Predicting and Monitoring Bug-proneness at The Feature Level

Shaozhi Wei, Ran Mo*, Pu Xiong, Siyuan Zhang, Yang Zhao, and Zengyang Li

School of Computer, Central China Normal University, China
wsz@mails.ccnu.edu.cn,moran@mail.ccnu.edu.cn, mos365@hotmail.com,
naonao990213@163.com,17302550892@163.com,zengyangli@ccnu.edu.cn

Abstract. Enabling quick feature modification and delivery is important for a project's success. Obtaining early estimates of software features' bug-proneness is helpful for effectively allocating resources to the bug-prone features requiring further fixes. Researchers have proposed various studies on bug prediction at different granularity levels, such as class level, package level, method level, etc. However, there exists little work building predictive models at the feature level. In this paper, we investigated how to predict bug-prone features and monitor their evolution. More specifically, we first identified a project's features and their involved files. Next, we collected a suite of code metrics and selected a relevant set of metrics as attributes to be used for six machine learning algorithms to predict bug-prone features. Through our evaluation, we have presented that using the machine learning algorithms with an appropriate set of code metrics, we can build effective models of bug prediction at the feature level. Furthermore, we build regression models to monitor growth trends of bug-prone features, which shows how these features accumulate bug-proneness over time.

Keywords: Code Metrics, Machine Learning, Feature Bug Prediction

1 Introduction

Bug prediction has been an active research area for decades, numerous studies have been proposed to predict the most bug-prone software units at different granularity levels [15, 33, 5, 36, 30, 24, 14, 18], such as class level, package level, method level, module level, etc. For example, Giger et al. [14] developed bug prediction models at method level. Gyimothy et al. [15] predicted the failure-proneness at class level. Schroter et al. [36] and Nagappan et al. [33] have proposed to predict defects at package and module levels respectively.

However, there has been little work that investigates to build bug prediction models at feature level. A recent work by Wan et al. [42] has presented that defect prediction at the feature level is the most preferred level of granularity by practitioners, such bug predictions could help practitioners gain an insight into software quality at the feature level. Being able to rapidly delivery and modify

* Corresponding Author

features is important for a project's success [11, 31], it is helpful to obtain early estimates on the bug-proneness of features. Such estimates could identify bug-prone features requiring further modifications, which helps developers effectively and efficiently allocate resources to the bug-prone features to assist in quick fixings.

In this paper, we focused our investigations on developing bug prediction models at the feature level and monitoring how bug-prone features accumulate bug-proneness over time. To proceed our study, we first leveraged a feature detection method in [31] to identify all features in a project. For each identified feature, we calculated its code metrics and labeled it either as *bug-prone* or *not bug-prone* based on this feature's bug-proneness calculated from a project's bug data. Next, to address the difficulty caused by noisy or redundant independent variables in each dataset, we leveraged CFS method [17] to select an appropriate set of code metrics for each dataset. Using the selected code metrics as the input attributes, we applied six machine learning algorithms to develop bug prediction models at the feature level. Furthermore, we monitor the evolution of each bug-prone feature. The accurate prediction could help developers identify which features are likely to be bug-pone, and monitoring their evolution could help developers further to understand how each feature has been accumulating bug-proneness over time.

Based on our evaluation analyses on six open source projects, we have found that, by using machine learning techniques and an appropriate set of code metrics, we can build predictive models which accurately predict bug-prone features. However, different projects may need different sets of code metrics as input attributes. Besides, we analyzed how a bug-prone feature evolves over time with respect to its bug-proneness. We have found that 47.8%, 18.9%, and 33.2% of all bug-prone features could fit into linear, exponential and logarithmic regression models respectively, indicating these features have been accumulating bug-proneness steadily, drastically or slowly.

This work extends the state of the art as follows:

- To the best of our knowledge, our work is the first empirical study on predicting bug-proneness at feature level. This work reports how to systematically build feature-level bug prediction models by using code metrics and machine learning techniques.
- We investigate whether a single set of code metrics are generalized to all projects on predicting bug-proneness at the feature level.
- Instead of just building predictive models, we extend to build regression models which could help analyze and monitor how a project's bug-prone features accumulate bug-proneness over time.

2 Background and Related work

Bug prediction has been extensively studied in the past decades. Researchers have proposed numerous predictive models at different granularity levels, such as method level, class level, file level, package level, module level etc.

2.1 Bug Prediction at Class or File level

Numerous studies have been proposed to predict bug-prone classes or files in a software project. Gyimothy et al. [15] calculated the code metrics from the source code of Mozilla, then based on these metrics, they leveraged two statistical methods and two machine learning techniques to predict the failure-proneness of each class. In the study of Ostrand et al. [34], the authors developed a negative binomial regression model using file size and file change information, and demonstrated the model could effectively predict the expected number of faults in each file in the next release of a software project. Cataldo et al.'s work [5] reported a strong correlation between density of change coupling and failure proneness of a file. Yan et al.'s study [43] used both supervised and unsupervised models to predict defects at file level and compared the effectiveness of two types of prediction models.

2.2 Bug Prediction at Method level

Researchers have also studied various bug prediction models at method level. For example, Kim et al. [24] proposed a model which could accurately predict future faults at method level by using cached histories. Giger et al.'s study [14] investigated to build bug prediction models at method level. The authors presented that their models based on method-level code metrics and change metrics could be used to accurately predict bug-prone methods. They also showed that change metrics outperformed source code metrics on bug prediction, and presented that their models are robust with respect to different distributions of samples. Hata et al. [18] extracted method-level histories and developed bug prediction model at method level by using the extracted histories. Their results presented that method-level bug prediction consumed less effort to find bugs than both package-level and file-level predictions. Yang et al.'s research [44] analyzed the relationship between dependency clusters and fault proneness at function (method) level. They demonstrated that their function-level prediction model could significantly increase the performance of fault-proneness prediction, and the inter-dependent functions are often more fault-prone than the other functions.

2.3 Bug Prediction at Coarse-grained level

Defect prediction at a coarse granularity has also been widely studied to facilitate understanding the bug-proneness of software systems. For example, Schroter et al. [36] leveraged failure history at different levels and the usage relationships between components to predict fault-prone components, and demonstrated that the prediction model using package-level defect history could have a better performance. Mishra et al. [30] proposed an approach of Support Vector based Fuzzy Classification System (SVFCS), which combined the advantages of SVM, FIS and Genetic algorithms to effectively predict defects at package level. Menzies et al. [29] applied Naive Bayes learner on a set of complexity metrics to

predict module-level defects. They presented that the static code metrics should be treated as probabilistic, not categorical indicators, and these code metrics could be used to accurately predict defects at module level. Nagappan et al. [33] found that fault-prone software modules are statistically correlated with code complexity measures. The authors investigated various complexity metrics and demonstrated that these metrics were useful and successful for defect prediction. They also demonstrated there was no single set of metrics could be the best predictors in all projects.

2.4 Bug Prediction at other granularity levels

Considering a software system could be structured as many interacting software entities at multiple granularity perspectives, such as component level, subsystem level, file level, etc. Zimmermann et al.'s work [46] explored the effective predictors for cross-project bug prediction. Mockus and Weiss [32] investigated change-level defect prediction, that is, examining the probability that a change to software will cause a failure. They used the properties of a change to be the predictors, such as added or deleted LOC in a change, the number of files and modules affected by a change, the type of a change. Their results showed that their approach was useful for predicting new failures. Kamei et al. [23] proposed an effort-aware linear regression (EALR) model for the defect prediction at change level. Yang et al. [45] also proposed a LT model for just-in-time change-level defect prediction, and compared the performance of unsupervised and supervised approaches.

As we have shown in the above studies, bug prediction at different granularity levels has been widely studied. Our study complementarily contributes to this field by investigating bug prediction at the feature level using code metrics and machine learning algorithms. Moreover, we provide regression models to represent how the bug-prone features evolve over time.

3 Study Design

3.1 Feature Identification

The objective of this work is to come up with bug prediction models at the feature level. For this purpose, we first need to identify features of software projects. Mo et al. [31] recently proposed an approach to identify features by examining a project's revision history and issue tracking system. They also demonstrated that features identified by their approach could form a maintenance unit that should be maintained and developed separately. Our work treats each feature as a software unit separately, predicts and monitors the bug-proneness of each unit at the feature level. Therefore, following the technique in [31], we identified each feature by matching a *new feature* ticket within a commit.

For example, Figure 1 shows a Git commit from one of our studied projects. Based on the message of this commit, we can observe that 1) this commit is

made to implement the issue, `AMQ-5123`, which is labeled as a *new feature* in this project's JIRA[1] issue tracking system; 2) Five java files were changed for this commit. Thus, we could extract a feature, which consists of all these java files. If there exists multiple commits in revision history for the same feature (i.e. multiple commits labeled with the same *new feature* ID), we then consider the feature contains the union of all involved files. After examining a project's revision history and issue tacking records, we can identify all features in this project, and for each feature, we can know its involved files.

```
commit 5da7ab3c0ee027a29c328e48614ffe1a69401577
Author: Hiram Chirino <hiram@hiramchirino.com>
Date:   2014-03-27 13:10:28 -0400

    Implements AMQ-5123: Optionally support encrypted passwords in ActiveMQ users.properties file.

activemq-console/src/main/java/org/apache/activemq/console/command/DecryptCommand.java
activemq-console/src/main/java/org/apache/activemq/console/command/EncryptCommand.java
activemq-console/src/main/java/org/apache/activemq/console/command/ShellCommand.java
activemq-jaas/src/main/java/org/apache/activemq/jaas/EncryptionSupport.java
activemq-jaas/src/main/java/org/apache/activemq/jaas/PrincipalProperties.java
```

Fig. 1: An Example Commit for Feature Identification

3.2 Attributes: Code Metrics

In this paper, we leveraged ten code metrics to build our bug prediction models at the feature level. These metrics could represent a software project's characteristics, and all of them have been widely used for bug prediction at different granularity levels [33, 29, 5, 25, 30, 14, 41]. In this paper, we presented how these metrics could be used for building predictive models at feature level. Next, we briefly described each of the code metrics as follows:

- ***Cyclomatic complexity.*** McCabe [28] proposed *cyclomatic complexity* to measure the code complexity by calculating the number of linearly independent paths through a programs source code. The higher of the metric value, the more complex the source code is.
- ***CK metrics suite.*** Chidamber and Kemerer [6] proposed six metrics which focus on measuring the characteristics of Object-oriented programs. We briefly introduced these metrics as follows: 1) WMC: the number of local methods in a class; 2) DIT: the maximum depth of the inheritance tree in a class; 3) NOC: the number of immediate subclassess of a class; 4) CBO: the number of other classes that are coupled to this class; 5) RFC: the sum of number of methods called within the class's bodies and the number of class's methods; and 6) LCOM: the number of methods in a class that are not related through the sharing of some of the class's fields.

[1] https://www.atlassian.com/software/jira

- ***Fan-in and Fan-out.*** Fan-in is calculated as the number of calling subprograms and the number of global variables read by a class; Fan-out is calculated as the number of called subprograms and the number of global variables set by a class.
- ***Lines of Code (LOC).*** LOC is the number of lines of code in a file. This size metric has been widely accepted among the software engineering community. Various studies [25, 41] have demonstrated that LOC could be used as an effective indicator for bug prediction.

Given a project's source code as input, we calculated these code metrics by using a static analysis tool, Understand[2]. In order to have all metrics apply to features, we summarized the metrics across each feature. For each metric X, we computed the total number per feature. As an example, considering the WMC metric, which counts the number of local methods per class, we calculated the sum of WMC of all files[3] involved in a feature to be the WMC of this feature.

3.3 Labeled Classes: Bug-prone & Not bug-prone feature

In this work, the developed prediction models are used to predict the outcome labels for each upcoming feature, that is, to classify an upcoming feature as *bug-prone* or *not bug-prone*.

Bug Data. To quantify the bug-proneness of a *feature*. We first proposed a history measure, Bug Rate (BR), which represents the bug-proneness of a *file*. To calculate a file's bug rate, we mined a project's revision history and bug reports by using the pattern matching method in [38]: if a bug ticket ID recorded in the bug tracking reports was identified in a commit's message, then we thought this commit was made for a bug fix, and each file in this commit were considered to be changed once for a bug fix. After analyzing all these bug-fixing commits, we could calculate each file's bug rate, which indicates how many times a file has been changed for bug fixes. To have this bug rate measure apply to features, we then calculated the sum of bug rates of all files involved in a feature to be the bug rate of this feature. A feature with a higher value of bug rate will be considered as more bug-prone.

Labeling. Our study trains and validates the prediction models at feature level with the binary target classes. Therefore, we labeled each feature in our dataset either as bug-prone or not bug-prone as follows:

$$Feature = \begin{cases} \text{bug-prone}, & \text{if } BR \geq P_t \\ \text{not bug-prone}, & otherwise \end{cases} \quad (1)$$

where P_t represents the value at a particular percentile rank following the distribution of all features' bug rates per project. A feature will be terms as bug-prone if its bug rate ranks above the particular percentile. In this work,

[2] https://scitools.com/

[3] In this paper, a file means a source file which contains one or more classes. A feature often contains multiple files.

we studied two levels of bug-proneness, denoted to $BR_{80_{th}}$ and $BR_{65_{th}}$. Using $BR_{80_{th}}$ as an example, if a feature's bug rate ranks over 80_{th} percentile in a project, meaning that this feature's bug rate is large than 80% of all features in the project, we then labeled this feature as *bug-prone*. Otherwise we labeled this feature as *not bug-prone*. To substantiate our analysis, we also studied the bug-proneness level of $BR_{65_{th}}$, where the bug rates of features that termed as *bug-prone* rank above 65_{th} percentile, meaning that these features are more bug-prone than 65% of all features in a project in terms of the bug rate measure.

3.4 Machine Learning Algorithms

We leveraged six supervised machine learning algorithms to build models for feature-level bug prediction. We conducted the predictions by using the default setting of Weka[4] tool, which contains a collection of machine learning algorithms for data mining tasks. Next, we briefly described various machine learning techniques used in our study.

- ***Decision Tree (DT)*** classifies data or predicts values by using a tree representation where each leaf node indicates a class label and the internal nodes of this tree represent the attributes of the data.
- ***Naive Bayes (NB)*** is one of the simple probabilistic classifiers [26]. Naive Bayes algorithm leverages Bayes Theorem to calculate the conditional probability of all classes from the training data, and assumes the attributes to be independent.
- ***K-Nearest Neighbors (KNN)*** is a non-parametric, lazy learning algorithm, which could be used to classify new samples based on a similarity measure [9].
- ***Random Forest (RF)*** is an ensemble of decision trees, which could build predictive models for classification [4]. It outputs the class by using the average results from its included decision trees to improve the predictive accuracy and control over-fitting of the decision trees.
- ***Multilayer Perceptron (MLP)*** is a class of feedforward artificial neural network. A MLP contains at least three layers that simulate the biological neurons [19]: an input layer for receiving data; an output layer for making a decision; and one or more hidden layers acting as computational engines.
- ***Support Vector Machine (SVM)*** is a discriminative classifier formally defined by a separating hyper-plane [8]. Given a set of labeled training samples, an SVM training algorithm could build a model to output an optimal hyper-plan which could differentiate new samples into one category or the other.

3.5 Researched Projects

To conduct our evaluation, we analyzed six open source projects with different size and in different domains: ActiveMQ[5] is a multi-protocol, Java-based mes-

[4] https://www.cs.waikato.ac.nz/ ml/weka/
[5] http://activemq.apache.org/

saging server; Camel[6] is an Integration framework; Cassandra[7] is a distributed NoSQL database management system; Hibernate ORM[8] is an object-relational mapping framework for java environments; Hive[9] is a data warehouse software infrastructure; Wicket[10] is a component-based web application framework.

For each project, we selected its latest release as our subject, Table 1 shows the basic facts for each studied project. Column "Rel." indicates the selected release of each project. Column "#Files" shows the number of files in the selected release of a project. For each project, our analysis just focuses on its source files, so all test or example files have been filtered out. Column "#Com." shows the number of commits extracted from the studied revision history of each project. Column "#Ft" shows the number of identified features from each project. Column "#Bugs" presents the number of bug issues calculated by examining a project's bug reports and revision history. Column"History" shows the number of months of each project's revision history we studied, from its beginning to the selected release date.

Table 1: Researched Projects

	Rel.	#Files	#Com.	#Ft	#Bugs	History
ActiveMQ	5.15.9	2,526	6,091	205	1,695	159
Camel	2.21.5	8,697	21,1260	968	2,776	142
Cassandra	3.11.4	1,744	14,701	270	2,914	119
Hibernate	5.4.1	4,073	5,686	141	1,506	138
Hive	2.3.4	4,203	7,785	403	4,921	122
Wicket	8.3.0	2,558	12,255	110	1,701	172

4 Evaluation

4.1 Research Questions

We proceeded our empirical study by investigating the following research questions:

RQ1: Is there a single set of metrics that predicts bug-prone features over all projects?

RQ2: Can we build effective bug prediction models at the feature level?

RQ3: Is it possible to monitor the bug-proneness of bug-prone features?

[6] http://camel.apache.org/
[7] http://cassandra.apache.org/
[8] http://hibernate.org/
[9] https://hive.apache.org/
[10] https://wicket.apache.org/

Using a set of code metrics as attributes often encounters the issue of Multicollinearity due to the existence of inter-correlations among the metrics. Thus, we first analyzed whether there exists a single set of code metrics which is generalized to all projects (RQ1).

Then we investigated how to build bug prediction models at the feature level, and whether the developed models could effectively predict a feature as *bug-prone* or *not bug-prone* (RQ2); Accurate prediction would identify the features that are likely to be bug-pone, which helps developers efficiently allocate resources for quick fixings.

Furthermore, we explored to model and monitor growth trends of the bug-prone features (RQ3). Positive answer to RQ3 would help developers understand how each bug-prone feature evolve over time in terms of bug-proneness, and even guide future modifications.

4.2 Prediction Model Development

To develop bug prediction models with code metrics and machine learning techniques, we used the ten-fold cross validation technique [40]. In ten-fold cross-validation, the original dataset will be randomly divided into 10 subsets with equal size. Next, the technique will iteratively performs 10 times where nine subsets will be used for training and the other one will be used for validation. During each iteration, a subset will be used exactly once as the validation data. The results from 10 folds would then be summarized to present a single estimation. The advantage of this technique is that all observations are used for both training and validation, thus could reduce validation bias [35, 10].

4.3 Evaluation Method

In order to evaluate the performance of our bug prediction models at the feature level, we selected three widely used performance metrics: *Accuracy*, *F-measure* and *AUC*.

We first selected a traditional metric, *Accuracy*, which indicates the ratio of the number of correct predictions to the total number of input samples. However, accuracy may not work very well when prediction models are applied on imbalanced data (i.e., the number of samples belonging to each class is different) [20, 13]. *F-measure* takes both precision and recall into consideration, it is the *Harmonic Mean* between precision and recall which assesses how precise and how robust a classifier is [7, 39]. Besides, various studies [20, 39, 37] proposed ROC analysis and demonstrated that Area Under ROC Curve (AUC) could effectively reflect the performance of prediction models built on imbalanced data, which just likes our dataset where the distributions of two classes (bug-prone or not bug-prone) are imbalanced. AUC could effectively deal with the skewness from class distributions.

4.4 Results

RQ1: Is there a single set of metrics that predicts bug-prone features over all projects?

One difficulty from using a set of metrics as attributes is the issue of multicollinearity among metrics, since there often exists inter-correlations among the code metrics, which causes noisy and redundant independent variables from each dataset [17, 12]. Thus, the initial step in building prediction models is often to select the relevant set of attributes to be used in a ML algorithm. Attribute selection methods [3, 17, 12] have been widely studied for reducing the dimensionality of attribute space and removing redundant or noisy attributes. In our paper, we leveraged the Correlation-based Feature Selection (CFS) method to select an appropriate set of attributes for building prediction models. Hall's work [17] has shown that the CFS method could identify the attributes that have high individual predictive ability on the class, but are not correlated between each other. Moreover, numerous existing studies [2, 10, 27] have used the CFS method and demonstrated its effectiveness in attribute selection for developing prediction models.

We conducted attribute selections for both bug-proneness levels. For each project, we applied CFS method on its two datasets, and the attributes selected from each dataset are shown in Table 2. According to this table, the first straightforward observation to make is that:

all the ten code metrics have been selected more than once for building feature-level prediction models, but there isn't a single set of metrics that has been selected for predicting bug-prone features over all projects. Even for the same project, if we choose different bug-proneness level for labeling, the selected attributes could be different. This result is consistent to the other work at different granularity levels [33, 14].

Answer: There is no single set of code metrics is suitable for the feature-level bug prediction over all projects.

RQ2: Is it possible to build effective bug prediction models at the feature level?

To answer this question, we used the selected attributes (discussed in RQ1) in the six machine learning algorithms to build prediction models for each dataset (each project has two datasets derived from both two bug-proneness levels). Table 3 shows the effectiveness of our predictive models reflected by the performance metrics. Column 3-5 show the *Accuracy*, *F-measure* and *AUC* values with respect to $BR_{80_{th}}$ bug-proneness level. Similarly, column 6-8 show values of the three performance metrics with respect to $BR_{65_{th}}$.

Using column 3-5 at the first row in Table 3 as an example, where the bug-proneness level of $BR_{80_{th}}$ was adopted to label *bug-prone* or *not bug-prone* features. We applied Decision Tree (DT) algorithm with the selected code metrics: *WMC, NOC, CBO, DIT, LCOM, Fan-out* (discussed in table 2) to develop a

Table 2: Selected code metrics after applying CFS method

	Bug-proneness level: $BR_{80_{th}}$
ActiveMQ	WMC, NOC, CBO, DIT, LCOM, Fan-out
Camel	NOC, CBO, Cyclomatic, Fan-in
Cassandra	WMC, CBO
Hibernate	WMC, CBO, RFC, Fan-in, LOC
Hive	Fan-in, LOC
Wicket	WMC, NOC, RFC, Fan-in
	Bug-proneness level: $BR_{65_{th}}$
ActiveMQ	NOC, CBO, RFC, LCOM, Fan-out
Camel	NOC, CBO, Cyclomatic, Fan-in, LOC
Cassandra	CBO, DIT, Cyclomatic, Fan-out, LOC
Hibernate	CBO, LCOM, Cyclomatic, Fan-out, LOC
Hive	Fan-in, LOC
Wicket	WMC, NOC, Cyclomatic, Fan-in

bug prediction model. The results show that the developed prediction model achieves a very good performance: *Accuracy* is 93.2%, *F-measure* is 0.929 and AUC is 0.838. According to the whole Table 3, we can observe that, for all the values of the three performance metrics, most of them (79% of all metric values) are higher than 0.8, and many of them (35% of all metric values) are even higher than 0.9. As a result, we believe that we are able to effectively build bug prediction model at the feature level by leveraging ML algorithms with an appropriate set of code metrics.

Answer: Using an appropriate set of code metrics for each dataset, we could apply machine learning techniques to build feature-level bug prediction models achieving high *Accuracy*, *F-measure* and AUC.

RQ3: Is it possible to monitor the bug-proneness of features?

So far we could use the developed prediction models to accurately classify a feature as bug-prone or not bug-prone. In this way, we can help development teams efficiently allocate resources to the bug-prone features for quick fixings. Besides, *could we monitor how each bug-prone feature accumulate bugs-proneness over time?* If so, development teams would be able to model the growth trends of each feature's bug-proneness, and guide possible future modifications. Using this kind of information, development teams could assess the severity of each bug-prone feature, rank even prioritize each feature's possible fixes. For example, if $feature_i$'s bug-proneness increases exponentially, and $feature_j$'s bug-proneness increases smoothly, we may rank $feature_i$ with a higher priority for future modifications.

At this step, we investigated how each of the bug-prone features accumulate bug-proneness during software evolution. To make a larger scope of investigation, we analyzed all the bug-prone features labeled at the bug-proneness level of

Table 3: Results of prediction models using different ML techniques

Project	ML	$BR_{80_{th}}$			$BR_{65_{th}}$		
		ACC	F-m	AUC	ACC	F-m	AUC
ActiveMQ	DT	0.932	0.929	0.838	0.854	0.855	0.883
	NB	0.898	0.896	0.935	0.834	0.824	0.865
	KNN	0.888	0.888	0.855	0.878	0.878	0.866
	RF	0.922	0.92	0.94	0.868	0.868	0.949
	MLP	0.907	0.905	0.943	0.849	0.841	0.907
	SVM	0.868	0.842	0.679	0.746	0.695	0.639
Camel	DT	0.90	0.896	0.902	0.831	0.830	0.864
	NB	0.885	0.881	0.892	0.79	0.771	0.844
	KNN	0.887	0.888	0.84	0.853	0.854	0.841
	RF	0.913	0.911	0.958	0.88	0.880	0.932
	MLP	0.888	0.882	0.888	0.813	0.810	0.862
	SVM	0.866	0.851	0.709	0.767	0.730	0.671
Cassandra	DT	0.863	0.865	0.866	0.881	0.881	0.893
	NB	0.774	0.744	0.697	0.644	0.566	0.685
	KNN	0.907	0.909	0.886	0.896	0.895	0.879
	RF	0.893	0.891	0.947	0.867	0.867	0.948
	MLP	0.874	0.869	0.921	0.837	0.836	0.911
	SVM	0.789	0.702	0.495	0.637	0.544	0.508
Hibernate	DT	0.922	0.922	0.909	0.823	0.826	0.786
	NB	0.901	0.896	0.918	0.78	0.757	0.880
	KNN	0.943	0.941	0.871	0.844	0.842	0.824
	RF	0.929	0.929	0.966	0.837	0.839	0.921
	MLP	0.922	0.921	0.925	0.801	0.796	0.904
	SVM	0.809	0.735	0.534	0.667	0.563	0.565
Hive	DT	0.916	0.919	0.912	0.913	0.914	0.926
	NB	0.806	0.766	0.855	0.677	0.598	0.814
	KNN	0.921	0.921	0.897	0.906	0.906	0.890
	RF	0.923	0.924	0.967	0.931	0.931	0.965
	MLP	0.916	0.919	0.950	0.898	0.899	0.950
	SVM	0.806	0.766	0.574	0.675	0.594	0.576
Wicket	DT	0.964	0.964	0.962	0.809	0.811	0.862
	NB	0.955	0.954	0.988	0.873	0.866	0.898
	KNN	0.955	0.954	0.904	0.909	0.910	0.909
	RF	0.936	0.936	0.968	0.873	0.873	0.946
	MLP	0.964	0.962	0.905	0.773	0.762	0.838
	SVM	0.809	0.739	0.543	0.673	0.566	0.538

$Bug_{65_{th}}$, since the set of bug-prone features at $Bug_{80_{th}}$ level is just a subset of it. To proceed this investigation, we first constructed an evolution sequence for each feature in terms of its bug rate: for each bug-prone feature, we back-forwardly calculated its bug rates based on different periods of revision history. For example, if $feature_i$ is one of the bug-prone features in project A, which was released in 2018-08, and the feature started to accumulate bugs (i.e. its bug rate became larger than 0, and started to increase) in 2016-10. For this $feature_i$,

we back-forwardly decreased its history period by a 6-month history interval to calculate a series of values of the bug rate. In this way, we would calculate the bug rate sequence of $feature_i$ by using four history periods: 2016-10 - 2017-02, 2016-10 - 2017-08, 2016-10 - 2018-02, and 2016-10 - 2018-08. For all bug-prone features in our studied projects, we repeated the calculations to obtain all bug rate sequences.

Secondly, we selected three widely used regression models to simulate the growth trend of each feature's bug-proneness. The three models reflect three types of growth trends in practice:

- Linear Model. It indicates a feature's bug rate increases linearly, meaning that this feature has been accumulating bug-proneness steadily.
- Exponential Model. It indicates a feature's bug rate increases exponentially, meaning that this feature has been accumulating bug-proneness dramatically, and the speed becomes faster and faster;
- Logarithmic Model. It indicates a feature's bug rate increases slower and slower, meaning that this feature accumulated bug-proneness quickly at the beginning, but it is accumulating bug-proneness very slow now.

Given a sequence of bug rates of a feature, we modeled the feature's growth trend to one of the three models: linear, exponential and logarithmic regression models, which indicated the feature was accumulating bugs in different trends. For each feature, the regression model with highest R^2 would be selected to be the best fit for it. Besides, the P-value of each fitting model should be less than 0.05, which guarantees that the derived model is significant.

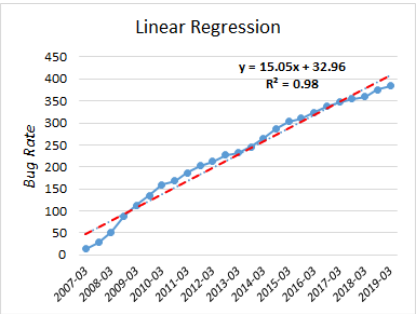
In this work, we categorized the regression models by following the guidelines in [16, 22], where the authors described $R^2 = 0.75$, 0.5 and 0.25 as substantial, moderate and weak models, respectively. We summarized all the fitting results in Table 4. Column "#Features" shows the total number of bug-prone features we analyzed for each project. Columns "Lin", "Exp" and "Log" present the number of features whose growth trends fit into linear, exponential or logarithmic models respectively. The following "P." columns show the corresponding ratios to the total number of bug-prone features. In the column of "$0.5 <= R^2 < 0.75$", we only used"Num" and "P." columns to show the number and percentage of the features fitted into a moderate model, that is, the R^2 ranges from 0.5 to 0.75.

Using "Camel" in Table 4 as an example, we can observe that 339 features were labeled as bug-prone at the bug-proneness level of $Bug_{65_{th}}$, and 99.7% of these features could be substantially modeled by the regression models ($R^2 >= 0.75$). 36.0% of all these features could fit into a linear model, meaning that these features has accumulated bug-proneness steadily. 1.8% and 61.9% of all these features could fit into the exponential and logarithmic models respectively. Only 1 feature, 0.3% of all the studied features, couldn't fit into a substantial, but fitted into a moderate regression model.

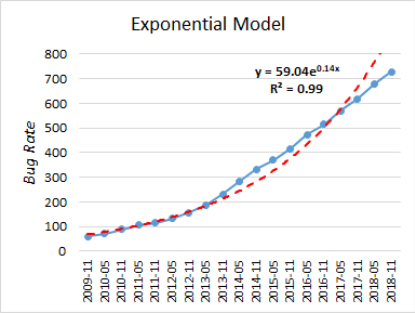
The last row of Table 4 presents that, considering all the studied features over all projects together, almost all of them (99.9%) could fit into a substantial regression model. 47.8% of them could be modeled by linear models. For both

Table 4: Distribution of Bug-prone Features' Regression Models in Terms of Bug Rate

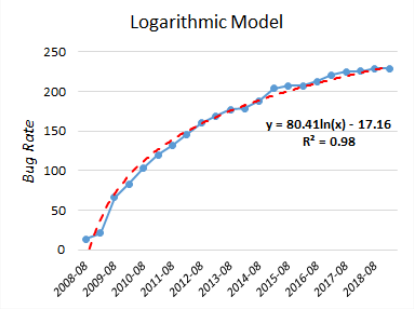
		$R^2 >= 0.75$								$0.5 <= R^2 < 0.75$	
Project	#Features	Lin	P.	Exp	P.	Log	P.	Total	P.	Num	P.
ActiveMQ	72	72	100.0%	-	-	-	-	71	100.0%	-	-
Camel	339	122	36.0%	6	1.8%	210	61.9%	338	99.7%	1	0.3%
Cassandra	95	81	85%	-	-	14	15%	95	100%	-	-
Hibernate	50	44	88%	6	12%	-	-	50	100%	-	-
Hive	142	15	10.6%	127	89.4%	0	0%	142	100%	-	-
Wicket	39	18	46%	-	-	21	54%	39	100%	-	-
Total	737	352	47.8%	139	18.9%	245	33.2%	736	99.9%	1	0.1%



(a) An Example of Linear Regression



(b) An Example of Exponential Regression



(c) An Example of Logarithmic Regression

Fig. 2: Example Features Fitting into Different Regression Models

exponential and logarithmic models, there are 18.9%, 33.2% of all the studied features that follow a substantial fitting respectively.

Figure 2 shows the examples for all types of regression models derived from our studied features. Using Figure 2a as an example to illustrate the results, we can observe that the selected release date of this project is "2019-03", and files

in this feature started to accumulate bugs before 2007-03. The growth trend of this feature's bug rates is fitted into a linear model, which has a $R^2 = 0.98$, with a formula as: $y = 15.05x + 32.96$. Figure 2b and 2c show the cases where growth trends of these two features' bug-proneness fit into an exponential and a logarithmic regression model respectively.

Answer: In terms of the bug rate measure, almost all of the bug-prone features (99.9% of all bug-prone features) can substantially fit into a regression model, wherein 47.8%, 18.9% and 33.2% of all these features could fit into the linear, exponential and logarithmic models respectively. Thus, we can effectively monitor the bug-proneness of each bug-prone feature and present how this feature accumulates bugs over time.

5 Threats to Validity

First, to calculate the bug rate, we use the pattern matching method in [38] to identify a bug-fixing commit if its change message contains a bug ticket ID. A file's bug rate will be the number of times a files involved in bug-fixing commits. However, we cannot guarantee that the bug data extracted from revision history are not biased. Prior studies [1, 21] have shown that: 1) a file changed in a bug-fixing commit doesn't necessarily implies this file is changed for a bug fix; 2) and there is sometimes no explicit link which could be used for targeting the bug-fixing commits in revision history. Thus findings with respect to the research questions could be impacted by the accuracy of available data. We acknowledge this is a threat to internal validity and requires more investigation.

Second, the selection of bug-proneness level may rise a threat to internal validity. We labeled each feature based on the bug rate measure. Different bug-proneness levels could lead to different distributions of labeled classes, which may have an influence on building the bug prediction models. To weaken the interference of noise in data result, we used both bug-proneness levels of $Bug_{65_{th}}$ and $Bug_{80_{th}}$. Our results have presented that, for both levels, our approach could develop accurate bug prediction models at the feature level. Besides, project practitioners could select bug-proneness level of the input dataset based on their own interests.

Third, a threat to external validity is in our data set. We only analyzed six open-source projects. To partially address this problem, we selected the projects having different sizes and in different domains.

Forth, we only analyzed the projects that use Git for version control and use JIRA for issue tracking, hence we can not claim that our results are generalizable to other projects managed by other version control or issue tracking systems. We are planning to repeat our experiments to a broader set of projects.

6 Conclusion

In this paper, we have studied how to develop bug prediction models at the feature level. More specifically, for each dataset, we selected an appropriate set of static code metrics as the attributes to be used for six machine learning algorithms, and developed a feature-level bug prediction model.

From our analyses on six open source projects, we have demonstrated that: based on an appropriate set of code metrics, we can apply machine learning algorithms to build feature-level bug prediction models achieving good performance in terms of *Accuracy*, *F-Measure* and *AUC* metrics. But there isn't a common set of code metrics applied to all studied projects; For all the bug-prone features, we can effectively model the growth trends of their bug-proneness, so that we can monitor how these bug-prone features accumulate bugs during software evolution.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under the grant No. 62002129, the Hubei Provincial Natural Science Foundation of China under the grant No. 2020CFB473, and the Fundamental Research Funds for the Central Universities under the grant No. CCNU19TD003.

References

1. G. Antoniol, K. Ayari, M. D. Penta, F. Khomh, and Y.-G. Gueheneuc. Is it a bug or an enhancement?: A text-based approach to classify change requests. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, pages 23:304–23:318, 2008.
2. E. Arisholm, L. C. Briand, and E. B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *J. Syst. Softw.*, 83(1):2–17, 2010.
3. A. L. Blum and P. Langley. Selection of relevant features and examples in machine learning. *Artif. Intell.*, 97(1-2):245–271, 1997.
4. L. Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, 2001.
5. M. Cataldo, A. Mockus, J. A. Roberts, and J. D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*, 35(6):864–878, July 2009.
6. S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.
7. N. Chinchor. Muc-4 evaluation metrics. In *Proceedings of the 4th Conference on Message Understanding*, pages 22–29, 1992.
8. C. Cortes and V. Vapnik. Support-vector networks. *Mach. Learn.*, 20(3):273–297, 1995.
9. T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Trans. Inf. Theor.*, 13(1):21–27, 2006.

10. A. B. de Carvalho, A. Pozo, and S. R. Vergilio. A symbolic fault-prediction model based on multiobjective particle swarm optimization. *J. Syst. Softw.*, 83(5):868–882, 2010.
11. B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: A taxonomy and survey. In *Journal of Software Maintenance and Evolution: Research and Practice*, 2011.
12. G. Forman. An extensive empirical study of feature selection metrics for text classification. *J. Mach. Learn. Res.*, 3:1289–1305, 2003.
13. K. Gao, T. M. Khoshgoftaar, and A. Napolitano. Combining feature subset selection and data sampling for coping with highly imbalanced software data. In *International Journal of Software Engineering and Knowledge Engineering*, pages 115–146, 2015.
14. E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall. Method-level bug prediction. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '12, pages 171–180, 2012.
15. T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2005.
16. J. F. Hair, C. M. Ringle, and M. Sarstedt. Pls-sem: indeed a silver bullet. *Journal of Marketing Theory and Practice*, 19(2):139–151, 2011.
17. M. A. Hall. Correlation-based feature selection for discrete and numeric class machine learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 359–366, 2000.
18. H. Hata, O. Mizuno, and T. Kikuno. Bug prediction based on fine-grained module histories. In *Proceedings of the 34th International Conference on Software Engineering*, pages 200–210, 2012.
19. S. Haykin. *Neural Networks: A Comprehensive Foundation*. 2nd edition, 2004.
20. H. He and E. A. Garcia. Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 21(9):1263–1284, 2009.
21. K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 392–401, 2013.
22. J. Joseph F. Hair, G. T. M. Hult, C. Ringle, and M. Sarstedt. *A Primer on Partial Least Squares Structural Equation Modeling (PLS-SEM)*. Sage, Thousand Oak, 2013.
23. Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013.
24. S. Kim, T. Zimmermann, J. E. James Whitehead, and A. Zeller. Predicting faults from cached history. In *Proc. 29thInternational Conference on Software Engineering*, pages 489–498, 2007.
25. A. G. Koru, D. Zhang, K. E. Emam, and H. Liu. An investigation into the functional form of the size-defect relationship for software modules. *IEEE Transactions on Software Engineering*, 35(2):293–304, 2009.
26. D. D. Lewis. Naive (bayes) at forty: The independence assumption in information retrieval. In *Proceedings of the 10th European Conference on Machine Learning*, ECML '98, pages 4–15, 1998.
27. R. Malhotra and M. Khanna. An empirical study for software change prediction using imbalanced data. *Empirical Software Engineering*, 22(6):2806–2851, 2017.
28. T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.

29. T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Trans. Softw. Eng.*, 33(1):2–13, 2007.
30. B. Mishra, C. Engg, and K. Shukla. Defect prediction for object oriented software using support vector based fuzzy classification model. In *International Journal of Computer Applications*, 2012.
31. R. Mo, Y. Cai, R. Kazman, and Q. Feng. Assessing an architecture's ability to support feature evolution. In *Proceedings of the 26th Conference on Program Comprehension*, pages 297–307, 2018.
32. A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.
33. N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proc. 28th International Conference on Software Engineering*, pages 452–461, 2006.
34. T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.
35. G. J. Pai and J. B. Dugan. Empirical analysis of software fault content and fault proneness using bayesian methods. *IEEE Trans. Softw. Eng.*, 33(10):675–686, 2007.
36. A. Schrter, T. Zimmermann, and A. Zeller. Predicting component failures at design time. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, pages 18–27, 2006.
37. R. Shatnawi. Improving software fault-prediction for imbalanced data. In *2012 International Conference on Innovations in Information Technology (IIT)*, pages 54–59, 2012.
38. J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, 2005.
39. M. Sokolova and G. Lapalme. A systematic analysis of performance measures for classification tasks. *Inf. Process. Manage.*, 45(4):427–437, 2009.
40. M. Stone. Cross-validation choice and assessment of statistical predictions. *Journal of the Royal Statistical Society*, 36:111–133, 1974.
41. M. D. Syer, M. Nagappan, B. Adams, and A. E. Hassan. Replicating and re-evaluating the theory of relative defect-proneness. *IEEE Transactions on Software Engineering*, 41(2):176–197, 2015.
42. Z. Wan, X. Xia, A. E. Hassan, D. Lo, J. Yin, and X. Yang. Perceptions, expectations, and challenges in defect prediction. *IEEE Transactions on Software Engineering*, 2018.
43. M. Yan, Y. Fang, D. Lo, X. Xia, and X. Zhang. File-level defect prediction: Unsupervised vs. supervised models. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 344–353, 2017.
44. Y. Yang, M. Harman, J. Krinke, S. Islam, D. Binkley, Y. Zhou, and B. Xu. An empirical study on dependence clusters for effort-aware fault-proneness prediction. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 296–307, 2016.
45. Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung. Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 157–168, 2016.
46. 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

Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE '09, pages 91–100, 2009.