Towards Predicting Feature Defects in Software Product Lines

Rodrigo Queiroz
University of Waterloo
Canada

Thorsten Berger
Chalmers | University of Gothenburg
Sweden

Krzysztof Czarnecki
University of Waterloo
Canada

Abstract

Defect-prediction techniques can enhance the quality assurance activities for software systems. For instance, they can be used to predict bugs in source files or functions. In the context of a software product line, such techniques could ideally be used for predicting defects in features or combinations of features, which would allow developers to focus quality assurance on the error-prone ones. In this preliminary case study, we investigate how defect prediction models can be used to identify defective features using machine-learning techniques. We adapt process metrics and evaluate and compare three classifiers using an open-source product line. Our results show that the technique can be effective. Our best scenario achieves an accuracy of 73 % for accurately predicting features as *defective* or *clean* using a Naive Bayes classifier. Based on the results we discuss directions for future work.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement; D.2.8 [*Software Engineering*]: Metrics; D.2.13 [*Software Engineering*]: Reusable Software

General Terms Measurement, Reliability

Keywords software product lines, features, defect prediction

1. Introduction

A Software Product Line (SPL) is a family of related programs (a.k.a., *variants*) that typically share a common codebase. The commonalities and variabilities among the variants are often described in terms of *features*—abstract entities mapped to implementation artifacts, such as files or components. As such, features align more naturally with an SPL's functionalities than implementation artifacts, helping users to understand and select variants, and developers to engineer and evolve an SPL. Reporting and discussing defects, which easily arise in complex SPLs, is also done per feature.

Unfortunately, quality assurance (QA) of features in SPLs is often challenging and costly. First, testing features in isolation is difficult, especially when they are scattered across the codebase [10, 11] and can only be tested at integration time [3]. Second, testing at integration time requires selecting individual variants, which is prone to a combinatorial explosion, which in turn requires feature-sampling strategies or lifting of QA techniques to SPLs.

Defect prediction models can reduce QA costs. Many studies investigate such models at the file [13] or code-change (commit) level [14] and show that they can achieve satisfactory results. However, it is still unknown if such models can be used at the feature level. If effective, feature defect predictions could (i) help developers to select (or prioritize) samples of features that are prone to defects and should be tested more thoroughly; (ii) increase the detection of defects scattered across implementation artifacts, as these will likely have a low impact on the prediction model if using traditional file-based prediction; and (iii) improve the actual QA (e.g., code reviews) when features are used for communicating and coordinating within and across teams.

Towards improving the cost-effectiveness of feature QA, we conduct a preliminary case study on using prediction models for identifying defective features. We show how prediction models can be adapted from standard file- or commit-based approaches by using a subset of known process metrics [13] we adapt to the feature level. We evaluate three machine-learning (ML) algorithms on the open-source project Busybox and discuss their effects on the prediction accuracy.

Our results show that the prediction models can be effective with an accuracy of up to 73%, but also that further investigations of defect prediction for features should be done.

2. Study Design

Our objective is to create a defect-prediction model for features and to evaluate three classifiers on a real system.

FOSD'16, October 30, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4647-4/16/10...$15.00
http://dx.doi.org/10.1145/3001867.3001874

Subject System. BusyBox is an open-source implementation of more than 300 common Linux shell tools (e.g., cat, echo, mount) in one compact, memory-efficient executable. We selected it because it is feature-based, uses static variability through conditional compilation (e.g., #ifdef) and therefore the locations of features in the codebase are easily identifiable. We also had prior experience using it in research.

We extract BusyBox' code history from its actively maintained Git repository[1] until the stable release 1_25_0, which amounts to commits over 13 releases covering the development from 2002 (0.6.0) to 2016 (1_25_0). In total, we explore 3.860 commits contributed by 244 different authors, with 821 unique features across all releases.

Mapping Feature Metadata. Our dataset consists of vectors that for each feature in a specific release, contain the feature name, the release number, values of five process metrics aggregated over all commits associated to the release (explained shortly), and the classification as *defective* or *clean*. More precisely, for each release, we collect its commits until the previous release. From each commit we then extract the changed files, lines of code, author, and commit message. We then associate the commit to it's features. If a commit's diff contains code changes within (or changes of) conditional-compilation directives that belong to one or more features (i.e., the feature appears in the directive's expression), we attribute the commit to each feature. This association is used to collect relations between the authors of the commits and the feature, and to perform the labeling process.

Metrics. Many defect-prediction studies rely on code metrics (e.g., cyclomatic complexity, number of distinct paths, fan in) defined at the file, class or function level [1]. However, the feature code does not necessarily align with these structures. Different metrics specifically designed to evaluate the complexity and size of features would need to be used to learn the metrics' association with defect-proneness of feature code. On the other hand, process metrics (e.g., number of commits made to a file, number of developers who changed the file, percentage of the lines authored by the highest contributor of a file, etc.) can be easily adapted to the feature level. Moreover, previous studies [9, 13] show that process metrics are more effective for prediction than code metrics.

We define the following set of process metrics, adapted from Rahman et al. [13]: `COMM` (number of commits associated to the feature in a release), `ADEV` (number of distinct developers who changed the feature in a release), `DDEV` (cumulative number of distinct developers who contributed to the feature up to the release), `EXP` (geometric mean of the *experiences* of all developers contributing to the feature in a release), `OEXP` (*experience* of the developer who authored most commits associated to the feature). For the latter two metrics, *experience* is defined as the number of added, changed or removed lines that a developer contributed to the project up to the given release.

[1] `https://git.busybox.net/busybox/`

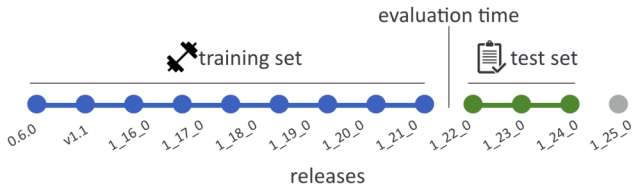

Figure 1. Prediction model training and testing set

Labeling. In our study, we evaluate a binary classifier: a feature is either *defective* or *clean*. The vector obtains the *defective* label when there is a corrective commit associated with the feature code in a later release; otherwise we label it as *clean*. Recall that any labeling is done per release range, so a feature can have different labels (e.g., after the corrective commit, a previously defective feature is clean if no other corrective commit is found in future releases).

We identify corrective commits by searching for the following keywords in the commit message: 'bug', 'fix', 'error', and 'fail'. This approach is commonly used [7, 14], since commit messages with these words were shown to be strongly correlated to actual bug-fix commits.

Model Building. To learn the association between defect proneness of features and their properties (measured by the metrics), the model must be trained (e.g., by creating a decision tree) with a set vectors (features in a specific release) labeled as *defective* or *clean* vectors and tested with another set of vectors. Consequently, we split our dataset into a *training* set, containing the first nine releases (0.6.0 to 1_21_0), and a *testing* set with the following three releases (1_22_0 to 1_24_0). Note that the most recent release 1_25_0 is excluded, since its features cannot be labeled as we have no future release to identify corrective commits touching the code of these features. Yet, its commits are used to identify corrective commits for features in earlier releases.

Fig. 1 summarizes our setup. Other defect-prediction studies use similar setups [13, 14], but also alternative strategies for separating the testing and the training set (e.g., cross validation with k-folds) exist. Our setup is closer to an application of prediction models in a real-world scenario, where developers and testers try to focus their QA efforts on defective features in future maintenance activities.

Classifiers. A classifier takes a set of vectors—containing the class the vector belongs to (*defective* or *clean* in our case) and the attributes (the metric values in our case)— as input, learns a prediction model, and is then able to accurately predict the class to which a new vector belongs. Recall that the same feature in different releases is treated independently, since a feature can have completely different attributes on different releases, or even a different class.

We explore three different classifiers: one based on decision trees (*J48*), one based on a variation of decision trees in case of overfitting (*Random Forest*), and one with independent attributes not based on decision trees (*Naive Bayes*). *J48* is an open-source Java implementation of the popular C4.5

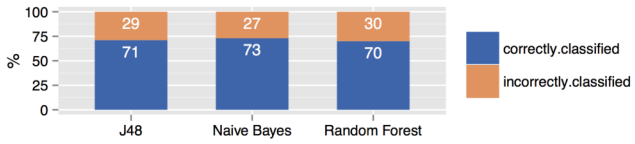

Figure 2. Accuracy of predicting the testing set

decision-tree algorithm [12]. We chose it as it was ranked first in a selection of the ten most influential data-mining algorithms [16]. *Random Forest* (a.k.a., Random Decision Forest) also relies on decision trees. When such trees become very deep, they tend to learn highly irregular patterns and describe random error or noise instead of the underlying relationship (overfitting). This happens in excessively complex models with too many parameters relative to the number of vectors in the training set. Random forest overcomes this problem by maintaining multiple deep decision trees trained on different parts of the training set, reducing the variance [4]. *Naive Bayes* (a.k.a., simple Bayes) is one of the most important methods to solve supervised classification problems. It is easy to construct (e.g., no need for complicated iterative parameter-estimation schemes), applicable to huge data sets, and usually has a good performance [5, 16]. An important assumption is that attribute values are independent from each other. For example, in our study, `COMM` (number of commits to the feature) and `ADEV` (number of distinct developers changing the feature) contribute independently to the probability of a feature being *defective* or *clean*.

Tooling. We use *Codeface* to extract commits with their respective authors and to analyze dependencies between features, commits, and the authorship. Codeface is a framework for the social and technical analysis of software development projects [7]. We write a *custom tool* in Python and Bash to calculate the five process metrics and to store the collected data in a MySQL database. Finally, we use the *WEKA* toolchain to run the three classifiers in their standard implementation.

3. Results

We obtained a training set with 1099 vectors and trained models using all three classifiers. As Fig. 2 shows, all models achieve a similar prediction accuracy. Naive Bayes with a correct classification of 73% testing-set vectors, J48 with 71%, and Random Forest with 70%. To better understand the quality of the model we need to look at results per class. For instance, a poorly designed model that predicts 100 % of features as *clean* in a system with few detective instances will have a high accuracy level (a.k.a., accuracy paradox).

The balance of data is important to evaluate the results. Ideally, both classes have a reasonable number of vectors. From our full set of 1455 vectors (training plus testing set), 531 are *defective* and 924 are *clean*. So, our data is slightly imbalanced, which can considerably affect the performance of a prediction model [14]. However, this scenario is expected, since a real system does likely have fewer defective features.

The confusion matrices in Table 1 help us to understand how the balancing of our data impacts the three classifiers. Although *Naive Bayes* has the best overall accuracy, *J48* performs best for predicting *defective* features.

Table 2 shows details about the prediction performance separated by class (*defective* or *clean*), including precision, recall, true positive (TP) rate, false positive (FP) rate, F-measure, and Receiver Operating Characteristic (ROC).

Good models have both high precision and recall. However, since increasing one often reduces the other we also check the F-measure (harmonic mean of both). It relies on a threshold to declare a feature as *defective* or *clean*. We use a common threshold of 0.5 [13, 17]. To evaluate the performance with a threshold invariant measure (which does not depend on a fixed threshold) we also use the ROC, which is a curve that plots the TP rate (or recall) against the FP rate for all possible thresholds between 0 and 1. The area under the curve (AUC) is used to evaluate how good the model is, from 0 (worst model) to 1 (best model), by comparing the model to random prediction (where AUC would always be 0.5).

All classifiers achieve good F-measures: 0.718 (*Random Forest*), 0.720 (*Naive Bayes*), and 0.724 (*J48*). Defect features are identified with a very low FP rate (0.146) with *Naive Bayes*. However, the same algorithm detects *clean* features with the highest FP rate (0.724). This contrast can be explained by the imbalanced data, but more investigation is still necessary. Overall, *J48* performs best considering F-measure (0.724) and ROC area (0.653), as opposed to *Random Forest* with lower F-measure (0.718) and ROC (0.594).

To understand how the classifiers work, Fig. 3 shows the decision tree generated by *J48* and how the attributes affect the model. For example, when the number of developers who changed the feature in a given release (`ADEV`) is greater than one, the classifier very likely labels the feature instance as defective. If `ADEV` is less or equal 1, the classifier assesses other attributes (e.g., COMM, EXP) before making a decision.

4. Discussion

Based on our results, we plan to improve the prediction technique and investigate the effects of different configurations.

Table 1. Confusion matrices

	Predicted ->	Defective	Clean	Total
J48	Actual Defective	43	33	76
	Actual Clean	72	208	280
	Total	115	241	356
Random Forest	Actual Defective	37	39	76
	Actual Clean	67	213	280
	Total	104	252	356
Naive Bayes	Actual Defective	21	55	76
	Actual Clean	41	239	280
	Total	62	294	356

Table 2. Results obtained with the three algorithms

		Defective	Clean	Weighted Avg.
J48	TP rate	0.566	0.743	0.705
	FP rate	0.257	0.434	0.396
	Precision	0.374	0.863	0.759
	Recall	0.566	0.743	0.705
	F-measure	0.450	0.798	0.724
	ROC area	0.653	0.653	0.653
Random Forest	TP rate	0.487	0.761	0.702
	FP rate	0.239	0.513	0.455
	Precision	0.356	0.845	0.741
	Recall	0.487	0.761	0.702
	F-measure	0.411	0.801	0.718
	ROC area	0.594	0.594	0.594
Naive Bayes	TP rate	0.276	0.854	0.730
	FP rate	0.146	0.724	0.600
	Precision	0.339	0.813	0.712
	Recall	0.276	0.854	0.730
	F-measure	0.304	0.833	0.720
	ROC area	0.612	0.612	0.612

We also aim at studying more and larger systems. In addition, our study suggests the following improvements.

Imbalanced Data. Our dataset had much less *defective* (37 %) than *clean* features. Resampling techniques could mitigate potential effects (accuracy paradox) and increase the percentage of *defective* features in the training set. A simple approach would be to randomly select features from the training set and remove (if *clean*) or duplicate the instance (if *defective*) until a balance is reached. Similar resampling techniques are used by others with satisfactory results [14].

Metrics. Although process metrics usually perform better than code metrics at the file-level, it is not clear whether that holds for the feature level. Additional metrics should be adapted and evaluated to investigate their effect on prediction performance. Examples could include variability model related metrics [2] or feature-related code metrics [8].

Stability. We evaluated the prediction for future releases using the software history up to a given point. However, if a major shift of activities or feature-code refactoring happened in the training set, the prediction performance can be compromised after this point. The aspects that cause defects in features may also shift over time. A prediction model is stable when it adapts well to major changes. We want to investigate how stable the models are. For example, we can investigate if a model trained with only the features from the previous release (or a limited release window) performs better than a model trained with features from all prior releases.

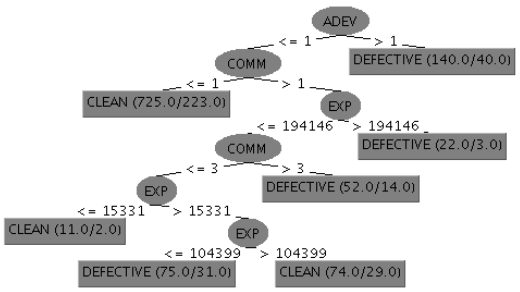

Figure 3. Decision tree generated with J48

Evaluate Stasis. A good metric for defect prediction must co-evolve significantly with the defect occurrence of the feature. When a metric stagnates, it tends to predict the same feature as *defective* in all releases. We need to investigate the stasis of our model to detect if the same features are repeatedly being predicted as *defective*. The model can still have a good performance if the same features repeatedly become *defective* over releases. However, features that are not defective in the training set, but in the testing set, may not be detected.

Cost-Effectiveness. A feature can comprise a significant amount of code and, unlike a file, can be scattered across the codebase. Even if successfully predicted as *defective*, large and scattered features are more costly to inspect. We want to investigate the prediction's cost-effectiveness, evaluating if the investment pays off in terms of defects covered.

5. Related Work

Many studies target prediction models for various domains and propose performance improvements. However, we do not know any study on such models for features in SPLs.

Rahman et al. [13] investigate the efficacy of code and process metrics for defect prediction in a large number of releases from many systems. They compare both kinds of metrics to understand when and why each kind may be suitable for a prediction model, suggesting that process metrics are preferable. We follow this advice.

Tan et al. [14] investigate defect prediction based on change classification (commit level). They perform a case study on a proprietary Cisco system and analyze the reasons for any low performance of the method. Among others, to address the problem of imbalanced data, they add a gap between training and testing set to allow more bugs to be detected. Although a better performance is achieved, the precision is still low, requiring further investigations.

Wang et al. [15] propose using deep learning to learn semantic representations of programs from source code. The goal is to go beyond traditional attributes and use semantic information for characterizing defects and improving defect prediction in within- and cross-project defect predictions. They achieve an average F-measure of 0.641 on 13 systems.

Jeon et al. [6] propose a defect-prediction model for consecutive software products in an SPL. They study historical trends in bug-tracking systems to predict the number of defects in upcoming products, but not for individual features.

6. Conclusion

We investigated how defect prediction models can be used to identify *defective* features using machine learning techniques, aiming at improving the cost-effectiveness of QA activities for features in SPLs. Providing a classification of features based on their error-proneness can help developers to better focus their testing and bug-detection efforts.

Our results show preliminary evidence of the technique's effectiveness. Already with a small set of five process metrics we achieved an accuracy of up to 73 % on predicting features as *defective* or *clean*. Yet, a high FP rate is still a challenge.

We plan to extend our study to systems from more domains, different sizes, and different developer communities. We also plan using different process metrics, comparing the performance of process metrics with code or model metrics, and evaluating different experiment setups (e.g., k-fold cross validation to separate training and testing set). Finally, improving the defect labeling and taking feature-code outside #ifdefs into account are further directions for future work.

References

[1] V. R. Basili, L. C. Briand, and W. L. Melo. A Validation of Object-Oriented Design Metrics As Quality Indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, 1996.

[2] T. Berger and J. Guo. Towards System Analysis with Variability Model Metrics. In *VaMoS*, 2013.

[3] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki. What is a feature? a qualitative study of features in industrial software product lines. In *SPLC*, 2015.

[4] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[5] P. Domingos and M. Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, 29 (2):103–130, 1997.

[6] C. Jeon, C. Byun, N. Kim, and H. In. An entropy based method for defect prediction in software product lines. *International Journal of Multimedia and Ubiquitous Engineering*, 9(3):375–377, 2014.

[7] M. Joblin, W. Mauerer, S. Apel, J. Siegmund, and D. Riehle. From developer networks to verified communities: A fine-grained approach. In *ICSE*, 2015.

[8] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *ICSE*, 2010.

[9] R. Moser, W. Pedrycz, and G. Succi. A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction. In *ICSE*, 2008.

[10] L. Passos, J. Padilla, T. Berger, S. Apel, K. Czarnecki, and M. T. Valente. Feature Scattering in the Large: A Longitudinal Study of Linux Kernel Device Drivers. In *MODULARITY*, 2015.

[11] R. Queiroz, L. Passos, M. T. Valente, C. Hunsen, S. Apel, and K. Czarnecki. The Shape of Feature Code: An Analysis of Twenty C-Preprocessor-Based Systems. *Software & Systems Modeling*, pages 1–20, 2015.

[12] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., 1993.

[13] F. Rahman and P. Devanbu. How, and Why, Process Metrics Are Better. In *ICSE*, 2013.

[14] M. Tan, L. Tan, S. Dara, and C. Mayeux. Online Defect Prediction for Imbalanced Data. In *ICSE*, 2015.

[15] S. Wang, T. Liu, and L. Tan. Automatically Learning Semantic Features for Defect Prediction. In *ICSE*, 2016.

[16] X. Wu, V. Kumar, J. Ross Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z.-H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1): 1–37, 2008.

[17] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process. In *ESEC/FSE*, 2009.