How Many Versions Does A Bug Live in? An Empirical Study on Text Features for Bug Lifecycle Prediction

Chuanqi Wang[1,2], Yanhui Li[1,2,*], Baowen Xu[1,2,*]
1. State Key Laboratory for Novel Software Technology, Nanjing University, China
2. Department of Computer Science and Technology, Nanjing University, China
* Corresponding authors: {yanhuili, bwxu}@nju.edu.cn

***Abstract*—During the software system's maintenance and evolution, finding and removing software bugs is a very important part that consumes a large amount of money and effort. To analyze different bugs' character, it is very essential to know how long or which period of versions does the bug live in.**

In this study, we define version-based bug lifecycle and propose a text features based classification model to predict the version-length of bug lifecycle. We collect 57000+ bugs from 10 well-know Apache Software Foundation projects to construct our dataset, and use the tf-idf method to collect our text features from bug report's summary and description.

Our experimental results show that the text feature based method performs better than other baseline methods on 10 projects. The text feature based Naive Bayes classifiers outperforms all other methods with different features and classifiers.

I. Introduction

Finding and removing Software bugs is a very important part of software evolution and maintenance that consumes a large amount of money and effort [1]. An extensive body of bug-related studies [2]–[6] have been proposed to help programmers to predict, detect and fix bugs. In all bug-related research areas, the **bug lifecycle** (the time difference between bug introduced time and bug fixed time) is an important time indicator, which is useful for many applications, such as predicting fault-proneness of code region [7] or identifying the origins of bugs [8]. The recent empirical study shows that some long-lifecycle bugs may remain alive for a very long time and in multiple versions [9], [10].

Some researchers have paid attention to bug lifecycle and their studies could be divided into two parts: **Fixing Period**, (the righthand side of bug lifecycle, from reported time to fixed time) and **Dormant Period** (the lefthand side, from introduced time to reported time). In Fixing Period part, the lengths of Fixing Period are usually considered as the bug fixing effort (BFE). To investigate which factor impacts the BFE, the correlation analysis was conducted [11]. Zhang et al. focused on the correlation of different factors and BFE by logistic regression models [12]. Furthermore, based on different datasets and classifiers, researchers have proposed prediction models to predict the BFE when the bug was reported. Song et al. proposed the association mining rules to build BFE prediction model on NASA's data sets [13]. Zhang et al. proposed k-Nearest Neighbors based method to construct the BFE prediction model on commercial projects [14]. In Dormant Period part, Chen et al. introduced affected version as an indicator for bug introduced time, then calculated the dormant period from introduced time to reported time [9].

Our work differs from existing studies in three important ways. **Firstly**, we consider the bug lifecycle as a whole in our study. Previous studies usually focus on the righthand or lefthand side of lifecycle. **Secondly**, we define version-based bug lifecycle and predict the version-length of bug lifecycle. Previous studies are mainly based on real time interval. We observe that in most projects, interval time between versions are different, so we believe that version could be considered an effect time units of measurement. **Thirdly**, we focus on **text** features from the summary and description in bug reports and construct the bug lifecycle prediction model. Our model aims to help the project manages and bug fixers trace the bug fixing back to the bug introducing commit.

Our main contribution consists of the following steps:

- We collect the datasets with 57000+ bugs on 10 Apache Software Foundation Projects from Jira and Github. Each projects have more than 2000 bugs.
- We use tf-idf [15], a statistic method to calculate a single word's importance, to construct our text features.
- Our evaluation results show that our text features significantly improves the performance of the version-based bug lifecycle prediction model.

The rest of this paper is organized as follows. Section II defines bug lifecycle. Section III shows our prediction experiment setups. Section IV evaluates the performance of our text features and prediction models. Threats to validity is presented in Section V. Section VI concludes our work.

II. Version-based Bug Lifecycle

This section describes our data collection approach, the studied projects, and defines bug lifecycle.

A. Linking Jira and Github

Our study mainly focuses on data from two sources: bug related data from Jira and code evolution data from Github. What makes Jira becomes our bug reports database is not only

DOI reference number: 10.18293/SEKE2018-176

TABLE I: An Overview of 10 Studied Apache Projects.

Project	Description	All	AV	AV%
CXF	Web services framework.	6431	1423	22.13%
Flink	Stream processing framework.	6783	895	13.19%
Flume	Service for efficiently dealing log data.	3167	834	26.33%
Groovy	Object-oriented programming language for Java.	8100	3697	45.64%
Hadoop	Software framework for distributed storage.	12723	6766	53.18%
NiFi	Enables data flow between systems.	4446	959	21.57%
OpenJPA	Implementation of the Java Persistence API specification.	2718	1654	60.85%
PDFBox	Pure-Java library.	2952	2055	69.61%
Tuscany	Developing and running software applications.	4,082	1420	34.79%
Wicket	A lightweight component-based web application framework.	6458	2169	33.59%
Totals		57860	21872	37.80%

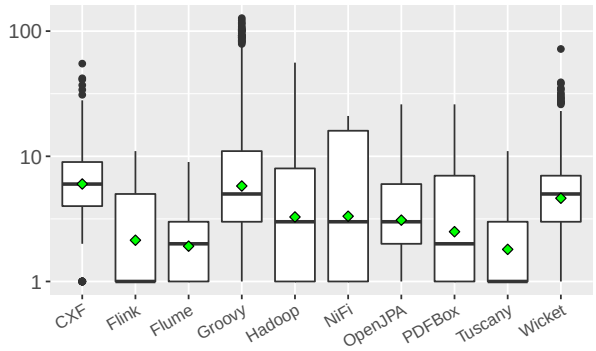


Fig. 1: The boxplots of 10 Projects distribution of the version-length from the affect version to the fixing version.

the popular it is, but also it provides the **affect-version** field. It is filled by the bug fixing developers and is an indicator of bug introducing time estimated by the development team [16]. Github is a web-based Git and Version Control System(VCS). We can get several collaboration features such as bug tracking, feature requests and task management from it [17].

To synchronize the bug reports from Jira and commits from Github, bug id with some key words such as "bug", "fix", "defect" are regarded as the link between a bug reports and commits [18], [19]. Although there are many new technique to link the bug reports with commits such as ReLink [20] and MLink [21], in this paper, a bug report can be linked with a commit only when their bug id is same which ensured the correctness of our data.

B. The Studied Projects

We choose 10 Apache Software Foundation (ASF) projects as our studied projects. First, they are well-known ASF projects available in Jira and Github. Second, these projects have enough proportion and number of bug reports with affect version. Third, these projects have the enough number of bug reports matching from Jira to Github.

Table I summarizes these 10 ASF projects with their description and the numbers of bug reports. The third column (ALL) is the number of all the bug reports recorded in Jira. The forth column (AV) is the bug reports that have affect version item filled in by developers and the last column (AV%) is the percentage of them comparing to all bug reports.

C. Definition of bug lifetime

As discussed in Introduction, we define the version-based life cycle $LC(x)$ of bugs x, as the version sequence from the introducing version V_i to the fixing version V_f:

$$LC(x) = V_i, V_{i+1}, \cdots, V_f$$

and the version-length of life cycle $LLC(x)$ as version difference from V_i to V_f.

In our experiment, we use the affect version in Jira as the the introducing version of bugs. Costa et al. [16] applied the affect version on evaluating the approach of identifying bug-introducing changes. If a bug has multiple affect versions filled in, we use the earliest affect version as bug introducing version. Jira also provides the fixing version in most bug reports, thus we use the fixing version as the endpoint version of bug lifecycle. Correspondingly, if a bug has multiple fixing versions, we use the latest one.

The AV% column in Table I shows that about 38% bugs have complete lifecycle information (affected version and fixing version). We use these 38% bugs as the studied dataset to construct and evaluate our prediction models. Figure 1 presents the bug's lifetime statistic in the chosen 10 projects. We observe that:

- in all projects, many bugs live in lifecycle with multiple version-length ($LLC(x) > 1$).
- the boxplots have much difference from project to project, and the quartile are also different which could be used to define the cutoff points in sectionIII.

III. Prediction Methodology

In this section, we will discuss the approach and evaluation metrics used in the following experiment.

A. Text Feature Extraction

It is three reasons that we use the text features in the bug lifecycle prediction scenario. First, text features are included in bug reports descriptions or summary and most of the bug reports have these items filled in. Second, there are numerous information that exists in bug report descriptions written by developers. We can extracted features from it to build the prediction model. Third, text feature based approaches have been used in many former researches [22]–[24] in other fields such as defect fixing effort prediction and software constructive cost prediction. Kikas et al. used the number of text comments or size of text comments as text features to predict when a bug will be closed in the future [25].

Text Feature. In our work, to generate text features from text descriptions of historical defect reports. We utilize tf-idf (short for term frequency-inverse document frequency) api of Scikit-learn package in Python language to extract word tokens from the textual descriptions.

The tf-idf method consists of two parts tf and idf. The full name of tf is term frequency which is the number of times a term appears in a document. The n word tokens we extract from a text descriptions can be defined as $w[0, ..., n]$, where w_i means the i-th word in the text. tf can be defined as:

$$\mathrm{tf}_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}},$$

where the $n_{i,j}$ means the number of the word w_i appears in document j. The $\sum_k n_{k,j}$ means the summation of document j's words.

The full name of idf is inverse document frequency, which gives us the word's frequency across the documents. The formula of idf can be defined as:

$$\mathrm{idf}_i = \log \frac{|D|}{|\{j : w_i \in d_j\}|},$$

where the $|D|$ means the number of documents included in the corpus. The $|\{j : w_i \in d_j\}|$ means the number of documents the w_i appears in the corpus.

Finally, the value of tf-idf can be calculated as:

$$\mathrm{tf} - \mathrm{idf} = \mathrm{tf}_{i,j} \times \mathrm{idf}_i$$

In information retrieval, tf-idf is a statistic method that is intended to perform a single word's importance of a certain document in a corpus [15]. It is also widely used in information retrieval or text mining as a weighting factor.

Basic Feature. We also use the basic metrics from Jira's bug reports as our baseline prediction features. There are many items that Jira provides for developers to fill in. But some of the items most developers (over 99%) did not fill in when they reported a bug, such as Components, Time Spent, Work Ratio, and Security Level. And there are also some items that can not be directly used as features, such as Assignee, Reporter, Creator and Environment. Thus, from our investigation, there are Priority, Votes and Watchers items that can be directly used as features to build the basic feature prediction model.

B. Prediction Settings

Two Classification Problem Definition. Numerous researchers have used the classification model in bug fixing effort prediction [5], [26], [27], which inspires us to build a classification prediction model for verson-based bug lifecycle.

In the view of every project's distribution of length of bug lifecycle (LLC) in Figure 1, the boxplots have much difference from project to project, and the three cutoff points (25%,50%,75%) of boxplots are also different. By the cutoff point, we transfer the bug lifecycle prediction to a two classification problem. Figure 2 shows how the transformation executes. In Figure 1, the boxplots of project CXF have three cutoff points: 4(25%), 6(50%), 9(75%). For 4 as 25% cutoff point, we group the bugs whose LLC less than 4 into a class ($<$ cutoff class) and greater equal than 4 into another class ($\geq$ cutoff class), then we do the two classification. For each projects, we will do the two classification at different three cutoff points (25%,50%,75%). There are two special cases in Figure 1 that Flink and Tuscany's cutoff points have some

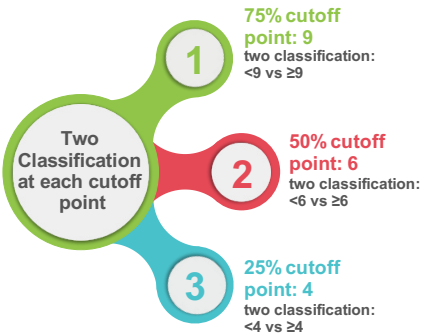


Fig. 2: An example of CXF Project's two classification at 3 Cutoff Points: 75%(9), 50%(6) and 25%(4).

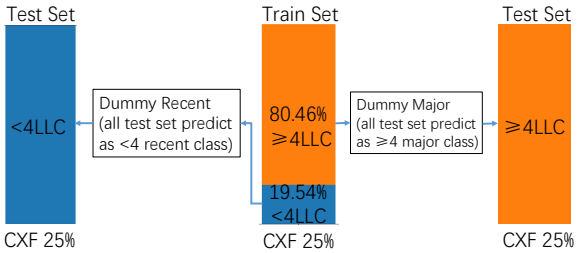


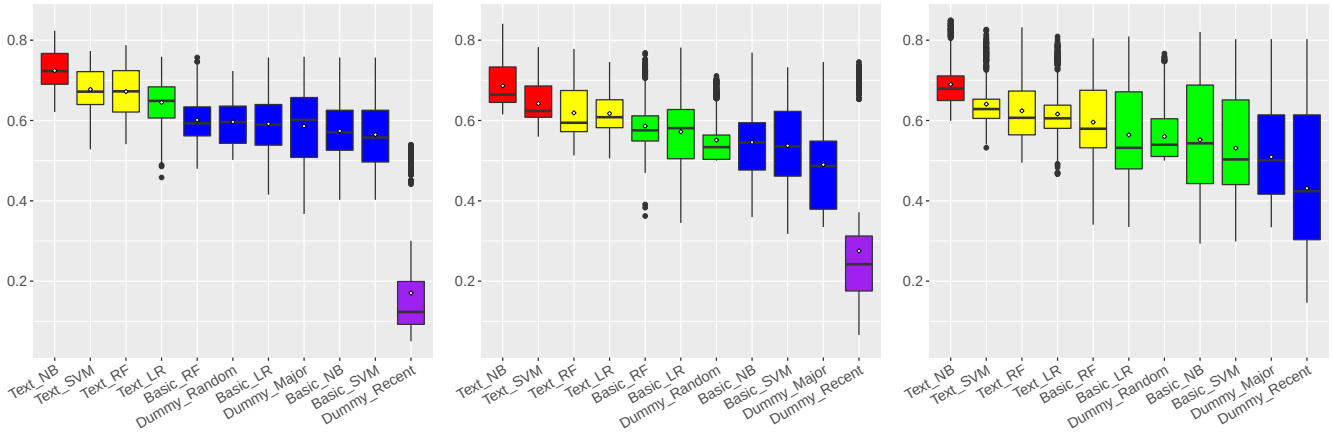
Fig. 3: An example of dummy recent and dummy major classifiers at CXF 25% cutoff point.

overlaps. Thus, if a project's cutoff point 25% equals to 50% or cutoff point 50% equals to 75%, we plus 1 LLC for the latter cutoff point.

Cross Validation. Two-fold cross validation is used to train and test the prediction model in the 10 studied projects. For a certain project, we break its data into two folds, one for train set and another for test set. In this experiment, we run two-fold cross validation for 50 times and totally get 100 prediction results for each cutoff point of each project.

Prediction Classifiers. To build our bug lifecycle prediction model, we adopt the following four commonly-used supervised classifiers: Naive Bayes(NB), Support Vector Machine(SVM), Logistic Regression(LR), and Random Forest(RF). In our experiments. we use the implementations of these classifiers in scikit-learn, a free software machine learning library for the Python programming language [28].

Dummy Classifiers. The dummy classifiers are used as the baseline classifiers in this experiment. A dummy random classifier is that all prediction classes are randomly guessed, which can also achieve a certain prediction accuracy. Another two dummy classifiers are the dummy recent classifier for predicting all bug introduced in recent version class, and the dummy major classifier for predicting all bug reports introduced in the major class of train set. Figure 3 presents an example of dummy recent and dummy major classifiers at CXF 25% cutoff point. For the dummy recent classifier, it predict all the test set as LLC$<$4 because the recent class is LLC$<$4. For the major recent classifier, it predict all the test set as LLC$\geq$4 because the major class of train set is LLC$\geq$4.



(a) Cutoff=0.25 (b) Cutoff=0.50 (c) Cutoff=0.75

Fig. 4: The boxplots of Weighted Average F-measure values of all studied combination of features and classifiers. Different colors represents different ranks (red>yellow>green>blue>purple) calculated from double Scott-Knott Tests.

Combination of Features and Classifiers. Here we make a brief conclusion of different features and classifiers mentioned above: there are 11 combinations of features and classifiers we will use in the following experiment. First, 3 dummy prediction models: Dummy_Random for randomly guess predict class, Dummy_Recent for guess all predict class as recent class, Dummy_Major for guess all predict class as major class. Moreover, using basic metrics as prediction features, there are 4 basic feature based prediction models: Basic_NB, Basic_SVM, Basic_LR, Basic_RF. Finally, using textual metrics as prediction features, there are 4 text based prediction models: Text_NB, Text_SVM, Text_LR, Text_RF.

C. Evaluation

F-measure. To evaluate the performance, we use the macro-average measurement [14], [29]. It is also commonly known as the metric weighted average F-measure. The F-measure can be calculated by the Precision and the Recall. For project j, the F-measure of m-th class $F_{m,j}$ can be defined as:

$$F_{m,j} = 2 \times \frac{precision_{m,j} \times reccall_{m,j}}{precision_{m,j} + reccall_{m,j}}.$$

Considering the class size, the weighted average F-measure of project j can be calculated as:

$$F_j = \frac{\sum_m F_{m,j} \times M_{m,j}}{\sum_m M_{m,j}}.$$

where $M_{m,j}$ is the whole number of bug reports in m-th class and the m equals to the class number.

Scott-Knott Test. To compare the performance of dummy, basic feature and text feature bug lifecycle prediction models, we use the Scott-Knott test [30]. The Scott-Knott test uses hierarchical cluster analysis to recursively group classification techniques into statistically distinct ranks. In this paper, we use weighted F-measure as the performance measure. If two groups have statistically significant difference of weighted F-measure, the Scott-Knott will execute again to further divide the ranks. The test terminates when there is no statistically distinct groups can be created [30].

Cliffs Delta δ. To quantify the improvement of performance on our textual feature bug lifecycle prediction model compared with the baseline models (dummy model and basic feature model), we introduce Cliff's delta δ [31]. The improvement magnitude is usually assessed by the thresholds: $|\delta| < 0.147$ negligible, $0.147 \leqslant |\delta| < 0.330$ small, $0.330 \leqslant |\delta| < 0.474$ medium, $|\delta| \geqslant 0.474$ large.

IV. Prediction Result

This section gives the result of our proposed textual feature based bug lifecycle prediction model comparing with the basic feature prediction model and dummy prediction model. First, we do a Scott-Knott test to show the performance of different prediction models in 3 cutoff points 25%, 50% and 75%. Moreover, we give the detail table of the result and calculate the Cliffs delta δ to compare the performance of the best method (from SK test it is text Naive Bayes method) with the others in each run of 50 times cross-validation.

Scott-Knott Test Result. To address our prediction result, Figure 4 presents an overview of our Scott-Knott test approach in dummy, basic feature and text feature bug lifecycle prediction models in 3 cutoff points 25%, 50% and 75%. We performed a double Scott-Knott test [32] to achieve the goal of generating statistically distinct groups. In the first run, the Scott-Knott test is run over each project and get a rank value for each project of 50 times cross-validation runs. In the second run, we put the Scott-Knott ranks of each project into another Scott-Knott test to get the final statistically distinct ranks of different prediction models.

Figure 4 shows that in each cutoff point, our proposed 4 text feature prediction models (Text_NB, Text_SVM, Text_LR, Text_RF) performs better than other baseline models.

TABLE II: F-measure means and Effect Sizes for each cutoff point in different projects. Numbers without parentheses are means and those with parentheses are Cliff's Delta δ comparative to Text Naive Bayes Method.

	Project	Dummy			SVM		LR		RF		NB	
		Random	Recent	Major	Basic	Text	Basic	Text	Basic	Text	Basic	Text
25%	CXF	0.688(-1.00)	0.063(-1.00)	0.721(-1.00)	0.714(-1.00)	0.735(-1.00)	0.714(-1.00)	0.722(-1.00)	0.714(-1.00)	0.753(-0.95)	0.714(-1.00)	**0.779**
	Flink	0.535(-1.00)	0.199(-1.00)	0.488(-1.00)	0.450(-1.00)	0.626(-0.93)	0.497(-1.00)	0.569(-0.99)	0.544(-1.00)	0.602(-1.00)	0.453(-1.00)	**0.674**
	Flume	0.539(-1.00)	0.499(-1.00)	0.499(-1.00)	0.630(-0.98)	0.655(-0.93)	0.651(-0.97)	0.581(-1.00)	0.615(-1.00)	0.622(-0.99)	0.610(-0.99)	**0.722**
	Groovy	0.560(-1.00)	0.161(-1.00)	0.542(-1.00)	0.542(-1.00)	0.653(-1.00)	0.560(-1.00)	0.648(-1.00)	0.584(-1.00)	0.664(-1.00)	0.558(-1.00)	**0.696**
	Hadoop	0.572(-1.00)	0.147(-1.00)	0.563(-1.00)	0.480(-1.00)	0.644(-0.88)	0.597(-0.99)	0.594(-1.00)	0.624(-0.99)	0.634(-0.97)	0.618(-0.99)	**0.693**
	NiFi	0.690(-1.00)	0.062(-1.00)	0.723(-0.99)	0.635(-1.00)	0.740(-0.88)	0.643(-1.00)	0.723(-0.99)	0.652(-1.00)	0.754(-0.63)	0.635(-1.00)	**0.773**
	OpenJPA	0.626(-1.00)	0.100(-1.00)	0.644(-1.00)	0.527(-1.00)	0.691(-1.00)	0.529(-1.00)	0.656(-1.00)	0.542(-1.00)	0.700(-1.00)	0.533(-1.00)	**0.753**
	PDFBox	0.507(-1.00)	0.272(-1.00)	0.399(-1.00)	0.607(-0.98)	0.625(-0.92)	0.614(-0.97)	0.616(-0.96)	0.597(-1.00)	0.580(-1.00)	0.570(-1.00)	**0.650**
	Tuscany	0.616(-1.00)	0.107(-1.00)	0.631(-1.00)	0.496(-1.00)	0.728(-0.98)	0.541(-1.00)	0.681(-1.00)	0.556(-1.00)	0.723(-0.99)	0.485(-1.00)	**0.778**
	Wicket	0.633(-1.00)	0.095(-1.00)	0.653(-1.00)	0.565(-1.00)	0.675(-0.99)	0.570(-1.00)	0.662(-1.00)	0.585(-1.00)	0.689(-0.97)	0.565(-1.00)	**0.722**
	Ave.	0.597	0.170	0.586	0.565	0.677	0.592	0.645	0.601	0.672	0.574	**0.724**
50%	CXF	0.542(-1.00)	0.186(-1.00)	0.506(-1.00)	0.490(-1.00)	0.607(-1.00)	0.490(-1.00)	0.578(-1.00)	0.545(-1.00)	0.597(-1.00)	0.491(-1.00)	**0.661**
	Flink	0.523(-1.00)	0.223(-1.00)	0.457(-1.00)	0.452(-1.00)	0.610(-0.91)	0.503(-1.00)	0.566(-0.99)	0.550(-1.00)	0.587(-1.00)	0.455(-1.00)	**0.655**
	Flume	0.672(-1.00)	0.701(-0.99)	0.701(-0.99)	0.673(-1.00)	0.728(-0.90)	0.712(-0.95)	0.702(-0.99)	0.704(-0.98)	0.732(-0.89)	0.705(-0.99)	**0.787**
	Groovy	0.501(-1.00)	0.357(-1.00)	0.357(-1.00)	0.547(-1.00)	0.610(-1.00)	0.586(-1.00)	0.607(-1.00)	0.572(-1.00)	0.574(-1.00)	0.553(-1.00)	**0.643**
	Hadoop	0.507(-1.00)	0.274(-1.00)	0.397(-1.00)	0.585(-0.99)	0.618(-0.84)	0.593(-0.99)	0.595(-0.94)	0.577(-1.00)	0.565(-1.00)	0.580(-1.00)	**0.653**
	NiFi	0.650(-1.00)	0.084(-1.00)	0.675(-0.99)	0.635(-1.00)	0.696(-0.89)	0.641(-1.00)	0.675(-0.99)	0.650(-1.00)	0.709(-0.75)	0.635(-1.00)	**0.737**
	OpenJPA	0.548(-1.00)	0.177(-1.00)	0.519(-1.00)	0.447(-1.00)	0.622(-1.00)	0.544(-1.00)	0.576(-1.00)	0.560(-1.00)	0.617(-1.00)	0.546(-1.00)	**0.694**
	PDFBox	0.501(-1.00)	0.312(-1.00)	0.355(-1.00)	0.628(-0.74)	0.631(-0.84)	0.634(-0.69)	0.628(-0.89)	0.609(-0.98)	0.576(-1.00)	0.572(-1.00)	**0.649**
	Tuscany	0.565(-1.00)	0.156(-1.00)	0.550(-1.00)	0.486(-1.00)	0.699(-0.98)	0.536(-1.00)	0.655(-1.00)	0.552(-1.00)	0.677(-1.00)	0.478(-1.00)	**0.747**
	Wicket	0.504(-1.00)	0.286(-1.00)	0.383(-1.00)	0.428(-1.00)	0.601(-0.99)	0.482(-1.00)	0.590(-1.00)	0.540(-1.00)	0.557(-1.00)	0.442(-1.00)	**0.637**
	Ave.	0.551	0.276	0.490	0.537	0.642	0.572	0.617	0.586	0.619	0.546	**0.686**
75%	CXF	0.502(-1.00)	0.303(-1.00)	0.365(-1.00)	0.380(-1.00)	0.591(-1.00)	0.424(-1.00)	0.587(-1.00)	0.587(-1.00)	0.550(-1.00)	0.394(-1.00)	**0.633**
	Flink	0.501(-1.00)	0.349(-1.00)	0.350(-1.00)	0.443(-1.00)	0.604(-0.90)	0.498(-1.00)	0.593(-0.93)	0.508(-1.00)	0.549(-1.00)	0.419(-1.00)	**0.641**
	Flume	0.721(-1.00)	0.756(-0.96)	0.756(-0.96)	0.751(-0.96)	0.767(-0.90)	0.760(-0.94)	0.756(-0.96)	0.753(-0.96)	0.775(-0.85)	0.772(-0.90)	**0.825**
	Groovy	0.612(-1.00)	0.624(-1.00)	0.624(-1.00)	0.655(-1.00)	0.648(-1.00)	0.675(-0.98)	0.639(-1.00)	0.683(-0.96)	0.683(-1.00)	0.707(-0.19)	**0.710**
	Hadoop	0.513(-1.00)	0.426(-1.00)	0.426(-1.00)	0.444(-1.00)	0.592(-0.98)	0.497(-1.00)	0.549(-1.00)	0.564(-1.00)	0.562(-1.00)	0.515(-1.00)	**0.652**
	NiFi	0.554(-1.00)	0.169(-1.00)	0.530(-1.00)	0.457(-1.00)	0.624(-0.99)	0.474(-1.00)	0.563(-1.00)	0.523(-1.00)	0.613(-1.00)	0.457(-1.00)	**0.687**
	OpenJPA	0.530(-1.00)	0.477(-1.00)	0.477(-1.00)	0.530(-1.00)	0.623(-0.99)	0.543(-1.00)	0.586(-1.00)	0.565(-1.00)	0.602(-1.00)	0.579(-1.00)	**0.677**
	PDFBox	0.512(-1.00)	0.423(-1.00)	0.423(-1.00)	0.539(-0.99)	0.623(-0.94)	0.601(-0.99)	0.603(-1.00)	0.595(-0.99)	0.576(-1.00)	0.564(-1.00)	**0.652**
	Tuscany	0.553(-1.00)	0.170(-1.00)	0.529(-1.00)	0.440(-1.00)	0.693(-0.99)	0.482(-1.00)	0.656(-1.00)	0.495(-1.00)	0.668(-1.00)	0.423(-1.00)	**0.747**
	Wicket	0.605(-1.00)	0.615(-1.00)	0.615(-1.00)	0.674(-0.70)	0.640(-1.00)	0.690(-0.29)	0.628(-1.00)	0.686(-0.41)	0.664(-0.95)	0.692(-0.19)	**0.697**
	Ave.	0.560	0.431	0.509	0.531	0.641	0.564	0.616	0.596	0.624	0.552	**0.692**

Moreover, the text Naive Bayes classifier performs the best across every cutoff point. In cutoff points 25% and 75%, text Naive Bayes method has been divided into a single class (the best class) by double Scott-Knott test. Although in 50% cutoff point, text Naive Bayes method is not grouped into a class individually, but it also performs best comparing to the other methods. Thus, in the following table, we use text Naive Bayes method as the chosen method in Cliffs delta δ calculation.

Detail Result with Cliffs Delta δ. The detail result of weighted F-measure is presented in table II, which is calculated by 50 times 2-fold cross-validation. The first column is the class cutoff point. The second column is the 10 projects we used in this experiment. The rest columns present the performance of the bug lifecycle prediction model in 3 dummy classifiers and 4 traditional classifiers (text and basic features).

In general, all text feature based classifiers can lead a weighted F-measure higher than 0.6. The text Naive Bayes method outperforms the related methods in every project and average value. For example, for predicting if the bug lifecycle is greater or less than 25% cutoff point of CXF's LLC, our text based Naive Bayes method achieves the weighted F-measure 0.779, which is higher than the other methods' results. In the view of all average results of 3 cutoff points, text based methods are all better than correlated basic methods and dummy methods. Although the basic feature methods perform better than other classifiers with text features in some projects, they can not exceed the text Naive Bayes method. In the view of Cliffs delta δ, most of the methods performs large magnitude ($|\delta| \geqslant 0.474$) compared to text Naive Bayes method, which means the text Naive Bayes method performs significantly better than other methods in each run of 50 times cross-validation of each project.

We observe that the text based methods perform better than basic methods and dummy methods in average of 10 studied projects. The text based Naive Bayes bug lifecycle prediction model outperforms all other prediction models.

V. Threats to Validity

This study provides a text feature based prediction model to predict the bug lifecycle. However, there are some threats to validity that should be taken into consideration.

Internal validity. All the ten data sets used in our experiments are well-known open source projects of Apache Software Foundation in Jira. The performance of bug lifecycle prediction in commercial projects may be different from the open source projects. Moreover, the ten open source projects are all Java projects where other programming language may have some differences in the character of text feature. We will study more projects in the future, including the commercial projects and the projects in other languages.

External validity. Our bug lifecycle prediction model requires that the projects have sufficient historical bug reports with affect version to train the prediction model. However, these information could be limited in some projects. Another external validity of our experiment is that the bug reports provide is the affect version not the exact bug introducing time. The commercial data sets may be more accurate in this bug introducing time compared to the open source projects.

Reliability validity. All the ten projects are publicly available in Apache Software Foundation of Jira. Any researchers who intend to replicate this study can get the data sets in

Apache Software Foundation of Jira. Moreover, the python implementation of our experiment will be provided online.

VI. Conclusion

Bug lifecycle prediction aims to know how long does a bug exists in software during it first be introduced and finally be fixed. It can help developers to efficiently revisit the bug introducing code. The former work mainly focus on fixing period or dormant period, and use features obtained from defect ranking, source code and CVS log to do the analysis. In this study, we consider the bug lifecycle as a whole and focus on text features (summary and description of bugs) in bug reports to construct prediction model.

Our evaluation on 10 well-know Apache Software Foundation projects with 57000+ bugs shows that our model achieves a good performance in predicting the bug lifecycle. We examine the results with 3 evaluation measurements: Weighted F-measure, Scott-Knott test, and Cliffs Delta δ. The Scott-Knott test and Cliffs Delta δ present that the text Naive Bayes method outperforms all the rest methods. In general, all text feature based classifiers can lead a weighted F-measure higher than 0.6. In the future, we intend to extend our text feature based model to more projects in bug lifecycle prediction and introduce more methods to train the prediction model.

Acknowledgement

This work was supported by the Natural Science Foundation of Jiangsu Province of China (Grant No. BK20140611), the National Natural Science Foundation of China (Grant No. 61403187, 61772259, 61472175, 61472178, 61772263, 61472077).

References

[1] L. Erlikh, "Leveraging legacy system dollars for e-business," *It Professional*, vol. 2, no. 3, pp. 17–23, 2000.

[2] S. Kim, T. Zimmermann, E. J. W. Jr., and A. Zeller, "Predicting faults from cached history," in *Proceedings of the 1st Annual India Software Engineering Conference*, 2008, pp. 15–16.

[3] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In, "Micro Interaction Metrics for Defect Prediction," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, Szeged, Hungary, 2011, pp. 311–321.

[4] J. Wang, B. Shen, and Y. Chen, "Compressed C4.5 models for software defect prediction," in *Proceedings of 12th International Conference on Quality Software*, 2012, pp. 13–16.

[5] W. Abdelmoez, M. Kholief, and F. M. Elsalmy, "Bug fix-time prediction model using naive bayes classifier," in *International Conference on Computer Theory and Applications*, 2012, pp. 167–172.

[6] Y. Liu, Y. Li, J. Guo, Y. Zhou, and B. Xu, "Connecting software metrics across versions to predict defects," in *25th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2018, pp. 232–243.

[7] E. J. Weyuker, T. J. Ostrand, and R. M. Bell, "Using developer information as a factor for fault prediction," in *Third International Workshop on Predictor Models in Software Engineering*, 2007, pp. 8–8.

[8] V. S. Sinha, S. Sinha, and S. Rao, "Buginnings: identifying the origins of a bug," in *India Software Engineering Conference*, 2010, pp. 3–12.

[9] T. H. Chen, M. Nagappan, E. Shihab, and A. E. Hassan, "An empirical study of dormant bugs," in *Working Conference on Mining Software Repositories*, 2014, pp. 82–91.

[10] R. K. Saha, S. Khurshid, and D. E. Perry, "An empirical study of long lived bugs," in *Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering*, 2014, pp. 144–153.

[11] P. Bhattacharya and I. Neamtiu, "Bug-fix time prediction models: Can we do better?" in *Working Conference on Mining Software Repositories*, 2011, pp. 207–210.

[12] F. Zhang, F. Khomh, Y. Zou, and A. E. Hassan, "An empirical study on factors impacting bug fixing time," in *19th Working Conference on Reverse Engineering*, 2012, pp. 225–234.

[13] Q. Song, M. Shepperd, M. Cartwright, and C. Mair, "Software defect association mining and defect correction effort prediction," *IEEE Transactions on Software Engineering*, vol. 32, no. 2, pp. 69–82, 2006.

[14] H. Zhang, L. Gong, and S. Versteeg, "Predicting bug-fixing time: An empirical study of commercial software projects," in *International Conference on Software Engineering*, 2013, pp. 1042–1051.

[15] A. Rajaraman and J. D. Ullman, *Mining of massive datasets*. Cambridge University Press, 2012.

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

[17] G. Gousios, B. Vasilescu, A. Serebrenik, and A. Zaidman, "Lean ghtorrent: Github data on demand," in *Working Conference on Mining Software Repositories*, 2014, pp. 384–387.

[18] T. Zimmermann and A. Zeller, "When do changes induce fixes?" in *International Workshop on Mining Software Repositories*, 2005, pp. 1–5.

[19] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead, "Automatic identification of bug-introducing changes," in *IEEE/ACM International Conference on Automated Software Engineering*, 2006, pp. 81–90.

[20] R. Wu, H. Zhang, S. Kim, and S. C. Cheung, "Relink: recovering links between bugs and changes," in *ACM Sigsoft International Symposium on the Foundations of Software Engineering*, 2011, pp. 15–25.

[21] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Multi-layered approach for recovering links between bug reports and fixes," in *ACM Sigsoft International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.

[22] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in *International Workshop on Mining Software Repositories*, 2007, pp. 1–1.

[23] U. Raja, "All complaints are not created equal: text analysis of open source software defect reports," *Empirical Software Engineering*, vol. 18, no. 1, pp. 117–138, 2012.

[24] A. Said, M. Borg, and D. Pfahl, "Using text clustering to predict defect resolution time: a conceptual replication and an evaluation of prediction accuracy," *Empirical Software Engineering*, vol. 21, no. 4, pp. 1437–1475, 2016.

[25] R. Kikas, M. Dumas, and D. Pfahl, "Using dynamic and contextual features to predict issue lifetime in github projects," in *Mining Software Repositories*, 2016, pp. 291–302.

[26] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," in *International Workshop on Recommendation Systems for Software Engineering*, 2010, pp. 52–56.

[27] L. Marks, Y. Zou, and A. E. Hassan, "Studying the fix-time for bugs in large open source projects," in *International Conference on Predictive MODELS in Software Engineering*, 2011, pp. 1–8.

[28] F. Pedregosa, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, and J. Vanderplas, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, no. 10, pp. 2825–2830, 2012.

[29] I. H. Witten, E. Frank, and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques (Third Edition)*. Morgan Kaufmann Publishers Inc., 2011.

[30] E. Jelihovschi, J. Faria, and I. Allaman, "Scottknott: a package for performing the scott-knott clustering algorithm in r," *Tema*, vol. 15, no. 1, pp. 3–17, 2014.

[31] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, "Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohen's d indices the most appropriate choices?" in *Annual meeting of the Southern Association for Institutional Research*, 2006, pp. 1–24.

[32] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, 2015, pp. 789–800.