worst)**

value of 0.78 could be observed – illustrating again that the choice of metric set is a key decisive factor for the success or failure of a particular predictor.

**Summary.** We find a strong effect of metric selection on the classifier performance. Remarkably, some classifiers become only feasible alternatives when used in combination with a suitable metric set. In most cases, we notice that considering a greater selection of more diverse metrics (that we introduce in this paper) lead to improved performance, including the identified top performers NN and RF. However, this tendency does not apply to all cases: a remarkable counterexample are SVMs, where the predictive ability declines with the availability of more metrics. Hence, identifying a metric set that improves the performance of the considered classifier appears to be a key prerequisite to successful adoption of machine learning techniques for feature-based defect prediction.

## 4.3 Threats to Validity

**External validity.** To mitigate overfitting of our models, a key threat in machine learning, we used the typical separation of the dataset into test data and training data [20]. Another standard technique to mitigate overfitting, cross validation, is not applicable to scenario, since it leads to the problematic situation of "using the future to predict the past", which is unrealistic for practical applications [34]. While providing improved techniques for avoiding overfitting in defect prediction contexts is an open research issue, we observe good predictive ability for 12 systems of diverse context and size, which gives us some confidence that our models are not severely affected by overfitting.

Despite diversity and size of the projects in our dataset, studying a broader selection of software projects is desirable, as it would increase the generalizability of our findings. We intend to aggregate larger datasets in future work, which would also contribute to our ongoing community initiative towards more mature benchmarks for techniques in the context of evolving variant-rich systems [71].

**Internal validity.** As observed in Section 3.1, we observed that some of our automatically retrieved features were not meaningful, as they represented "header features", in the style of a certain C pattern. While we manually processed all identified features to remove header features, it is possible that in some projects header features are not explicitly identified by names. A possible solution could be enabled by a tool that automatically analyses the code to detect header features. Such a tool does not exist at the moment. Still, the manual removal of the recognizable header features allowed us to

reduce the amount of these noise datapoints. Furthermore, we take a conservative approach by considering only features referenced through `#ifdef`and `#ifndef`. We also do not explicitly exclude standard predefined macros such as `__FILE__`, `__LINE__`, etc., since we treat them as features with associated code if they are referenced through our selected preprocessor macros above.

During dataset creation, we rely on a mapping from all features changed in a particular release to the associated files. This mapping is obtained from analyzing all commit change sets within the release. Thus, a feature is considered relevant if it is mentioned in a diff (either within a changed line or in the context provided with change lines, which, per default, extends to three lines before and after changed lines). This heuristic is subject to imprecision related to preprocessor macros outside the provided context. Extending the implementation to take into account all files is subject to future work.

**Construct validity.** Our ground truth for the identification of defective and clean features relies on an available heuristic technique, the SZZ algorithm. An associated threat is concerned with possible imprecisions of this algorithm. According to a recent study [76], available implementations of SZZ, including those of PyDriller, can identify only about 69% of all bug-introducing commits. In addition, about 64% of the identified commits were found to be incorrectly identified. These imprecisions arise from violations to implicit assumptions of the SZZ algorithm. Furthermore, the authors of the study empirically found that the results of eight out of ten earlier studies were significantly influenced by the imprecise algorithm [76]. This may, therefore, also apply to this work. However, there is currently no alternative method for identifying bug-introducing commits. Whenever an improved method becomes available, we will repeat the main steps of this work, taking the new method into account, and compare with our results.

## 5 PERSPECTIVES AND DIRECTIONS

Our results give rise to the following research directions.

**Compare file-based and feature-based defect prediction.** To understand the benefits of feature-oriented defect prediction, we shall compare the performance of classifiers on the same set of projects at the two granularity levels (file and feature), and the number of files predicted defective at each level. We shall also investigate whether any correlations exist between project characteristics (e.g., language, average feature size, frequency of changes, sizes of changes, etc.) and performance to investigate project-specific characteristics that may affect performance.

**Apply feature metrics to file-granularity defect prediction.** Even when just performing defect prediction at the granularity of files, we believe that taking information about features into account (with dedicated feature metrics) might improve prediction accuracy. To study this conjecture, a feasible direction is to map our feature metrics back to relevant files, derive a metrics set with mixed file and feature information, and study the impact on prediction quality. To this end, we plan to perform a further study, in which we compare this setup to traditional file-based feature prediction.

**Predict unwanted feature interactions**. Unwanted feature interactions [6] are a special kind of bug, which, when taken into account, may improve the predictive ability of defect prediction techniques, as well as provide meaningful insights regarding whether some machine learning classifiers perform better than others on specific kinds of bugs. While there is work on predicting feature interactions [23], predicting *unwanted* feature interactions has not been attempted yet, to the best of our knowledge. To identify unwanted feature interaction bugs, one possible direction is to apply techniques for identifying variability-aware bugs, such as the one proposed by Abal et al. [2]. Another is to automatically generate test cases for features and use these as partial specifications of a feature behavior (similar to what has recently been done for semantic merge-conflict detection [67]), then exploit these test cases for revealing unwanted feature behavior when features interact.

**Change-based defect prediction for features.** While our study has focused on release-based defect prediction, investigating the prediction of defective features whenever developers commit a new change would contribute to defect prediction techniques that provide immediate feedback to developers.

**Rich feature metrics.** We used feature metrics calculated upon code structure and project history. However, features carry more semantics and richer information, referred to as features facets (e.g., position in the hierarchy or architectural responsibility) [13, 40, 41]. We conjecture crafting metrics taking such facets into account can improve prediction accuracy further.

**Improve generalizability.** Expanding our dataset to include more software projects, as well as considering other machine learning techniques, such as deep learning, can improve our classification results and the generalizability of our technique. Furthermore, investigating cross-project defect prediction is one other possible direction. Here, we hope to gain insights such as whether developers can reuse classification models for unseen projects or classifiers need to be retrained for such projects.

## 6 CONCLUSION

We presented a systematic investigation of feature-based defect prediction. Aiming to predict possible software defects on the granularity of features, we construct a dataset based on 12 real revision histories of feature-based software projects. Using a design science approach, we systematically investigated feature engineering, and finally derived two new carefully crafted metric sets, one solely based on process metrics, one based on a combination of process and structure metrics. We evaluated the predictive ability of seven classifiers in combination with our new metric sets and an additional metric set from a previous work. We conclude:

- Using a more diverse metrics set leads to more robustness and on-average better prediction results.
- Simple classifiers, such as NB, in combination with a simpler metric set (process metrics only) can produce high-quality results; however, at the cost of robustness.
- Enabled by our most advanced metric set, we find two best-performing models (precision, recall, robustness): one based on a random forest classifier, the other based on a neutral network.

## ACKNOWLEDGMENTS

This work has been supported by the Swedish Research Council Vetenskapsrådet and the Wallenberg Academy.

# REFERENCES


[1] 2017. *Encyclopedia of Machine Learning and Data Mining*. Springer US.

[2] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 variability bugs in the linux kernel: a qualitative analysis. In *ASE*. 421–432.

[3] Fumio Akiyama. 1971. An Example of Software System Debugging.. In *IFIP Congress (1)*, Vol. 71. 353–359.

[4] Ethem Alpaydin. 2010. *Introduction to Machine Learning* (second edition ed.). The MIT Press, Cambridge, Massachusetts.

[5] Abdullah Alsaeedi and Mohammad Zubair Khan. 2019. Software Defect Prediction Using Supervised Machine Learning and Ensemble Techniques: A Comparative Study. *Journal of Software Engineering and Applications* 12, 05 (2019), 85–100.

[6] Sven Apel, Joanne M Atlee, Luciano Baresi, and Pamela Zave. 2014. Feature interactions: the next generation (dagstuhl seminar 14281). In *Dagstuhl Reports*, Vol. 4. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[7] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.

[8] Erik Arisholm, Lionel C Briand, and Magnus Fuglerud. 2007. Data mining techniques for building fault-proneness models in telecom Java software. In *ISSRE*. IEEE, 215–224.

[9] Alberto Bacchelli, Marco D'Ambros, and Michele Lanza. 2010. Are popular classes more defect prone?. In *FASE*. 59–73.

[10] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. 2004. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering* 30, 6 (2004), 355–371.

[11] Thorsten Berger and Jianmei Guo. 2013. Towards System Analysis with Variability Model Metrics. In *Eighth International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS '14)*. ACM, New York, NY, USA, Article 23, 8 pages.

[12] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. What is a feature? a qualitative study of features in industrial software product lines. In *SPLC*. 16–25.

[13] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines. In *International Software Product Line Conference (SPLC)*. ACM, 16–25.

[14] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 2011. Don't touch my code! Examining the effects of ownership on software quality. In *ESEC/FSE*. 4–14.

[15] Glenn Bruns. 2005. Foundations for Features. In *Feature Interactions in Telecommunications and Software System*. IOS Press, 3–11.

[16] Venkata Udaya B. Challagulla, Farokh B. Bastani, I. Ling Yen, and Raymond A. Paul. 2008. Empirical assessment of machine learning based software defect prediction techniques. *International Journal on Artificial Intelligence Tools* 17, 2 (2008), 389–400.

[17] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. 2002. SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research* 16 (June 2002), 321–357.

[18] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2010. An extensive comparison of bug prediction approaches. In *MSR*. 31–41.

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

[20] Pedro Domingos. 2012. A few useful things to know about machine learning. *Commun. ACM* 55, 10 (2012), 78–87.

[21] Sascha El-Sharkawy, Nozomi Yamagishi-Eichler, and Klaus Schmid. 2019. Metrics for analyzing variability and its implementation in software product lines: A systematic literature review. *Information and Software Technology* 106 (2019), 1–30.

[22] Norman Fenton, Martin Neil, William Marsh, Peter Hearty, Łukasz Radliński, and Paul Krause. 2008. On the effectiveness of early life cycle defect prediction with Bayesian Nets. *Empirical Software Engineering* 13, 5 (2008), 499.

[23] Stefan Fischer, Lukas Linsbauer, Alexander Egyed, and Roberto Erick Lopez-Herrejon. 2018. Predicting higher order structural feature interactions in variable systems. In *ICSME*.

[24] Javad Ghofrani, Ehsan Kozegar, Arezoo Bozorgmehr, and Mohammad Divband Soorati. 2019. Reusability in artificial neural networks: an empirical study. In *SPLC*. 122–129.

[25] Javad Ghofrani, Ehsan Kozegar, Anna Lena Fehlhaber, and Mohammad Divband Soorati. 2019. Applying Product Line Engineering Concepts to Deep Neural Networks. In *SPLC*. 72–77.

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

[27] Maurice Howard Halstead et al. 1977. *Elements of software science*. Vol. 7. Elsevier New York.

[28] Awni Hammouri, Mustafa Hammad, Mohammad Alnabhan, and Fatima Alsarayrah. 2018. Software Bug Prediction using Machine Learning Approach. *International Journal of Advanced Computer Science and Applications* 9, 2 (2018).

[29] Ahmed E Hassan. 2009. Predicting faults using the complexity of code changes. In *ICSE*. IEEE, 78–88.

[30] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. 2012. Bug prediction based on fine-grained module histories. In *ICSE*. IEEE, 200–210.

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

[32] Alan R Hevner, Salvatore T March, Jinsoo Park, and Sudha Ram. 2004. Design science in information systems research. *MIS quarterly* (2004), 75–105.

[33] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. 2016. Preprocessor-based variability in open-source and industrial software systems: An empirical study. *Empirical Software Engineering* 21, 2 (2016), 449–482.

[34] Matthieu Jimenez, Renaud Rwemalika, Mike Papadakis, Federica Sarro, Yves Le Traon, and Mark Harman. 2019. The importance of accounting for real-world labelling when predicting software vulnerabilities. In *FSE*. 695–705.

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

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

[37] Kyo C. Kang, Sholom Cohen, James Hess, William Nowak, and Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21. Carnegie-Mellon University.

[38] Sunghun Kim, E James Whitehead Jr, and Yi Zhang. 2008. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering* 34, 2 (2008), 181–196.

[39] Michael Kläs, Frank Elberzhager, Jürgen Münch, Klaus Hartjes, and Olaf von Graevemeyer. 2010. Transparent combination of expert and measurement data for defect prediction: an industrial case study. In *ICSE*. 119–128.

[40] Jacob Krueger, Wanzi Gu, Hui Shen, Mukelabai Mukelabai, Regina Hebig, and Thorsten Berger. 2018. Towards a Better Understanding of Software Features and Their Characteristics: A Case Study of Marlin. In *Twelfth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*.

[41] Jacob Krüger, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. 2019. Where is my feature and what is it about? a case study on recovering feature facets. *Journal of Systems and Software* 152 (2019), 239–253.

[42] Craig Larman. 2008. *Scaling lean & agile development: thinking and organizational tools for large-scale Scrum*. Pearson Education India.

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

[44] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. 2008. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering* 34, 4 (2008), 485–496.

[45] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An analysis of the variability in forty preprocessor-based software product lines. In *ICSE*.

[46] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc. 2008. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering* 34, 2 (2008), 287–300.

[47] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[48] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. 2015. The Love/Hate Relationship with the C Preprocessor - An Interview Study. In *ECOOP*. 495–518.

[49] Thilo Mende. 2010. Replication of defect prediction studies: problems, pitfalls and recommendations. In *PROMISE*. 1–10.

[50] Tim Menzies, Jeremy Greenwald, and Art Frank. 2006. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering* 33, 1 (2006), 2–13.

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

[52] Mukelabai Mukelabai, Damir Nešić, Salome Maro, Thorsten Berger, and Jan-Philipp Steghöfer. 2018. Tackling Combinatorial Explosion: A Study of Industrial Needs and Practices for Analyzing Highly Configurable Systems. In *International Conference on Automated Software Engineering (ASE)*. ACM, 155–166.

[53] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2015. Where do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Transactions on Software Engineering* 41, 8 (2015), 820–841.

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

[55] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. 2013. Transfer defect learning. In *ICSE*. IEEE, 382–391.

[56] Damir Nesic, Jacob Krueger, Stefan Stanciulescu, and Thorsten Berger. 2019. Principles of Feature Modeling. In *27th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*.

[57] Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. 2005. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering* 31, 4 (2005), 340–355.

[58] Leonardo Passos, Krzysztof Czarnecki, Sven Apel, Andrzej Wąsowski, Christian Kästner, and Jianmei Guo. 2013. Feature-Oriented Software Evolution. In *VAMOS*.

[59] Leonardo Passos, Rodrigo Queiroz, Mukelabai Mukelabai, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Jesus Padilla. 2018. A Study of Feature Scattering in the Linux Kernel. *IEEE Transactions on Software Engineering* (2018). Preprint.

[60] Juliana Alves Pereira, Hugo Martin, Mathieu Acher, Jean-Marc Jézéquel, Goetz Botterweck, and Anthony Ventresque. 2019. Learning Software Configuration Spaces: A Systematic Literature Review. *arXiv preprint arXiv:1906.03018* (2019).

[61] Rodrigo Queiroz, Thorsten Berger, and Krzysztof Czarnecki. 2016. Towards predicting feature defects in software product lines. In *FOSD*. ACM Press, 58–62.

[62] Rodrigo Queiroz, Leonardo Passos, Marco Tulio Valente, Claus Hunsen, Sven Apel, and Krzysztof Czarnecki. 2015. The shape of feature code: an analysis of twenty C-preprocessor-based systems. *Software & Systems Modeling* 16, 1 (July 2015), 77–96.

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

[64] Foyzur Rahman, Daryl Posnett, Abram Hindle, Earl Barr, and Premkumar Devanbu. 2011. BugCache for inspections: hit or miss?. In *ESEC/FSE*. 322–331.

[65] Jacek Ratzinger, Thomas Sigmund, and Harald C. Gall. 2008. On the relation of refactorings and software defect prediction. In *MSE*. ACM Press, 35–38.

[66] Shivkumar Shivaji, E James Whitehead, Ram Akella, and Sunghun Kim. 2012. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering* 39, 4 (2012), 552–569.

[67] Leuson Da Silva, Paulo Borba, Wardah Mahmood, Thorsten Berger, and Joao Moisakis. 2020. Detecting Semantic Conflicts Via Automated Behavior Change Detection. In *36th IEEE International Conference on Software Maintenance and Evolution (ICSME)*.

[68] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes* 30, 4 (July 2005), 1.

[69] Le Son, Nakul Pritam, Manju Khari, Raghvendra Kumar, Pham Phuong, and Pham Thong. 2019. Empirical Study of Software Defect Prediction: A Systematic Mapping. *Symmetry* 11, 2 (Feb. 2019), 212.

[70] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python framework for mining software repositories. In *ESEC/FSE*. ACM Press, 908–911.

[71] Daniel Strüber, Mukelabai Mukelabai, Jacob Krüger, Stefan Fischer, Lukas Linsbauer, Jabier Martinez, and Thorsten Berger. 2019. Facing the truth: benchmarking the techniques for the evolution of variant-rich systems. In *SPLC*. 26:1–26:12.

[72] Daniel Strueber, Anthony Anjorin, and Thorsten Berger. 2020. Variability Representations in Class Models: An Empirical Assessment. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS)*.

[73] Paul Temple, Mathieu Acher, Gilles Perrouin, Battista Biggio, Jean-Marc Jézéquel, and Fabio Roli. 2019. Towards quality assurance of software product lines with adversarial configurations. In *SPLC*. 277–288.

[74] Paul Temple, José A. Galindo, Mathieu Acher, and Jean-Marc Jézéquel. 2016. Using machine learning to infer constraints for product lines. In *SPLC*. 209–218.

[75] The Authors. 2020. Online Appendix. https://bitbucket.org/easelab/onlineappendixdefectpred.

[76] Ming Wen, Rongxin Wu, Yepang Liu, Yongqiang Tian, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. 2019. Exploring and exploiting the correlations between bug-inducing and bug-fixing commits. In *ESEC/FSE*. ACM Press, 326–337.

[77] Pamela Zave. 2004. FAQ Sheet on Feature Interactions. Available at http://www.research.att.com/~pamela/faq.html.

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

[79] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. 2009. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *ESEC/FSE*. 91–100.

[80] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting Defects for Eclipse. In *PROMISE*. IEEE, 1–7.