Predicting Bug-Fixing Time with Rating Features - A Comparison Between Ratings and Reputations

Bruno Rafael de Oliveira Rodrigues
FUMEC University
Belo Horizonte, MG, Brazil
brunorodriguesti@yahoo.com.br

José Maurício Costa
FUMEC University
Belo Horizonte, MG, Brazil
jose.costa@fumec.br

ABSTRACT

Predicting bug-fixing time plays an important role in allowing a software manager and team to make decisions about allocation of resources, prioritization and scheduling. Estimating the time to fix a bug is not a simple task. In the literature, machine learning (ML) models have been proposed to help software managers decide whether a bug might be fixed now or later. One feature highlighted in ML models for predicting bug-fixing time is reporter reputation. However, these features are based on the participation of the reporter or developer in the project, but do not take into account the time taken to fix the bugs. In this study, we propose new two features called "reporter rating" and "developer rating." Unlike reputations, ratings are based on the time taken to fix a bug. In this study, we carried out an experiment in two datasets containing bug reports. We ran the reputation and rating features in ten ML models and compared the results. Additionally, we verified the features together and combined them with textual features. As a result, we found that ratings can improve the performance of the models. Ratings had the best results in probabilistic models, while reputation was better in models that use the decision tree approach. When used together, reputations and ratings do not substantially increase the performance of the models when compared to individual results. However, ratings improve performance when combined with textual features more than reputations.

KEYWORDS

bug fix time, machine learning, reputation, predict

1 INTRODUCTION

Predicting bug-fixing time plays an important role in Software Maintenance and Software Quality. Accurate predictions make it possible to allocate resources correctly, plan the project schedule and estimate costs. Usually, the estimation of the time it takes to fix a bug is given by developers and is supported by their intuition and experience. In the literature, studies have proposed the application of Machine Learning (ML) to predict bug-fixing time. One feature highlighted by studies in the literature is the bug-opener's reputation (reporter's reputation) [1], also called submitter reputation [2]. Studies have shown that the person who opens the bug has a positive impact on predicting bug-fixing time in ML models [3–6]. Rodrigues and Parreiras [7] have demonstrated that collaborative filtering can be utilized to predict bug-fixing time. In this approach, the users (reporter and developer) are the focal point of predictions. The algorithms employing collaborative filtering outperformed those that do not incorporate collaborative filtering.

In addition to reporter reputation, textual approaches have been used to predict bug-fixing time. This approach enables models to predict the time to fix a bug through the summary description and comments of bug reports [8, 9]. Predicting bug-fixing time can be seen as a classification problem [10], where the system suggests whether the bug can be fixed quickly or slowly, helping the software teams prioritize the tasks, where they may select which bug can be fixed first. Classifying bugs as fast or slow aids software managers in deciding if the bug can be fixed now or later.

In this paper, we proposed two new features called "reporter rating" and "developer rating." The rating is a score from 1 to 5 based on a bug report. The ratings are computed by quartiles of bug resolution time. For instance, if a reporter submitted a bug report that was resolved in less than one day, we score it as a 5, as in a five-star service. If the bug report was resolved in more than the median bug-resolution time, 75% of the quartile, we scored it as a 1. We use ML techniques to classify whether the bug will be fixed quickly or slowly using reporter and developer ratings. We compared the performance of our proposed features with the concept of reporter and developer reputations. Reporter and developer reputations are features based on the number of previously fixed bug reports. Their reputation helps us know whether the bug will be fixed or not. The more bug reports are submitted and fixed, the higher the reputation score will be [1]. In contrast to this approach, we proposed features based on the time taken to fix previous bugs. Accordingly, we intend to answer the following research questions:

(1) RQ1: Is reporter rating superior to the reporter reputation feature in supervised machine learning?
(2) RQ2: Is developer rating superior to the developer reputation feature in supervised machine learning?
(3) RQ3: Can the combination of reporter and developer ratings and reputations improve the performance of the models?
(4) RQ4: Can reporter and developer ratings, when combined with text features, increase the performance of predictions when compared to reporter and developer reputations?

To address these inquiries, we expand upon the work of Rodrigues and Parreiras [7] by conducting an experiment using bug reports from the NetBeans project and Eclipse Platform as datasets. We applied the reputation and rating features in ten supervised ML techniques. In our experiment, we found that reporter reputation is a feature that performed well in Decision tree and Random Forest algorithms while the reporter and developer rating have the best results in Gradient Boosting and Logistic Regression. The combination of the features can improve the performance of the models. Ratings may improve the performance of the models when combined with text features. This paper contributes with the proposed of two new features that can be used in ML models. In addition, this work contributes to improving the understanding of the impact of reporter and developer on predicting bug-fixing time models.

The rest of the paper is organized as follows: Section 2 presents the bakcground foundations. Section 3 presents related studies. Section 4 describes the proposed features. Section 5 shows the experimental methods in this study. Section 6 shows the results of the experiment. Section 7 presents the discussion of the study. Section 8 shows the threats to the validity of the research. Finally, Section 9 provides conclusions and future studies.

2 BACKGROUND

2.1 Bug Life Cycle

The life cycle of a bug comprises three phases: bug understanding, bug triage and bug fixing [11]. In bug understanding, the manager filters the bug to be fixed. In bug triage, the manager assigns the bug to a developer. If the bug was misunderstood, the likelihood of the bug being reassigned is high. In the bug-fixing phase, the developer must find and fix the bug in the source code [11].

2.2 Machine Learning

Machine learning is a multidisciplinary area comprising disciplines such as biology, statistics, mathematics and physics. The main idea of machine learning is to make computers learn [12]. In this work we use the supervised learning. This method uses algorithms trained by sets of examples that have correct responses or targets and try to respond correctly to all possible inputs. The algorithms in this category can treat the problem as *Regression* that uses continuous data, or *Classification* that uses discrete data [12].

3 RELATED WORK

Predicting bug-fixing time using reputation as a feature is recurrent in the literature. Guo et al. [1] carried out an experiment to estimate the likelihood that a given bug will be fixed. They noticed that not all bugs that are reported are fixed. They proposed a statistical model that was applied to Windows Vista and Windows 7 projects to identify the bugs that will be fixed. Their model uses logistic regression, which obtained 68% precision and a recall of 64%. They also conducted a survey with Microsoft employers in order to get their perception regarding the factors that influence the bugs that can be fixed. In their findings, they highlight the influence of the person who opens the bug, also called ''bug opener reputation," corroborating with the work of Hooimeijer and Weimer [2], and developer reputation, where many reassignments decrease the likelihood that a bug will get fixed. In their work, the authors found that the person who opens the bug has an influence on whether the bug will be fixed or not and the time taken to fix it. In our study, we evaluated reporter reputation and proposed features for predicting the time needed to fix a bug based on the reporter and developer, but we did not evaluate if the bug will be fixed or not.

Yücel and Tosun [13] proposed a novel reporter reputation scoring method. Differing from Guo [1] and Zhang et al. [3], the authors utilize temporal reputation changes and Doc2Vec to train the textual information. They compare the new reputation against the reputations mentioned in the literature by applying two open-source projects, Chromium and WebRTC, and employing two machine learning techniques: Stochastic Gradient Descent (SGD) and Extreme Gradient Boost (XGB). In our study, we also propose new features based on the reporter and developer and compare them to the reporter reputation in the literature. Unlike Yücel and Tosun [13], we classify the reporter and developer based on the time taken to resolve the bug report. Additionally, we experiment with the new features together with textual information, using Term Frequency-Inverse Document Frequency (TF-IDF). Furthermore, we evaluate the performance of ten machine learning techniques to analyze the behavior of our proposed features.

Peralta et al. [6] investigate the features present in bug reports to predict bug-fixing time. They employed Bayesian Network to model the relationship between the variables of the bug reports, using data from Mozilla as their dataset. Their findings highlight attachments and the reporter's reputation as variables relevant for predicting whether the bug will be fixed, while severity and comments are crucial for predicting quick bug-fixing. Similar to Peralta et al. [6], our work analyzes bug-fixing time and utilizes reporter reputation as a significant feature. However, we propose two new features based on the developer and reporter.

4 RATING FEATURES

In this study, we proposed a feature named "rating score." The reporter and developer's reputation features are based on the number of bugs informed by the reporter or assigned to the developer that were fixed. We understand that information about the time taken to fix previous bugs related to reporters and developers can be used in models to increase the performance of predictions of bug-fixing time. In other words, we proposed features based on the time taken to fix bugs and not on the number of bugs fixed to predict bug-fixing time. Table 1 shows classification details. We consider bugs fixed with a rating of 3 or less to be fast. With this information, the software manager can choose whether or not to prioritize a bug.

In our proposed feature, the rating score can be related to the reporter or a developer. For example, if the median bug-fix time is 15 days and a developer fixed a bug in 10 days, the rating is 4. Both the reporter and the developer were classified as 4 for their work. We expect that information in the bug report can make models more accurate after adding information about classification times used by the reporter and developer for previous bugs. The rating score can be combined with other features like textual features or priority and help the models predict time more precisely.

Before computing the rating of the reporter and developer, it is necessary to filter the bugs that were fixed and remove the outliers. The outliers can raise or lower the median bug-fixing time of the project. For example, depending on the dataset, the median number of days taken to fix a bug can be 27. After treating the outliers it can be 17. Details about outliers is shown in section 5.1.

5 EXPERIMENT

To answer the research questions, we carried out an experiment using the Eclipse Platform and NetBeans datasets. These datasets are available in MSR 2011 [14] and the work of Rodrigues and Parreiras [7]. Eclipse and NetBeans are two popular open-source projects for integrated development environments (IDE). These datasets contain bug reports stored in MySQL format. Both the Eclipse and Netbeans projects use Bugzilla as their bug-issue tracking system. Thus, the structures of these datasets are similar. Although the data utilized in this experiment originates from a dataset

Table 1: Rating Score

Rating	Time in days
5	less than or equal to 1 day to fix
4	greater than 1 and less than or equal to 25% of percentile
3	greater than 25% and less than or equal to 50% of percentile
2	greater than 50% and less than or equal to 75% of percentile
1	greater than 75%

dating back to 2011, our objective is to assess the features: ratings and reputations. By doing so, we understand that the features get the context of the team independent of the technology used in the project. In other words, the features are adapted to the reality of each project in each epoch. After getting the datasets, we first explored the database, selecting attributes from the bug reports of our experiments and exporting them in CSV format. Next, attributes from Bugzilla were selected which were initially noted when the person opened a bug. Our goal is to predict bug-fixing time at the beginning of the triage.

We used Google Colab to carry out the experiment. Google Colab is an environment prepared to execute data analyses directly in a browser [1]. We selected ML algorithms present in bug-fixing time prediction literature: Random Forest, Logistic Regression, KNN, Naive Bayes Gaussian, Naive Bayes Multinomial, SVM, MultiLayer Perceptron, SGD Classifier and Gradient Boosting [3, 7, 10, 15].

Data exploration was performed with Python (version 3.6.9) and the libraries NumPy (version 1.18.5), Pandas (version 1.0.5) and Matplotlib (version 3.2.2). These Python libraries are used to understand the data, remove the outliers and discretize the data. The code used in this experiment is available at https://github.com/brunorodriguesti/BFT_Rating_Features.

The following steps were followed to perform the experiment. First, we obtained the datasets and analyzed them according to their descriptions in sections 5.4. Second, we removed outliers described in section 5.1. We consider outliers to be bugs that are fixed in a time of 0 days or that take over 75% of quartile. Third, we classify bugs as fast or slow using the median time taken to resolve them according to the formula 2. We balanced the dataset so it had the same number of fast and slow bugs. Next, we computed the reporter and developer rating scores, as well as the reporter and developer reputations. The ratings were computed according to Table 1 and the reputation according to the formula 1 [1, 2]. We also calculated average ratings in order to verify if the average could be a relevant feature. The ratings and reputations were calculated according to each bug at that specific moment in time. For example, the first bug that the reporter submitted received a rating of 3, because we had no prior information about the bug. The same goes for a reputation which received a first score of 0. In this way, the next time that we have a bug report submitted by a given reporter or assigned to a given developer, we already know what the real rating and reputation were before computing the next rating and reputation. In other words, we created a column in our datasets with information about the ratings and reputations known at that moment in time and a column with the real rating and reputation used to compute the next rating and reputation.

$$reputation = \frac{|OPENED \bigcap FIXED|}{|OPENED + 1|} \quad (1)$$

5.1 Outliers and Classification

In our experiment, we only considered bugs that had already been resolved, thus, we used bugs with a status resolution of fixed. The bug-fix time was calculated in days. For example, if bugs were resolved on the same day, they have zero days of resolution.

In treating the outliers, we removed the 75% quartile and the bugs whose resolution were equal to zero days. The quality of the data and the accuracy of the models was improved by removing the outliers [16]. On analyzing our dataset, we noticed that the bug reports with zero day resolutions were not always bugs. These reports are tests of environment or bugs resolved prior to reporting, in other words, they were reported only for registration purposes. We removed bugs over 75% quartile from the dataset as they were outliers that do not represent the reality of all types of bugs.

In order to classify the bugs as fast or slow, the approach proposed by Giger et al. [10] was used. The data were divided into fast and slow according to the formula 2.

$$BugClass = \begin{cases} Fast & days\ resolution <= median \\ Slow & days\ resolution > median \end{cases} \quad (2)$$

5.2 Experiment Execution

After removing the outliers, we used the following features in the supervised ML approach: *reporter id, the developer id of the person who fixed the bug, the priority, the severity, the product, the component, the operational system, the month of creation, the year of creation.* The priority and severity attributes were transformed into categorical features. These set of features we called initial features. We also use, *the title of bug and the first comment* as textual features. We then combined them with *reporter and developer reputation and reporter and developer ratings.* For textual features, we took the content of the title of the bug and description of the first comment and put them in one field. In this field, we extracted the bag of words and treated the bug report texts with the following steps [17]:

(1) convert all words to lower case
(2) remove links and html tags
(3) remove numbers and special characters
(4) remove stop words
(5) stem words

We built a document-terms matrix with the frequency of terms, removing one percent of the most infrequent terms. Next, we used term frequency (TF) and term frequency-inverse document frequency (TF-IDF) to create features for classification algorithms.

In our experiment, we tested the behavior of algorithms without the textual features and then with textual elements (title and comment) to classify the bug report as fast or slow. Thus, we ran the initial features and then the initial features with reputations, followed by the initial features with ratings, and finally, the initial

[1] https://colab.research.google.com/

features with both. We then followed the same steps with textual features.

The objective of this experiment is to classify whether a bug will be fixed quickly or slowly according to the initial attributes informed, thus, allowing a manager to decide if a bug should be fixed now or later, as quickly as possible. The exception attribute we used to test the models is the developer. We decide to select the developers that fixed the bug instead of those first assigned to the task because they directly affected the time taken. The problems of assigning the bug to the correct developer and reopened bugs are not treated in the scope of the experiment.

To evaluate the performance of the models, we used precision, recall and f-measure described in section 5.3. The tests were performed with 10-fold cross-validation.

5.3 Evaluation Metrics

In this paper we use Precision, Recall and F-measure metrics in order to evaluation the models. Precision, also called Confidence, is the ratio of predicted positive cases that are correctly true-positives, while the Recall or Sensitivity is the ratio of true-positive cases that are correctly predicted positive. The combination of precision and recall can be formulated by the F-measure, also called F-Score. The F-measure is the harmonic mean between the precision and the recall [18].

5.4 Dataset

In order to evaluate our features, an experiment was performed using bug tracking data from the following systems: Eclipse and Netbeans available from MSR 2011 [14] and used in the work of Rodrigues and Parreiras [7]. Only bug reports with the resolution status of "FIXED" were used in this experiment. We classified an equal number of bugs as "fast" and "slow" in the datasets. The Eclipse dataset has 24706 bugs and the Netbeans has 3068 bugs.

6 RESULTS

This section presents the results of our experiment and our answers to our research questions (RQ).

6.1 RQ1: Is reporter rating superior to the reporter reputation feature in supervised machine learning?

To answer this research question, we selected initial features like reporter, the developer who fixed the bug, priority, severity, product, component, operational system, the month of creation, and the year of creation. These features are available in the bug reports and are used to identify the type of bug report, we called these features as initial features. We ran the initial features in supervised ML algorithms and then the reputations with the initial features. Finally, we ran the rating features and the initial features.

Table 2 shows the results of our experiment in Eclipse Platform. Random Forest and Gradient Boosting show the best results. In Random Forest, the initial features and the reporter reputation had similar values. Gradient Boosting and SVM had results equal to all combinations of features. In these models, reputation and rating do not increase or decrease the performance. The reputation and rating features had results similar to the initial features, the exceptions were SGD Classifier, MultiLayer Perceptron, Logistic Regression and Naive Bayes Gaussian. The results of the features of reporter reputation and rating are close. However, in Logistic Regression, the reporter rating achieved 12% of F-measure more than reporter reputation.

Table 2: Average F-measure of reporter reputation and rating in Eclipse Platform

	Initial features	Reporter reputation	Reporter ratings
***Random Forest**	**67%**	**67%**	65%
Gradient Boosting	65%	65%	65%
***Decision Tree**	62%	**62%**	60%
****KNN**	**61%**	60%	**61%**
****SGD Classifier**	51%	51%	**56%**
MultiLayer Perceptron	44%	**52%**	46%
SVM	47%	47%	47%
****Logistic Regression**	47%	47%	**59%**
****Naive Bayes Multinomial**	42%	42%	**43%**
****Naive Bayes Gaussian**	35%	35%	**40%**

Different from the Eclipse project, the Naive Bayes Multinomial algorithm presented the best results in the NetBeans project.Table 3 shows the average F-measure of reporter reputation and rating in NetBeans. In our experiment, the Naive Bayes Multinomial, all combinations of features had the same results. Reporter reputation was better in Random Forest, Gradient Boosting, Decision Tree, KNN and SGD Classifier. As well as in the Eclipse dataset, the features when applied in NetBeans dataset show few variations, the expressive variation is present in Multilayer Perceptron which achieve the difference between reporter reputation and reporter ration of a F-Measure of 7%.

Table 3: Average F-measure reporter reputation and rating in NetBeans

	Initial features	Reporter reputation	Reporter ratings
Naive Bayes Multinomial	**65%**	**65%**	**65%**
***Random Forest**	60%	**63%**	60%
****Naive bayes Gaussian**	**62%**	61%	**62%**
****SVM**	60%	61%	**62%**
***Gradient Boosting**	58%	**60%**	59%
***Decision Tree**	59%	**60%**	56%
***KNN**	56%	**57%**	55%
**Logistic Regression	**57%**	56%	**57%**
***SGD Classifier**	**49%**	45%	42%
****Multilayer Perceptron**	34%	36%	**43%**

Reporter rating was better in probability techniques like Naive Bayes and Logistic Regression, while reporter reputation was better in tree-based machine learning like Decision Tree and Random Forest. Depending on the dataset, the reporter can be more influential in predicting bug-fixing time, i.e. the Eclipse project. Reporter rating performed better than reporter reputation with Logistic Regression and Naive Bayes Gaussian in both datasets, whereas, reporter reputation performed better in Random Forest and Decision Tree. In both datasets, reporter rating decreased the performance in the Decision Tree algorithm when compared to initial features.

The results of the ML algorithms show that the results are close between the initial features, reporter reputation and reporter rating. However the variation is superior using the reporter ratings in Logistic Regression (Eclipse) and Multilayer Perceptron (NetBeans).

6.2 RQ2: Is developer rating superior to the developer reputation feature in supervised machine learning?

To evaluate the performance of developer reputation and developer rating, we ran the experiment as described in the answer to RQ1 and Section 5.

Table 4 shows the average F-measure for developer reputation and rating in Eclipse Platform. Like the result for reporters, developers had the best results in Random Forest, Gradient Boosting and Decision Tree. The behavior of the ratings is similar. In both cases they decreased the results in Random Forest and Decision Tree. On the other hand, they were better in Gradient Boosting and Multilayer Perceptron. The results of KNN and SVM did not have a difference between reputation and rating features. Logistic Regression, Naive Bayes Gaussian and Naive Bayes Multinominal also had a greater F-measure for developer rating in relation to developer reputation and initial features. An expressive variation between the developer rating and developer reputation is present in the Logistic Regression with 19% of difference and Naive Bayes Gaussian with 11% of difference.

Table 4: F-measure of developer reputation and rating in Eclipse Platform

	Initial features	Developer reputation	Developer ratings
***Random Forest**	67%	**68%**	66%
****Gradient Boosting**	65%	65%	**67%**
***Decision Tree**	62%	**63%**	61%
KNN	**61%**	60%	60%
****SGD Classifier**	51%	*54%*	52%
***MulitLayer Perceptron**	44%	49%	**54%**
****Logistic Regression**	47%	47%	**66%**
SVM	**47%**	**47**	**47%**
***Naive Bayes Multinomial**	42%	42%	**43%**
****Naive Bayes Gaussian**	35%	36%	**47%**

Table 5 presents the results of developer reputation and rating in the NetBeans project. Naive Bayes Multinomial had the best results, however, the values of predictions are the same for all features tested. MultiLayer Perceptron, SGD Classifier, Gradient Boosting and Naive Bayes Gaussian had better results using developer ratings. MultiLayer Perceptron stood out from the others, with an average developer rating F-measure 10% greater than the average developer reputation.

Table 5: Average F-measure of developer reputation and rating in NetBeans

	Initial features	Developer reputation	Developer rating
Naive Bayes Multinomial	**65%**	**65%**	**65%**
****Naive bayes Gaussian**	*62%*	61%	**62%**
***SVM**	60%	**62%**	60%
***Random Forest**	60%	**61%**	60%
***Decision Tree**	59%	**60%**	57%
****Gradient Boosting**	58%	58%	**62%**
Logistic Regression	**57%**	56%	**57%**
***KNN**	56%	**57**	56
****SGD Classifier**	49%	44%	**51%**
****Multilayer Perceptron**	34%	43%	**53%**

In the Eclipse Platform and NetBeans projects, Gradient Boosting, SGD Classifier, MultiLayer Perceptron, Logistic Regression and Naive Bayes Gaussian performed better using developer rating than developer reputation, while developer reputation performed better in Random Forest and Decision Tree. We can see that developer reputation was better in Eclipse Platform using Random Forest. In Netbeans they had equals results in the Naive Bayes Multinominal. The result obtained by developer rating in Naive Bayes Gaussian and Gradient Boosting was the same as that obtained by developer reputation in SVM. In other words, developer ratings might be superior to reputations in five ML models, while the opposite is true in two ML models in both datasets. Comparing the difference between the results of developer reputation and developer rating the developer rating achieves superior F-measure. The best results of the developer reputations are close to initial features and developer ratings.

6.3 RQ3: Can the combination of reporter and developer ratings and reputations improve the performance of the models?

To answer this question, we first ran the models putting reporter and developer reputations together. Next, we ran the models combining reporter and developer ratings. Finally, we used reputations (reporter and developer reputation) and ratings together (reporter and developer ratings). Both models use the initial features.

Table 6 presents the average F-measure in Eclipse Platform. We can see that Random Forest and Gradient Boosting present the best results. In Random Forest, the average F-measure achieved the

same value obtained by developer reputation. Gradient Boosting achieved the same values as Random Forest when using the combination of reputations and ratings. In this case we can see that the combinations increase performance as the results show in Table 2 and 4. SGD Classifier also increased performance when it used a combination of reputations and ratings. Naive Bayes Gaussian increased performance with the combination of reporter and developer ratings and when reporter and developer reputations and ratings were added together.

Logistic Regression and Naive Bayes Gaussian achieved the best results using only ratings (reporter and developer ratings together), while Random Forest and Decision Three had the best results using only reputations (reporter and developer reputations).

Table 6: The Average F-measure of Features in Eclipse Platform

	Reputations	Ratings	Reputations and Ratings
Random Forest	**68%**	65%	66%
Gradient Boosting	65%	67%	**68%**
Logistic Regression	48%	**65%**	**65%**
Decision Tree	**63%**	60%	60%
KNN	60%	60%	60%
SGD Classifier	55%	44%	**58%**
MultiLayer Perceptron	52%	46%	**54%**
Naive Bayes Gaussian	36%	**52%**	51%
SVM	47%	47%	47%
Naive Bayes Multinomial	42%	42%	**43%**

Table 7 shows the average F-measure for the NetBeans project. In the NetBeans project, the Naive Bayes Multinomial algorithm had the best performance of all initial features. The combination of reputations and ratings was superior to the results of the Naive Bayes Gaussian, Logistic Regression and SGD Classifier. In these models the performance improved when compared to previous results presented in Table 3 and 5.

Gradient Boosting had the same performance using developer ratings and reporter ratings. SVM had the same performance when only the reporter ratings were tested. Differently, KNN and MultiLayer Perceptron had the best results only with developer ratings. In regard to reputation, combining reputations (reporter and developer) improved the performance in Decision Tree.

6.4 RQ4: Can reporter and developer ratings, when combined with text features, increase the performance of predictions when compared to reporter and developer reputations?

To answer this question, we first ran the models using only TF-IDF. We then ran the models using the ratings and, lastly, the reputations.

Table 7: Average F-measure of Features in NetBeans

	Reputations	Ratings	Reputations and Ratings
Naive Bayes Multinomial	65%	65%	65%
Naive Bayes Gaussian	61%	**63%**	**63%**
Gradient Boosting	59%	**62%**	**62%**
SVM	**62%**	61%	**62%**
Random Forest	**62%**	60%	61%
Logistic Regression	56%	58%	**59%**
Decision Tree	**61%**	57%	57%
KNN	**56%**	55%	55%
SGD Classifier	48%	46%	**54%**
MultiLayer Perceptron	40%	41%	41%

Table 8 shows the F-measure of the contextual features in Eclipse Platform.

In Eclipse Platform, the combination between TF-IDF and ratings features increased the performance in Gradient Boosting, Random Forest and Logistic Regression. In these models, the reputations also increased the performance of TF-IDF, however, ratings were superior to reputations in these models. KNN and Decision Tree had the same performance with reputations and ratings. Reputation was only superior to the others in the MultiLayer Perceptron model. For all other models, TF-IDF had the best performance.

Table 8: F-measure of the contextual features in Eclipse Platform

	TF-IDF	TF-IDF + Reputation	TF-IDF + Ratings
Gradient Boosting	60%	65%	**68%**
Random Forest	61%	64%	**68%**
Logistic Regression	59%	48%	**65%**
KNN	55%	**60%**	**60%**
Decision Tree	53%	**59%**	**59%**
Naive Bayes Gaussian	**57%**	36%	52%
SVM	**60%**	47%	47%
Naive Bayes Multinomial	**60%**	43%	43%
MultiLayer Perceptron	58%	**60%**	41%
SGD Classifier	**62%**	57%	39%

The results of textual features in the NetBeans project can be seen in Table 9, which shows the average F-measure when using textual features. In our results, ratings increased the performance in the NetBeans project using Random Forest, Gradient Boosting, Logistic Regression and Decision Tree. Reputation was better in

the SVM model. Naive Bayes Multinomial, Naive Bayes Gaussian and KNN had the same performance for both reputation and rating but increased the performance of TF-IDF. On the other hand, with Multi-layer Perceptron and SGD Classifier, the best performance was TF-IDF without other features.

Table 9: Performance in F-measure using Text Features in the NetBeans Project

	TF-IDF	TF-IDF + reputation	TF-IDF + ratings
Naive Bayes Multinomial	60%	**65%**	**65%**
Random Forest	61%	62%	**65%**
Gradient Boosting	59%	59%	**65%**
Naive Bayes Gaussian	54%	**62%**	**62%**
LogisticRegression	59%	58%	**60%**
SVM	59%	**62%**	60%
Decision Tree	57%	55%	**59%**
KNN	54%	**56%**	**56%**
Multi-layer Perceptron	**58%**	51%	44%
SGD Classifier	**59%**	54%	37%

7 DISCUSSION

In this study, we present features called "reporter rating" and "developer rating." They are based on the time taken to fix a bug. In contrast to reputations, which are based on the frequency that the reporter or developer submits or fixes a given number of bugs, ratings are based on the time taken to fix them.

Our intuition was that a feature based on time would be more relevant to machine learning models than the number of bug submissions. However, reputations showed an average F-measure greater than ratings in Eclipse Platform. In Eclipse Platform, the Random Forest algorithm shows the best F-measure values. Reporter and developer reputation performed better than ratings in this case. However, we can not claim that the difference between these features are significant. In Eclipse Platform, Random Forest had a F-measure of 67% for reporter reputation, while the reporter ratings arrived at 65%. The same occurred in the Netbeans project, where reporter reputation achieved 63% and reporter rating 60%. For developers, reputations and ratings followed a similar pattern. Developer reputation achieved an average F-measure of 68% and developer rating 66%.

In our experiment, it is possible to see that, when combined, reputations and ratings can improve the performance of the models. For example, Gradient Boosting, SGD Classifier and Multilayer Perceptron were slightly improved using the two features at the same time in Eclipse Platform. We can observe the same behavior in Netbeans with SGD Classifier. It is important to say that neither reputation nor rating are independent variables. For this experiment we used the initial feature as listed in sec 5. The combination of these features, as well the choice in datasets, could have influenced the results of our experiment [19]. According to Bhattacharya and Neamtiu [19], even though we can not generalize prediction models, they concluded that open source projects are not influenced by bug-opener reputation; whereas, commercial projects may be influenced as shown by Guo et al. [1] and confirmed by Zhang et al. [3]. We deduce that commercial projects may be more influenced by the person who opened a bug due to the organizational hierarchy. For example, if a person with a high position in the company opens a bug, it may be fixed more quickly merely because of the existing hierarchy, while a bug opened by a person in a lower position than the developer that received it might be fixed more slowly. In contrast to companies, influence in open source projects is built over time.

Naive Bayes Gaussian and Logistic Regression show the best results with rating features in both datasets tested. The rating features were superior to reputation in these models. Rating features proved to be better when used together with textual features. Random Forest and Decision Tree showed the best results when used with reputations in both datasets. The exception to this observation is when we used textual features in conjunction with reputations. In this case, these algorithms showed better results with ratings.

In the NetBeans project, the Naive Bayes Multinomial model had the best performance. The performance achieved in the Naive Bayes Multinomial was the same for initial features as it was with textual features combined with reputations and ratings. The worst result for the Naive Bayes Multinomial in the NetBeans project was when we ran only text. In the literature, non-textual features have shown to be a good option for predicting bug-fixing time. Habayeb et al. [4] argues that textual features are computationally expensive.

Random Forest and Gradient Boosting performed well in our experiment. Both techniques are ensemble machine learning. The Dehghan et al. study [20] has shown that ensemble learning may improve the performance of bug-fixing time predictions. In our study, we can see that ensemble learning was in fact shown to be useful for predicting bug-fixing time.

Other studies have shown that incorporating post-submission features [2, 4], which are updated over time [13], may improve accuracy. We understand that ratings could be combined with features such as the number of developers available at the time the bug was opened, the total number of bugs opened until that point, the number of attachments and comments [6], as well as the role of the person who reported the bug within the project. As well as other textual features, such as Word Embeddings, Topic Modeling, or Word2Vec. In summary, by integrating project-specific features with our proposed features, the predictive results can be further improved.

8 THREATS TO VALIDITY

Internal validity: Despite the increasing use of deep-learning approaches in predicting bug-fixing time, which has been promising, our experiment focused only on supervised ML approaches. The hyperparameters of algorithms were not tuned, instead the default hyperparameters of libraries were employed.

External validity: Only two datasets were used. These datasets are generally used in the prediction of bug-fixing time research. We can not generalize our results. Eclipse and Netbeans are two popular open-source projects which have their own peculiarities arising from their community and processes. In literature, the person who

open the bug has better performance in proprietary software. In this work, we only analyzed open source projects.

Construct validity: in our experiment, we calculated the time to fix a bug in days. This assumption may not have captured the real time to fix a bug. Also, we remove the bug reports that have taken less than one day to be fixed and the outliers are a great 75% quartile. We based the equation to calculate the bug-fixing time and to the decision regarding removing the outliers based on the current literature. The remotion of outliers shows to be usefull to predict bug-fixing time. We take these decision based on the literature support [9, 16, 21]. Another combination of features can influence models. Features such as severity and priority can be changed during the process. We decided to use these features because our work is aimed at classifying the bug early in order to help the software manager make the decision about the bug as soon as possible.

9 CONCLUSION

In this study, we proposed features based on the time taken to fix bugs and related them to the reporter and developer. We compare these features with reputations which are based on reporter and developer participation. In our experiment, we can see that the reputation is a relevant feature which might improve the performance of the models, more specifically, when decision tree models and initial features are used in conjunction. The ratings features might improve the performance of probabilistic models like Naive Bayes, Gradient Boosting and Logistic Regression. Even though ratings show relevant results only with initial features, we can see that the best performance was found when combined with textual features.

This study proposes developer reputation and ratings and contributes to improving the understanding of social impact on prediction of bug-fixing time using machine learning models. For future research we propose using these features utilizing new combinations, as well as deep learning models. We also intend to develop a model based on rating features.

REFERENCES

[1] Philip J. Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 495–504, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806871. URL http://doi.acm.org/10.1145/1806799.1806871.

[2] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 34–43, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. doi: 10.1145/1321631.1321639. URL http://doi.acm.org/10.1145/1321631.1321639.

[3] Hongyu Zhang, Liang Gong, and Steve Versteeg. Predicting bug-fixing time: An empirical study of commercial software projects. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1042–1051, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL http://dl.acm.org/citation.cfm?id=2486788.2486931.

[4] M. Habayeb, S. S. Murtaza, A. Miranskyy, and A. B. Bener. On the use of hidden markov model to predict the time to fix bugs. *IEEE Transactions on Software Engineering*, pages 1–1, 2017. ISSN 0098-5589. doi: 10.1109/TSE.2017.2757480.

[5] Pranav Ramarao, K. Muthukumaran, Siddharth Dash, and N. L. Bhanu Murthy. Impact of bug reporter's reputation on bug-fix times. In *2016 International Conference on Information Systems Engineering (ICISE)*, pages 57–61. IEEE, April 2016. ISBN 978-1-5090-2287-8 978-1-5090-2288-5. doi: 10.1109/ICISE.2016.18. URL http://ieeexplore.ieee.org/document/7486274/.

[6] Sien Reeve Ordonez Peralta, Hironori Washizaki, Yoshiaki Fukazawa, Yuki Noyori, Shuhei Nojiri, and Hideyuki Kanuka. Analysis of Bug Report Qualities with Fixing Time using a Bayesian Network. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, EASE '23, pages 235–240, New York, NY, USA, June 2023. Association for Computing Machinery. ISBN 9798400700446. doi: 10.1145/3593434.3593484. URL https://dl.acm.org/doi/10.1145/3593434.3593484.

[7] Bruno Rafael de Oliveira Rodrigues and Fernando Silva Parreiras. Predicting Bug-Fixing Time with Machine Learning - A Collaborative Filtering Approach. *Anais do Computer on the Beach*, 13:021–028, July 2022. ISSN 2358-0852. doi: 10.14210/cotb.v13.p021-028. URL https://periodicos.univali.br/index.php/acotb/article/view/18705.

[8] Pasquale Ardimento. Predicting Bug-Fixing Time: DistilBERT Versus Google BERT. In Davide Taibi, Marco Kuhrmann, Tommi Mikkonen, Jil Klünder, and Pekka Abrahamsson, editors, *Product-Focused Software Process Improvement*, Lecture Notes in Computer Science, pages 610–620, Cham, 2022. Springer International Publishing. ISBN 978-3-031-21388-5. doi: 10.1007/978-3-031-21388-5_46.

[9] Dietmar Pfahl, Siim Karus, and Myroslava Stavnycha. Improving expert prediction of issue resolution time. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, EASE '16, pages 42:1–42:6, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3691-8. doi: 10.1145/2915970.2916004. URL http://doi.acm.org/10.1145/2915970.2916004.

[10] Emanuel Giger, Martin Pinzger, and Harald Gall. Predicting the fix time of bugs. In *Proceedings of the 2Nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 52–56, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-974-9. doi: 10.1145/1808920.1808933. URL http://doi.acm.org/10.1145/1808920.1808933.

[11] Tao Zhang, He Jiang, Xiapu Luo, and Alvin T. S. Chan. A Literature Review of Research in Bug Resolution: Tasks, Challenges and Future Directions. *The Computer Journal*, 59(5):741–773, May 2016. ISSN 0010-4620. doi: 10.1093/comjnl/bxv114. URL https://academic.oup.com/comjnl/article/59/5/741/2568647.

[12] Stephen Marsland. *Machine Learning: An Algorithmic Perspective, Second Edition*. CRC Press, September 2015. ISBN 978-1-4987-5978-6.

[13] Muhammed Kadir Yucel and Ayse Tosun. Measuring bug reporter's reputation and its effect on bug resolution time prediction. In *2022 7th International Conference on Computer Science and Engineering (UBMK)*, pages 110–115, 2022. doi: 10.1109/UBMK55850.2022.9919454.

[14] Adrian Schröter. Msr challenge 2011: Eclipse, netbeans, firefox, and chrome. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 227–229, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0574-7. doi: 10.1145/1985441.1985478. URL http://doi.acm.org/10.1145/1985441.1985478.

[15] Nguyen Duc Anh, Daniela S. Cruzes, Reidar Conradi, and Claudia Ayala. Empirical Validation of Human Factors in Predicting Issue Lead Time in Open Source Projects. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, Promise '11, pages 13:1–13:10, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0709-3. doi: 10.1145/2020390.2020403. URL http://doi.acm.org/10.1145/2020390.2020403.

[16] A. Lamkanfi and S. Demeyer. Filtering bug reports for fix-time analysis. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 379–384, March 2012. doi: 10.1109/CSMR.2012.47.

[17] Shirin Akbarinasaji, Bora Caglayan, and Ayse Bener. Predicting bug-fixing time: A replication study using an open source software project. *Journal of Systems and Software*, 136:173 – 186, 2018. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2017.02.021. URL http://www.sciencedirect.com/science/article/pii/S0164121217300365.

[18] Charu C. Aggarwal. *Recommender Systems*. Springer International Publishing, 2016. doi: 10.1007/978-3-319-29659-3. URL https://doi.org/10.1007/978-3-319-29659-3.

[19] Pamela Bhattacharya and Iulian Neamtiu. Bug-fix time prediction models: Can we do better? In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 207–210, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0574-7. doi: 10.1145/1985441.1985472. URL http://doi.acm.org/10.1145/1985441.1985472.

[20] Ali Dehghan, Kelly Blincoe, and Daniela Damian. A hybrid model for task completion effort estimation. In *Proceedings of the 2Nd International Workshop on Software Analytics*, SWAN 2016, pages 22–28, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4395-4. doi: 10.1145/2989238.2989242. URL http://doi.acm.org/10.1145/2989238.2989242.

[21] W. AbdelMoez, Mohamed Kholief, and Fayrouz M. Elsalmy. Improving bug fix-time prediction model by filtering out outliers. In *2013 The International Conference on Technological Advances in Electrical, Electronics and Computer Engineering (TAEECE)*, pages 359–364. IEEE, May 2013. ISBN 978-1-4673-5613-8 978-1-4673-5612-1 978-1-4673-5611-4. doi: 10.1109/TAEECE.2013.6557301. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6557301.