Kanwal J, Maqbool O. Bug prioritization to facilitate bug report triage. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 27(2): 397–412 Mar. 2012. DOI 10.1007/s11390-012-1230-3

Bug Prioritization to Facilitate Bug Report Triage

Jaweria Kanwal and Onaiza Maqbool

Department of Computer Science, Quaid-i-Azam University, Islamabad, Pakistan

E-mail: kjaweria09@yahoo.com; onaiza@qau.edu.pk

Received April 27, 2011; revised January 12, 2012.

Abstract The large number of new bug reports received in bug repositories of software systems makes their management a challenging task. Handling these reports manually is time consuming, and often results in delaying the resolution of important bugs. To address this issue, a recommender may be developed which automatically prioritizes the new bug reports. In this paper, we propose and evaluate a classification based approach to build such a recommender. We use the Naïve Bayes and Support Vector Machine (SVM) classifiers, and present a comparison to evaluate which classifier performs better in terms of accuracy. Since a bug report contains both categorical and text features, another evaluation we perform is to determine the combination of features that better determines the priority of a bug. To evaluate the bug priority recommender, we use precision and recall measures and also propose two new measures, Nearest False Negatives (NFN) and Nearest False Positives (NFP), which provide insight into the results produced by precision and recall. Our findings are that the results of SVM are better than the Naïve Bayes algorithm for text features, whereas for categorical features, Naïve Bayes performance is better than SVM. The highest accuracy is achieved with SVM when categorical and text features are combined for training.

Keywords bug triaging, bug priority, classification, mining bug repositories, evaluation measures

1 Introduction

During software development and maintenance, errors (bugs) in a software system are introduced due to many reasons, which include misunderstanding the requirements, poor design or implementation strategy, lack of error handling methodology in coding, and continuous changes to the code after development. In software projects, usually a database is maintained to collect and manage the large number of bug reports from users and developers. This database is known as a bug repository or a bug tracking system. Bug repositories are used for open as well as for closed software systems, but they are mostly used for open source systems where the developers, users and other team members are distributed all over the world.

In a bug repository, bugs are reported from different sources, e.g., from users, developers or any other member of the technical support team[1]. These bugs need to be analyzed carefully to determine, for example, whether the bugs are duplicate (reported earlier) or unique, valid or invalid (caused by the operating system, not by the product), important or unimportant, and who will resolve them. This process is called bug triaging. The person who analyzes the bug reports is called a triager. A triager's task is to manage the bug repository so that it contains only real bugs and important bugs are addressed quickly[2].

Managing the new bug reports received in a day in large software projects is a challenging task for the triager because usually there are a large number of such reports, and it is difficult to triage them within the available time and resources[3]. In software projects, the number of bug reports received is on an average 20∼30 per day[2]. If each report takes 5 minutes for triaging, then more than 2 person hours are spent on managing the reports. In large projects such as Mozilla[4], on an average 300 bug reports are received in a day[2]. Triagers can be overwhelmed by the number of reports that need to be triaged. Moreover, bug triaging is a time consuming task because to triage a bug report, the triager needs a lot of information about that bug, e.g., to which component it belongs, what type of a problem it is, how severe it is and how important it is for the software project to be solved earlier[5-6]. If the triager starts reading all the reports one by one without prioritization of the bug reports, it is possible that some important bugs are left untreated for a long time, negatively affecting the project.

To solve this problem, bug reports contain a field where the person who reports the bug may assign its priority (how important the bug is), but sometimes this

Regular Paper

field is left blank. Moreover, the reporter may not correctly assign the priority level[7] as his opinion about the importance of a bug may be different from that of the triager, who has more information about the software as a whole. To assign a proper level of priority to the bug reports, the triager analyzes their contents and also uses his own project knowledge which he gained in triaging the bug reports of that project. This process consumes time. Moreover it is possible that at this early stage, the triager cannot assign proper priority to the bugs especially in open source software projects where the triager may be a volunteer or not much experienced.

In recent years, there has been an increasing interest in mining of software repositories such as source code repositories, bug repositories and email archives. Mining techniques find hidden patterns from the data stored in these repositories, thus revealing useful information. Different recommenders have been built by researchers to assist in bug triaging, e.g., for assigning a bug report to an appropriate developer for resolution[1,8], for predicting the component for which a bug is reported[2], for predicting the time that a bug will take to resolve[9-10]. However, there has been little work on recommending priority and severity for a new bug report[11-13].

Given the large number of new bug reports that a triager needs to handle, it would be useful to explore mining techniques to facilitate in triaging, i.e., assigning priorities to the newly arrived bug reports automatically. The recommended priority can further be analyzed by the triager to confirm or refine the automatically assigned priority.

Machine learning classification techniques may be used to develop such a recommender by mining the bug data present in a bug repository. Classification techniques build models (using training data) by finding patterns in the data to categorize it into different classes. These models are then used to predict the class of new unseen data (test data) according to predefined classes. Some well-known classification techniques are Naïve Bayes, Support Vector Machine (SVM), Decision Trees and Neural Networks.

In this paper, we propose a classification based approach for building a bug priority recommender, and conduct experiments using SVM and Naïve Bayes classifiers. Moreover, we propose measures for the evaluation of classification results. The main contributions of this paper are:

1) Proposing and evaluating a classification based approach for automatic bug priority prediction. Support Vector Machine (SVM) and Naïve Bayes classification algorithms are used for this purpose, and their results are analyzed to determine which algorithm performs better for bug priority assignment.

2) Exploring different features within bug reports to determine which features contribute more towards bug priority classification.

3) Defining new measures, Nearest False Negatives (NFN) and Nearest False Positives (NFP), for the evaluation of bug priority recommender. These measures support existing precision and recall measures, and may be used to enhance understanding of the results produced by precision and recall.

The rest of the paper is organized as follows. Section 2 presents the research work related to bug prioritization and automatic bug triage. Section 3 gives an overview of the bug triage process and bug report features. Section 4 presents our proposed classification based approach and the classification techniques that we used for building the classifier. Section 5 describes the experimental setup. Section 6 details the results of our experiments. In Section 7, we conclude the paper with a discussion of results.

2 Related Work

Bug finding tools are used to find bugs from the source code and prioritize them, but this prioritization usually has a high false positive rate. Thus researchers have proposed ways to improve bug prioritization. Kim and Ernst[14] analyzed the bug life time and priority levels assigned by bug finding tools and reprioritized the bug categories according to bug life time. Life time of a bug was computed using the software change history data and bugs with a shorter life time were assigned a higher priority level. In subsequent work, Kim and Ernst[15] reprioritized the bug categories on the basis of weight of a bug category. The weight of a bug category was increased if bugs in that category were resolved (called a fix-change). On the other hand, if bugs in a category were not resolved (a non-fix change), the weight was decreased. Kremenek and Engler[16] also reprioritized the bugs found by bug finding tools on the basis of the frequency count of successful and failed checks. Checks are classified into successful and failed on the basis of a tool's analysis decisions.

To automatically assign a developer to new bug reports, Anvik *et al.*[17] applied machine learning algorithms on the bug report data. Support Vector Machines, Naïve Bayes algorithm and Decision Trees were used on the bug data of Eclipse, Firefox and GCC projects. SVM achieved high precision. In subsequent work, Anvik and Murphy[5] evaluated their approach by extracting the developer expertise from bug repositories and source code repositories. In [18], Anvik and

Murphy proposed a machine learning (ML) based approach for creating recommenders for a variety of development oriented decisions, e.g., for assigning a developer to a new bug report, predicting the component of a new bug and finding people interested in a bug. Recommenders were evaluated using the bug data of five software projects and achieved more than 70% precision. An automatic approach was also proposed for configuring the recommenders and was found to be useful in reducing the effort of configuration.

Canfora and Cerulo[1] proposed an approach to assign a developer to a new bug report, using historic information stored in the bug and source code repositories. To classify bugs according to developer, Aljarah *et al.*[19] evaluated different term selection techniques to find discriminating terms from the summary field of a bug. Three different techniques based on Log-Odds-Ratio were compared with the Gain Ratio and Latent Semantic Analysis (LSA) techniques. Results showed that the selection of discriminating terms improved the assignment of an appropriate developer to a bug. For automatic bug report triage, Ahsan *et al.*[20] evaluated different feature reduction and classification techniques. Bug report data from the Bugzilla bug tracking system was changed into five different datasets using various indexing and dimension reduction methods. Experimental results showed that accuracy of the classifier based on Latent Semantic Indexing (LSI) technique and SVM was better than that of other techniques. Jeong *et al.*[21] introduced a graph model based on the bug tossing history available in bug repositories (tossing refers to a bug being reassigned to different developers in its lifetime). This model is helpful in extracting team structure and also in assigning an appropriate developer to a bug. To reduce the tossing steps, a reduction method is used on graphs to find a path with fewer steps. The path reduction algorithm is based on the weighted breadth-first search algorithm, which is similar to the breadth-first search algorithm. Experiments conducted on the bug data of Eclipse and Mozilla show that the approach reduces tossing steps by up to 72% and results in better developer assignment. A fuzzy set-based approach for automatic assignment of developers was proposed by Tamrawi *et al.*[22]. For each technical term in a new bug report, fuzzy sets were computed to identify the developers who were more suitable for fixing the bugs relevant to that term. The fuzzy set technique achieved higher prediction accuracy than ML approaches for automatic bug triage.

Lamkanfi *et al.*[11] built a classifier model to classify bugs into severe and non-severe bugs. The summary information of a bug report is used to train the Naïve Bayes classifier. Results of the classifier using precision and recall measures reveal that severity assignment is improved as compared to random assignment. They enhanced their work by applying different classification techniques, i.e., SVM, Naïve Bayes, Multinomial Naïve Bayes and Nearest Neighbour to evaluate which technique performs better in classifying the bugs according to severity[23]. Eclipse and Gnome bug data was used for experiments. Results were evaluated using ROC (receiver operating characteristic) curve[24]. Performance of Multinomial Naïve Bayes was found to be better than that of other classification algorithms.

Gegick *et al.*[25] proposed a text mining approach to identify security bug reports (SBR) from the set of mislabeled non-security bug reports (NSBR). A bug report's summary and long description fields were used for training the model. The bug data of Cisco software project was used for experiments. The model was evaluated using the precision, recall and success rate measures. Results revealed that the approach was able to detect a large percentage (78%) of SBRs that had been manually labeled as NSBRs. Zaman *et al.*[26] analyzed the characteristics of different types of bugs such as security and performance bugs to find how they behave differently from each other and from the other bugs in terms of the bug fix time, the number of developers assigned and the number of files impacted. Results revealed that security bugs were more complex, required more developers with experience, and affected a larger number of files but took less triage and fix time than performance and other bugs. Similarly, performance bugs were more complex and needed more experienced developers than the other bugs.

Yu *et al.*[12] predicted the priority of defects found during the software testing process. The Artificial Neural Network (ANN) and Naïve Bayes classifiers were used to train the prediction model. Features used were not extracted from bug reports in the bug repository, rather from the testing process. Precision, recall and F-measure were used to evaluate the classifier performance. Experimental results indicate that the ANN's performance is better than the Naïve Bayes for defect priority prediction. Kanwal and Maqbool[13] built a classifier model using SVMs to prioritize the newly arrived bug reports. Bug report features were categorized into different feature categories to evaluate which feature category better determines the priority of a bug. Accuracy of the classifier improved when training features were combined.

Runeson *et al.*[27] detected duplicate bug reports by measuring the similarity of bug reports (submitted in a timeframe of 60 days) using the vector space model along with cosine similarity. Results show that 2/3 of the duplicates can be found using this technique.

Expanding the timeframe up to 100 days decreases the recall rate. Wang *et al.*[28] also proposed an approach to detect duplicate bug reports using natural language information present in the text field of a bug report and execution traces of that bug. Recall increased by 30% by using execution information. Clustering technique was used in [29] for duplicate detection. In more than 80% clusters created by automatic clustering, majority of failures were due to same causes. For duplicate bug detection, Prifti *et al.*[30] proposed an approach that improved the performance of Information Retrieval techniques by limiting the search space to recently reported bug reports on the assumption that the time interval between duplicate bug reports is short. The approach also reduced the number of bug reports because instead of reporting a duplicate bug, a user can add the information of his/her bug in the existing bug report.

Wu *et al.*[31] developed a BugMiner tool which utilizes the historic data of bug repositories for predicting the missing categorical fields of new bug reports on the basis of the available fields. Moreover this tool finds the duplicate bugs and displays all the similar bug reports of a new bug report. Trend analysis of the bugs occurring in a project since its first launch is also carried out. Marks *et al.*[32] analyzed different features of a bug report to find the characteristics of bug fix-time using the bug data of Mozilla and Eclipse bug repositories. The most influential factors of a bug report according to fix-time were bug location and time of reporting the bug. Using a random forest classification technique, 65% bugs were correctly classified.

Different machine learning approaches have also been applied on bug repository data for automating bug triage[2-3,8], effort estimation for resolving a bug[9], impact analysis[29] and predicting the number of bugs for next versions[33-34].

3 Bug Repositories

Bug repositories are used to manage bug reports so that they are assigned to an appropriate developer or maintenance team[8]. A bug repository is helpful for software systems in many ways, as it provides a communication forum where developers can discuss bug resolution, design implementations and enhancement features, experts can help in design deliberations of a complex problem and users can be made aware of the status of a bug[2].

3.1 Bug Triage Process

In a bug repository, bug triage is a process in which a triager makes decisions about the bugs entered in the bug repository by examining them in different ways. These decisions can be divided into two categories, repository oriented and development oriented. In repository oriented decisions, the triager makes sure that the bug has not already been reported before and the bug has enough information for the developers and makes sense. The purpose of repository oriented decisions is to remove those bug reports from the repository that do not need to be resolved.

As a result of repository oriented decisions, the triager selects a set of unique (not duplicate) and valid bug reports which are further analyzed for development oriented decisions. Development oriented decisions involve checking that a bug is filed for the correct component, product and version. Moreover, an important purpose of development oriented decisions is to allocate time and resources to resolve bugs[6], for which triagers examine severity and priority levels of the bugs. These levels may be changed if they are found to be inappropriate. Assigning the correct priority level is important to resolve more important bugs first. After this, the triager writes comments for the bug and assigns this bug report to an appropriate developer to resolve the bug.

3.2 Bug Report Features

Bug repositories collect bug reports from users and developers. The structure of a bug report is more or less the same across bug repositories. We describe in detail the Bugzilla[35] bug reports. Bugzilla is the most widely used bug repository for open source projects[2].

A bug report in Bugzilla contains a number of fields which represent attributes or features of the report. Some fields are categorical such as bug-id, date of submission, component, product, resolution, status, severity (how serious a bug is), priority (how important a bug is, represented normally as levels P1∼P5 with P1 being most important), platform, operating system, reporter, assignee and cc-list. Some of the categorical fields are fixed at the time of report submission, e.g., bug id, report submission time and reporter name. Some fields such as product, component, severity, priority, version, platform and operating system are entered by the reporter but may be changed by the triager or developer if needed[3]. Other fields such as developer who resolves the bug, list of people who are interested in bug resolution, bug-status, and resolution, change throughout bug life time.

Text fields consist of summary and long description. Summary is the title of a bug report or short description of a bug in one line written by the reporter. In long description, the problem is described in detail by the reporter. The user writes the problem he/she faced while using the product. The triager or developer also writes about the problem in this field in detail after analyzing

it. Text field contains all the discussion about the bug and all the phases through which a bug has passed, such as cause of the problem and possible solutions to fix it. Implementation detail of the problem and possible solutions for complex problems are also discussed in this field[3], which serves as the forum for discussion.

Bugs move through different stages from submission time to closing time. The bug-status and resolution fields track the life cycle of a bug. When a bug report is submitted by a user its status is set to "new". When a bug is assigned to a developer to resolve, then the bug status is set to "assigned". If a developer resolves the bug, then status of the bug is set to "resolved". After resolution, the bug status is set to "closed" or checked by the quality assurance team to verify that an appropriate resolution of the bug has been performed. If a bug is verified then its status is changed to "verified", otherwise it is reconsidered for resolution and the status is changed to "reopen".

4 Proposed Priority Recommender

In this section, we describe our approach for developing a priority recommender.

4.1 Classification Based Approach

Data mining techniques are used to find patterns from large amount of data to transform it into useful information. There are different data mining techniques for recognizing patterns from the data, e.g., classification, association rule mining and clustering. Data mining classification techniques classify the data according to some predefined categorical labels.

Classification is the process of building a model by learning from a dataset. Classification is a two-step process, learning and classification. In the first step a classifier model is built by determining the characteristics of each class from the given training dataset which consists of training instances with associated class labels[36]. For example, suppose $X(x_1, x_2, x_3, \ldots, x_n)$ is a training instance where $x_1, x_2, \ldots, x_n$ represent the features (attributes) and n is the number of features. Each training feature provides a piece of information to the classifier that helps in determining the characteristics of the class. For each X_i there is a special attribute (class label) which represents its class y_i. For example, in Table 1, component, platform and product are the features used for building the classifier model to determine the class label, i.e., priority. This step can be viewed as learning of a function, $y = f(X)$, where y is the predicted class label and $f(X)$ may be some rules or mathematical formulae.

In the second step, the function is used to predict the class label y for new instances. In case of a rules-based classifier, rules are used to characterize the new instance in an appropriate class. In case of mathematical formula, feature values are plugged into the equation to find its class label.

Table 1. Sample Bug Report: Features and Class Label

	Features				Class
Bug-ID	Component	Platform	Product	···	Priority
1	SWT	Macintosh	JDT	···	P4
2	Team	PC	Platform	···	P3
3	SWT	Macintosh	PDE	···	P1

Accuracy of the classifier is the percentage of test tuples correctly classified by the learned classifier model. The dataset that is used for testing (validating) the classifier is unseen data (not used for training). In the validation process, we know the actual class labels of the test data but the classifier is not aware of them.

Although a number of classification algorithms are available e.g., Naïve Bayes, Support Vector Machines, Decision Trees, Neural Networks and K-Nearest Neighbors, in the following subsections we describe Naïve Bayes and Support Vector Machine (SVM). These two classifiers have been used by various researchers for text classification and have shown promising results[2,37].

4.2 Naïve Bayes Classifier

The Naïve Bayes algorithm classifies a new instance by calculating its probability for a particular class using the Bayes rule of conditional probability[38]. The probability of a new instance is calculated for each class and the class with the greatest probability is assigned to the new instance.

In Bayesian classification, the posterior probability of a hypothesis i.e., $P(H|X)$, is calculated from prior probabilities $P(H), P(X)$ and posterior probability of tuple X conditioned on H, $P(X|H)$. H represents the hypothesis that a data tuple X belongs to a specified class C, given the attribute description of X. The Bayes theorem can be stated as:

$$P(H|X) = (P(H) \times P(X|H))/P(X).$$

Replacing H by C (hypothesis that a tuple X belongs to a class C):

$$P(C_i|X) = (P(X|C_i) \times P(C_i))/P(X),$$

for $i = 1$ to m, where m represents the number of classes. As $P(X)$ is same for all classes so we only need to maximize $P(X|C_i) \times P(C_i)$.

To calculate $P(X|C_i)$, the Naïve Bayes classifier makes the simplifying assumption that the value of an attribute is independent of the value of other attributes

given the class. Thus $P(X|C_i)$ is calculated as:

$$P(X|C_i) = P(x_1|C_i) \times P(x_2|C_i) \times \cdots \times P(x_n|C_i). \quad (1)$$

The classifier assigns the class C_i to a new instance X such that

$$P(C_i|X) > P(C_j|X) \quad \text{for} \quad 1 \leqslant m \leqslant j \neq i.$$

From (1), it is clear that if the probability of some attribute value given a class (e.g., $P(x_1|C_1)$) is zero, it makes the probability of that class (i.e., $P(C_1|X)$) zero. This problem is corrected by using the Laplacian Correction[39].

4.3 Support Vector Machines

Support Vector Machines build non-linear classification models from the training data for each class. These models are then used for predicting the class of new instances. SVMs transform the original data into higher dimensionality and find a separating hyperplane in the new mapped data[40]. This type of classification is independent of the dimensionality of the vector space. Training data can be separated by various lines but SVM finds a line with the maximum margin. For some classes of well behaved data, the choice of maximum margin will lead to maximal generalization when predicting the class of unseen new data[41].

A separating hyperplane for a two-class classifier can be written as

$$\boldsymbol{w} \cdot x + b = 0, \quad (2)$$

where x represents the training instances that lie on the hyperplane, $\boldsymbol{w}$ is a weight vector, b is a scalar, often referred to as a bias and "$\cdot$" represents the dot product. These are determined by the SVM from the training dataset.

In classification of text documents, the input feature space is very large as each word is considered as a feature. When dealing with a large number of features, most of the algorithms apply dimensionality reduction methods to remove terms that are irrelevant but in text classification there are very few terms that are irrelevant[37]. Thus text classification is challenging due to its high dimensional feature space. Since SVMs are independent of data dimensionality and robust to overfitting[42], they are well suited for text classification.

5 Experimental Setup

In this section, we describe the test system used for our experiments, the selection of features and the criteria used for evaluation.

5.1 Dataset and Pre-Processing

The Eclipse project[43] bug reports were used for our experiments. Eclipse projects are focused on building an open development platform with extensible frameworks, tools and runtimes for building and managing software throughout its life time. Eclipse data has been used by different researchers for their experiments[3,8,27-28]. We used bug data of Eclipse from 2001 to 2006, which has many products and components. It is stored in two main versions: version 2 and version 3 and subversions (e.g., 2.1, 2.1.1, 2.2, 3.1, 3.2).

From the 49 Eclipse products, we selected the platform product for experiments as it contains larger number of bug reports according to priority levels as compared to other products. The number of bug reports of platform product is given in Table 2.

Table 2. Statistics of Each Bug Priority Class in Version 2 of Platform Product

Bug Priority Class	No. Instances
P1	705
P2	1 073
P3	9 441
P4	536
P5	327

Eclipse bug reports were originally in the form of XML files. An application was developed in C# to extract the desired features. The extracted bug report features were product, component, platform, operating system, version, bug resolution, bug status, bug priority, bug severity, summary and long description.

5.2 Training and Testing Dataset

Eclipse bug report data is presented in two main versions, version 2 and version 3. Version 2 data is used for training the classifiers for bug priority. In our dataset, the number of bug reports for priority level P3 is very high as compared to the other priority levels. So we selected an equal number of instances (bug reports) for each bug priority class to train the classifier. The least number of instances available for a priority class is 327 (number of instances of P5). Therefore, we selected 327 instances from each of the classes P1 $\sim$ P5, making our training dataset size equal to $327 \times 5 = 1\,635$. Version 3 bug data of the platform product is used to evaluate the classifier model①. The number of bug reports in version 3 of the platform product is given in Table 3. For version 3 bug reports, we have the actual class labels (priority levels assigned at bug resolution time), but to evaluate our recommender we hid the class labels from the classifier and compared the actual priority levels with the predicted ones.

①The Eclipse data used in our experiments is available at http://cs.qau.edu.pk/profiles/onaiza.htm

Table 3. Statistics of Each Bug Priority Class in Version 3 of Platform Product

Bug Priority Class	No. Instances
P1	147
P2	423
P3	8 650
P4	246
P5	42

Only resolved bug reports (having status value "resolved", "closed" or "confirmed") were selected for training and testing because priority of a bug report is changed throughout the bug life time. It is finally set by the triager or developer who fixes the bug and records the bug priority in the bug report.

5.3 Text Processing

Text attributes extracted from the bug reports were bug summary and long description. Text attributes contain a number of words that are not meaningful. So we applied a standard text categorization approach to transform the text data into a meaningful representation. First, the whole text was converted into words by removing punctuation, brackets and special symbols (e.g., @, $, %). From the list of words, stop words (e.g., is, am, I, he), common words (e.g., actually, because, everywhere) and non-alphabetic words (e.g., 12345, −343, 2010) were removed because these are unimportant and provide little information about the problem described in the bug report. Stemming is applied on the text to convert a word into its ground meaning. For example, "experimental" and "experiments" were converted to "experiment". Verbs were also converted into their original form, e.g., "was" and "being" were converted to "be".

After finding the ground form of each word, the number of occurrences of each word in the bug report was calculated. Due to stemming the text, a word with different grammatical structure is considered as one word. So a word is represented in a vector form having two dimensions: a word and its frequency. All the categorical and text features were converted into numeric representation because SVM takes only numeric features within training and testing files.

5.4 Feature Selection

Experiments were performed by taking different combinations of the bug report attributes. We divided the bug report attributes into 5 categories as training features for classifiers. These categories are: Categorical (CF), Summary (SF), Categorical and Summary (CSF), Summary and Long description (TF), and Categorical, Summary and Long description (CSTF). Classifiers are trained with different feature combinations for bug priority classification, to check which features contribute more towards bug priority classification.

5.4.1 Categorical Features

Categorical attributes of a bug report that we used as training features for bug priority classification are component, severity, platform, operating system, bug lifetime and developer. We used the bug creation date and last modification date to calculate the life time of a bug because it is not recorded in an Eclipse bug report. As we selected only the bug reports that are resolved, the latest modification date means it is the date when a bug is resolved.

These attributes were selected because they have an impact on the priority of a new bug report. Bug priority may depend on functionality of the component, making the "component" feature important for determining priority. For example, if the component's functionality is critical to the system's working, bug reports submitted for the component may have higher priority as compared to bugs reported for a less important component. Similarly, high severity level of a bug report may represent that this bug should be resolved as early as possible. This is also sometimes true for high priority bugs. Platform and operating system may also provide useful information to categorize the bug reports, e.g., bugs occurring on Linux operating system may be given higher priority than a bug of Windows. Bug life time is the time required to resolve a bug. It is an important factor to determine the bug priority. If a bug is resolved quickly by the project developer, its priority may be high. If a bug is not resolved for a long time, it may be of low importance.

We performed bug priority classification experiments with two different combinations of categorical attributes. These combinations are described below:

Basic Features (*BF*): Categorical attributes available at report submission time: severity, component, operating system and platform. From the above six selected attributes, only four are available at report submission time (when a reporter reports the bug). To build a recommender to facilitate priority assignment at submission time, we take only these features for training.

Basic and Predicted Features (*BPF*): Categorical attributes of BF (severity, component, operating system and platform), bug lifetime and developer.

"Bug lifetime" and "developer" attributes are not available when the reporter reports the bug but these values can be predicted using the information available at report submission time. The effort (bug lifetime) to fix a new bug is predicted in [9, 44]. Anvik[8] and Canfora *et al.*[1] built a developer recommender which

automatically assigns a developer to a new bug report. We combine these attributes with the attributes in BF to check whether this set of training features improves the classifier performance.

5.4.2 Text Features (Summary and Long Description)

Summary or title of a bug report contains a short description of the bug mentioned by the bug reporter in one line. This short description tells about the type of the problem (or problem category). In bug finding tools, bugs are prioritized on the basis of the bug category (e.g., bug with "overflow" category has precedence over "empty static initializer" category). The problem category determines how important a bug is.

In the long description field, the problem faced by the user is described in detail. One line summary and full text description fields of a bug report uniquely characterize the report because a bug is uniquely described in these fields.

The text features of a bug report are formed by combining summary and long description of a bug, and have been used by different researchers[1,3,17,27] to perform their experiments. The reason for combining the two is that the summary attribute conveys important details about a bug which adds meaningful information to the long description. Since it is short, including it with the long description in the text features adds valuable information without significantly increasing the number of features (and hence the computational cost). Thus including summary adds value and does not increase cost whereas removing it does not reduce cost much, but may result in important details being omitted.

For these reasons, and similar to the approach used by other researchers, when using text features we have combined summary and long description (TF) and have not used description alone. On the other hand, we have used summary alone (as in [11]), and also in combination with categorical features.

5.5 Evaluation Criteria

We evaluated the classifiers' performance by using the precision and recall measures[45]. We also propose two new evaluation measures, NFN and NFP, for evaluating the recommendations made by the priority recommender. These measures are described in this subsection in detail.

5.5.1 Precision and Recall

Precision and recall measure the accuracy of a classifier. Precision of a class C is the number of instances correctly classified as class C divided by the total number of instances classified as class C. Precision measures the percentage of correct predictions related to the predictions made by the classifier. Recall of a class C is the number of instances correctly classified as class C divided by the total number of instances in the dataset having class label C. Recall measures the percentage of correct predictions related to actual classes.

For a bug priority classifier we need high precision and recall especially for higher priorities. As an example, consider the P1 priority class. Low recall of P1 means a high false negative rate, i.e., most of the bugs with priority P1 are given a low priority so their resolution will be delayed. If precision of P1 is low it means a high false positive rate, i.e., most of the bugs that are not actually P1 are given P1 priority so they will be handled first by the triagers and developers, which results in delay in the resolution of important bugs. In any case we need to avoid delay in the resolution of important bugs and want them to be resolved as early as possible.

5.5.2 Proposed Evaluation Measures

Our proposed measures, Nearest False Negatives (NFN) and Nearest False Positives (NFP), evaluate the misclassified (false negatives and false positives) bug reports, whether they are assigned a priority level close to the correct priority level or not. Our intuition for defining these evaluation measures is that if a bug with a certain priority level is misclassified to one level lower or higher priority (*nearest* priority level) it is less dangerous as compared to when there is a difference of more than one priority level. For example, if a bug with P1 priority is given P4 or P5 priority by the classifier it is more dangerous than if it is assigned P2 priority level because a bug with priority P2 will be resolved just after the bugs with priority P1.

Furthermore, if a bug with higher priority (e.g., P1) is misclassified to its nearest lower priority level (P2), it is more dangerous as compared to if a bug with lower priority (e.g., P2) is misclassified to its nearest higher priority level (P1). This is because if a bug with priority P1 is misclassified as P2, it means that the resolution of bug with high priority will be delayed. If a bug with priority P2 is misclassified as P1, it may not be as risky[②].

NFN and NFP measure how good a classifier is in

[②]It is relevant to note that although if a bug with priority P2 is misclassified as P1 priority, it may be less dangerous than a bug with priority P1 being assigned P2, but it may lead to lower priority bugs being handled earlier than important bugs. Also, if many bugs are given higher priorities than actual, there will be a large number of bugs in higher priority classes making it difficult for the triager to handle all of them. Both of these problems result in delay in resolution of the actual high priority bugs.

assigning the nearest priority levels. Greater value of NFN indicates that most of the misclassified bug reports are assigned the nearest priority levels with respect to their actual priority levels, and greater value of NFP indicates that most of the mis-predictions of the classifier for a priority class are for its nearest priority levels. To compute NFN and NFP, we assign weights to each false negative or false positive class (priority level) with respect to a bug priority class as shown in Table 4.

To differentiate between misclassification to higher or lower priority levels, when a bug of low priority is misclassified to its nearest higher priority class (e.g., P2 classified as P1), we have given a higher weight ($W_{(h)}$) and when a bug of high priority is misclassified to its nearest lower priority class (e.g., P1 classified as P2), it is given a lower weight ($W_{(l)}$)[③].

Table 4. Weights for Each False Negative or False Positive Class w.r.t. Bug Priority Class

Priority Class	False Negatives or False Positives				
	P1	P2	P3	P4	P5
P1		$W_{1(l)}$	$W_{2(l)}$	$W_{3(l)}$	$W_{4(l)}$
P2	$W_{1(h)}$		$W_{1(l)}$	$W_{2(l)}$	$W_{3(l)}$
P3	$W_{2(h)}$	$W_{1(h)}$		$W_{1(l)}$	$W_{2(l)}$
P4	$W_{3(h)}$	$W_{2(h)}$	$W_{1(h)}$		$W_{1(l)}$
P5	$W_{4(h)}$	$W_{3(h)}$	$W_{2(h)}$	$W_{1(h)}$	

Note: Values of Weights: $W_{1(h)} = 1$, $W_{1(l)} = 0.84$, $W_{2(h)} = 0.70$, $W_{2(l)} = 0.56$, $W_{3(h)} = 0.42$, $W_{3(l)} = 0.28$, $W_{4(h)} = 0.14$, $W_{4(l)} = 0$.

To calculate NFN/NFP, the percentage of false negatives (or false positives) of each class are multiplied with the corresponding weights and then added. For example, for priority class P1, NFN/NFP may be calculated as:

$$\text{NFN/NFP of priority class P1} = p2 \times W_{1(l)} + p3 \times W_{2(l)} + p4 \times W_{3(l)} + p5 \times W_{4(l)},$$

where $p2$, $p3$, $p4$ and $p5$ represent the percentage of false negatives (or false positives in case of NFP) of priority class P1 classified as P2, P3, P4 and P5 classes respectively. $W_{i(l)}$ represent the weights of priority classes which have lower priority than the actual class, and i denotes how many levels away a priority class is from the actual class. For priority class P1, we have $W_{i(l)}$ only, since a bug with priority P1 will always be misclassified to a class with lower priority.

Similarly, the NFN/NFP of various classes may be calculated as follows:

$$\begin{aligned}&\text{NFN/NFP of priority class P2}\\ &= p1 \times W_{1(h)} + p3 \times W_{1(l)} + p4 \times W_{2(l)} + p5 \times W_{3(l)},\\ &\text{NFN/NFP of priority class P3}\\ &= p1 \times W_{2(h)} + p2 \times W_{1(h)} + p4 \times W_{1(l)} + p5 \times W_{2(l)},\\ &\text{NFN/NFP of priority class P4}\\ &= p1 \times W_{3(h)} + p2 \times W_{2(h)} + p3 \times W_{1(h)} + p5 \times W_{1(l)},\\ &\text{NFN/NFP of priority class P5}\\ &= p1 \times W_{4(h)} + p2 \times W_{3(h)} + p3 \times W_{2(h)} + p4 \times W_{1(h)}.\end{aligned}$$

$W_{i(h)}$ represent the weights of priority classes which have higher priority than the actual class. For priority class P5, we have $W_{i(h)}$ only, since a bug with priority P5 will always be misclassified to a class with higher priority. For priority classes P2∼P4, the formulas for NFN/NFP include both $W_{(h)}$ and $W_{(l)}$.

As an example, consider a test set in which the number of bug reports with priority P1 is 100. If all the bug reports are assigned P1 priority, then recall is 100% and NFN is zero. If recall of priority class P1 is low, e.g., 10%, then NFN of priority class P1 tells whether most of the bug reports are assigned nearest classes or not. If 90 false negatives distributed in P2, P3, P4 and P5 classes are 80, 8, 2 and 0 respectively, then NFN is calculated as follows:

$$\begin{aligned}&\text{NFN of priority class P1}\\ &= (80/90) \times 100 \times 0.84 + (8/90) \times 100 \times 0.56 + (2/90) \times 100 \times 0.28 + (0/90) \times 100 \times 0 = 78.5\%.\end{aligned}$$

Recall of P1 is 10% but 78.5% NFN tells us that most of the false negatives are predicted for nearest classes, which may not be very dangerous.

Similarly, to calculate the NFP of priority class P1, we calculate the total false positives of P1 and then the number of false positives classified as P2, P3, P4 and P5 classes. If a dataset contains 5 bug reports with priority P1 but the classifier assigns P1 priority to 100 bug reports, precision is 5% and false positives of P1 are 95%. If the false positives distributed in P2, P3, P4 and P5 classes are 90, 5, 0 and 0 respectively, NFP is calculated as follows:

$$\begin{aligned}&\text{NFP of priority class P1}\\ &= (90/95) \times 100 \times 0.84 + (5/95) \times 100 \times 0.56 + (0/95) \times 100 \times 0.28 + (0/95) \times 100 \times 0 = 82\%.\end{aligned}$$

[③] In case nearest higher and nearest lower levels are not to be differentiated, both one level lower and one level higher priority for a certain priority level may be considered as nearest priority levels, and the weights may be kept symmetric, e.g., the weight of P1 being misclassified to P2 is set to be equal to that of P2 being misclassified to P1, i.e., $W_{1(h)} = W_{1(l)}$.

Precision is 5% but NFP Valne = 82% tells that most of the false positives are predicted for nearest classes.

According to the formula, NFN (or NFP) of a class is 100% when all the false negatives (or false positives) lie in the nearest higher class (in case of priority class P2, these should lie in P1 class) and NFN (or NFP) is 84% if all the false negatives (or false positives) lie in the nearest lower priority class. Since there is no nearest higher priority class for P1, so the highest value of NFN (NFP) for P1 will be 84%. $W_{2(h)}$ and $W_{2(l)}$ represent the weights for the 2nd nearest priority classes. If all the false negatives (false positives) lie in the 2nd nearest priority classes then NFN (NFP) will be 70% (for the 2nd nearest higher class) or 56% (for the 2nd nearest lower class). Thus when false negatives (or false positives) are distributed in different classes, more than 56% NFN (NFP) may be considered better as it indicates that most misclassifications are up to two priority levels away.

6 Experimental Results

In this section, we report results of our experiments using the SVM and Naïve Bayes classifiers. We also evaluate different feature categories and training dataset sizes. For evaluation, we use precision and recall measures, and to gain further insight into the results we use our proposed NFN and NFP measures.

6.1 Bug Priority Classification Results for SVM

Fig.1 presents the results of SVM based bug priority classification using recall, precision, NFN and NFP measures.

It is relevant to note that we are more interested in high accuracy for higher priority classes, e.g., P1, since they indicate the more important bugs. It can be seen from Fig.1 that recall and NFN of the P1 class is better for TF and CSTF categories. Precision of the P1 class is less than 40% for all categories but NFP of the class is more than 70% for CF, CSF and CSTF, which indicates that most incorrect predictions are to the nearest lower class, i.e., P2. For SF and TF, NFP of the P1 class is 60% or more, which indicates that most incorrect predictions are to two nearest lower classes, i.e., P2 or P3. It can also be seen from Fig.1 that precision, recall, NFN and NFP of the P3 class are higher than those of other classes for all feature categories. (For CF, CSF and CSTF categories, NFP of the P3 class is not shown in the figure because precision of P3 for these categories is 100%.)

Fig.2 presents the recall, NFN, precision and NFP for all categories, averaged over all priority classes. It can be seen from Fig.2(a) that for the CSTF category, average recall and NFN are higher than all other categories which means that most of the bugs are assigned correct priority levels and most of the misclassified bugs are classified to the nearest priority levels. For CF, CSF and TF categories, recall is more than 45% and NFN is almost 70% which indicates that although bugs of a priority class are not classified into the correct priority levels but most are predicted for the nearest priority level. For SF, both average recall and NFN are low as compared to other categories.

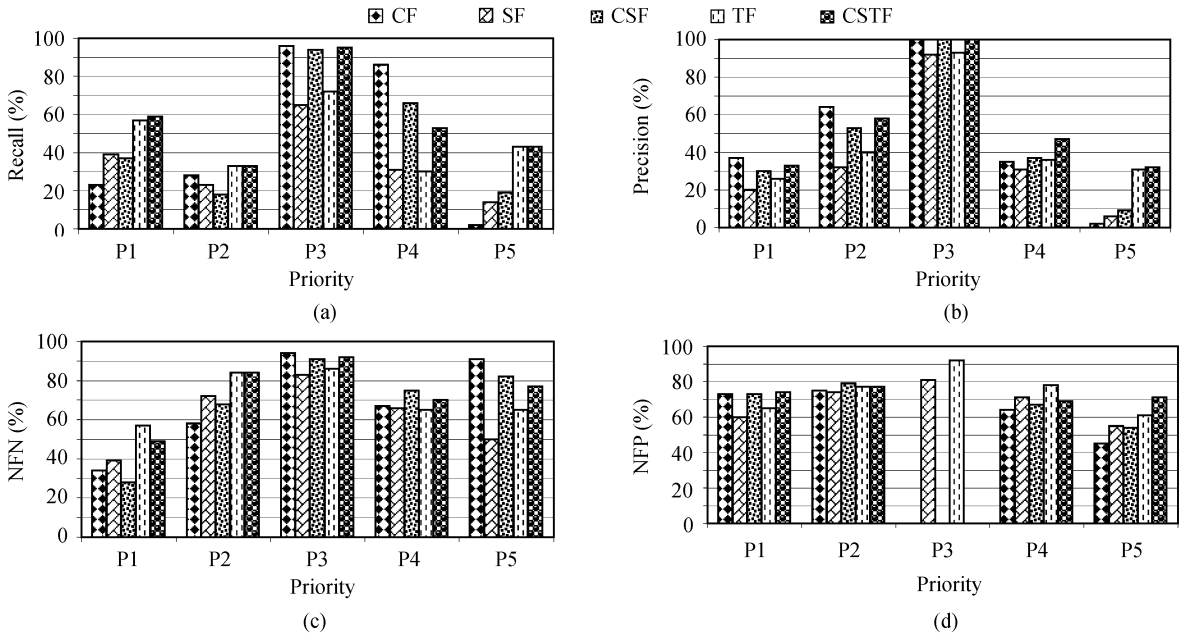

Fig.1. Recall, precision, NFN and NFP of each priority class for SVM.

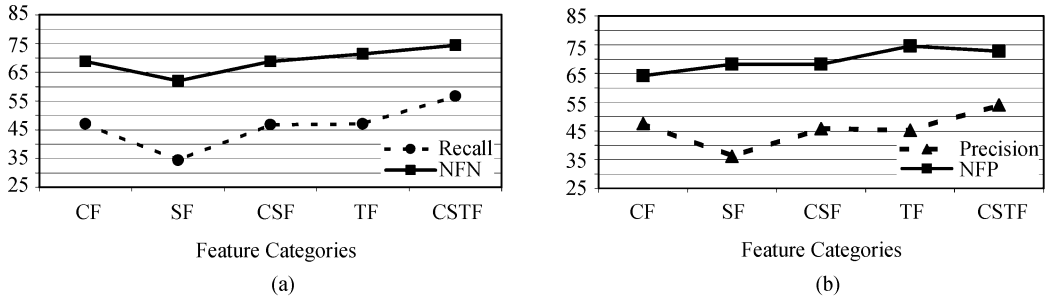

Fig.2. Average recall and NFN, precision and NFP for SVM.

From Fig.2(b) it can be seen that for CF, CSF, TF and CSTF categories, precision is more than 45% which means that 45% of the predictions made by the classifier for a priority class are correct.

Moreover, NFP of these categories is around 65% or more which indicates that most of the incorrectly predicted bug reports are placed into classes which are up to 2 priority levels away④. Average precision of CF, CSF and TF is almost the same but NFP of TF is better than that of the other categories.

From Fig.2, it can be noted that in the CF and TF categories, recall and precision are 45∼47% but when categorical and text features are combined (CSTF), then recall and precision increase by almost 10%. This indicates that a combination of features improves the classifier performance. The best results achieved by bug priority classifier are with the CSTF category.

6.2 Results of Bug Priority Classification for Naïve Bayes

Fig.3 presents the recall, precision, NFN and NFP of each bug priority class for different feature combinations for the Naïve Bayes classifier. It can be seen that recall and NFN of the P1 class are better for the TF and CSTF categories whereas precision and NFP of the P1 class are better for CSF. Moreover, precision, recall, NFN and NFP of the P3 class are better than other classes for almost all the feature categories.

Fig.4 presents the recall and NFN, precision and NFP, for different feature categories averaged over

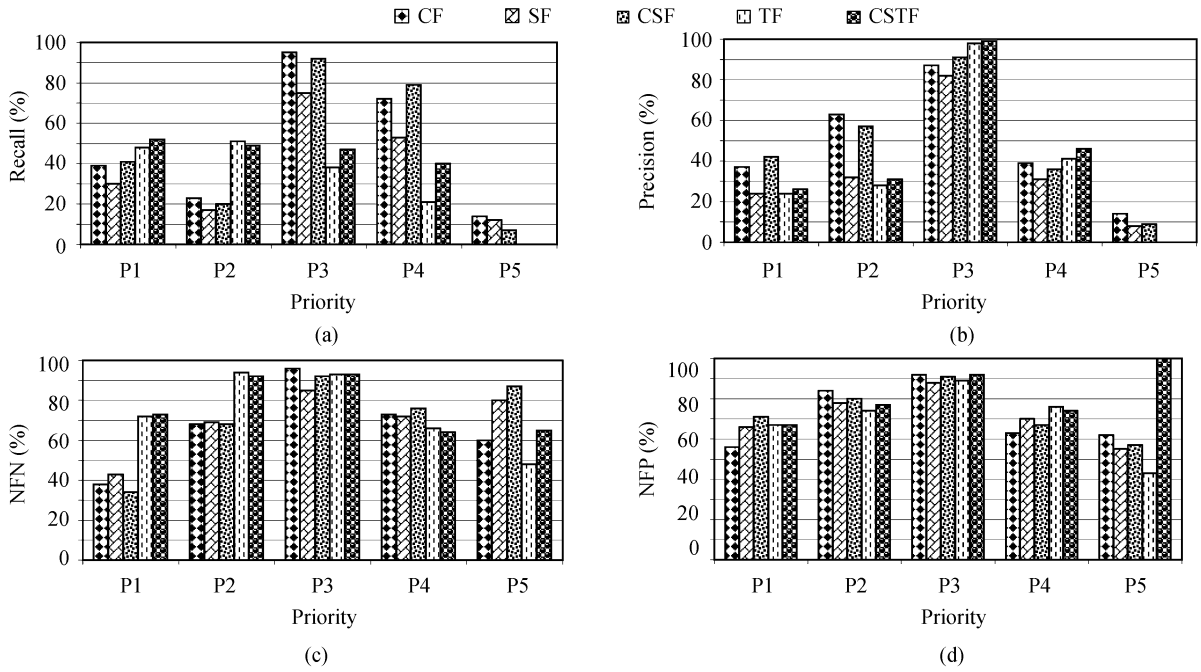

Fig.3. Recall, precision, NFN and NFP of each priority class for Naïve Bayes.

④An NFN value > 70% indicates that more than 50% misclassified bug reports are placed into nearest priority classes, whereas an NFN value > 42% indicates that more than 50% misclassified bug reports are placed into classes up to 2 levels away. 65% is closer to 70% than to 42%, thus although we cannot say that most misclassifications are to the nearest priority classes, we expect about half of them to be so.

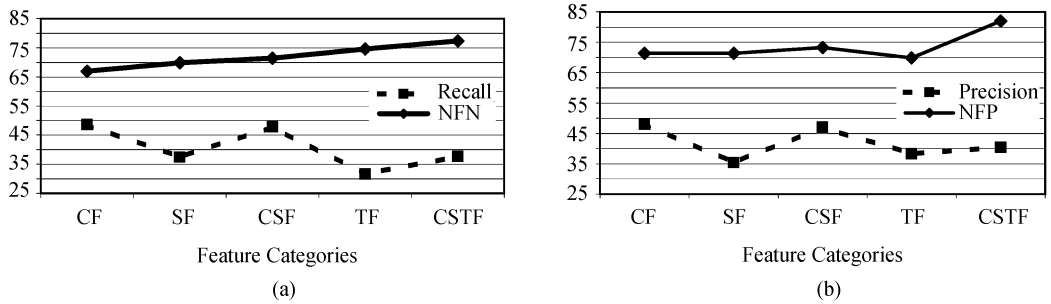

Fig.4. Average recall and NFN, precision and NFP for Naïve Bayes.

different priority classes for Naïve Bayes. It can be seen from Fig.4(a) that average recall of CF (49%) and CSF (48%) are better than other categories. However, NFN of CF is lower than that of the other categories. Average recall of CSTF (38%) is lower than that of the other categories but NFN is more than 75% which is better than the other categories. Although the average recall of CSF is slightly lower than that of CF, and average NFN of CSF is lower than that of CSTF, results of CSF are overall better than the other categories in terms of recall and NFN.

From Fig.4(b) it can be seen that average precision of the CF and CSF categories is more than 45%, which is better than average precision of the other categories. Moreover, NFP of these two categories is greater than 70%. This means that 45% of the predictions are for the correct priority levels, and most of the incorrect predictions are to the nearest priority levels.

Thus overall, the performance of Naïve Bayes is better for the CSF category compared to other categories.

6.3 Comparison of Naïve Bayes and SVM

Fig.5 presents the average recall and NFN of different feature categories for SVM and Naïve Bayes. It can be seen that for the CF, SF and CSF categories, recall of Naïve Bayes is slightly better than that of SVM. For TF and CSTF, recall of SVM is much better than that of Naïve Bayes. NFN of Naïve Bayes remains better than that of SVM for all categories.

Fig.6 presents the average precision and NFP of different feature categories for SVM and Naïve Bayes. It can be seen that for the CF, SF and CSF categories, precision of SVM and Naïve Bayes is almost the same but NFP of Naïve Bayes is better than that of SVM. For the TF category, both precision and NFP of SVM are better than that of Naïve Bayes. For the CSTF category, precision of SVM (54%) is much better than Naïve Bayes (40%) but NFP of Naïve Bayes is better.

Overall it can be seen that the performance of Naïve Bayes is better for categorical and summary features

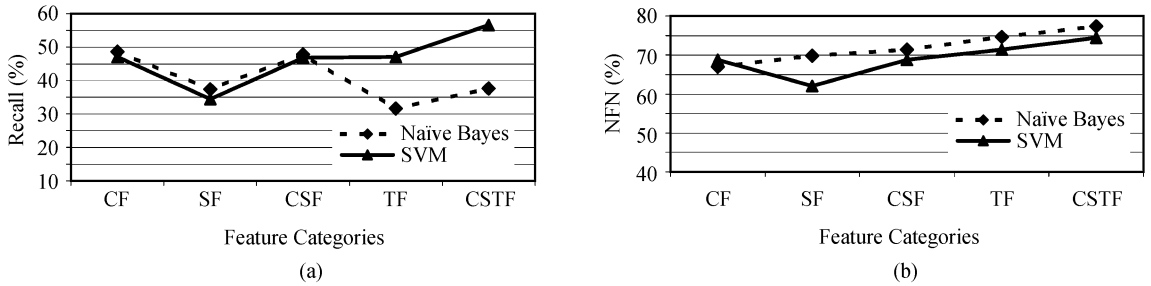

Fig.5. Average recall and NFN for SVM and Naïve Bayes.

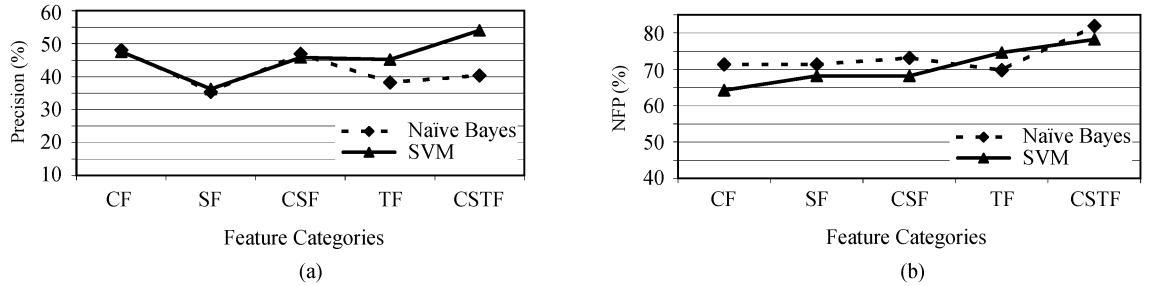

Fig.6. Average precision and NFP for SVM and Naïve Bayes.

but not as good as SVM for text features (when long description is included), which shows that as compared to SVM, Naïve Bayes is not good in handling the high dimensionality of text features.

6.4 Combinations of Categorical Fields

As described in Subsection 5.4, we performed experiments with two combinations of categorical fields. We analyze the results to determine which categorical feature combination is better in determining the priority of a bug.

From Fig.7, it can be seen that the average precision, recall and NFP of BPF are better than those of BF combination whereas the average NFN of both combinations is almost the same. Thus overall performance of BPF is better than that of BF which shows that when the predicted features ("developer" and "bug lifetime") are combined with the basic features, not only does the accuracy of results (in terms of precision and recall) improve, but also most of the false positives and false negatives are predicted for the two nearest classes (as NFN and NFP are more than 60%). These results indicate that developer and bug lifetime features contain useful information for determining the bug priority.

6.5 Training Data Size

Before using a classifier for predictions, it should be sufficiently trained so that it learns the underlying properties of the bug reports[11]. We used an equal number of instances, i.e., 327 in each class, to train the classifier. It is possible that the classifier achieves good performance with a smaller dataset size, and there is no significant improvement in its performance beyond a certain number of training examples. In this case we can use less number of examples, and thus reduce time and resources required to train the classifier. Moreover, it is possible that training data requirements vary for different feature categories.

To check whether fewer training examples can be used, and how training data requirements vary for different features, we conducted experiments using different dataset sizes. We started training by taking 50 samples for each priority class (total training dataset size is 250) and increased the dataset size by adding 50 samples in the preceding dataset. For each dataset size, we took different sub-samples from the training data, e.g., for size 250, we divided the data of size 1 500 into 6 sub-samples of size 250 and trained the classifier on sub-samples. The average of results of the sub-samples was taken for each dataset size. Fig.8 presents these results.

It can be seen from Fig.8 that average recall and precision increase with an increase in the number of training samples for all feature categories when sample size increases from 50 to 100. Beyond 100 samples, the changes in recall and precision for the categories are different.

For categorical and summary features (CF, SF and CSF), there is no significant increase in recall and precision beyond 100 samples. For example, CSF's recall for 100 samples is 46%, and for 327 samples it increases to 47%. Similarly, CSF's precision for 100 samples is 44%, and for 327 samples it increases to 46%. On the other hand, for TF and CSTF categories, accuracy of the classifier improves with the increase in training data size and for sample size 327, it is clearly better than for other sample sizes. For example, TF's recall for 100 samples is 39% which increases to 47% for 327 samples, and precision is 37% for 100 samples, which increases to 45% for 327 samples. Thus for categorical and summary features, due to the relatively less number of possible feature values, there is no significant increase in classifier performance when the training dataset size increases beyond 100 samples. This is an indication that a relatively smaller dataset size may be enough for training if only these features are used. Since the TF and CSTF categories contain text features and training the classifier for text features is difficult due to the high dimensionality of data, so a larger training data size is needed for better training.

It can also be seen from Fig.8 that there is no apparent trend in change in average NFN or NFP with the increase in sample size.

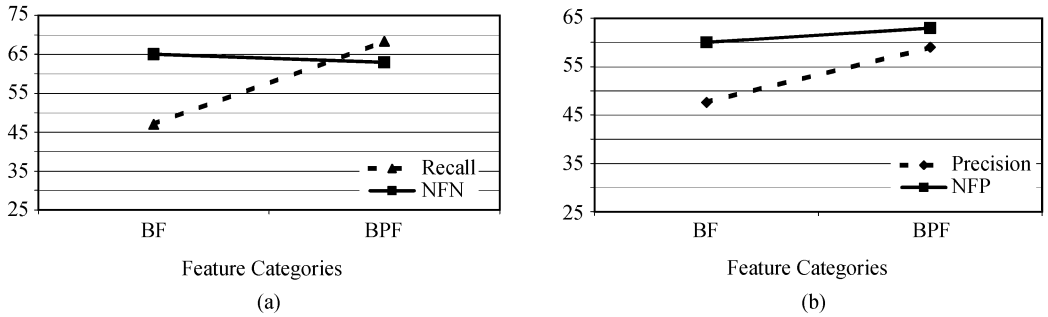

Fig.7. Average recall and NFN, precision and NFP for categorical feature combination.

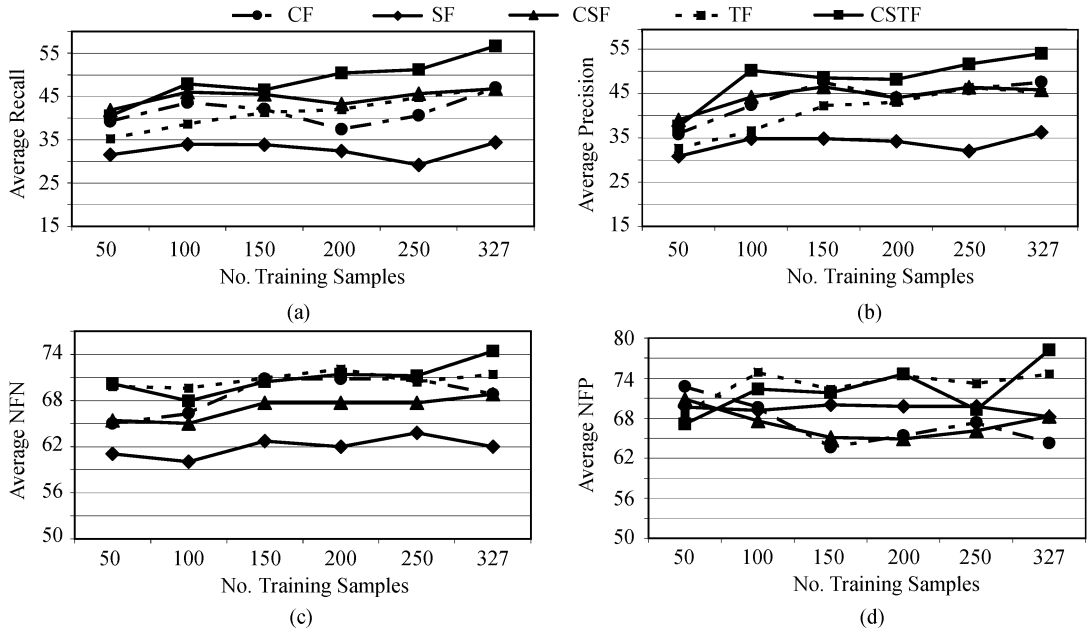

Fig.8. Average recall, precision, NFN and NFP for varying dataset size.

6.6 Threats to Validity

In this subsection we discuss the factors that may affect the validity of our approach and the steps we took to reduce their impact.

Construct Validity. The results of our classification approach depend on the selection of appropriate features and the correct values of these features. If the contents of a new bug report are inaccurate, this obviously affects the correct prediction of priority. For Eclipse, the user base consists mostly of developers, so it is expected that in most cases a bug report is filled carefully leading to fewer chances of error.

Internal Validity. Our approach to building a priority recommender is based on the assumption that there is a relationship between bug report features and the bug priority. Previous research suggests that such causal relationships do exist, e.g., between bug report features and severity[23]. Instead of restricting ourselves to a small set of features (e.g., categorical only, summary only), we have tried to select a more meaningful set by experimenting with categorical as well as text features. Using a larger feature set reduces the threat that priority may not depend upon the selected features.

External Validity. External threat to validity for our study is related to the dataset used in our experiments. To evaluate our proposed classification approach for bug priority assignment, we focused on the bug data of the Eclipse project. Eclipse is an open source community with more than 200 projects. Eclipse data has been used by different researchers for their experiments[3,8,27-28]. It was for this reason that we selected Eclipse for evaluation. To generalize our results and support our conclusions, experiments may be performed on the bug reports of other software systems. Since Eclipse uses Bugzilla as its bug tracking system, which is the most widely used bug tracking system for open source projects, our approach can be applied to other software systems without difficulty. For other bug repositories such as GNATS[46] and JIRA[47] which have a different bug report structure, the approach may be adapted by utilizing the available bug report attributes as features.

7 Conclusions

In this paper, we developed a bug priority classifier to automate the process of assigning bug priority to new bug reports in a bug repository using SVM and Naïve Bayes classification algorithms and compared their results. Accuracy of the classifiers indicates that our proposed priority recommender can help triagers in assigning an appropriate priority level to bug reports so that they are resolved at appropriate time. We also proposed new measures NFN and NFP for evaluation, which indicate whether the incorrectly predicted bug priorities are predicted for nearest priority classes or not.

Experiments for bug priority classification were performed using different feature categories of bug reports. Experimental results using precision, recall, NFN and NFP reveal that overall performance of SVM is better than that of Naïve Bayes. For SVM, the results of bug priority classifier are better when the categorical

and text features are combined for training the classifier (CSTF). Results of the Naïve Bayes classifier are better when the long description of a bug is not included as a feature. This shows that as compared to SVM, Naïve Bayes is not good at handling the high dimensionality of text features. We also trained the bug priority classifier using different training data size ranging from 50 to 327 bug reports per class. Results show that for categorical features, there is no significant improvement in classifier accuracy when the training data size is increased from 100 to 327 bug reports, although accuracy improves for text features.

Our experiments reveal that NFN and NFP are useful measures for evaluating the recommendations made by the classifier. They can be used to gain understanding into the results produced by precision and recall. For example, low precision and recall indicate that the assigned priority is incorrect. If in this case NFN and NFP are high, it indicates that the misclassified bugs are assigned the nearest priority levels, which is less cause for concern as compared to when NFN and NFP are low. Thus these two measures provide meaningful insight into the results produced by precision and recall.

References

[1] Canfora G, Cerulo L. Supporting change request assignment in open source development. In *Proc. ACM Symposium on Applied Computing*, Dijon, France, April 2006, pp.1767-1772.

[2] Anvik J. Assisting bug report triage through recommendation [PhD Thesis]. University of British Columbia, 2007.

[3] Cubranic D, Murphy C. Automatic bug triage using text categorization. In *Proc. Software Engineering and Knowledge Engineering,* Banff, Canada, June, 2004, pp.92-97.

[4] Mozilla. http://www.mozilla.org, 2010.

[5] Anvik J, Murphy G C. Determining implementation expertise from bug reports. In *Proc. the 4th MSR*, Minneapolis, USA, May 2007, Article No.2.

[6] Tucek J, Lu S, Huang C, Xanthos S, Zhou Y. Triage: Diagnosing production run failures at the user's site. *ACM SIGOPS Operating Systems Review*, 2007, 41(6): 131-144.

[7] Herraiz I, German D M, Gonzalez-Barahona J M, Robles G. Towards a simplification of the bug report form in eclipse. In *Proc. International Working Conference on Mining Software Repositories*, Leipzig, Germany, May 2008, pp.145-148.

[8] Anvik J. Automating bug report assignment. In *Proc. the 28th International Conference on Software Engineering*, Shanghai, China, May 2006, pp.937-940.

[9] Weib C, Premraj R, Zimmermann T, Zeller A. Predicting effort to fix software bugs. In *Proc. Workshop on Software Reengineering*, Bad Honnef, Germany, May 2007.

[10] Kim S, Whitehead J. How long did it take to fix bugs? In *Proc. International Workshop on Mining Software Repositories*, Shanghai, China, May 2006, pp.173-174.

[11] Lamkanfi A, Demeyer S, Gigery E, Goethals B. Predicting the severity of a reported bug. In *Proc. the 7th Working Conference on Mining Software Repositories,* Cape Town, South Africa, May 2010, pp.1-10.

[12] Yu L, Tsai W, Zhao W, Wu F. Predicting defect priority based on neural networks. In *Proc. the 6th Int. Conf. Advanced Data Mining and Applications*, Wuhan, China, November 2010, pp.356-367.

[13] Kanwal J, Maqbool O. Managing open bug repositories through bug report prioritization using SVMs. In *Proc. International Conference on Open-Source Systems and Technologies*, Lahore, Pakistan, December 2010.

[14] Kim S, Ernst M D. Prioritizing warning categories by analyzing software history. In *Proc. the 4th International Workshop on Mining Software Repositories*, Minneapolis, USA, May 2007, Article No. 27.

[15] Kim S, Ernst M D. Which warnings should I fix first? In *Proc. the 6th ESEC-FSE*, Dubrovnik, Croatia, September 2007, pp.45-54.

[16] Kremenek T, Engler D. Z-Ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Proc. the 10th International Conference on Static Analysis,* June 2003, pp.295-315.

[17] Anvik J, Hiew L, Murphy G C. Who should fix this bug? In *Proc. the 28th International Conference on Software Engineering*, Shanghai, China, May 2006, pp.361-370.

[18] Anvik J, Murphy G C. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Transactions on Software Engineering and Methodology*, 2011, 20(3): Article No.10.

[19] Aljarah I, Banitaan S, Abufardeh S, Jin W, Salem S. Selecting discriminating terms for bug assignment: A formal analysis. In *Proc. the 7th International Conference on Predictive Models in Software Engineering*, Banff, Canada, September 2011, Article No.12.

[20] Ahsan S N, Ferzund J, Wotawa F. Automatic software bug triage system (BTS) based on Latent Semantic Indexing and Support Vector Machine. In *Proc. the 4th International Conference on Software Engineering Advances*, Washington, USA, September 2009, pp.216-221.

[21] Jeong G, Kim S, Zimmermann T. Improving bug triage with bug tossing graphs. In *Proc. the 7th ESEC-FSE*, Amsterdam, Netherlands, August 2009, pp.111-120.

[22] Tamrawi A, Nguyen T, Al-Kofahi J, Nguyen T N. Fuzzy set-based automatic bug triaging. *In Proc. the 33rd International Conference on Software Engineering* (*NIER Track*), Miami, USA, May 2011, pp.884-887.

[23] Lamkanfi A, Demeyer S, Soetens Q D, Verdonck T. Comparing mining algorithms for predicting the severity of a reported bug. In *Proc. the 15th European Conference on Software Maintenance and Reengineering*, Oldenburg, Germany, March 2011, pp.249-258.

[24] Ling C, Huang J, Zhang H. Auc: A better measure than accuracy in comparing learning algorithms. In *Lecture Notes in Computer Science 2671*, Xiang Y, Chaib-Draa B (eds.), Springer-Verlag, 2003, pp.329-341.

[25] Gegick M, Rotella P, Xie T. Identifying security bug reports via text mining: An industrial case study. *In Proc. the 7th Working Conference on Mining Software Repositories*, Cape Town, South Africa, May 2010, pp.11-20.

[26] Zaman S, Adams B, Hassan A E. Security versus performance bugs: A case study on Firefox. In *Proc. the 8th Working Conference on Mining Software Repositories*, Hawaii, USA, May 2011, pp.93-102.

[27] Runeson P, Elexandersson M, Nyholm O. Detection of duplicate defect reports using natural language processing. In *Proc. the 29th International Conference on Software Engineering*, Minneapolis, USA, May 2007, pp.499-510.

[28] Wang X, Zhang L, Xie T, Anvik J, Sun J. An approach to detecting duplicate bug reports using natural language and execution information. In *Proc. the 30th International Conference on Software Engineering*, Leipzig, Germany, May 2008, pp.461-470.

[29] Canfora G, Cerulo L. Impact analysis by mining software and change request repositories. In *Proc. the 11th International Software Metrics Symposium*, Como, Italy, September 2005, Article No.29.

[30] Prifti T, Banerjee S, Cukic B. Detecting bug duplicate reports through local references. In *Proc. the 7th International Conference on Predictive Models in Software Engineering*, Banff, Canada, September 2011, Article No.8.

[31] Wu L, Xie B, Kaiser G, Passonneau R. BugMiner: Software reliability analysis via data mining of bug reports. In *Proc. the 25th International Conference on Software Engineering and Knowledge Engineering*, Miami, USA, July 2011, pp.95-100.

[32] Marks L, Zou Y, Hassan A E. Studying the fix-time for bugs in large open source projects. In *Proc. the 7th International Conference on Predictive Models in Software Engineering*, Banff, Canada, September 2011, Article No.11.

[33] Gyimothy T, Ferenc R, Siket I. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 2005, 31(10): 897-910.

[34] Zimmermann T, Premraj R, Zeller A. Predicting defects for Eclipse. In *Proc. International Workshop on Predictor Models in Software Engineering*, Minneapolis, USA, May 2007, Article No.9.

[35] Bugzilla. http://www.bugzilla.org, 2010.

[36] Han J, Kamber M. Data Mining: Concepts and Techniques. 2nd edition, Morgan Kaufmann, 2006.

[37] Joachims, T. Text categorization with support vector machines: Learning with many relevant features. In *Proc. European Conference on Machine Learning*, Chemnitz, Germany, April 1998, pp.137-142.

[38] Kantardzic M. Data Mining: Concepts, Models, Methods, and Algorithms. New York, USA: Wiley-Interscience, 2003.

[39] Witten H I, Frank E. Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations. New York, USA: Morgan Kaufmann, 2000.

[40] Noble W S. What is a support vector machine? *Nature Biotechnology*, 2006, 24: 1565-1567.

[41] Vapnik V N. Statistical Learning Theory. New York, USA: Wiley-Interscience, 1998.

[42] Sebastiani F. Machine learning in automated text categorization. *ACM Computing Surveys*, 2002, 34(1): 1-47.

[43] Eclipse. http://www.eclipse.org, 2010.

[44] Panjer L D. Predicting Eclipse bug lifetimes. In *Proc. the 4th International Workshop on Mining Software Repositories*, Minneapolis, USA, May 2007, pp.1-8.

[45] Baeza-Yates R, Ribeiro-Neto B. Modern Information Retrieval. Boston, USA: Addison-Wesley Longman, 1999.

[46] GNATS. http://www.gnu.org/software/gnats, 2010.

[47] JIRA. http://www.atlassian.com/software/jira, 2010.

Jaweria Kanwal received her Master's degree in computer science from Gomal University, Pakistan in 2004, and M.phil. degree from Quaid-i-Azam University, Islamabad, Pakistan in 2011. Her M.phil thesis involved the automatic assignment of priority to newly arrived bugs by mining the historical bug repositories data, as presented in this paper. Her research interests lie in data mining, software repositories and text categorization.

Onaiza Maqbool received her Ph.D. degree in computer science from the Lahore University of Management Sciences in 2006. She is an assistant professor at the Department of Computer Science, Quaid-i-Azam University, Islamabad, Pakistan. Prior to joining Quaid-i-Azam University, she worked in the software industry for some years. Her research interests lie in exploring machine learning techniques to solve software engineering problems. She has published more than twenty papers in well reputed journals and conferences.