Feature-Oriented Defect Prediction

Stefan Strüder†
University of Koblenz-Landau, Germany

Daniel Strüber†
Radboud University Nijmegen, Netherlands

Mukelabai Mukelabai
Chalmers | University of Gothenburg, Sweden

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SPLC '20, October 19–23, 2020, MONTREAL, QC, Canada
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7569-6/20/10...$15.00
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) 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., Naive 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

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

9

https://doi.org/10.1145/3382025.3414960
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 (AUROCE) [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
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 were 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 macro_name_h_. Through a manual
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.

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 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 otherwise.
process metrics, whereas ProcStructMet represents metrics.
We obtained three metric sets: an existing one from our metric set considered by structure metrics: size and complexity. Overall, feature structure metrics, aiming to benefit from two types of metrics by bundling them. We added six additional process metrics. In the absence of dedicated feature 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 cyclo-
matic 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 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 ~

---

**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 (PROC MET, 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).

- **PROC MET:** 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.

- **PROC STRUCT MET:** 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 cyclo-
matic 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.

---

~https://www.cs.waikato.ac.nz/ml/weka/
we studied two research questions: A common mitigation strategy is to apply over-sampling, by generation 3.1) and the three considered metric sets (see subsection 3.2), Using the classifiers we trained based on our dataset (see subsection 4 EVALUATION available implementation in WEKA, in its standard configuration. apply the SMOTE \[ \text{classifier towards misclassification of the under-represented class.} \]

Table 2 reveals that our dataset is imbalanced: Table 2: Selection of classification algorithms

<table>
<thead>
<tr>
<th>classifier</th>
<th>abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>J48 Decision Trees</td>
<td>J48</td>
</tr>
<tr>
<td>k-Nearest-Neighbors</td>
<td>KNN</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>LR</td>
</tr>
<tr>
<td>Naive Bayes Bayes</td>
<td>NB</td>
</tr>
<tr>
<td>Artificial Neural Networks</td>
<td>NN</td>
</tr>
<tr>
<td>Random Forest</td>
<td>RF</td>
</tr>
<tr>
<td>Support Vector Machines</td>
<td>SVM</td>
</tr>
</tbody>
</table>

- **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?

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:

\[ \text{FP rate} = \frac{FP}{FP + TN} \]

Data points 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-performing 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)**

<table>
<thead>
<tr>
<th></th>
<th>QueirozMet</th>
<th>ProcMet</th>
<th>ProcStructMet</th>
</tr>
</thead>
<tbody>
<tr>
<td>J48</td>
<td>0.44</td>
<td>0.24</td>
<td>0.46</td>
</tr>
<tr>
<td>KNN</td>
<td>0.01</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>LR</td>
<td>0.29</td>
<td>0.09</td>
<td>0.14</td>
</tr>
<tr>
<td>NB</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>NN</td>
<td>51.85</td>
<td>53.13</td>
<td>65.34</td>
</tr>
<tr>
<td>RF</td>
<td>10.89</td>
<td>6.04</td>
<td>5.83</td>
</tr>
<tr>
<td>SVM</td>
<td>0.6</td>
<td>0.75</td>
<td>1.65</td>
</tr>
</tbody>
</table>

**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 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). 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 7: Results RQ2: Influence of metrics for top classifiers

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Eval. metric</th>
<th>QUEIROZMET</th>
<th>Metric set</th>
<th>PROCSTRUCTMET</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>detective</td>
<td>clean</td>
<td>w.a.</td>
</tr>
<tr>
<td>J48</td>
<td>Recall</td>
<td>0.57</td>
<td>0.66</td>
<td>0.64</td>
</tr>
<tr>
<td></td>
<td>Precision</td>
<td>0.27</td>
<td>0.87</td>
<td>0.77</td>
</tr>
<tr>
<td></td>
<td>F score</td>
<td>0.37</td>
<td>0.75</td>
<td>0.68</td>
</tr>
<tr>
<td></td>
<td>AUC area</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
</tr>
<tr>
<td>KNN</td>
<td>Recall</td>
<td>0.53</td>
<td>0.56</td>
<td>0.55</td>
</tr>
<tr>
<td></td>
<td>Precision</td>
<td>0.21</td>
<td>0.84</td>
<td>0.73</td>
</tr>
<tr>
<td></td>
<td>F score</td>
<td>0.30</td>
<td>0.67</td>
<td>0.61</td>
</tr>
<tr>
<td></td>
<td>AUC area</td>
<td>0.50</td>
<td>0.50</td>
<td>0.50</td>
</tr>
<tr>
<td>LR</td>
<td>Recall</td>
<td>0.40</td>
<td>0.73</td>
<td>0.67</td>
</tr>
<tr>
<td></td>
<td>Precision</td>
<td>0.25</td>
<td>0.85</td>
<td>0.74</td>
</tr>
<tr>
<td></td>
<td>F score</td>
<td>0.30</td>
<td>0.78</td>
<td>0.70</td>
</tr>
<tr>
<td></td>
<td>AUC area</td>
<td>0.64</td>
<td>0.64</td>
<td>0.64</td>
</tr>
<tr>
<td>NB</td>
<td>Recall</td>
<td>0.38</td>
<td>0.94</td>
<td>0.84</td>
</tr>
<tr>
<td></td>
<td>Precision</td>
<td>0.58</td>
<td>0.87</td>
<td>0.82</td>
</tr>
<tr>
<td></td>
<td>F score</td>
<td>0.50</td>
<td>0.91</td>
<td>0.82</td>
</tr>
<tr>
<td></td>
<td>AUC area</td>
<td>0.61</td>
<td>0.61</td>
<td>0.61</td>
</tr>
<tr>
<td>NN</td>
<td>Recall</td>
<td>0.28</td>
<td>0.75</td>
<td>0.66</td>
</tr>
<tr>
<td></td>
<td>Precision</td>
<td>0.20</td>
<td>0.82</td>
<td>0.71</td>
</tr>
<tr>
<td></td>
<td>F score</td>
<td>0.23</td>
<td>0.78</td>
<td>0.68</td>
</tr>
<tr>
<td></td>
<td>AUC area</td>
<td>0.55</td>
<td>0.55</td>
<td>0.55</td>
</tr>
<tr>
<td>RF</td>
<td>Recall</td>
<td>0.57</td>
<td>0.63</td>
<td>0.62</td>
</tr>
<tr>
<td></td>
<td>Precision</td>
<td>0.26</td>
<td>0.87</td>
<td>0.76</td>
</tr>
<tr>
<td></td>
<td>F score</td>
<td>0.35</td>
<td>0.73</td>
<td>0.66</td>
</tr>
<tr>
<td></td>
<td>AUC area</td>
<td>0.59</td>
<td>0.59</td>
<td>0.59</td>
</tr>
<tr>
<td>SVM</td>
<td>Recall</td>
<td>0.12</td>
<td>1.00</td>
<td>0.84</td>
</tr>
<tr>
<td></td>
<td>Precision</td>
<td>0.83</td>
<td>0.84</td>
<td>0.84</td>
</tr>
<tr>
<td></td>
<td>F score</td>
<td>0.21</td>
<td>0.91</td>
<td>0.78</td>
</tr>
<tr>
<td></td>
<td>AUC area</td>
<td>0.56</td>
<td>0.56</td>
<td>0.56</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Eval. metric</th>
<th>QUEIROZMET</th>
<th>Metric set</th>
<th>PROCSTRUCTMET</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>detective</td>
<td>clean</td>
<td>w.a.</td>
</tr>
<tr>
<td>J48</td>
<td>Recall</td>
<td>0.57</td>
<td>0.66</td>
<td>0.64</td>
</tr>
<tr>
<td></td>
<td>Precision</td>
<td>0.27</td>
<td>0.87</td>
<td>0.77</td>
</tr>
<tr>
<td></td>
<td>F score</td>
<td>0.37</td>
<td>0.75</td>
<td>0.68</td>
</tr>
<tr>
<td></td>
<td>AUC area</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
</tr>
</tbody>
</table>

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
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 preprocessing 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 65% 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 both 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


